John Brittain

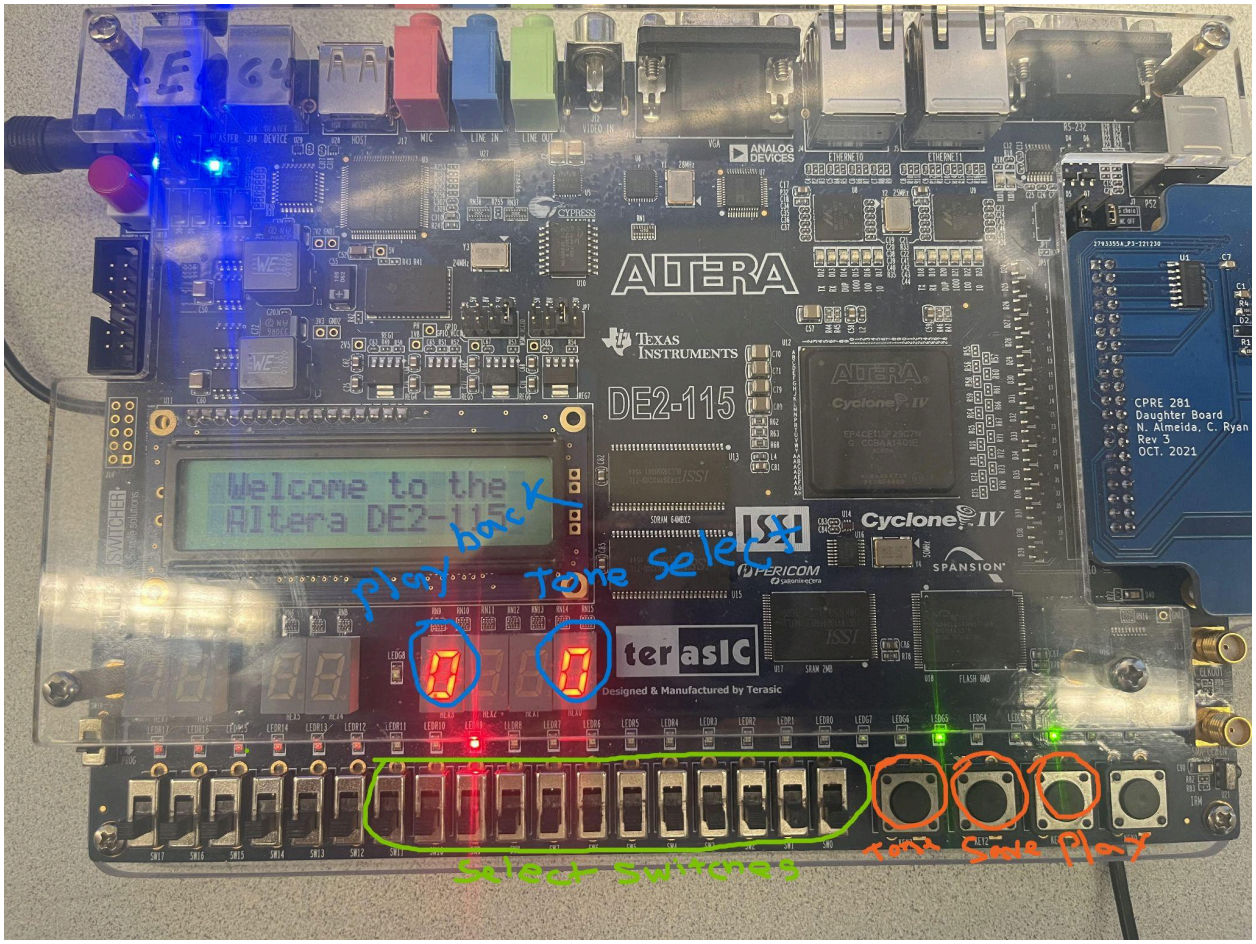
4 / 30 / 2024

CPRE 281

Student ID:

Final Project Report

For this project, I decided to create a 12 step digital sequencer. Which is a musical device that sequences through beats and plays set tones back to the user in that sequence. The user can save a note to each beat that will then will be played back when the sequencer arrives at that beat's position. To achieve this I used many different types of common digital logic devices throughout this project such as; clock dividers, button toggles, MUXes, Registers, decoders etc. This report begins with the Top Level Diagram overviewing the general structure then later on, focuses on specific sections of the project. *Controls and FSM Diagram, Data Processing and Register File, Outputs,* and *Final Thoughts.*



Above is the FPGA while running the device, play back shows the note which is currently saves at each beat, ***tone*** *select shows the note you want to save onto each beat, and the* ***tone*** *button cycles through 0 - 12 corresponding to the tone, holding down* ***save****, saves a tone to the* ***switches*** *you have engaged, and* ***play*** *starts and stops the sequencer.*

Top Level Diagram

The diagram above is the complete schematic of the device, there is quite a bit to unpack here so I'll keep this description very general for describing the function of each section. On the left hand side you will see three input nodes, these are the buttons which are used to control the device. The top, is the **PlayStopbutton** which toggles the sequencer to begin. Below that is the **SaveStatebutton**, which is anded with each of the twelve switches and effectively acts as an enable for the registers to save the 4 bit tone data into the registers. Below that button is the **toneselectbutton**, which counts from 0 - 12 to supply tone data into the registers. 0 being no note played, and 1 - 12 being the notes in a standard musical C scale.

To the right of the buttons, you will see a large portion of the device are input nodes which correspond to twelve switches on the FPGA, these switches toggle which beat of the **sequencer** to save the tone data too. That way you can control which note you would like to play on a beat.

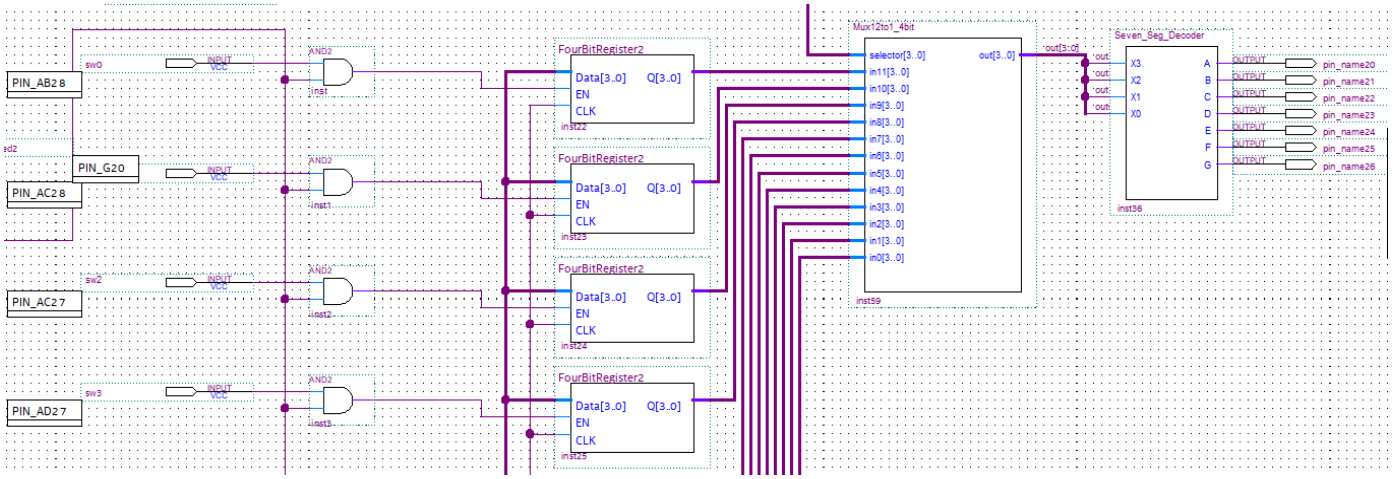
Above you have a button toggle feeding into a **180bpmclockdivider** which slows down the native clock frequency of 50MHz to 3 hz or 180 beats per minute. That signal then is connected to a sequencer, which is essentially a counter that repeats from 1 - 12 cycling through each beat. From the sequencer is splits off into a LED portion that lights up which beat it's currently on and a selector node in the **4bit12to1MUX**. The MUX then outputs the tone value corresponding to the beat that the sequencer is currently on. The MUXes inputs are each of the 4 bit registers of which there are twelve.

Below the switches you will find the source clock of 50MHz, the tone select button which connects into a **debouncer**, then into an **incrementcounter** from 0 - 12, from there it feeds a **4 bit binary number data bus** into each of the twelve **4bitregisters** gated by the switch and save

state controls. The signal also splits from the increment counter into a **seven segment display**, which displays to the user which tone they are currently set to save.

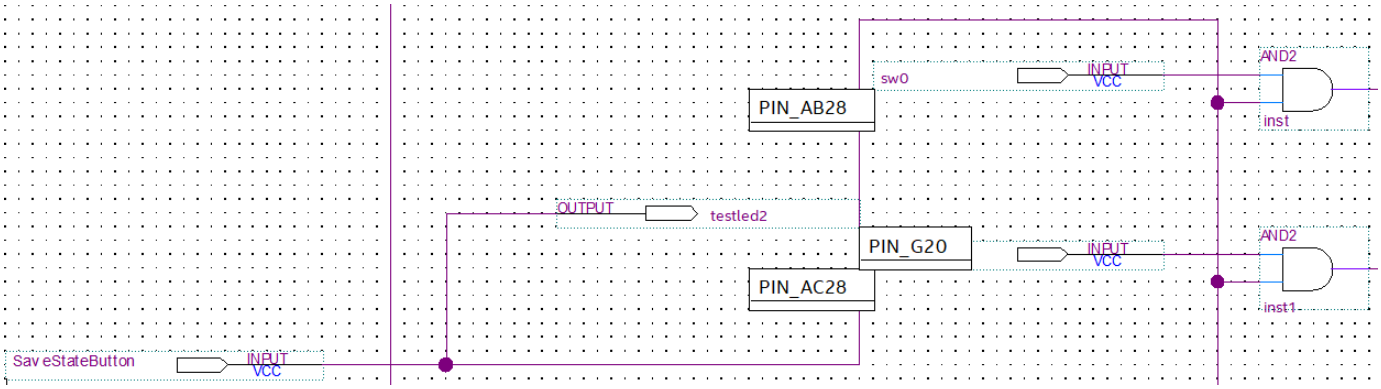
Controls and FSM Diagram

Toggle Switches

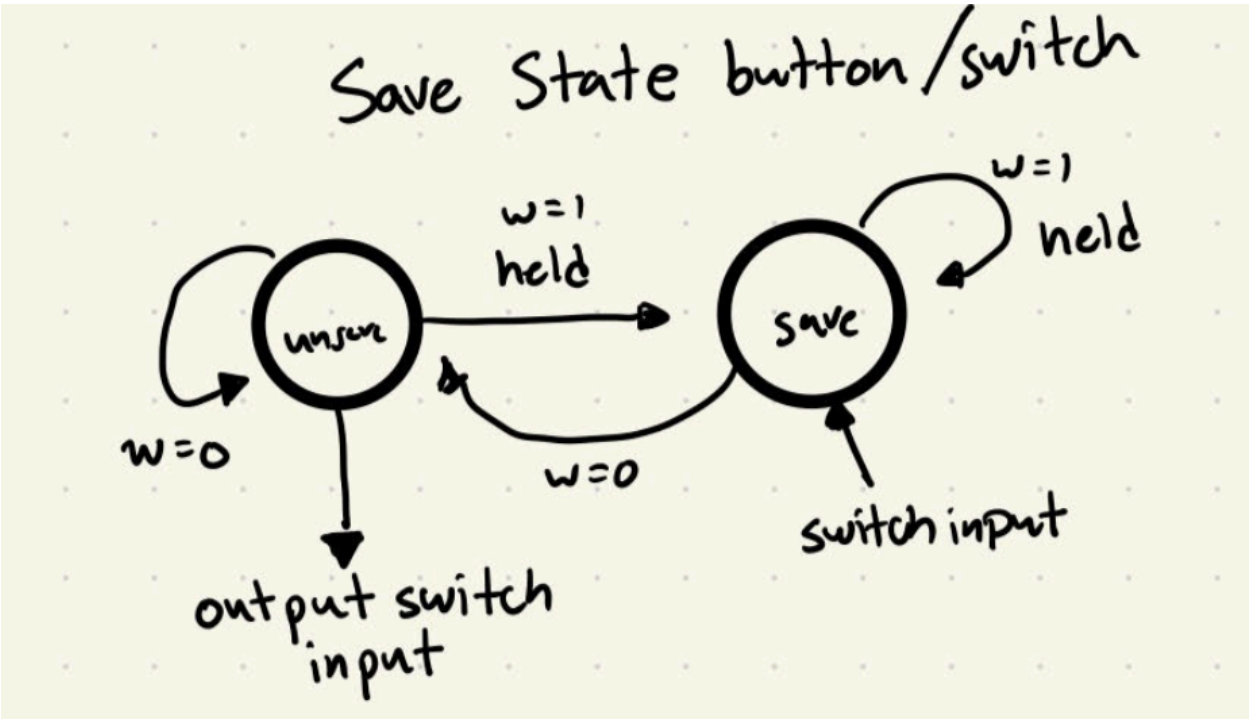


The toggle switches play an essential role in saving the tone value into a **4bitregister**. (Note only 4 of the 12 total switches are shown). These switches correspond the **SW0 - SW11** switches on the FPGA. Each switch is wired into an and gate with the **SavestateButton** and then directly wired in the **EN** input of a register. When a switch is toggled, a logic of 1 enters a gate anded with the **SaveStatebutton**, and when both of those are 1 the **EN** input of the corresponding **4bitregister** is 1, loading the registers with the tone data. If the switch is not toggled, it outputs a 0 and does not satisfy the conditions of the and gate, which then outputs a 0 into the **EN** inputs of a register, the register then holds whatever value that has been stored before.

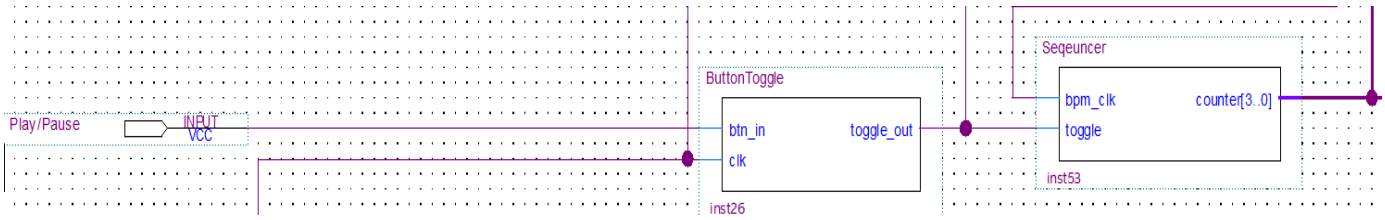
Save/State Button



The **SaveStatebutton** which is set to **KEY2** on the FPGA works in tandem with the switches to ensure that tone data is not saved into one of the **4bitregisters** without the user enabling it. To do this the **SaveStateButton** is wired into twelve and gates along with each switch. Gating the output, and enabling/disabling the registers.



Play/Pause Button



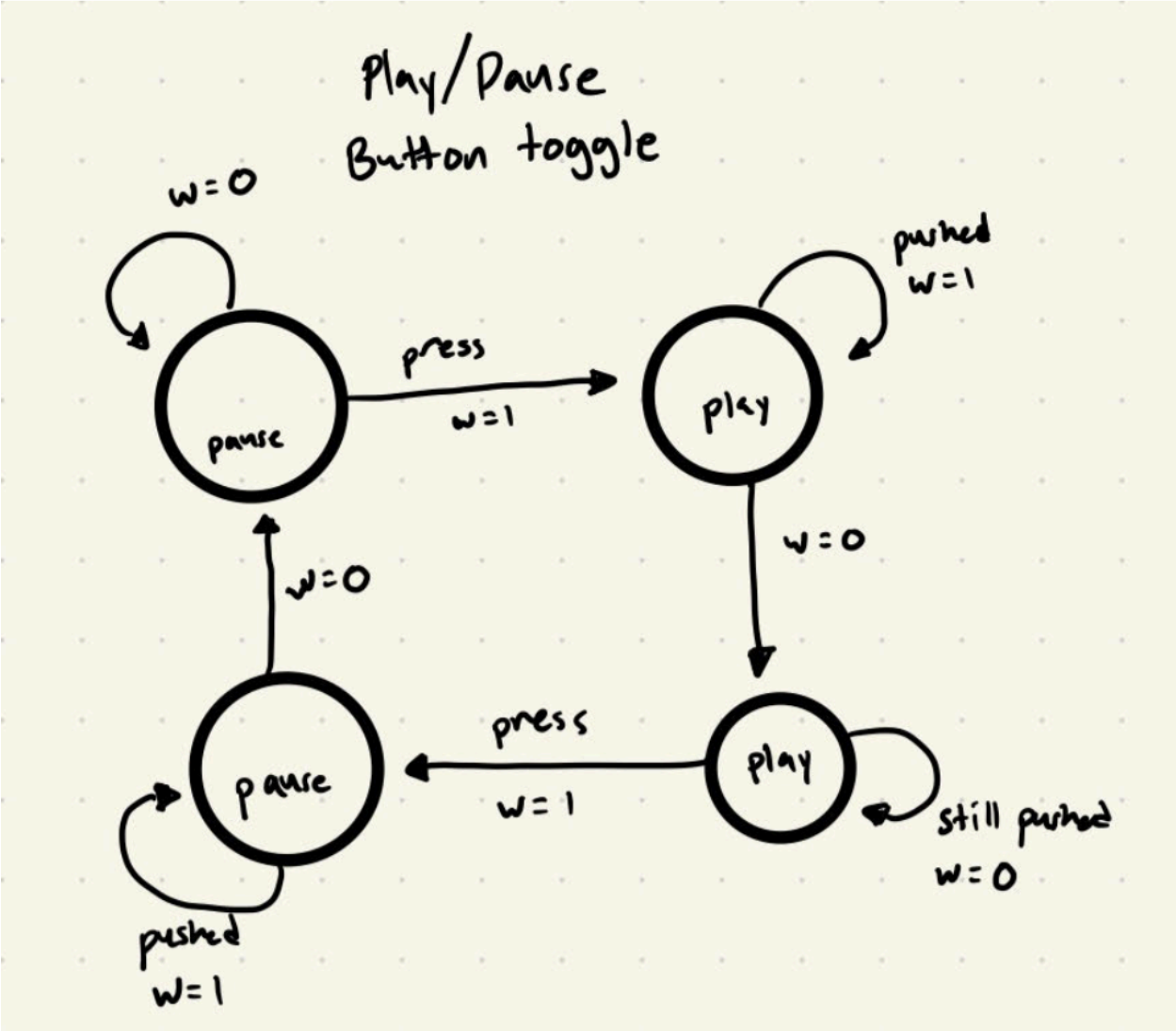
The **PlayPause** button **KEY1** on the FPGA is pressed and toggles the sequencer on and off, allowing user-control of the sequencer. To do this, an input is wired into **ButtonToggle** and then into the **toggle** input of **Sequencer**.

```
module ButtonToggle(
    input btn_in,
    input clk,
    output reg toggle_out = 0
);

reg btn_in_prev = 0;

always @(posedge clk) begin
    btn_in_prev <= btn_in;

    if (btn_in_prev == 0 && btn_in == 1) begin
        toggle_out <= ~toggle_out;
    end
end

endmodule
```

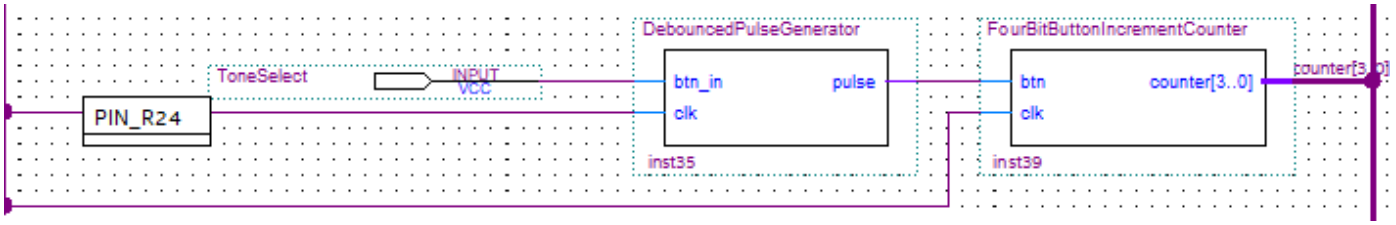
The **KEY1** button(and all the other buttons like it) does not toggle between 0 or 1 natively, it only outputs a 1 for the duration the button is held down. In order to add a functioning toggling button I created code.

Above is the verilog code for ButtonToggle, there are two inputs; **btn_in** and **clk**, and one output **toggle_out** (set default of 0). On each positive edge of the **clk** the previous state of the button **btn_in_prev** is set less than or **equal to btn_in**, limiting **btn_in_prev** to just 0 or 1 values. And saving the state of the previous button value.

If **btn_in_prev** equals 0 and **btn_in** equals 1 (the button is pressed) the output signal **toggle_out** is set less than or equal to the inverse of toggle_out, effectively flipping between 0 or 1. This code creates a functional toggling button that holds a value until the button is pressed and the output flips again.



Tone Select Button



The **ToneSelect KEY3** on the FPGA outputs a pulse signal that increments a 4bit binary number between 0 - 12. This allows the user to select which tone to save to one of the registers.

To do this I have an input **ToneSelect** wired into the **btn_in** input of **DebouncedPulseGenerator** which ensures that when the button is pressed only a short output pulse is generated for one clock cycle, without this a single press of the buttons could increment the counter very quickly and uncontrollably.

The pulse that is generated from the **DebouncedPulseGenerator** is then wired into the input **btn** of **FourBitButtonIncrementCounter.** The function of which counts up from 0 to 12 then resets back to 0(no note). Because there are 12 notes in a scale the button is only set to count up to 12, yet that still requires 4 bits of data to represent. Hence the entire device, is designed to handle mostly 4 bit binary numbers.

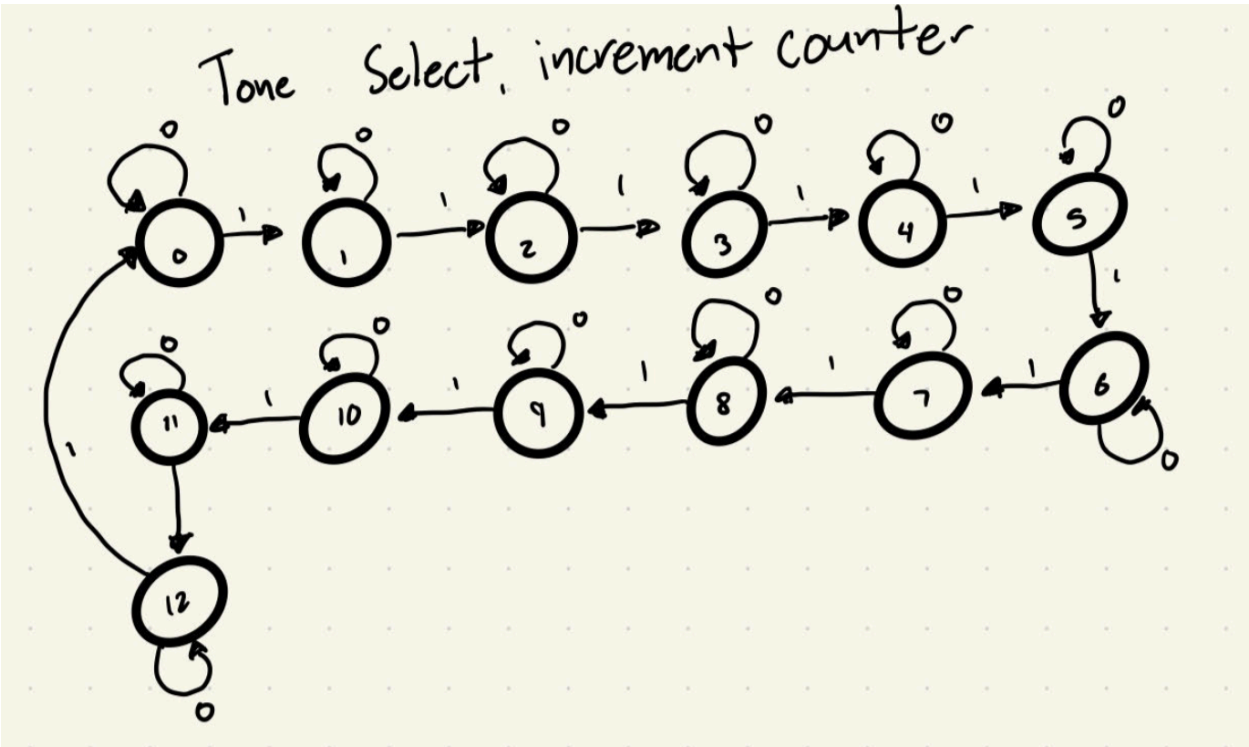
```
module FourBitButtonIncrementCounter(
    input btn,
    input clk,
    output reg [3:0] counter = 0
);

reg btn_prev = 0;

always @(posedge clk) begin

    btn_prev <= btn;

    if (!btn_prev && btn) begin

        if (counter == 12) begin
            counter <= 0;
        end else begin
            counter <= counter + 1;
        end
    end
end

endmodule
```

*Above is the verilog code for **FourBitIncrementCounter.*** It counts two inputs **btn** and **clk**, and one output bus **counter**.

On each positive edge of the clock **btn_prev** is set equal to **btn** and stores the previous button press, this helps detect when the button is pressed.

A group of nested if statements which first detects if **btn_prev** is equal to 0 and **btn** is equal to 1 to ensure the counter is only incremented with a button press.

Then the reset logic if **counter** already equals 12 then it gets reset to 0 otherwise **counter** increments by 1 decimal value.



Data Processing and Register File

Clock Divider

```
module ClockDivider180BPM(
    input sys_clk,
    output reg bpm_clk = 0
);

localparam HALF_PERIOD_COUNT = 8333333;

reg [22:0] counter = 0;

always @(posedge sys_clk) begin
    if (counter >= HALF_PERIOD_COUNT - 1) begin
        bpm_clk <= ~bpm_clk;
        counter <= 0;
    end else begin
        counter <= counter + 1;
    end
end

endmodule
```

In music rhythm is very important, typically rhythm follows a tempo, tempo is measured as beats per minute. For this device I have designed the tempo to be 180 beats per minute or bpm. To do this I created a clock divider that would take the **clk** signal of 50MHz and outputs a much much slower signal of 3hz (180 bpm). This would serve as the tempo of the **sequencer.**

Above is the verilog code for **ClockDivider180BPM**, there is only one input **sys_clk** which is the 50MHz signal of the FPGA clock. The output **bpm_clk** outputs 3hz.

I create a local parameter **HALF_PERIOD_COUNT** which is the value of the desired period divided by 2. I use the half period because I want the rising edge of the signal to match 180 bpm.

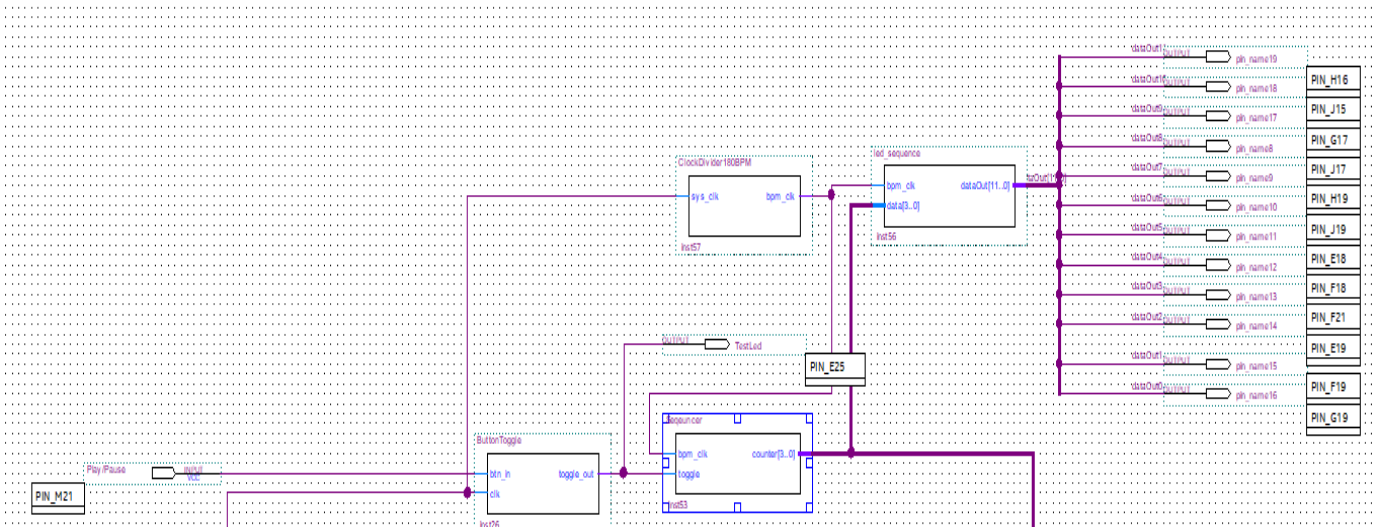
For each positive of the clock a register **counter** increments to keep track of how many clock cycles have occurred.

If the counter is less than **HALF_PERIOD_COUNT** - 1 a full period of 3hz has occurred, therefore **bpm_clk** is inverted, which creates a square wave pattern, and the **counter** is reset to 0.

Otherwise the **counter** will increment with each clock cycle.

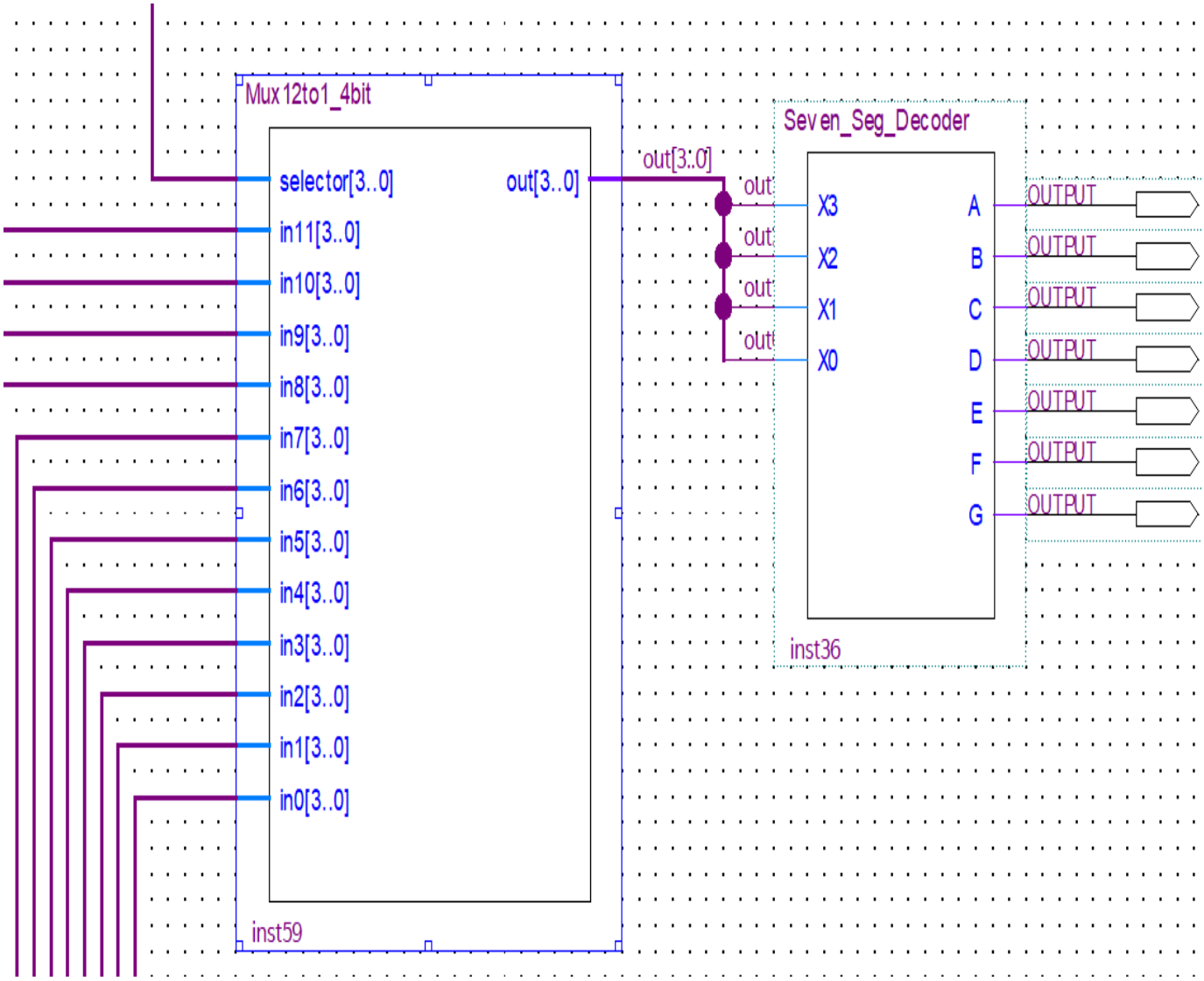
This code creates a square wave of 3hz or 180bpm.

Sequencer



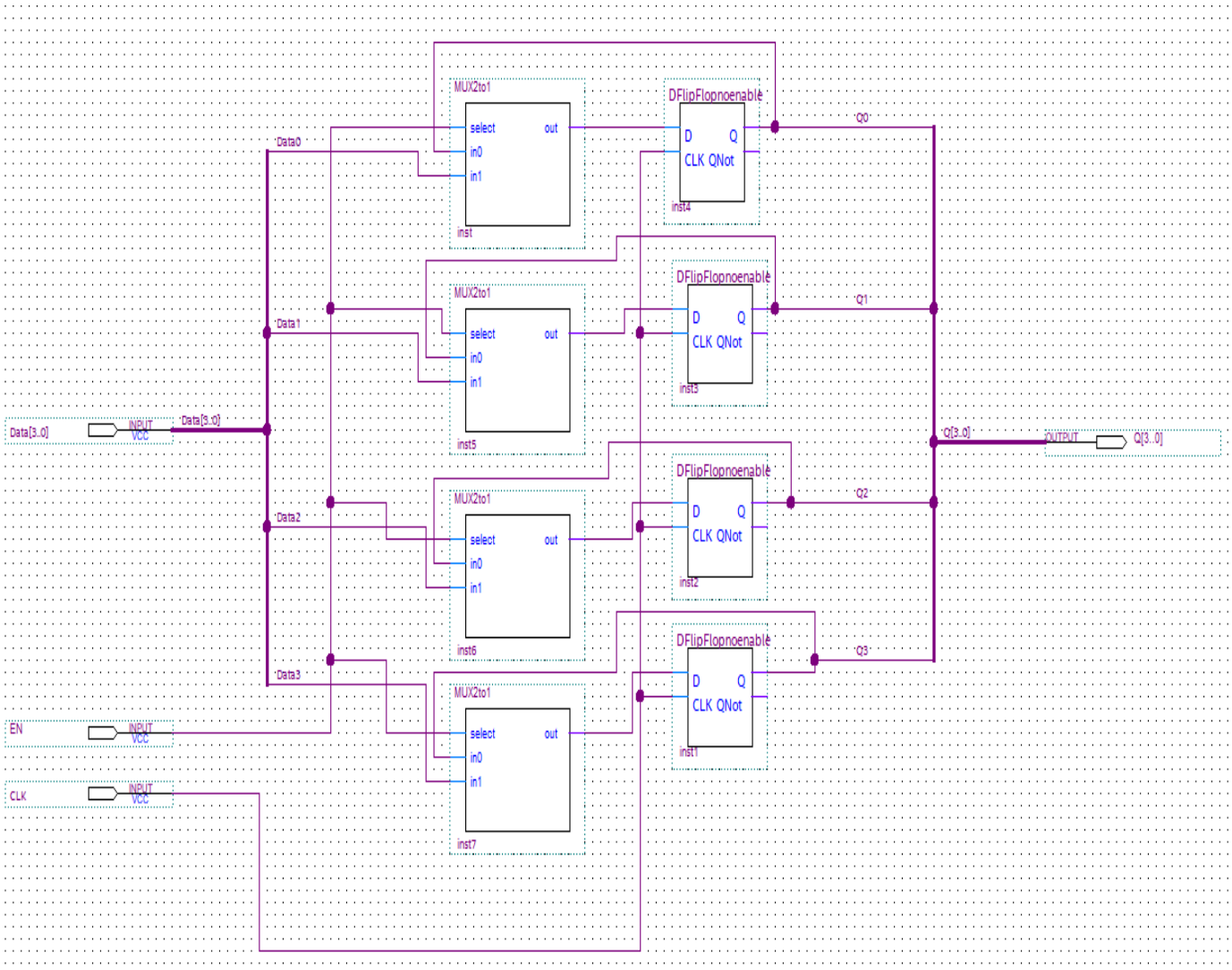
Once the system's native clock frequency is slowed to 180bpm the output signal goes into the **Sequencer**. The **sequencer** much similarly to **FourBitButtonIncrementCounter** increments 12 times and resets on each rising edge of the newly generated 3hz signal. The **sequencer** outputs a 4bit bus into a module **LED_Sequence** controlling turning on the LED's and into a select input of the **12-1MUX** below.

MUXes



This MUX is a fairly standard and recognizable component in digital logic, all this MUX does is output one of the twelve saved tones from **4BitRegister** depending the cycle of the **sequencer**. For example if the note 10 is saved on **4BitRegister** 3, when the **sequencers** output is 3 the MUX outputs a 10. (In this case the MUX's output is wired into a seven segment display, in practical application this output would be wired into a square wave generator which then generates a square wave corresponding to the specific pitch of the tone, but due to the lack of knowledge of how to output a square wave as an audio signal I opted for just visually showing it via a display)

Register File



4bitRegisterFile is also a very straightforward and recognizable component in digital logic, this allows a 4 bit number **Data[3..0]** to be loaded or saved depending on the **EN** enable input. To do this a bus **Data[3..0}** sends a 4 bit signal into 4 **2-1Mux** which controls the load/store functionality by determining the path of the data into 4 **Dflipflops**. These flip flops each hold a single data bit. This is known more specifically as a parallel access register. The **DFlipFlops** then output these bits into a bus **Q[3..0].** This component practically serves as the memory of the entire sequencer in which the tone data can be saved and played back later.

Outputs

Seven Segment Display

```
module Seven_Seg_Decoder(
    input X3,
    input X2,
    input X1,
    input X0,
    output reg A,
    output reg B,
    output reg C,
    output reg D,
    output reg E,
    output reg F,
    output reg G
);

always @(*)
begin
    case({X3, X2, X1, X0})
        4'b0000: {A, B, C, D, E, F, G} = 7'b0000001;
        4'b0001: {A, B, C, D, E, F, G} = 7'b1001111;
        4'b0010: {A, B, C, D, E, F, G} = 7'b0010010;
        4'b0011: {A, B, C, D, E, F, G} = 7'b0000110;
        4'b0100: {A, B, C, D, E, F, G} = 7'b1001100;
        4'b0101: {A, B, C, D, E, F, G} = 7'b0100100;
        4'b0110: {A, B, C, D, E, F, G} = 7'b0100000;
        4'b0111: {A, B, C, D, E, F, G} = 7'b0001111;
        4'b1000: {A, B, C, D, E, F, G} = 7'b0000000;
        4'b1001: {A, B, C, D, E, F, G} = 7'b0000100;
        4'b1010: {A, B, C, D, E, F, G} = 7'b0001000;
        4'b1011: {A, B, C, D, E, F, G} = 7'b1100000;
        4'b1100: {A, B, C, D, E, F, G} = 7'b0110001;
        4'b1101: {A, B, C, D, E, F, G} = 7'b1000010;
        4'b1110: {A, B, C, D, E, F, G} = 7'b0110000;
        4'b1111: {A, B, C, D, E, F, G} = 7'b0111000;
        default: {A, B, C, D, E, F, G} = 7'b1111111;
    endcase
end

endmodule
```

Above is the verilog code for module **Seven_Seg_Decoder** which takes a 4bit input and then converts that into a 7bit binary to be read and output into a seven segment display. This component is seen multiple times in the device and stays the same. Specifically they display the

Tone Select, and **Tone Playback** of the device. Functioning more just to show the user what is happening instead of affecting any internal logic.

Red LEDs

```
module led_sequence(
    input bpm_clk,
    input [3:0] data,
    output reg [11:0] dataOut
);

always @(posedge bpm_clk) begin
    dataOut <= 12'b0;
    case(data)
        4'b0001: dataOut[11] <= 1'b1;
        4'b0010: dataOut[10] <= 1'b1;
        4'b0011: dataOut[9] <= 1'b1;
        4'b0100: dataOut[8] <= 1'b1;
        4'b0101: dataOut[7] <= 1'b1;
        4'b0110: dataOut[6] <= 1'b1;
        4'b0111: dataOut[5] <= 1'b1;
        4'b1000: dataOut[4] <= 1'b1;
        4'b1001: dataOut[3] <= 1'b1;
        4'b1010: dataOut[2] <= 1'b1;
        4'b1011: dataOut[1] <= 1'b1;
        4'b1100: dataOut[0] <= 1'b1;
        default: dataOut <= 12'b0;
    endcase
end

endmodule
```

Above is the verilog code for the module led_sequence which controls the LEDs that light up above the switches on the FPGA. Whenever there is a positive edge in the **bpm_clk** a red LED that corresponds with the current beat of the **sequencer** lights up, otherwise stays unlit.
This component serves as a visual representation of where the beat is for the user to see, it does not affect any internal logic of the device.

To do this a 4 bit data is fed into **led_sequence** and depending on the value of that data a red led lights up. This is directly connected to the same **bpm_clk** of the **sequencer** so the LED and the note is directly correlated with one another.

Green LEDS

There are only two green LEDS connected to the device and they show the state of the buttons, **PlayPause** and **SaveState.** When the LED is lit up the sequence is in play state, and save state. Otherwise it is paused or not in save state.

Final Thoughts

That concludes the overall and technical description of my final project's innerworkings and design. Although this project in it’s current state works just fine I believe I can make it better by adding a reset function, and a way to dynamically change the tempo on the fly.
Of course a major problem with this device being a sequencer is that it does not actually output any sound, despite this I believe the design to be a very functional scaffold for a digital sequencer and in theory it would work exactly as intended. Unfortunately converting digital to audio using the FPGA’s onboard DAC proved to be too technically challenging for a person of my skill level. Though I hope that this report has cleared any confusions about the functionality of the device, and that a person could imagine the whole purpose of my project which is creating and listening back to some great sounding music. Overall, I used many of the topics, devices, and skills I learned directly from this class, and it was very cool and educational to create something like this and well worth the hardwork and effort I put into it.