University of California Berkeley
Department of Electrical Engineering and Computer Science
EECS151/251A Fall 2019

Project Report

Li Yang KAT
Xuan SU

February 28, 2020

1 Project Functional Description and Design Requirements

In this project, we designed and implemented a 3-stage pipelined RISC-V central processing unit [CPU] with a memory mapped input/output [I/O] interface. A polyphonic synthesizer and a digital-to-analog converter [DAC] were also implemented to generate different audio waves and produce polyphonic tone output.

1.1 Pipeline Structure

The pipeline structure of the RISC-V CPU is shown in Figure 1. The instruction is fetched from the instruction memory [IMEM] before the start of the first stage. During the first stage, the instruction is decoded and the registers from the register file used in the instruction are accessed and the immediate is generated at the same time. In the second stage, the arithmetic and logic unit [ALU] performs the necessary operations and data is written to and read from memory (BIOS/IMEM/DMEM/MMIO). The branch comparison is also performed in the second stage. In the last stage, the data is written back to the register file.

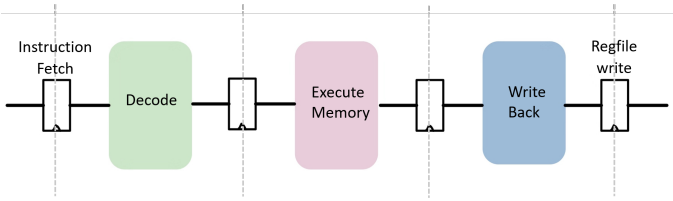

Figure 1: RISC-V 3-stage pipeline structure

1.2 Memory Hierarchy

The memory architecture consists of three random access memories [RAM] - the BIOS memory [BIOS], the instruction memory [IMEM] and the data memory [DMEM]. In addition, there is also memory-mapped I/O to peripheral registers that control and access data from peripherals such as the universal asynchronous receive and transmit [UART] interface, buttons, switches, LEDs, wave generator co-processor and DAC. All the memories are both synchronous read and write. A BIOS program is run at processor reset to load instructions into the instruction memory over UART and jump to the start address in the instruction memory to execute a program.

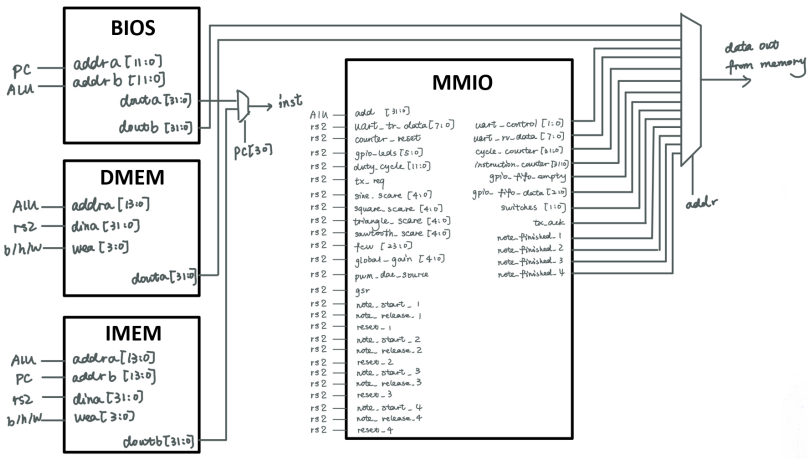

Figure 2: RISC-V Memory Hierarchy

2 High-level Organization

2.1 Block diagram of Z1TOP

The block diagram of the top module is shown in Figure 3. It consists of three parts, the RISC-V CPU, an asynchronous first-in-first-out [FIFO] and a DAC. The phase-locked loops [PLL] that generate the clocks for both the CPU and DAC are not shown. The asynchronous FIFO transfers 12-bit digital data generated by the wave generator, from the clock domain of CPU to the clock domain of DAC. The DAC was designed to generate a pulse density modulated [PDM] signal. The 12-bit digital input of the PDM module, whether generated by the CPU or by the wave generator, is selected by the DAC source multiplexer [MUX].

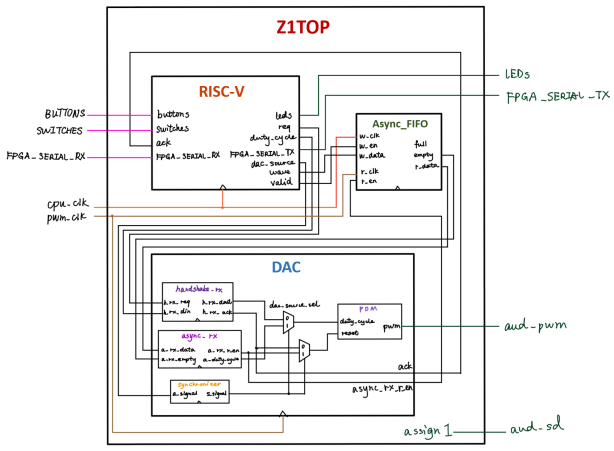

Figure 3: Block diagram of Z1TOP

2.2 Block diagram of RISC-V CPU (without Wave Generator)

The block diagram of the RISC-V processor is shown in Figure 4.

In the first stage, the register file is accessed and the immediate is generated based on the instruction fetched. A branch predictor is included in this stage to reduce control hazards.

In the second stage, the branch comparator compares the data in two registers for *BRANCH* instructions. The ALU calculates any arithmetic results, memory addresses for *LOAD/STORE* instructions as well as the branch target address for *BRANCH* instructions. The **MEM_wsel** module determines which memories the data should be written to based on the address calculated by the ALU.

In the last stage, the **MEM_rsel** module determines which data output of the memories should be written back to the registers and the load extension module extends the data output as necessary. The outputs of the load extension, ALU and PC+4 are selected in the **wb_sel** module. Since the register file is synchronous write, the write back data is forwarded to both the first and second stages to avoid data hazards.

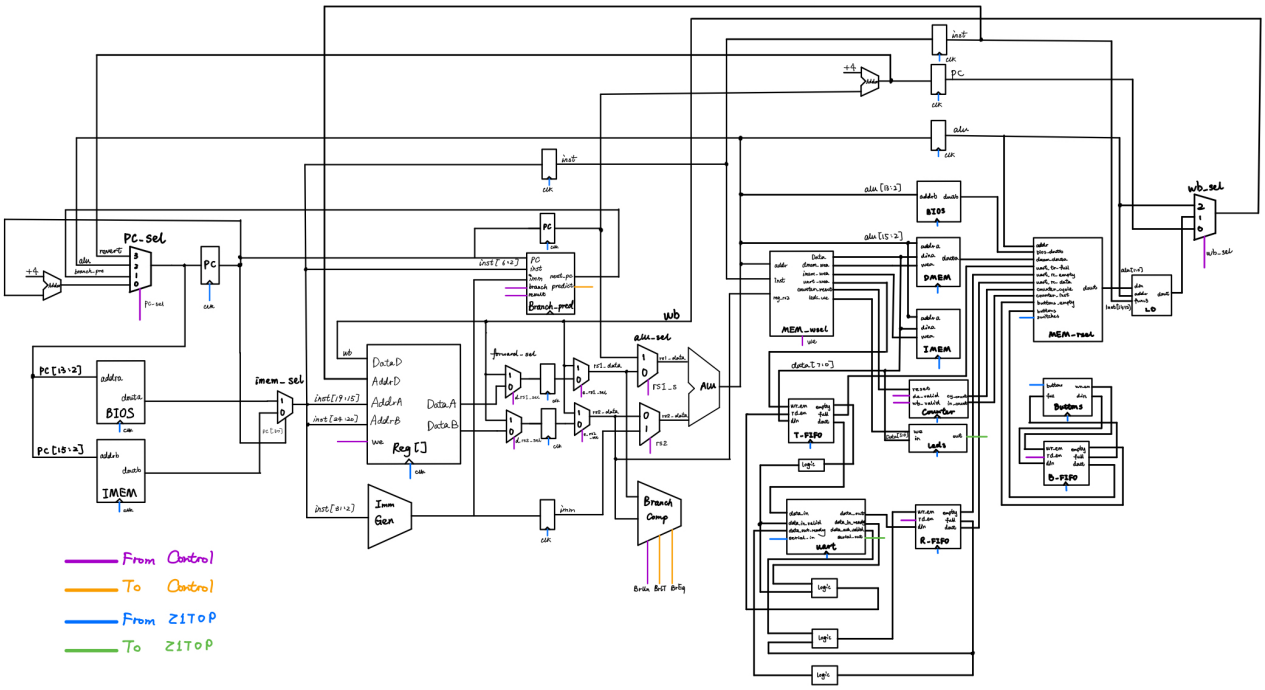

Figure 4: Block diagram of RISC-V (without Wave Generator)

2.3 Block diagram of the Audio Synthesizer

The block diagram of the Audio Synthesizer is shown in Figure 5. There are two ways the RISC-V CPU can generate a PDM output, either by writing a 12-bit digital value directly to a register mapped in MMIO, or by writing to registers that control a co-processing wave generator. Directly writing the 12-bit digital value to the MMIO register passes the value directly to the DAC using a 4-phase handshake protocol for clock domain crossing, while the wave generator passes values using an asynchronous FIFO.

The **Wave_Generator** module comprises a numerically-controlled oscillator [NCO] with four independent phase accumulators and their corresponding state variable filter [**SVF**] modules and attack-decay-sustain-release [**ADSR**] modules. In addition, there is also a **Gain_Truncate** module that scales the output and converts the signed fixed-point sample to a 12-bit unsigned value.

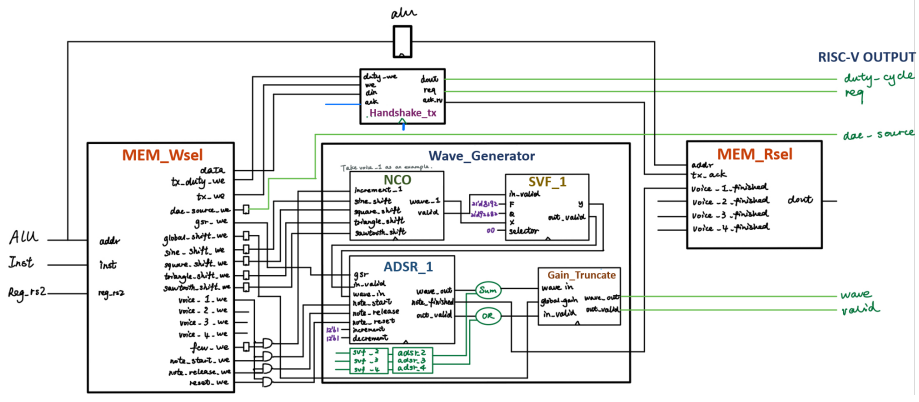

Figure 5: Block diagram of the Audio Synthesizer

3 Detailed Description of Sub-pieces

3.1 RISC-V Core

3.1.1 Branch Predictor

A branch predictor was implemented in the ***Decode*** stage to minimize the killing of instructions in the default "branch not taken" implementation of the CPU. An overview of the branch predictor is shown in Figure 6. In the **Branch_Pred** module, a 5-bit register keeps track of the last 5 branch results. The contents of this register addresses a 32-row, 2-bit wide table, where each entry is a saturating counter. This saturating counter represents a Finite State Machine [FSM] with four states - 00, 01, 10 and 11. 00 implies that an incoming branch should be strongly not taken, 01 implies that it should be weakly not taken, 10 implies that it should be weakly taken and 11 implies that it should be strongly taken. The FSM is shown in Figure 7. A branch is predicted whenever bit 1 of the counter is 1, that is to say, the FSM is in the weakly taken and strongly taken states.

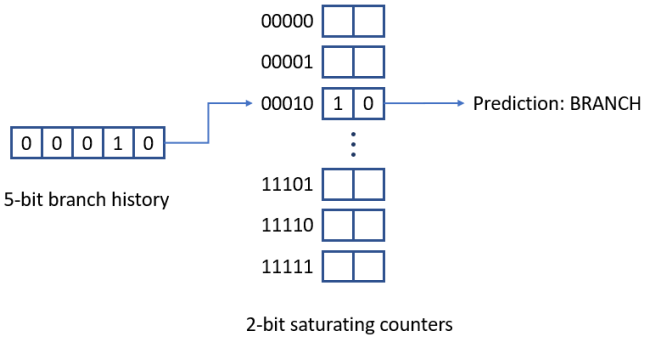

Figure 6: Branch Predictor Overview

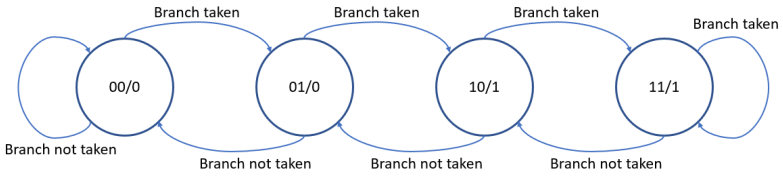

Figure 7: State Transition Diagram of the Branch Predictor

The branch history and the saturating counters are updated whenever a branch instruction reaches the ***Execute*** stage, regardless of whether the branch was predicted or not. The saturating counter increments if the result is a branch and decrements if the result is a no-branch. The branch history is left shifted and the branch result inserted into the lowest significant bit [LSB].

The improvement in cycles per instruction [CPI] when using the branch predictor is dependent on the bit width of the branch history register. The wider the bit width, the more accurate the prediction and the lower the CPI. However, increasing the bit width worsens the critical path delay. This relationship is elaborated more in the Optimization section, but as an overview, our CPI for the mmult benchmark program improved from 1.18 without branch prediction to 1.06 with 4-bit history, 1.03 with 5-bit history, and 1.03 with 6-bit branch prediction.

3.2 Wave Generator

3.2.1 Numerically Controlled Oscillator

The NCO is a digital circuit that generates quantized, discrete-time waveforms. Our NCO consists of four independent channels, with each channel capable of generating sinusoidal, square, triangle and sawtooth waveforms. Four 24-bit phase accumulators generate phase representations of each independent channel which are then used to address lookup tables, implemented as read-only memories [ROM]. A simple comparator is used for the square wave to reduce ROM use. The output of the NCO is a 21-bit signal, with 1 sign bit, 4 integer bits and 16 fractional bits.

Quarter Wave Symmetry

Generating a quantized output from the phase accumulators implies that the resolution of the output waveforms largely depends on the depth of the lookup table. A deeper lookup table would generate more precise outputs, since more bits from the phase accumulator can be used to address into the table. While the recommended implementation involves the storing of one complete 2π revolution of each waveform in the lookup table, our implementation involves the storing of only the first quarter, or $0 < \phi < \frac{\pi}{2}$, of the waveform, exploiting the fact that all the waveforms required can be fully represented using the first quarter of the waveform. This enables us to either reduce the depth of the lookup tables by a factor of 4 to achieve the same precision, or increase the precision by 2 bits without increasing the depth of the lookup tables.

One of the subtleties in this implementation is the choice of values to store in the lookup tables. Figure 8 demonstrates the importance of this choice.

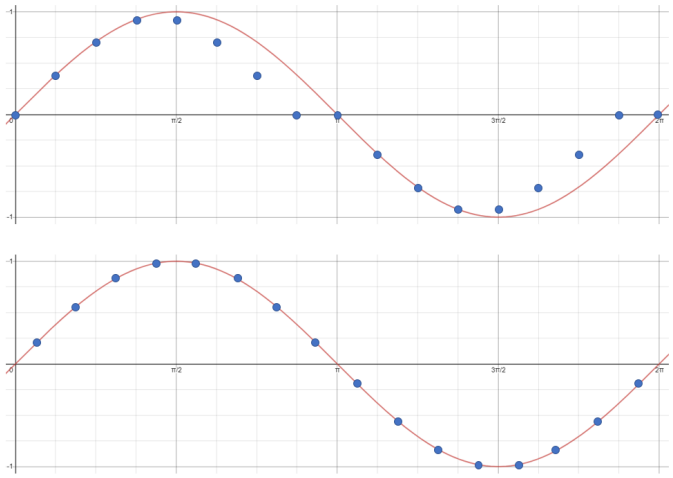

Figure 8: Quarter wave symmetry ROM contents consideration

In Figure 8, a sine wave is sampled at 16 discrete points. By exploiting quarter wave symmetry, we can reduce the number of samples to 4. However, the top graph shows a sine wave generated by the NCO if we naïvely choose to store the first four values from the full wave lookup table. The reflection of the wave for $\frac{\pi}{2} < \phi < \pi$ and $\frac{3\pi}{4} < \phi < 2\pi$ leads to errors in the output values which would surface as increased harmonic distortions in the output spectrum. On the contrary, the bottom graph shows a sine wave generated by the NCO if we shifted the sampled values by half a sample. This leads to a phase shift in the output sine wave but results in a much more accurate reconstruction of the sine wave with less harmonic distortion.

Interleaved Read-Only Memory Access

The graduate student requirement for EECS 251A is to implement a polyphonic audio synthesizer instead of a monophonic one. To do this, one method is to simply create four instances of the NCO module and sum the output. However, doing so consumes four times more memory resources for the NCO. This is largely unnecessary, especially for the ROMs used for the lookup tables, since all the ROMs contain the same lookup tables. Furthermore, since the audio sample rate is much lower than the CPU clock frequency, there are no problems with sharing one ROM for each waveform across all four NCOs.

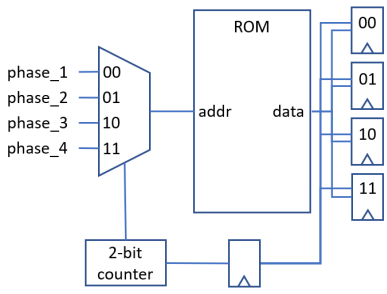

Figure 9: Conceptual view of interleaved ROM access

Figure 9 shows a conceptual view of how the interleaved memory access is performed. A MUX selects the relevant address based on the phase of operation of the NCO. Registers are then used to store the output of the ROM for different phase outputs of the four phase accumulators. The two-bit counter is delayed by one cycle before being used to enable the different registers, since the ROMs are synchronous read.

In practice, the control of the MUX and registers is implemented using a shift register. Using "valid" signals from the phase accumulator to indicate when a phase output is valid, the signal is shifted through the shift register and the bits in the shift register are used to enable different elements of the design.

First Order Interpolation

From the 24 bits output of the phase accumulator, the MSB is used to determine whether the wave is in the positive or negative region. The next bit is used to determine if the wave should be reflected (for the sine and triangle waveforms) or offset (for the sawtooth waveform). The next 8 bits are used to address into the ROM, leaving 14 bits. For our implementation, these 14 bit are used to perform linear interpolation between the quantized values to reduce the spectral distortion caused by the 10-bit quantization.

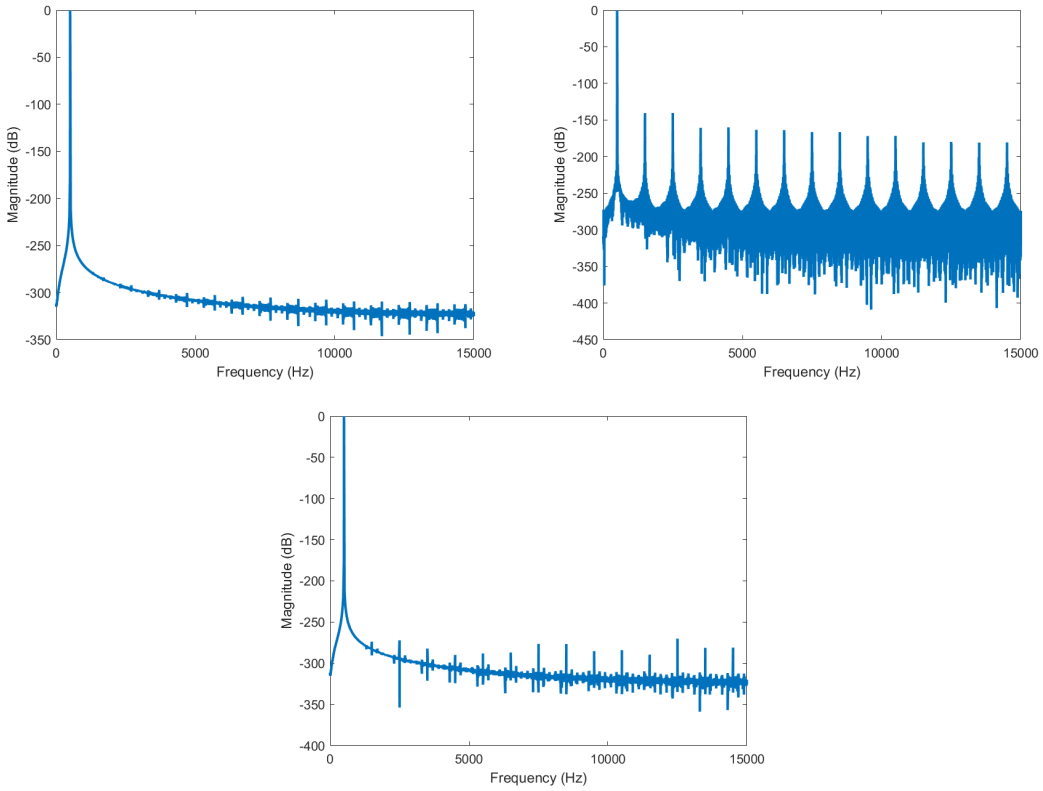

Figure 10: Spectral distortion caused by quantization and interpolation

Figure 10 shows how interpolation improves the spectral distortion. The spectrum of a 500 Hz wave sampled at 30 kHz, where all 24 bits of phase are used to calculate the amplitude of the waveform at each sample, is shown in the top left graph. The top right graph shows the spectrum of the same wave where only the most significant 10 bits, as in the case where quarter wave symmetry is exploited, are used to calculate the amplitude. Clearly, there is significant distortion. The bottom graph shows the spectrum of the same wave when only the most significant 10 bits are used to calculate the amplitude, but the remaining 14 bits are used to interpolate to the first order. The harmonic distortion is greatly reduced, almost to the level where all 24 bits are used.

3.2.2 State Variable Filter

The SVF is an infinite impulse response [IIR] filter that filters the input signal x, into three bands, high pass, band pass and low pass. The notch filter output is generated simply by summing the high pass and low pass outputs. By modifying Q, the quality factor, and F, which controls the cutoff frequency, different audio effects can be obtained in combination with the four different waveforms. We implemented the SVF using the difference equations provided, as follows.

$$
\begin{aligned}
y_l(n) &= Fy_b(n-1) + y_l(n-1) \\
y_h(n) &= x(n) - y_l(n) - Qy_b(n-1) \\
y_b(n) &= Fy_h(n) + y_b(n-1) \\
y_n(n) &= y_h(n) + y_l(n)
\end{aligned}
$$

We can transform these equations to be all in terms of register outputs. This optimization was performed to reduce the effects of the SVF on the critical path of the design.

$$
\begin{aligned}
y_l(n) &= Fy_b(n-1) + y_l(n-1) \\
y_h(n) &= x(n) - Fy_b(n-1) - y_l(n-1) - Qy_b(n-1) \\
&= x(n) - (F+Q)y_b(n-1) - y_l(n-1) \\
y_b(n) &= F[x(n) - (F+Q)y_b(n-1) - y_l(n-1)] + y_b(n-1) \\
&= Fx(n) - [F(F+Q)-1]y_b(n-1) - Fy_l(n-1) \\
y_n(n) &= y_h(n) + y_l(n) \\
&= x(n) - (F+Q)y_b(n-1) + Fy_b(n-1)
\end{aligned}
$$

Using this transformation, we assume that the coefficients, $(F+Q)$ and $[F(F+Q)-1]$ can be pre-computed and passed as inputs to the **SVF** module. This greatly reduces the critical path since it removes the dependencies between the different filter outputs.

In computing the multiplication operations, an intermediate wire with double the bit width is used since F and Q (and their derivatives) are also represented using 1 sign bit, 4 integer bits and 16 fractional bits. With constrained values of F and Q, the intermediate and final values of the SVF do not overflow and the filter demonstrates stability. However, we do observe some small transient oscillatory behavior characteristic of an IIR filter.

3.2.3 Attack-Decay-Sustain-Release

The implementation of the ADSR module is relatively simple. Within the ADSR module, there is a factor by which the output of the SVF is multiplied to scale the output of the ADSR module. At every clock cycle after the "start" signal is given, this factor is increased by a certain "increment" value provided by the CPU. When the factor is detected to be about to overflow, the ADSR enters the state where it sustains the output, with the factor at its maximum value.

In a similar manner, the decay is implemented by decreasing the same factor every cycle, by a "decrement" value provided by the CPU, after the "release" signal is given. The "finished" signal is output when the factor is detected to be about to underflow and the ADSR enters the state where it is done and awaiting reset, with the factor reset to zero.

3.3 Clock Domain Crossing

3.3.1 Asynchronous FIFO

Unlike a synchronous FIFO where the read and write channels are in the same clock domain, an asynchronous FIFO operates across two clock domains. The asynchronous FIFO is a common technique used to transfer multi-bit data from one clock domain to another. In our case, the asynchronous FIFO is used to transfer data from the output of the **Wave_Generator** module operating at 100 MHz to the **DAC** module operating at 150 MHz.

The read pointer of the asynchronous FIFO is kept in the read clock domain, r_clk, while the write pointer is kept in the write clock domain, w_clk. Thus, we cannot simply compare the two pointers. For example, the write pointer cannot be simply passed to and sampled in the read clock domain, as changes in the write pointer could lead to metastability of any register in the read clock domain that samples logic with the write pointer as an input.

To pass the read and write pointers to the opposing clock domain, the binary pointers are converted to gray code, so that only 1-bit would change at every transition. To be more specific, the write pointer and read pointers are converted to gray code and subsequently transmitted to the opposing clock domain through two flip-flop synchronizers. After the synchronizers, the two pointers are converted from gray code back to binary code to be compared, to check for

the full and empty conditions. When the read and write pointers point to the same address, the FIFO is empty, and when the MSB of the read and write pointers are different while the remaining bits are identical, the FIFO is full.

3.4 Digital-to-Analog Converter

3.4.1 Pulse Density Modulation

Instead of the suggested pulse width modulation [PWM], we implemented PDM instead. At the 150 MHz operating frequency of the DAC, the highest audio sampling rate that can be generated reliably and accurately by a PWM module is $\frac{150\times10^6}{4096} = 36621Hz$. This is below the Nyquist frequency of the limit of human hearing of $20kHz$. Furthermore, PWM posed other problems, such as producing an inaccurate reproduction of the signal especially at higher sampling rates when the sampling rate is not a whole multiple of 4096.

PDM solves this problem as the pulse density in the pulse density in PDM adapts quickly to changes in the input signal, whereas PWM takes an entire 4096 cycles to represent one output value. The fundamental understanding behind the digital PDM is similar to sigma-delta modulation used in sigma-delta ADCs and DACs. The error in the output is accumulated in an accumulator every clock cycle by subtracting the input from the actual PDM output, which is represented by the 1-bit PDM output replicated by the bit width of the input signal, and adding the error in the previous clock cycle. The 1-bit output of the PDM is generated by comparing the input with the accumulated error. If the input is greater than the accumulated error, the output is high, otherwise it is low. Using this negative feedback mechanism, the fractional average value of the PDM output is equal to the fractional input data. With PDM output instead of PWM, we were able to raise our sampling rate to 100 kHz for better audio quality. Unfortunately, the audio jack output of the Pynq board and the headphones used were unable to maximize the quality from the higher sampling rate.

4 Status and Results

4.1 Testing

4.1.1 Sub-module Testing

Testbenches for some of the more crucial sub-modules, including register file, ALU, branch comparator, branch predictor, immediate generator, load extension, memory write multiplexor, NCO, SVF and wave generator, were designed to test the functionality of each module. Many errors in the register transfer level [RTL] code were detected using these testbenches. As a testament to the testbenching of our sub-modules, we were able to pass the isa-tests in one iteration.

Testing the Branch Predictor

A simple testbench was designed for functional testing of our branch predictor. 13 branch results were fed into the branch history register inside the branch predictor sequentially. During this 13 test cycles, the branch was taken in the first 6 tests and the last 3 tests, while not taken from Test 7 to 10. As is shown in Figure 11, our branch predictor predicts the branch to be taken after the first three branches were predicted to be not taken. Similarly, when the branch changed to “not taken” at Test 7, the branch predictor predicted the branch to be not taken after two failures. The same situation happened when the branch changed back to “taken” at Test 11.

```
Test 1 failed:  branch:00001, result:1, predict =:0
Test 2 failed:  branch:00011, result:1, predict =:0
Test 3 failed:  branch:00111, result:1, predict =:0
Test 4 passed:  branch:01111, result:1, predict =:1
Test 5 passed:  branch:11111, result:1, predict =:1
Test 6 passed:  branch:11111, result:1, predict =:1
Test 7 failed:  branch:11111, result:0, predict =:1
Test 8 failed:  branch:11111, result:0, predict =:1
Test 9 passed:  branch:11111, result:0, predict =:0
Test 10 passed:  branch:11111, result:0, predict =:0
Test 11 failed:  branch:11111, result:1, predict =:0
Test 12 failed:  branch:11111, result:1, predict =:0
Test 13 passed:  branch:11111, result:1, predict =:1
```

Figure 11: Simulation result of the branch predictor

Testing the Wave Generator

One of the most important techniques in RTL design is the comparison of outputs in simulation against a high-level behavioral model. Because our design differed from the suggested design significantly, in that we exploited quarter wave symmetry and implemented the SVF and ADSR modules, we wrote our own behavioral model in MATLAB to compare the outputs of our simulations against. These behavioral models varied in complexity as well. We started out by testing the algorithms without using fixed point arithmetic, to check that our understanding of the algorithms was correct. Next, we introduced fixed-point arithmetic so that we could compare the outputs of our simulations with the outputs of the behavioral model. To verify that our fixed point arithmetic in MATLAB was identical to the behavioral models provided in Python, we also implemented the NCO using the suggested method first and compared the outputs of the MATLAB and Python models.

4.1.2 System Level Testing

Our general strategy for system level testing for different feature sets is to first run a short assembly test to read and write from different registers and check that the desired output is achieved. Next, we would write a simple C testbench to test that compiled code runs in a similar manner. Then, we would test the different programs provided in simulation. Lastly, we would test the program and RTL design on the FPGA.

Many simulations were conducted to test the functionality of our design in incremental steps, especially in the third part of the project. Before running the square_piano program on the FPGA, we first loaded the duty cycle values and the TX request signal to the corresponding addresses inside the RISC-V MMIO via a PDM assembly test, in order to check that our 4-phase handshake and PDM module works. Next, the square_piano.hex file was loaded into the IMEM and DMEM of the RISC-V in simulation and the off-chip UART was instantiated and connected to the top-level module in our testbench. In this way, we could simulate the input of the UART and check whether the top-level module could generate the PDM output successfully.

Several assembly files were also written to test our synthesizer. Taking the polyphonic synthesizer assembly file as an example, we first loaded the DAC source and the shift amounts of the global, sine, square, triangle and sawtooth waveform to their corresponding addresses, and then loaded the frequency control words, as well as the note start signals of each voice to start sending samples to the PDM in the DAC. Before raising the note release signals, a loop was inserted to generate some time delay between the start and the release actions. After the notes were released, we checked the note finished flags stored from the MMIO addresses and then reset the program. In addition, a C program was designed to test our final polyphonic synthesizer. Using the C program provided in square_piano as a basis for modifications, the switches were defined to select different waveforms. The frequency control words replaced the switch periods in the original file, and all the note release signals as well as the note reset signals were set to high when the time counter was larger or equal to the note length. Before running the program on the FPGA, we simulated the result by loading it to the IMEM and DMEM and sent a "#" symbol to our off-chip UART via the testbench. The simulation result is shown in Figure 12.

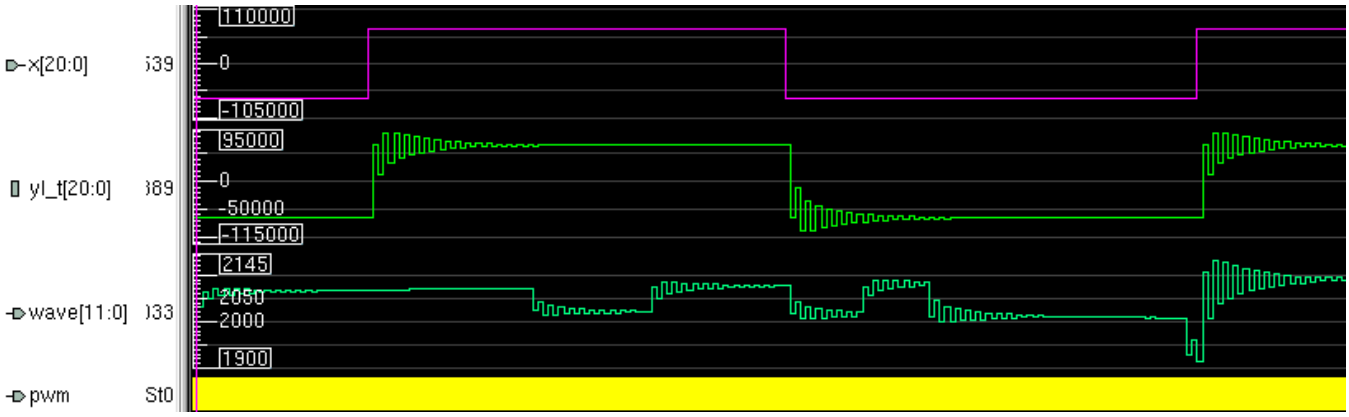

Figure 12: Simulation result from system level testbench

In Figure 12, we can see that the output waveform for the input to the PDM is a sum of three low pass filtered square waves. Here x is the input wave to the SVF. yl_t is the low pass output of the SVF, when $F = 1$ and $Q = \sqrt{2}$. We observe that there is indeed low pass filtering of the square wave, with some ringing due to the Q value. wave is the output of the **Wave_Generator** module. It is the result of the sum of three low pass filtered square waves, with an increasing amplitude due to the ADSR modules.

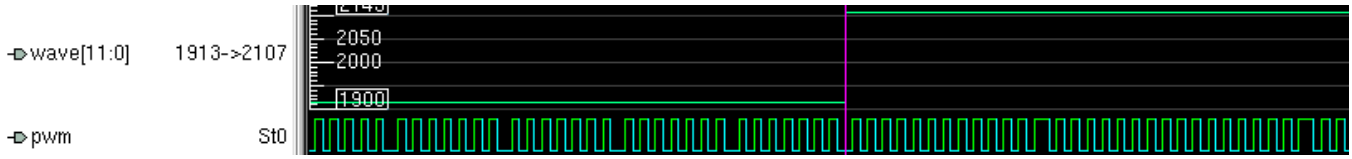

Figure 13: Simulation result of PDM

In Figure 13, we can see that the PDM is working as expected. Towards the left of the waveform, the input value is less than half, and so there are parts where the PDM output is low for more than 1 consecutive cycle. Towards the right of the waveform, the input value is more than half, and so there are parts where the PDM output is high for more than 1 consecutive cycle. Since both values are relatively close to half, the PDM output oscillates between 0 and 1 for the most part of the PDM waveform.

4.2 Status

At the end of the project, we have all aspects of the project, including optional parts such as the branch predictor, SVF, ADSR, asynchronous FIFO and PDM working as intended at a clock frequency of 100 MHz. The mmult program runs with a CPI of 1.06.

4.3 Resource Consumption

Our design uses a total of 1439 SLICEs and 4096 LUTs. Our resource consumption may be higher as we implemented several optional modules including the **SVF** and **ADSR** modules.

4.4 Design Space Exploration and Optimizations

Design	**Clock frequency**	**mmult CPI**	**mmult execution time**
CPU from checkpoint 2	95 MHz	1.18	0.157s
Branch always taken	95 MHz	1.17	0.156s
2-bit history branch prediction with wave generator	90 MHz	1.13	0.159s
4-bit history branch prediction with wave generator	90 MHz	1.06	0.149s
5-bit history branch prediction with wave generator	90 MHz	1.03	0.145s
6-bit history branch prediction with wave generator	85 MHz	1.03	0.145s
6-bit history branch prediction with dedicated branch ALU and wave generator	85 MHz	1.03	0.145s
6-bit history branch prediction with dedicated branch ALU, reduced SVF parameter bit width and wave generator	85 MHz	1.03	0.145s
5-bit history branch prediction with flattened SVF and wave generator	95 MHz	1.03	0.138s
4-bit history branch prediction with dedicated branch ALU, flattened SVF and wave generator	100 MHz	1.06	0.134s

The most significant optimization we managed to do was to do branch prediction in our ***Decode*** stage. Because our original CPU started from a relatively high clock frequency, it was difficult to push the frequency further without increasing the number of pipeline stages or removing the data forwarding paths.

We observed that including the wave generator in our design reduced our highest achievable clock frequency. This is likely because the increase in resource consumption implies that there is an increase in the number of logic paths that need to meet the same timing requirement, leading to congestion along certain routes. This was of particular concern for the SVF, which required performing consecutive multiplication operations in feedback loops that are difficult to pipeline.

We also noted that increasing the bit width of the branch history register improved the accuracy of the branch prediction. This is to be expected since an increase of 1 bit in the branch history register increases the number of entries by two times. This means that the branch history table can record the branch outcomes of double the number of patterns, based on the branch history. At the same time, we also noted the diminishing marginal returns

of increasing the bit width. Once we reached approximately a bit width of 5 to 6, there was no longer much benefit to be gained from increasing the bit width further. Instead, doubling the number of entries in the branch history table constrains the place and route of the design and reduced our highest achievable clock frequency instead.

In the design iteration where we had a 6-bit history branch prediction, the critical path was observed to be the forwarding path from the output of the **DMEM**, through the writeback select MUX, through the forwarding MUX, and through the ALU, back to the address of the **DMEM**. In an attempt to reduce the path delay and compensate for the reduction in highest achievable clock frequency due to the increase in branch history bit width, we added a dedicated branch ALU, that only performs the addition of the program counter [PC] and the immediate. However, this was unsuccessful as the critical path became one of the combinational paths in the SVF IIR filter.

Following that, we removed the additional module and attempted to reduce the critical path by reducing the bit width of the F and Q parameters in the **SVF** module, hoping that the reduced resource consumption would improve the timing of our design. However, the critical path remained at a path in the SVF.

We then proceeded to flatten the **SVF** module by making the assumption that certain coefficients derived from F and Q in the SVF can be pre-computed and passed as inputs to the module. Doing so removed the combinational dependencies between the SVF filter outputs and we achieved the original highest achievable clock frequency of 95 MHz.

We tried to reach 100 MHz by adding the dedicated branch ALU back to the design again. However, we were unable to reach 100 MHz frequency with 5-bit branch history prediction. We were close with a slack of -0.039 ns, but eventually we had to reduce the branch history register to 4 bits to achieve 100 MHz. However, since our CPI increased by less than 3% while our clock frequency increased by more than 5%, it is still an overall improvement in the execution time of the mmult program.

5 Conclusions

Through this experience of designing an entire CPU almost from scratch, we learned a lot about various aspects of digital design.

a. When designing the system architecture, it is easier to start from a high level specification and work towards smaller implementation details than the other way around. With this in mind, drawing out block diagrams greatly helps with the subsequent implementation, since we would have already partitioned the design into logical modules that can be easily implemented gradually.

b. It is often more productive to split up the work between team members for design and verification. For example, for a particular module, one team member would design the module, while the other performs functional verification. It is often difficult to spot our own mistakes, and having someone else check through the RTL is a great way to reduce debugging time.

c. Ensuring strict specifications for different modules allows us to operate at a higher level of abstraction. For example, we were able to quickly replace our original NCO, that only generated a single channel, with one that was capable of generating all four channels using interleaved ROM access, since our surrounding modules generated and accepted particular "valid" signals that indicated when a sample was ready.

There are several aspects we would definitely like to improve on if we had the chance to complete another similar project in future.

a. We think our CPU would benefit from an additional pipeline stage. Our CPI for the mmult program is rather low. This means that our branch prediction is pretty accurate. As such, increasing the number of pipeline stages would probably not result in an unreasonable increase in CPI in the case of mispredictions.

b. We would definitely like to try more advanced techniques in limiting the critical path of the SVF filter using loop unrolling techniques.