EECS 151 FPGA Lab: Final Project Report

Rami Hijab, Neelesh Ramachandran

December 23, 2020

Contents

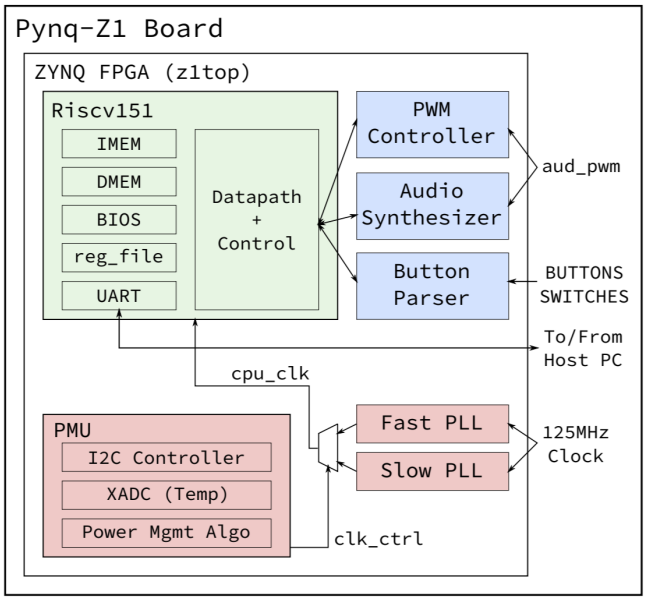


Figure 1: Overview of Project, Block-Level Diagram

1 Design Requirements

In this project, we designed a pipelined RISC-V CPU that can carry out a fully-featured set of RISC-V instructions and has broad memory-mapped I/O functionality. The memory-mapped peripherals part of the `Pynq-Z1` FPGA board and allowed us to implement a functional audio synthesizer that the user could interact with. A block-level layout of the project can be seen in fig. 1. Since performance of the processor is key, our objective was initially to meet a processor CPI $\left(\frac{\text{cycles}}{\text{instructions}}\right)$ ratio of 1.2, and we were ultimately tasked with minimizing the Iron Law, as in eq. (1). We were given an intensive reference workload to run (a defined, "untouchable" `mmult.c` matrix multiplication program), so only the latter two terms of this expression were within our control to optimize.

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} \tag{1}$$

1.1 Three-Stage Pipeline

One of the key initial design decisions we had to make was the placement of the pipelining registers, which would be the primary factor determining how difficult it would be to account for data and control hazards. This decision also impacts our critical path lengths, ability to debug the processor, and flexibility to restructure the datapath or incorporate additional modules. The structure we chose is outlined in fig. 2 and our motivation is described in section 2.1.

1.2 Memory Architecture

There are 4 memories involved in the project, outlined below:

1. BIOS Read-Only Memory (ROM): User programs are uploaded over a UART interface into the BIOS ROM, and this defines the program that the FPGA runs upon startup.
2. Instruction Memory (IMEM): Contains instructions for currently loaded program.
3. Data Memory (DMEM): Same memory instance as IMEM, with separated ports for clarity.

4. MMIO: See below, or section 2.2.3, for details.

1.3 Memory-Mapped I/O

The upload of programs from a laptop to the FPGA board happened over a memory-mapped UART interface, and we also mapped LEDs, buttons, switches, a PWM controller, and subtractive audio synthesizer to the board. These allowed the user to interact with the uploaded program directly through the hardware on the board, and the synthesizer itself could play notes of various pitches (set by a frequency-control-word, which was mapped to letters of the keyboard). The audio was generated by custom-weighting different fundamental waveforms (sine, square, triangle, and sawtooth). The synthesizer directly sent the audio data to the FPGA audio output, allowing the user to play tones directly from the board over UART.

2 High-level Organization

When designing a large system, such as our RISC-V CPU with integrated I/O, it is important to maintain functional organization, abstraction, and hierarchy. To do so, we began the design phase with a detailed block diagram. After several iterations on the diagram, we moved toward implementing our design in Verilog. This phase required high-levels of precision and abstraction so as to maintain an understandable code base that would be easy to both modify and debug.

2.1 Block Diagram

A functional block diagram of our three stage RISC-V pipeline is shown in fig. 2. Though the diagram labels four stages, the write-back (WB) stage occurs in parallel with the instruction fetch (IF) stage, making it a three-stage pipeline. Key elements of the design are the synchronous-read and -write memories that effectively act like pipeline registers. On the other hand, the register file is synchronous-write and asynchronous-read.

A fundamental component of our design approach was to analyze and iterate on the block diagram extremely thoroughly. With a three-stage pipeline, data and control hazards become a dominant concern, so even after getting checked off for the block diagram design, we spent another week to analyze the forwarding paths and control hazards. In doing so, we set up the following forwarding paths: write-back result forwarded to the input of the execute stage; write-back result forwarded to the output of the instruction decode stage; ALU result forwarded to the output of the instruction decode stage; and immediate and register outputs forwarded to the instruction fetch stage. The last of the forwarding paths alleviates control hazards for jump instructions. In addition to these forwarding paths the control logic must handle branch control hazards by injecting a `NOP` on a failed branch prediction.

It was also useful to analyze the critical path. Without forwarding, this is simply proportional to the stage that takes the longest to compute. However, with our extensive forwarding this path could potentially be longer. After analysis of the timing summary and the physical hardware design in Vivado, we concluded the following critical path: output of memory forwarded to the execute stage; ALU result forwarded to the output of the register file; the first register forwarded for use in JALR address forwarding in the instruction fetch stage; and finally that address is used to calculate the new program counter.

While extensive and time consuming, enumerating the possible forwarding paths and hazards, in addition to simply spending more time with the functional block diagram, led to a deeper understanding of the design and a much smoother implementation experience.

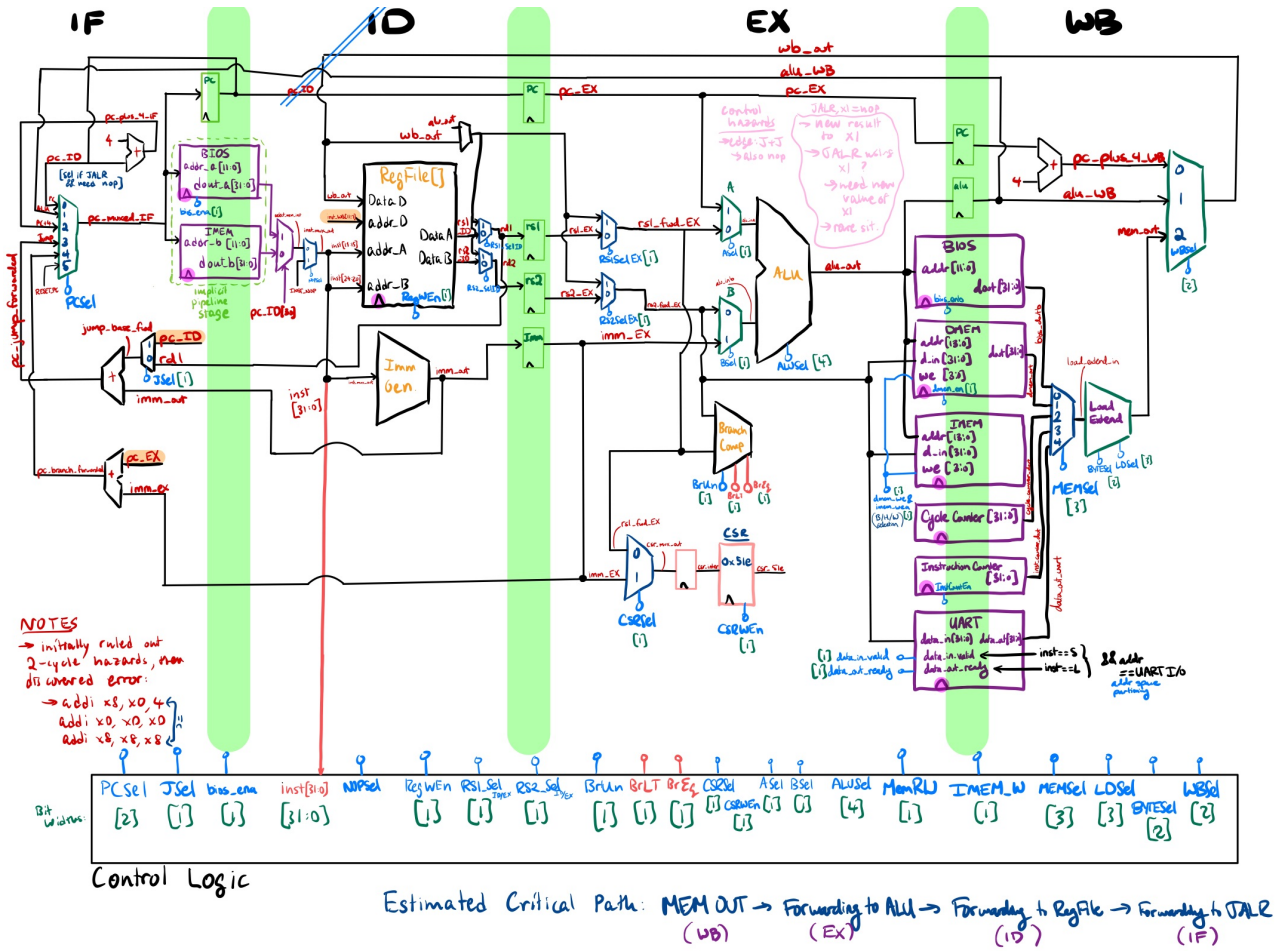


Figure 2: Three stage RISC-V Processor Design, including modules, forwarding logic, wiring, mutiplexers, control logic signals, and pipeline registers.

2.2 Module Hierarchy

In addition to maintaining a detailed functional block diagram, module abstraction and hierarchy was an essential element of our design. Utilizing modules allows for sub-block functional testing to verify correct performance in a piecemeal fashion, and also maintains abstraction barriers that make integration easier to implement. Furthermore, module-based design allows for hierarchical reuse of functions through multiple instances.

2.2.1 RISC-V Modules

In our RISC-V processor there are several key building blocks that were necessary to create and test individually. Within the instruction decode stage we implemented dedicated building blocks for the immediate generator, register file, and branch predictor. The arithmetic logic unit and branch comparator reside in the execute stage, performing the key operations relevant for evaluation. The load extend module in the write-back stage sign-extends outputs from memory to be written back to registers. For each building block we created a dedicated test bench to verify functionality prior to integration into the total processor. In addition to these specified functions we designed 2-to-1, 4-to-1, and 8-to-1 multiplexers to implement the logic utilized throughout the block diagram in fig. 2.

2.2.2 Control Logic

Another key design decision was the control logic organization. We created a dedicated control logic module so as to maintain organization of the intricate pipelined control signals and how they are driven while abstracting them away from the rest of the processor implementation. When implementing the control logic, we focused on three design principles: separating control signals based on the stage they belonged to; using combinational case statements to drive signals based on the instruction; and utilizing numerous local parameters for driving signals. Through these principles, we were able to maintain logical flow and separation throughout the control logic in a readable format. For details on the inputs and outputs of our control logic (inputs in red, outputs in blue, bit-widths in green), see the bottom block in fig. 2.

2.2.3 Memory-Mapped I/O

Similar to the control logic, we decided to isolate the memory-mapped input and output from the core processor module. The memory-mapped I/O modules contain the interface for all memory-mapped addresses, including, but not limited to, LEDs, buttons, PWM audio, and our subtractive synthesizer. As with the control logic, we heavily utilized local parameters to define address and control words. Additionally, we separated out the write and read addresses functionally within the module. Since there are many I/O functionalities, from GPIO to UART, dedicating an abstraction barrier for I/O proved fruitful for our design.

3 Key Sub-blocks and Unique Design Choices

As mentioned in section 2.2, designing sub-blocks were an essential part of decomposing our complex design. While many of the sub-blocks and design choices are more or less standardized, there were several unique approaches we took in our design.

3.1 FIFO

A first-in-first-out (FIFO) buffer was an essential building block for our design. In this buffer the first data written is also the first data read, maintaining the ordering of data across writes and reads. On either side (read or write) of the FIFO there are three fundamental signals: data, enable, and status. The data signal is the data to be written as an input or the data to be read as an output. In order to control the FIFO, there are write- and read-enable signals where FIFO data is only written when the write-enable is high and FIFO data output is only valid when the read-enable signal is high. Finally,

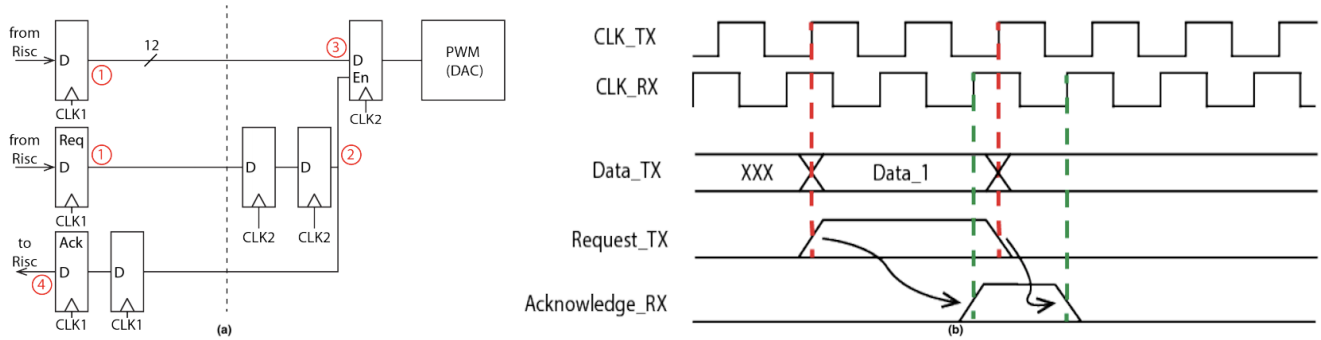


Figure 3: (a) Clock domain crossing functional diagram. (b) Clock domain crossing four-phase signal timing.

the status signal indicates whether the FIFO is full or empty, relevant to the write and read operations respectively.

Our FIFO design operates as a cyclic buffer, where two pointers define the valid space of memory within the FIFO. Every time data is written to the FIFO the write pointer location increases by one, and similarly on every read the read pointer location increases by one. With this methodology, when the read and write pointers have the same address, the FIFO is either full or empty. By adding an extra bit to the pointer address we can take advantage of parity and overflow to determine whether the FIFO is full or empty. When the full pointer addresses are equal, the FIFO is empty. However, when the pointer addresses are equal in every bit except the MSB, then one pointer has completely wrapped around the other, and the FIFO is therefore full. This design takes advantage of overflow in the addresses to determine the true relative location of the addresses.

We utilized the FIFO buffer described above in two instances of our design. First, we used the FIFO for general purpose I/O. When buttons are pressed on the hardware it is not guaranteed that the processor is ready to read that data input. Therefore, by using a FIFO for button presses, we can delay that input until the processor is ready to read the data, at which point it will read from the FIFO buffer. This technique allows stable and reliable use of transient I/O by our processor.

Secondly, we used a FIFO for our audio synthesizer. In order to produce synthesized digital sounds, we needed a sequence of look-up-tables (LUTs) as described in section 3.3. However the values from these LUTs are produced roughly at the rate of the clock frequency, and not at the desired sample frequency. To reduce the rate of data transfer, we utilized a FIFO. Data produced by the series of LUTs is written to the FIFO at the rate it is produced until the FIFO becomes full, at which point the LUTs are given a disable signal to stop producing samples. At the same time, samples are read from the FIFO at the well-controlled rate of the sampling frequency. In this way, the FIFO was able to systematically and reliably slow down the rate of data production for our synthesizer.

3.2 Clock Domain Crossing

In our design, we utilized two clock domains; one for the core processor and another for the audio synthesizer and pulse width modulation. This was essential for synthesizing sounds at an appropriate resolution; however, it poses control problems for data transfer between the clock domains. To alleviate this issue, we implemented a four-phase handshake to transfer data across the clock domains.

A functional diagram of the clock-domain-crossing (CDC) circuit and a signal timing is shown in fig. 3. When the processor in the first clock domain desires to send a signal to the second domain, it sets the data to be sent in a register that is kept constant for the duration of the transfer. A request for

transmit is then set high. Both actions occur in the first clock domain. After two rising edge clock cycles in the second clock domain, the request transmit becomes a request receive that enables the data to be stored in the second clock domain. This same signal propagates back as an acknowledge receive through another two clock cycle synchronization through the first clock domain. At the output of this synchronization the signal is considered an acknowledge transmit, as which point the processor has permission to begin transmitting more data.

3.3 Audio Synthesizer

A core part of the integrated I/O was our audio subtractive synthesizer. Building a fully integrated subtractive synthesizer required several key components, including a PWM-based digital-to-analog converter (DAC), a numerically controlled oscillator (NCO), and the hardware to synchronize all the components together.

3.3.1 NCO

At its core, an NCO is a systematic LUT. In order to create arbitrary waveforms we have LUTs with the values of the waveforms for a given input. The NCO manages how to apply inputs to the LUTs so as to get the appropriate waveform at a given signal frequency. To do so we have a phase accumulator, which increments an index linearly according to a particular signal frequency.

This index is then used as an input to the LUT to produce samples for a particular sampling frequency at a continuous rate. Since samples are produced continuously (at the clock frequency), it was important to design an enable interface that could be used to halt the NCO from producing samples (as described at the tail-end of section 3.1).

To achieve high precision in a LUT requires many data points and immense storage. In order to reduce the size of our LUTs, we incorporated linear interpolation of values between indices. The approach we took to linear interpolation was elegant and computationally simple, which is important given LUT-indexing frequency. We simply treated the interpolation bits as a weight in the interval $[0, 1)$, and computed a weighted average of the surrounding 2 index values from the LUTs. Once the weight was computed once, it could be used in all 4 waveforms (referenced 8 times in total) Our approach to interpolation allowed low memory usage in the LUTs while maintaining the fidelity of the signal.

3.3.2 Synth

Our subtractive synthesizer is made of several building blocks. Samples are produced for a given frequency by the NCO for all four designed waveforms: sine, triangle, sawtooth, and square. These waveforms then receive their own gain factor (equivalent to a right shift) before they are summed up and multiplied by a global gain factor (again a right shift).

The data produced by the NCO is both signed and 20 bits wide, but our PWM-based DAC takes in 12 bit unsigned values. Therefore, we truncate the data by taking only the 12 MSBs and adding 2^{11} to ensure the result is always positive. After that processing, samples are fed to a FIFO buffer, as described in section 3.1, followed by a CDC, as described in section 3.2. Finally, the resulting data is fed, in the PWM clock domain, to the PWM-based DAC for audio output.

3.4 `NOP` Injection

Another unique design choice was our approach to `NOP` injection. For branch instruction control hazards, if the branch prediction is incorrect then it is necessary to inject a `NOP` to correct the instruction flow. To do so, we utilized unused opcodes from the RISC-V specification manual. In particular, we designated an opcode for `NOP` instructions (`` `OPC_NOP ``), which would only be used by a failed branch prediction. This allowed us to easily modify the control logic for a `NOP` and correct the datapath

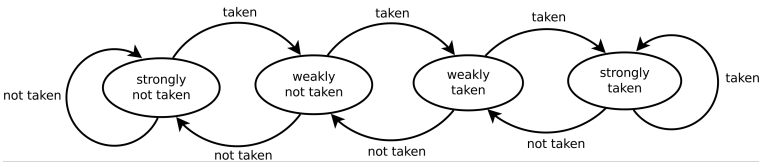


Figure 4: Two bit saturating counter state machine.

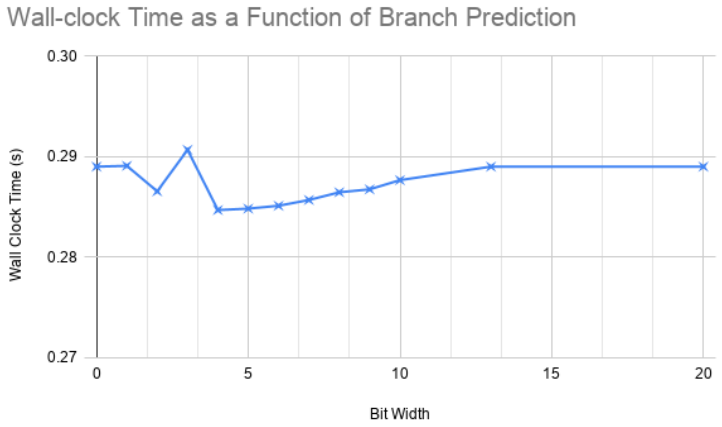


Figure 5: Wall clock time as a function of branch prediction saturating counter bit width.

based on assumptions that the `NOP` was set by a failed branch.

3.5 Extra Credit: Saturating Counter Branch Prediction

In pipelined processors, branch prediction is essential for increasing the cycles per instruction (CPI), and in turn the speed of the processor. Without branch prediction, the only way to resolve control hazards is by injecting `NOP` instructions when the control flow is unknown. However, this is very costly. In our first iteration of design, we implemented this method and achieved a very poor CPI. As a next improvement, we moved towards an always-taken prediction scheme. While this improved the CPI significantly, that was only for the `mmult.c` program. To further increase the prediction ability of our processor we moved towards a true branch prediction scheme, the saturating counter.

A saturating counter branch predictor works as shown in fig. 4. In this scheme the branch is predicted 'take' for the larger half of the values, and 'not take' for the smaller half of the values. Every time a branch is taken, the counter increments, whereas when the branch is not taken, it decrements. In this way the counter utilizes the history of previous branch instructions to predict the outcome of future branch instructions. Parameterizing the bit width of the counter modulates the impact of branches from long ago, and was tuned based on the reference workload.

The impact of branch prediction on our wall-clock time for the `mmult.c` program is shown in fig. 5. As shown, the optimal bit width for our saturating counter was 4. With larger bit widths the impact of history is too large and prevents accurate prediction, whereas with smaller bit widths the results are very noisy as they depend more on the given workload.

4 Status and Results

4.1 Functional Summary of Blocks

By the end of this project, we completed our implementations of a three-stage pipelined RISC-V processor that successfully runs the `mmult.c` program with the correct checksum. We also fully memory-mapped and integrated all functional blocks into our core CPU design, with the ability to demonstrate all programs' functionality in hardware. Our audio synthesizer is fully integrated and functional, and all of our various programs and blocks run at the same clock frequency.

4.2 Clock Frequency and Planned Improvements

Our processor operates at a maximum frequency of 50MHz (minimum clock period of 20ns).

If we continued working on this project, we would divide our critical path (outlined in section 2.1) into several substages. Currently, our critical path is caused by our ambitious forwarding, in which it is possible to exercise nearly the entire datapath in a single cycle if the hazards propagate across several instructions. Our post-route timing summary indicates that this path is the longest one by far, and we are approximately 0.132ns away from reaching 60MHz. This result is especially unfortunate since the reference workload likely does not even exercise the forwarding for this specific case.

Our solution to this performance pitfall is to forcibly insert `NOP`'s whenever the data that *can* be forwarded has already been forwarded. That is, even though we have the wiring in place to enable comprehensive, CPI-reducing forwarding, when excessive forwarding is occurring, it is performance-wise beneficial to reduce our CPI and insert a `NOP` to enable much greater clock frequency. By incorporating the division of the critical path in this way, and having analyzed the other lengthy paths in our design, we anticipate that our critical path will be divided into roughly 3 equal sections, which would enable our clock frequency to meet 100MHz or greater at ideally minimal increase in CPI since the critical paths are likely not exercised by `mmult.c`.

4.3 LUTs and SLICE register counts

The LUT and SLICE usage of our FPGA is outlined in section 4.3. This data was collected after the completion of the entire project. Unfortunately, due to timing constraints and a focus on core performance and functionality, we were unable to optimize the hardware usage.

Site Type	Used	Fixed	Available	Util %
Slice LUTs	5455	0	53200	10.25 %
Slice Registers	8049	0	106400	7.56 %

Table 1: FPGA Hardware Utilization

4.4 CPI and `mmult` Performance

Our processor operates with a CPI of 1.124 enabled by our tuned branch-prediction scheme (without which we operate at over 1.3 CPI due to the branching/jumping nature of the `mmult.c` program semantics). Our "walltime" for the program is approximately 0.284s.

4.5 Verification Approach and Summary

At every step of the project, we wrote an extensive number of unit tests, block-level testbenches, and integration-style system-level testbenches to ensure ease of debugging and for performance verification. We went beyond the provided testbench structures and generated our own to ensure full

coverage.

4.5.1 RISC-V and Processor Testing

In the early stages of our RISC-V processor, we wrote block-level testbenches for all core components of the pipeline, most notably for the ALU, Branch Comparator, Load Extender, Immediate Generator, Regfile, and others. This allowed us to have more confidence in our work and pointed at locations to debug when needed.

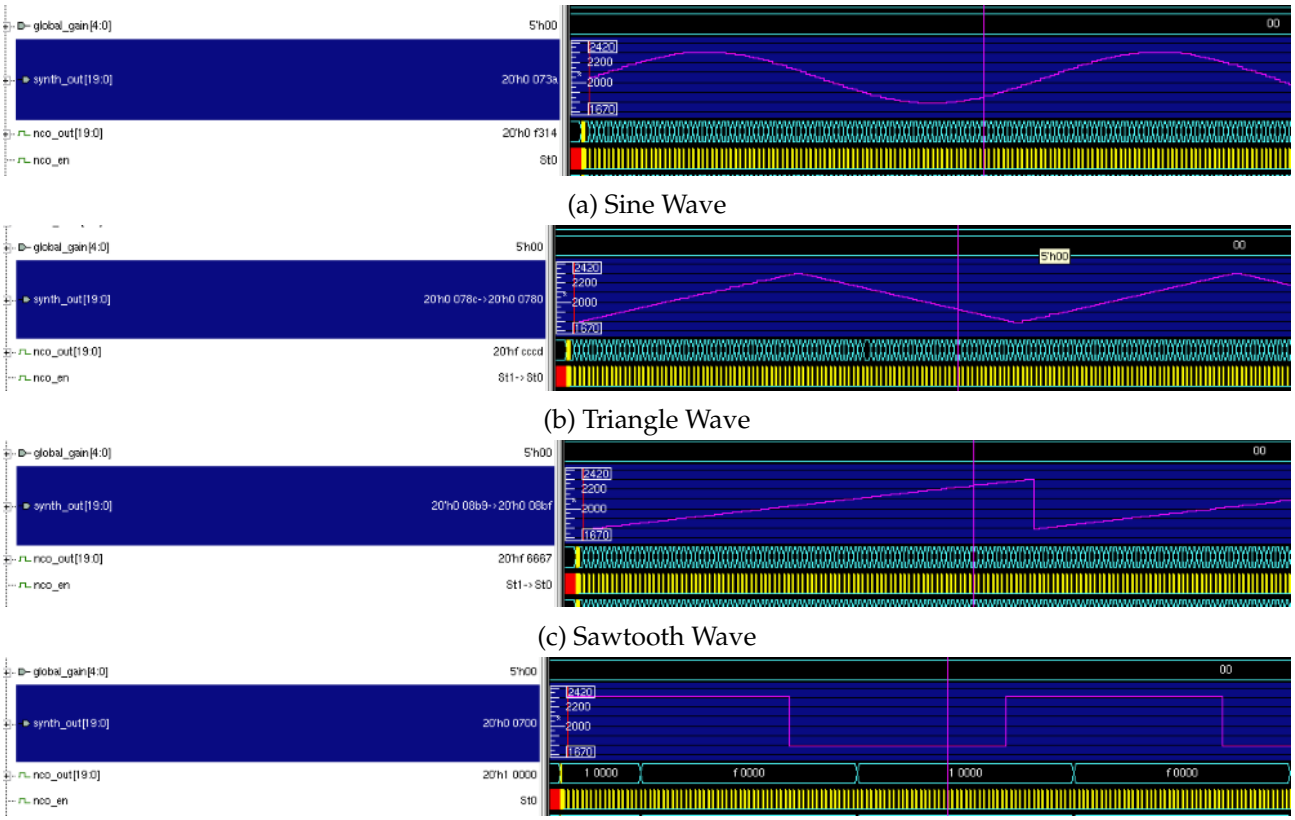
In order to make sure our processor worked for *all* instruction-ordering cases, we began by writing a comprehensive set of unit-tests for each instruction to exercise all functional options in the datapath. Once we graduated beyond unit-testing to running more integrated programs, we wrote a fully-featured pseudo-randomized assembly code generator in Python. The goal was to emulate an industrial-strength verification tool (exercising maximum functional coverage and code coverage) with a narrowly defined scope for this particular project. The script incorporated all categories of instructions, where we could set the relative frequencies of each instruction to tune the workload and lean towards a particular set of edge cases in a given trial run. This also allowed us to use fewer "simple" instructions, such as `slt, sltu` and more "complex" instructions, such as `xor, jalr`, etc. We could toggle the encouragement of data and control hazards, customize the number of instructions, and format expected register results. Each instruction was given a label for ease of specifying jumps, and the issue of infinite-looping was resolved by restricting jumps to a predefined, well-behaved range.

In the end, the assembly program was written to a text file where the expected register values could be computed using Venus for smaller programs, or the RISC-V compiler itself for larger programs ($\approx$ 10,000 instructions). The generation of this script saved us time in isolating several niche bugs and in later stages of the project, we never had to doubt our processor functionality. Especially when optimizing the datapath for clock frequency at the end of the project timeline, this script proved immensely useful for regression-testing, as the provided `isa-tests` did not seem to comprehensively test hazards and corner cases.

4.5.2 Audio Synthesizer Testing

For the second major stage of the project, we wrote testbenches to ensure functionality of the Phase Accumulator, NCO, and Synth individually and in concert. These testbenches verified our hardware observations with the benefit of being able to inspect waveforms at a very refined data resolution, to catch errors that might not be discernible in hardware. We also discovered how to use DVE in much more powerful ways than before, including being able to search a set of waveforms for a specific condition (which is near-impossible to do in a 10ms simulation by eye).

We ran the provided `NCO.py` and `Synth.py` scripts to compare against our waveform for a wide variety of harmonic combinations (different weightings of various waveform types) and frequency control words. We found that our waveforms matched the predicted values in every case, lending credence to our approach and providing functional verification of our code. Not only did the golden values and other specific numbers match between the script and waveform, but the analog version of the output waves matched the expected shape, and became even better after incorporating linear interpolation, as shown in fig. 6. We also performed an audio match between our FPGA board's output sound and some online references for various waveform shapes.



(a) Sine Wave

(b) Triangle Wave

(c) Sawtooth Wave

(d) Square Wave

Figure 6: Verified analog waveforms produced by the subtractive synthesizer with linear interpolation.

5 Conclusions (both)

5.1 Reflections

Overall, our project was definitely successful. We spent more time than many other groups on our actual block diagram, but the familiarity we gained with the processor design, the layout of various modules, and the semantics of pipelining enabled us to successfully complete the processor checkpoint comfortably within the time limit.

When it came time for checkpoint three, however, we were not as on top of the time-table, with finalized implementation of the audio synthesizer happening the night before check-off. This is partially attributable to the lack of clarity in the project specification, particularly as compared to checkpoint two. In checkpoint two, the detail in the specification was outstanding and made implementation a matter of reading the specification and figuring out how that mapped to our design. However, in checkpoint three, at times the design was confusing. This was particularly the case with the frequency control word, which, barring two mentions in the spec, was never properly defined and led to significant (two full days worth) confusion. The provided C program also had a bug that took us time to figure out, and we ended up doubting our design (which was correct all along). Both of these issues were resolved in late-week office hours. As a result we were unable to spend the amount of time on optimization as we would have liked to, specifically with regards to increasing our clock frequency.

We also learned the importance of systematic functional verification to ensure the behavior of our

modules. By creating many unit-tests, block-level tests, and integration tests along the way, the usage of which took significantly more time than we spent on the actual code, we had complete confidence in our processor's correctness in later stages of the project.

5.2 Insights for Next Time

If we were to do this project over again, we wouldn't change anything about our fundamental approach to the project; however, we would incorporate considerations about clock speed into our initial processor and forwarding-logic design so that we don't unintentionally create a very lengthy critical path. Additionally, we would plan our work schedule around the office hours of the TAs; this would allow us to get our questions clarified more quickly as they come up. For checkpoint 3, we would have started the checkpoint earlier, though the timeline of other class projects made this difficult.

Overall, we were able to complete all the primary functional aspects of the project, and the feeling of seeing the audio synthesizer and the processor work was unparalleled. The incorporation of practical hardware-based memory-mapped I/O peripherals was tangible and quite satisfying.

6 Division of Labor

Documents have been submitted separately.