

Urządzenia cyfrowe i systemy wbudowane
Organy z możliwością kontroli obwiedni
dźwięku

Karol Noga, 241259

Leszek Błażewski, 241264

Semestr letni 2019/2020

Spis treści

1 Wstęp	5
1.1 Cel i zakres projektu	5
1.2 Opis sprzętu	5
1.3 Zagadnienia teoretyczne	6
1.3.1 Generowanie fal	6
1.3.2 Generowanie obwiedni	7
2 Projekt	8
2.1 Schemat szczytowy	8
2.2 Hierarchia modułów	8
2.2.1 Module_Generation	8
2.3 Przepływ danych między modułami	11
2.4 Opis poszczególnych modułów	11
2.4.1 Wykorzystane gotowe moduły	11
2.4.2 Freq_calc	12
2.4.3 Env_control	15
2.4.4 Env_gen	18
2.4.5 Oscillator	22
2.4.6 Multiplier	24
3 Implementacja	26
3.1 Rozmiar	26
3.1.1 Zajętość LUT	26
3.1.2 Plastry	26
3.1.3 Pozostałe	26
3.2 Prędkość	26
3.3 Podręcznik użytkownika zaimplementowanego urządzenia	26
3.3.1 Konfiguracja potrzebnego sprzętu	26
3.3.2 Instrukcja użytkowania	27
4 Wykresy fal oraz generowanie pliku wave	28
4.1 Wstęp	28
4.2 Generowanie wykresu fali na podstawie danych z projektu	28
4.2.1 Uzyskanie momentów czasowych oraz wartości sygnału z symulacji	29
4.3 Generowanie danych dla zadanej piosenki	34
4.3.1 Skrypt odpowiedzialny za przełożenie nut na wartości wejściowe modułu generującego falę	34

4.3.2	Wykorzystanie wygenerowanych danych w symulacji	37
4.4	Zamiana uzyskanych danych na plik wave	38
5	Podsumowanie	41
5.1	Ocena krytyczna	41
5.2	Ocena pracy	41
5.3	Możliwe kierunki rozbudowy układu	41
6	Literatura	42

Spis rysункów

1	Sprzęt wykorzystany podczas realizacji projektu	5
2	Schemat szczytowy	8
3	Schemat modułu Module_Generation	8
4	Zbliżony rzut z symulacji modułu Module_Generation	10
5	Zrzut z symulacji modułu Module_Generation z podpisanymi fazami obwiedni	10
6	Schemat modułu Freq_calc	12
7	Symulacja modułu Freq_calc	15
8	Schemat modułu Env_control	15
9	Symulacja modułu Env_control	17
10	Schemat modułu Env_gen	18
11	Graf maszyny stanów modułu Env_gen	19
12	Zrzut z symulacji modułu Freq_Calc z podpisanymi fazami obwiedni	22
13	Schemat modułu Oscillator	22
14	Symulacja modułu Oscillator	24
15	Schemat modułu Multiplier	24
16	Symulacja modułu Multiplier	25
17	Podłączenie wymaganych urządzeń do płyty	27
18	Mapowanie klawiszy pianina na klawiaturze komputerowej - górna linia: cyfry 1-0, dolna linia: litery Q-P	28
19	Wygenerowane wykresy fali	33
20	Przykład ciągu nut MIDI w programie FL Studio dla głównej melodii z utworu Avicii - Levels	34
21	Podgląd przebiegu fali w wygenerowanym pliku wave w programie Audacity	39

Spis listingów

1	Definicja stanów opisujących wszystkie dostępne dźwięki	13
2	Proces wyznaczający następny stan modułu na podstawie wejścia z klawiatury	13
3	Proces odpowiedzialny za sygnalizację naciśnięcia i zwolnienia klawisza	14
4	Przypisanie do wyjścia modułu wartości sterującej wysokością dźwięku (obliczonej wg wzoru nr 1), na podstawie aktualnego stanu modułu	14
5	Definicja sygnałów z wartościami startowymi oraz stałych używanych w module	16
6	Proces odpowiedzialny za obliczanie wartości podawanych na wyjście	17
7	Proces odpowiedzialny za określanie następnego stanu - czyli fazy obwiedni - na podstawie stanu klawiszy oraz aktualnego mnożnika obwiedni, realizuje maszynę stanów wg grafu	20
8	Proces odpowiedzialny za obliczanie aktualnej wartości mnożnika	21
9	Zależność pozostałych wyjść od aktualnego stanu modułu	21
10	Implementacja modułu Multiplier	25
11	Podstawowa sekwencja dźwięków	29
12	Proces przekazujący wartości	30
13	Proces odpowiedzialny za zapis wartości do pliku	30
14	Fragment pliku z próbками z symulacji	31
15	Skrypt w języku Python odpowiedzialny za wygenerowanie wykresu	32
16	Tablica nut dla generowanej piosenki	35
17	Skrypt odpowiedzialny za generowanie wejścia dla modułu symulacji	36
18	Fragment pliku z wygenerowanymi danymi	37
19	Proces odpowiedzialny za odczyt wartości z pliku	38
20	Skrypt odpowiedzialny za utworzenie pliku wave z danych symulacji	40

Spis tabelic

1	Mapowanie klawiszy pianina do danych nut	28
2	Mapowanie klawiszy do obsługi obwiedni	28

1 Wstęp

1.1 Cel i zakres projektu

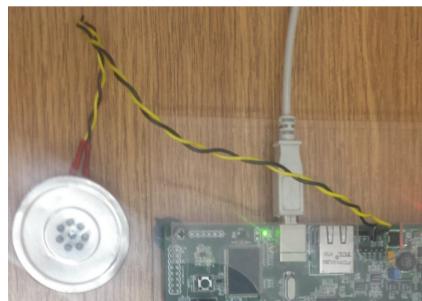
Celem projektu było stworzenie jedno oktawowych organów wydających dźwięki za pomocą głośnika podpiętego do układu Spartan 3E. Odgrywanie pojedynczych dźwięków odbywało się za pomocą klawiszy klawiatury połączonej przy pomocy portu PS/2. Dodatkowa część projektu obejmowała implementację obwiedni, która pozwalała na modulację dźwięku wydawanego przez instrument. Z powodu epidemii wywołanej wirusem nie mieliśmy możliwości przetestowania zaimplementowanego projektu dlatego jako dodatkowe wymaganie określiliśmy, możliwość wizualizacji wygenerowanej fali oraz odtworzenie jej z wykorzystaniem pliku wave. Dodatkowe wymagania zostały zrealizowane z wykorzystaniem języka Python, który w znacznym stopniu ułatwił całą implementację.

1.2 Opis sprzętu

Projekt został zrealizowany na układzie programowalnym FPGA Spartan-3E [1] dostępnym w laboratorium, który wraz z głośniczkiem dołączanym na odpowiednich pinach i klawiaturą stanowił komplet wymaganych urządzeń.



(a) Układ Spartan-3E



(b) Głośnik podpięty do układu



(c) PS/2

Rysunek 1: Sprzęt wykorzystany podczas realizacji projektu

Dźwięki wydobywały się z prostego głośniczka z dwoma przewodami podpiętymi do układu. Głośnik przetwarzał zmiany napięcia z wyjścia przetwornika cyfrowo-analogowego na fale akustyczne o danej częstotliwości. Granie na organach oraz zmiana ustawień obwiedni odbywało się za pomocą standardej klawiatury komputerowej, komunikującej się z układem FPGA poprzez port szeregowy PS/2.[2].

1.3 Zagadnienia teoretyczne

1.3.1 Generowanie fali

Jako sygnał dźwiękowy generowaliśmy falę piłokształtną, ponieważ implementacja modułu odpowiedzialnego za realizację piły była prosta i opierała się na wcześniej poznanym modelu licznika. Cały układ generujący falę składał się z dwóch liczników, gdzie zewnętrzny odpowiadał, za osiągnięcie odpowiedniej częstotliwości (a dokładniej długość trwania okresu fali), natomiast wewnętrzny generował na wyjściu próbki reprezentujące sygnał piłokształtny.

Sygnał powstaje dzięki inkrementacji liczb od 0 do 31, po czym licznik resetuje się i zaczyna pracę od nowa. Taktowanie zegara na płycie wynosi 50 MHz, dlatego w celu wydobycia dźwięku o niższej częstotliwości należało odpowiednio zmniejszyć częstotliwość inkrementacji licznika, co jest sterowane poprzez zmianę zakresu drugiego z liczników.

Obliczenie potrzebnego zakresu polegało na podzieleniu wartości takto-wania zegara przez zakres amplitudy generowanego sygnału ($2^5 = 32$) oraz częstotliwość odpowiadającą docelowej wysokości dźwięku[3].

Przykładowo dla dźwięku C_6 o częstotliwości 1046,5Hz wykonywaliśmy następujące działanie:

$$\left\lfloor \frac{50 * 10^6}{32 * 1046,5} \right\rfloor = 1493_{10} = 5D5_{16} \quad (1)$$

Według wyniku powyższego równania, drugi z liczników powinien liczyć modulo 1493, a każde jego zresetowanie powinno inkrementować pierwszy licznik, co zmienia wartość sygnału wyjściowego. Analogicznie do zaprezentowanego przykładu obliczyliśmy wymagane wartości dla pozostałych klawiszy organów.

Przed wysłaniem sygnału do głośnika, przechodzi on przez przetwornik cyfrowo-analogowy[2], który generował na podstawie jego wartości odpowiednie napięcie sterujące głośnikiem.

1.3.2 Generowanie obwiedni

W projekcie został zaimplementowany mechanizm kontroli obwiedni dźwięku. Obwiednia[4] jest oparta na modelu *ASR* i składa się z trzech faz:

- Attack - czas od wygenerowania dźwięku do osiągnięcia najwyższej amplitudy
- Sustain - próg do którego poziom amplitudy stabilizuje się po fazie Attack, przy czym w projekcie jest on stały i jest równy maksymalnemu możliwemu poziomowi amplitudy sygnału 12-bit
- Release - czas wygaszania fali od poziomu w fazie Sustain do całkowitego przerwania dźwięku

Uzyskany efekt polega na modulacji amplitudy podczas trwania sygnału, dzięki czemu dźwięk nie zaczyna emitować się nagle i nie urywa się, tylko brzmi naturalniej. Co więcej, manipulacja parametrami obwiedni - długością faz *Attack* i *Release* - pozwala na dodatkowe możliwości ekspresji poza wyborem wysokości dźwięku i rytmem gry.

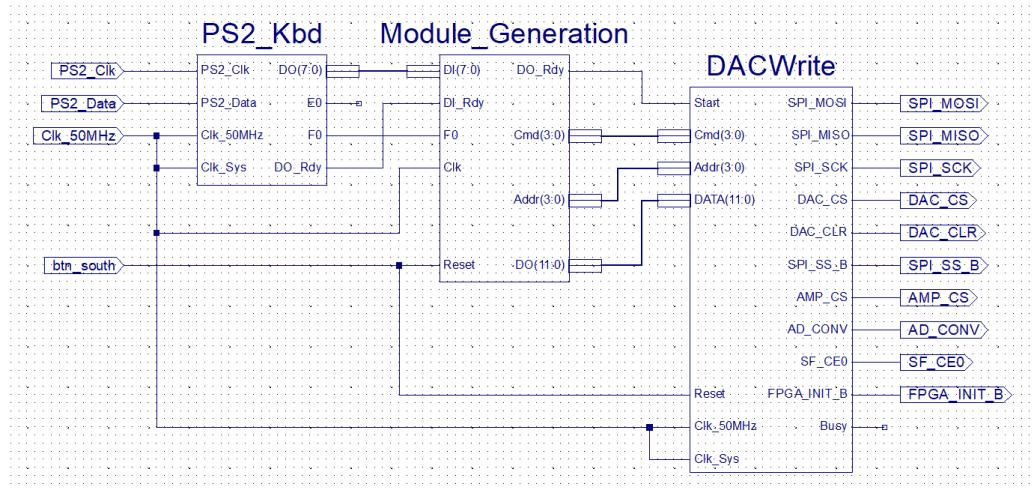
Realizacja obwiedni sprowadzała się do implementacji dwóch modułów VHDL:

- Maszyny stanów z licznikiem, której zadaniem było przełączanie pomiędzy wymienionymi fazami obwiedni - *Attack*, *Sustain*, *Release* - oraz generowanie mnożnika
- Układu mnożącego, który mnożył sygnał przez podany mnożnik w fazach *Attack* i *Release*

Dodatkowo utworzyliśmy skrzynkę *Env_control*, która pozwala na manipulację długością trwania faz *Attack* i *Release* za pomocą klawiatury PS/2.

2 Projekt

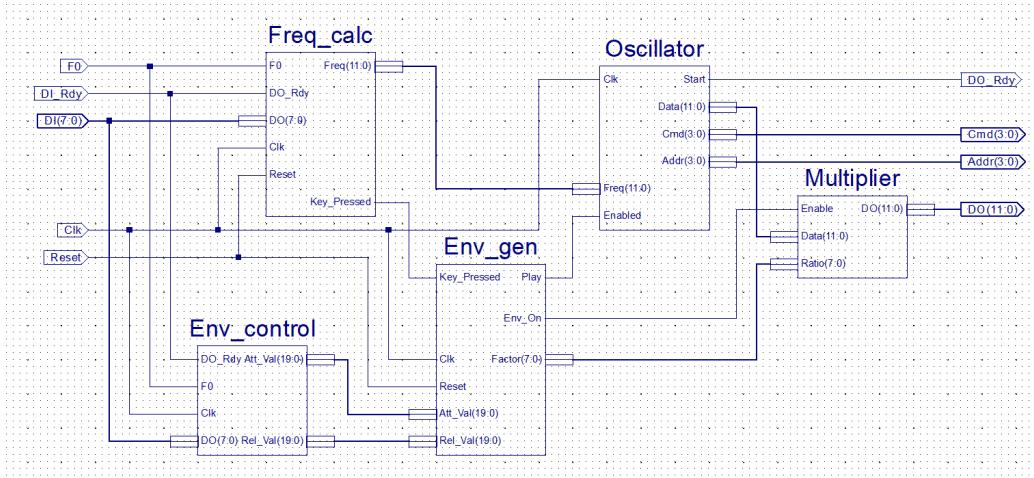
2.1 Schemat szczytowy



Rysunek 2: Schemat szczytowy

2.2 Hierarchia modułów

2.2.1 Module_Generation



Rysunek 3: Schemat modułu Module_Generation

W aplikacji wyróżniliśmy jeden zbiorczy moduł o nazwie *Module_Generation*, który zawiera wszystkie napisane przez nas skrzynki i po wygenerowaniu symbolu pozwala na użycie go w dowolnym schemacie.

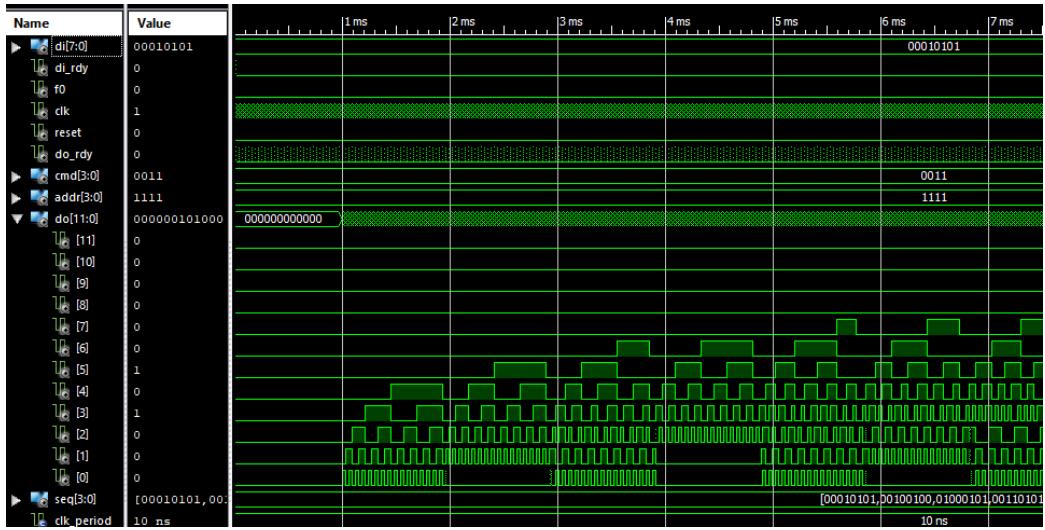
Moduł ten składa się z następujących skrzynek:

- *Freq_calc* - odpowiada za ustalenie wysokości granego dźwięku w zależności od naciskanego klawisza
- *Env_control* - pozwala kontrolować długość poszczególnych faz obwiedni przy użyciu klawiatury
- *Env_gen* - odpowiada za generowanie obwiedni w momencie wciskania i puszczenia klawisza
- *Oscillator* - generuje falę piłokształtną o odpowiedniej częstotliwości na podstawie danych z *Freq_calc*
- *Multiplier* - dokonuje mnożenia wartości wygenerowanych przez *Env_gen* z falą wygenerowaną przez *Oscillator*, tym samym nakładając obwiednię na generowany dźwięk

Moduł ten realizuje wszystkie z założeń projektowych dotyczących implementacji organów oraz obsługi obwiedni pozwalającej na modulowanie dźwięku.

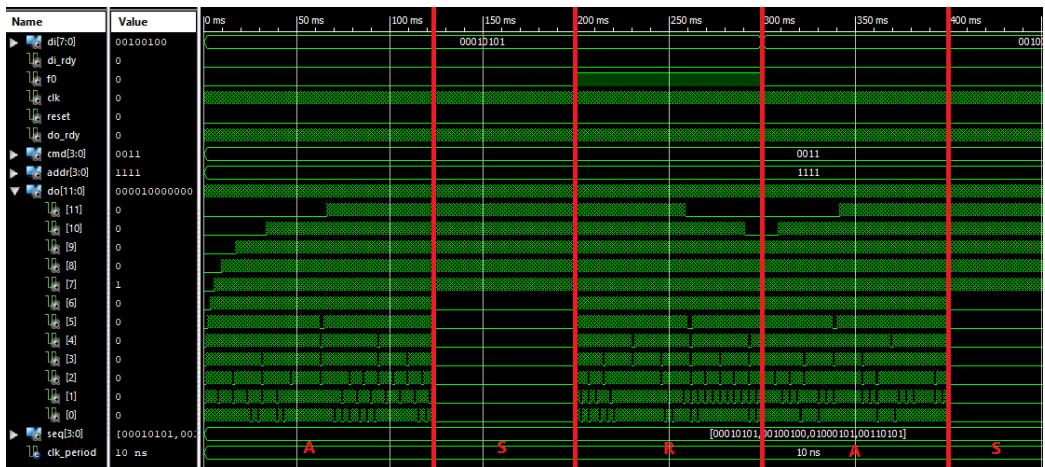
Dokładny opis poszczególnych komponentów oraz sposób implementacji danej funkcjonalności został zawarty w opisie każdej z skrzynek w poniższym rozdziale.

Poniżej załączono zrzuty ekranu symulacji całego modułu wraz z objaśnieniami poszczególnych faz.



Rysunek 4: Zbliżony zrzut z symulacji modułu Module_Generation

Na przybliżonym zrzucie ekranu z symulacji układu widać wyraźnie moment *1ms*, w którym wciśnięty został pierwszy klawisz i rozpoczęło się generowanie dźwięku. Rozwinięty przebieg sygnału *DO* pokazuje narastanie amplitudy związane z fazą *Attack* obwiedni dźwięku - okres fali dźwiękowej generowanej przez oscylator wynosi niecałe *1ms* co daje prawie 7 przebiegów widocznych na obrazku.



Rysunek 5: Zrzut z symulacji modułu Module_Generation z podpisanyimi fazami obwiedni

Drugi zrzut jest bardziej oddalony i pokazuje dokładniej działanie obwiedni - najpierw obserwujemy narastanie sygnału związane z fazą *Attack*,

po ok. 125ms widzimy moment, gdy 7 najmłodszych bitów jest równe 0, co jest związane z brakiem mnożenia w fazie *Sustain* i tym, że generowany sygnał ma rozdzielcość 5-bitową. Po 200ms widać fazę *Release* spowodowaną puszczeniem klawisza. W chwili 300ms naciśnięto kolejny klawisz, przekierując do fazy *Attack*, jednak zachowując osiągniętą wartość mnożnika, co powoduje naturalnie brzmiące przejście między kolejnymi dźwiękami - brak nagłego urwania dźwięku i resetu obwiedni.

2.3 Przepływ danych między modułami

Schemat szczytowy składa się z 3 modułów, w tym dwóch gotowych, pochodzących ze strony uczelni i odpowiednio zintegrowanych z resztą schematu. Do każdego z wejść zegarowych podpięty jest zegar wbudowany w układ z taktowaniem 50 MHz. Na samym początku znajduje się gotowy moduł *PS2_Kbd*, odpowiadający za odczyt kodów klawiatury, które z kolei odpowiadają poszczególnym dźwiękom. Odpowiednie kody na wyjściu przechodzą do już samodzielnie zaimplementowanego modułu *Module_Generation*, który składa się z podkomponentów, opisanych w poprzednim podrozdziale. Odczytany kod klawisza trafia do skrzynek *Freq_calc* oraz *Env_control*, gdzie pierwsza ustala wysokość granego dźwięku natomiast druga steruje parametrami obwiedni. Następnie wyjście modułu *Freq_calc* trafia do modułu *Oscillator*, gdzie jest traktowane jako zakres jednego z liczników, co pozwala wygenerować falę piłokształtną o zadanej częstotliwości, natomiast wyjście *Env_control* trafia do skrzynki *Env_gen*, która wyznacza współczynnik przez który mnożyć będziemy wyjściowy sygnał, modulując tym samym jego amplitudę. Ostatecznie wyjście skrzynek *Oscillator* oraz *Env_gen* trafia do modułu *Multiplier*, który dokonuje mnożenia danych i wyprowadza docelową wartość próbki na gotowy moduł *DACWrite*, odpowiadający za przetworzenie cyfrowego sygnału na wejściu na analogowy, który grany jest przez głośnik.

2.4 Opis poszczególnych modułów

2.4.1 Wykorzystane gotowe moduły

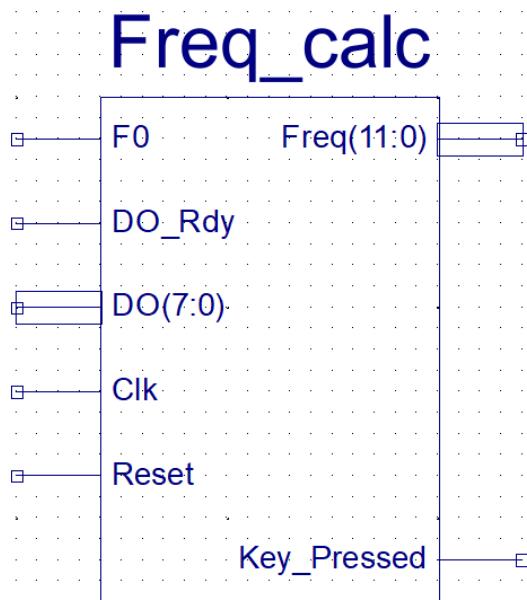
W celu realizacji całego projektu wykorzystaliśmy dwa gotowe moduły[2], które zostały przygotowane przez prowadzącego:

- PS2_Kbd
- DACWrite

Pierwszy z nich odpowiedzialny był za odbieranie kodów wysyłanych z klawiatury podłączonej na porcie szeregowym PS/2 i wykorzystany został do grania odpowiednich dźwięków po naciśnięciu zdefiniowanych klawiszy.

Natomiast moduł DACWrite odpowiadał za wysyłanie wygenerowanych sygnałów do przetwornika cyfrowo/analoga LTC2624.

2.4.2 Freq_calc



Rysunek 6: Schemat modułu Freq_calc

Moduł Freq_calc odpowiada za obliczanie częstotliwości generowanego sygnału na podstawie wejścia z klawiatury.

Wejścia

- F0 - sygnał mówiący czy klawisz został zwolniony
- DO_Rdy - sygnalizacja gotowości do odczytu danych z klawiatury
- DO(7:0) - kod klawisza odebrany z klawiatury PS/2
- Clk - wejście zegara
- Reset - sygnał resetujący układ

Wyjścia

- Freq(11:0) - wartość przekazywana do licznika modułu *Oscillator*, która jest proporcjonalna do długości generowanej fali piłokształtnej i odpowiada za sterowanie wysokością dźwięku
- Key_Pressed - sygnalizuje, czy klawisz przypisany do wysokości dźwięku jest aktualnie naciśnięty, używane do sterowania obwiednią

```
type state_type is (None, C1, Cis1, D1, Dis1, E1, F1, Fis1, G1, Gis1,
                     A1, Ais1, B1, C2, Cis2, D2, Dis2, E2);
```

Listing 1: Definicja stanów opisujących wszystkie dostępne dźwięki

```
state_proc : process( state, DO, F0, DO_Rdy )
begin
    next_state <= state;

    if DO_Rdy = '1' and F0 = '0' then
        case DO is
            when X"15" =>
                next_state <= C1;
            when X"1E" =>
                next_state <= Cis1;
            (...)

            when X"4D" =>
                next_state <= E2;
            when others =>
                next_state <= state;
        end case;
    end if;
end process state_proc;
```

Listing 2: Proces wyznaczający następny stan modułu na podstawie wejścia z klawiatury

```

keystate_proc : process( Clk, D0_Rdy, F0, D0, state )
begin
    if rising_edge(Clk) then
        if D0_Rdy = '1' and F0 = '0' then
            key_state <= '1';
        elsif D0_Rdy = '1' and F0 = '1' then
            if (state = C1 and D0 = X"15")
            or (state = Cis1 and D0 = X"1E")
            or (...)

            or (state = E2 and D0 = X"4D") then
                key_state <= '0';
            end if;
        end if;
    end if;
end process keystate_proc;

Key_Pressed <= key_state;

```

Listing 3: Proces odpowiedzialny za sygnalizację naciśnięcia i zwolnienia klawisza

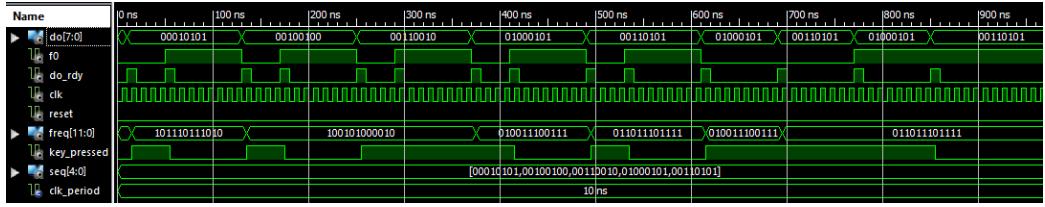
Logika wykrywania puszczonego klawisza została zaimplementowana w taki sposób, żeby był on rejestrowany tylko wtedy, gdy odpowiada ostatnio naciśniętemu klawiszowi. Zrealizowano to poprzez porównanie kodu klawisza z kodem przypisanym do aktualnego stanu modułu. Niweluje to sytuacje kiedy wciskając następny przycisk przed zwolnieniem poprzedniego, przewróimy odgrywany dźwięk puszczając jakikolwiek z trzymanych klawiszy, nawet taki nieprzypisany do żadnego dźwięku (w przypadku braku sprawdzania ich kodu).

```

with state select
    Freq <= X"BBA" when C1,
    X"B02" when Cis1,
    ...
    X"4A1" when E2,
    X"000" when others;

```

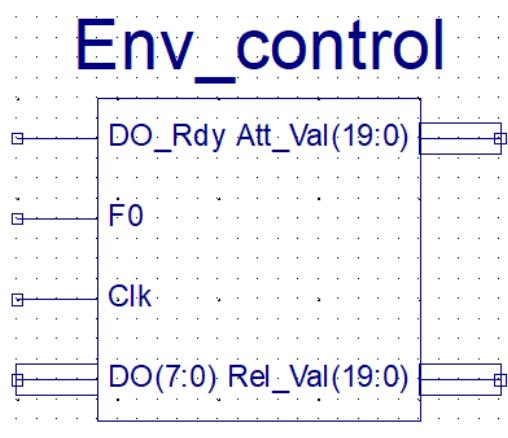
Listing 4: Przypisanie do wyjścia modułu wartości sterującej wysokością dźwięku (obliczonej wg wzoru nr 1), na podstawie aktualnego stanu modułu



Rysunek 7: Symulacja modułu Freq_calc

Symulacja pokazuje, że moduł prawidłowo reaguje na symulowane sygnały z klawiatury, ignoruje nieprzypisane klawisze (trzeci klawisz w symulacji) oraz sygnalizuje zwolnienie klawisza tylko, gdy ostatni naciśnięty klawisz został podniesiony.

2.4.3 Env_control



Rysunek 8: Schemat modułu Env_control

Moduł *Env_control* odpowiada za sterowanie obwiednią przy pomocy klawiatury. Odbiera on sygnały z klawiatury PS/2 i na ich podstawie zwiększa lub zmniejsza wartości manipulujące długością faz *Attack* i *Release* obwiedni, które to podaje na dwa 20-bitowe wyjścia. Zakres sterowania od 0 do 2,5s ze skokiem o 250ms. Wartość którą musimy podać na wyjściu, żeby otrzymać docelową długość fazy obwiedni liczymy ze wzoru:

$$l = \frac{50 * 10^6 * t}{2^7} \quad (2)$$

gdzie t to czas w sekundach, a l to wartość sterująca, którą musimy przekazać do modułu *Env_gen*.

Wejścia

- DO_Rdy - sygnalizacja gotowości do odczytu danych z klawiatury
- F0 - sygnał mówiący czy klawisz został zwolniony
- Clk - wejście zegara
- DO(7:0) - kod klawisza odebrany z klawiatury PS/2

Wyjścia

- Att_Val(19:0) - wartość odpowiadająca za długość trwania fazy *Attack*
- Rel_Val(19:0) - wartość odpowiadająca za długość trwania fazy *Release*

```
signal att_len : UNSIGNED(19 downto 0) := X"0BEDE"; -- 125 ms
signal rel_len : UNSIGNED(19 downto 0) := X"0BEDE"; -- 125 ms

constant STEP : UNSIGNED(19 downto 0) := X"0BEDE"; -- 125 ms
constant LIMIT : UNSIGNED(19 downto 0) := X"EE6D8"; -- 2500 ms
```

Listing 5: Definicja sygnałów z wartościami startowymi oraz stałych używanych w module

```

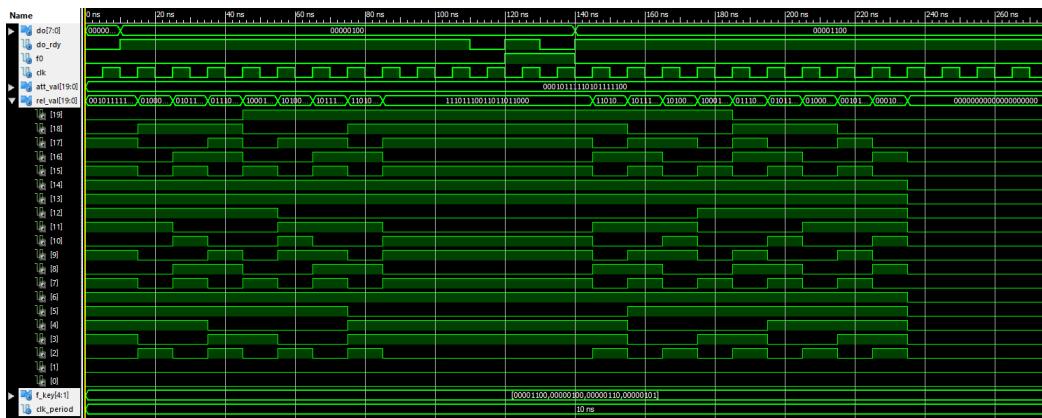
main_proc : process( Clk, DO, DO_Rdy, F0 )
begin
    if rising_edge(Clk) and DO_Rdy = '1' and F0 = '0' then
        if DO = X"05" and att_len < LIMIT then
            att_len <= att_len + STEP;
        elsif DO = X"06" and att_len > 0 then
            att_len <= att_len - STEP;
        elsif DO = X"04" and rel_len < LIMIT then
            rel_len <= rel_len + STEP;
        elsif DO = X"0C" and rel_len > 0 then
            rel_len <= rel_len - STEP;
        end if;
    end if;
end process main_proc;

Att_Val <= STD_LOGIC_VECTOR(att_len);

Rel_Val <= STD_LOGIC_VECTOR(rel_len);

```

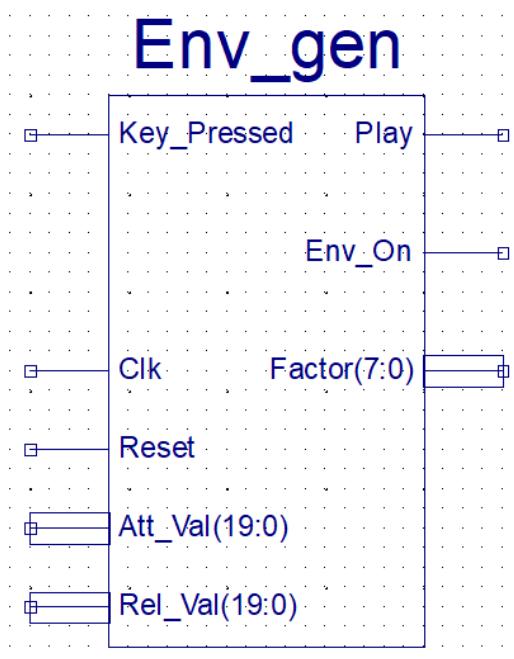
Listing 6: Proces odpowiedzialny za obliczanie wartości podawanych na wyjście



Rysunek 9: Symulacja modułu Env_control

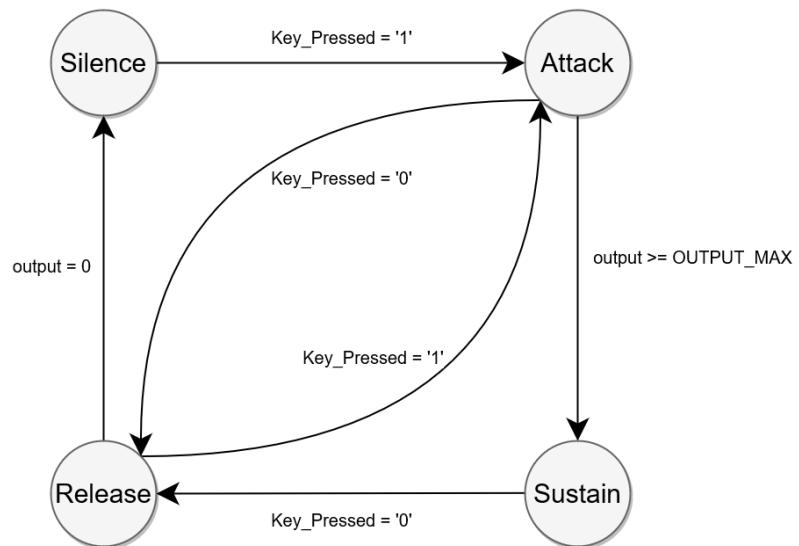
Na symulacji najpierw widzimy rosnące wartości *Att_Val*, w miarę trzymania przycisku *F3*, do momentu osiągnięcia limitu. Później symulacja przytrzymańia przycisku *F4* powoduje stopniowy spadek *Att_Val*, aż do osiągnięcia zera. Pokazuje to, że układ prawidłowo reaguje na klawisze sterujące obwiednią oraz nie przekracza ustalonych limitów.

2.4.4 Env_gen



Rysunek 10: Schemat modułu Env_gen

Moduł Env_gen odpowiada za określanie aktualnej fazy obwiedni oraz wartości jej mnożnika na podstawie sygnału określającego stan klawiszy oraz wartości określających długość trwania poszczególnych faz.



Rysunek 11: Graf maszyny stanów modułu Env_gen

Wejścia

- Key_Pressed - sygnalizacja stanu klawiszy przypisanych do dźwięków
- Clk - sygnał zegarowy
- Reset - resetowanie układu
- Att_Val(19:0) - wartość określająca długość fazy narastania, obliczona wg wzoru nr 2
- Rel_Val(19:0) - wartość określająca długość fazy wygaszania, obliczona wg wzoru nr 2

Wyjścia

- Play - sygnalizacja czy moduł *Oscillator* powinien generować dźwięk
- Env_On - sygnalizacja czy moduł *Multiplier* powinien mnożyć sygnał przez wartość *Factor*
- Factor(7:0) - wartość aktualnego mnożnika obwiedni

```

state_proc : process( state, Key_Pressed, Att_Val, Rel_Val, output )
begin
    next_state <= state;

    if state = Silence and Key_Pressed = '1' then
        next_state <= Attack;

    elsif state = Attack then
        if Key_Pressed = '0' then
            next_state <= Release;
        elsif output >= OUTPUT_MAX then
            next_state <= Sustain;
        end if;

    elsif state = Sustain and Key_Pressed = '0' then
        next_state <= Release;

    elsif state = Release then
        if Key_Pressed = '1' then
            next_state <= Attack;
        elsif output = 0 then
            next_state <= Silence;
        end if;
    end if;
end process state_proc;

```

Listing 7: Proces odpowiedzialny za określanie następnego stanu - czyli fazy obwiedni - na podstawie stanu klawiszy oraz aktualnego mnożnika obwiedni, realizuje maszynę stanów wg grafu

```

output_proc : process( Clk, count )
begin
    if rising_edge(Clk) then
        if state = Silence and next_state = Attack then
            output <= X"00";
        elsif state = Attack and output <= OUTPUT_MAX and count = 0 then
            output <= output + 1;
        elsif state = Sustain and next_state = Release then
            output <= OUTPUT_MAX;
        elsif state = Release and output > X"00" and count = 0 then
            output <= output - 1;
        end if;
    end if;
end process output_proc;

Factor <= STD_LOGIC_VECTOR(output);

```

Listing 8: Proces odpowiedzialny za obliczanie aktualnej wartości mnożnika

O ile wyznaczanie wartości *count* to prosty licznik w dół, zsynchronizowany z zegarem *Clk*, w którym jedyną zmienną jest wartość początkowa zależna od stanu *state* i wartości na odpowiednim wejściu (*Att_Val* lub *Rel_Val*) to obliczanie mnożnika jest właściwie zrealizowane jako 8-bitowy licznik dwukierunkowy, który inkrementuje/dekrementuje się w momencie osiągnięcia zera przez sygnał *count*, zależnie od fazy obwiedni - dla *Attack* zwiększamy wartość *output* o 1, dla *Release* zmniejszamy.

```

with state select
    Play <= '0' when Silence,
                '1' when Attack | Sustain | Release,
                'X' when others;

with state select
    Env_On <= '0' when Silence | Sustain,
                '1' when Attack | Release,
                'X' when others;

```

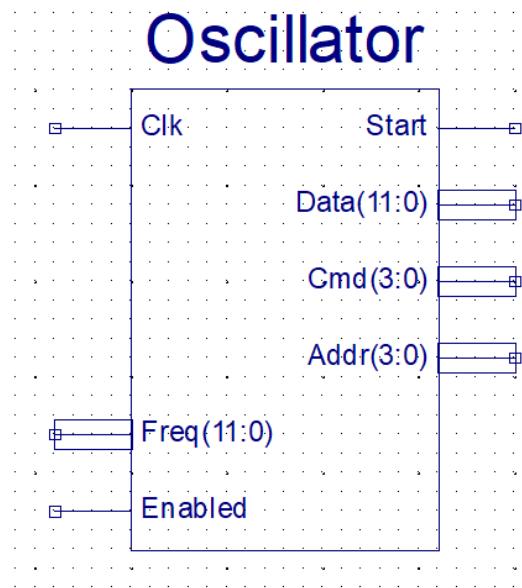
Listing 9: Zależność pozostałych wyjść od aktualnego stanu modułu



Rysunek 12: Zrzut z symulacji modułu Freq_Calc z podpisanymi fazami obwiedni

Na symulacji zostały wyszczególnione fazy obwiedni dźwięku. Moduł był testowany dla 125ms narastania i 250ms wygaszania. Na zrzucie widzimy, że zgadza się długość poszczególnych faz, sygnał *factor* dla fazy *Attack* zmienia swoje wartości dwa razy szybciej niż podczas wygaszania, a mnożnik zostaje zachowany przy naciśnięciu klawisza jeszcze przed osiągnięciem końca fazy *Release*.

2.4.5 Oscillator



Rysunek 13: Schemat modułu Oscillator

Moduł *Oscillator* odpowiada za generowanie fali piłokształtnej o zadanej częstotliwości oraz emisję pozostałych sygnałów potrzebnych do sterowania skrzynką *DAC_Write*.

Wartość *Freq* potrzebną do generowania sygnału o częstotliwości f_1 obliczamy z wzoru:

$$\frac{50 * 10^6 \text{ Hz}}{2^5 * f_1} \quad (3)$$

gdzie 2^5 dotyczy zakresu licznika, który odpowiada z generowanie próbek fali piłokształtnej - czyli po prostu zakresu jej amplitudy - natomiast licznik ułamka to częstotliwość zegara układu (50MHz).

Wejścia

- Clk - wejście zegarowe
- Freq(11:0) - wartość odpowiadająca za częstotliwość generowanej fali, proporcjonalna do długości fali
- Enabled - sygnalizuje, czy moduł powinien generować sygnał

Wyjścia

- Start - sygnalizacja gotowości danych do odczytu
- Data(11:0) - wygenerowany sygnał piłokształtny o zadanej częstotliwości
- Cmd(3:0) - sygnał sterujący modułem *DAC_Write*, który odpowiada za rozkaz wysłany do przetwornika (stała wartość oznaczająca emisję sygnału przetwornikiem)
- Addr(3:0) - sygnał sterujący modułem *DAC_Write*, który wskazuje na które piny na płytce powinien zostać wysłany sygnał analogowy

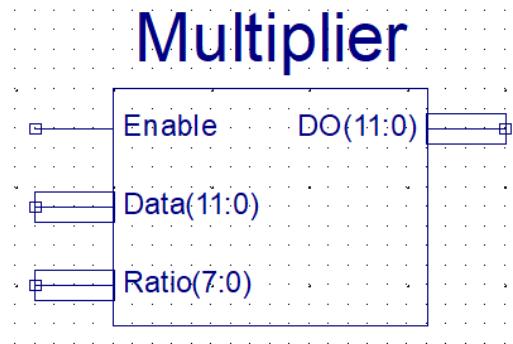
Moduł oscylatora jest zaimplementowany jako dwa połączone liczniki. Zakres pierwszego jest kontrolowany przez 12-bitowy sygnał wejściowy *Freq*, natomiast wartość drugiego, 5-bitowego, służy do wyznaczania przebiegu 12-bitowego sygnału wyjściowego i jest używana jako jego pięć najstarszych bitów.



Rysunek 14: Symulacja modułu Oscillator

Na zrzucie z symulacji widać, że sygnał jest generowany tylko przy podaniu niezerowej wartości na wejście *Freq*. Po powiększeniu symulacji można odczytać długość okresu fali i obliczyć jego częstotliwość, w ten sposób sprawdzić prawidłowość działania układu dla danych parametrów.

2.4.6 Multiplier



Rysunek 15: Schemat modułu Multiplier

Moduł *Multiplier* to układ mnożący bez znaku 12-bitowy sygnał *Data* przez zadany współczynnik. Mnożenie można aktywować podając '1' na wejście *Enable*. Mnożnik *Ratio* jest 8-bitowy w kodzie naturalnym binarnym, stałoprzecinkowy, przy czym przecinek jest między najstarszymi bitami - siódmym i ósmym.

Wejścia

- Enable - wskazuje czy sygnał *Data* powinien być mnożony
- Data(11:0) - sygnał, który ma być mnożony

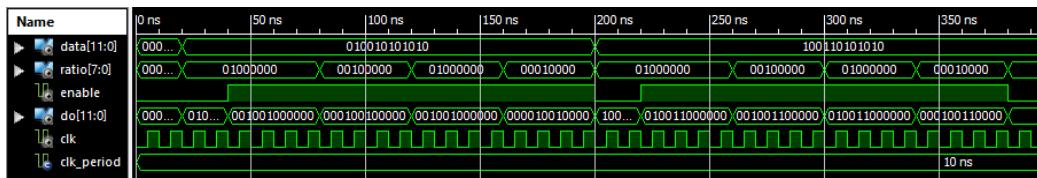
- Ratio(7:0) - współczynnik, przez który mnożymy *Data*

Wyjścia

- DO(11:0) - sygnał wynikowy

```
architecture Behavioral of Multiplier is
    signal result: UNSIGNED(12 downto 0) := (others => '0');
begin
    result <= UNSIGNED(Data(11 downto 7)) * UNSIGNED(Ratio);
    DO <= STD_LOGIC_VECTOR(result(11 downto 0)) when Enable = '1' else
        Data;
end Behavioral;
```

Listing 10: Implementacja modułu Multiplier



Rysunek 16: Symulacja modułu Multiplier

Na symulacji widać prawidłowe wyniki mnożenia dwóch wartości sygnału o kolejno $\frac{1}{4}$, $\frac{1}{2}$ i $\frac{1}{8}$ (przesunięcia bitowe w prawo o 2, 1 i 4) oraz to że sygnał *Enable* wynoszący '0' powoduje tzw. „bypass” sygnału - przestaje on być mnożony.

3 Implementacja

3.1 Rozmiar

Całkowity rozmiar, jaki zajmuje program w formacie .bit to 278 KB.

3.1.1 Zajętość LUT

- Ogólna liczba LUT: 397 z 9312 (4%)
- Użyte jako logika: 378
- Użyte jako route-thru: 19

3.1.2 Plastry

- Liczba plastrów przerzutników: 190 z 9312 (2%)
- Ogólna liczba zajetych plastrów: 228 z 4656 (4%)

3.1.3 Pozostałe

- Liczba buforów zegara BUFGMUX: 1 z 24 (4%)
- Liczba układów mnożących MULT18X18SIO: 1 z 20 (5%)

3.2 Prędkość

Minimalny okres: 8.968 ns (maks. częstotliwość - 111.508MHz) w porównaniu do zadanego w UCF okresu zegara 20ns (50MHz).

3.3 Podręcznik użytkownika zaimplementowanego urządzenia

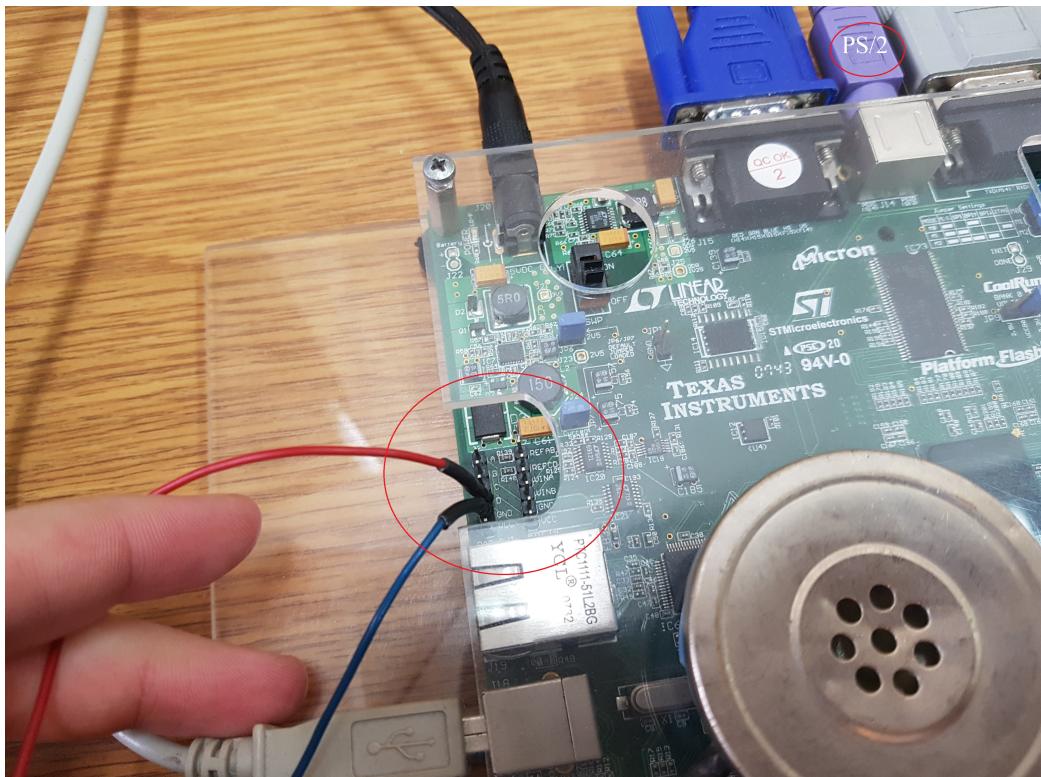
3.3.1 Konfiguracja potrzebnego sprzętu

W celu poprawnego użytkowania układu potrzebne są następujące komponenty:

- płyta spartan 3E
- klawiatura z portem szeregowym PS/2
- głośnik

Po podłączeniu układu FPGA do zasilania należy podpiąć w odpowiednie wejście płyty klawiaturę. Następnie należy podłączyć głośnik do odpowiednich pinów w lewej części płyty zgodnie z rysunkiem załączonym poniżej.

Po poprawnej konfiguracji wszystkich urządzeń należy włączyć płytę przyciśkiem ON w lewym górnym rogu. Ostatni krok obejmuje wgranie zaimplementowanego programu do pamięci płyty.



Rysunek 17: Podłączenie wymaganych urządzeń do płyty

3.3.2 Instrukcja użytkowania

W celu gry na organach należy wciskać wybrane obsługiwane klawisze, których mapowanie do nut przedstawiono poniżej. Dodatkowo wykorzystane są klawisze F1-F4, które odpowiadają za ustawienia obwiedni.

Po wciśnięciu odpowiedniego klawisza przypisanego do danej nuty usłyszmy żądany dźwięk. Nie istnieje mechanizm włączania oraz wyłączania obwiedni, ponieważ można ją wyciszyć poprzez ustawienie długości narastania i wygaszania na *0ms*.

Na przycisku *btn_south* zaimplementowana została możliwość zresetowania całego układu.

Tabela 1: Mapowanie klawiszy pianina do danych nut

dźwięk	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E
klawisz	q	2	w	3	e	r	5	t	6	y	7	u	i	9	o	0	p



Rysunek 18: Mapowanie klawiszy pianina na klawiaturze komputerowej - górna linia: cyfry 1-0, dolna linia: litery Q-P

Tabela 2: Mapowanie klawiszy do obsługi obwiedni

klawisz	funkcja
F1	zwiększenie długości narastania
F2	zmniejszenie długości narastania
F3	zwiększenie długości wygaszania
F4	zmniejszenie długości wygaszania

4 Wykresy fali oraz generowanie pliku wave

4.1 Wstęp

W poniższym rozdziale opisano wszystkie metody jakie wykorzystane zostały do wizualizacji fali i generowania pliku *wave* z wybraną piosenką na podstawie danych uzyskanych z symulacji oraz utworzenia nut zrozumiałych dla symulacji, które odpowiadają wybranym piosenkom.

4.2 Generowanie wykresu fali na podstawie danych z projektu

Z powodu epidemii nie mieliśmy możliwości przetestowania organów w laboratorium i sprawdzenia czy generowane dźwięki mają odpowiednią częstotliwość, dlatego zdecydowaliśmy się zwizualizować uzyskane dane przy pomocy wykresu. Niestety środowisko *ISE* nie pozwala na zobaczenie fali

sygnału, dlatego wykorzystaliśmy w tym celu język *Python* oraz moduł *matplotlib*, który stanowi rozbudowane narzędzie do tworzenia interaktywnych wykresów. Pozwala to odczytać długość fali i na tej podstawie obliczyć jej częstotliwość.

4.2.1 Uzyskanie momentów czasowych oraz wartości sygnału z symulacji

Pierwszym krokiem do wizualizacji sygnału jest uzyskanie odpowiednich danych z symulacji zaimplementowanego modułu. Jako najprostszy przykład, który posłużył nam do sprawdzenia wartości generowanych częstotliwości oraz kształtu fali, wybraliśmy prostą sekwencję czterech dźwięków:

```
SIGNAL seq : VECTOR_VECTOR( 3 DOWNTO 0) := (
    X"15", -- Q - C1
    X"24", -- E - E1
    X"45", -- O - D#2
    X"35" -- Y - A1
);
```

Listing 11: Podstawowa sekwencja dźwięków

Wartości te podawaliśmy kolejno na wejście naszego modułu, symulując działanie klawiatury PS/2 i zapisywaliśmy do pliku aktualną wartość sygnału oraz chwilę czasową gdy wystąpiła tylko wtedy gdy sygnał *DO_Rdy* osiągał wartość 1 (czyli wartość byłaby podawana do przetwornika DAC). Dzięki temu uzyskiwaliśmy w pliku wyłącznie próbki, które wchodzą w skład końcowego zestawu danych reprezentujących sygnał.

Proces odpowiedzialny za przekazywanie zdefiniowanych wartości na wejście modułu został zrealizowany w następujący sposób:

```
stim_proc : process
begin
    signal_loop: for i in 3 downto 0 loop
        DI <= seq(i);
        DI_Rdy <= '1';
        F0 <= '0';
        wait for clk_period;
        DI_Rdy <= '0';
        wait for 200 ms;
        DI_Rdy <= '1';
        F0 <= '1';
        wait for clk_period;
        DI_Rdy <= '0';
        wait for 100 ms;
    end loop signal_loop;
    wait;
end process;
```

Listing 12: Proces przekazujący wartości

Utworzyliśmy także proces, który odpowiadał za zapis interesujących nas wartości do pliku:

```
log_proc : process
    variable v_OLINE : line;
begin
    wait for 10 us;
    file_open(file_RESULTS, "output_results.txt", write_mode);

    signal_loop: while now < 1201 ms loop
        wait until DO_Rdy = '1';
        write(v_OLINE, DO);
        write(v_OLINE, "," & time'image(now));
        writeline(file_RESULTS, v_OLINE);
    end loop signal_loop;

    file_close(file_RESULTS);
    wait;
end process;
```

Listing 13: Proces odpowiedzialny za zapis wartości do pliku

Ograniczenie w pętli oznacza długość „nagrywania” próbek do pliku. Po skończonej symulacji otrzymujemy plik w postaci:

```
000000000000,180195 ns  
000000000110,210225 ns  
000000000111,240255 ns  
000000001000,270285 ns  
(...)
```

Listing 14: Fragment pliku z próbками z symulacji

Uzyskane dane, które znajdują się w pliku *output_results.txt* przetwarzamy z wykorzystaniem skryptu, który dokonuje odpowiedniego formatowania oraz konwersji. Należy pamiętać, że format danych w pliku dla czasu wyrażony jest w nanosekundach natomiast wartości amplitudy podane są w kodzie naturalnym binarnym.

```

import matplotlib.pyplot as plt
import numpy as np
import matplotlib.ticker as ticker
import csv

def to_hex(x, pos):
    """ Used to display signal values in hex on plot"""
    return "%x" % int(x)

time_list, value_list = [], []

with open("output_results.txt") as f:
    for value, time in csv.reader(f, delimiter=","):
        # time in file is stored in nanoseconds as 30035 ns so strip
        # the suffix and convert to ms
        time_in_ms = int(time[:-3]) * 1e-6
        time_list.append(time_in_ms)
        # values are stored in normal binary
        value_list.append(int(value.strip(), 2))

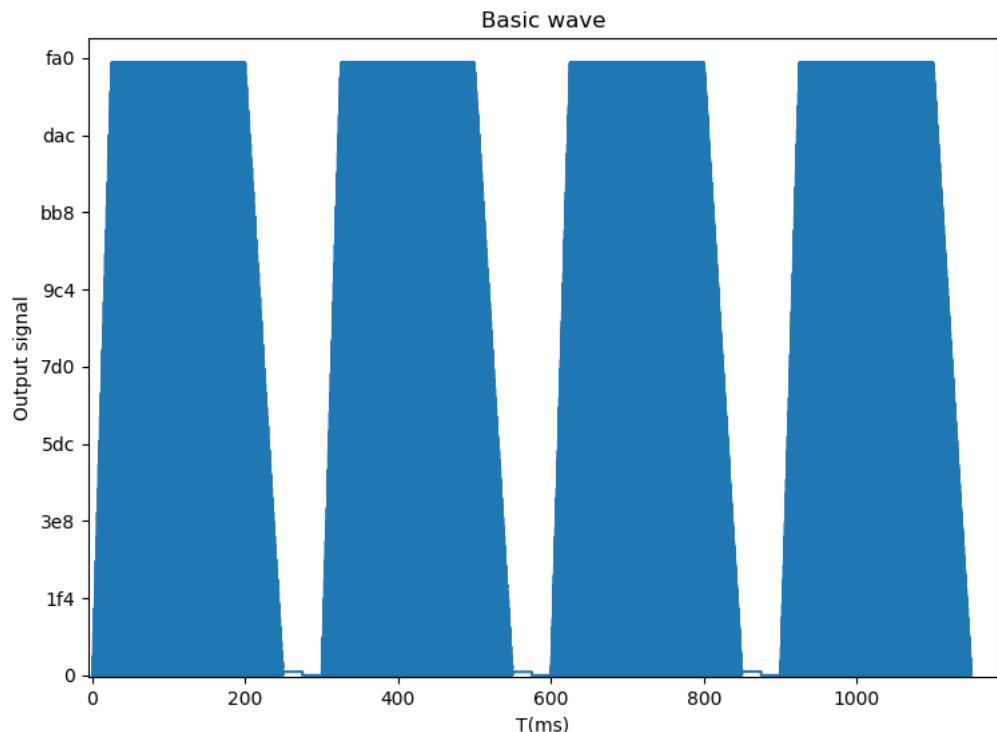
x = np.array(time_list) # plot accepts numpy arrays so convert lists
y = np.array(value_list)
plt.plot(x, y, drawstyle="steps-mid")
ax = plt.gca()
fmt = ticker.FuncFormatter(to_hex)
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_formatter(fmt)
ax.set_xlabel("T(ms)")
ax.set_ylabel("Output signal")
ax.set_title("Basic wave")
plt.show()

```

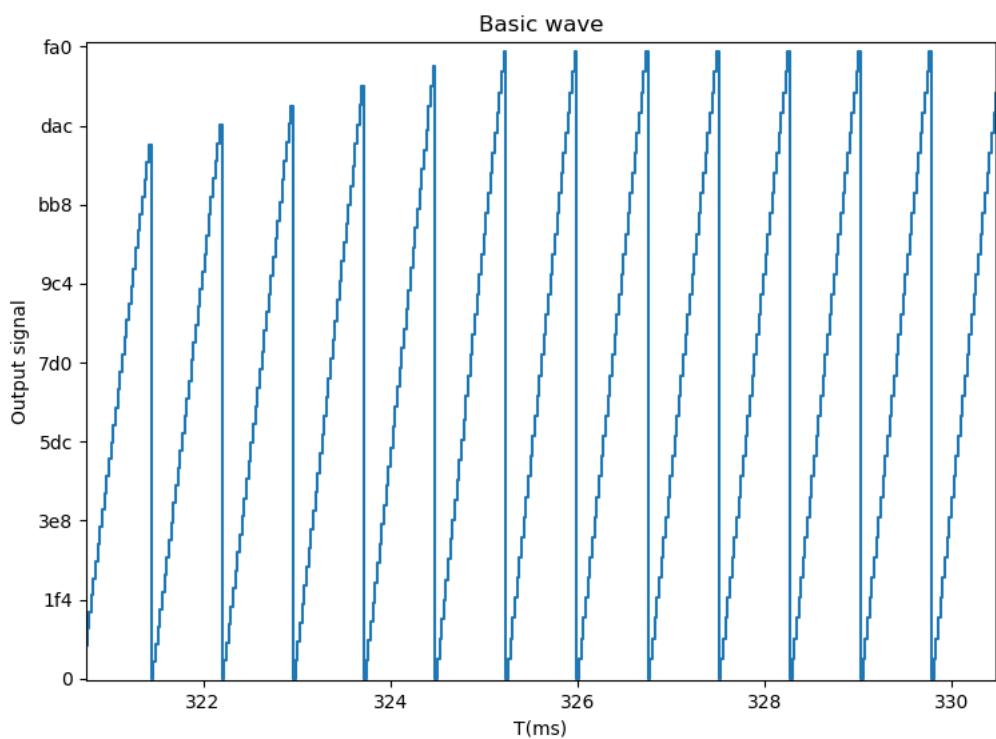
Listing 15: Skrypt w języku Python odpowiedzialny za wygenerowanie wykresu

Po wykonaniu skryptu, otrzymujemy wynik analogiczny do zaprezentowanego poniżej. Dużą zaletą biblioteki *matplotlib* jest tworzenie interaktywnych wykresów, dzięki czemu mogliśmy przybliżać oraz oddalać dane części w celu identyfikacji potencjalnych błędów oraz sprawdzenia wartości brzegowych.

Na poniższych wykresach widać, że uzyskane wartości zgadzają się z założeniami oraz wyraźnie widoczna w postaci schodków jest rozdzielcość zakodowanej cyfrowo fali, wygenerowanej przez licznik.



(a) Wykres całego przebiegu



(b) Przybliżenie przebiegu fali

Rysunek 19: Wygenerowane wykresy fali

4.3 Generowanie danych dla zadanej piosenki

Kolejnym rozszerzeniem jakie dodaliśmy do projektu była możliwość wygenerowania sekwencji wciskanych klawiszy w celu zagrania wybranej melodie na zaimplementowanym instrumencie. Aby zrealizować tą funkcjonalność rozszerzyliśmy dotychczasową symulację wykorzystaną w poprzednim przykładzie o możliwość odczytu sekwencji klawiszy wraz z czasem ich naciśnięcia/zwolnienia z pliku.

4.3.1 Skrypt odpowiedzialny za przełożenie nut na wartości wejściowe modułu generującego falę

Aby umożliwić granie dowolnej melodii, postanowiliśmy utworzyć prosty skrypt, który z zadanej zestawu nut utworzy sekwencję odpowiadającą wciskaniu/puszczeniu danych klawiszy na klawiaturze. Poniższy przykład został stworzony w oparciu o sekwencję nut MIDI odtworzoną w programie FL Studio tak jak na obrazie załączonym poniżej:



Rysunek 20: Przykład ciągu nut MIDI w programie FL Studio dla głównej melodii z utworu Avicii - Levels

Pozwoliło to odsłuchać melodię, aby upewnić się, że została dobrze odwzorowana oraz w czytelny sposób zwizualizować jej dźwięki w dziedzinie wysokości oraz czasu. Nuty rozmieszczone są w rozdzielczości ósemki, co dla tempa piosenki 128 uderzeń na minutę daje $\frac{1}{256}$ minuty. Za pomocą tych informacji tworzymy następującą tabelę sekwencji klawiszy, gdzie każdy rekord przechowuje informację, kiedy dany klawisz został wciśnięty oraz puszczyony. Wiersze z samymi wartościami *None* oznaczają ciszę natomiast pozostałe wartości wypełnione zostaną chwilą w której klawisz puszczaemy/naciskamy.

```

output = [
    [[Fis1, None, F0], [Fis1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[None, None, None], [None, None, None]],
    [[E1, None, F0], [E1, None, F1]],
    [[Dis1, None, F0], [Dis1, None, F1]],
    [[Dis1, None, F0], [Dis1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[None, None, None], [None, None, None]],
    [[Cis2, None, F0], [Cis2, None, F1]],
    [[B1, None, F0], [B1, None, F1]],
    [[Gis1, None, F0], [Gis1, None, F1]],
    [[Fis1, None, F0], [Fis1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[None, None, None], [None, None, None]],
    [[E1, None, F0], [E1, None, F1]],
    [[Dis1, None, F0], [Dis1, None, F1]],
    [[Dis1, None, F0], [Dis1, None, F1]],
    [[Cis1, None, F0], [Cis1, None, F1]],
    [[Cis1, None, F0], [Cis1, None, F1]],
    [[None, None, None], [None, None, None]],
    [[Cis2, None, F0], [Cis2, None, F1]],
    [[B1, None, F0], [B1, None, F1]],
    [[Gis1, None, F0], [Gis1, None, F1]],
]

```

Listing 16: Tablica nut dla generowanej piosenki

```

# Mapping of key numbers to notes
Fis1 = "00101110"
E1 = "00100100"
Dis1 = "00100110"
Cis2 = "01000110"
B1 = "00111100"
Cis1 = "00011110"
Gis1 = "00110110"

F0 = 0 # press the key
F1 = 1 # release the key

# output looks as follow:
# [[hexkeycode, start_timestamp, F0], [hexkeycode, end_timestamp, F0]]
# where:
# hexkeycode - hex value of a PS/2 scancode of the key which was pressed/released
# timestamp - time when it happened in ns
# F0 - 1 if key is being released, 0 otherwise (PS/2 standard)
# All None records are just for easier time calculation and will not be included

# Skipped some notes for clarity
output = [
    [[Fis1, None, F0], [Fis1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
    [[E1, None, F0], [E1, None, F1]],
]

time = 1e6 # start time in nano seconds (equals 1ms)

step = 1 / 256 # 1/256 of a minute

for record in output:
    if record[0][0]:
        record[0][1] = f"{{round(time)}}ns" # start time when the key is pressed
        release_time = round(time + 3 / 4 * step * 60 * 1e9) # release time in ns
        record[1][1] = f"{{release_time}}ns"
        time += round(step * 60 * 1e9)

with open("levels.txt", mode="w") as file:
    for record in output:
        if record[0][0]:
            start_row_str = " ".join(str(e) for e in record[0])
            end_row_str = " ".join(str(e) for e in record[1])
            file.write(f"{{start_row_str}}\n")
            file.write(f"{{end_row_str}}\n")

```

Listing 17: Skrypt odpowiedzialny za generowanie wejścia dla modułu symulacji

Po wykonaniu skryptu otrzymujemy plik w którym każdy wiersz odpowiada wciśnięciu lub puszczeniu klawisza wraz z jego kodem i czasem wystąpienia.

```
00101110 1000000ns 0  
00101110 176781250ns 1  
00100100 235375000ns 0  
00100100 411156250ns 1  
...
```

Listing 18: Fragment pliku z wygenerowanymi danymi

4.3.2 Wykorzystanie wygenerowanych danych w symulacji

Tak przygotowane dane wczytywaliśmy bezpośrednio do symulacji i podawaliśmy na wejścia odpowiednich portów. We wcześniej wykorzystanym pliku symulacyjnym zaimplementowaliśmy nowy proces odpowiedzialny za odczyt wygenerowanych danych z pliku.

Wykorzystując wbudowane funkcje w języku *VHDL* udało nam się z łatwością odczytać dane i przekazać je na odpowiednie porty.

```

read_proc : process
    variable v_ILINE      : line;
    variable v_keycode   : STD_LOGIC_VECTOR(7 downto 0);
    variable v_time       : time;
    variable v_released  : STD_LOGIC;
    variable v_SPACE      : character;
begin
    wait for 10 us;
    file_open(file_INPUT, "levels.txt", read_mode);

    while not endfile(file_INPUT) loop
        readline(file_INPUT, v_ILINE);
        read(v_ILINE, v_keycode);
        read(v_ILINE, v_time);
        read(v_ILINE, v_released);

        DI <= v_keycode;
        F0 <= v_released;
        wait for v_time - now;
        DI_Rdy <= '1';
        wait for clk_period;
        DI_Rdy <= '0';
    end loop;

    file_close(file_INPUT);
    wait;
end process read_proc;

```

Listing 19: Proces odpowiedzialny za odczyt wartości z pliku

4.4 Zamiana uzyskanych danych na plik wave

Ostatnie z zaimplementowanych rozszerzeń obejmowało generowanie poprawnego pliku z piosenką w formacie *wave*. Tak na prawdę wizualizacja fali oraz sprawdzenie odpowiednich wartości częstotliwości była wystarczająca do stwierdzenia, że dany układ działa lecz chcieliśmy usłyszeć czy faktycznie nasz instrument gra poprawnie.

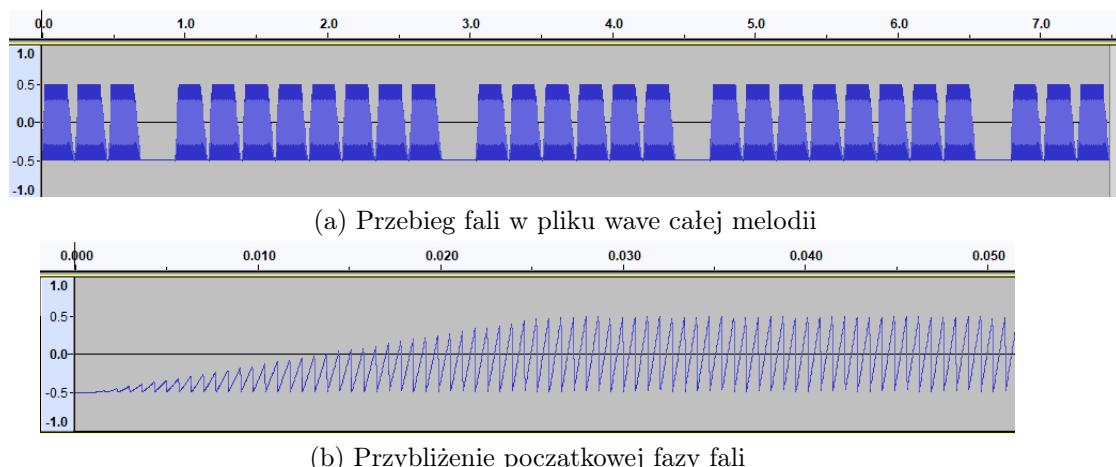
Wyбралиmy format *wave*[7], ponieważ jest on najprostszy w edycji, ze względu na sposób przechowywania danych i brak kompresji. Pliki *wave* przyjmują stałą częstotliwość próbkowania, najczęściej równą 44100 Hz.

Dane, które otrzymaliśmy poprzednio z symulacji były wystarczające do wygenerowania serii dźwięków granych na organach. Jedyny problem jaki napotkaliśmy to brak jednostajnego rozmieszczenia uzyskanych próbek w czasie w związku z czym nie mogliśmy bezpośrednio próbować sygnału, tylko mu-

sieliśmy dokonać jego interpolacji, czyli obliczyć na jego podstawie zestaw nowych próbek w regularnych odstępach czasowych z docelową częstotliwością próbkowania 44100 Hz.

Do wygenerowania pliku *wave* posłużyliśmy się gotowym modułem dostępnym w języku Python. Poniżej zamieszczono pełny listing skryptu wraz z komentarzami tłumaczącymi dane kroki.

Dodatkowo sprawdziliśmy wygenerowany plik *wave* w darmowym programie Audacity[8], który oprócz mnóstwa funkcji związanych z obróbką dźwięku, pozwala na podejrzenie fali oraz obliczenie widma częstotliwościowego zaznaczonego fragmentu, dzięki czemu mieliśmy pewność, że uzyskane dane są poprawne, co można było również usłyszeć poprzez odtworzenie pliku.



Rysunek 21: Podgląd przebiegu fali w wygenerowanym pliku *wave* w programie Audacity

```

import wave
import struct
import csv
import numpy as np

sample_rate = 44100 # sample rate for wave file

signal_file = wave.open("signal.wav", "w")
signal_file.setparams((1, 2, sample_rate, 0, "NONE", "not compressed"))
# one channel (mono)
# sample width in bytes (shorts)
# sample rate
# number of frames in whole file (changed automatically later when writing)
# none compression
# compression description

time_list, value_list = [], []

with open("output_results.txt") as f:
    for value, time in csv.reader(f, delimiter=","):
        # time in file is stored in nanoseconds as 30035 ns so strip
        # the suffix and convert to ms
        time_in_ms = int(time[:-3]) * 1e-6
        time_list.append(time_in_ms)
        # values are stored in normal binary
        value_list.append(int(value.strip(), 2))

# step based on which we generate time points in which we sample the signal
step = 1 / sample_rate * 1000 # convert to miliseconds

# create a vector of time points in which we will sample the data
x = np.arange(0, time_list[-1], step)

# interpolation of the signal in order to properly sample the data
interpolated = np.interp(x, time_list, value_list)

# normalize the data
interpolated /= np.max(np.abs(interpolated))

values_to_write = []

for sample in interpolated:
    # pack each value to C struct
    # interpolated values are floats so convert them to ints with proper
    # value scaling. This scaling ensures that values in output wave file
    # are in -0.5 +0.5 range because we use normal binary to represent values.
    # we multiply by 2^15 because we pack our samples into shorts (h)
    packed_value = struct.pack("h", int(sample * (2 ** 15 - 1) - 2 ** 14))
    values_to_write.append(packed_value)

value_str = b"".join(values_to_write) # convert them to one long byte stream

signal_file.writeframes(value_str) # save to file
signal_file.close()

```

Listing 20: Skrypt odpowiedzialny za utworzenie pliku wave z danych symulacji

5 Podsumowanie

5.1 Ocena krytyczna

Całość implementacji przebiegała sprawnie i nie mieliśmy większych problemów z utworzeniem danych modułów.

Jednym aspektem, który nie został w pełni zrealizowanym była ilość efektów pozwalających na modulację granego dźwięku. Obwiednia pozwala na znaczne modyfikacje lecz istnieją również inne efekty, które warto byłoby zaimplementować, tak jak na przykład: vibrato, tremolo, echo.

5.2 Ocena pracy

Zrealizowaliśmy wszystkie z założeń podjętych na początku semestru oraz dodatkowo napisaliśmy skrypty, które rozszerzają zakres całego projektu oraz pozwalają na jego znaczną rozbudowę. Wszystkie z zaimplementowanych modułów syntezują się bez niepożądanych ostrzeżeń oraz działają poprawnie.

W projekcie wykorzystaliśmy w pełni wiedzę zdobytą na laboratoriach z UCiSW 1 i dodatkowo opanowaliśmy obsługę plików w języku VHDL. Przydatna była również podstawowa wiedza z dziedziny przetwarzania sygnałów, którą wykorzystaliśmy do poprawnej obróbki danych oraz wygenerowania prawidłowego pliku *wave*.

Wykresy i możliwość grania dowolnej piosenki, stanowią ciekawe urozmaicenie dla całego projektu i są też dobrymi wyznacznikami poprawności jego działania. Wykorzystaliśmy możliwość wizualizacji fali podczas przerabiania układu do sprawdzenia czy dane zmiany w kodzie VHDL nie wpłynęły negatywnie na działanie całego projektu.

5.3 Możliwe kierunki rozbudowy układu

Projekt został zbudowany w taki sposób, że pozwala na łatwą integrację z większą liczbą modułów. Przykładowo do projektu dodać można inne efekty dźwiękowe, obsługę polifonii lub wyświetlanie wciskanych klawiszy, bądź aktualnych wartości parametrów obwiedni na ekranie.

6 Literatura

- [1] Spartan-3E Starter Kit Board User Guide
https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf
- [2] Opracowanie modułów dostępnych na płycie, Dr inż. Jarosław Sugier
http://www.zsk.ict.pwr.wroc.pl/zsk_ftp/fpga/
- [3] Częstotliwości poszczególnych nut, data dostępu: 20.05.2020
https://en.wikipedia.org/wiki/Piano_key_frequencies
- [4] Opis działania oraz modele obwiedni dźwięku, data dostępu: 20.05.2020
https://pl.wikipedia.org/wiki/Generator_obwiedni
- [5] Dokumentacja biblioteki matplotlib do języka Python, data dostępu:
20.05.2020
<https://matplotlib.org/contents.html>
- [6] Dokumentacja bibliotek numpy i scipy do języka Python, data dostępu:
20.05.2020
<https://docs.scipy.org/doc/>
- [7] Dokumentacja formatu plików dźwiękowych WAVE, data dostępu:
20.05.2020
<http://soundfile.sapp.org/doc/WaveFormat/>
- [8] Strona programu Audacity, data dostępu: 20.05.2020
<https://www.audacityteam.org/>