LOGIC DESIGN LABORATORY

Team 4 Final Project Report

陳鮑比109006202, 劉其生109006240

Report Introduction

This report is about how we design, code, and implement Verilog to do our Final Project which is a simple recreation of an Arpeggiator and the tool we use to draw our logic gates diagram is https://app.diagrams.net/ and online.visual-paradigm.com

Team Contributions

陳鮑比109006202 - 50% :

- Do everything together

劉其生109006240 - 50% :

- Do everything together

What We Learned During Final Project

In this final project, this is our culmination of our hard work of doing all of task given during class time, and we are finally able to make a new project from our own idea, by using all the materials given from previous classes, such as clock dividers, pattern definition, finite state machines, peripherals modules such as keyboards, and audio modules to create sounds.

Challenges

Trying to figure out how to make the notes go in a sequential order during the very beginning of writing the code was very confusing, as we didn’t think a Finite State Machine was required, and we tried to make the tones only based on inputs.

Trying to figure out what features to add and improve for the Arpeggiator.

We wanted to add a filtering algorithm that can synthesize string-like sounds, but when we tried it, it failed badly and we decided to abandon that plan instead.

And as usual our nemesis, we can’t store speed values correctly at first, but finally found a solution to overcome the problem.

Final Project Introductions

In the final project we want to create an Arpeggiator. Arpeggiator is a tool on synthesizer that musicians use to explore new melodies, by experimenting on chords, and helping them to play the specified chord repeatedly without having the musician have to re-entering the input multiple times. In this project, we wanted to create a simple Arpeggiator that produces a simple melody.

Figure 1.1 Arpeggiator Example

This is an image of an Arpeggiator. Here we see that the inputs are being driven by piano keys and some knobs to modify patterns and modes. We decided to use buttons, switches and keyboard as input and use the audio module given as the output and some led, and the seven segment display in fpga to help debugging and indicator.

For example, in the image of Arpeggiator he is pressing down the 1 3 5 keys, then the arpeggiator will process that, and make sounds based on the currently pressed keys and the modification settings. It will play the notes in quick succession, (and repeat the notes pressed). And we will do more testing so we can get a better result.

We then can pre-program the patterns of the selected notes, and have the FPGA run the pattern sequentially from the lowest note to the highest, highest to lowest, random, etc.

An Arpeggiator can do alot of modification such as gain, tempo, delay, volume, pattern, transpose ,octave, etc. we want out arpeggiator at least be able to control pattern, octave, and tempo, and we will continue to add more Arpreggiator function to our machine, and try to make it as best as we can.

Material

All of the material that we are going to use are all available from the materials given in the class such as :

1. Basys 3 Artix 7.
2. Keyboard.
3. Audio PMOD.
4. Speaker.
5. Variable Resistor.
6. Cables.

Project Goals

For this project our goal is to replicate what an Arpeggiator can do as much as we can and as best as we can despite the existing limitations. And the features in our arpeggiator are :

1. Play in 3 different Octaves. Octaves 3, 4, and 5.

NOTE	OCTAVE 0	OCTAVE 1	OCTAVE 2	OCTAVE 3	OCTAVE 4	OCTAVE 5	OCTAVE 6	OCTAVE 7	OCTAVE 8
C	16.35	32.7	65.41	130.81	261.63	523.25	1046.5	2093	4186.01
C#/Db	17.32	34.65	69.3	138.59	277.18	554.37	1108.73	2217.46	4434.92
D	18.35	36.71	73.42	146.83	293.66	587.33	1174.66	2349.32	4698.63
D#/Eb	19.45	38.89	77.78	155.56	311.13	622.25	1244.51	2489.02	4978.03
E	20.6	41.2	82.41	164.81	329.63	659.25	1318.51	2637.02	5274.04
F	21.83	43.65	87.31	174.61	349.23	698.46	1396.91	2793.83	5587.65
F#/Gb	23.12	46.25	92.5	185	369.99	739.99	1479.98	2959.96	5919.91
G	24.5	49	98	196	392	783.99	1567.98	3135.96	6271.93
G#/Ab	25.96	51.91	103.83	207.65	415.3	830.61	1661.22	3322.44	6644.88
A	27.5	55	110	220	440	880	1760	3520	7040
A#/Bb	29.14	58.27	116.54	233.08	466.16	932.33	1864.66	3729.31	7458.62
B	30.87	61.74	123.47	246.94	493.88	987.77	1975.53	3951.07	7902.13

Figure 1.2 Frequency Table

2. Play in 3 different Patterns. Ascending, Descending, Ascending Descending (Bounce).

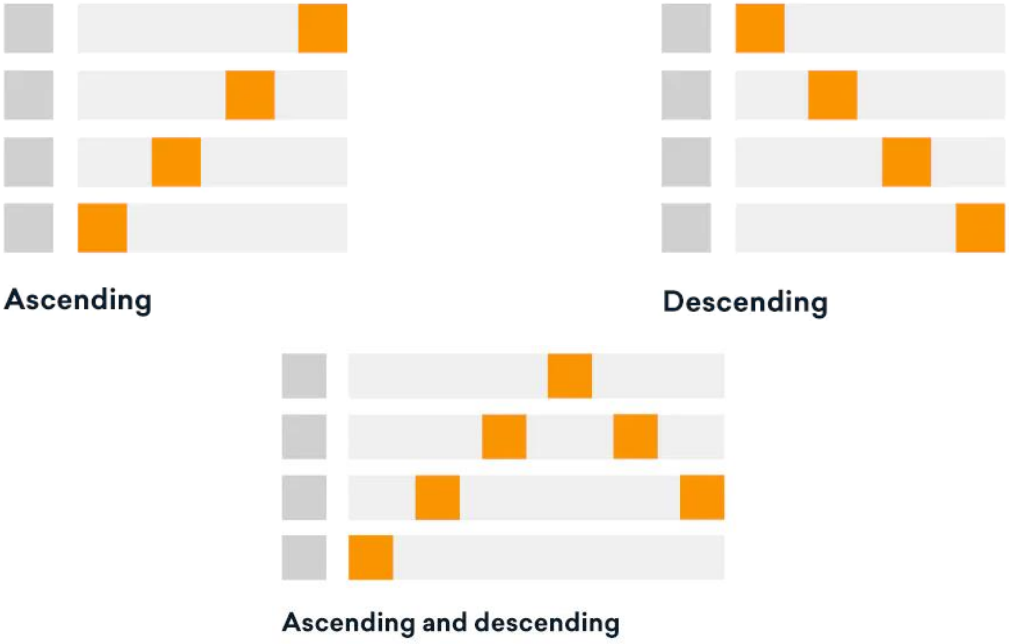

Figure 1.3 Pattern

3. Transpose.
4. Play in 8 different Tempo from Starting from 30 BPM to 240BPM.

Figure 1.4 Tempo

Hardware Design

For the hardware design there are nothing new the scheme are exactly like how we use speaker module on our previous labs

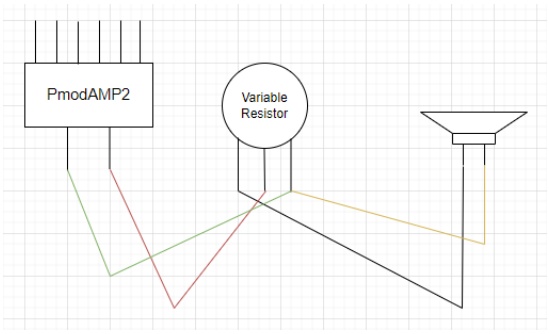

Figure 2.1 Speaker Diagram

And we used jumper cables to connect all of the connections and put them inside a mask box so the component won't be interrupted or fall off during the development of this project.

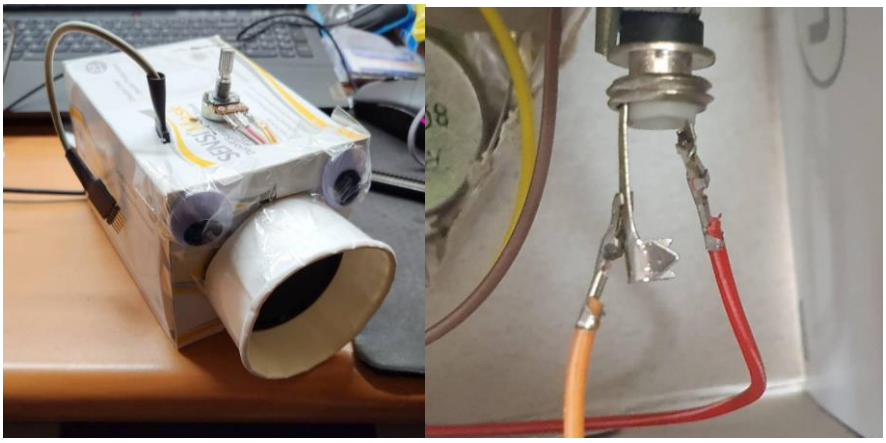

Figure 2.2 Speaker Cable Connections

Software Design

Firstly, we want to show you guys our top module, and we will be explaining top to bottom.

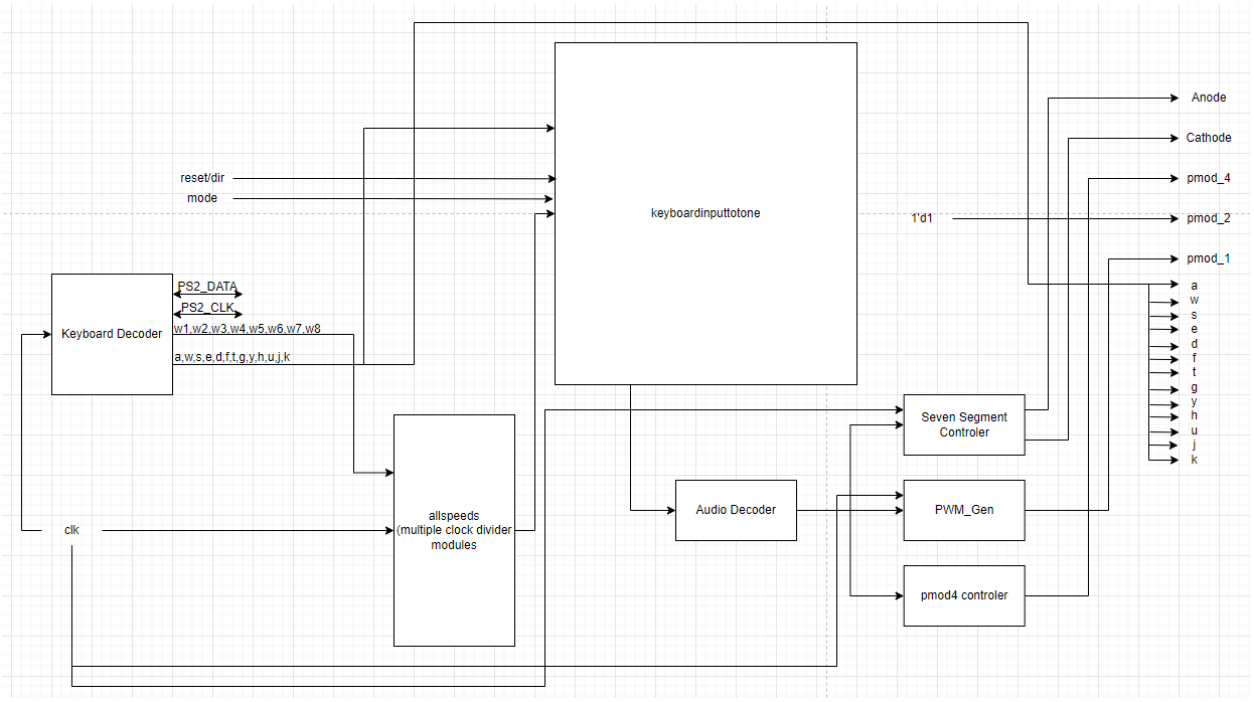

Figure 3.1 Top Module

This top module will manage all of the individual modules, and connect them together. We can see on the left side, there is a Keyboard Decoder, which takes the data from USB, and we can bind this data into keys. We pass these pressed keys into wires, where then later we can use them as our inputs in our other mini modules.

The used keys for sounds are a,s,d,f,g,h,j,k,w,e,t,y,u. And number keys which are being used for speeds are 1, 2, 3, 4, 5, 6, 7, 8.

These inputs, along with some switches we have defined (high octave, mid octave, low octave, mode, dir/reset) will also be used as inputs.

The number keys are used to control the clock speed, whereas the alphabetical keys will be used to control the keyboard_to_tone module that will create the tones, where later the tones are being decoded into frequencies which will be driven by the PWM to generate the sound.

Now, I am going to explain the heart of this project, which lies inside the Keyboard To Tone Module.

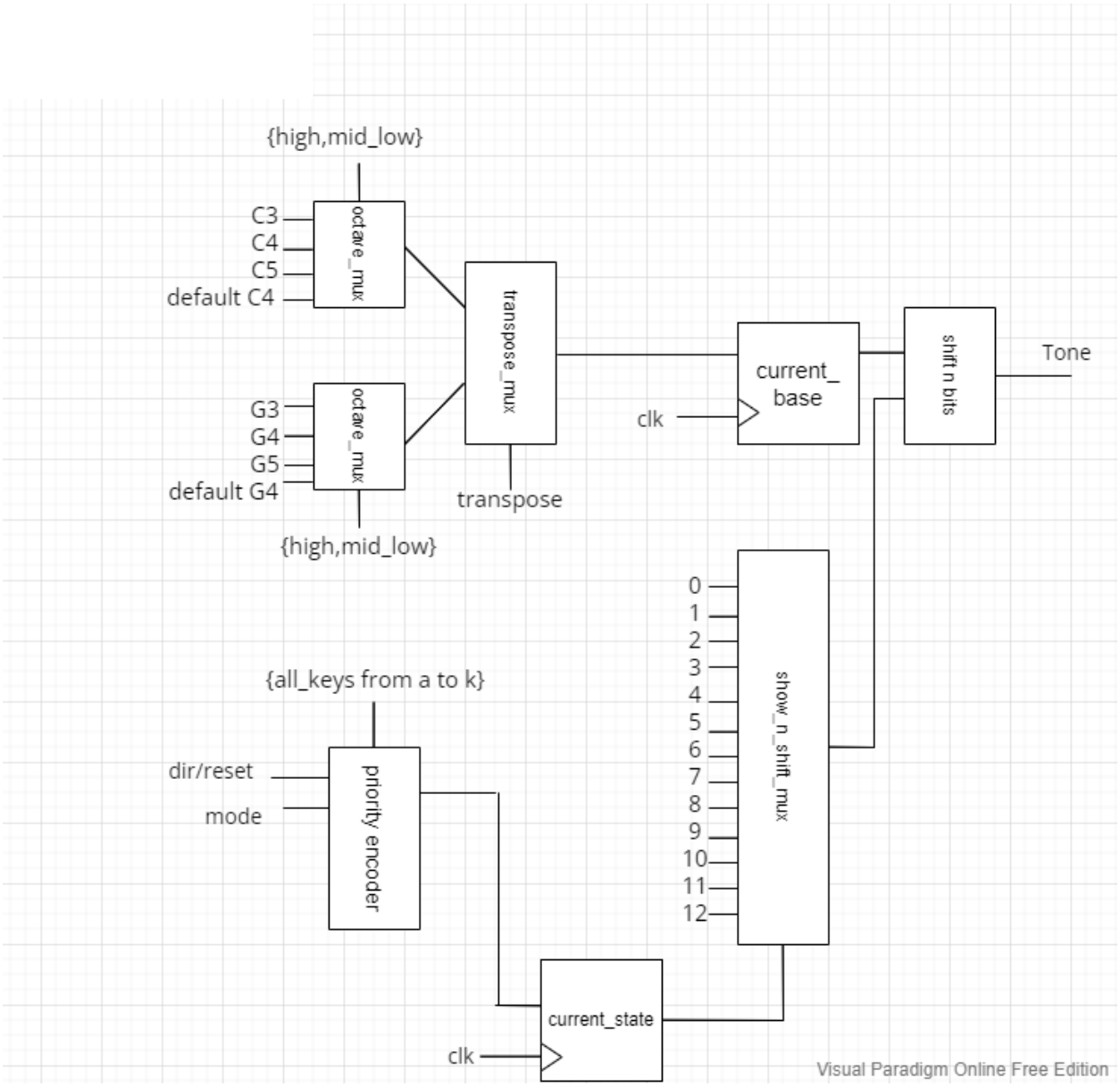

Figure 3.2 KeyboardToTone Module

This module will help us to translate our pressed tonal keys into tones, which later become the input for our audio decoder. But firstly let's explain the simple parts of this module.

Firstly, I defined the base octave for each octave, for the transposed and non transpose.

```
    parameter base_bawah =
46'b000_000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0001;
    parameter base_tengah =
46'b000_000_0000_0000_0000_0000_0000_0010_0000_0000_0000;
    parameter base_atas =
46'b000_000_0000_0000_0000_0010_0000_0000_0000_0000_0000_0000;
    parameter base_bawah_G =
46'b000_000_0000_0000_0000_0000_0000_0000_0000_1000_0000;
    parameter base_tengah_G =
46'b000_000_0000_0000_0000_0000_0001_0000_0000_0000_0000_0000;
    parameter base_atas_G =
46'b000_000_0000_0001_0000_0000_0000_0000_0000_0000_0000_0000;
    parameter NoTone=
46'b000_000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
```

These are the parameters used for my base octave. I will further explain this in the tone decoder.

To be able to choose the octave we want to play, we can use the switches from the FPGA to change what octave we want, and also we can choose whether we want to transpose our keys from C to G.

```
    always@(posedge clk)begin
        if(transposed) begin
            if(bawah && !tengah && !high) C4 <= base_bawah_G;
            else if(!bawah && tengah && !high) C4 <= base_tengah_G;
            else if(!bawah && !tengah && high) C4 <= base_atas_G;
            else C4 <= base_tengah_G;
        end else begin
            if(bawah && !tengah && !high) C4 <= base_bawah;
            else if(!bawah && tengah && !high) C4 <= base_tengah;
            else if(!bawah && !tengah && high) C4 <= base_atas;
            else C4 <= base_tengah;
        end
    end

    reg [4-1:0] state, nextstate;
    always@(posedge clk) begin
        if(speed) begin state <= nextstate; end
        else begin state <= state; end
    end
```

As we can see, the base tone will change whether the transpose or the octave selection has been activated. I have set the base octave to only change into its specific base if there are only 1 octave

switch set to high. If more than one, or none is set to high, I will revert to the default tone which is C4 or G4 depending if it is transposed or not.

I will then use another sequential block to define the next state logic. We see that if the speed is high, then it will change the state. The clock variable here is provided from the FPGA clock, and meanwhile the speed variable is coming from the clock divider.

The next state logic is using the Finite State Machine. where each state sends a tone that has been shifted accordingly, and later this tone output will be sent into the audio decoder to generate the correct frequency. The finite state machine has 13 states for each mode with the total edges of 169 edges. Although the edges are only 169 there are around 420 conditions that control the next frequency. Since the finite state machine will be too big to see we can make it simpler by using the few most left and few most right as the point.

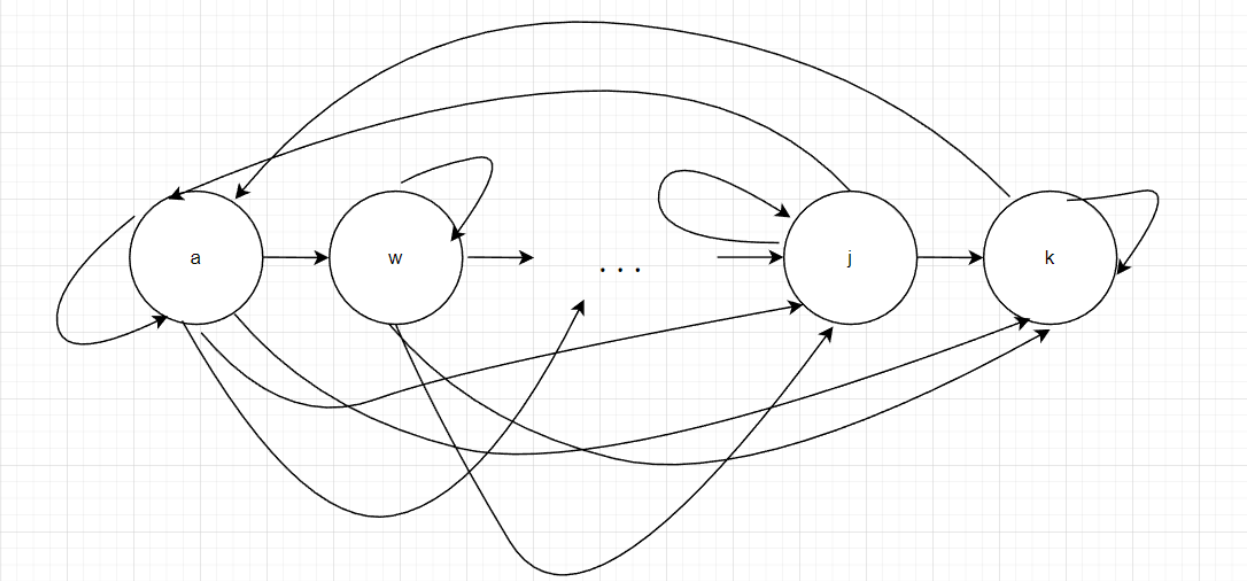

Figure 3.3 Tone State Diagram

This state diagram is actually a priority encoder, where I will explain with the next part of the code, but note that I am only showing you 1 or 2 states, since this will be very long.

```
                press_a : begin
                     if(a) tone = C4; else tone = NoTone;
```

```
                    if (w) begin
                        nextstate = press_w;
                    end else if (s) begin
                        nextstate = press_s;
                    end else if (e) begin
                        nextstate = press_e;
                    end else if (d) begin
                        nextstate = press_d;
                    end else if (f) begin
                        nextstate = press_f;
                    end else if (t) begin
                        nextstate = press_t;
                    end else if (g) begin
                        nextstate = press_g;
                    end else if (y) begin
                        nextstate = press_y;
                    end else if (h) begin
                        nextstate = press_h;
                    end else if (u) begin
                        nextstate = press_u;
                    end else if (j) begin
                        nextstate = press_j;
                    end else if (k) begin
                        nextstate = press_k;
                    end else nextstate = press_a;
                end
```

The priority encoder here is being used for the next state logic. The first if statement will govern whether to send out a tone or not. The tone sent here will be shifted from the base tone, in order to choose the note. If the key is not pressed, then it will not create any tones.

Then if it detects any other keys, such as w s e, etc, this will make the state change into the nearest pressed key (this is why we called it as a priority encoder). With this priority encoder in place, we can be sure that for key a, the only possible way to go is going to the right side of the keyboard, (note this is when the direction is going up).

If there is nothing pressed, then we just let the nextstate to keep the current state. This is to prevent a latch from happening.

I will show you another state so you know what I mean.

```
        press_w : begin
                if(w) tone = C4<<1; else tone = NoTone;
                if (s) begin
                    nextstate = press_s;
                end else if (e) begin
                    nextstate = press_e;
                end else if (d) begin
                    nextstate = press_d;
                end else if (f) begin
                    nextstate = press_f;
                end else if (t) begin
                    nextstate = press_t;
                end else if (g) begin
                    nextstate = press_g;
                end else if (y) begin
                    nextstate = press_y;
                end else if (h) begin
                    nextstate = press_h;
                end else if (u) begin
                    nextstate = press_u;
                end else if (j) begin
                    nextstate = press_j;
                end else if (k) begin
                    nextstate = press_k;
                end else if (a) begin
                    nextstate = press_a;
                end else nextstate = press_w;
            end
```

As we see the code structure is the same, but this time we will shift the keys on the left side of the w key as the last priority. Again, this is to ensure the direction of the note. We also see here that now the base tone is shifted by 1 bit, so w means that we are playing C# if it is not transposed, or a G# if it is transposed.

We are basically done explaining the priority encoder. One thing to note, this code I am currently showing to you is for the up direction, to go to reverse, this is the part that manages it.

```
    always@(*) begin
        if(dir) begin
            case (state)
```

This is still the same always block as the previous priority encoder. But as you can see, originally there was a dir signal here. If the dir signal was false, then the nextstate will use the priority encoder

on the reverse order. So if you play the note C4, the next note must be C5, which means it will go into the down direction.

```
end else begin
          case(state)
              //a,w,s,e,d,f,t,g,y,h,u,j,k
              press_a : begin
                  if(a) tone = C4; else tone = NoTone;
                  //write it in reverse order
                  if (k) begin
                      nextstate = press_k;
                  end else if (j) begin
                      nextstate = press_j;
                  end else if (u) begin
                      nextstate = press_u;
                  end else if (h) begin
                      nextstate = press_h;
                  end else if (y) begin
                      nextstate = press_y;
                  end else if (g) begin
                      nextstate = press_g;
                  end else if (t) begin
                      nextstate = press_t;
                  end else if (f) begin
                      nextstate = press_f;
                  end else if (d) begin
                      nextstate = press_d;
                  end else if (e) begin
                      nextstate = press_e;
                  end else if (s) begin
                      nextstate = press_s;
                  end else if (w) begin
                      nextstate = press_w;
                  end else nextstate = press_a;
              end
```

See, the priority has reversed, and we have managed to create the down direction.

Now after finishing defining the up and down direction, now I need to explain the up-down (ping pong) direction. Actually the way we implement this is by using two blocks of keyboard decoder,

where one will be enabled and disabled using the mode switch, but it's basically almost the same, with very minimal difference. We actually can fit both of these into one module, but it will just be so hard to read the errors.

This is the definition of the Keyboard to Tone module, the first 1 is the up or down direction, and the second is the ping pong direction.

```
    KeyBoardInputToTone translator(wa,ws,wd,wf,wg,wh,wj,wk, ww,we,wt,wy,wu,
clk, chosenspeed, getTone, just, dir_switch, bawah, tengah, high, transpose);
    KeyBoardInputToTonetoingtoing bolakbalik(wa,ws,wd,wf,wg,wh,wj,wk,
ww,we,wt,wy,wu, clk, chosenspeed, getTone2, just2, dir_switch
     ,bawah, tengah, high, transpose);
```

To choose which output we want to take, I just use this simple block to choose which tone values I should take, and it is governed by the mode switch.

```
    always@(*) begin
        if(mode) chosentone = getTone;
        else chosentone = getTone2;
    end

    assign finaltone = chosentone;

    AudioDecoder audiodecoder0(finaltone, freq);
```

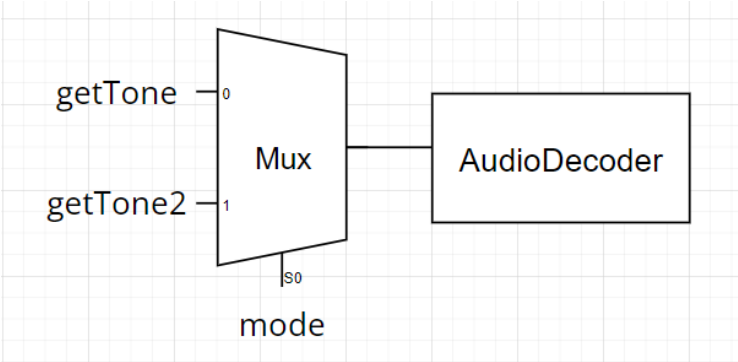

Figure 3.4 Choosing which Mode to Play

Which is simply this, so truthfully the Keyboard Input To Tone is both running at same time, even though we “disabled” them. If mode is set to high, we will use the up and down direction, and if mode is set to low, we will choose the ping pong direction.

Back again, now I’m explaining the code from my ping pong directional Keyboard to Tone.

```
    always@(*) begin
        if(reset)begin
            nextdir = 1'd1;
        end else begin
            nextdir = dir;
        end
        if(dir) begin
            case (state)
                press_a : begin
                    if(a) tone = C4; else tone = NoTone;
                    if (w) begin
                        nextstate = press_w;
                    end else if (s) begin
                        nextstate = press_s;
                    end else if (e) begin
                        nextstate = press_e;
                    end else if (d) begin
                        nextstate = press_d;
                    end else if (f) begin
                        nextstate = press_f;
                    end else if (t) begin
                        nextstate = press_t;
                    end else if (g) begin
                        nextstate = press_g;
                    end else if (y) begin
                        nextstate = press_y;
                    end else if (h) begin
                        nextstate = press_h;
                    end else if (u) begin
                        nextstate = press_u;
                    end else if (j) begin
                        nextstate = press_j;
                    end else if (k) begin
                        nextstate = press_k;
                    end else nextdir = 1'd0;
                end
```

In this section, we can see the priority encoder is basically the same, but the direction this time will be governed by whether we have reached the end of the priority encoder. The reset signal here is given from the direction switch previously in the up down module. Because we no longer require an up or down switch, we reuse that switch as our reset, so we can reset the direction. At the end of the priority encoder, the direction is flipped, so basically if the pressed key is on the end of the

priority encoder, it will switch direction, and by switching direction, and using the opposite priority encoder, we have successfully created the ping pong pattern.

And finally we are done discussing the keyboard to tone.

Now to convert this tone into frequency, we will just use a simple audio decoder, which actually is just a mux.

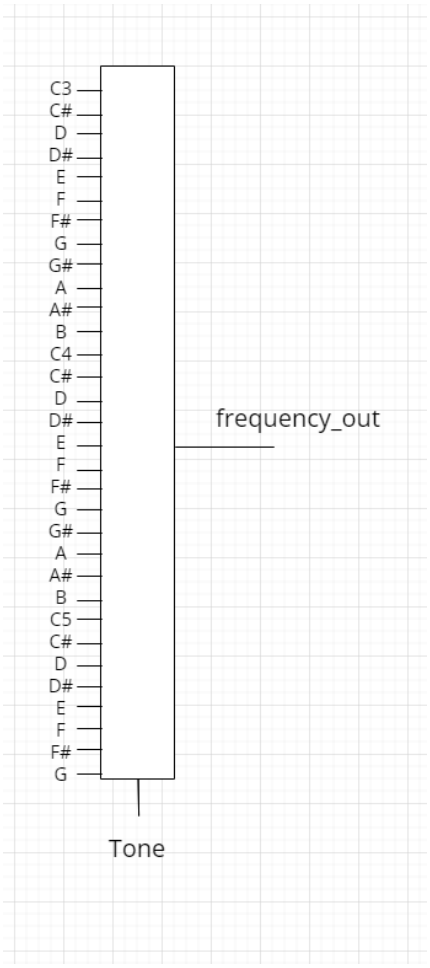

Figure 3.5 Tone Decoder to Frequency

The frequencies that are outputted from this module are the same frequencies from the frequency table shown in Figure 1.2.

Now this frequency will be driven through the PWM module, and we can output the sound as we need.

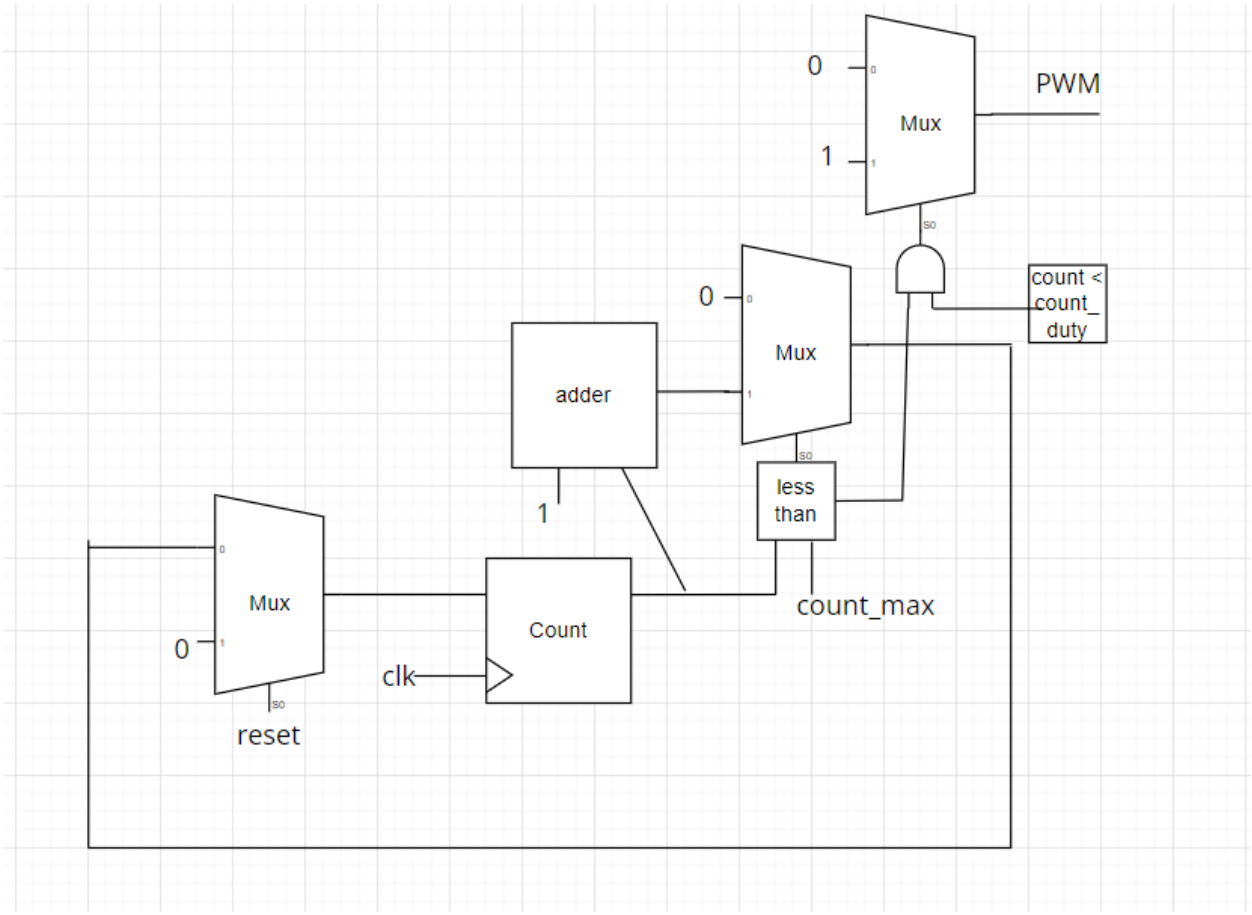

Figure 3.6 PWM Module

This is the PWM. The frequency driven into the PWM is used to count the count_max, by using the code of count_max = 100_000_000 / freq, and the count_duty we just set it to 10'd512.

Before I forget, the last module is the clock module. This module allows us to choose to play the music at different speeds, where we can control the speed by pressing the keys from 1 to 8, each key will correspond to each own BPM and can instantly change the tempo of the music.

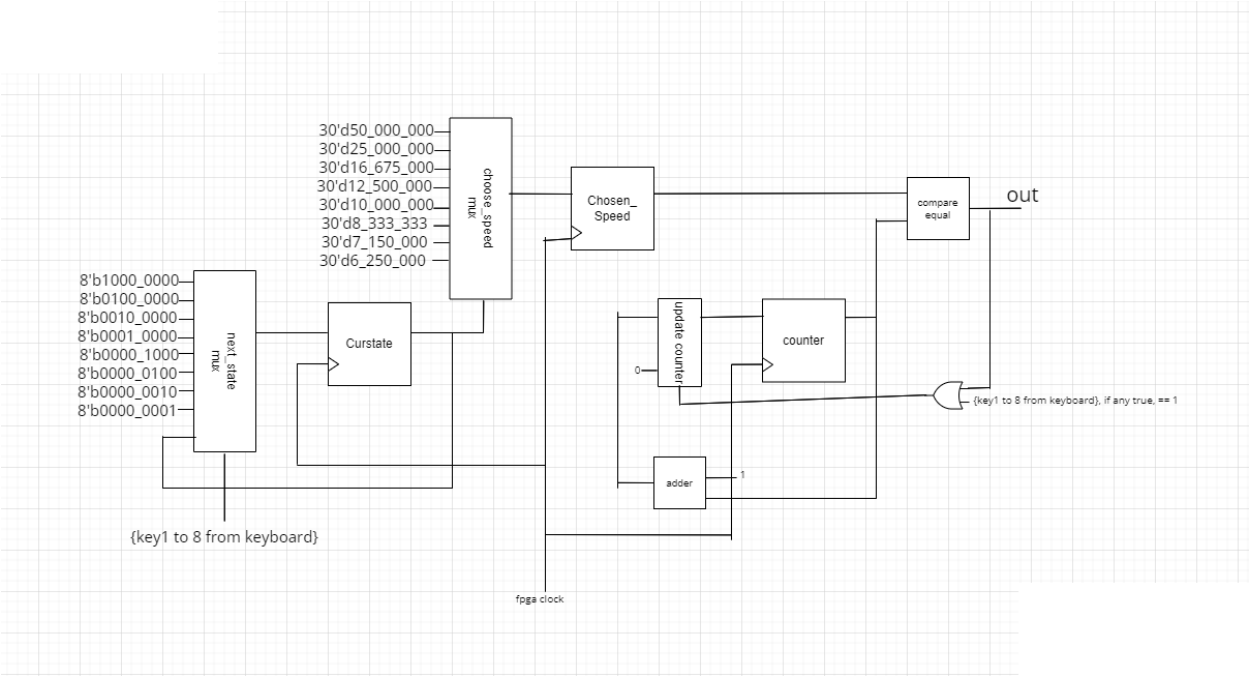

Figure 3.7 Multi-speed Clock

The way that this module works is by reading the current state, and choosing the correct comparator for the clock. The clock itself is a counter, which will add everytime the FPGA clock gets a posedge. The current state will define what speed we are currently using, and I have preset some clock timing as values stored in registers. The counter will go up, and if it is the same, then it will send a clock signal out and reset the counter back to 0.

All of the clock calculations were computed from this website:

https://tuneform.com/tools/time-tempo-bpm-to-milliseconds-ms

I am using this website to get the value for all BPMs that we are going to use. I am using the time division of 1/4, but that will work for the entire period, so the duration for each key will need to be divided again by 4, since my time signature is 4/4 .

BPM 30: 2000ms /4 = 500ms = 30'd50_000_000
BPM 60: 1000ms/4 = 250ms = 30'd25_000_000
BPM 90: 667ms/4 = 167ms = 30'd16_675_000
BPM 120: 500ms/4 = 125ms = 30'd12_500_000
BPM 150: 400ms/4 = 100ms = 30'd10_000_000
BPM 180: 333ms/4 = 83ms = 30'd8_333_333
BPM 210: 286ms/4 = 71ms = 30'd7_150_000
BPM 240: 250ms/4 = 62ms = 30'd6_250_000

For the 7 segment to display the frequency of the output sound we follow this BASYS 3 scheme as we can see in the following figures.

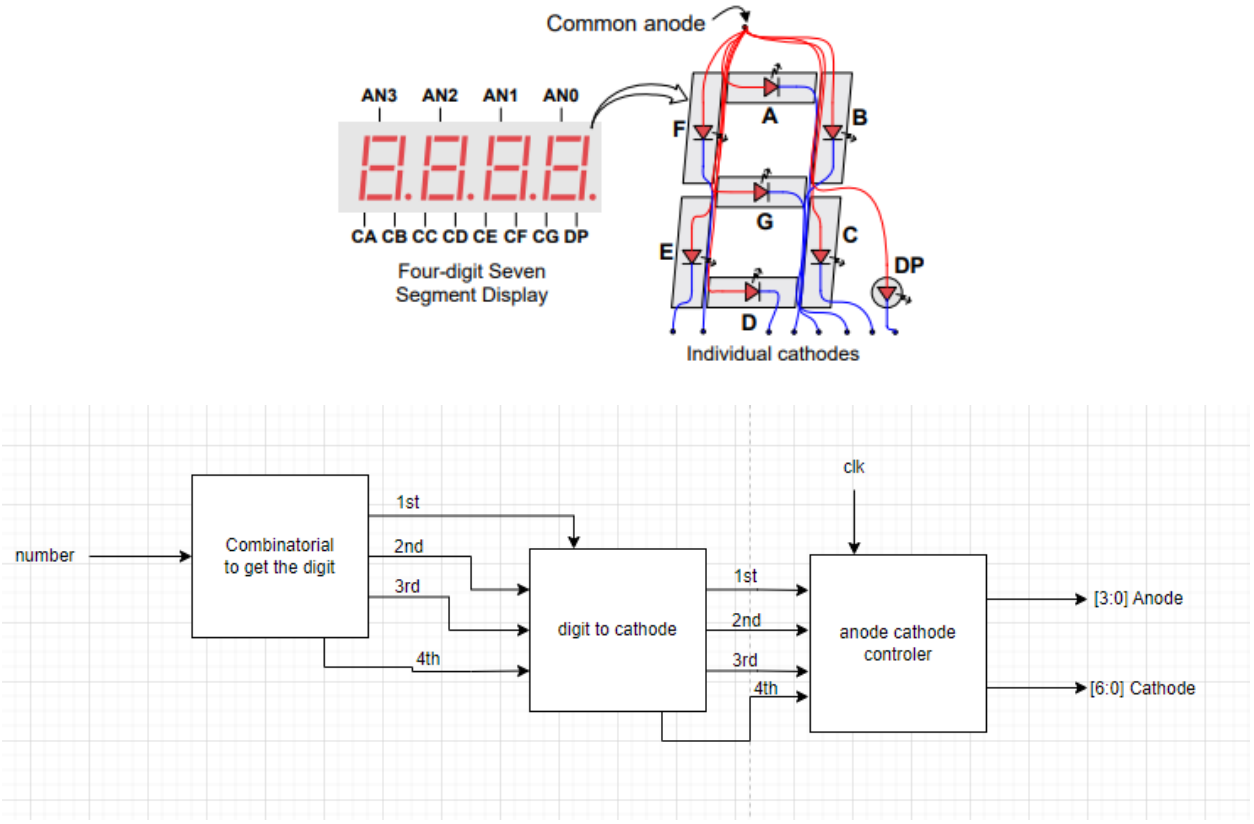

Figure 3.8 Seven Segment Display Module

For the seven segment we implemented the multiple digit implementation by dividing the time for each digit so we will be able to display different digits. The seven segments are mostly used for debugging to make sure that the produced sound is the correct frequency.

Conclusion

In conclusion, our team has managed to finish our predetermined goals, and have added some new features. We feel our work is very satisfying and we are very happy to be able to present our work to the classes, although there are some problems that we faced.

We have a lot of fun attending this class. Thank you for having us for this semester!