Eksperymenty z FPGA (10)

Podłączamy mikrofon

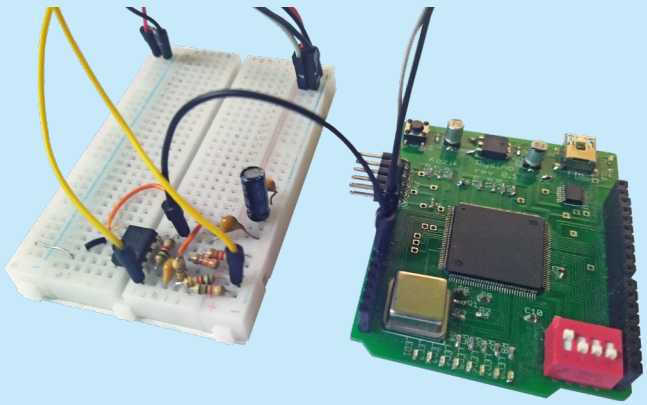
FTP WSZYSTKIE POPRZEDNIE ODCINKI KURSU SĄ DOSTĘPNE NA STRONIE WWW.MEDIA.AVT.PL

W tym odcinku wracamy do przetwornika ADC. Jednak zamiast potencjometru podłączymy do niego mikrofon. Dzięki temu wykonamy eksperymenty z cyfrowego przetwarzania sygnałów. Przed przystąpieniem do pracy jak zwykle przypominam o aktualizacji repozytorium z przykładami (poprzez wywołanie polecenia `git pull`).

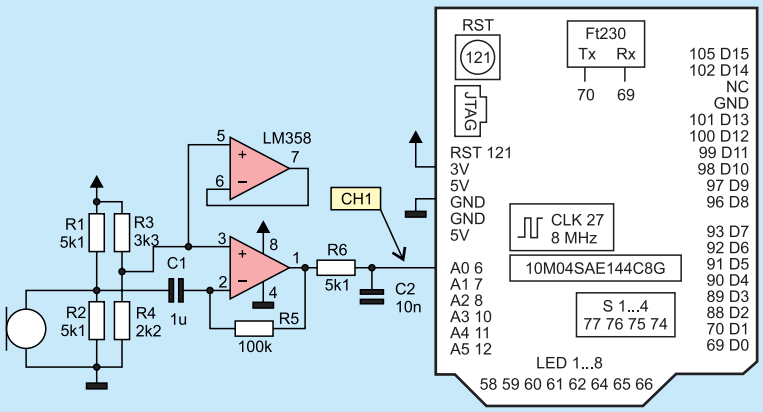
Do eksperymentów zastosujemy popularny mikrofon elektretowy. Niestety zmiany napięcia na jego wyjściu są zbyt małe, aby można je było zmierzyć za pomocą naszego przetwornika ADC. Musimy więc zbudować prosty przedwzmacniacz. Użyjemy do tego celu wzmacniacza operacyjnego LM358 i kilku elementów pasywnych. Schemat układu został pokazany na **rysunku 1**. Rezystory R3 i R4 tworzą dzielnik napięcia ustalający „sztuczną masę" dla naszego przedwzmacniacza. Kondensator C1 usuwa składową stałą z wejścia wzmacniacza operacyjnego. Zmieniając wartość R5, możemy regulować wzmocnienie. Na wyjściu znajduje się prosty filtr RC złożony z R6 i C2. Jego częstotliwość graniczna to:

$$f = \frac{1}{2\pi R_6 C_2} \approx\sim 3100Hz$$

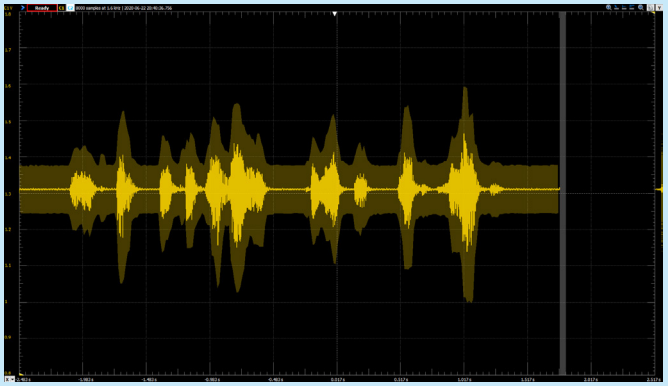
Sygnał podłączamy do analogowego wejścia A0 płytki *Rysino*. Musimy jeszcze zatroszczyć się o niewykorzystaną połowę LM358. Wyjście drugiego wzmacniacza zostało połączone z jego wejściem odwracającym, natomiast wejście nieodwracające zwarto do masy. Realizacja na płytce stykowej jest pokazana na **fotografii 1**. Działanie



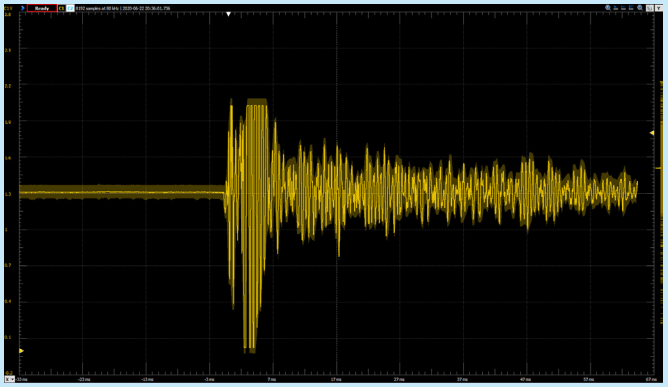
Fotografia 1. Układ eksperymentalny zbudowany na płytce stykowej



Rysunek 1. Schemat podłączenia mikrofonu



Rysunek 2. Napięcia na wejściu ADC przy rejestracji mowy

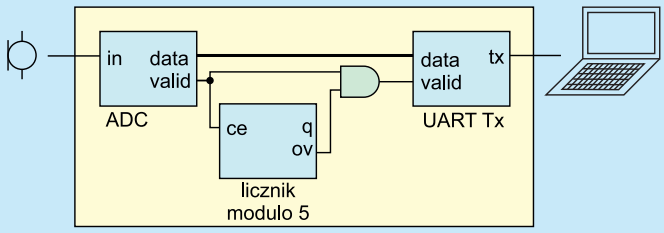


Rysunek 3. Napięcia na wejściu ADC przy rejestracji klaśnięcia

układu najłatwiej sprawdzić za pomocą oscyloskopu. Na **rysunku 2** widoczna jest rejestracja mowy. Widać na nim, że sztuczna masa naszego układu odpowiada napięciu około 1,3 V. Natomiast rejestrowane sygnały znajdują się w przedziale napięciowym 1,15 V...1,45 V. Aby sprawdzić, przy jakich wartościach następuje nasycenie, na **rysunku 3** pokazano zapis klaśnięcia. Wynika z niego, że roboczy zakres napięciowy zawiera się pomiędzy wartościami około 0,05 V a 2 V.

Rejestracja dźwięku

Kiedy mamy już gotowy układ doświadczalny, możemy zabrać się do przygotowania wsadu dla układu FPGA. Naszym pierwszym celem będzie wysłanie pomiarów z przetwornika ADC przez port szeregowy. Uproszczony schemat tego modułu pokazuje **rysunek 4**. Aby nie komplikować projektu, będziemy przesyłać jedynie 8 bitów danych. Musimy także zdecydować się na częstotliwość próbkowania. Przyjmuje się, że w mowie występują częstotliwości do około 3 kHz. Oznacza to, że wystarczyłoby

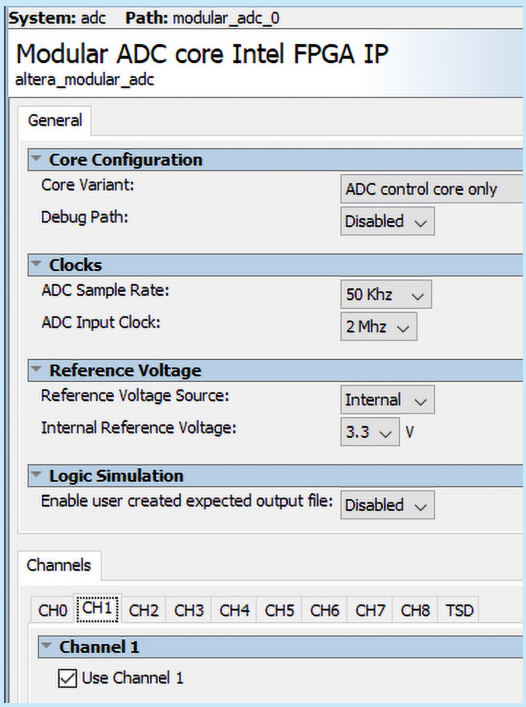


Rysunek 4. Schemat rejestratora dźwięku

próbkować z częstotliwością około 6 kHz. Jednak nie możemy wybrać dowolnej wartości. Ich lista jest dostępna w dokumentacji [1]. Kiedy wybierzemy 50 kHz, możemy łatwo otrzymać próbkowanie z częstotliwością 10 kHz poprzez zwykłe odrzucanie czterech i przepuszczanie co piątej wartości. Uzyskamy to przez wygaszanie sygnału `valid`. Czynność ta spowoduje, że dane mimo że przejdą przez magistralę, nie zostaną wysłane. Wykorzystamy tu dobrze nam znany licznik modulo. Do przesłania dziesięciu tysięcy 8-bitowych pomiarów, po doliczeniu bitów startu i stopu, uzyskamy minimalną prędkość transmisji równą 100 kbd. Możemy więc skonfigurować UART do standardowej prędkości 115,200 kbd, tak jak w poprzednich eksperymentach, i nadal będziemy mieli pewien zapas.

Budowę rozpoczniemy od wygenerowania bloku IP, obsługującego przetwornik analogowo-cyfrowy. Konfigurację pokazuje **rysunek 5**. Ustawiamy tam wybraną przez nas częstotliwość próbkowania. Zegar taktujący ADC ustawiamy na 2 MHz (będziemy musieli także odpowiednio skonfigurować pętlę PLL). Tym razem włączamy tylko kanał numer 1.

Główny moduł naszego projektu pokazuje **listing 1**. W liniach 30…44 mamy instancję modułu ADC. W tym przypadku konfiguracja magistrali wejściowej jest bardzo prosta, ponieważ chcemy czytać tylko jeden kanał z maksymalną częstotliwością. Wpisujemy numer kanału w sygnale `command_channel` i ustawiamy `command_valid` na 1. W linii 50 ustawiamy wysyłanie tylko ośmiu najstarszych bitów. Dalej znajduje się moduł licznika, który zlicza kolejne pomiary z przetwornika, a dla co piątej próbki następuje ustawienie sygnału `ce_5`. Linia 59 przygotowuje sygnał wyzwalający transmisję. Musimy zastosować bramkę AND, ponieważ sygnał przepełnienia licznika pozostanie aktywny, aż zakończy się następny pomiar. Mogłoby to spowodować



Rysunek 5. Konfiguracja przetwornika ADC

Listing 1. Kod modułu nagrywającego (10_record/rec.sv)

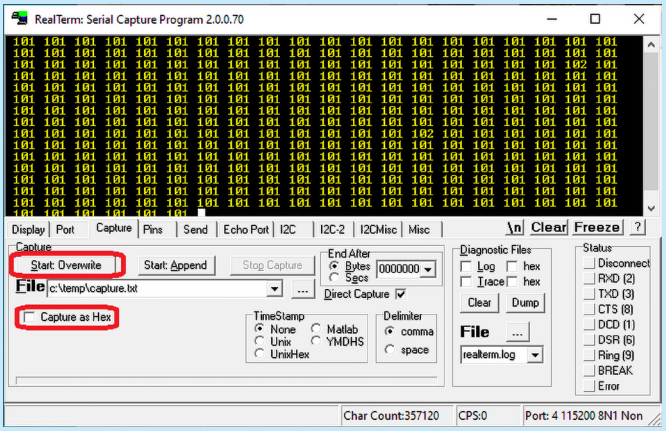
```
30 adc adc_inst (
31  .clock_clk(clk),
32  .reset_sink_reset_n(rst),
33  .adc_pll_clock_clk(clk_adc),
34  .adc_pll_locked_export(clk_adc_locked),
35  .command_valid(1'b1),
36  .command_channel(1'b1),
37  .command_startofpacket(1'b0),
38  .command_endofpacket(1'b0),
39  .command_ready(),
40  .response_valid(adc_valid_out),
41  .response_channel(),
42  .response_data(adc_data_out),
43  .response_startofpacket(),
44  .response_endofpacket());

50 assign bus.data = adc_data_out[11-:8];
51
52 counter #(.N(5)) every5 (
53  .clk(clk),
54  .rst(rst),
55  .ce(adc_valid_out),
56  .q(),
57  .ov(ce_5));
58
59 assign bus.valid = ce_5 & adc_valid_out;
60
61 uart_tx #(
62  .F(F),
63  .BAUD(115200)
64 ) uart (
65  .bus(bus),
66  .tx(tx));
```

kilkukrotne wysłanie pojedynczej próbki. Teoretycznie nie powinno tak się zdarzyć, ponieważ wysłanie jednego bajtu zajmuje więcej czasu niż wynosi odstęp pomiędzy pomiarami przetwornika. Na samym końcu umieszczony został moduł nadajnika portu szeregowego.

Kiedy mamy już gotowy kod, możemy otworzyć w środowisku *Quartus* projekt *10_record/10_rec.qpf* i rozpocząć jego budowę. Po wgraniu bitstreamu do układu FPGA przechodzimy do odczytywania danych. W tym celu można wykorzystać znany nam już program *RealTerm*. Aby wyświetlane dane były bardziej czytelne, możemy w zakładce *Display* wybrać opcję *uint8* (*unsigned int 8* – 8-bitowa liczba całkowita bez znaku), co pokazuje **rysunek 6**.

Rysunek 6. Wybór sposobu wyświetlania danych

Kiedy powiemy coś do mikrofonu, dane powinny się zmieniać, ale pojawiają się zbyt szybko, by zaobserwować to wyraźnie na ekranie konsoli. Dlatego zapiszemy odbierane informacje w pliku tekstowym do późniejszej analizy. W tym celu otwieramy zakładkę *Capture* (przechwytywanie), co zaprezentowano na **rysunku 7**. Następnie wybieramy plik docelowy i zaznaczamy opcję *Capture as Hex* (przechwytuj w formacie szesnastkowym). Rozpoczynamy naciśnięciem *Start: Overwrite* (zacznij: nadpisz), mówimy do mikrofonu i klikamy przycisk *Stop Capture* (zatrzymaj przechwytywanie).



Rysunek 7. Przechwytywanie danych w programie RealTerm

Listing 2. Skrypt w języku Python parsujący odebrane dane (10_record/parse/Parse.ipynb)

```
01 %matplotlib inline
02
03 import matplotlib
04 import numpy as np.
05 import matplotlib.pyplot as plt
06 from scipy.io.wavfile import write
07
08 def pairwise(iterable):
09   "s -> (s0, s1), (s2, s3), (s4, s5), …"
10   a = iter(iterable)
11   return zip(a, a)
12
13 file="voice.txt"
14 with open(file) as f:
15  data = f.read()
16 x = []
17 for a,b in pairwise(data):
18  s = a + b
19  x.append(int(s, 16))
20 xn = np.uint8(x)
21
22 plt.plot(xn)
23 plt.xlabel("czas, próbki")
24 plt.ylabel("amplituda, j.w.")
25
26 write('voice.wav', 10000, xn)
```

Wykonane przeze mnie przykładowe nagranie znajduje się w pliku *10_record/parse/voice.txt*. Kiedy go otworzymy, zobaczymy jedną, długą linię bez żadnych przerw:

```
656565656565
```

Każde dwa kolejne znaki odpowiadają kolejnym odebranym bajtom. Są one zapisane w formacie szesnastkowym. Aby je zdekodować oraz zmienić na format dźwiękowy, przygotujemy krótki skrypt w języku Python. Do jego uruchomienia wykorzystam notatnik *Jupyter*. Można go pobrać razem z pakietem *Anaconda* [2]. Domyślnie notatnik jako swój główny katalog uznaje folder użytkownika. Jeżeli przechowujemy pliki projektu w innym miejscu, można uzyskać do nich dostęp poprzez stworzenie dowiązania do wybranego katalogu, na przykład wywołaniem w wierszu poleceń komendy:

```
mklink /D c c:\
```

Powyższe działanie spowoduje stworzenie dowiązania *c*, które wskazuje na dysk *C:*. Parametr `/D` informuje, że mamy do czynienia z katalogiem. Teraz możemy załadować notatnik *10_record/parse/Parse.ipynb*. Zawarty w nim kod pokazuje **listing 2**. W pierwszej linii konfigurujemy wyświetlanie wykresów. Następnie ładujemy niezbędne biblioteki. *Matplotlib* umożliwi nam tworzenie wykresów, a *Numpy* pozwala na wykonywanie obliczeń numerycznych. *Scipy* jest rozbudowaną biblioteką ułatwiającą obliczenia naukowe. Wykorzystamy z niej jedynie moduł, pozwalający zapisywać pliki w dźwiękowym formacie *wav*. W liniach 8…11 znajdziemy funkcję `pairwise`, która ułatwi nam wczytywanie dwóch kolejnych znaków. Wiersz 13 zawiera nazwę pliku z zapisanymi danymi. Otwieramy go i wczytujemy zawartość do zmiennej `data`. Następnie w pętli wczytujemy po dwa znaki i konwertujemy je na liczbę. W linii 20 konwertujemy otrzymaną tablicę na wektor liczb 8-bitowych bez znaku. Na końcu wyrysowujemy wykres z zebranych danych i zapisujemy plik dźwiękowy *voice.wav*.

Uzyskany wykres pokazuje **rysunek 8**. Sygnał oscyluje wokół liczby 100. Jego wartości mieszczą się w przedziale 80…115. Oznacza to, że większość z dostępnego nam zakresu (0…255) jest niewykorzystana. Używamy jedynie przedziału o szerokości poniżej 35. Odpowiada on nieco ponad pięciu bitom danych, przy czym przesyłamy ich aż osiem. W dalszej części spróbujemy poprawić ten wynik.

Sposób zbierania danych oraz konwersji ich do pliku tekstowego został także przedstawiony na filmie [3].

Usuwanie składowej stałej

Jak zauważyliśmy już przy analizie na rysunku 8, nasz sygnał z mikrofonu nie oscyluje wokół 0, co oznacza, że znajduje się w nim pewna składowa stała. Jest ona równa napięciu na dzielniku, złożonym z oporników R3 i R4. Istnieje kilka sposobów, aby ją usunąć. Możemy ustalić stałą, którą będziemy odejmować od pomiarów, albo mierzyć napięcie na dzielniku za pomocą drugiego kanału przetwornika. Zastosujemy jednak filtr górnoprzepustowy o nieskończonej odpowiedzi impulsowej: NOI. Często spotyka się także oznaczenie IIR, co pochodzi od angielskiego *Infinite Impulse Response* [4]. Jego transmitancję wylicza się wzorem:

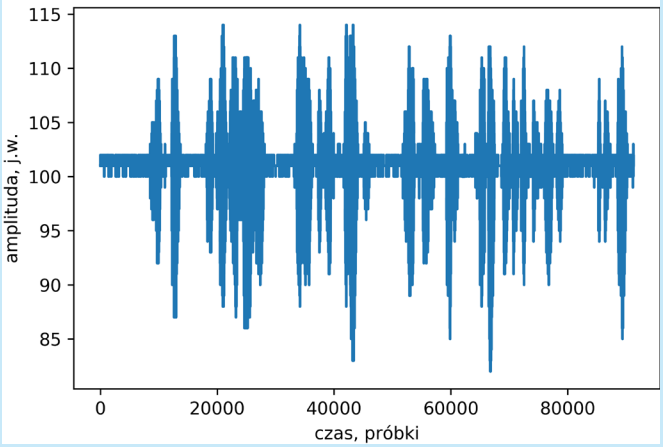
$$H(z)=\frac{1-z^{-1}}{1-\alpha z^{-1}}$$

Taki filtr ma interesującą charakterystykę amplitudową, która wynosi 0 dla składowej stałej, a następnie szybko rośnie. Wyrysujemy ją w notatniku *Jupyter*. Odpowiedzialny za to kod pokazany jest na **listingu 3**. Na początku przyjmujemy współczynnik α równy 31/32. Warto tu zwrócić uwagę, że język Python pozwala nam używać greckich liter jako nazw zmiennych. Można dyskutować, czy jest to dobra praktyka, ale jest to dopuszczalne. Korzystając z biblioteki *Scipy*, definiujemy transmitancję H. W formie wektorów podajemy współczynniki wielomianów, składających się na jej licznik oraz mianownik. Ostatni parametr to okres próbkowania. Podajemy tu 100 µs, co odpowiada częstotliwości 10 kHz, z którą pracuje nasz przetwornik. Na końcu generujemy i wyrysowujemy charakterystykę amplitudową (zwaną także charakterystyką Bodego). Uzyskany wykres pokazuje **rysunek 9**.

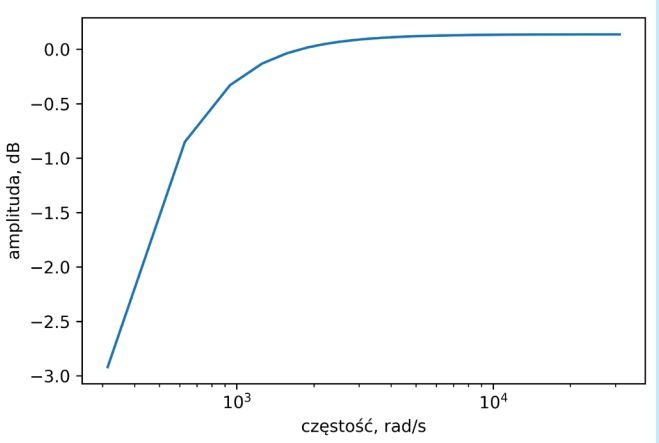
Przed przygotowaniem realizacji filtra w strukturze układu FPGA najpierw pokażmy jego transmitancję w formie graficznej.

Listing 3. Generowanie charakterystyki amplitudowej w Pythonie (11_dc_r\parse\dc_r.ipynb)

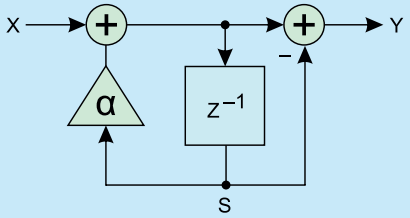
```
01 α = 31/32
02 H = signal.TransferFunction([1, -1], [1, -α], dt=1e-4)
03 w, mag, phase = H.bode()
04 axi = plt.semilogx(w, mag)
05 plt.xlabel("częstość, rad/s")
06 plt.ylabel("amplituda, dB")
```



Rysunek 8. Wykres wygenerowany z zebranych danych



Rysunek 9. Charakterystyka amplitudowa dla α=31/32



Rysunek 10. Schemat filtra realizującego transmitancję H(z)

Na **rysunku 10** widzimy jedną z wielu możliwych realizacji. X symbolizuje wejście, a Y wyjście. Blok z^{-1} oznacza operację opóźnienia o jeden krok. Możemy łatwo pokazać, że nasz schemat naprawdę odpowiada transmitancji H. Najpierw rozpiszmy wartość w punkcie oznaczonym jako S:

$$S = z^{-1}\left(X + \alpha S\right)$$

Po przeniesieniu S na jedną stronę równania i uporządkowaniu współczynników otrzymamy:

$$S = \frac{z^{-1}}{1 - z^{-1}\alpha} X$$

Teraz możemy rozpisać równanie na wyjście:

$$Y = X + \alpha S - S = X - \left(1 - \alpha\right) S$$

Podstawiając wcześniej obliczoną wartość S, otrzymujemy:

$$Y = X - \left(1 - \alpha\right)\frac{z^{-1}}{1 - z^{-1}\alpha} X = \frac{1 - z^{-1}}{1 - \alpha z^{-1}} X$$

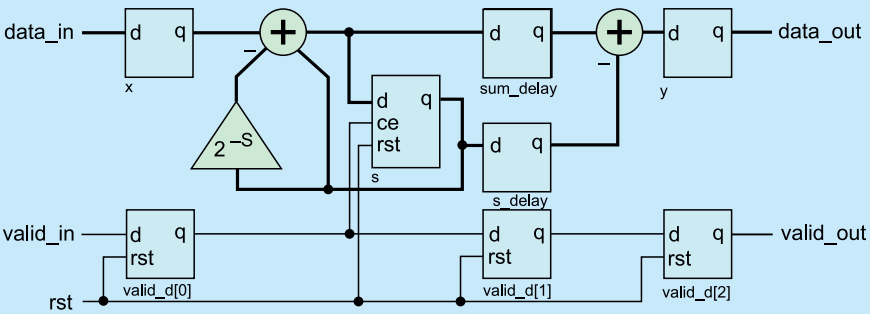
Stąd dzieląc przez X, otrzymujemy naszą wyjściową transmitancję:

$$H(z) = \frac{Y}{X} = \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

Przekładając schemat z rysunku 10 na coś, co można zaimplementować w strukturze układu FPGA, otrzymujemy bardziej złożony schemat pokazany na **rysunku 11**. Pierwsza rzecz, która rzuca się w oczy, to znacznie większa liczba przerzutników. Zostały one dodane w celu uzyskania wyższej częstotliwości pracy (co prawdopodobnie nie jest potrzebne w naszym, taktowanym z niską częstotliwością 8 MHz projekcie). Zwracam uwagę, że jedynym przerzutnikiem z wejściem ce jest ten oznaczony jako *S*. Jest on włączany przez sygnał `valid`. To główna różnica pomiędzy opóźnieniami wynikającymi z logiki projektu (z^{-1}) a tymi „technicznymi", ułatwiającymi realizację w krzemie. W punkcie *S* ma zostać zapisana ostatnia „poprawna" (*valid*) wartość.

Usuwanie składowej stałej realizowane jest w trzech krokach. Pierwszy polega na zatrzaśnięciu wejść. Odpowiadają za to przerzutniki `x` i `valid_d[0]`. W drugim kroku przygotowujemy cząstkowe wyniki i zapisujemy je w rejestrach `sum_delay` i `s_delay`. Razem z nimi, w `valid_d[1]` podąża sygnał sterujący. W ostatnim etapie, w przerzutnikach `y` i `valid_d[2]` zapisujemy wyjście filtru. Warto zauważyć, że sygnał kontrolny `valid` niejako płynie przez filtr równolegle z danymi.

Drugi szczegół implantacyjny polega na założeniu, że współczynnik α będzie miał postać $1–2^S$, gdzie s jest liczbą naturalną. Pozwoli to zastąpić mnożenie za pomocą przesunięcia bitowego i odejmowania. Są to operacje dużo łatwiejsze w realizacji (w FPGA) niż mnożenie.



Rysunek 11. Realizacja filtru w układzie FPGA

Listing 4. Moduł usuwający składową stałą (11_dc_r/dc_r.sv)

```
10 module dc_r #(
11  parameter N = 8,
12  parameter S = 5
13 ) (
14  StreamBus in,
15  StreamBus out
16 );
17  parameter L = N+S;
18  logic [2:0]valid_d;
19  logic signed [N:0]x;
20  logic signed [L:0]s;
21  logic signed [L:0]sum;
22  logic signed [L:0]s_delay;
23  logic signed [L:0]sum_delay;
24  logic signed [L+1:0]y;
25
26  always_ff @(posedge in.clk or negedge in.rst)
27  if (!in.rst)
28      valid_d <= '0;
29  else
30      valid_d <= {valid_d[1:0], in.valid};
31
32  always_ff @(posedge in.clk)
33  x <= {1'd0, in.data};
34
35  assign sum = s - (s >>> S) + x;
36
37  always_ff @(posedge in.clk or negedge in.rst)
38  if (!in.rst)
39      s <= '0;
40  else if (valid_d[0])
41      s <= sum[L:0];
42
43  always_ff @(posedge in.clk) begin
44      s_delay <= s;
45      sum_delay <= sum[L:0];
46  end
47
48  always_ff @(posedge in.clk)
49  y <= sum_delay - s_delay;
50
51  assign out.data = y[N-1:0];
52  assign out.valid = valid_d[2];
53 endmodule
```

Musimy jeszcze zastanowić się, ile bitów musimy przeznaczyć na reprezentację stanu. W tym celu przejdźmy do opisu w dziedzinie czasu. Możemy wyrazić wartość w rejestrze s w chwili k+1 jako:

$$s_{k+1} = \alpha s_k + x_k$$

Dla danego (stałego) wejścia x po pewnym czasie stan się ustali. Oznacza to, że wartość s_{k+1} będzie równa s_k. Możemy wtedy zapisać:

$$s = \alpha s + x$$

I obliczyć wartość stanu ustalonego:

$$s = \frac{1}{1 - \alpha} x$$

Gdy podstawimy przyjętą przez nas wcześniej wartość współczynnika:

$$\alpha = 1 - 2^{-S}$$

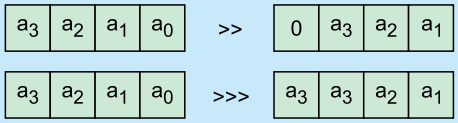
Otrzymamy zależność:

$$s = 2^S x$$

Teraz, jeżeli nasze wejście x ma długość N bitów, to do zapisania stanu s potrzebować będziemy L=N+S bitów.

Mamy już wszystkie niezbędne informacje, aby rozpocząć implementację. Kod napisany w języku SystemVerilog znajdziemy na **listingu 4**. Na początku definiujemy dwa znane nam już parametry. N oznacza liczbę bitów przeznaczonych na wejście, a `S` ustala wartość współczynnika α. Następnie utworzone są dwa interfejsy: wejściowy `in` i wyjściowy `out`. Dla uproszczenia nie będziemy implementować sygnału `ready`. W liniach 17…24 zdefiniowane zostały sygnały wykorzystywane wewnątrz modułu. Te, które przechowują liczby ze znakiem, mają dodatkowy parametr `signed`.

Pierwszy blok `always` (linie 26…30) buduje linie opóźniającą dla sygnału `valid`. W wierszach 32…33 zatrzaskujemy wartość wejściową w rejestrze x. Dodatkowo, poprzez dołączenie na początek bitu równego `0`, dokonujemy konwersji z liczby bez znaku do dodatniej liczby ze znakiem. Z tego powodu, mimo że wchodząca wartość jest zapisana na N bitach, nasz rejestr `x` musi mieć długość N+1 bitów.



Rysunek 12. Różnica pomiędzy logicznym (>>) i arytmetycznym (>>>) przesunięciem bitowym

Listing 5. Testbench dla modułu dc_r (11_dc_r/dc_r_tb.sv)

```
12 module dc_r_tb;
13  parameter N = 10;
14  logic clk, rst;
15  StreamBus #(.N(N)) in (.clk(clk), .rst(rst));
16  StreamBus #(.N(N)) out (.clk(clk), .rst(rst));
17
18  initial begin
19      clk <= 1'b0;
20      forever #1 clk <= ~clk;
21  end
22
23  initial begin
24      rst <= 1'b0;
25      #4 rst <= 1'b1;
26  end
27
28  initial begin
29      in.valid = 1'b1;
30      for (int i = 0; i < 1000; i++) begin
31          @(posedge clk);
32          in.data = 120*$sin(2*pi*i/100)+480;
33      end
34      $stop;
35  end
36
37  dc_r #(.N(N), .S(5)) dut (
38  .in(in),
39  .out(out));
40 endmodule
```

Linia 35 definiuje logikę kombinacyjną, która oblicza wartość pomocniczą sum. Korzystamy tu z operatora arytmetycznego przesunięcia bitowego. W przeciwieństwie do przesunięcia logicznego, zachowuje ono znak. Różnica pomiędzy nimi została pokazana na **rysunku 12**.

Blok synchroniczny z linii 37…41 realizuje obsługę stanu. Wartość początkowa, ustawiana przy resecie, to 0. Natomiast wartość z sygnału sum jest zatrzaskiwana tylko wtedy, gdy (odpowiednio opóźniony) sygnał `valid` jest aktywny. Następnie w wierszach 43…46 wykonujemy drugi krok obliczeń. W rejestrach `s_delay` i `sum_delay` zatrzaskujemy wartości, których użyjemy do obliczeń. Wynik uzyskujemy w kroku numer trzy (48…49). Zwracam uwagę, że do zapisania różnicy dwóch liczb ze znakiem potrzebujemy rejestru o jeden bit dłuższego niż najdłuższa z wejściowych liczb. Jednak w naszym przypadku z powrotem na wyjście wypisujemy jedynie N najmłodszych bitów.

Musimy teraz przygotować testbench dla nowego modułu, którego kod pokazuje **listing 5**. Najpierw definiujemy parametr `N`, który określa długość szyny danych. Wykorzystamy go dalej do stworzenia interfejsów: wejściowego i wyjściowego. Następnie w dwóch osobnych blokach `initial` generujemy sygnał zegarowy oraz reset. Linie 28…35 odpowiadają za stworzenie danych. Tworzymy sinusoidę o amplitudzie 120, z przesunięciem 480. Generujemy w pętli 1000 próbek, po czym kończymy symulację. Na samym końcu został umieszczony testowany moduł. Włączamy symulację, wywołując w programie *ModelSim* komendę:

```
do dc_r_sim.do
```

Skrypt jest bardzo podobny do tych, których używaliśmy wcześniej. Warto jednak zwrócić uwagę na nowy przełącznik `-decimal`. Powoduje on, że dany sygnał jest interpretowany jako liczba ze znakiem i wyświetlany w systemie dziesiętnym.

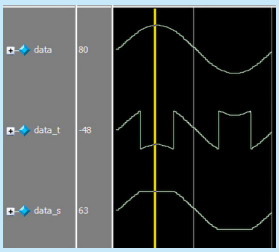
Wynik symulacji widoczny jest na **rysunku 13**. Dwa pierwsze wiersze to sygnał zegarowy oraz reset. Następnie widzimy dane wejściowe i ich wersję skonwertowaną na liczbę ze znakiem (oznaczone jako `x`). Dalej mamy cząstkowy sygnał `sum` i `s`. Możemy zaobserwować, jak narastają one w początkowym okresie. Powiązane jest to ze znikaniem składowej stałej w wyjściowym sygnale `data`, pokazanym w ostatnim wierszu. Po ustabilizowaniu się sygnału wyjściowego otrzymujemy sygnał oscylujący wokół zera. Na zaprezentowanym rysunku nie są widoczne wszystkie detale, dlatego zachęcam do samodzielnego uruchomienia symulacji.

Nasycenie

Wiemy już w jaki sposób usunąć składową stałą. Nadal jednak musimy zmniejszyć liczbę bitów. Dwie najbardziej popularne opcje to obcięcie albo nasycenie. Ich porównanie widzimy na **rysunku 14**, wygenerowane w programie *ModelSim* za pomocą komendy:

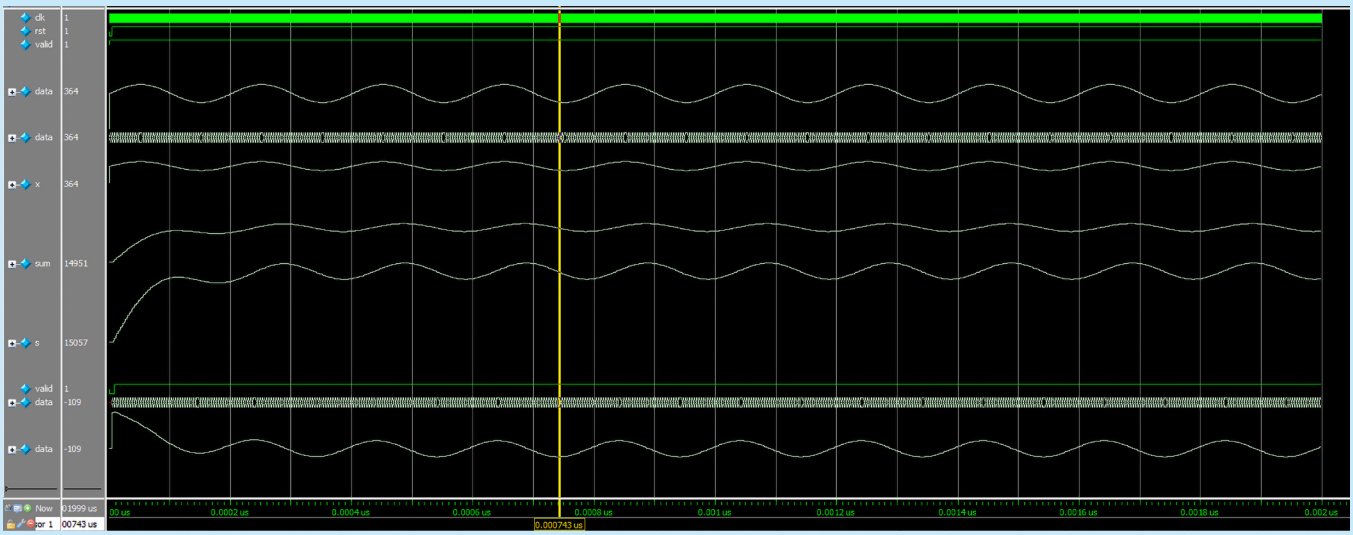
```
do truncation_saturation.do
```

Kod do generowania pokazuje **listing 6**. Na górnym wykresie widać zawartość 8-bitowego rejestru data. Linia 21 wskazuje, że zawiera on sinusoidę o amplitudzie 80. Kolejne dwa wykresy zawierają tę samą funkcję, ale przyciętą do długości siedmiu bitów. Możliwa jest prezentacja sygnału z zakresu –64…63. Następuje więc strata części informacji.



Rysunek 14. Porównanie obcięcia i nasycenia

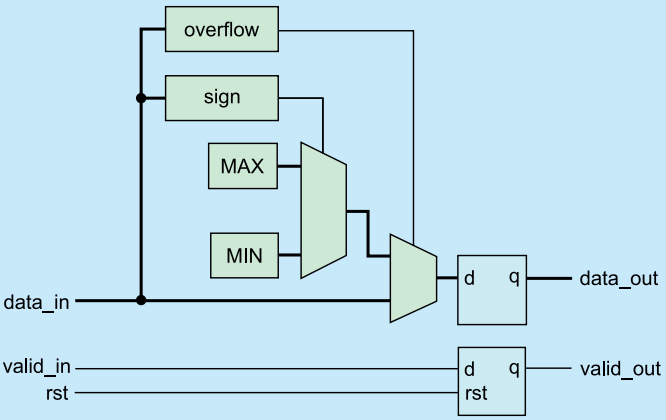
Zmienna `data_t` pokazuje co się stanie, gdy odetniemy najstarszy bit. Kiedy przekraczamy maksymalną wartość, bit znaku, który wcześniej był równy zero, zostaje nadpisany jedynką, pochodzącą



Rysunek 13. Wynik symulacji modułu dc_r

Listing 6. Generowanie wykresów z rysunku 15 (11_dc_r/truncation_saturation.sv)

```
12 module truncation_saturation;
13  parameter N = 8;
14  logic signed [N-1:0]data;
15  logic signed [N-2:0]data_t;
16  logic signed [N-2:0]data_s;
17
18  initial begin
19      for (int i = 0; i < 100; i++) begin
20          #1;
21          data = 80*$sin(2*pi*i/100);
22          data_t = data[N-2:0];
23          if (data > 2**(N-2)-1)
24              data_s = 2**(N-2)-1;
25          else if (data < -(2**(N-2)))
26              data_s = -2**(N-2);
27          else
28              data_s = data[N-2:0];
29      end
30      $stop;
31  end
32 endmodule
```



Rysunek 15. Realizacja saturacji w układzie FPGA

z danych, co powoduje zmianę znaku. W przypadku przetwarzania sygnałów dźwięku jest to zjawisko bardzo niekorzystne. Sygnał opisany jako `data_s` prezentuje sytuację, gdy zastosujemy nasycenie. Uzyskany przebieg jest bliższy wartościom wejściowym. Przed dokonaniem obcięcia sprawdzamy, czy nastąpiło przepełnienie. Jeżeli tak, to zamiast wejściowych danych na wyjście podajemy minimalną albo maksymalną możliwą wartość – w zależności od znaku odebranych danych.

Przykładową realizację modułu pokazuje **rysunek 15**. Mamy w nim dwa bloki kombinacyjne. Pierwszy z nich, nazwany `overflow`, wykrywa, czy przy obcięciu wystąpi przepełnienie. Na jego

Listing 7. Moduł realizujący skalowanie z nasyceniem(11_dc_r/saturation.sv)

```
10 module saturation #(
11  parameter N_IN = 10,
12  parameter N_OUT = 8
13 ) (
14  StreamBus in,
15  StreamBus out
16 );
17 logic overflow, sign;
18 logic [N_IN-1:N_OUT-1]truncated;
19 logic [N_OUT-1:0]data_out;
20
21  always_comb begin
22      sign = in.data[N_IN-1];
23      truncated = in.data[N_IN-1:N_OUT-1];
24      overflow = &truncated != |truncated;
25      if (overflow)
26          data_out = (sign) ? 2**(N_OUT-1) : 2**(N_OUT-1)-1;
27      else
28          data_out = in.data[N_OUT-1:0];
29  end
30
31  always_ff @(posedge in.clk)
32  out.data <= data_out;
33
34  always_ff @(posedge in.clk or negedge in.rst)
35  if (!in.rst)
36      out.valid <= '0;
37  else
38      out.valid <= in.valid;
39
40 endmodule
```

Listing 8. Testbench dla modułu stauration (11_dc_r/saturation_tb.sv)

```
12 module saturation_tb;
13  parameter N_IN = 10;
14  parameter N_OUT = 8;
15
16  logic clk, rst;
17  StreamBus #(.N(N_IN)) in (.clk(clk), .rst(rst));
18  StreamBus #(.N(N_OUT)) out (.clk(clk), .rst(rst));
19
30  initial begin
31      in.valid = 1'b1;
32      for (int i = -2**(N_IN-1); i < 2**(N_IN-1); i++) begin
33          @(posedge clk);
34          in.data = i;
35      end
36      $stop;
37  end
38
39  saturation dut (
40      .in(in),
41      .out(out)
42  );
43 endmodule
```

podstawie zewnętrzny multiplekser ustala, czy przekazujemy na wyjście wejściowe dane, czy którąś ze stałych. Wyboru stałej dokonuje pierwszy multiplekser na podstawie znaku wejściowych danych. Jest on zwracany przez blok oznaczony jako `sign`. Na końcu wynik jest zatrzaskiwany w rejestrze. Jak zwykle sygnał `valid` płynie razem z danymi przez drugi, tym razem resetowany przerzutnik.

Przejdźmy teraz do implementacji, którą pokazuje **listing 7**. Moduł `saturation` przyjmuje dwa parametry: `N_IN` i `N_OUT`, określające szerokość wejścia i wyjścia. Dalej, podobnie jak w module `dc_r`, mamy zdefiniowane interfejs wchodzący i wychodzący z modułu. Główna część logiki znajduje się w bloku `always_comb`. W linii 22 wykrywamy znak wejściowych danych. Jest to bardzo prosta operacja: wystarczy wyciągnąć najstarszy bit. Następnie przygotowujemy wektor zawierający wszystkie bity, które zostaną obcięte i jeden (najstarszy), który zostanie nowym bitem znaku. Wektor ten został nazwany `truncated`. W kolejnym wierszu wykrywamy, czy nastąpiło przepełnienie. Korzystamy tutaj z prostej zależności. Jeśli nie nastąpiło przepełnienie, wtedy wszystkie obcięte bity (oraz nowy bit znaku) mają tę samą wartość. Jest to równoważne stwierdzeniu, że logiczna operacja i wykonana na tym wektorze dadzą taki sam wynik jak logiczne `OR`. Następnie w liniach 25…29 zrealizowane są oba multipleksery z rysunku 16. Pierwszy za pomocą operatora `?:`, a drugi z wykorzystaniem instrukcji `if/else`.

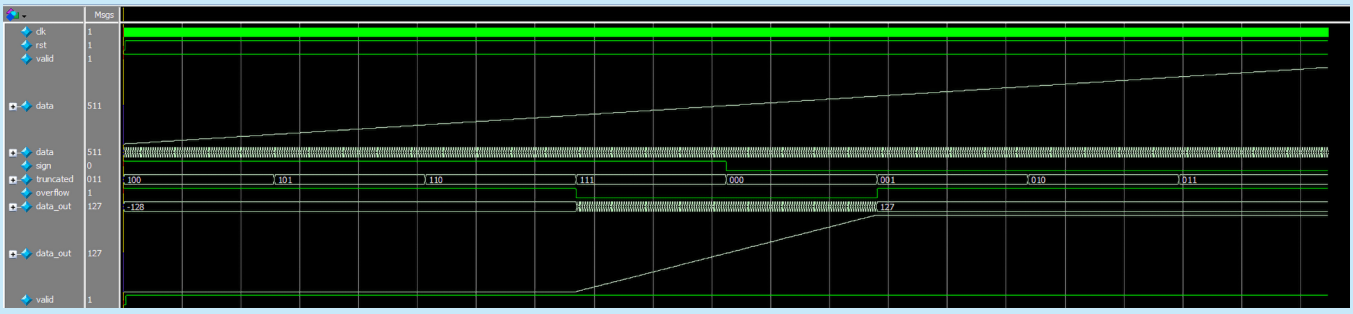
Zostało nam już tylko zaimplementowanie dwóch przerzutników, zatrzaskujących obliczone wyjście. Pierwszy z nich (31…32) obsługuje dane, a drugi (34…38), wyposażony w reset, dba o kontrolny sygnał `valid`.

Testbench dla naszego modułu znajduje się na **listingu 8**. Na początku definiujemy dwa parametry określające rozmiar wejścia i wyjścia. Zostały one później wykorzystane przy tworzeniu interfejsów. Generowanie sygnałów zegarowego i resetu jest identyczne jak w poprzednim testbenchu, dlatego zostały pominięte. Zmiana następuje przy generowaniu danych. Za pomocą pętli `for` (linia 32) generujemy funkcje liniową, która przechodzi przez wszystkie dopuszczalne wartości wejścia. Dla 10 bitów rozpoczniemy więc od –512 i dojdziemy do wartości 511, po czym zakończymy symulację.

Teraz w programie *ModelSim* możemy wywołać polecenie:

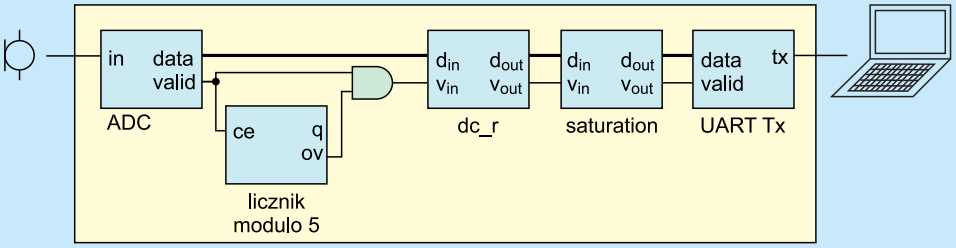
```
do saturation_sim.do
```

Pojawią się przebiegi podobne do tych z **rysunku 16**. W pierwszych trzech wierszach znajdują się sygnały: zegarowy, resetu oraz `valid`. Następnie znajdziemy dane wejściowe przedstawione w formie wykresu. Pod nim są wewnętrzne sygnały modułu. Linia `sign` (znak) zmienia się z `1` na `0`, gdy dane wejściowe staną się nieujemne. Dwa kolejne wiersze to wektor `truncated` oraz sygnał `overflow`. Widzimy, że dla bardzo małych i bardzo dużych wejść bity wektora `truncated` mają różne wartości, co powoduje pojawienie się jedynki logicznej w sygnale `overflow`. Pod nim znajdziemy dane wyjściowe. Dla mocno ujemnych



Rysunek 16. Symulacja modułu saturation

wymuszeń otrzymujemy na wyjściu stałą wartość ujemną. Następnie, gdy pojawi się sygnał, który mieści się w obsługiwanym przedziale, zobaczymy funkcję liniową, która narasta, aż do osiągnięcia maksymalnej wartości dodatniej, gdzie znowu ulega nasyceniu.



Rysunek 17. Nagrywanie dźwięku z usunięciem składowej stałej

Nagrywamy po raz drugi

Mamy już przygotowane i przetestowane nowe bloki, więc możemy je teraz połączyć razem. Schemat naszego drugiego projektu został zaprezentowany na **rysunku 17**. Jest on bardzo podobny do pierwszej wersji z rysunku 4. Po prostu pojawiły się dwa dodatkowe moduły, przez które przepływają nasze dane.

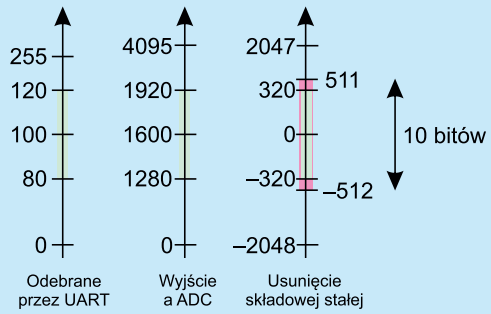
Musimy jeszcze ustalić, ile bitów danych z wejścia ADC potrzebujemy. Jak pamiętamy z wykresu przedstawionego na rysunku 8, gdy przekazywaliśmy osiem najstarszych bitów pomiaru, użyteczny sygnał mieścił się, mniej więcej (z pewnym marginesem bezpieczeństwa), w zakresie 80…120. Dla przypomnienia zostało to pokazane na pierwszym słupku z **rysunku 18**.

Ponieważ przetwornik zwraca nam 12 bitów danych, oznacza to, że nasz sygnał znajduje się pomiędzy wartościami 1280…1920. Pokazuje to środkowy słupek. Potrafimy już usunąć składową stałą, która wynosi około 1600. Po odjęciu jej okaże się, że użyteczne dane mieszczą się w zakresie około –320…320. Oznacza to, że można bez problemu zmieścić go w 10-bitowej liczbie ze znakiem. Ponieważ, tak jak poprzednio, chcemy wysyłać po 8 bitów danych na jeden pomiar, możemy od razu na blok `dc_r` podać jedynie 10 najstarszych bitów, co będzie odpowiadało podzieleniu przez cztery.

Przejdźmy teraz do kodu, którego fragmenty znajdziemy na **listingu 9**. Najpierw definiujemy pomocniczo dwie stałe `N_ADC` i `N`, które będą określać szerokości interfejsów. Zostały one zdefiniowane w liniach 26…28. Do pierwszej magistrali `bus_adc` podłączamy wyjście z przetwornika ADC. Niżej znajdują się instancje modułów składowych.

Listing 9. Połączenie przygotowanych modułów (11_dc_r\rec.sv)

```
18 parameter N_ADC = 10;
19 parameter N = 8;

26 StreamBus #(.N(N_ADC)) bus_adc(clk, rst);
27 StreamBus #(.N(N_ADC)) bus_dc(clk, rst);
28 StreamBus #(.N(N)) bus_uart(clk, rst);

55 assign bus_adc.data = adc_data_out[11-:N_ADC];

65 dc_r #(.N(N_ADC)) dc_remove (
66  .in(bus_adc),
67  .out(bus_dc));
68
69 saturation #(.N_IN(N_ADC), .N_OUT(N)) sat (
70  .in(bus_dc),
71  .out(bus_uart));
72
73 uart_tx #(
74  .F(F),
75  .BAUD(115200)
76 ) uart (
77  .bus(bus_uart),
78  .tx(tx));
```



Rysunek 18. Ile bitów danych zbieramy

Tym razem zmodyfikujemy sposób symulacji przetwornika ADC. Skorzystamy z możliwości dodania własnego pliku z listą kolejnych wartości napięcia. Najpierw jednak przygotujemy nowy sygnał testowy, który przyda nam się także w kolejnych projektach. Zamiast mowy wygenerujemy sygnał *chirp*. Po polsku czasami nazywa się go świergot albo ćwierkanie. Jest to sygnał sinusoidalny, ale z liniową zmianą częstości pomiędzy początkową wartością ω_0, a końcową ω_1. Oznacza to, że w momencie t chwilowa częstość jest równa:

$$\omega = \left(1 - \frac{t}{T}\right)\omega_0 + \frac{t}{T}\omega_1$$

Wyjściowy sygnał otrzymujemy po obliczeniu funkcji sinus dla aktualnej częstości:

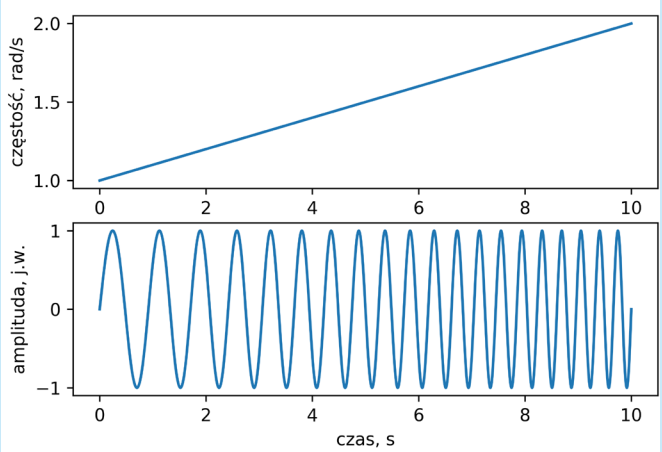
$$y = A\sin(\omega t)$$

Na **rysunku 19** pokazany jest przykładowy przebieg częstości, zmieniającej się od 1 do 2 radianów na sekundę.

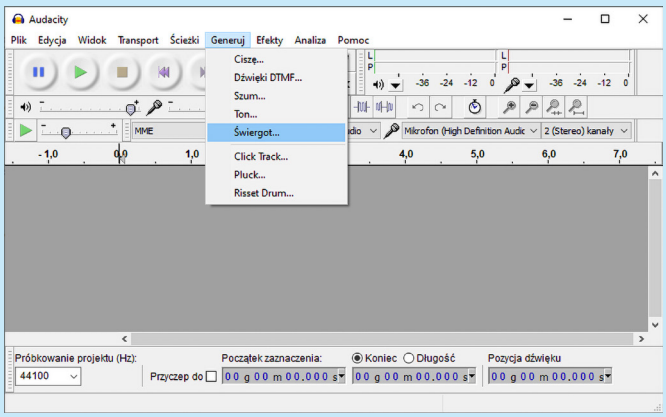
Do wygenerowania pliku dźwiękowego został wykorzystany program *Audacity* [5]. Jego główne okno pokazuje **rysunek 20**. Na pasku zadań wybieramy opcję *Generuj*. Z rozwiniętego menu wybieramy opcję *Świergot*.

Pojawi się nowe okno (**rysunek 21**), gdzie możemy skonfigurować parametry naszego sygnału. Jako częstotliwość początkową wybieramy 0, a końcową 5000 Hz. Amplitudę (zarówno początkową, jak i końcową) zmieniamy na 1. Klikamy *OK*. Teraz naciskając zielony przycisk *Play*, możemy odegrać sygnał. Uwaga, jest to dość głośny i nieprzyjemny odgłos. W repozytorium znajduje się także plik mp3 *11_dc_r\parse\chirp_0_5.mp3*, zawierający stworzony przebieg.

Do zarejestrowania danych ponownie wykorzystujemy projekt *10_record/10_rec.qpf*. Tym razem zamiast zapisać wynik z powrotem do pliku dźwiękowego, stworzymy plik tekstowy, w którym w każdej linii znajduje się numer próbki, a następnie liczba



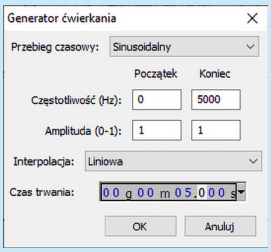
Rysunek 19. Sygnał chirp (świergot, ćwierkanie)



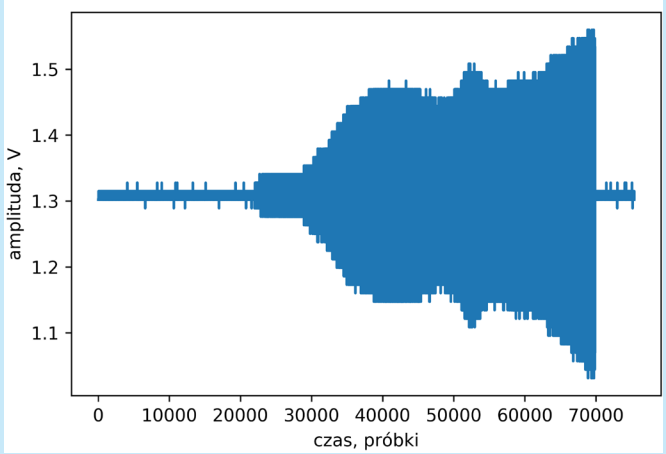
Rysunek 20. Okno programu *Audacity*

zmiennoprzecinkowa, odpowiadająca napięciu. Przykładową zawartość pokazuje **listing 10**.

Do wygenerowania pliku tekstowego użyjemy kolejnego skryptu. Jego fragment znajdziemy na **listingu 11**. Najpierw odczytane z portu szeregowego wartości zmieniamy na wolty, mnożąc przez wartość napięcia odniesienia 3,3 V, a następnie dzieląc przez odpowiadającą jej wartość 256. Dalej w pętli tworzymy nasz plik do symulacji. Ponieważ przesyłamy tylko co piątą próbkę, musimy każdy pomiar zapisać w pliku pięciokrotnie. Aby symulacja nie trwała zbyt długo, wybieramy tylko fragment zebranych



Rysunek 21. Okno generowania sygnału chirp



Rysunek 22. Zmiana napięcia w funkcji czasu

Listing 10. Fragment danych do symulacji (11_dc_r\parse\adc_sample.txt)

```
01 39877 1.2761718034744263
02 39878 1.2761718034744263
03 39879 1.2761718034744263
04 39880 1.2890625
05 39881 1.2890625
06 39882 1.2890625
07 39883 1.2890625
08 39884 1.2890625
09 39885 1.3019530773162842
10 39886 1.3019530773162842
```

Listing 11. Skrypt w języku Python przygotowujący dane do symulacji (11_dc_r\parse\adc_sample.ipynb)

```
01 Vmax = 3.3
02 ADC_max = 256
03 v = np.float32(xn)*Vmax/ADC_max
04 plt.plot(v)
05
06 file="adc_sample.txt"
07 vp = v[20000:30000]
08 with open(file, 'w') as f:
09  for i in range(len(vp)):
10  for j in range(5):
11      f.write("{} {}\n".format(5*i+j, vp[i]))
```

danych. Uzyskane wartości napięcia możemy także wyrysować w formie wykresu, który został pokazany na **rysunku 22**.

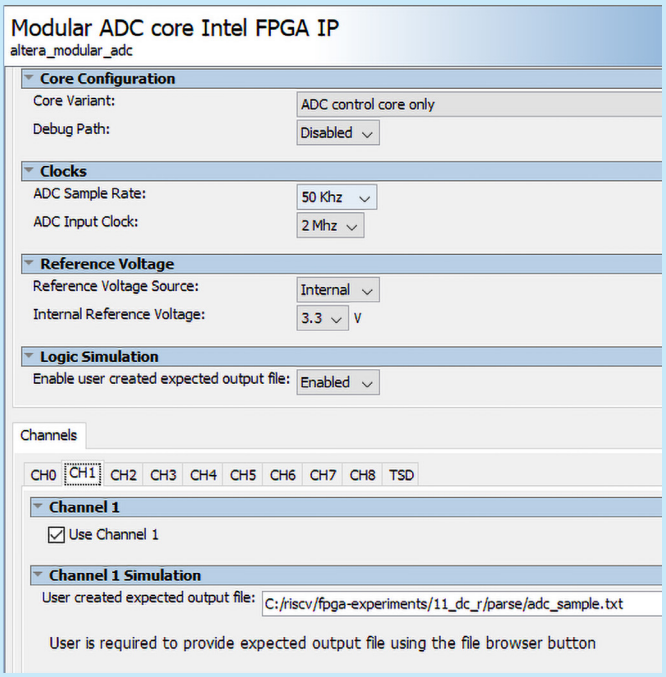
Teraz, generując blok ADC dla nowego projektu, możemy w zakładce *Logic Simulation* (*symulacja logiki*) wybrać opcję *Enable (włączone)*, a następnie, przy konfiguracji kanału 1 (CH1), dodać ścieżkę do wygenerowanego przez nas pliku. Kiedy przyjrzymy się wygenerowanym modułom, zobaczymy, że pojawiła się tam odpowiednia konfiguracja, co zostało pokazane na **listingu 12**.

Pozostaje jeszcze przygotowanie testbenchu. Jest on bardzo prosty, ponieważ tym razem generujemy jedynie sygnał zegarowy i reset. Dane będą odczytywane z pliku i dostarczane przez blok ADC. Tym razem, pomiędzy kolejnymi próbkami, będziemy mieli długą przerwę ze stanem nieustalonym. Dlatego, aby wyrysować czytelne wykresy, stworzymy dodatkowe sygnały, gdzie będziemy zatrzaskiwać jedynie poprawne dane. Odpowiadający za to kod przedstawiony jest na **listingu 13**.

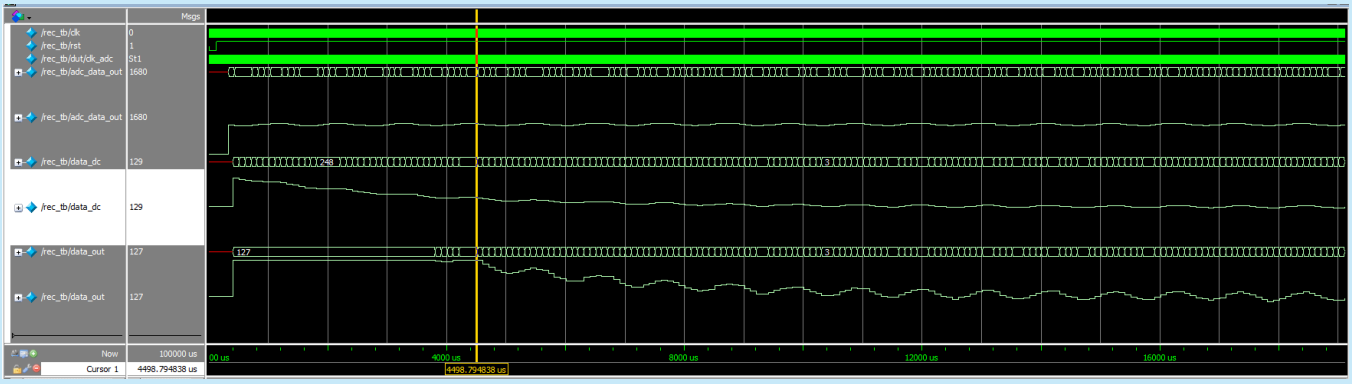
Symulację uruchamiamy komendą:

```
do ./rec_sim.do
```

Ponieważ symulujemy czas kilku sekund, jej wykonanie może zająć nawet kilkanaście minut. Fragment uzyskanego wyniku pokazuje **rysunek 24**.



Rysunek 23. Konfiguracja danych symulacyjnych dla bloku ADC



Rysunek 24. Symulacja całego modułu nagrywania

Listing 12. Konfiguracja pliku z danymi dla ADC (11_dc_r\adc\simulation\submodules\adc_modular_adc_0.v)

```
26 altera_modular_adc_control #(
39  .simfilename_ch0 (""),
40  .simfilename_ch1 ("C:/riscv/fpga-experiments/11_dc_r/parse/adc_sample.txt"),
41  .simfilename_ch2 (""),
```

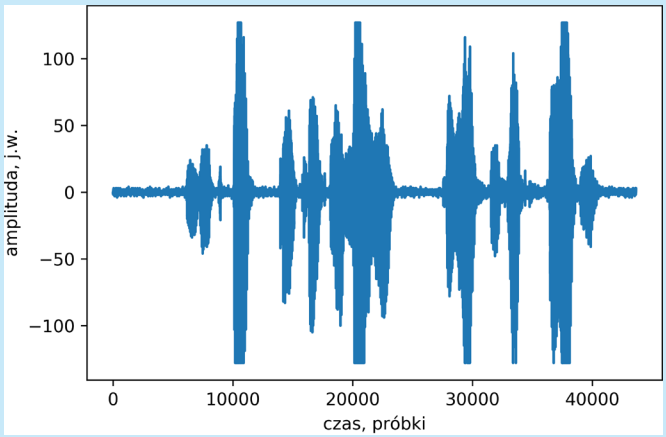
Listing 13. Zapisywanie danych w testbenchu (11_dc_r\rec_tb.sv)

```
33 always_ff @(posedge clk) begin
34  if (dut.adc_valid_out)
35      adc_data_out <= dut.adc_data_out;
36  if (dut.bus_dc.valid)
37      data_dc <= dut.bus_dc.data;
38  if (dut.bus_uart.valid)
39      data_out <= dut.bus_uart.data;
40 end
```

Na końcu otwórzmy projekt *11_dc_r/11_rec.qpf* w środowisku Quartus i rozpocznijmy jego budowę. Kiedy się zakończy, możemy wgrać bitstream i zarejestrować kolejny fragment nagrania. Do jego konwersji na format dźwiękowy możemy wykorzystać skrypt *11_dc_r/parse/Parse.ipynb*. Od pokazanego na listingu 2 różni się traktowaniem odebranych danych jako liczby ze znakiem:

```
xn = np.int8(x)
```

Wykres z zarejestrowanymi danymi przedstawia **rysunek 25**. Widzimy, że gdy wartości przechodzą poza dopuszczalne wartości 127, albo –128, następuje nasycenie.



Rysunek 25. Mowa zarejestrowana przez układ z usuwaniem składowej stałej

Podsumowanie

W tym odcinku rozpoczęliśmy eksperymenty z przetwarzaniem dźwięku w układzie FPGA. W następnej części zaimplementujemy transformatę *Fouriera* i będziemy prezentować aktualne spektrum odbieranego sygnału na diodach LED.

Rafał Kozik
rafkozik@gmail.com

[1] Intel MAX 10 Analog to Digital Converter User Guide, 2020-08-15, ***https://intel.ly/3iIQ6Q2***
[2] anaconda.com, 2020-08-15, ***https://bit.ly/3iN89oi***
[3] Film pokazujący, jak skonwertować nagranie ***https://bit.ly/3iMh5u4***
[4] Lyons r.G., *Wprowadzenie do cyfrowego przetwarzania sygnałów*, Wydawnictwo Komunikacji i Łączności, Warszawa 2010
[5] Audacity, 2020-08-15, ***https://bit.ly/3h5vjWz***

Miesięcznik **„Elektronika Praktyczna”** (12 numerów w roku) jest wydawany przez AVT-Korporacja Sp. z o.o. we współpracy z wieloma redakcjami zagranicznymi.

Wydawnictwo:
AVT-Korporacja Sp. z o.o.
03-197 Warszawa, ul. Leszczynowa 11
tel. 22 257 84 99, faks 22 257 84 00

Wydawca:
Wiesław Marciniak

Adres redakcji:
03-197 Warszawa, ul. Leszczynowa 11
tel. 22 257 84 60
faks 22 257 84 00
e-mail: redakcja@ep.com.pl
www.ep.com.pl

Redaktor Naczelny:
Damian Sosnowski

Redaktor Programowy, Przewodniczący Rady Programowej:
Piotr Zbysiński

Zastępca Redaktora Naczelnego, Menedżer Magazynu:
Marcin Karbowniczek

Szef Pracowni Konstrukcyjnej:
Grzegorz Becker

Redakcja strony internetowej *www.ep.com.pl*
MAD Sp. z o.o.

Zespół marketingu i reklamy:
Katarzyna Gugała, tel. 22 257 84 64
Bożena Krzykawska, tel. 22 257 84 42
Grzegorz Krzykawski, tel. 22 257 84 60

Sekretarz Redakcji:
Grzegorz Krzykawski, tel. 22 257 84 60

DTP i okładka:
MAD Sp. z o.o.

Stali Współpracownicy:
Nikodem Czechowski, Jakub Tyburski, Lucjan Bryndza, Jarosław Doliński, Andrzej Gawryluk, Krzysztof Górski, Tomasz Jabłoński, Michał Kurzela, Szymon Panecki, Sławomir Skrzyński, Ryszard Szymaniak, Adam Tatuś, Robert Wołgajew

Uwaga!
Kontakt z wymienionymi osobami jest możliwy via e-mail, według schematu: imię.nazwisko@ep.com.pl

Prenumerata w Wydawnictwie AVT
www.avt.pl/prenumerata
lub tel. 22 257 84 22
e-mail: prenumerata@avt.pl
www.sklep.avt.pl, tel. 22 257 84 66

Prenumerata w RUCH S.A.
www.prenumerata.ruch.com.pl
lub tel. 801 800 803, 22 717 59 59
e-mail: prenumerata@ruch.com.pl

Wydawnictwo AVT-Korporacja Sp. z o.o. należy do **Izby Wydawców Prasy**