Fall 2019 EECS 151 Project Final Report

Team 06
Jerry Zhou, 3032707381
Jakob Kuki, 3033249819

December 10, 2019

Contents

1 Functional Description and Design Requirements

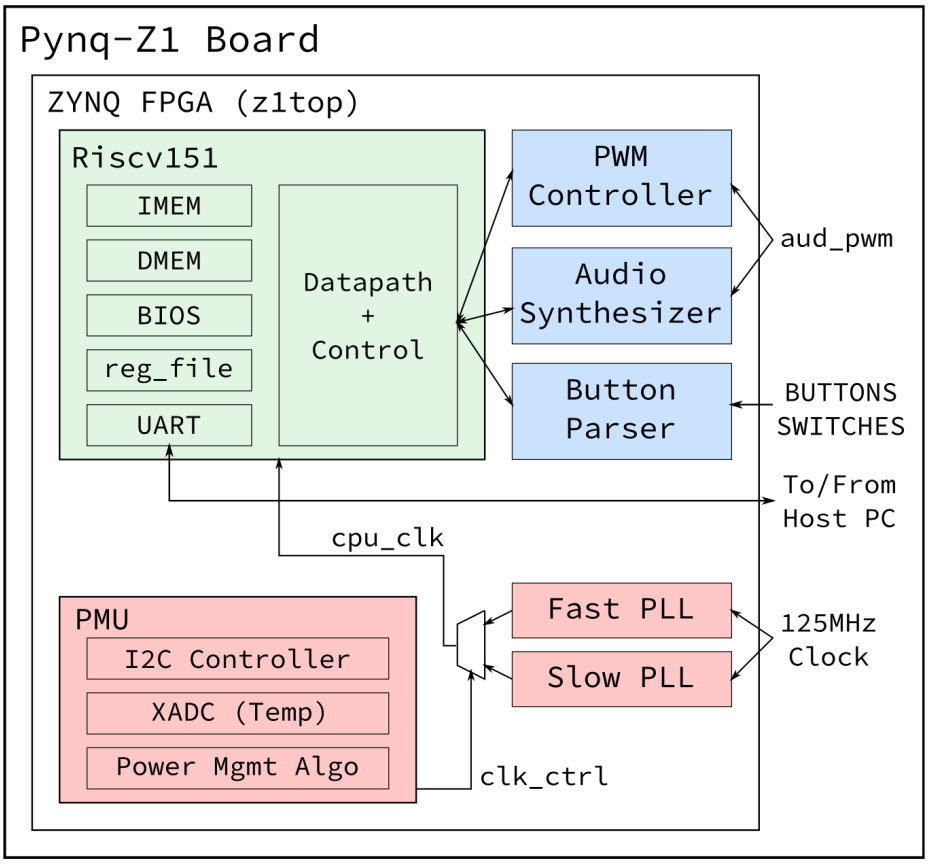

Figure 1: High-Level Overview of the System

The objective of this project is to create a 32-bit pipelined RISC-V CPU equipped with various memory mapped peripherals (Figure 1). The processor pipeline is divided into 3 stages with organization designed around data and control hazards. The processor consists of 3 separate memories: a BIOS ROM, an instruction memory, and a data memory. On boot, a small BIOS program will be executed which would allow the machine to initialize itself and wait for a user program to be uploaded over the memory mapped UART. Aside from the UART, user I/Os, a PWM controller, and a subtractive synthesizer are also connected and memory mapped to the RISC-V CPU. The user I/Os include the switches, buttons, and LEDs on the FPGA board so user inputs can be read from the memory mapped I/O and the LEDs can be controlled programmatically. The PWM and Synthesizer are designed to read memory mapped values to create varying waveforms in hardware for output to the FPGA's audio port. The synthesizer generates four waveforms: sine, square, triangle and sawtooth. Each of these waveforms is scaled down by a memory mapped register, then added together. The processor can either control the PWM controller itself or hand the control over

to the synthesizer. The frequency of the wave outputted by the synthesizer is set by a memory mapped frequency control word.

The end goal of this project is to implement all of the features above with a stress test cycles per instruction (CPI) of below 1.2 and 100 MHz system clock.

2 High-Level Organization

2.1 RISC-V Core Datapath

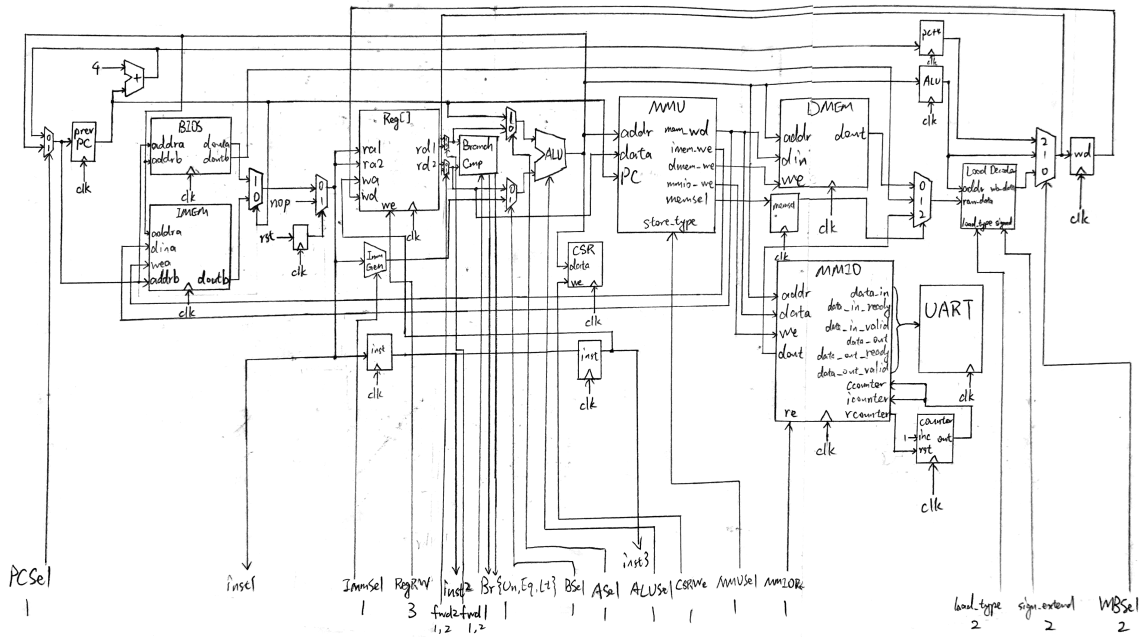

Figure 2: RISC-V Core Datapath Diagram

The datapath is the most integral part of our design. Figure 2 is the high-level diagram of the design. The layout of our datapath allows a three stage pipeline to be implemented with no NOPs inserted for branch and jump instructions (meaning a CPI of exactly 1).

The first stage of our pipeline consists of the program counter (PC), the BIOS memory, and the instruction memory (IMEM). The program counter is sampled from the input of the register directly into the instruction memory. This causes the value of the PC to be one instruction behind. We refer to this value as "prev PC". This layout is intentional to avoid an extra cycle of latency to read the PC. A mux is used based on the PC value to select between the BIOS memory and IMEM.

The second stage of our pipeline consists of the register file, the immediate generator, the branch comparator, the Arithmetic Logic Unit (ALU), the Memory Management Unit (MMU), the Memory Mapped I/O (MMIO), the data memory (DMEM),

the control status register (CSR), and a Universal Asynchronous Receiver/Transmitter (UART) that connects to the host computer. The register file implements the integer registers specified in the RV32I ISA and supports asynchronous reads through two read ports and synchronous writes through the write port. The branch comparator reads the output of the regfile and outputs signals to the control logic in the event of a branch instruction. The ALU takes two operands and supports all the basic arithmetic operations of the RV32I ISA. Aside from the traditional blocks, we have two organizational logic blocks to assist with memory operations. The MMU is one such block. The inputs to the MMU are a memory address, data to be stored, and the PC of the instruction in the stage. The role of the MMU is to use its inputs to format the write data in the proper format into the IMEM, DMEM, and MMIO, with the correct write enable signal set for all these kinds of memories according to the address being written to. It also ensures that only instructions in the BIOS PC range can write to the IMEM. It's final job is to detect the source (IMEM, DMEM or MMIO) of a load operation and pipeline the result into a mux used to select during writeback. The other organizational logic block is the MMIO block. The MMIO block uses the address of a memory operation to select the correct memory mapped peripheral (UART, counters, user I/Os, PWM controller, or the synthesizer) to preform the memory operation on.

The final stage is our write back stage which commits an instruction result to the regfile if needed. Note that due to data hazards, the writeback data needs to be forwarded from the third stage back to the second stage right after the regfile access.

2.2 Peripheral I/Os

2.2.1 User I/Os

As described in the functional description, the user I/Os consist of the switches, the buttons, and the LEDs. The switches and the LEDs are directly connected to the MMIO module because they are persistent signals. However, button presses are transient events, thus we put a 32-entry FIFO in between the button parser and the MMIO module to buffer the button presses so the CPU can poll the button presses later on.

2.2.2 PWM Controller and Subtractive Synthesizer

In order for the CPU to output sound, we designed a PWM controller (which, when combined with the low pass filter on the FPGA, becomes a DAC (Figure 3)) and a subtractive synthesizer. They are also directly connected to the MMIO module. There is also an MMIO flag that controls whether the PWM controller is directly written by the CPU or the synthesizer. Because the PWM controller has its own faster clock, we implemented a 4 phase handshake to synchronize the signal between the CPU and the PWM controller.

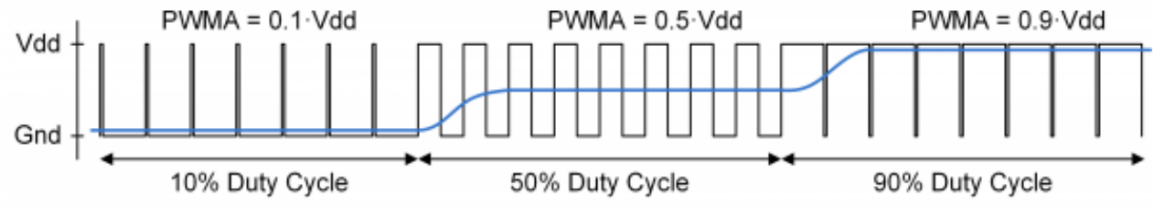

Figure 3: DAC

3 Detailed Description of Sub-Pieces

3.1 RISC-V Core

Note: this section only mentions the modules, which accommodate some additional requirements of the project specification, that are nonexistent or different from the same modules in a regular RISC-V core.

3.1.1 Control Logic

The control logic is really the heart of the CPU. It controls the path of data for each instruction in each stage of the pipeline. Here is a detailed description of what the inputs and outputs are of our control logic:

Inputs:

- inst1: The instruction in the first stage of the pipeline.
- inst2: The instruction in the second stage of the pipeline.
- inst3: The instruction in the third stage of the pipeline.
- BrEq: The branch equal result from the branch comparator.
- BrLt: The branch less-than result from the branch comparator.

Outputs:

- PCSel: Select the source of the next PC, either from the current PC + 4 or from the ALU. It is controlled by inst1, BrEq, and BrLt.
- ImmSel: Select the type of immediate to generate. It is controlled by inst1.
- RegRW: Select whether to write back the data. It is controlled by inst3.
- Fwd2: Select whether the writeback data needs to be forwarded from the third stage to rd2. It is controlled by inst1 and inst2.

- Fwd1: Select whether the writeback data needs to be forwarded from the third stage to rd1. It is controlled by inst1 and inst2.
- BrUn: Select whether the branch instruction is an unsigned compare. It is controlled by inst1.
- BSel: Select whether to pipe rd2 or the immediate to the second input of the ALU. It is controlled by inst1.
- ASel: Select whether to pipe rd1 or the current PC to the first input of the ALU. It is controlled by inst1.
- ALUSel: Select the operation of the ALU. It is controlled by inst1.
- CSRWE: Select whether the current instruction is a write to the CSR register. It is controlled by inst1.
- MMUSel: Select whether the store instruction is a store word, a store halfword, or a store byte. It is controlled by inst1.
- MMIORE: Select whether the current instruction is a load instruction. It is controlled by inst1.
- LoadType: Select whether the load instruction is a load word, a load halfword, or a load byte. It is controlled by inst2.
- SignExtend: Select whether the load instruction requires the result to be sign extended. It is controlled by inst2.
- WBSel: Select whether to write back PC + 4, the ALU result, or the load result to the register file. It is controlled by inst2.

Our control logic has a very clean design. We have wires for each instruction indicating the type of that instruction, and the outputs of the control logic is a combinational logic of these wires, so it's very easy to read.

3.1.2 Register File

The register file is a write-then-read register file, meaning that it will automatically forward the value of a pending write as the current read value if the source and destination registers are detected to be the same (and obviously, not x0, in which case the output should be 0 regardless).

3.1.3 Immediate Generator

The immediate generator differs from a regular ImmGen in that it has an additional logic to generate immediates for csrwi ({27'b0, inst[19:15]}).

3.1.4 Arithmetic Logic Unit (ALU)

The ALU supports the following operations:

- ADD: Output the sum of the 2 inputs.
- SUB: Output the difference of the 2 inputs.
- AND: Output the bitwise AND result of the 2 inputs.
- OR: Output the bitwise OR result of the 2 inputs.
- XOR: Output the bitwise XOR result of the 2 inputs.
- SLL: Output the logical left shift result of the 2 inputs.
- SRL: Output the logical right shift result of the 2 inputs.
- SRA: Output the arithmetic right shift result of the 2 inputs.
- SLT: Output 1 if the first input is less than the second input (signed), and 0 otherwise.
- SLTU: Output 1 if the first input is less than the second input (unsigned), and 0 otherwise.
- A: Output the first input directly (for lui).
- B: Output the second input directly (for CSR instructions).

3.1.5 Memory Management Unit (MMU)

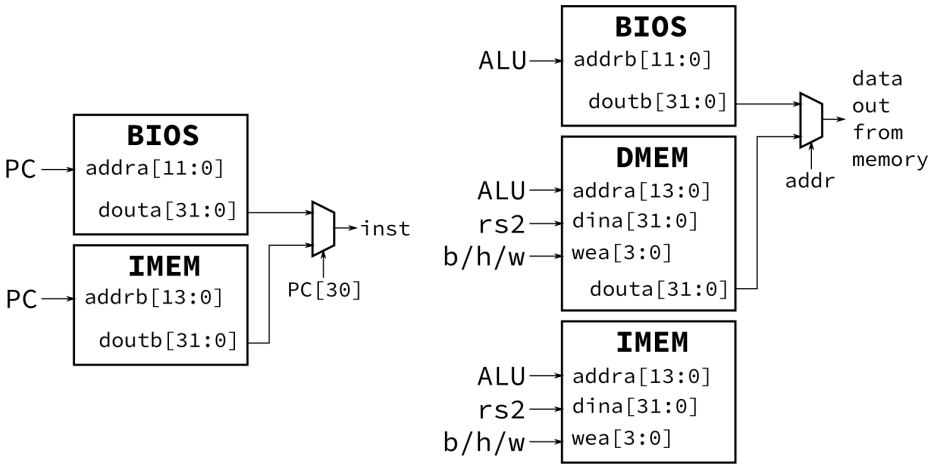

Figure 4: Memory Architecture

Due to the complexity of the memory architecture (the fact that there are 4 types of memories: BIOS, IMEM, DMEM, and MMIO, in which only BIOS is not writable and only IMEM is not readable (Figure 4)), we introduced a standalone module called "Memory Management Unit" (not to be confused with the one that walks the page table) to manage reading from and writing to memories. The MMU, according to the write address and write type, will generate write data and write enable signal that are understood by different types of memories. It will also route the write enable signal to appropriate memories indicted by the write address (taking care of the fact that the code in IMEM cannot write to IMEM). As an additional feature, the MMU generates, according to the read address, a multiplexer select signal that selects which memory is being read from, which is consumed by a multiplexer in the next stage.

3.1.6 Memory Mapped I/O (MMIO)

To create an abstraction of a uniform device interface, we created an MMIO module that has the same interface as other memories (like DMEM) so we can access the devices like accessing memory. The MMIO module connects to all the MMIO devices and exposes to the CPU an interface including address, write enable, read enable (because reading might have some side effects), data in, and data out. The MMIO module will interpret and route, according to the address, read and write requests to devices that are connected to it, including the UART module, the cycle and instruction counter, the user I/Os, the PWM controller, and the subtractive synthesizer.

3.2 PWM Controller and Subtractive Synthesizer

3.2.1 PWM Controller

The PWM controller is a very simple module. It mostly just contains a counter that increments on every PWM clock edge and cycles from 1 to $2^{12} - 1$. The output is pulled high whenever the counter is less than or equal to the duty cycle and pulled low otherwise.

However, because the frequencies of the CPU clock and the PWM clock are different, to make sure that the PWM controller has received the duty cycle, we need to implement a 4-phase handshake circuit (Figure 5) that synchronizes the data request between the CPU and the PWM controller (Figure 6). The request and acknowledge signals are also memory-mapped.

3.2.2 Numerically Controlled Oscillator (NCO)

We implemented the first part of a subtractive synthesizer (Figure 7), called Numerically Controlled Oscillator (NCO). This module can generate 4 kinds of waves: sine, square, triangle, and sawtooth, scale them by some configurable amounts, sum them up, apply a global gain on it, and truncate the result so it can be understood by the

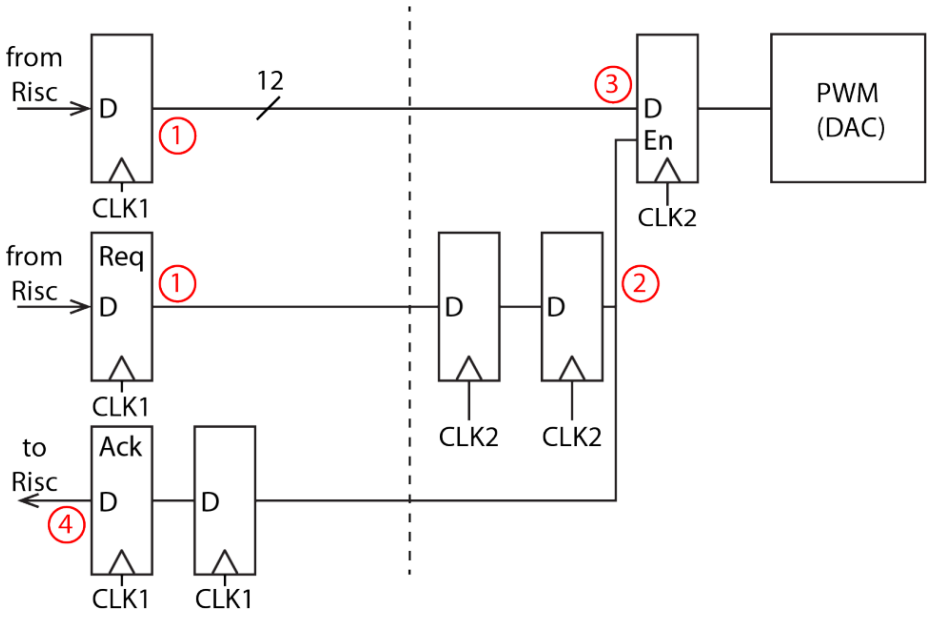

Figure 5: 4-Phase Handshake Circuit

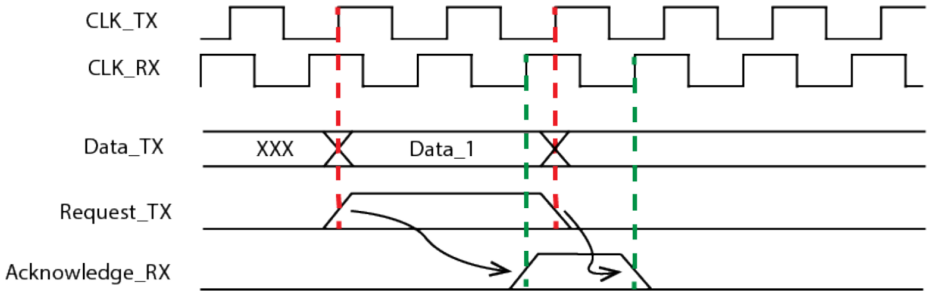

Figure 6: 4-Phase Handshake

PWM controller. The way this module generates waves is by indexing into 4 look-up tables (LUTs) by the current phase. The phase is incremented by a fixed amount (called the frequency control word, coming from the CPU) every time it needs to be (Figure 8). Our phase is a 24-bit integer, but the index is only 8-bit. To increase the accuracy of the NCO, we also perform a fast linear interpolation on the values we get from the LUTs.

3.2.3 Linear Interpolation

The way we do fast linear interpolation can be best understood by code:

```
module interpolation (
    input signed [19:0] a,
    input signed [19:0] b,
```

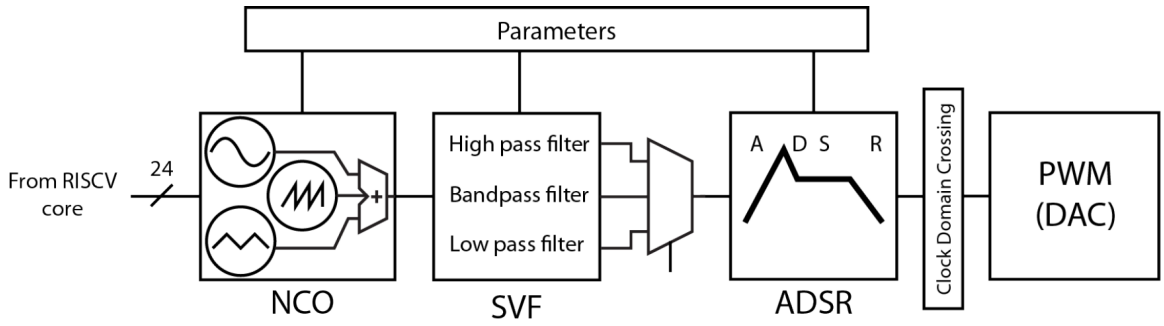

Figure 7: Subtractive Synthesizer

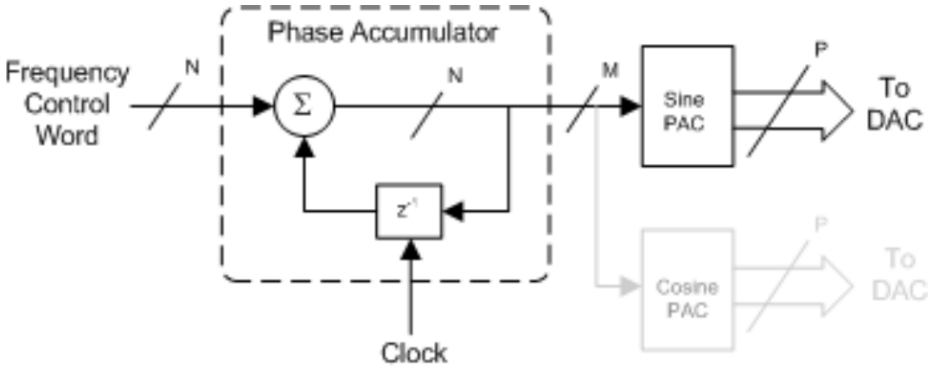

Figure 8: Phase Accumulator

```
    input [15:0] x,
    output signed [19:0] out
);
    reg [3:0] pri;

    integer i;
    always @(*) begin
        pri = 0;
        for (i = 0; i < 16; i = i + 1)
            if (x[i] == 1'b1)
                pri = i;
    end
    assign out = a + ((b - a) >>> (16 - pri));
endmodule
```

In this module, we used a for loop to implement a priority encoder. *pri* will hold the value of $\lfloor \log(x) \rfloor$. Then we perform a rough linear interpolation on the number a and b by using only 1 arithmetic right shift (rather than a complex multiplication, so it's super fast).

3.2.4 Sample Rate Synchronizer

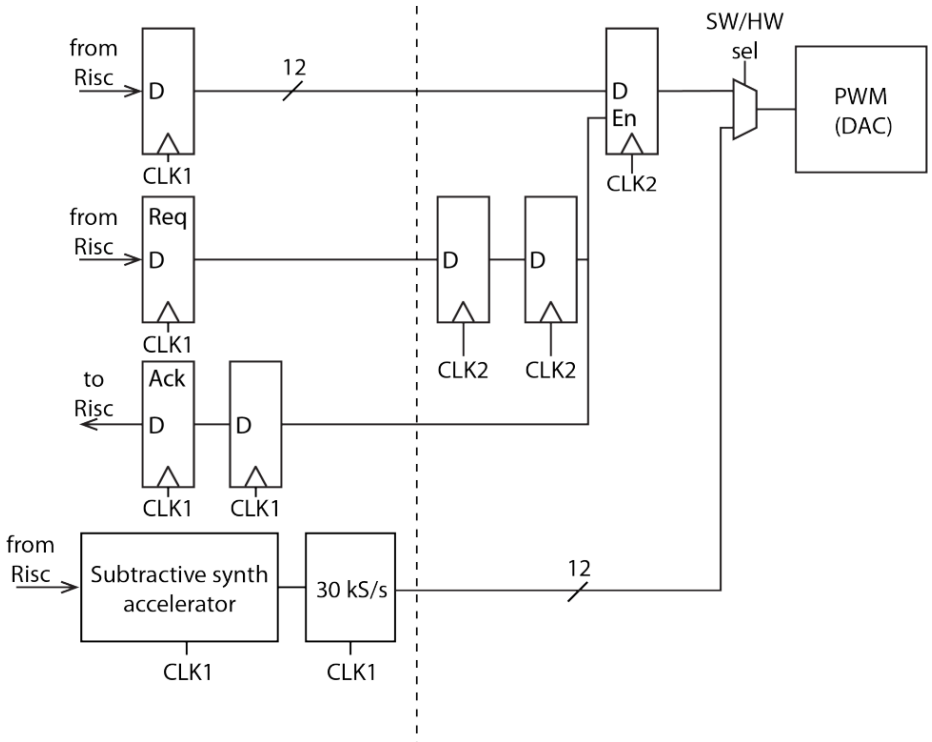

Figure 9: Subtractive Synthesizer Circuit

Now we need to connect the subtractive synthesizer to the PWM controller (Figure 9). Because our synthesizer is assumed to have a sample rate of 30 kHz, we need to output one sample every 1/30000 s. We implemented a very simple synchronizer that tells the NCO to increment the phase accumulator every CPU_CLOCK_FREQ/SAMPLE_RATE cycles. Because the duty cycle will be persistent this many cycles, we don't need to insert a 4-phase handshake between the synchronizer and the PWM controller. Instead, we added 2 registers clocked by the PWM clock to synchronize different clock periods.

4 Status and Results

At the end of this project, we were able to complete our implementation of the three stage RISC-V processor. All logical units outlined above are completely functional and are successfully memory mapped and integrated with the main RISC-V core. The operating frequency of our RISC-V core is 75 MHz (minimum clock period 13.3 ns) with a CPI of 1. We are able to achieve this CPI because of our pipeline layout which has no control hazards on branches or jumps. Our timing constraints report

that the amount of slack in the system is only 0.033 ns, leaving us with very little room to tweak any further timing parameters. All checkpoints operate at the same clock speed as the critical path was set by our pipeline layout set in Checkpoint 1. The utilization of our FPGA is as follows:

Site Type	Used	Fixed	Availible	Util%
Slice LUTs	2071	0	53200	3.89
Slice Registers	542	0	106400	0.51

The full timing report can be found here and utilization report found here.

Our design was fully functional with the only caveat of running at 75 MHz. The trade off in our pipeline layout of having no control hazards during branch and jumps left us with a very large second stage. Because of this design, we were unable to reach the target goal of 100 MHz. The effective throughput of our processor, however, is very comparable. Increasing the operating frequency would require us to re-balance our pipeline in a way that would introduce several control hazards that could not be resolved without introducing NOPs/delays in our pipeline. The target goal of a CPI for mmult of 1.2 would match the throughput of our current implementation running at 80MHz. We did not feel that the improvement of 5 MHz in our clock would justify the much greater complexity introduced by the control hazards from re-balancing our stages. The simpler implementation also allowed us to require less resources on the FPGA leaving more room for future IO expansions.

5 Conclusions

Overall, our project was very successful. We faced some trouble in the beginning identifying our pipeline design. Initially, what we identified as a "3" stage pipeline ended up later being classified as a "2" stage pipeline. This introduced a large hurdle to overcome for checkpoint 2 as we now had to redo work we previously thought we had done. Fortunately, we were able to devise a solution fairly quickly with minimal implementation effort. In hindsight, we should have examined our design more thoroughly to ensure our pipeline stages matched the specification we were supposed to be implementing.

This project has given us a thorough insight into the life cycle of chip design. We were able to briefly touch on the several main stages of layout and architecture, implementation and finally verification. We were able to implement real world IO peripherals that could interface with a functional processor and communicate with other machines across a shared protocol.

Team 06 Division of Labor (Jerry)

As a team, we worked together towards final results. For the first checkpoint, Jakob and I both worked on the design of the pipeline stages. We also verified the correctness of the datapath together. For checkpoint 2, I was responsible for implementing all the sub-pieces in the datapath and connecting them all up, while Jakob worked on the control logic. Jakob also wrote a series of useful tests that eventually caught a bug that is very hard to debug, in our jal instruction. For checkpoint 3, I connected all kinds of peripherals to the MMIO module, and Jakob designed the NCO part of the synthesizer. We both worked on the report together. We both contributed a very significant amount of effort to this project.

Team 06 Division of Labor (Jakob)

We both worked on the initial design of the processor. In the first checkpoint we both worked together to create the pipeline stages. Jerry recreated all our work digitally by drafting the entire pipeline design. In checkpoint 2, I worked an a good chunk of the control logic and helped write tests while Jerry implemented most of the pipeline stages. After he implemented the CPU I identified a bug that took me a while to pinpoint and he was eventually able implement a fix. For the final checkpoint, I implemented the NCO while Jerry implemented and connected all the other IO peripherals. He was also responsible for implementing the 4 way handshakes required for the IO synchronization. We divided up the report fairly evenly. I contributed a fair amount to the project but I feel Jerry ended up contributing more overall.