Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Programowanie kładów FPGA

(projekt)

Implementacja algorytmów cyfrowego przetwarzania sygnałów audio na bazie układu FPGA z interfejsem UART

Pierczyk Krzysztof

Warszawa, 31 maja 2021

Spis treści

I Analiza teoretyczna 3

1 Wstęp 4

2 Analiza interfejsu komunikacyjnego 4
2.1 Koncepcja implementacji 6

3 Efekty dźwiękowe 7
3.1 Overdrive 8
3.2 Delay 8
3.3 Flanger 9
3.4 Tremolo 10

4 Interfejs użytkownika 10

5 Planowane symulacje 11

6 Planowane testy 11

7 Wybór platformy 11

II Implementacja 13

1 Struktura projektu 14

2 Interfejs komunikacyjny 16

3 Interfejs analogowy 20

4 Efekt *overdrive* 23

5 Efekt tremolo 25

6 Efekt pogłosu 27

7 Efekt *flanger* 29

8 Integracja potoku 31

9 Integracja systemu 32

10 Podsumowanie 33

Część I.

Analiza teoretyczna

1. Wstęp

Celem projektu jest opracowanie i zaimplementowanie zestawu wybranych metod przetwarzania sygnałów audio znanych z popularnych multiefektów gitarowych. Zadaniem tego typu rozwiązań jest modyfikowanie próbkowanego dźwięku w czasie rzeczywistym w taki sposób, aby urozmaicić jego brzmienie np. poprzez modulację, przesunięcie fazowe lub wprowadzenie dodatkowych składowych. Przykładami takich efektów są m.in.

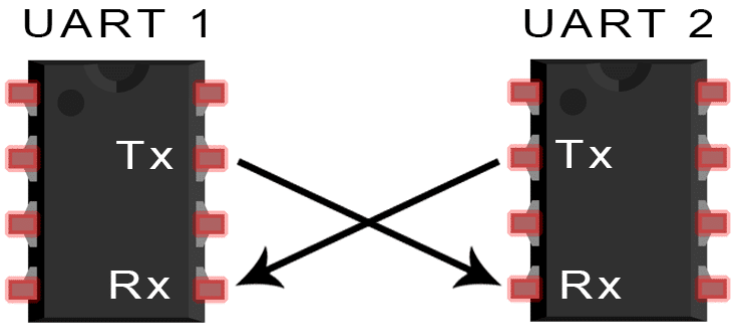
- **echo** (opóźnienie, ang. *delay*) - do sygnału dodawana jest jedna lub kilka jego kopii opóźnionych o określoną liczbę próbek; efekt ma symulować warunki panujące w halach widowiskowych
- **overdrive** - nazwa ogółu metod prowadzących do znacznego zniekształcenia sygnału bazowego; jednym z popularnych sposobów jego implementacji jest nałożenie obustronnych ograniczeń na wyjściowe wartości przepuszczanego sygnału
- **flanger** - kolejny efekt wykorzystujący opóźnione próbki sygnału; w tym przypadku wielkość opóźnienia podlega cyklicznym zmianom, co przekłada się na pulsacyjny charakter wyjściowego dźwięku
- **tremolo** - efekt modulujący amplitudę sygnału zgodnie z przebiegiem pewnej funkcji okresowej (np. sinus lub fala trójkątna); jego celem jest symulowanie rodzaju artykulacji polegającego na szybkim wydobywaniu dźwięków o tej samej częstotliwości (np. poprzez szybkie szarpanie pojedynczej struny gitarowej)

Powyższe efekty stanowią jedynie niewielki wycinek stosowanych rozwiązań, wśród których wymienić można także szeroko pojęte metody equalizacji, czy modyfikowania częstotliwości sygnału. Minimalna wersja projektu zakłada implementację scharakteryzowanych metod przetwarzania wraz z prostymi mechanizmami wprowadzania i wyprowadzania danych z urządzenia a także dostosowywania parametrów filtrów. Jako metodę komunikacji wybrano popularny (choć może w niec innych zastosowaniach) interfejs UART (ang. *universal asynchronous receiver-transmitter*). Jego prostota umożliwi przyspieszenie procesu implementacji, a co za tym idzie szybsze przejście do zasadniczej części projektu. Cyfrowy charakter UARTa pozwoli w przyszłości przejść na popularny interfejs I^2S, dzięki któremu możliwe będzie proste dołączenie do urządzenia układów przetwornikowych. Testowanie urządzenia obywać się będzie z pomocą prostej aplikacji w języku Python, która za pośrednictwem wirtualnego portu szeregowego wysyłać będzie do urządzenia próbki dźwięku. Sygnał wychodzący z układu FPGA będzie następnie odtwarzany za pomocą jednej z wielu dostepnych w Pythonie bibliotek audio jak np. `pyaudio`.

2. Analiza interfejsu komunikacyjnego

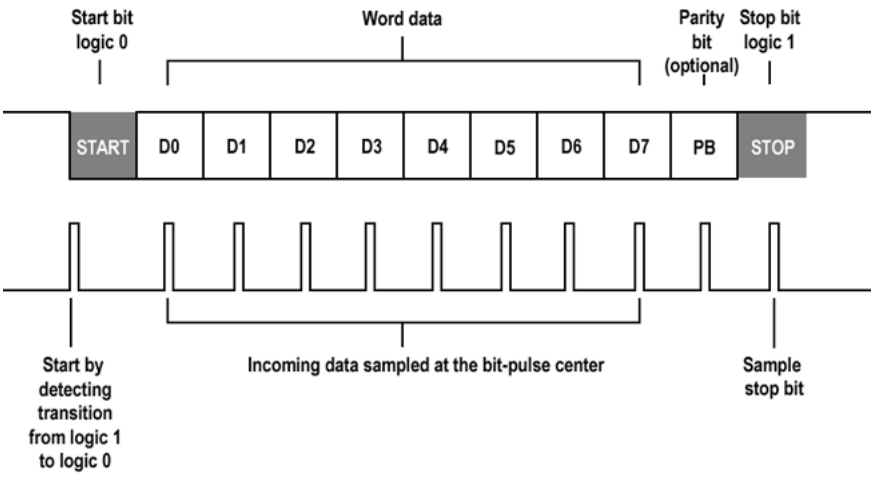
Interfejs UART jest dzisiaj dostępny w niemal wszystkich obecnych na rynku mikrokontrolerach oraz w wielu układach typu SoC. Jego popularność wynika zarówno z (jak sama nazwa wskazuje) uniwersalnego charakteru jak i prostoty implementacji. UART to cyfrowe urządzenie peryferyjne umożliwiające szeregową komunikację asynchroniczną. W większości implementacji parametry komunikacji takie jak szybkość, czy format danych mogą być konfigurowane poprzez zmianę wartości odpowiednich rejestrów sterujących. Nierzadko możliwe jest też ustawienie trybu komunikacji spośród *simplex*, *duplex* lub *half-duplex*.

Interfejsy tego typu, szczególnie w zastosowaniach przemysłowych, są często sprzęgane z konwerterami poziomów logicznych odpowiednich dla standardów RS-232 lub RS-485.



Rysunek 1: Typowa struktura komunikacji dwóch węzłów z wykorzystaniem układu UART, źródło: [1]

Komunikacja asynchroniczna wymaga aby wszystkie węzły nadawały i odbierały dane o z góry ustalonym formacie i długości znaku (wynikającym z szybkości transmisji). Ponadto należy wziąć pod uwagę, że zegary obecne w poszczególnych urządzeniach mogą się z czasem rozsynchronizowywać, a co za tym idzie konieczny jest mechanizm ponownej synchronizacji. W przypadku komunikacji z wykorzystaniem modułu UART mechanizm ten wynika z formatu przesyłanych danych. Typowa ramka składa się z trzech elementów: **bitu startu**, **bitów danych** oraz **bitów stopu**. Bit startu oznacza początek nowej ramki i jest sygnalizowany zmianą stanu linii ze spoczynkowego na aktywny. Po nim następuje pewna liczba bitów danych - zazwyczaj 7 lub 8 - a na końcu jeden lub dwa bity stopu. Bit startu odpowiada za synchronizację zegarów wykorzystywanych do próbkowania stanu linii, natomiast bity stopu definiują minimalną przerwę między kolejnymi ramkami. Fakt że każdy bit startu to ponowna okazja do zsynchronizowania zegarów sprawia, że nie muszą one pracować z dokładnie tymi samymi szybkościami. Niewielkie różnice nie powodują błędów w odbiorze danych.



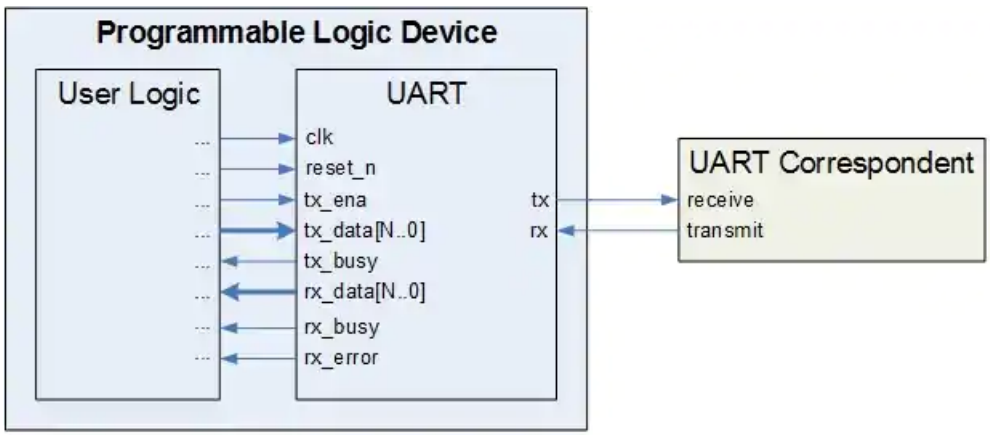
Rysunek 2: Struktura ramki UART, źródło: [2]

Format ramki może zostać rozszerzony o element kontrolny w postaci **bitu parzystości**. Jeśli występuje, przyjmuje on wartość zależną od ilości bitów w stanie wysokim w przesyłanych danych i znajduje się przed bitami stopu. Możliwy jest bit parzystości (ang. *even*) - ustawiony, gdy ilość ta jest parzysta - lub nieparzystości (ang. *odd*) - ustawiany w przypadku przeciwnym. Dodatkowy element pozwala wykrywać ewentualne błędy transmisji. Format ramki często oznacza się w postaci trzyznakowego identyfikatora postaci DPS, gdzie D oznacza ilość bitów danych, P - typ bitu kontrolnego (E - bit parzystości, O - bit nieparzystości, N - brak bitu kontrolnego) a S ilość bitów stopu. Typowowymi prędkościami transmisji przez UART są:

- 9600 bit/s
- 19200 bit/s
- 38400 bit/s
- ...

Jest to pewna zaszłość historyczna wynikająca z częstotliwości standardowych oscylatorów dostępnych na rynku. Moduły współcześnie implementowane w układach scalonych wzbogacone są często o dodatkowe wyprowadzenia zegarowe umożliwiające komunikację synchroniczną. Tego typu urządzenia określane są zazwyczaj mianem USART (ang. *universal synchronous asynchronous receiver-transmitter*).

2.1. Koncepcja implementacji



Rysunek 3: Przykładowa struktura zewnętrzna bloku UART, źródło: [3]

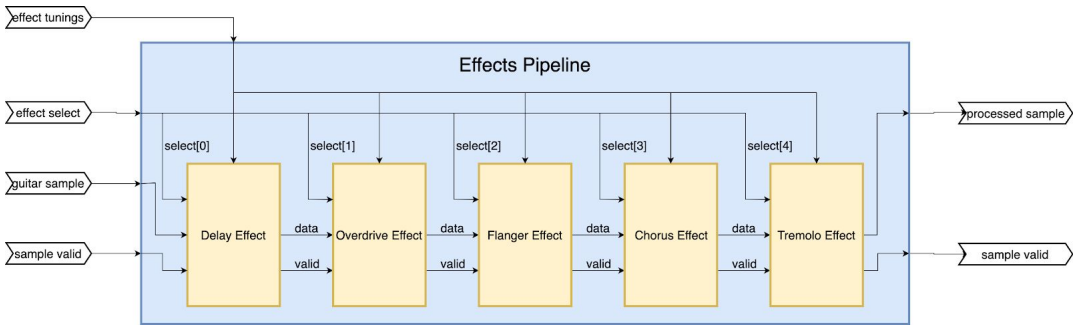
W pełni funkcjonalny układ UART można podzielić na trzy zasadnicze części: **generator sygnału taktującego**, moduł **nadawczy** oraz moduł **odbiorczy**. Pierwsza z nich odpowiada za wytworzenie dwóch sygnałów zegarowych - jednego o częstotliwości równej częstotliwości transmisji (ang. *baud ratate*) i drugiego - taktowanego z szybkością typowo ośmio- lub szesnastokrotnie większą (ang. *oversampling rate*). Sygnał wolniejszy wykorzystywany jest przez moduł nadawczy do określania momentów transmisji kolejnych bitów. Z kolei sygnał szybszy determinuje chwile próbkowania linii RX przez moduł odbiorczy. Skonstruowanie takiego generatora wymaga dwóch liczników oraz dwóch komparatorów.

Potrzebne są również dwa rejestry przechowujące stosunki częstotliwości zegara systemowego do częstotliwości obu sygnałów generowanych. Rejestry te w projekcie będą przechowywały wartości stałe, ponieważ zmienna szybkość transmisji nie jest wymagana.

Moduły nadawczy oraz odbiorczy stanowią dwóstanowe, synchroniczne automaty. Mogą się one znajdować w stanie aktywnym (nadawanie/odbieranie) lub w stanie bezczynnym (ang. *idle*). Do nadawania wykorzystywany jest prosty rejestr przesówny, którego wyjście podłączone jest do linii TX natomiast wejście zegarowe (jak napisano wyżej) taktowane jest sygnałem *baud*. Zamknięcie danych w rejestrze oraz obliczenie bitu kontrolnego następuje po podaniu odpowiedniego stanu na dedykowanej linii wejściowej układu (TX_EN). Analogiczna sytuacja zachodzi w przypadku odbioru danych, przy czym sygnałem skutkującym przejściem moduł odbiorczego w stan aktywny jest pojawienie się bitu startu na linii RX. Jest ona próbkowana z częstotliwością równą częstotliwości sygnału *oversampling*. Stan odbieranego bitu określany jest w momencie połowy jego trwania na linii odbiorczej. Po zarejestrowaniu odpowiedniej liczby bitów sprawdzana jest parzystość danych oraz poprawność bitów stopu, po czym zakończenie danych sygnalizowe jest poprzez odpowiednią linię wyjściową układu (wraz z ewentualnym ustawieniem linii błędu). Przykładowa struktura interfejsu urządzenia UART (wykorzystana w tym projekcie) została przedstawiona na Rys. 3

3. Efekty dźwiękowe

Pierwszą decyzją projektową dotyczącą efektów było stworzenie jednolitego interfejsu implementowanych bloków przetwarzających. Ma to umożliwić arbitralne połączenie ich w potok oraz dodanie w przyszłości nowych efektów bez wprowadzania znacznych zmian w projekcie. Wykorzystano w tym celu strukturę zaczerpniętą z [4], która została przedstawiona na Rys.4.



Rysunek 4: Planowana struktura potoku efektów, źródło: [4]

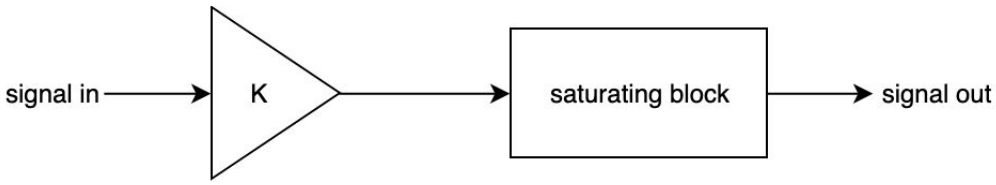
Każdy blok posiada trzy standardowe wejścia oraz dwa standardowe wyjścia. Projekt zakłada wykorzystanie 16-bitowych próbek dźwięku, co determinuje szerokość szyn danych. Wejścia `valid` aktywowane są zboczem narastajacym i oznaczają pojawienie się nowej próbki na wejściu bloku. Po przetworzeniu próbki moduł ma obowiązek wystawienia danych na linię wyjściową oraz wygenerowanie zbocza narastającego na wyjściu `valid`, które podłączone jest do odpowiadającego wejścia następnego modułu. Każdy z bloków posiada także jednobitowe wejście `enable`. Stan niski na tej linii oznacza, że blok powinien przekazywać na swoje wyjście próbkę wejściową bez jej modyfikowania. Każdy blok

może dodatkowo implementować arbitralne wejścia konfiguracyjne specyficzne dla działania danego algorytmu. W przypadku bloku *overdrive* może być to na przykład 8-bitowa wartość wzmocnienia sygnału i 16-bitowe wartości górnego i dolnego nasycenia (szczegóły opisano w dalszej części dokumentu).

Tak zaprojektowana struktura pozwala w łatwy sposób modyfikować obecne w systemie efekty oraz nie nakłada ścisłych ograniczeń na interfejs użytkownika wykorzystywany do ich kontrolowania. Również sposób dostarczania i odbierania danych z potoku nie jest dzięki temu ograniczony do konkretnego interfejsu komunikacyjnego.

3.1. Overdrive

Jak nakreślono we wstępie, efekt przesterowania (ang. *overdive*) określa zespół metod prowadzących do znacznego zniekształcenia sygnału bazowego poprzez dodanie do niego składowych harmonicznych. Częstym sposobem implementacji takiego efektu jest obustronne przycinanie sygnału wejściowego. W niniejszym projekcie zastosowana zostanie struktura przedstawiona na Rys. 5.

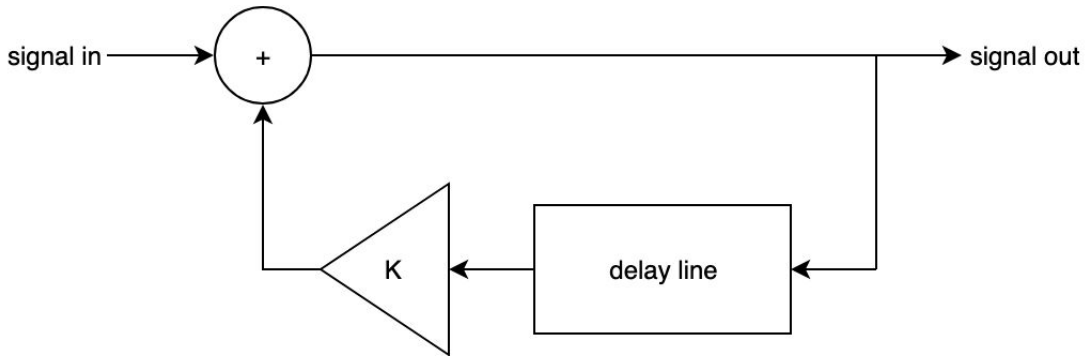


Rysunek 5: Schemat bloku *overdrive*, źródło: [4]

Pierwszym etapem przetwarzania jest wzmocnienie sygnały wejściowego o wartość kontrolowaną przez rejestr wejsciowy bloku. Zakres potencjalnego wzmocnienia zostanie ustalony na etapie testowania układu. Będzie ono realizowane poprzez 16-bitowe mnożenie z nasyceniem. Wzmocniony sygnał przejdzie następnie przez blok nasycenia o zmiennym poziomie, który zostanie zrealizowany przy użycia dwóch 16-bitowych komparatorów oraz multipleksera.

3.2. Delay

Efekt pogłosu uzyskiwany jest poprzez sumowanie przychodzących próbek z próbkami opóźnionymi o określoną liczbę cykli. Realizacja takiego bloku może bazować na filtrze typu FIR (ang. *Finite Impulse Response*) lub IIR (ang. *Infinite Impulse Response*). W przypadku pierwszego z nich przeszłe próbki są opóźnionymi wersjami **sygnału wejściowego**. Liczba próbek sumowanych może być stała lub parametryzowana. Drugi sposób implementacji przewiduje sumowanie próbki wejściowej z pojedynczą próbką opóźnioną sygnału. Jest ona jednak pobierana z **wyjścia układu**, co oznacza, że zależna jest od wszystkich poprzednich próbek sygnału. Właśnie to rozwiązanie zostanie zastosowane w niniejszym projekcie. Jego struktura została przedstawiona na Rys.6. W celu uzyskania stabilnego układu konieczne jest wprowadzenia tłumienia składowej opóźnionej. Kierując się informacjami zawartymi w [4] wartości tego tłumienia przyjęto w zakresie od 0 do 0.5. Jego realizacja będzie wymagała przesunięcia wyniku mnożenia o ilość bitów o jeden większą od szerokości współczynnika K.

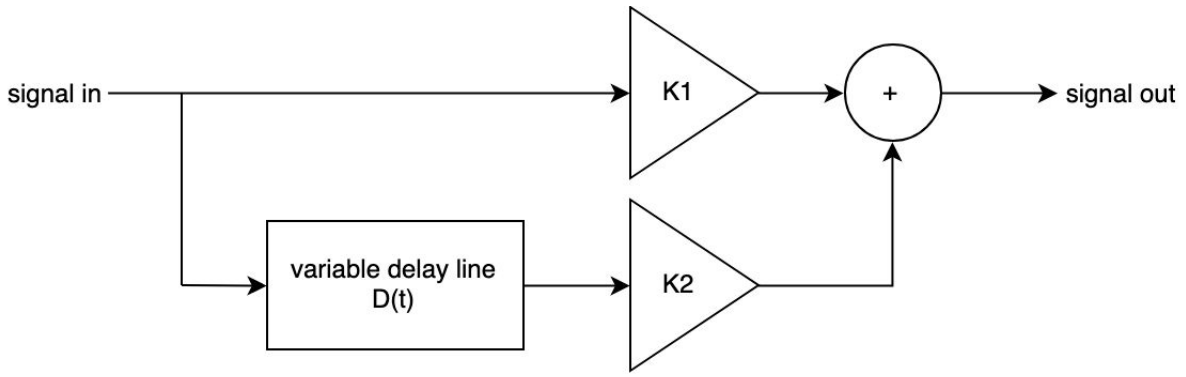


Rysunek 6: Schemat bloku *delay*, źródło: [4]

Od strony technicznej implementacyja takiego bloku wymaga układu mnożącego, kolejki typu FIFO oraz multipleksera. Siła tłumienia echa ustalana jest poprzez wartość jednego z wejść do układu mnożącego. Głębokość echa definiuje z kolei indeks rejestru w kolejce, którego wartość wystawiana jest na wyjście modułu opóźniającego. Oba parametry efektu regulowane będą przez rejestry wejściowe bloku. Maksymalna głębokość kolejki zostanie ustalona na etapie testowania.

3.3. Flanger

Flagner jest efektem, którego brzmienie trudno opisać, jednak zasada jego działania jest stosunkowo prosta. Efekt ten powstaje poprzez nałożenie na sygnał filtru grzebieniowego, którego charakterystyka amplitudowa wykonuje sinusoidalne oscylacje wzdłóż osi częstotliwości. W prakyce układ taki realizuje się poprzez sumowanie sygnału z jego opóźnionymi (nieprzetworzonymi) wersjami. Wartość tego opóźnienia jest jednak okresowo zmienna. Struktura takiego rozwiązania przedstawiona została na Rys.7. Aby kontrolować siłę efektu do ukłądu wprowadzone zostaną dodatkowe bloki tłumienia. Ich wartość zawierać się będzie w przedziale od 0 do 1, natomiast suma będzie stale równa 1. Podobnie jak w przypadku pogłosu wykorzystywana głębokość kolejki FIFO (a tym samym efektywna amplituda oscylacji wartości opóźnienia) będzie ustalana poprzez zewnętrzny parametr.



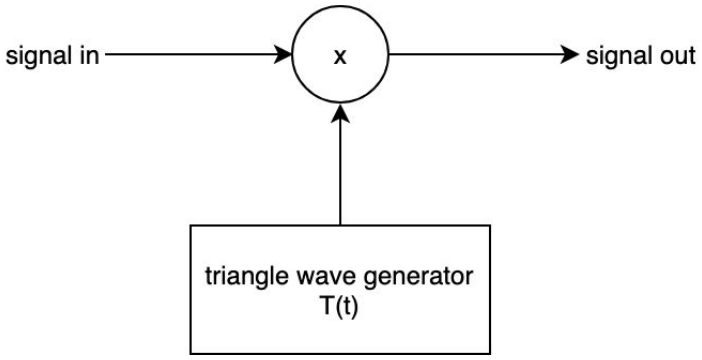
Rysunek 7: Schemat bloku *flanger*, źródło: [4]

Implementacja efektu będzie wykorzystywać bloki stworzone podczas realizacji poprzedniego efektu takie jak układy mnożące oraz kolejka FIFO. Dodatkowym elementem będzie

tutaj generator funkcji sinus o zmiennej częstotliwości. Czestotliwość ta będzie również ustalana poprzez zewnętrzny parametr.

3.4. Tremolo

Efekt tremolo polega na modulacji amplitudy sygnału wejściowego. Jest to realizowane poprzez mnożenie sygnału z pewną funkcją okresową - jak np. sinus lub fala trójkątna - której wartości mieszczą się w przedziale $[0, 1]$. W projekcie zaimplementowane zostaną oba rodzaje modulacji. Po wykonaniu testów wybrany zostanie ten, który będzie pozwalał uzyskać ciekawsze brzmienie. Schemat blokowy rozwiązania został przedstawiony na Rys.8. Podobnie jak w przypadku pogłosu wykorzystane zostanie tu n-bitowe przesówanie wyniku mnożenia celem uzyskania pożądanej amplitudy fali modulującej (przy założeniu, że próbki sygnału modulującego mają szerokość n bitów).



Rysunek 8: Schemat bloku *tremolo*, źródło: [4]

4. Interfejs użytkownika

Ostatnim elementem projektu jest zaimplementowanie wygodnego interfejsu użytkownika. Powinien on umożliwiać niezależną aktywację każdego z efektów oraz pozwalać na regulowanie ich parametrów. Układ FPGA udostępniać będzie cztery wejścia cyfrowe (przyciski). Zostaną one podłączone (za pośrednictwem przerzutników) do wejść `enable` poszczególnych bloków przetwarzających. Aktywacja efektu możliwa będzie dzięki zmianie stanu przełącznika. Kontrola parametrów bloków przetwarzana zostanie zrealizowana za pomocą potencjometrów. Ich interfejs może zostać stworzony na dwa sposoby. Ostateczny wybór zostanie podjęty po głęszbym przeanalizowaniu zagadnienia.

Pierwszym pomysłem jest wykorzystanie modułu XADC od Xilinx. Pozwoliłoby to na podłączenie wszystkich potencjometrów do układu FPGA z wykorzystaniem multipleksera analogowego (np. CD74HC4067), którego wejście przełączane byłoby z pewną częstotliwością przez układ sterujący. Wymagałoby to jednak wykorzystanie gotowego bloku IP oraz stworzenia odpowiedniego interfejsu od strony aplikacji.

Drugim pomysłem jest wdrożenie dodatkowego modułu UART, który komunikowałby się z zewnętrznym mikrokontrolerem realizującym pomiar napięć na potencjometrach poprzez wbudowane kanały przetwornika A/C. W takim przypadku układ FPGA odpytywałby podrzędny względem niego mikrokontroler w sposób cykliczny, pozyskując cyfrowe

wartości orientacji potencjometrów. Niezależnie od wyboru metody cyfrowe wartości pomiarów zostaną podłączone poprzez przerzutniki do wejśc konfiguracyjnych odpowiendich bloków przetwarzających sygnały.

5. Planowane symulacje

Planowane symulacje można podzielić na dwie zasadnicze kategorie. Pierwsza z nich obejmuje proste testy jednostkowe **w skali mikro**. Mowa tu o modułach takich jak generator taktowania dla UARTa, układ mnożący, czy generator fali trójkątnej wykorzystywany w algorytmach przetwarzania sygnału. Każdemu z tych elementów powinien odpowiadać prosty podprojekt typu *testbench*, który weryfikuje poprawność jego działania. Chociaż tworzenie tak drobnych elementów symulacyjnych może być do pewnego stopnia uciążliwe przy większej ilości testowanych elementów, to jednak doświadczenie pochodzące z programowania pokazuje, że pozwala to zapobiec eskalacji wpływu drobnych błędów na działanie systemu jako całości.

Drugą kategorią symulacji będą testy systemowe sprawdzające **makroskopowe** działanie poszczególnych bloków funkcjonalnych. Tutaj również każdy z modułów powinien otrzymać dedykowany projekt testowy. W tym przypadku poza poprawnością działania podsystemów zostanie również zweryfikowana ich wydajność. Oszacowanie czasów przetwarzania danych przez dany blok funkcjonalny oraz jego złożoność przestrzenna (wymagana liczba zasobów układu FPGA) pozwolą z jednej strony określić warunki krańcowe ich funkcjonowania, a z drugiej oszczować potencjalne możliwości rozwoju systemu np. o dodatkowe efekty.

6. Planowane testy

Pierwszym testem poprawności działaniu układu będzie oczywiście empiryczna ocena dźwięku uzyskanego za pomocą poszczególnych efektów. Pozwoli ona nie tylko wykryć potencjalne błędy w algorytmach przetwarzania (objawiające się np. wprowadzaniem nadmiernego szumu), ale także dostosować stałe parametry układu takie jak długości zastosowanych kolejek FIFO.

Drugi krok weryfikacji rozwiązania zostanie zrealizowany dzięki zastosowanemu interfejsowi danych w postaci aplikacji języka Python. Fakt, że posiadać będzie ona dostęp zarówno do surowych jak i przetworzonych danych pozwoli w łatwy sposób wyrysować wykresy przedstawiające przebiegi sygnałów i np. ich transformaty. Możliwe będzie dzięki temu dokładne przyjrzenie się efektom działania poszczególnych filtrów celem wykrycia źródeł potencjalnych problemów.

7. Wybór platformy

Ostateczny wybór platformy zostanie dokonany po stworzeniu fundamentów projektu pozwalających oszacować wymagania urządzenia dotyczące zasobów układu FPGA. Na ten moment potencjalny wybór został zawężony do trzech zestawów ewaluacyjnych. Pierwszy z nich to **Digilent Cmod A7** wyposażony w moduł XC7A35T-1CPG236C z rodziny Artix-7. Niewielkie rozmiary oraz cena nieprzekraczająca 400zł są największymi zaletami tego wariantu. Posiada on 8-bitową pamięć SRAM o pojemności 512KB oraz pamięć szeregową Quad-SPI wielkości 4MB. Druga z rozważanych platform to **Digilent Arty S7**. Jest

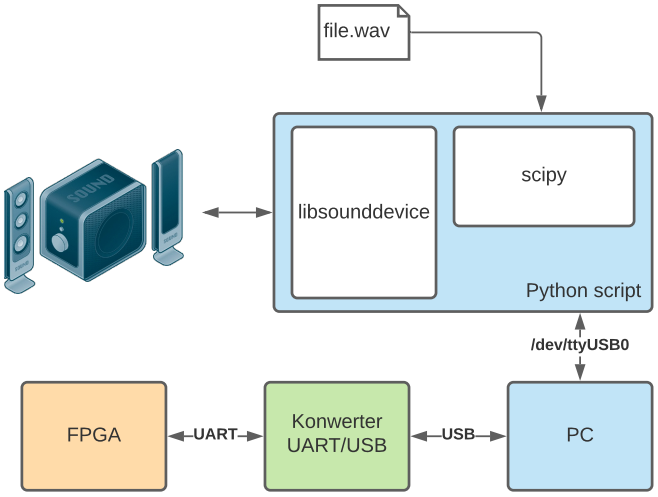
to układ bardziej rozbudowany od poprzedniego, oparty o moduł XC7S50-1CSGA324C (Spartan-7) zawierający ponad 50% więcej bloków logicznych. Sama płytka oferuje ponadto kilka diod LED, przełączniki mono- i bistabilne oraz cztery złącza Pmod. Cena tej platformy jest o około 18% wyższa, co oznacza wysoki stosunek zasobów do ceny, jednak liczyć się trzeba ze znacznie większymi wymiarami płytki.

Trzeci wariant to z kolei zestaw **Digilent Cora Z7**. Jest on najuboższy w bloki logiczne a przy tym wyceniany na podobnym poziomie co Arty S7. Posiada jednak układ XC7Z007S-1CLG400C z rodziny Zynq co oznacza, że poza modułem FPGA zawiera on również jednordzeniowy procesor w architekturze ARM Cortex-A9. Zestaw ten brany jest pod uwagę jedynie ze względu na prywatną sympatię autora do mikrokontrolerów bazujących na architekturze Cortex-M i wynikającej z niej chęci zapoznania się z architekturą aplikacyjną rodziny ARM. Zostanie wybrany wówczas, gdy zasoby zawartego w nim modułu FPGA okażą się wystarczające do zaimplementowania tworzonego rozwiązania.

Część II.

Implementacja

1. Struktura projektu

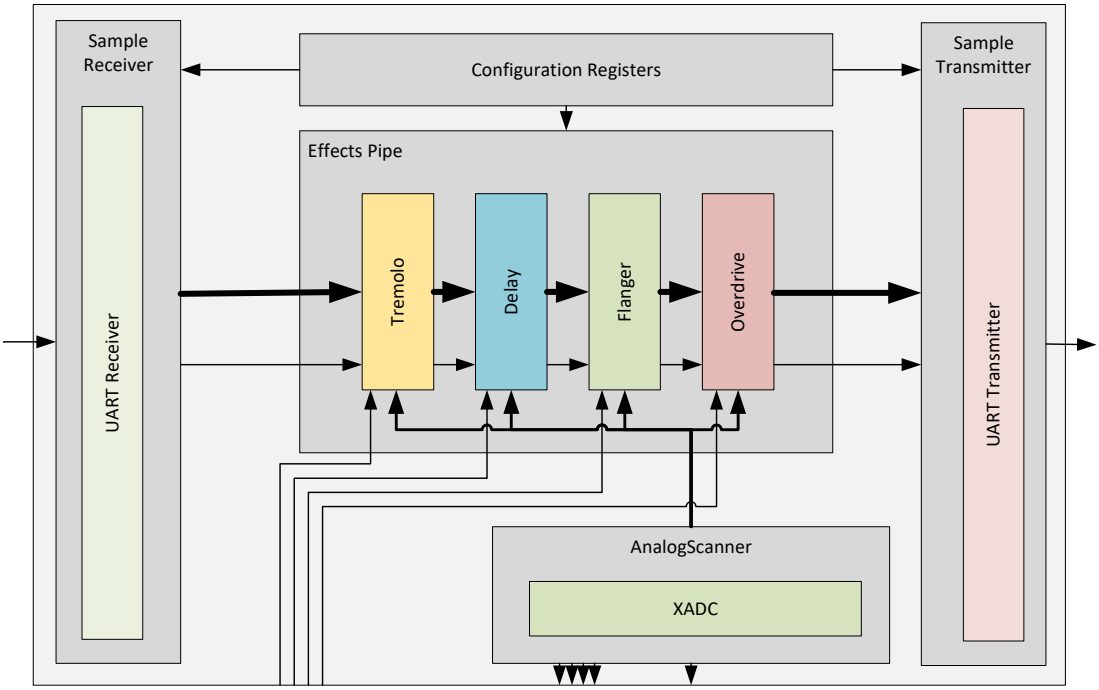


Rysunek 9: Struktura systemowa projektu

Realizację przedstawionej w części pierwszej koncepcji rozpoczęto od nakreślenia struktury projektu na trzech płaszczyznach: **systemowej**, **implementacyjnej** i **projektowej**. Pierwsza z nich obejmowała zdefiniowanie węzłów, które będą brały udział w procesie przetwarzania dźwięku od otwarcia zawierajacego go pliku do chwili wyemitowania przetworzonej wersji przez urządzenie audio. Jak przedstawiono na Rys. 9, wyróżniono cztery zasadnicze jednostki. *Skrypt języka Python* odpowiedzialny jest za załadowanie pliku dźwiękowego do pamięci RAM, przekazanie i odebranie go z układu przetwarzającego, a także wysterowanie domyślnego urządzeni audio w oparciu o zmodyfikowane próbki. Urządzeniem przetwarzającym jest oczywiście układ *FPGA*, który implementuje wymienione wyżej filtry. Pośrednikami w komunikacji pomiędzy układem scalonym a aplikacją są oparty o układ FT232R *mostek USB/UART* oraz *komputer klasy PC*. Taka konfiguracja pozwala na dwukierunkową transmisję danych z prędkością do 375KB na sekundę, co przy założeniu 16-bitowych próbek oraz częstotliwości próbkowania na poziomie 44100Hz powinno wystarczyć do strumieniowania dźwięku zarówno w trybie mono jak i stereo. W projekcie wykorzystano biblioteki `sounddevice` oraz `scipy` języka Python. Pierwsza z nich udostępnia interfejs dla szerokiej gamy urządzeń audio, natomiast druga pozwala przetwarzać pliki w formacie WAV do postaci tablic danych biblioteki `numpy`. Aby zredukować czas wykonywania operacji na wirtualnym porcie szeregowym, zdecydowano się podzielić strumieniowane dane na bloki o wielkości 5000B, które wysyłane są do układu FPGA pojedynczo.

Określenie struktury implementacyjnej obejmowało doprecyzowanie zaproponowanych w części pierwszej koncepcji konfiguracji urządzenia. Ostatecznie zdecydowano się na model przedstawiony na Rys. 10. Wyróżnić w nim można cztery zasadnicze części. Odbiornik oraz nadajnik UART zostały wcielone w ramy dwóch szerszych struktur, które umożliwiają wymianę N-bajtowych próbek danych w ramach pojedynczej transakcji. W ramach interfejsu użytkownika wykorzystano moduł XADC (ang. *Xilinx Analog/Digital Converter*), który w ramach jednostki `AnalogScanner` pozwala konwertować sekwencję do 16 sygnałów analogowych podłączonych do układu poprzez zewnętrzny multiplekser. Umożliwi to dołączenie szeregu potencjometrów odpowiedzialnych za określanie parametrów

filtrów przedstawionych w centralnej części rysunku. Aby zmaksymalizować elastyczność konfiguracji postanowiono pozostać przy pierwotnej koncepcji jednolitego interfejsu efektów, którego szczegóły zostaną omówione w dalszej części dokumentu. Dodatkowym elementem projektu jest także zestaw rejestrów konfiguracyjnych, które określają porametry interfejsów komunikacyjnych oraz sposób mapowania mierzonych wartości analogowych na parametry filtrów. Na ten moment zostały one zaimplementowane w sposób niejawny, jednak w przyszłości planowane jest zmiana tego podejścia na rzecz ułatwienia rekonfiguracji urządzenia.

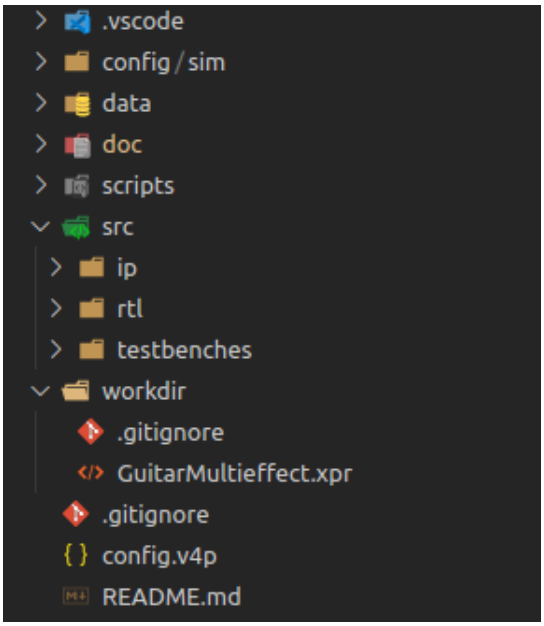


Rysunek 10: Struktura implementacyjna projektu

Ostatnią - choć nie mniej ważną z perspektywy projektanta - kwestią wymagającą określenia była struktura samego projektu rozumiana jako podział drzewa plików i katalogów. Oprogramowanie *Vivado* wykorzystywane na etapie symulacji oraz syntezy nie jest w opinii autora optymalnym środowiskiem programistycznym[1]. Elementami, które składają się na taki stan rzeczy są m.in. męczący oczy interfejs graficzny oraz brak integracji z systemami kontroli wersji. Z tego względu zdecydowano się na wykorzystanie oprogramowania *Visual Studio Code* jako domyślnego narzędzia tworzenia kodu oraz zarządzania projektem. Aby było to możliwe koniecznym stało się zapoznanie się z odpowiednimi dokumentami ([5], [6]) udostępnianymi przez firmę Xilinx. Pozwoliło to określić te z plików generowanych przez *Vivado*, które są konieczne do odtworzenia struktury projektu. Na tej podstawie zdecydowano się na podział przedstawiony na Rys. 11. Elementami wartymi wyróżnienia są katalogi *src/ip* oraz *workdir*. Pierwszy z nich przechowuje pliki w formacie XML opisujące konfigurację wykorzystywanych w projekcie bloków XADC oraz BRAM (ang. *Block Random Access Memory*), które zostały wyekstrahowane ze struktury pro-

[1] Vivado jest oczywiście środowiskiem służącym do **konfiguracji** a nie programowania układów FPGA. Z uwagi na brak lepszego terminu postanowiono jednak pozostać przy określeniu *środowisko programistyczne*

jektowej *Vivado*. Drugi stanowi z kolei miejsce docelowe dla hierarchii plików generowenj przez samo oprogramowanie. Jedynym jej elementem, który włączony został do systemu kontroli wersji jest plik `GuitarMultieffect.xpr` zawierający pełny opis struktury projetktu. Warto zauważyć, że przyjęte podejście do kontroli wersji pozwoli na uruchomienie projektu jedynie przy użyciu aktualnie wykorzystywanej wersji *Vivado* (2020.2).



Rysunek 11: Struktura projektowa

2. Interfejs komunikacyjny

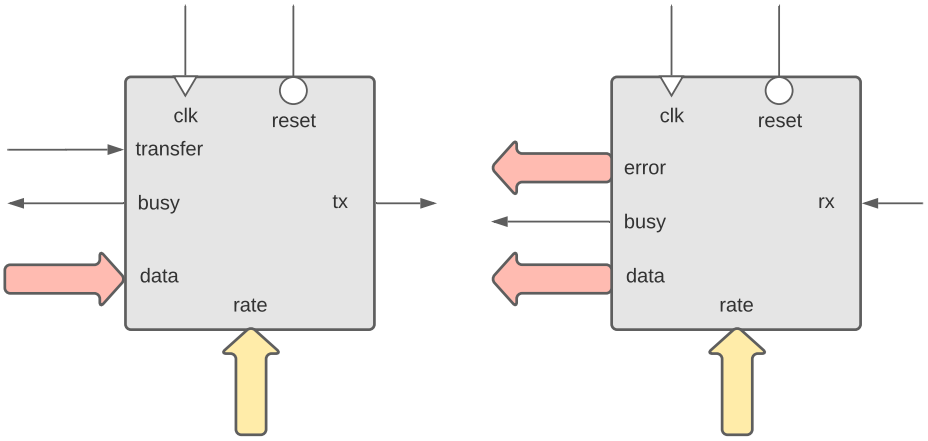
Budowę interfejsu komunikacyjnego rozpoczęto od zaimplementowania modułów UART. Układ ich wyprowadzeń przedstawiono na Rys. 12. Wąskie strzałki symbolizują linie jednobitowe. Strzałki czerwone oraz żółte to linie wielobitowe. Pierwsze z nich oznaczają, że są port skierowany jest do "wewnątrz" struktury urządzenia. Z kolei w przypadku drugich porty skierowane są "na zewnątrz". Konwencja ta została zachowana w przypadku kolejnych rysunków.

Wejście `clk` to podłączenie do głównego zegara systemowego, natomiast `reset` to asynchroniczny reset układu (aktywny stanem niskim). Wejścia te są uniwersalne dla wszystkich implementowanych modułów. Porty `rate` pozwalają określić szybkość transmisji. Wyrażana jest ona jako ilość cykli zegara systemowego przypadających na jeden znak wysyłany/transmitowany (minus 1). W przypadku interfejsu odbierającego ustawienie wartości 0 na tym porcie zablokuje odbiór danych[2]. Oba moduły implementują także linię wyjściową `busy`, która, gdy ustawiona w stanie wysokim, oznacza, że układ zajęty jest odbiorem/transmisją danych.

Podsystem odbiorczy udostępnia wyjście odebranych danych `data` oraz oczywiście linię szeregową `rx`. Port `error` reprezentowany jest jako trzybitowa linia typu `record`. Usta-

[2]Stan linii RX testowany jest w połowie trwania znaku, co wymusza maksymalny stosunek prędkości transmisji do prędkości zegara systemowego na poziomie 1:2

wiana jest w momencie zmiany stanu linii `busy` ze stanu wysokiego na niski i zawiera flagi błędu odbioru bitu startu, bitów stopu oraz parzystości. Wystąpienie błędu sygnalizowane jest stanem wysokim. Równolegle z flagami błędów ustawiany jest również port `data`[3]. Podsystem nadawczy posiada porty `data` i `tx` o znaczeniu analogicznym do przypadku odbiorczego. Udostępnia on także linię `transfer`. Ustawienie jej w stan wysoki[4] w czasie, gdy linia `busy` znajduje się w stanie niskim, rozpoczyna transmisję danych znajdujących się na wejściu `data`.



Rysunek 12: Struktura wyprowadzeń modułów UART TX orax UART RX

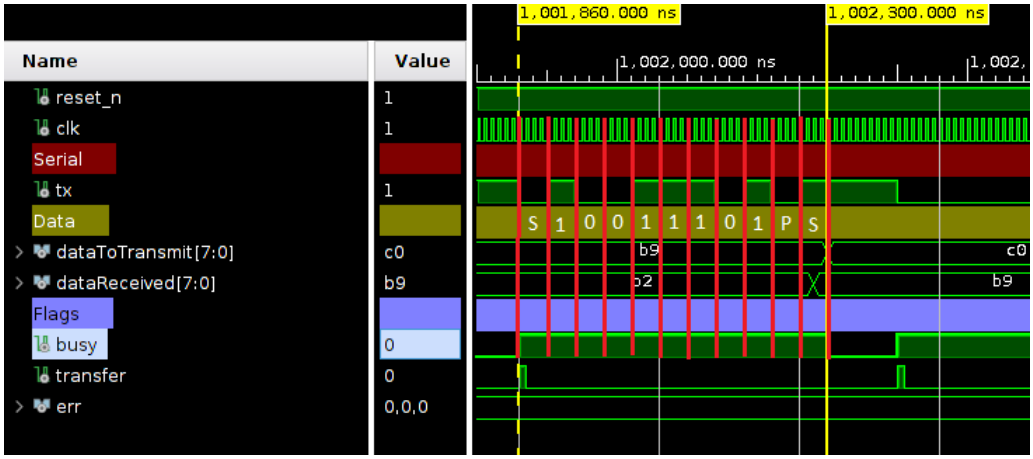
Struktury `entity` reprezentujące obie podjednostki wyposarzono w identczny zestaw parametrów (ang. *generic*). Pozwalają one określić szerokość portu `rate` i format przesyłanych danych (liczbę bitów danych i stopu oraz rodzaj parzystości). Przy ich pomocy możliwe jest też zanegowanie sygnału transmisyjnego tak, aby bezczynność linii transmisyjnej symbolizowana była stanem wysokim oraz (niezależne) zanegowanie bitów danych. W projekcie wykorzystane zostały oba rodzaje negacji.

Obie jednostki zostały zaimplementowane jako synchroniczne automaty skończone o pięciu stanach. Pierwszy z nich - *idle* - przyjmowany jest w momentach bezczynności, tj. gdy moduł oczekuje na wykrycie stanu aktywnego linii odbiorczej lub linii `transfer`. W momencie zajścia warunków startu automat przechodzi w stan odbioru/transmisji bitu startu. Przy przejściu tym resetowany jest wewnętrzny licznik długości znaku oraz ustawiana jest linia `busy`. Wartość znajdujące się na wejściu `rate` zapisywana jest w wewnętrznym buforze. W przypadku transmisji zapisywana jest także wartość na wejściu `data` a stan linii `tx` zmieniany jest na aktywny. Rzeczony licznik inkrementowany jest następnie o jeden w każdym następnym cyklu zegara systemowego. Moduł odbiorczy oczekuje aż jego wartość dojdzie do połowy zapisanej wartości `rate` (powiększonej o 1), a następnie próbkuje stan linii `rx` w celu zweryfikowania stanu bitu startu. Moduł nadawczy czeka natomiast czas dwukrotnie dłuższy. W przypadku obu jednostek sytuacje te wiążą się z przejściem do kolejnego stanu - odbioru/transmisji bitów danych. Przejście do wszystkich kolejnych stanów (odbioru/nadawania bitów danych, parzystości i stopu) wiąże się ze zresetowaniem licznika długości trwania znaku. W przypadku odbioru bitów startu, stopu oraz parzystości odbywa się także ewentualne ustawieniem odpowiedniej flagi

[3] W przypadku wystąpienia błędu odbioru port *data* jest zerowany
[4] Linia aktywna jest poziomem wysokim

błędu w wewnętrznym buforze. Przejścia do kolejnych stanów wywoływane są osiągnięciem wartości `rate` przez licznik. W następnym cyklu po odebraniu/nadaniu wszystkich bitów, stan linii `busy` ustawiany jest na niski, a dane i flagi błędu (w przypadku odbioru) kopiowane są z wewnętrznych buforów na wyjście.

Dla każdego modułu stworzona została symulacja mająca na celu zweryfikowanie poprawności działania. W jej ramach zostały stworzone procedury biblioteczne realizujące niezależnie funkcjonalność transmisji/odbioru, których kod został oparty na materiałach udostępnionych w trakcie wykładów. Posiłkowanie się nimi miało na celu eliminację potencjalnych błędów w procesie weryfikacji.

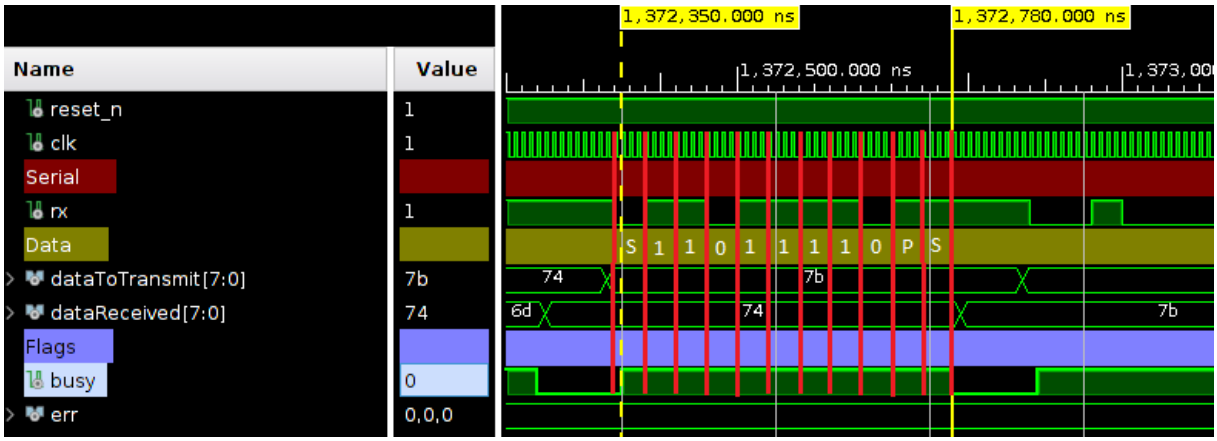


Rysunek 13: Wycinek sumylacji modułu UART TX

Symulacje miały charakter całościowy, tj. przetestowane zostały wszystkie kombinacje bitów transmitowanego słowa. Na Rys. 13 przedstawiono wycinek symulacji modułu TX. Transmitowane dane mają w tym przypadku format 8E1. Prędkość transmisji to 25MHz natomiast prędkość zegara systemowego to 100MHz. Zastosowano tu zarówno negację sygnału jak i danych. Pionowe linie w kolorze czerwonym rozmieszczone zostały w odstępach czterech cykli zegara systemowego, co odpowiada długości trwania transmitowanego bitu. Przedziadziały oznaczone literami S odnszą się kolejno do bitu startu oraz bitu stopu. Zgodnie z oczekiwaniami odpowiadający im stan linii `tx` to odpowiednio niski i wysoki. Bity danych to kolejno 10011101. Po odwróceniu ich klejności (słowa transmitowane są w kolejonści od najmniej do najbardziej znaczącego bitu) otrzymujemy 10111001. Dane te zapisane w formacie heksadecymalnym mają postać 0x$B9$. Identyczna wartość ukazana jest na obrazie symulacji w wierszu `dataToTransmit`[5], który obrazuje aktualnie transmitowane słowo. Oznacza to, że bity danych zostały nadane poprawnie. Bit parzystości ustawiony jest w stanie niskim. Biorą pod uwagę nieparzystą liczbę jedynek w przesyłanym słowie oraz negację sygnału i danych można stwierdzić, że również ten bit został przesłany poprawnie. Sygnał `dataReceived` jest ustawiany na wartość odebraną przez ww. funkcję biblioteczną. Jak widać, jego wartość zmienia się na oczekiwaną (0x$B9$) w momencie połowy trwania bitu stopu. Na rysunku zaznaczone zostały także momenty ustawienia i zresetowania stanu linii `busy`. Różnica między nimi opiewa na 440 ns. Jest to również wartość oczekiwana, ponieważ czas trwania pojedynczego bitu wynosi

[5]Wysyłane słowo były wektorami losowymi z realizacji rozkładu jednostajnego implementowanego za pomoca funkcji *uniform* biblioteki *math_real*

40 ns (cztery takty zegara systemowego) a sumaryczna liczba transmitowanych bitów to 11 (1+8+1+1). Aby ułatwić weryfikację działania modułu symulacja została zaprojektowana tak, aby w momencie niezgodności odbieranego znaku z sygnałem `dataToTransmit` stan bitów linii `dataReceived` ustawiany był na X reprezentowany w symulacji kolorem czerwonym.



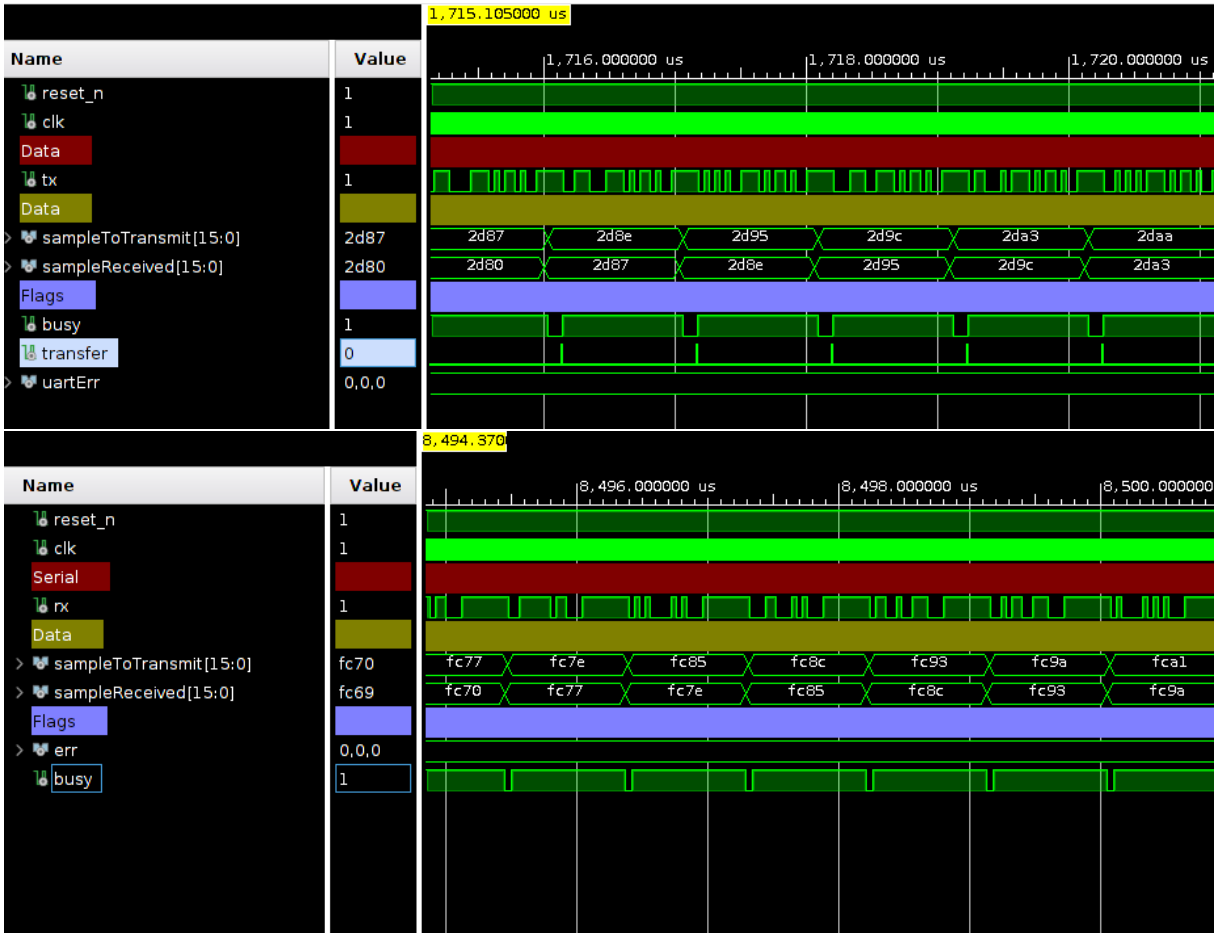
Rysunek 14: Wycinek sumylacji modułu UART RX

Wycinek analogicznej symulacji dla modułu RX (przeprowadzonej przy tych samych parametrach transmisji) został zaprezentowany na Rys. 14. Ponowienie analizy przeprowadzonej dla przypadku transmisji pozwala stwierdzić, że również odbiór danych odbywa się prawidłowo. Warto zauważyc, że czas wysoki na linii `rx` trwa o 10 ns krócej, niż w poprzednim przypadku. Wynika to z faktu, że automat skończony modułu RX wychodzi ze stanu *idle* dopiero w momencie odczytania stanu aktywanego na linii odbiorczej[6].

Przeprowadzone symulacje pozwoliły poprawić początkowe błędy implementacji a tym samym przejść do kolejnego etapu projektowania interfejsu komunikacyjnego. Było nim stworzenie jednostek akumulujących odbierane i transmitowane słowa do postaci N-bajtowych próbek (określanych niżej mianem *SampleTx* i *SampleRx*). Ich interfejs jest niemal identyczny jak ten przedstawiony na Rys. 12. Jedynymi różnicami są szerokość portów `data` oraz brak portów wejściowych `rate`. Szybkość transmisji jest w ich przypadku ustalana poprzez parametry (*generic*). Wzbogacają one interfejs UART o dodatkowy N-bajtowy bufor wraz z multiplekserem o N 8-bitowych wejściach, którego wyjście połączone jest z portem `data` wewnętrznej instancji podjednostki UART TX/RX. Układy te pozwalają na odbiór/transmisję próbek w formacie *little endian* wybranym ze względu na wykorzystanie w projekcie komputera opartego o architekturę x86-64. Również w ich przypadku zaprojektowane zostały symulacje mające na celu sprawdzenie poprawności dekompozycji słów na kolejno wysyłane/odbierane bajty. Filozofia ich działania jest również analogiczna do tej przedstawionej powyżej. Funkcjonalność komplementarna względem testowanego modułu jest implementowana poprzez wykonane wcześniej funkcje biblioteczne wzbogcone o logikę pozwalającą operować na N-bajtowych słowach. Aktualnie wysyłane dane mogą być obserwowane na symulacji jako sygnał *sampleToTransmit*, natomiast dane odebrane

[6]Po przygotowaniu grafik do sprawozdania zauważono, że moduł RX czeka o jeden takt zegara za długo na wystawienie odberanych danych (które są gotowe już w połowie trwania ostatniego bitu stopu). Błąd ten wynikał z ustawienia niewłaściwego momentu próbkowania bitu startu, który przesuwał pozostałe chwile próbkowania. Został on poprawiony.

jako sygnał *sampleReceived*. Jeden z nich ustawiany jest zawsze przez moduł testowany, a drugi przez proces wywołujący funkcję biblioteczną. Wycinek rzeczonych sumlacji dla przypadku $N = 2$ został zaprezentowany na Rys. 15. Niezgodność danych odbieranych z danymi wysyłanymi lub błąd odbioru sygnalizowane są ponownie ustawieniem bitów sygnału *sampleReceived* w stan X.

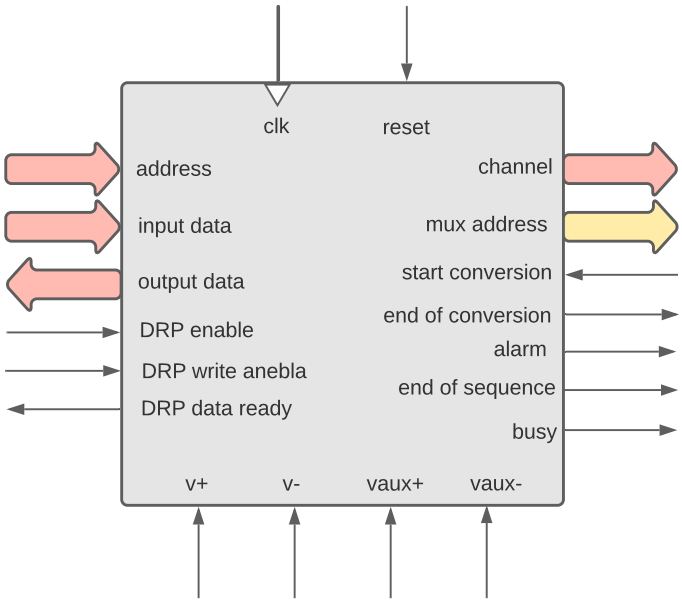


Rysunek 15: Wycinek sumylacji modułów *SampleTx* (u góry) i *SampleRx* (u dołu)

3. Interfejs analogowy

Drugą z zaimplementowanych sekcji projektu był interfejs analogwy, który umożliwić miał odczyt wartości napięć na potencjometrach z wykorzystaniem zewnętrznego multipleksera analogowego. Zdecydowano się wykorzystać w tym celu generator IPC (ang. *Intelectual Property Core*) *XADC Wizard* (wersja 3.3). Prace rozpoczęto od zapoznania się zarówno z dokumentacją modułu ADC dostępnego w układach FPGA serii sióddmej ([7]) jak i samego generatora ([8]). Z ich pomocą udało się rozszyfrować znaczenie poszczególnych parametrów rdzenia. Zdecydowano się na wykorzystanie interfejsu DRP (ang. *Dynamic Reconfigurable Port*) ze względu na jego relatywną prostotę. XADC został skonfigurowany w trybie sekwencyjnym (ang. *channel sequencer*) z manualnym wyzwalanym (ang. *event mode*). Prędkość sygnału zegarowego przyjęto na poziomie 100MHz (zgodnie

z założeniami projektu) natomiast częstotliwość konwersji na poziomie 50KSPS (ang. *KiloSamples Per Second*). Dodatkowo skorzystano z udostępnianej przez XADC możliwości automatycznego wysterowania linii *select* zewnętrznego multipleksera analogowego. Jako kanał roboczy wybrano *vaux0*, który skonfigurowano w trybie unipolarnym. Zrezygnowano z uśredniania wartości próbek w kanałach oraz dezaktywowano wszystkie alarmy. Ostatnim krokiem było ustawienie odpowiednich wartości w sekcji *Analog Sim Options* zakładki *Basic*. Wpisano w niej ścieżkę do zewnętrznego pliku zawierającego przebiegi symulowanych wartości analogowych. Jak się później okazało sfinalizowanie tej konfiguracji celem uzyskania możliwości uruchomienia rzeczonej symulacji, wymagało spędzenia dodatkowych kilku godzin na przeglądaniu forum `forum.xilinx.com` celem odszukania informacji brakujących w dokumentacji.



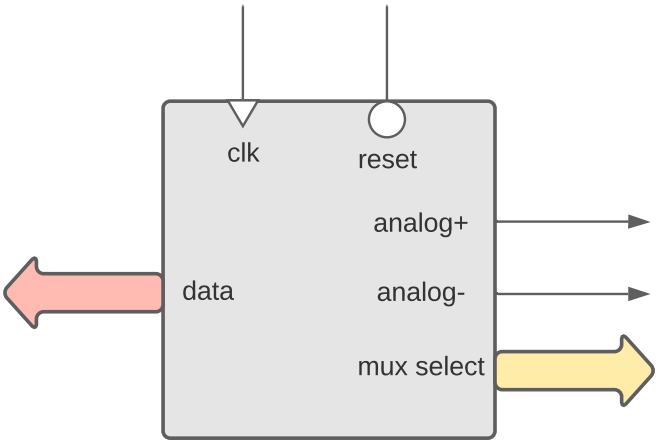
Rysunek 16: Struktura portów wygenerowanego modułu XADC

Strukturę wyprowadzeń wygenerowanego bloku funkcjonalnego przedstawiono na Rys. 16. Po lewej stronie przedstawiono sygnały związane z odczytem i zapisem wewnętrznych rejestrów modułu. Na dolnej krawędzi znalazły się bipolarne wejścia analogowe. Znaczenie sygnałów zaprezentowanych naprawej krawędzi są następujące:

- *channel* - port wyjściowy reprezentujący adres rejestru zawierającego dane z ostatniego kanału spróbkowanego przez XADC; ustawiany jest na końcu konwersji (niewykorzystany w projekcie)
- *mux address* - linie wysterowujące wejścia *select* zewnętrznego multipleksera analogowego
- *start conversion* - linia aktywujące nową konwersję
- *end of conversion* - linia ustawiana w stan wysoki na jeden takt zegara po zakończeniu konwersji
- *alarm* i *end of sequence* - linie alaramu oraz sygnalizacji końca sekwencji aktywne w stanie wysokim (nieużywane w projekcie)

- *busy* - linia ustawiana w stan wysoki na czas trwania konwersji

Wokół tak przedstawiającej się struktury stworzony został prosty moduł interfejsujący, który automatyzuje cykliczne wyzwalania konwersji oraz odczyt danych z wewnętrznych rejestrów XADC. Jego struktura została przedstawiona na Rys. 17. Port `data` reprezentowany jest przez tablicę N wektorów 12-bitowych ($N <= 16$) odzwierciedlających wartości napięcia na kolejnych kanałach multipleksera. W projekcie wykorzystano dziewięć kanałów (dziewięć potencjometrów), których znaczenie przedstawiono w dalszej części dokumentu. Na zewnątrz modułu wystawione zostały poza tym wejścia analogowe kanału *vaux0* (oznaczone omyłkowo jako wyjścia) oraz linie *select* dla zewnętrznego multipleksera. Dedykowane wejścia analogowe (*vp/vn*) zostały podłączone na stałe do masy. Parametry (ang.*generic*) modułu pozwalają skonfigurować ilość wykorzystywanych kanałów[7] oraz częstotliwość ich próbkowania.



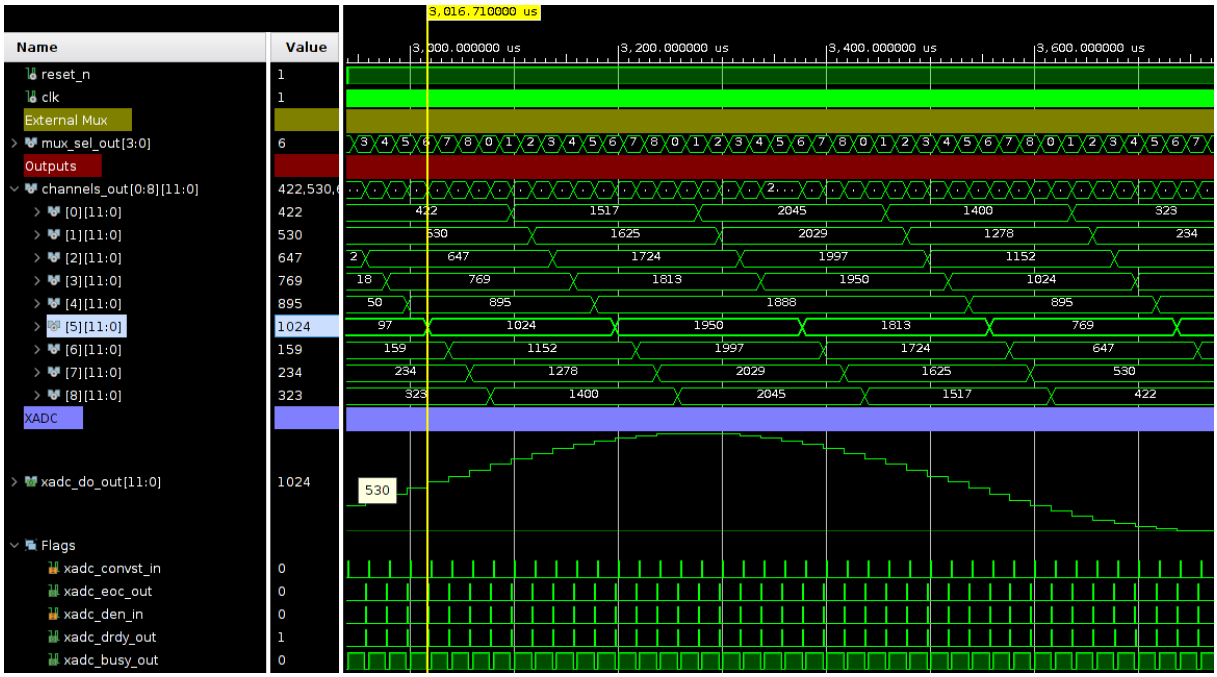
Rysunek 17: Struktura portów bloku skanera kanałów analogowych

Zasada działania modułu zasadza się na dwóch synchronicznych procesach. Pierwszy z nich wyposażony jest w wewnętrzny licznik, który odmierza takty zegara systemowego między kolejnymi konwersjami. Gdy jego wartość spadnie do zera linia `start conversion` bloku XADC zostaje ustawiona w stan wysoki na jeden cykl zegara. Równolegle wartość wyjścia `mux address` - powiększona o wartość 0x10 (adres danych pierwszego kanału pomocniczego *vaux0*) - zostaje przepisana na wejście adresowe interfejsu DRP. Drugi proces implementuje dwustanowy automat skończony. W pierwszym ze stanów proces oczekuje na pojawienie się logicznej jedynki na linii `end of conversion`. Gdy sytuacja ta zajdzie, linia `DRP enable` również ustawiana jest w stan wysoki, co rozpoczyna odczyt danych spod adresu znajdującego się na liniach adresowych (ustawianych przez pierwszy proces). Akcja finalizowana jest przejściem do drugiego stanu, w którym proces pozostaje do wystawienia przez XADC logicznej jedynki na linii `DRP ready` oznaczającej pojawienie się odczytanych danych na procie `output data`. Dane te kopowiane są do odpowiedniego wektora w tablicy `data` i proces przechodzi ponownie do stanu pierwszego.

W celu weryfikacji zaimplementowanego rozwiązania ponownie przygotowano stosowną symulację. Tym razem weryfikacja nie miała charakteru formalnego, tzn. efekty działania testowanego modułu nie były programowo porównywane z wartościami oczekiwanymi. Ze

[7] Musi być zgodna z konfiguracją podaną na etapie generowania XADC

względu na złożoność implementacji takiego rozwiązania ograniczono się jedynie do weryfikacji jakościowej poprzez manualną analizę zasymulowanych przebiegów. Aby jednak symulacja była możliwa należało najpierw stworzyć odpowiedni plik zawierający przebiegi wartości napięcia na kolejnych kanałach ADC. Generację takowego pliku zautomatyzowano za pomocą prostego skryptu języka Python, który pozwala zdefiniować długość trwania generowanych sygnałów, ich przebieg oraz częstotliwość "próbkowania". Z jego pomocą wygenerowano sinusoidalny przebieg napięcia na kanale *vauxp0* ustawiając jednocześnie pozostałe kanały na wartość 0. Fragment symulacji obrazuje Rys. 18. Przyjęto w niej częstotliwość konwersji na maksymalnym poziomie 50KSPS. Jak widać wektory tabeli *channels_ out* (podłączej do portu wyjściowego `data` skanera) są aktualizowane w zakładany sposób. Dla lepszego zobrazowania sposobu działania układu rysunek ukazuje także bezpośrednie przebiegi niektórych portów modułu XADC.



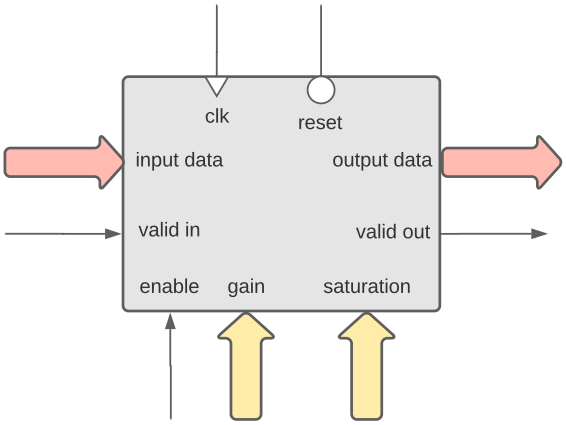
Rysunek 18: Fragment symulacji bloku skanera analogowego

4. Efekt *overdrive*

Pierwszym z zaimplementowanych efektów dźwiękowych był tzw. przester (ang. *overdrive*). Został on wybrany ze względu na możliwość relatywnie prostej realizacji. Schemat jego wyprowadzeń został przedstawiony na Rys. 19. Inkorporuje on siedem portów uniwersalnych dla wszystkich realizowanych efektów. Pierwsze dwa to `clk` i `reset` opisane przy okazji omawiania modułów UART. Porty `input data` oraz `output data` stanowią wejście i wyjście danych z efektu[8]. Przepisanie danych wejściowych do wewnętrznego bufora odbywa się po pojawieniu się stanu wysokiego na wejściu `valid in`. Z kolei na wyjściu `valid out` logiczna jedynka ustawiana jest w momencie wystawienia przetworzonych danych na czas trwania jednego cyklu zegarowego. Gdy wejście `enable` znajduje się w stanie

[8]Dane traktowane są jako N-bitowe liczby w formacie U2

niskim dane przepisywane są synchronicznie z wejścia na wyjście bez wprowadzania zmian. Tak zaprojektowany interfejs pozwala - zgodnie z założeniami - łączyć poszczególne efekty w dowolnej kolenności bez potrzeby ingerowania w ich wewnętrzną strukturę.



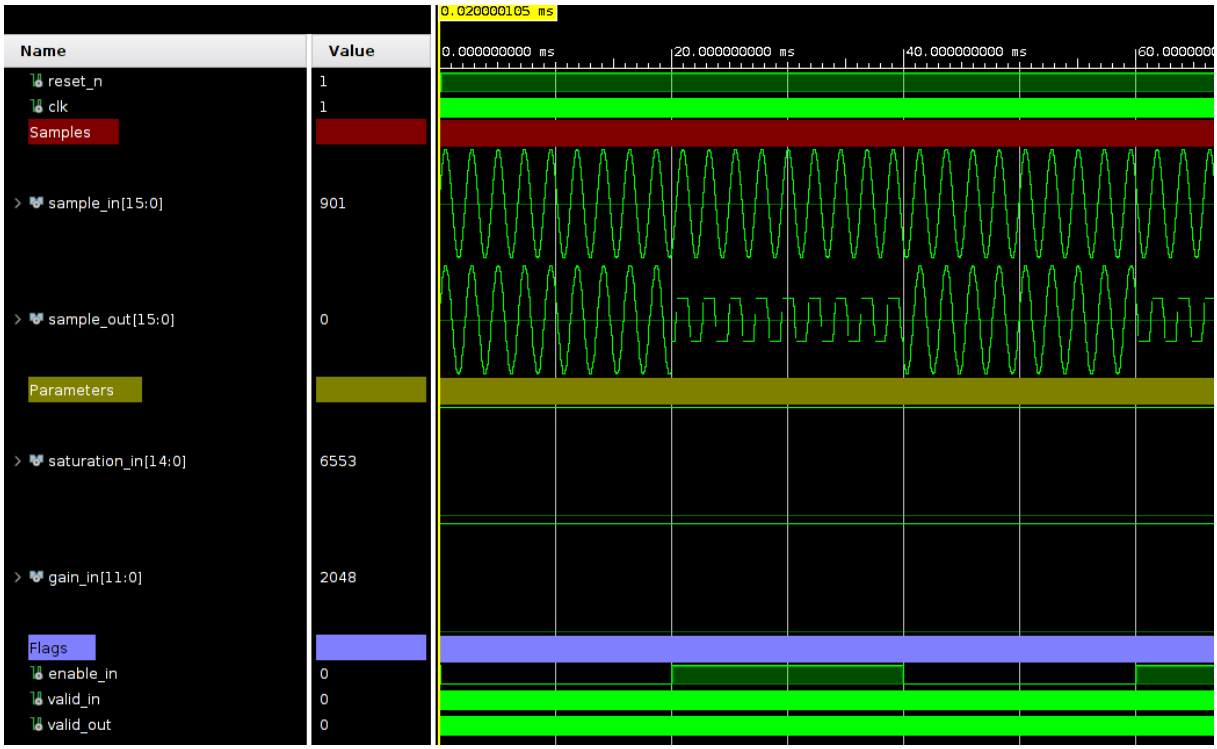
Rysunek 19: Struktura wyprowadzeń bloku *overdrive*

Dwa dodatkowe porty wejściowe - `gain` oraz `saturation` - pozwalają dostosowywać parametry charakterystyczne filtra. Pierwszy z nich określa czynnik przez jaki mnożona jest próbka wejściowa, z kolei drugi determinuje ograniczenie na wartość bezwzględną wyniku mnożenia przed wystawieniem go na wyjście. Obie wartości traktowane są jako liczby bez znaku. Parametry modułu (*generic*) pozwalają określić szerokość przetwarzanych próbek oraz wejścia `gain`. Szerokość wejścia `saturation` jest zawsze o jeden bit mniejsza od szerokości. Opcjonalnie możliwe jest też określenie przesunięcia bitowego (w prawo) wyniku mnożenia przed podaniem go do bloku nasycenia. Pozwala to przeskalować efektywny zakres parametru `gain` do obszaru wartości niecałkowitych.

Zasada działania modułu jest stosunkowo prosta. Po wykryciu stanu wysokiego na linii `valid in` dane wejściowe, oraz parametry `gain` i `saturation` przepisywane są do wewnętrznych buforów. Zawartość bufora danych i wzmocnienia jest asynchronicznie mnożona (z nasyceniem w zakresie reprezentacji[9]). W następnym takcie zegara wynik mnożenia - opcjonalnie przesunięty o skonfigurowaną liczbę bitów - zostaje nasycony w granicach określonych przez bufor `saturation` i wystawiony na wyjście układu. Czas przetwarzania próbki wynosi **1 cyklu**. Jeżeli wejście `enable` znajduje się w stanie niskim, czas ten spada oczywiście do zera.

Tak jak w przypadku interfejsu analogowego stworzona została symulacja, której celem miała być jakościowa ocena pracy układu. Jej fragment ukazuje Rys. 20. Do testów wykorzystano falę sinusoidalną o częstotliwości 440Hz reprezentowaną za pomoca próbek 16-bitowych. Szerokość wejścia `gain` określono na 12 bitów, przy czym wyjście z układu mnożącego przesuwane było o 10 bitów. Pozwala to na ustawianie wzmocnienia sygnału w zakresie $[0, 4)$. Jak widać na ukazanych przebiegach, włączenie modułu skutkuje wystąpieniem charakterystycznego "przycinania" próbek wejściowych.

[9] W ramach prac nad modułem *overdrive* stworzona zostały cztery pomniejsze moduły DSP implementujące operacje dodawania i mnożenia z nasyceniem liczb z i bez znaku o arbitralnej długości. Wykorzystano je przy implementacji kolejnych efektów.



Rysunek 20: Fragment symulacji działania efektu *overdrive*

5. Efekt tremolo

Następnym pod względem skompilikowania, a co za tym idzie pod względem kolejności implementacji, był efekt tremolo. Jak nakreślono w części pierwszej, polega on na modulacji sygnału wejściowego wolnozmiennym sygnałem okresowym (LFO, ang. *Low Frequency Oscilator*). Schemat wyprowadzeń gotowego efektu przedstawia Rys. 21. Podobnie jak w przypadku poprzedniego efektu mamy tu do czynienia z dwoma parametrami: `depth` oraz `period`. Pierwszy z nich - głębokość - określa siłę efektu modulacji. Jego nazwa wzięła się od formuły służącej do obliczania wartości próbki zmodulowanej:

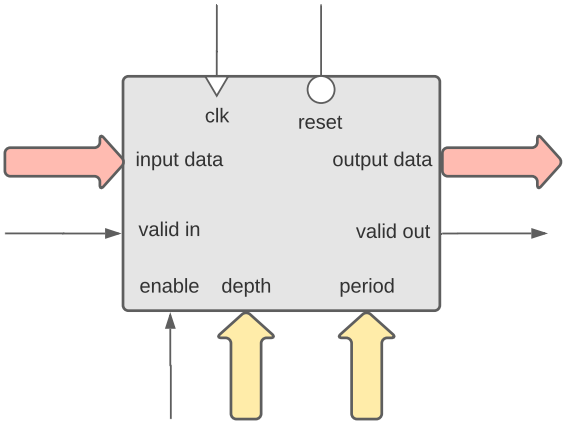
$$y[t] = x[t] \times (1 - d \times m[t]) \tag{1}$$

gdzie $y[t]$ - wartość wyjściowa, $x[t]$ - wartość wejściowa, $m[t]$ - wartość fali modulującej (w zakresie $[0, 1]$), d współczynnik głębokości (w zakresie $[0, 1]$). Jak widać, zwiększenie wartości d powoduje, że efektywny zakres modulacji zbliża się do poziomu 0. Drugi parametr - okres - określa ilość taktów zegara systemowego przypadających na pojedynczą próbkę fali modulującej. Jest to dość dziwna wartość parametryzująca, jednak pozwala na elastyczne dostosowywanie zakresu częstotliwości modulujących oferowanych przez efekt. Parametry układu pozwalają określić szerokość poszczególnych wejść oraz rozdzielczość próbek sygnału wolnozmiennego.

Implementacja jednostki umożliwia wykorzystanie jednej z dwóch funkcji modulujących: sinus oraz falę trójkątną. Generatory obu z nich zostały zrealizowane jako niezależne bloki funkcjonalne. Funkcję sinus zaimplementowano przy użyciu modułów RAM obecnych w FPGA ([9]). Do ich zinterfejsowania wykorzystano IPC *Block Memory* w wesji 8.3 ([10]). Pamięć skonfigurowano w trybie *Native* stosując pojedynczy bufor wyjściowy[10].

[10] Wiązało się to ze zwiększeniem opóźnień operacji odczytu do dwóch cykli.

Jej celem jest przechowywanie N-bitowych próbek z pierwszej ćwiartki okresu funkcji sinus. Pozostała część fali generowana jest przez moduł z wykorzystaniem odpowiednich symetrii. Określenie zawartości pamięci odbywa się na etapie konfiguracji IPC. Wykorzystywane są w tym celu pliki w formacie COE (skrót od ang. *coefficient*). Ich tworzenie odbywa się dwuetapowo. W pierszym kroku wykorzystywany jest skrypt stworzony w ramach projektu, który generuje plik tekstowy zawierający kolejne próbki fali w formacie heksadecymalnym. Następnie plik wynikowy jest przekazywany do skryptu zaczerpniętego z [11], który przetwarza dane wejściowe do pożądanego formatu.

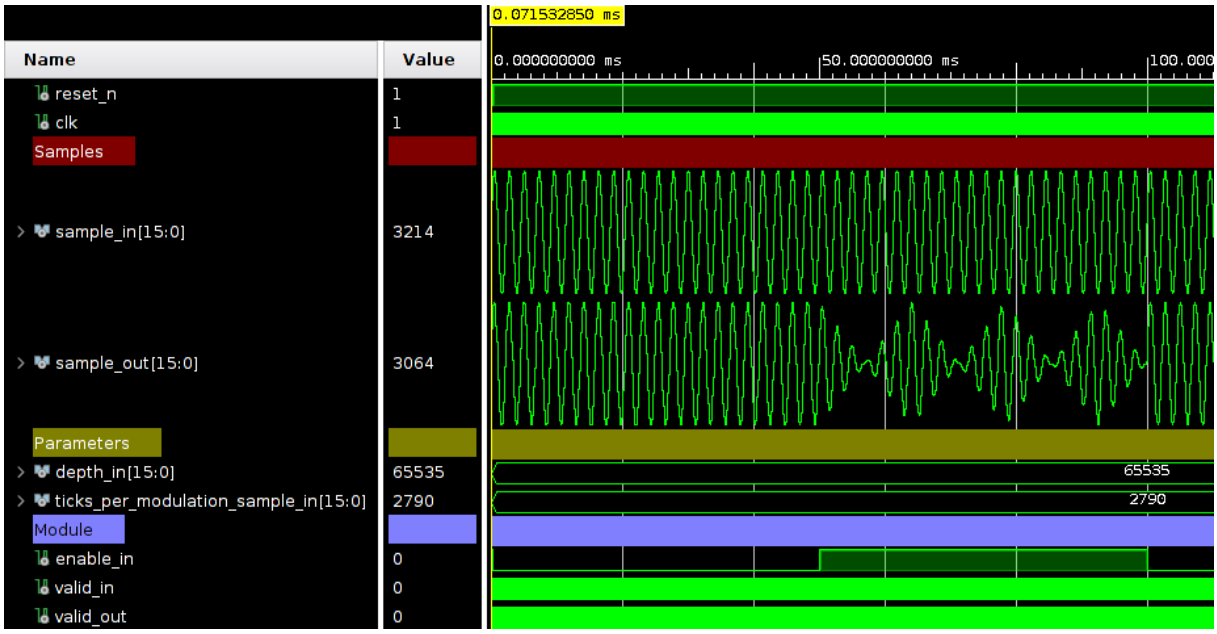


Rysunek 21: Struktura wyprowadzeń bloku tremolo

Implementacja generatora fali trójkątnej była dużo prostsza i sprowadzała się do realizacji dwukierunkowego licznika. Gdy oba generatory zostały wdrożone[11], realizacja efektu stała się formalnością. Jego architektura podzielona została na dwa procesy. Pierwszy z nich odpowiada za generowanie fali modulującej. W każdym takcie zegara wartość wewnętrznego licznika porównywana jest z wartością na wejściu `period`. Jeżeli są one równe, generowana jest następna próbka. W przeciwnym wypadku licznik jest inkementowany. Drugi z procesów monitoruje wejście `valid in`. Gdy jego stan zostanie ustawiony na wysoki, próbka wejściowa oraz parametr *depth* zostają przepisane do wewnętrznych buforów. Obliczenie wartości próbki wynikowej zgodnie z ww. wzorem zachodzi asynchronicznie. Jako że iloczyn $d \times m[t]$ nie zawiera się w przedziale $[0, 1]$ koniecznym staje się przesunięcie wyniku tej operacji w prawo o liczbę bitów będącą sumą szerokości próbki generatora i szerokości wejścia `depth`. W następnym takcie po zapisaniu buforów przetworzona próbka zostaje wystawiona na wyjście układu. Zastosowany rozdział obsługi LFO oraz przetwarzania sygnału wyjściowego pozwolił uniezależnić czas przetwarzania próbki wejściowej od czasu generowania próbek LFO. W ten sposób udało się uzyskać opóźnienie na poziomie **1 cyklu**.

Wycinek symulacji przeprowadzonych dla efektu tremolo został przedstawiony na Rys. 22. Sygnał wejściowy to ponownie fala sinus o częstotliwości 440Hz. Wartość współczynnika `period` dobrana została tak, aby uzyskać częstotliwość modulacji na poziomie 70Hz. Współczynnik głębokości ustawiono na najwyższą wartość. Zastosowano modulację z wykorzystaniem 8-bitowej fali trójkątnej. Analogiczne symulacje przeprowadzone zostały również dla drugiego typu modulacji. W obu przypadkach układ wydaje się działać prawidłowo.

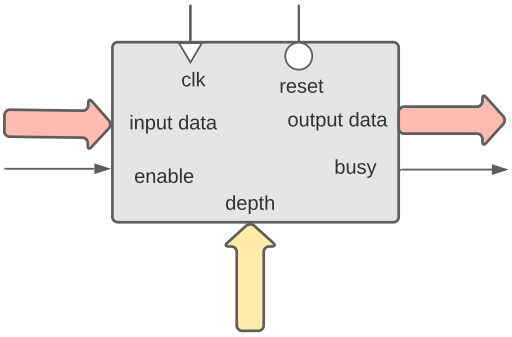
[11] ...oraz przetestowane z wykorzystaniem dedykowanyc symulacjii



Rysunek 22: Fragment symulacji działania efektu tremolo

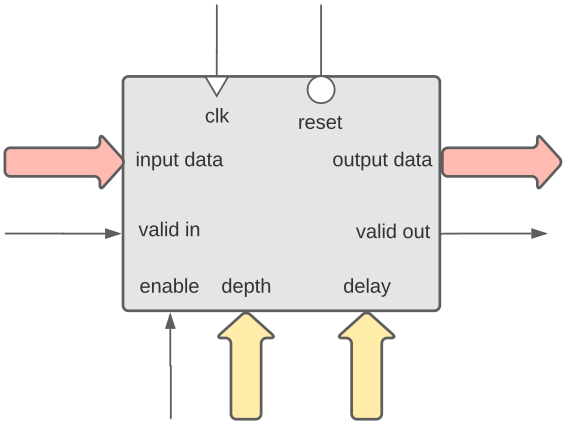
6. Efekt pogłosu

Doświadczenia w wykorzystywaniu bloków BRAM nabyte w czasie implementacji generatora funkcji sinus pozwoliły na sprawną realizację kolejnego z zaplanowanych efektów - pogłosu. Prace nad nim rozpoczęto od zaimplementowania parametryzowalnego **modułu opóźnienia**, ktory miał zostać użyty również w przypadku kolejnego efektu. Zasada jego działania jest stosunkowo prosta i zasadza się na sekwencyjnym zapisie próbek wejściowych w pamięci RAM - traktowanej jako bufor kołowy - oraz odczycie próbek spod adresu będącego wynikiem odejmowania adresu aktualnie zapisywanej próbki i parametru zewnętrznego (stopnia opóźnienia). Blok ten implementuje wyjście i wejście danych oraz długości opóźnienia. Port `busy` informuje o możliwości odebrania opóźnionej próbki z wyjścia. Wejście `enable` umożlwia rozpoczęcie przetwarzenia próbki. Jego strukturę zewnętrzną przedstawiono na Rys. 23.



Rysunek 23: Struktura wyprowadzeń modułu opóźniającego

Porty bloku efektu zostały z kolei ukazane na Rys. 24. Posiada dwa wejścia określające kolejno głębokość echa (`depth`) oraz jego siłę (`delay`). Pierwszy parametr interpretowany jest jako opóźnienie próbki odczytywanej z bloku opóźniającego. Z kolei drugi traktowany jest jako liczba z zakresu $[0, 0.5)$ przez którą mnożone jest wyjście bloku opóźniającego przed zsumowaniem go z aktualną próbką wejściową.



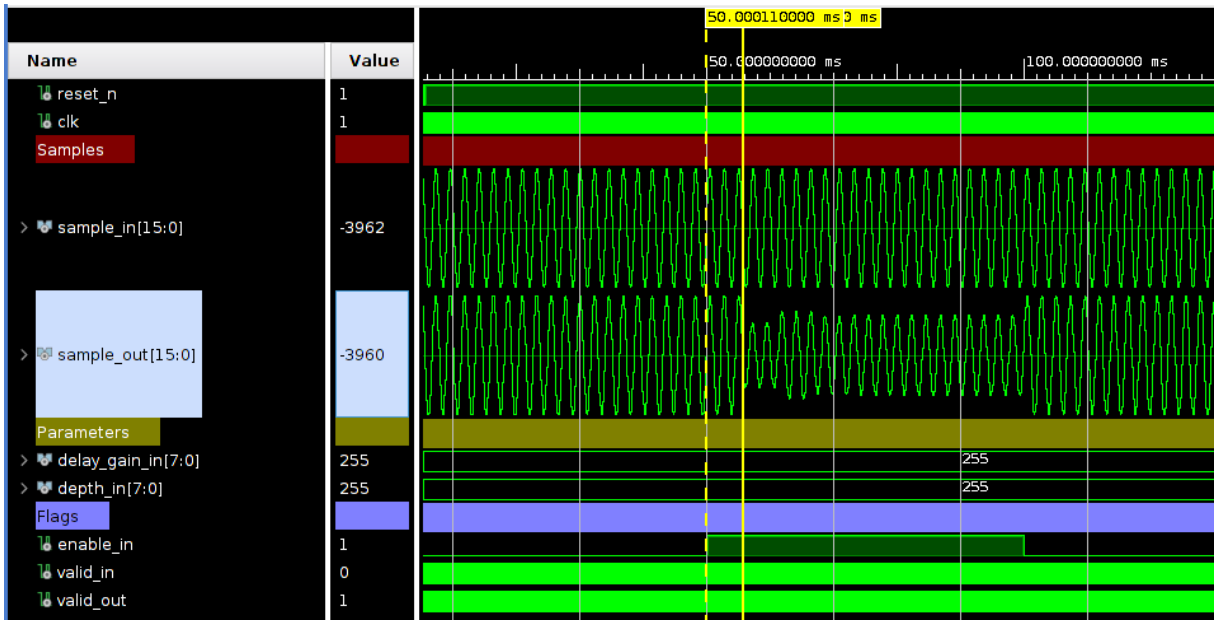
Rysunek 24: Struktura wyprowadzeń bloku *delay*

Dzięki wydzieleniu większego elementu efektu w postaci odrębnego bloku funkcjonalnego, proces dalszej jego realizacji był znacznie ułatwiony. Logika modułu sprowadzona została do dwustanowego automat skończony. W stanie *idle* oczekuje on na pojawienie się jedynki logicznej na porcie `valid in`. Gdy zostanie ona wykryta, wartości z portów danych oraz parametrów są przepisywane do wewnętrznych buforów. Jednocześnie stan wyjścia `busy` efektu oraz linii `enable` modułu opóźniającego ustawiany jest na wysoki. Wejście danych modułu podłączone jest na stałe do bufora **wyjściowego** efektu co oznacza, że w pamięci BRAM zapisana zostanie próbka wynikowa **poprzedniego cyklu przetwarzania**. Po uruchomieniu bloku opóźnienia moduł przechodzi do stanu oczekiwania na wynik. Jego pojawienie się na wyjściu bloku sygnalizowane jest przez opadające zbocze portu `busy`. W momencie jego wykrycia aktualizowany jest stan bufora danych wyjściowych. W zależności od tego czy zbuforowana wartość parametru `depth` jest zerowa czy nie, na wyjście trafia nieprzetworzona wersja danych wyjściowych lub wynik (asynchronicznego) dodawania zawartości bufora wejściowego i iloczynu wyjścia z bloku opóźniającego z wartością bufora `delay`[12]. Dodawanie odbywa się z nasyceniem w zakresie reprezentacji. Warto dodać, że blok opóźniający posiada (opcjonalną) funkcję miękkiego startu, która ogranicza wartość realizowanego opóźnienia do ilości próbek zapisanych w pamięci BRAM od ostatneigo resetu. Ma ona na celu zredukowanie potencjalnych zniekształceń sygnału na początku działania efektów wynikajacych z wykorzystania niezainicjalizowanych komórek pamięci jako właściwych danych.

Do realizacji bloku opóźniającego w przypadku efektu *delay* wykorzystano 21 bloków RAM, których łączna pojemność umożliwi przechowanie do 45056 16-bitowych próbek danych. Dla częstotliwości próbkowania na poziomie 44100Hz oznacza to możliwość osiągnięcia opóźnień rzędu jednej sekundy. Ze względu na zastosowanie bufora na wyjściu bloku BRAM opóźnienie efektu wynosi **4 cykle**. Analogicznie do przypadku poprzednich efektów celem weryfikacji poprawności działania modułu skonstruowany został projekt

[12]Wynik iloczynu jest przesunięty w prawo o liczbę bitów o jeden większą niż szerokość wejścia *depth*

symulacji umożliwiający realizację różnych scenariuszy testowych. Rys. 25 przedstawia fragment symulacji, w której wartość opóźnienia została ustalona na poziomie 255 próbek, natomiast jego siła miała wartość maksymalną (0.5). Sygnał wejściowy to ponownie fala sinusoidalna o częstotliwości 440Hz próbkowana z częstotliwością 44100Hz. Jak widać włączenie efektu nie powoduje natychmiastowej zmiany charakteru sygnału. Zmiana ta następuje po czasie około 5.8 ms. Zgadza się to z przewidywaniami teoretycznymi ($1s/44100 \times 255 \approx 5.8ms$). Ponadto symulacja ukazuje zmniejszenie amplitudy sygnału po dodaniu do niego składnika opóźnionego. To również zgadza się z przewidywaniami. Opóźnienie 255 próbek odpowiada nieznacznie ponad 2.5 okresom sygnału wejściowego. Oznacza to, że znaki próbki wejściowej i opóźnionej są względem siebie odwrócone przez większość czasu, co powoduje efektywne zredukowanie amplitudy fali wyjściowej.

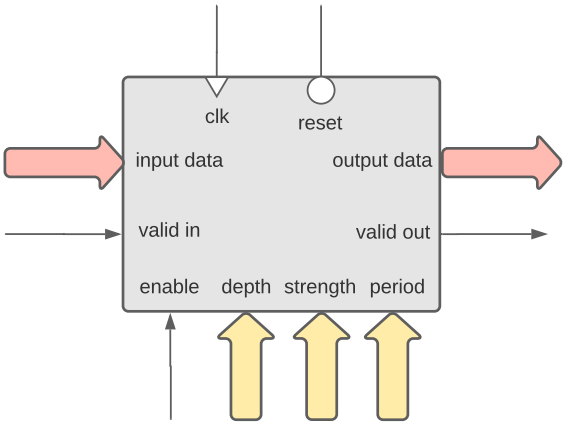


Rysunek 25: Fragment symulacji działania efektu *delay*

7. Efekt *flanger*

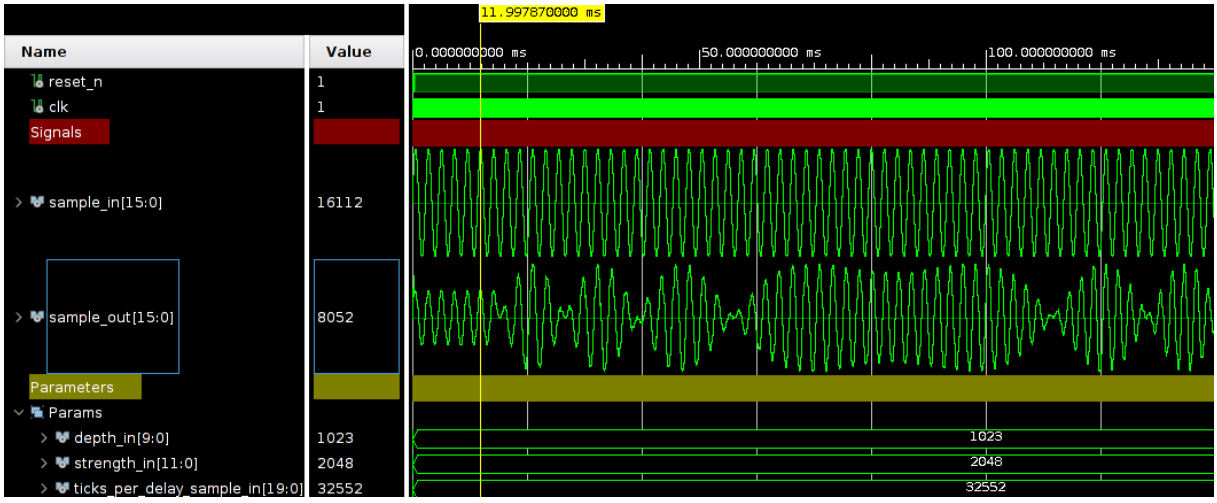
Ostatnim z zaimplementowanych efektów był *flanger*. Jest on wariacją na temat poprzedniego efektu, w której czas opóźnienia jest wolnozmiennym sygnałem sinusoidalny o niewielkiej amplitudzie. Struktura tego efektu jest najbardziej złożona ze wszystkich zaimplementowanych bloków funkcjonalnych, jednak jego realizacja nie była wcale najtrudniejszym zadaniem. Wynikało to z faktu, że komponenty potrzebne do jego budowy - linia opóźniająca oraz generator funkcji sinus - zostały utworzone na wcześniejszych etapach projektu. Struktura zewnętrzna efektu została przedstawiona na Rys. 26. Port `depth` określa czynnik skalujący amplitudę sygnału opóźniającego. Jest interpretowana jako wartość z zakresu $[0, 1)$. Wejście `strength` określa stosunek sygnału opóźnionego do sygnału wejściowego w wynikowej próbce. Wartości skrajne przekładają się na wyeliminowanie jednej ze składowych, natomiast wartość środkowa zakresu oznacza proporcję $1 : 1$. Ostatni z parametrów - `period` - kontroluje częstotliwość sygnału opóźnienia.

Wyraża on (podobnie jak w przypadku efektu tremolo) ilość cykli zegara systemowego przypadających na jedną próbkę LFO. Analogicznie do pozostałych modułów szerokości danych oraz poszczególnych parametrów mogą być ustalone za pomoca parametrów modułu (*generic*).



Rysunek 26: Struktura wyprowadzeń bloku *flanger*

Zasada działania modułu jest syntezą działania efektów tremolo i *delay*. W jego architekturze zaimplementowane zostały dwa procesy. Pierwszy z nich odpowiada za generowanie kolejnych próbek LFO (jak w przypadku tremolo). Z kolei drugi inicjalizuje działanie linii opóźniającej oraz uaktualnia zawartość bufora wyjściowego (analogicznie do modułu *delay*). Jeżeli wartość wejścia `depth` była niezerowa w momencie wykrycia stanu wysokiego na linii `valid in`, to na wyjściu pojawi się wynik działania $x[t] \times (1-S) + x[t-n] \times S$, gdzie S jest wartością na porcie `strength` interpretowaną jako liczba z przedziału $[0, 1)$. W przeciwnym wypadku zapisana zostanie w nim niezmodysikowana próbka wejściowa. Należy podkreślić, że - w przeciwieństwie do jednostki *delay* - wejście `input data` bloku opóźniającego jest podłączone do **bufora danych wejściowych** efektu, a nie danych wyjściowych.



Rysunek 27: Fragment symulacji działania efektu *flanger*

Na Rys. 27 przedstawiono fragment symulacji działania zaimplementowanego efektu dla częstotliwości oscylacji opóźnienia na poziomie 3Hz i amplitudy wartości 1024 próbek oraz zrównoważonej proporcji sygnału opóźnionego i wejściowego. Moment ustawienia linii `enable` w stan wysoki został oznaczony na rysunku żółtą flagą. Przebieg wartości wyjściowej modułu jest charakterystyczny dla działania efektu *flanger*.

8. Integracja potoku

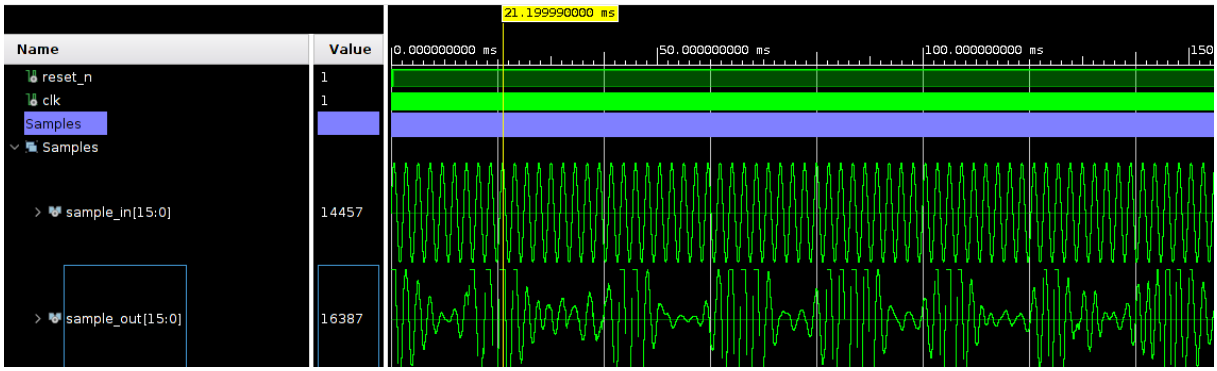
Po zakończeniu prac nad implementacją algorytmów przetwarzania przyszedł czas na ich zintegrowanie do postaci pojdeynczego modułu języka VHDL. Pierwszym krokiem do tak postawionego celu było ustalenie sposobu połączenia wszystkich efektów w potok. Zdecydowano się na kolejność tremolo > delay > flanger > overdrive. Był to wybór raczej arbitralny wynikający z braku doświadczenia autora w dziedzinie projektowania filtrów cyfrowych.

Kolejnym celem było ustalenie wartości parametrów poszczególnych efektów. W celu ułatwienie późniejszych rekonfiguracji systemu stworzony został pakiet `pipe_config` przechowujące stałe wykorzystywane do ich instancjonowania. W przypadku efektu **overdrive** przyjęto szerokość wejścia `gain` na poziomie 10 bitów. Zastosowano także przesówanie wewnętrznego wyniku mnożenia o 8 bitów. Umożliwi to kontrolę wzmocnienia sygnały w zakresie $[0, 4)$ z dokładnością ok. 0.39%. Efekt **tremolo** skonfigurowano do pracy z generatorem fali trójkątnej o 10-bitowych próbkach. Szerokość wejścia `depth` ustalono na 10 bitów. Z kolei szerokość wejścia `period` określono na poziomie 17 bitów. Pozwoli to na generowanie sygnału modulującego o częstotliwości z zakresu $[0.74, 97000]$Hz. W przypadku bloku **delay** zdecydowano się na wykorzystanie 12-bitowego wejścia `delay` oraz 16-bitowego wejścia `depth`. Parametry wykorzystanego do jego regalizacji modułu BRAM opisano we wcześniejszej części dokumentu. Ostatni z efektów - **flanger** - wymagał ustalenia wartości największej ilości parametrów. Szerokości wejść `strength`, `depth` i `period` ustawiono kolejno na 12, 10 i 20 bitów. Pozwoli to kontrolować siłę oraz amplitudę efektu z dostateczną dokładnością a jednocześnie zapewni możliwość ustawienia częstotliwość LFO w zakresie $[0.1, 97000]$Hz. Warto zauważyć, że zakres ten jest zdecydowanie zbyt szeroki dla efektu *flanger*. Aby zaprojektowany filtr mógł działać poprawnie częstotliwości te nie powinny przekraczać ok. 3Hz ([12]). Niedoskonałość ta wynika z przyjętego sposobu określania okresu fali modulującej i powinna zostać skorygowana na etapie podłączania wejścia `period` do wyjścia interfejsu analogowego. Do realizacji efektu *flanger* potrzebne były dwa moduły BRAM. Pierwszy z nich wykorzystywany jest przez generator fali sinusoidalnej. Przechowuje on 257 równo oddalonych, 10-bitowych próbek funkcji z przedziału $[0, 2\pi]$[13]. Przy próbkowaniu z częstotliwością 44100Hz pozwli to osiągnąć amplitudę opóźnień rzędu 23ms. Blok zastosowany do buforowania danych wejściowych został z kolei skonfigurowany do przechowywania 1024 16-bitowych próbek, które mogą zostać zaadresowane przez wartości LFO.

Ostatnim krokiem na drodze do integracji potoku było odpowiednie przeskalowanie wejść parametryzujących poszczególne efektu w celu uzyskania porządanych zakresów przy użyciu 12-bitowej reprezentacji próbek wyjściowych interfejsu analogowego. Proces ten został szczegółowo opisany w komentarzach kodu źródłowego potoku. W tym miejscu warto zwrócić jedynie uwagę na fakt, że wejścia `period` efektów *tremolo* oraz *flanger* zostały przeskalowane w taki sposób aby zakres pozycji potencjometrów odpowiadał prze-

[13] Amplituda funkcji została rozszerzona do pełnego zakresu 10-bitowej reprezentacji

działom $[0.74, 94]$Hz dla pierwszego z nich oraz $[0.1, 3]$Hz dla drugiego. Tak skonstruowany potok został jeszcze raz poddany symulacji celem weryfikacji poprawności przepływu danych przez poszczególne segmenty. Fragment jej wyników ukazano na Rys. 28.



Rysunek 28: Fragment symulacji działania ostatecznego potoku efektów

9. Integracja systemu

Ostatecznym etapem projektu było połączenie stworzonych w ten sposób modułów w ramach pojedynczego bloku języka VHDL. Ze względu na regularne testowanie wdrażanych elementów oraz z góry przemyślaną strukturę projektu proces ten przebiegł bezpoleśnie. Ostatecznie projekt udało się zsyntezować, czego efekt przedstawia Rys. 29.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
synth_1 (active)	**constrs_1**	**synth_design Complete!**								**1166**	**886**	**0.0**	**0**	**7**
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	19.646	0	1218	932	22.0	0	7
Out-of-Context Module Runs														
FlangerEffectGeneratorBram_synth_1	FlangerEffectGeneratorBram	synth_design Complete!								1	10	0.5	0	0
QuadrupletGeneratorTbBram_synth_1	QuadrupletGeneratorTbBram	synth_design Complete!								1	16	0.5	0	0
TremoloEffectBram_synth_1	TremoloEffectBram	synth_design Complete!								1	10	0.5	0	0
FlangerEffectBram_synth_1	FlangerEffectBram	synth_design Complete!								1	16	0.5	0	0
DelayEffectBram_synth_1	DelayEffectBram	synth_design Complete!								60	20	21.0	0	0
DelayLineTbBram_synth_1	DelayLineTbBram	synth_design Complete!								1	16	0.5	0	0
AnalogSequenceReaderXadc_synth_1	AnalogSequenceReaderXadc	synth_design Complete!								0	0	0.0	0	0

Rysunek 29: Efekt syntezy projektu

Niestety możliwość wykonania testów integracyjnych z wykorzystaniem symulacji była dość ograniczona w przypadku kompletnego projektu. Wynika to z faktu, że parametry wszytskich efektów określane są przez wartości pobierane przez interfejs analogowy z **pojedynczego** kanału. Przezwyciężenie tej trudności wymagałoby zmodyfikowania skryptu generującego plik z przebiegami sygnałów analogowych w taki sposób, aby uzględniał czasy próbkowania przetwornika ADC i zapisywał wartości napięcia odpowiadające aktualnie odczytywanego parametru. Ze względu na i tak już szeroki zakres projektu zrezygnowano z realizji takiego podejścia.

10. Podsumowanie

Zrealizowany projekt stanowił solidny wstęp do projektowania układów cyfrowych z wykorzystaniem języka VHDL[14]. Pozwolił on nie tylko zapoznać się ze składnią języka w stopniu umożliwiającym jego (stosunkowo) swobodne wykorzystywanie, ale także wdrożyć się w sposób myślenia w kategoriach konfiguracji zasobów sprzętowych tak różnego od tego, co znane jest z tworzenia oprogramowania. Dodatkowym benefitem płynącym z jego realizacji była możliwość poznania platformy projektowej w postaci oprogramowania *Vivado*, które umożliwi w przyszłości znacznie szybsze zainicjalizowanie właściwej części prac nad nowymi projektami[15].

Ilość pracy włożonej w projekt sprawiła, że stworzony w jego ramach kod postanowiono doprowadzić do postaci, w której będzie mógł on zostać wykorzystany z autentycznym sprzętem gitarowym. Wymagać to będzie stworzenia odpowiedniej platformy sprzętowej. Zadanie to obejmuje określenie wykorzystanych komponentów (ze szczególnym uwzględnieniem elementów analogowych) oraz (potencjalnie) zaprojektowanie własnego obwodu drukowanego. Na dzień dzisiejszy prace te zostają odłożone w czasie do momentu ukończenia projektów realizowany w ramach jakże służnie mijającego semestru.

[14] W niektórych kręgach można by nawet określić, że było to istny *skok na głęboką wodę*.

[15] Zapoznanie się z rzeczonym oprogramowaniem utwierdziło także autora w przekonaniu, że zintegrowane środowiska projektowe dostarczane przez dużych producentów nie są tym, co misie lubią najbardziej. W przypadku przyszłych projektów duży nacisk zostanie położony na wykorzystanie skryptów języka *Tcl*, które umożliwią jeszcze większą separację projektu od tego typu rozwiązań na rzecz środowiska stworzonego według własnych gustów i przyzwyczajeń.

Bibliografia

[1] (). „Introduction to UART,” adr.: https://www.weekitech.com/2019/05/13/introduction-to-uart/.

[2] (). „UART Explained,” adr.: https://developer.electricimp.com/resources/uart.

[3] (). „UART (VHDL),” adr.: https://www.digikey.com/eewiki/pages/viewpage.action?pageId=59507062.

[4] C. Robles, „An FPGA Implementation of Digital Guitar Effects,” 2019.

[5] Xilinx, „Vivado Design Suite User Guide: Design Flows Overwiev,” 2019.

[6] ——, „UltraFast Design Methodology Guide for the Vivado Design Suite,” 2019.

[7] ——, „7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1MSPS Analog-to-Digital Converted,” 2018.

[8] ——, „XADC Wizard v3.3: LogiCORE IP Product Guide,” 2016.

[9] ——, „7 Series FPGAs Memory Resources: User Guide,” 2019.

[10] ——, „Block Memory Generator v8.3: LogicCORE IP Product Guide,” 2017.

[11] (), adr.: https://github.com/kooltzh/xilinx-coe-generator.

[12] A. F. Ros, „Analysis of Flanging and Phasing Algorithms in Music Technology,” 2019.