CHAPTER 4

RESULTS AND DISCUSSION

4.1 Simple beep

FPGAs can easily implement binary counter. Starting from the 25MHz clock, the clock can be simply divided using a 16 bits counter. 16 bits counter counts from 0 to 65535, so 65536 different values. That gives us a frequency of 25000000/65536=381 Hz.

For the music1.v (refer Appendix A), the LSB (counter [0]) would toggle with a frequency of 12.5 MHz. "counter [1]" with 6.125 MHz. The MSB (bit 15) of the counter is used to drive the output. A nice 381 Hz square signal comes out of "q" (make sure to specify the pin assignment in the FPGA software).

Since16 bits counter are use for this project, there is a problem for displaying a complete waveform, due to the huge transition time from a low to high pulse. However this can be explained with a simple four bits counter.

The output is low at the eight first positive clocks while for the next eight positive clocks the output is high. The pattern of the waveform will be continued alternating from low to high every eight positive clocks (refer Figure 4.0).

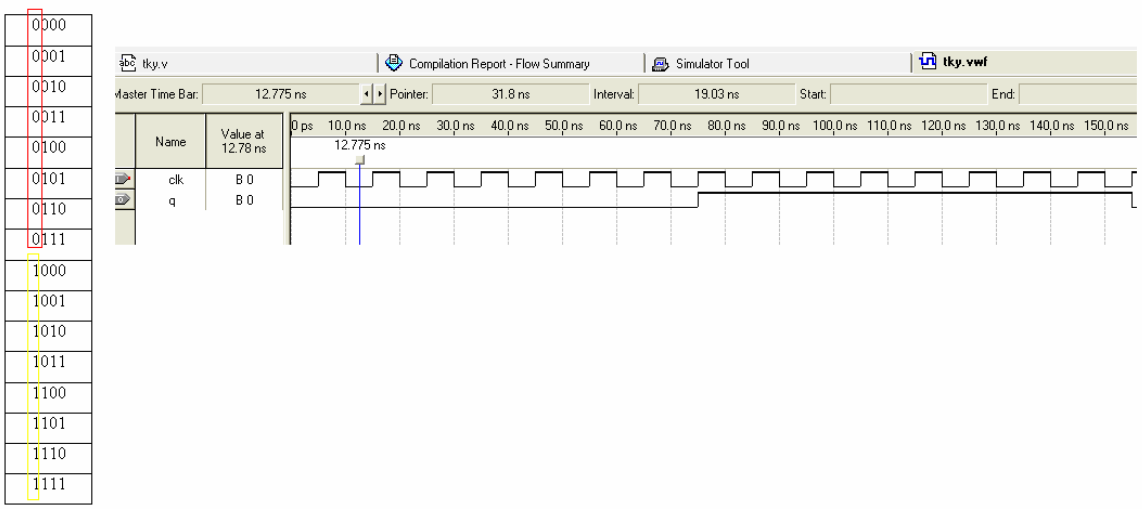

Figure 4.0: Simulation waveform for 4-bits counter.

4.2 Ambulance siren

For music2.v (refer Appendix B), 2 tones are alternated. Firstly, a 24 bits counter "tone" is used to produce a slow square wave. The MSB bit (tone [23]) toggles with a frequency of about 1.5 Hz. Then this bit is used to switch between 2 frequencies for the main counter, and that alternates between the 2 tones.

4.3 Police siren

For the music3.v (refer Appendix C), a ramp that sounds like a police siren is needed to generate. Firstly, start with "tone" counter. Only 23 bits is used to make it twice as fast (MSB toggles at around 3 Hz). Then here comes the trick to get an up-ramp. Bits 15 through bit 21 of the tone counter are extracted, such as: tone [21:15]. This will give 7 bits, which go from 0 to 127 at some medium speed. After it reaches 127, it rolls back to 0 and up again.

To get a down-ramp, the same bits are inverted, such as: (~tone [21:15]). This will give 7 bits again, that go down from 127 to 0. To switch between the up-ramp and the down-ramp, tone [22] is used. As soon as the up-ramp hits 127, the down-ramp is switched, until it goes to 0, and then back to the up-ramp.

```
wire [6:0] ramp = (tone[22] ? tone[21:15] : ~tone[21:15]);

// That means
// "if tone[22]=1 then ramp=tone[21:15] else ramp=~tone[21:15]"
```

So "ramp" value goes from 7'b0000000 to 7'b1111111. To get a usable value to produce a sound, the 2 bits '01' is padded up front, and the 6 bits '000000' in the back.

```
wire [14:0] clkdivider = {2'b01, ramp, 6'b000000};
```

This way, "clkdivider" have a value ranging from 15'b010000000000000 to 15'b011111111000000 or in hex 15'h2000 to 15'h3FC0, or in decimal 8192 to 16320. With a 25 MHz input clock that produces "q" from 765 Hz to 1525 Hz. That will give a high pitch siren.

4.3.1 High-speed pursuit

For music4.v (refer Appendix D), the siren is sometimes slow, sometimes fast. So "tone [21:15]" will give a fast ramp, while "tone [24:18]" will give a slow one.

4.4 Playing notes

For music5.v (refer Appendix E, a way to play notes like on a keyboard is needed. If 6 bits are used to encode a note, 64 notes are getting. There are 12 notes per octaves, so 64 notes will give more than 5 octaves, more than enough for a little tune.

4.4.1 Step 1

To play a range of increasing notes, a 28 bits counter is instantiated, from which the 6 most significant bits are extracted, to give the 6 bits of the note to play.

```
reg [27:0] tone;
always @(posedge clk) tone <=  tone + 1'b1;
wire [5:0] fullnote = tone[27:22];
```

With a 25 MHz clock, each note lasts 167 ms and it takes 10.6 s to play all 64 notes.

4.4.2 Step 2

The "fullnote" is divided by 12. It will give the octave (5 octaves, so 3 bits are enough, since it goes from 0 to 4) and the note (from 0 to 11, so 4 bits).

```
wire [2:0] octave;
wire [3:0] note;
divide_by12 divby12(.numer(fullnote[5:0]), .quotient(octave), .remain(note));
```

A sub-module called "divide_by12" is instantiated which takes case of the division. Refer to Appendix F.

4.4.3 Step 3

To go from one octave to the next, frequency is multiplied by "2". To go from one note to the next, frequency is multiplied by "1.0594". It is not really easy to do in hardware. So a look-up table with pre-calculated values is used.

The main clock is divided by 512 for note A, by 483 for note A#, by 456 for note B. Remember, dividing by a lower value will give a higher frequency/higher note, that is what wanted.

```
always @(note)
case(note)
4'b0: clkdivider = 9'b100000000 - 1'b1; // A
4'b0001: clkdivider = 9'b111100011 - 1'b1; // A#/Bb
4'b0010: clkdivider = 9'b111001000 - 1'b1; // B
4'b0011: clkdivider = 9'b110101111 - 1'b1; // C
4'b0100: clkdivider = 9'b110010110 - 1'b1; // C#/Db
4'b0101: clkdivider = 9'b110000000 - 1'b1; // D
4'b0110: clkdivider = 9'b101101010 - 1'b1; // D#/Eb
4'b0111: clkdivider = 9'b101010110 - 1'b1; // E
4'b1000: clkdivider = 9'b101000011 - 1'b1; // F
4'b1001: clkdivider = 9'b100110000 - 1'b1; // F#/Gb
4'b1010: clkdivider = 9'b100011111 - 1'b1; // G
4'b1011: clkdivider = 9'b100001111 - 1'b1; // G#/Ab
4'b1100: clkdivider = 1'b0; // should never happen
4'b1101: clkdivider = 1'b0; // should never happen
```

```
 4'b1110: clkdivider = 1'b0; // should never happen
 4'b1111: clkdivider = 1'b0; // should never happen
endcase

always @(posedge clk)
        if(counter_note==0)
                counter_note <= clkdivider;
        else
                counter_note <= counter_note - 1'b1;
```

Every time "counter_note" equals 0, which will give a tick for the next stage: the octave divider.

4.4.4 Step 4

For the lowest octave, "counter_note" is divided by 256. For octave 1, it is divided by 128 and so on.

```
reg [7:0] counter_octave;
always @(posedge clk)
if(counter_note==0)
begin
 if(counter_octave==0)
  counter_octave <= (octave==0?8'b11111111:
                     octave==1?8'b01111111:
                     octave==3?8'b00111111:
                     octave==3?8'b00011111:
                     octave==4?8'b00001111:7);
else
```

```
counter_octave <= counter_octave - 1'b1;
end

reg q;
always @(posedge clk)
        if(counter_note==0 && counter_octave==0)
                q <= ~q;
```

4.5 Summary

Through the experiments, the value of the speaker used has to be low ohm. While a higher ohm speaker is used, the desired output cannot be obtained. As the output voltage from the UP2 board is small, therefore small value of the speaker is used.