EECS151LB Final Project Report

Yousef Al-Shinnawi and Eduardo Diaz

University of California - Berkeley
December 2020

Contents

1 Project Functional Description and Design Requirements

The goal of this project was to design and implement a 3-stage pipelined Central Processing Unit (CPU) using the RISC-V ISA programmed in Verilog to run on a Zynq 7000-series FPGA. In addition to having a processor that could run the base RV32 instruction set, there had to also be separate modules in the datapath for handling input/output (IO) to transmit and receive data. An on-chip Universial Asynchronous Reciever-Transmitter (UART) using the technique of Memory Mapped IO (MMIO) was to be implemented to send and receive bytes using RISC-V load and store instructions. This interface would then allow the CPU to store the user's instructions into BIOS memory which could then be read on the following cycle instead of reading from Instruction Memory (IMEM).

The next goal was to utilize the on-board buttons and switches to interact with the CPU. This required a first in, first out (FIFO) buffer that would allow the processor to read user input whenever it had time to do so. The CPU would then be able to drive on-board LEDs and output audio using a Pulse Width Modulation (PWM) controller functioning like a Digital-Analog-Converted (DAC) to determine output signals.

With these designs in place, it was required that the CPU clock speed was beyond 50 MHz whilst minimizing the Cycles Per Instruction (CPI). A fast clock speed was to be achieved using the 3-stage pipeline to divide up work between different stages of the processor. With the addition of pipeline stages, several hazards arose such as control, structural and data hazards. To resolve these conflicts, forwarding between different modules in the CPU was to be implemented, and a scheme to handle control hazards for branching and jumps was also to be considered to minimize wasted cycles.

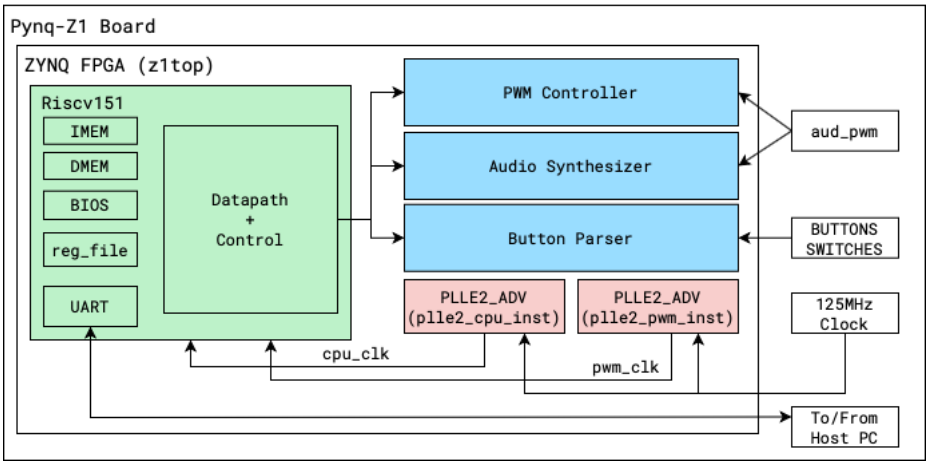

Figure 1: High-Level Overview of System

2 High Level Organization

The processor was divided into 3 sections, the Instruction Fetch Stage (IF Stage), Instruction Decode and Execute Stage (IDEX Stage), and Memory Write Back Stage (MEMWB Stage). In the IF Stage, new instructions are loaded from either the IMEM or BIOS which are then decoded and passed through the Register File (RegFile) to load data for the IDEX Stage. The IDEX Stage then passes this data through combinational submodules such as the Arithmetic Logic Unit (ALU) or Branch Comparator to compute the logic of the current instruction. Output from this stage is saved to the IMEM and Data Memory (DMEM) which is then passed through to the MEMWB Stage. Following this, the MEMWB Stage handles what data is to be written back to the RegFile for the current instruction. A more detailed explanation of each stage is described on the following pages.

2.1 Instruction Fetch Stage

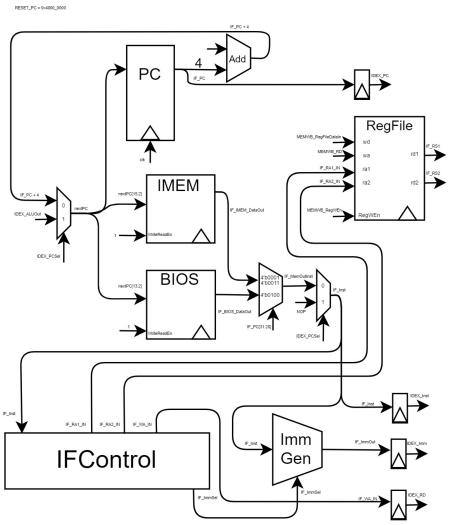

Figure 2: IF Stage

To fetch new instructions, a counter (PC) that begins at 0x40000000 and increments by 0x4 every cycle is used to address into the IMEM and BIOS. For jump and branch instructions, the output of the ALU determines the next PC value by using PCSel. Once the next instruction is loaded from the IMEM and BIOS, the upper 4 bits of the PC counter determine which memory to select from. If the next instruction is a jump or branch, a single NOP is inserted.

After we have our new instruction, it is passed through the Immediate Generator (ImmGen) and IF Control Unit. The ImmGen produces a 32bit immediate, and the IF Controller decodes the instruction to fetch the addresses needed to input into the synchronous RegFile acting as a register between the IF and IDEX Stages. Additionally, the PC counter, instruction, immediate, and write address for the RegFile are all pipelined to the next stage. The write address is pipelined because data is only written to the RegFile during the MEMWB Stage.

2.2 Instruction Decode and Execute Stage

With all pipelined signals from the IF stage, the IDEX Stage then passes the current instruction into the IDEX Control Unit to determine many control signals. IDEX Control outputs are used for selecting: the ALU operation, ALU inputs, data forwarded from previous instructions, what data to write back in the MEMWB Stage, creating the masks for data written to DMEM and IMEM, the branch type for the Branch Comparator, and if data should be written to the RegFile or Control and Status Register (CSR). By using these control signals, the other combinational blocks such as the ALU and Branch Comparator are able to execute the operations defined by the RISC-V ISA.

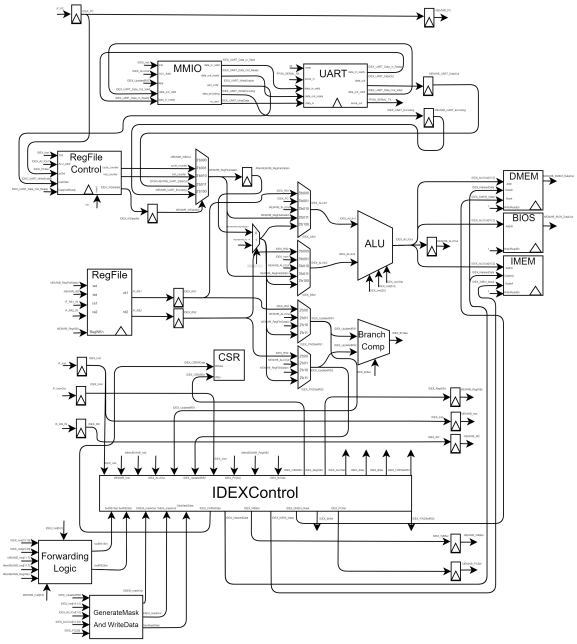

Figure 3: IDEX Stage

Additionally, in this stage, the UART and MMIO blocks communicate to output data to the RegFile Control Block which then determines what data should be written to the RegFile during the MEMWB Stage. This is precisely how the CPU is able to interact with the user by writing data to registers that are expected. For instance, the current cycle count or instruction count is relayed to the user through this control, or whether the Synthesizer block is finished playing a note. Most of the processor's logic is conducted during this stage, and outputs are saved to the IMEM and DMEM which act as a pipeline for the next stage. The following signals are also pipelined to the next stage: PC counter, UART Data, MMIO Data Encoding, ALU

Output, RegFile write enable, the current instruction, RegFile write address, RegFile write back select, PCSel, and which data to write to the RegFile.

2.3 Memory Write-back Stage

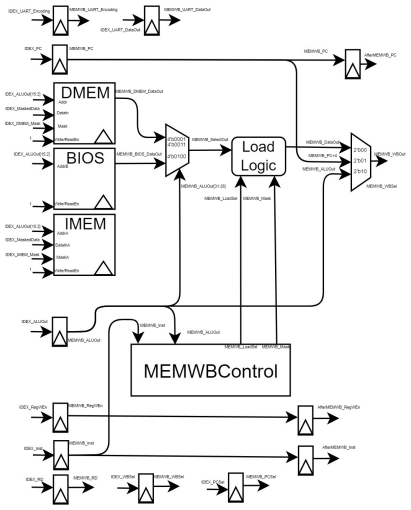

Figure 4: MEMWB Stage forwarding.

The MEMWB Stage begins by using the upper 4 bits of the pipelined ALU output data to choose between the output of the BIOS and DMEM memories to feed into the Load Logic block. At the same time, the ALU output and current instruction are also used to feed into the MEMWB Control Block to determine the Load Logic block's mask and select signals. The Load Logic block is used specifically for LW, LH, and LB and the load select signal chooses between the three while the mask select is used to choose the desired bytes of data to output.

This formatted data is used as output for the write back muliplexer (mux), and this mux outputs either the data from memory, ALU or PC+4 based on the pipelined WBSel signal from the IDEX Stage. Following that, the data is then sent to the mux that chooses what data writes to the RegFile. This stage also pipelines a few signals, namely: the PC counter, RegFile write enable, and the current instruction. Some of these signals are also pipelined into the AfterMEMWB stage which are used specifically to resolve data and control hazards via

3 Detailed Description of Sub-pieces

The most important sub-circuits in this design were the control unit blocks, namely the IDEX Control Unit and RegFile Control Unit. These blocks did most of the heavy lifting in terms of providing control signals for all other modules. In the following sections we will take a closer look at each of these modules in addition to the PWM Controller and Subtractive Synthesizer.

3.1 IDEX Control Unit

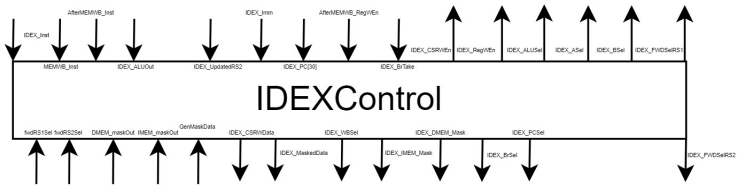

Figure 5: IDEX Control Unit

As can be seen in the figure above, the IDEX Control Unit takes in a lot of signals and also outputs a lot to many modules. The reason for all these signals is because once an instruction reaches the IDEX Stage, a lot of the logic can be computed or pre-computed for the MEMWB Stage. The process of computing these signals was relatively simple. For instance, an I-Type instruction would output high for RegFile write enable, use its funct3 for ALUSel, and most of the other signals would output low because of their associations with other instruction types. In the case of an I-Type instruction, PCSel would always be low because it is not a jump or branch type, and BrSel would also not matter because it's not a branch type.

The most important signals coming out of the IDEX Control Unit are the signals that deal with forwarding, these signals are specifically ASel, BSel, FWDSelRS1, and FWDSelRS2. ASel and

BSel help provide forwarded data to the ALU, and FWDSelRS1 and FWDSelRS2 update the data coming out of the RegFile during the IDEX Stage so that it can be properly used in the Branch Comparator, MMIO, and for writing data to the DMEM and IMEM. The Forwarding Logic Unit explained further in the next subsection is used to compute these signals.

In addition to forwarding, this control unit is also responsible for producing the mask and data for the DMEM and IMEM to use when storing data into memory. This process is explained in the Generate Mask and Data subsection.

3.1.1 Forwarding Logic

To determine what data should be forwarded to the IDEX Stage, this logic unit uses the current RegFile input addresses RS1 and RS2, the previous RegFile return address prevRd, and the RegFile return address from 2 cycles ago, prevPrevRd, as well as the RegFile write enable from 2 cycles ago, prevPrevRegWEn. If prevRd and RS1 are equal, then we know that the data from the previous cycle should be used in the computation of the current instruction.

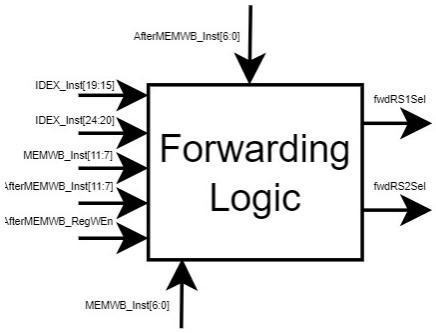

Figure 6: Forwarding Unit

The same logic is used in relating RS2 to prevRd, and prevPrevRd and prevPrevRegWEn are similarly used with RS1 and RS2 to determine forwarding from 2 cycles ago. Additionally, the previous opcode and opcode from 2 cycles ago are used to handle special cases with I-Type load and SB-Type Branch instructions. By using all of this information, the Forwarding Logic Unit is able to produce a 2-bit signal that informs the IDEX Control Unit what data is to be forwarded to update the data of RS1 and RS2. An output of 2'b00 = no forwarding, 2'b01 = forward ALU Output, 2'b10 = forward MEMWB Output, and 2'b11 = forward AfterMEMWB Output.

3.1.2 Generate Mask and Data

This control block is used exclusively for the S-Type store instructions using the forwarded RS2 data, funct3 of the instruction for determining the type of store, SW, SH, or SB, the last 2 bits of the ALU's output for the memory offset, the upper 4 bits of the ALU's output to select what memory to write to, and the 30th bit of the PC counter for writing to the IMEM. By using these signals, this block is able to determine the masks for the DMEM and IMEM and what data to write.

Figure 7: Gen Mask and Data Module

The mask and output data of this block are used for writing specific bytes of data to the memory blocks. For instance, if the current instruction is SB with offset = 2'b00, ALUOut[31:28] = 4'b0010 and PC[30] = 1'b1, then maskIMEM = 4'b0001, maskDMEM = 4'b0000, and the outputData = 24'b0, fwdRS2[7:0]. By supplying this information to the DMEM and IMEM, the DMEM would write nothing to its memory and the IMEM would only write the first byte of data fwdRS2[7:0] to the address supplied by ALU's output, ALUOut[15:2].

3.2 RegFile Control Unit

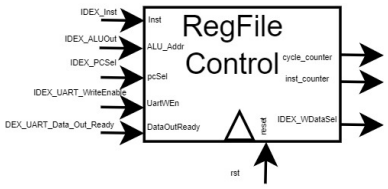

Figure 8: RegFile Control

A great deal of the IO interface is handled by this control unit. Specifically, this block determines what data should be written to the RegFile, whether it is data from the GPIO interface, PWM block, Synthesizer, or to simply control the CPU's cycle/instruction counters. This block also counts the number of cycles/instructions the processor has had in addition to supplying the GPIO FIFO with a read enable signal.

3.3 PWM Controller (DAC)

As mentioned in the Introduction, the PWM Controller was designed to function like a DAC (Figure 10). The input is a 12-bit number that determines the duty cycle of the signal that the PWM will output (Figure 9). This made the implementation pretty straightforward, it turned out to be a counter that initially starts at 1 and is incremented at every positive edge of the PWM clock. If the counter is less than or equal to the 12-bit input, then the output is set high otherwise, the output is set low.

$$\text{Duty Cycle} = \frac{\text{Input}[11:0]}{2^{12} - 1}$$

Figure 9: Duty Cycle Calculation

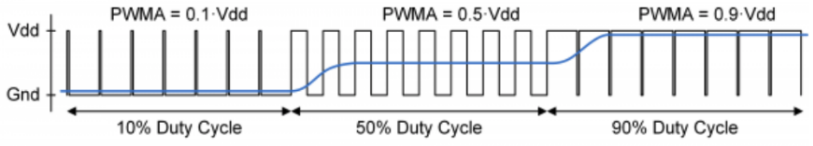

Figure 10: Duty Cycle Output

The only issue with the PWM Controller was that it was running inside of a different clock domain, much faster than that of the CPU. To ensure a reliable transfer of the 12-bit data from the CPU's clock domain to the PWM Controller's clock domain, a 4-phase handshake was implemented (Figure 11 and 12). This helped synchronize the request for data between the CPU and PWM Controller.

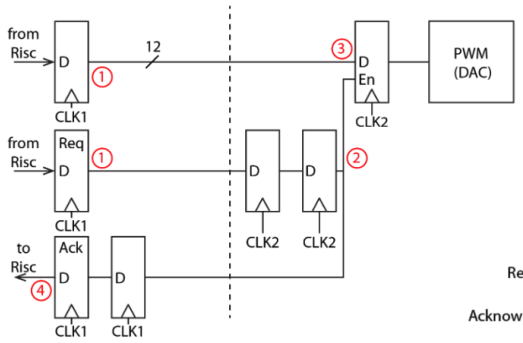

Figure 11: 4-Phase Handshake Diagram

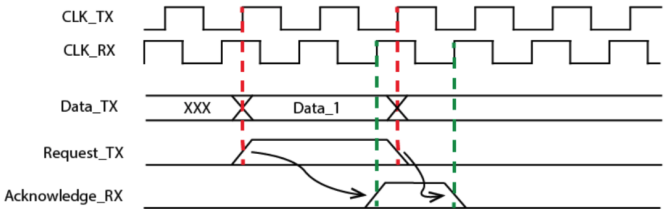

Figure 12: 4-Phase Handshake Timing

3.4 Subtractive Synthesizer

To play different tones using the on-board audio port with the CPU, hardware acceleration was required. In the case of this design, a Subtractive Synthesizer was implemented. The first part of implementing the Subtractive Synthesizer (Figure 13) was the Numerically Controlled Oscillator (NCO) (Figure 14).

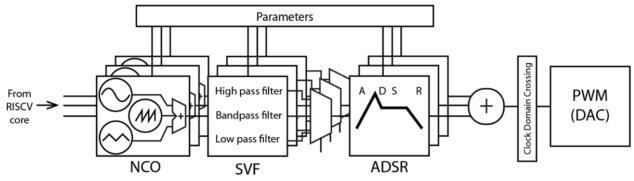

Figure 13: Subtractive Synthesizer

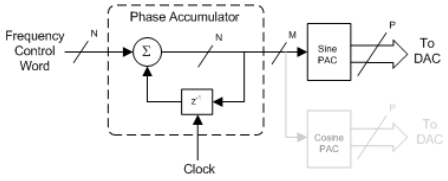

Figure 14: Generic NCO

The NCO module was able to generate four waveform types: sine, square, triangle, and sawtooth. This was possible by using several Lookup Tables (LUTs) one for each type of wave and a 24-bit phase accumulator that was incremented by the phase increment (Frequency Control Word). The LUTs contained the sample points for one period of their respective waveform in fixed point representation (16 fractional bits, 4 integer bits inclusive of 1 sign bit). The 8-MSBs of the phase accumulator helped to compute the next index of the LUTs, and the 16 remaining bits were used as the residual error when linearly interpolating.

Linear interpolation was tricky to implement because the LUT values were in fixed-point representation so it is best understood as code:

```
module interpolation(
        input signed [23:0] a,
        input signed [23:0] b,
        input [23:0] residual,
        output signed [23:0] out
);

reg signed [23:0] diff;
reg signed [23:0] total;
reg signed [47:0] mult;
always @(*) begin
        diff = b - a;
        mult = $signed(residual) * diff;
        total = (a + {{8{mult[31], mult[31:16]}});
end
assign out = total;
```

Linear interpolation was used on the LUT values to increase the accuracy of our NCO. The interpolated value was calculated using the values of the LUT[i] and LUT[i + 1], the value of LUT[i] and LUT[i + 1] can be thought of as the inputs a and b respectively. The input residual is the 16-remaining bits of the phase accumulator previously mentioned as the error term. The diff register held the difference between b and a. The mult register was 48 bits to hold a value large enough for the product of the residual and diff registers. Once the multiplication was done, the middle 16 bits of mult were split and sign-extended using the 31st bit. Once the interpolation was done the result was then scaled by an arithmetic right shift, and the same was done for all four wave types. All these results were then summed, and a global gain was applied using an arithmetic right shift. This result was truncated to 12-bits and finally unsigned. This way the sample value could be understood by the PWM Controller (DAC) as a duty cycle.

The synthesizer was only allowed to send samples to our PWM DAC at a sample rate of 30KHz. To accomplish this the phase accumulator was instead incremented every CPUCLOCKFREQUENCY/SAMPLINGRATE. With this buffer in the place, the data was then sent off using the 4-phase handshake to synchronize the CPU and PWM Controller clock domains (Figure 15).

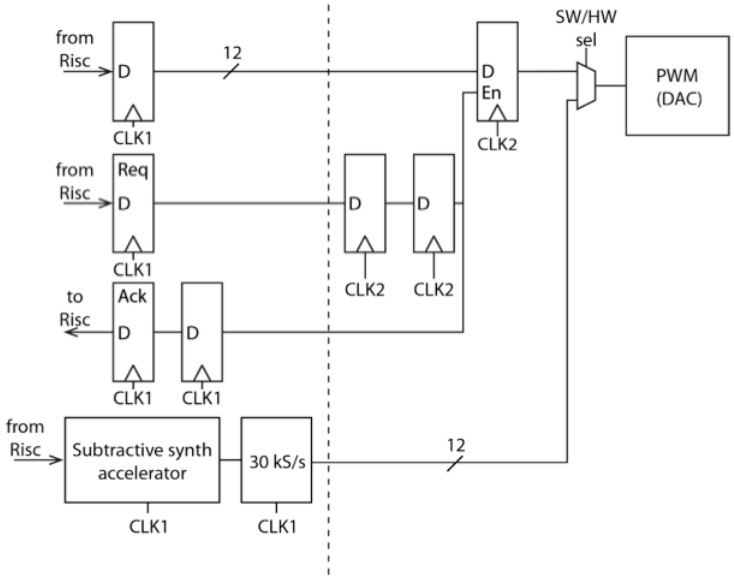

Figure 15: Synchronized Subtractive Synthesizer

4 Status and Results

In terms of the original goal, the CPU checked all the marks. The entire datapath was implemented with a 3-stage pipeline that could send and receive data through the UART MMIO interface. Through this interface, the CPU was able to interact with on-board buttons, switches, LEDS, and the audio output with our Subtractive Synthesizer (Figure 16).

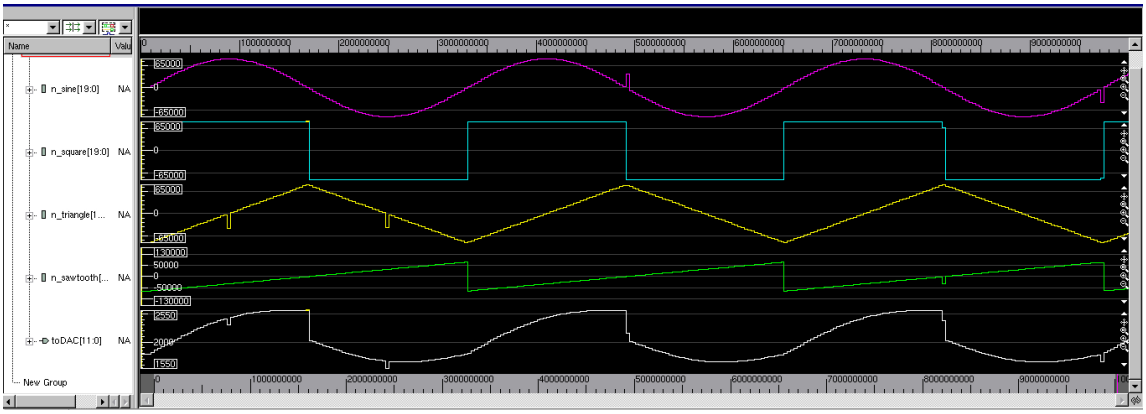

Figure 16: Subtractive Synthesizer Output

By the final checkpoint, the processor was able to reach a clock speed of 70MHz. The max path delay turned out to be in the IDEX Stage of the 3-stage pipeline. Specifically the delay was between the initial instruction input to the IDEX Stage to the Synthesizer block at the end of this stage. It is speculated that this delay occurs because of the need for signed multiplication and buffer that exists in the PWM block for delivering audio to the on-board audio port. Multiplication is a very intensive process and the current method used in this design is not optimal.

Additionally, it is important to mention that before implementing the NCO, PWM and Synthesizer, the critical path instead existed between the MEMWB stage and IDEX stage during forwarding. In the previous design we had reached a final clock speed of 76MHz. It is believed that this critical path existed because of the need for the RegFile return address and write enable from 2 cycles ago.

For both designs, the CPI of the mmult program was 1.18 most likely due to the design's scheme of handling jumps and branch instructions by simply adding a NOP to the next cycle. The minimum clock period for the final design was 0.214s for the mmult program, the number of Slice LUTs used was 2,283, and the number of Slice Registers used was 905.

Site Type	**Used**	**Fixed**	**Available**	**Util%**
Slice LUTs	2210	0	53200	4.15
LUT as Logic	2162	0	53200	4.06
LUT as Memory	48	0	17400	0.28
LUT as Distributed RAM	48	0		
LUT as Shift Register	0	0		
Slice Registers	905	0	106400	0.85
Register as Flip Flop	867	0	106400	0.81
Register as Latch	38	0	106400	0.04
F7 Muxes	17	0	26600	0.06
F8 Muxes	0	0	13300	0.00

5 Conclusion

In the end, implementing the 3-stage pipelined CPU with integrated I/O helped to better understand the processes that go into chip design, meeting timing constraints, synchronizing data between two different clock domains, and how to better handle hardware/software integration. The project was successful and all tasks were completed. However, there were some improvements in the design that could have made the CPU function better. There are several optimizations that could improve CPI and clock speed.

By identifying that the critical path was in the IDEX stage, rebalancing the CPU's stages to better distribute computations that took place in the IDEX stage could have been looked into. Especially, after adding the Subtractive Synthesizer since the multiplication that was used for interpolation was an expensive task. For instance, if the RegFile was instead in the IF Stage, this could decrease the load on the IDEX Stage and also allow for branching/jumping to be computed in the first stage. In this way, there would be no need for a NOP after a jump or branch instruction. This would increase CPI and also possibly make the processor's clock speed faster because of the stage rebalance.