Jak używać układów SoC Xilinx Zynq-7000 z Linuksem – proste przykłady (5)

Obsługa audio na płytce Zedboard

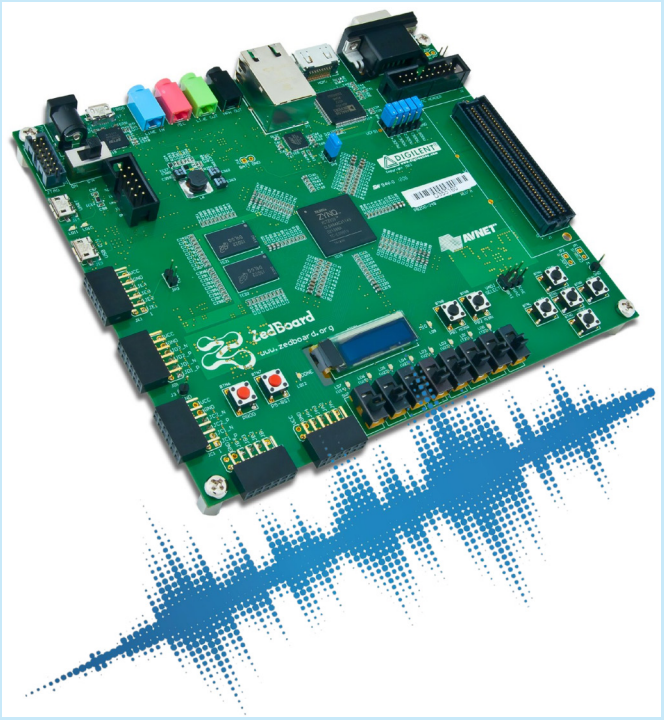
Każdy, kto interesuje się przetwarzaniem sygnałów audio zainteresuje się na pewno możliwościami płytki Zedboard w tym zakresie. Samodzielna implementacja obsługi peryferiów odpowiedzialnych za analogowe wejścia i wyjścia audio nie jest jednak łatwa. Z pomocą znów przychodzi projekt referencyjny firmy Analog Devices. W tym artykule pokażę, jak obsłużyć te peryferia z poziomu Linuksa.

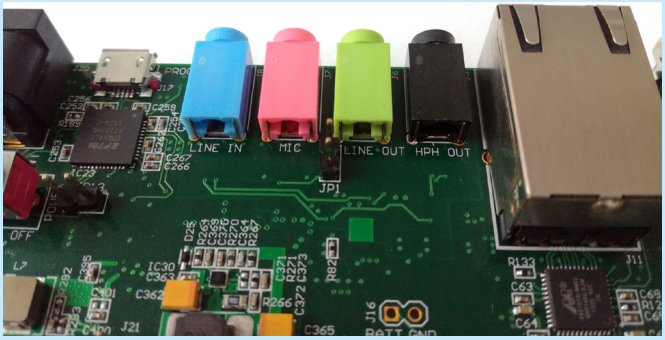
Płytka Zedboard wyposażona jest w dwa układy scalone obsługujące sygnały audio. Pierwszy z nich, to układ odpowiedzialny za cyfrowy interfejs HDMI, drugi zaś za analogowe gniazda jack. W projektach omawianych w tym artykule wykorzystane zostaną wejścia i wyjścia analogowe.

Podobnie jak w wielu innych współczesnych urządzeniach, za obsługę analogowych sygnałów audio odpowiada gotowy układ scalony. Na płytce Zedboard jest to układ ADAU1761. Jest to kodek audio, co w elektronice oznacza układ zajmujący się zakodowaniem sygnałów analogowych w cyfrowe i odwrotnie – dekodowanie cyfrowych w analogowe. Jest więc wyposażony zarówno w przetworniki analogowo-cyfrowe (A/C) i cyfrowo-analogowe (C/A).

Na **rysunku 1** widoczne są gniazda jack podłączone do układu ADAU1761, zaś **rysunek 2** przedstawia schemat podsystemu audio na płytce Zedboard. Niebieskie gniazdo to wejście liniowe stereo, różowe – wejście mikrofonowe, zielone to wyjście liniowe stereo, zaś czarne – wyjście słuchawkowe. Jak pokazuje schemat, każde gniazdo podłączone jest do osobnego wejścia lub wyjścia kodeka. Oprócz układów dopasowania i filtrów nie ma po drodze żadnych dodatkowych elementów. Do lewego kanału złącza mikrofonowego podłączone jest stałe napięcie, niezbędne do spolaryzowania komputerowych mikrofonów elektretowych i pojemnościowych, typowych dla tego typu zastosowań. Opcjonalnie, po zamontowaniu zworki w gnieździe JP1, można dostarczyć napięcie polaryzacji również do prawego kanału.

Rysunek 3 pokazuje schemat blokowy układu ADAU1761. Jest to układ przeznaczony dla odtwarzaczy audio, kamer, telefonów itp., więc ma dużo opcji pozwalających odciążyć procesor i oszczędzić energię. Wśród nich jest przede wszystkim elastyczny w konfiguracji mikser analogowy sterowany cyfrowo i wbudowany niewielki procesor DSP. Nam jednak te opcje nie są potrzebne, a układ będzie skonfigurowany w jak najprostszy sposób, aby jak najwięcej przetwarzania sygnałów odbywało się w układzie Zynq. Jak widać na schemacie, mimo wielu kanałów wejściowych i wyjściowych układ ma tylko dwa przetworniki ADC i dwa DAC. Aby używać wszystkich wejść i wyjść, trzeba odpowiednio skonfigurować mikser wejściowy i wejściowy, aby dodał do siebie odpowiednie sygnały.

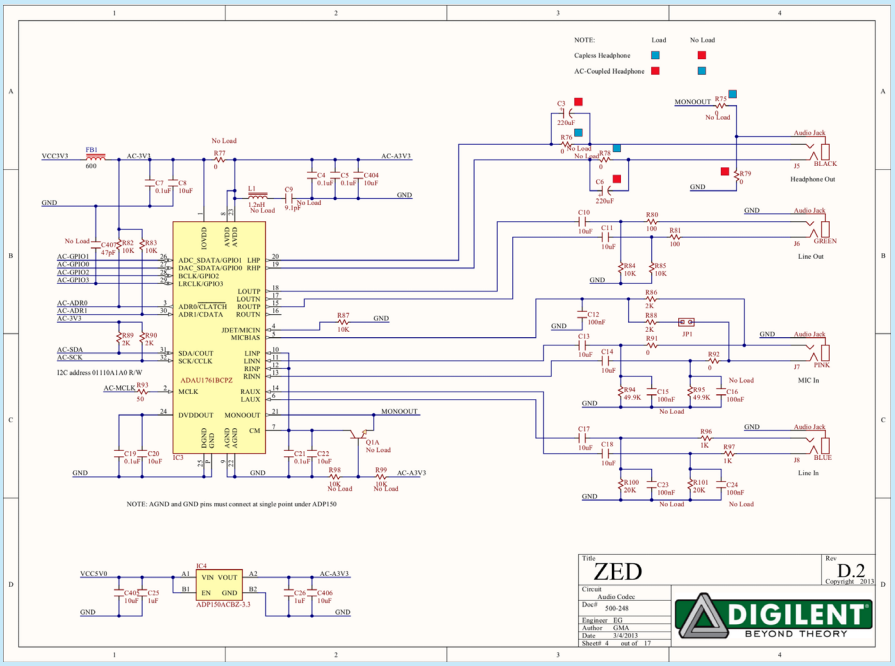




Rysunek 1. Gniazda jack dołączone do układu ADAU1761

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie **www.media.avt.pl**

Do konfiguracji układu służy interfejs I²C. Na płytce Zedboard podłączony jest on do części PL Zynq, podobnie jak interfejs cyfrowy do przesyłania danych. Do obsługi układu potrzebny jest odpowiedni IP core (Intellectual Property Core, blok funkcjonalny w układach FPGA), który jest w stanie obsłużyć interfejs I²C i drugi do obsługi przesyłania danych audio. Próbki audio mogą być przesyłane za pomocą różnych protokołów, opisanych w dokumentacji. Jednym z nich jest I²S. Blok w FPGA obsługujący I²S jest dostępny w znanym już z poprzednich artykułów projekcie referencyjnym udostępnionym przez firmę Analog Devices. Do odpowiednich pinów podłączony jest też blok I²C, więc jeśli ktoś już uruchamiał poprzednie projekty, to nie musi nic zmieniać, aby obsłużyć układ ADAU1761.

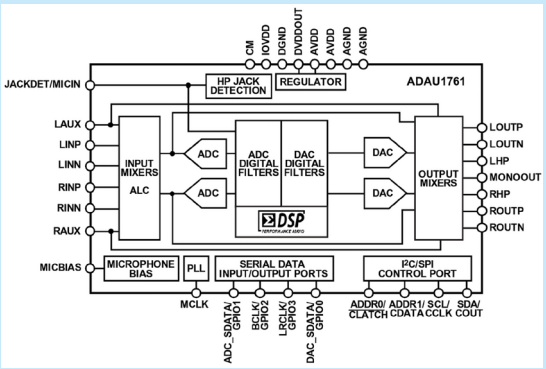


Rysunek 2. Schemat podsystemu audio na płytce Zedboard

Software

Blok I²S podłączony jest bezpośrednio do części PS Zynq poprzez interfejs AXI i dodatkowe sygnały sterujące do DMA. Próbki są więc przesyłane bezpośrednio do procesora, który musi je odebrać, przetworzyć i wysłać z powrotem, jeśli jest taka potrzeba. Samo DMA obsługiwane jest na poziomie systemu operacyjnego, co zapewnia wysoką wydajność.

Do odtwarzania i nagrywania dźwięku pod Linuksem potrzeby jest sterownik, który dostarczy API w przestrzeni użytkownika, pozwalające na dostęp do próbek. Jego rolą jest też połączenie strumieni z różnych programów, obsługę rożnych sposobów kodowania itd. Obecnie najczęściej taką rolę pełni Alsa – projekt, który składa się ze sterownika, jak i biblioteki w przestrzeni użytkownika zwanej alsa-lib. Sama Alsa jak i sterownik obsługujący IP Core I²S są już zawarte w domyślnej konfiguracji jądra Linuksa, którego źródła dostarcza ADV, więc nie będzie potrzebna modyfikacja jądra. Trzeba będzie jedynie dodać bibliotekę, którą dodamy razem z dodatkowymi narzędziami.



Rysunek 3. Schemat blokowy układu ADAU1761

Projekty

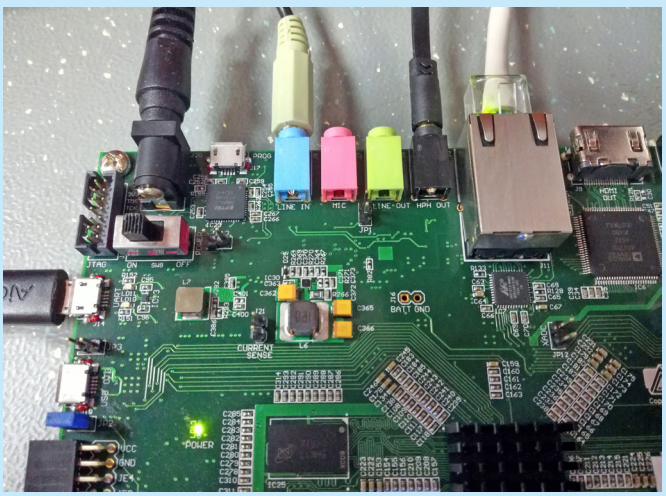
Uruchomione zostaną trzy programy. Pierwszy z nich implementuje prosty bypass, czyli pobiera próbki wejściowe (sumę sygnałów z wejścia liniowego i mikrofonowego) i wysyła na wyjście (zarówno liniowe, jak i słuchawkowe). Opcjonalnie można włączyć dolnopasmowy filtr półkowy o stałych współczynnikach. Jest to przerobiona wersja aplikacji demonstracyjnej, dostępnej w dokumentacji projektu Alsa. Drugi projekt dodaje parametryzowany, dolnopasmowy filtr półkowy, w którym można zmieniać częstotliwość odcięcia i wzmocnienie (lub tłumienie, jeśli jest ujemne). Implementuje też serwer HTTP, w znany już z poprzedniego projektu sposób. Parametry można więc regulować poprzez GUI w przeglądarce. Obliczenia wykonuje procesor ARM. Trzeci projekt różni się od drugiego tym, że obliczenia wykonuje blok w FPGA, więc niezbędna będzie modyfikacja projektu referencyjnego. Drugi i trzeci projekt bazuje na przykładowej aplikacji dostępnej na stronie projektu Alsa.

Opis teorii obliczeń dla filtrów cyfrowych wykracza poza ramy tego artykułu, zainteresowanych odsyłam do książek na temat cyfrowego przetwarzania sygnałów i do publikacji, na podstawie której zaimplementowałem ten filtr ***http://bit.ly/2uByiSD***.

Przygotowanie

W pierwszej kolejności uruchomimy dwa pierwsze programy. Do pierwszej i drugiej aplikacji są potrzebne zmiany w projekcie Vivado. Nie trzeba też nic zmieniać w FSBL, Uboot, ani Devicetree – mają wszystko, co jest potrzebne. Domyślna konfiguracja Linuksa ma dodany zarówno sterownik Alsa, jak i współpracujący z nim sterownik obsługujący blok podłączony do ADAU1761, więc nie trzeba również modyfikować jądra Linuksa. Sposób budowania powyższych komponentów opisałem w jednej z poprzednich części kursu.

Jedne, co wymaga modyfikacji, to rootfs. Trzeba dodać do niego bibliotekę Alsa, najlepiej razem z podstawowymi narzędziami. Tym



Rysunek 4. Połączenia aplikacji

Rysunek 5. Okno programu Alsamixer

razem dodamy też dodatkowy serwer HTTP, służący tylko do statycznego udostępniania plików frontendu (HTML, CSS i Javascript). Zwykle backend (np. napisany w PHP) również jest uruchamiany przez serwer HTTP, a nie jest implementowany w aplikacji jak w naszym przypadku. Jeśli niezbędne są jakieś działania w tle, pisana jest w tym celu druga aplikacja, która w jakiś sposób komunikuje się z backendem. W tych projektach występuje uproszczenie, polegające na tym, że jest tylko jedna aplikacja implementująca serwer i jednocześnie działająca w tle. Dodatkowo dodamy jeszcze serwer ssh, który ułatwi konfigurację sterownika Alsa. Przejdźmy więc do modyfikacji rootfs.

W pliku poky/zedboard/conf/local.conf szukamy zmiennej IMAGE_INSTALL_append i dodajemy do niej pakiety alsa-utils openssh lighttpd, pamiętając o spacji przed pierwszą nazwą pakietu

```
IMAGE_INSTALL_append = " libgcc libstdc++ libmicrohttpd alsa-utils openssh lighttpd"
```

W konsoli przechodzimy do folderu poky i konfigurujemy środowisko komendą

```
source oe-init-build-env zedboard
```

I budujemy obraz komendą

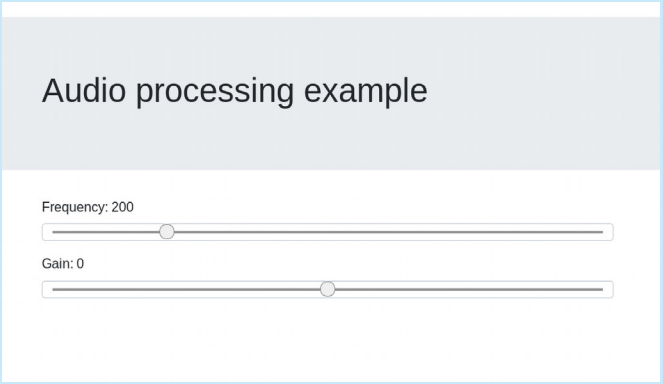
```
bitbake core-image-minimal
```

Jeśli obraz był już wcześniej zbudowany, to samo dodanie pakietów powinno potrwać kilka minut. Budowanie obrazu od początku potrwa jednak do kilku godzin.

Po zbudowaniu, dodajemy obraz zgodnie z preferowaną metodą uruchamiania Linuksa, opisaną w jednym z poprzednich artykułów. Po uruchomieniu Linuksa jesteśmy gotowi do wypróbowania pierwszych dwóch aplikacji.

Uruchomienie aplikacji

Kod źródłowy aplikacji można znaleźć na stronie ep.com.pl. Po uruchomieniu środowiska XSDK, w pierwszej kolejności trzeba utworzyć projekt, w taki sposób jak przy poprzednich przykładach. Pierwszy z nich można nazwać np. bypass, a drugi soundhttpserver.



Rysunek 6. Okno interfejsu użytkownika

Po utworzeniu projektów, dodajemy do nich pliki źródłowe i budujemy.

Jeśli wszystko przebiegło pomyślnie, można spróbować uruchomić pierwszą aplikację. Przedtem jednak warto podłączyć do niebieskiego gniazda źródło dźwięku np. komputer lub jakiś odtwarzacz i jakiś odbiornik, np. słuchawki do czarnego gniazda, jak na **rysunku 4**. Po podłączeniu sprzętu i uruchomieniu systemu można uruchomić odtwarzanie dźwięku w urządzeniu podłączonym do niebieskiego gniazda JACK i spróbować uruchomić aplikację. Prawdopodobnie jednak nic nie będzie słychać, gdyż Alsa domyślnie jest niewłaściwie skonfigurowana. Trzeba zmienić ustawienia sterownika, poprzez panel alsamixer. Tu przyda się zainstalowany serwer ssh, dzięki któremu dostępna będzie w konsoli kolorowa grafika i który zapewnia lepszą wydajność odświeżania interfejsu niż konsola UART. Aby połączyć się z płytką przez ssh, trzeba wpisać w konsoli na komputerze komendę:

```
ssh root@<ip płytki>
```

np.

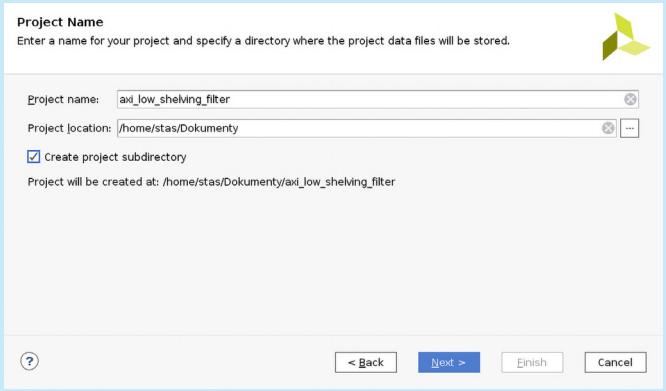
```
ssh root@192.168.0.123
```

Warto przy okazji dodać, że po zainstalowaniu serwera ssh można się łatwo zalogować na płytkę bez hasła na konto root'a, co może być potencjalnie niebezpieczne, jeśli płytka jest w publicznej sieci.

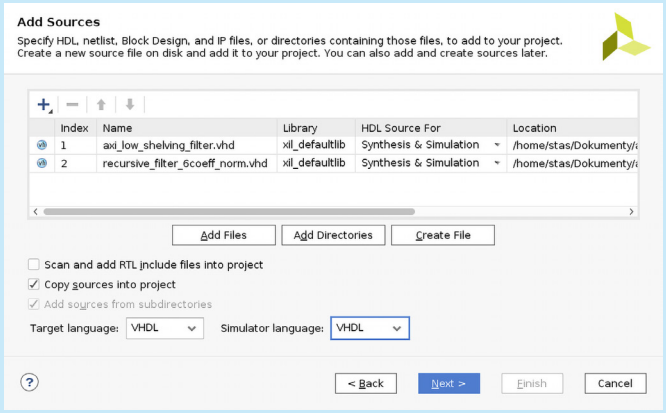
Po zalogowaniu uruchamiamy alsamixer (komenda alsamixer). Włączamy opcje dla ADAU (F6, dwa razy strzałka w dół, enter). Zostanie wyświetlone okno, jak na **rysunku 5**. Trzeba zmodyfikować pięć ustawień:

- capture mux: decimator
- DAC playback Mux: AIFIN
- Left Playback Mixer Left DAC: Wyłączyć mute (klawisz m)
- Right Playback Mixer Right DAC: Wyłączyć mute (klawisz m)
- aux: minimum 50, można więcej.

Po tych modyfikacjach program powinien zadziałać – na słuchawkach powinien być słyszalny sygnał, który podawany jest na wejście.



Rysunek 7. Okno parametrów projektu w programie Vivado



Rysunek 8. Dodanie dwóch plików VHDL

Warto teraz zapisać działającą konfigurację do pliku, aby móc ją później przywołać. Służy do tego komenda:

```
alsactl store -f <file_path>
```

Aby później przywołać tę konfigurację, należy wykonać komendę:

```
alsactl restore -f <file_path>
```

Jeśli ktoś stosuje sposób bootowania, w którym pliki są zachowywane pomiędzy bootowaniami, to można zapisać konfigurację na stałe. Jeśli nie, to warto skopiować plik na swój komputer. Wtedy za każdym razem trzeba przesyłać ten plik i od nowa konfigurować Alsę.

Jeśli aplikacja nie działa, można spróbować wykonać testy. Pierwszy z nich sprawdzi, czy działają gniazda wyjściowe:

```
speaker-test -c 2 -F S32_LE --device hw:ADAU1761
```

Drugi nagra sygnał o czasie trwania 3 sekund do pliku test.wav:

```
arecord -f S32 -r 48000 -c 2 --device hw:ADAU1761 -t wav -d 3 test.wav
```

Można go później odtworzyć komendą:

```
aplay --device hw:ADAU1761 test.wav
```

lub, jeśli nic nie słychać, można wysłać go na swój komputer i sprawdzić, czy cokolwiek się nagrało. Jeśli tak, to problem jest z odtwarzaniem i trzeba jeszcze raz sprawdzić konfigurację alsamixer.

Drugą aplikację uruchamiamy w ten sam sposób, ale podobnie jak przy aplikacji z ledami trzeba w argumentach wywołania podać port dla serwera http. Dodatkowo trzeba skopiować do odpowiedniego folderu pliki interfejsu webowego (można je znaleźć w folderze www w archirum ze źródłami – link wyżej). Zgodnie z domyślną konfiguracją lighttpd jest to folder /www/pages/. Jeśli rootfs uruchomionego systemu jest na serwerze NFS, wystarczy po prostu skopiować tam pliki. W przeciwnym razie trzeba wysłać je komendą, wykonaną z folderu, w którym jest folder ze wszystkimi plikami źródłowymi:

```
scp www/* root@<ip płytki>:/www/pages/
```

np.

```
scp www/* root@192.168.0.123:/www/pages/
```

Po uruchomieniu drugiej aplikacji znów powinien być słyszalny dźwięk. Aby zobaczyć interfejs, należy otworzyć przeglądarkę i w polu adresu wpisać IP płytki. Tym razem, w przeciwieństwie do przykładu z diodami, numer portu jest niepotrzebny – lighttpd korzysta z portu 80, domyślnego dla serwerów HTTP. Powinien się pokazać interfejs, jak na **rysunku 6**. Górny pasek reguluje częstotliwość odcięcia filtru, a dolny wzmocnienie filtru, które może być też ujemne – w takiej sytuacji sygnał będzie tłumiony. Testy najlepiej przeprowadzać przy użyciu ścieżki dźwiękowej, której widmo ma dużo składowych o niskich częstotliwościach, np. próbek gitary basowej.

W tym przykładzie obliczenia wykonywane są przez procesor ARM. Aby uruchomić kolejny, w którym wykorzystamy do obliczeni FPGA, trzeba jeszcze zmodyfikować projekt Vivado i dodać do niego niezbędny do obliczeń IP core.

Kolejne modyfikacje

W pierwszej kolejności należy utworzyć osobny projekt ze źródłami VHDL filtru, aby spakować go w IP core. Następnie trzeba dodać go do głównego projektu i ponownie wygenerować bitstream.

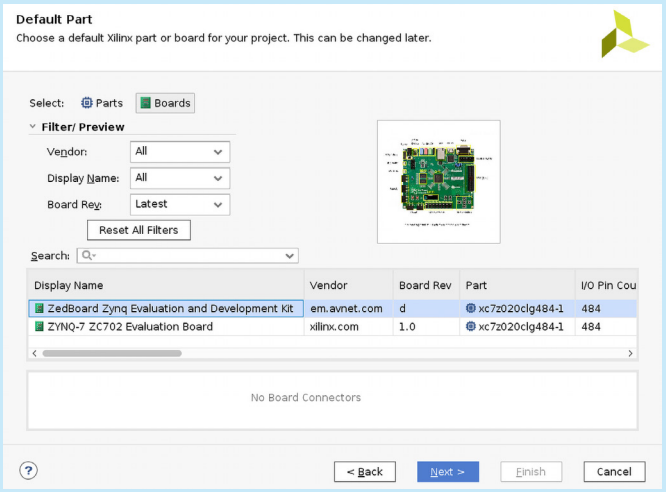
Utwórzmy więc nowy projekt w Vivado, klikając *File* → *New Projekt*... W oknie które się pokaże klikamy *next*. W kolejnym kroku (**rysunek 7**) wpisujemy nazwę projektu i ścieżkę, w której chcemy utworzyć projekt np. tę samą, w której jest folder ze źródłami od ADV. W kolejnym oknie wybieramy *Rtl Project* i odznaczamy *Do not specify sources for this project*. W następnym kroku dodajemy pliki do projektu. Można je znaleźć w folderze vhdl w archiwum projektu. Klikamy *Add Files* i dodajemy dwa pliki VHDL, jak na **rysunku 8**. Zaznaczamy *Copy sources into project*, aby mieć w projekcie kopie, które można potem swobodnie modyfikować. Wybieramy VHDL jako język projektu i symulacji i klikamy *next*. W kolejnym kroku dodajemy plik z ograniczeniami projektu (constraints) w formacie xdc – constraints.xdc. Po kliknięciu *add files* należy go znaleźć w folderze constraints w archiwum ze źródłami. Podobnie jak źródła VHDL, warto skopiować go do projektu. Po kliknięciu *next* pokaże się okno z wyborem układu lub płytki. W naszym przypadku należy na górze kliknąć *boards* i z listy wybrać płytkę Zedboard (**rysunek 9**). Jeśli ktoś posiada starszy model płytki, kupiony kilka lat temu, warto się upewnić, że wybrana jest odpowiednia rewizja. Klikamy *next* i w ostatnim oknie *finish*.

Jeśli ktoś jest zainteresowany w implementacją filtru, można obejrzeć pliki źródłowe. Nie omawiam ich jednak, gdyż wykracza to poza temat artykułu.

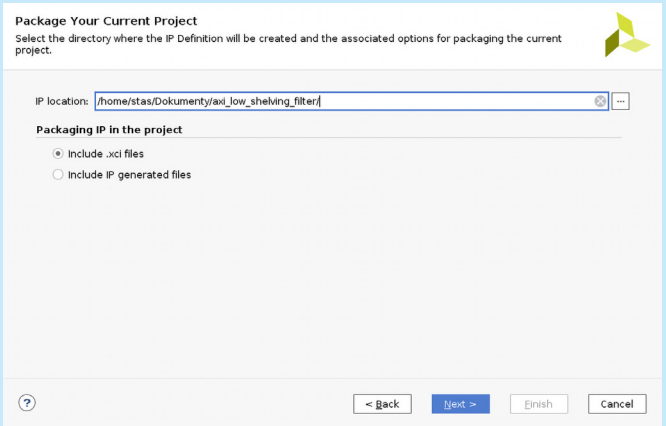
Tego projektu nie będziemy budować, tylko spakujemy go jako IP core. Wybieramy z menu *Tools* → *Create and package new IP*... W pierwszym oknie klikamy *next*, w drugim zaznaczamy *package your current project* i znów klikamy *next*. Potem wybieramy ścieżkę, najlepiej folder, w którym jest projekt i zaznaczamy *include .xci files* (**rysunek 10**). Na koniec jeszcze raz klikamy *next* i *finish*.

W głównym obszarze programu pojawi się panel konfiguracji bloku. Prawie wszystkie karty zaznaczone są na zielono, więc nie trzeba tu wiele zmieniać. Jedyny problem jest w karcie *Addressing and memory*. W tabelce *Address block*, należy wyczyścić komórkę w kolumnie *Range dependency*, a pod kolumną Range wpisać 4096 (jest to minimalna wartość, 256 by w zupełności wystarczyło). Dodatkowo w karcie *Identification* można zmienić nazwę bloku. Po dokonaniu zmian przechodzimy do karty *Review and package* i klikamy *package IP*, aby spakować projekt.

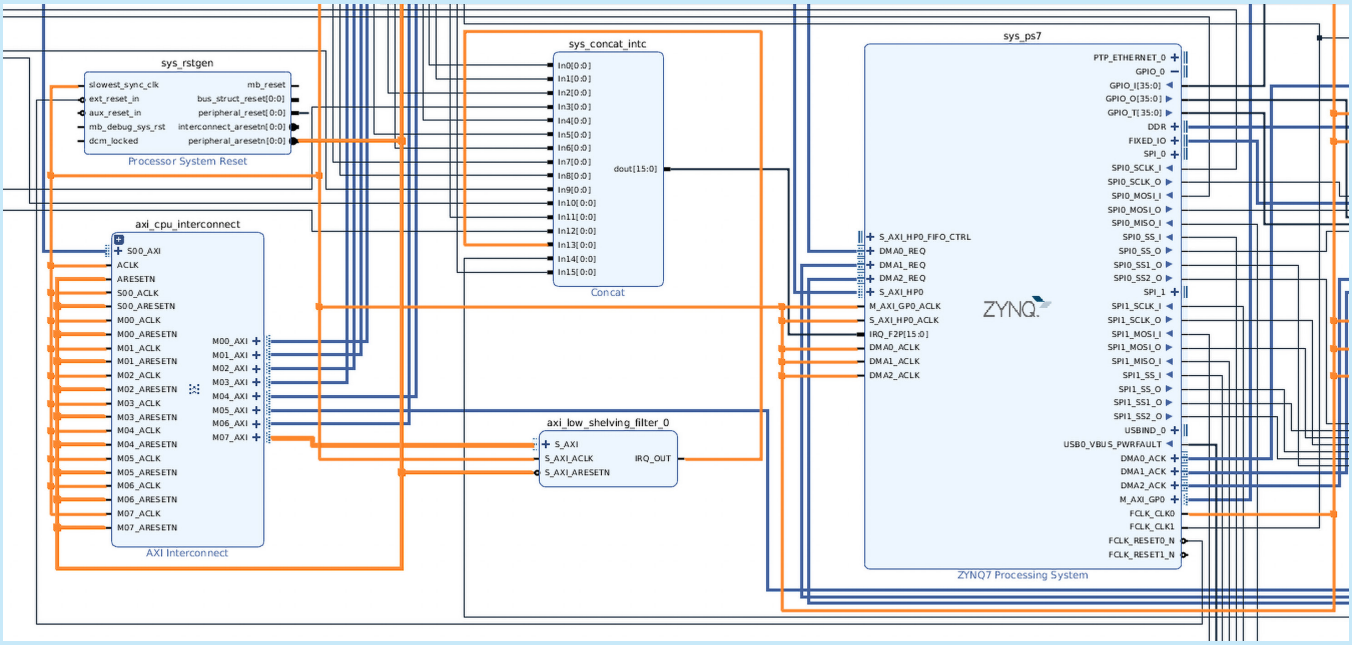
Teraz trzeba dodać spakowany blok do głównego projektu. Otwórzmy go więc i w pierwszym kroku dodajmy nasz blok do biblioteki IP core'ów. Klikamy *Settings* → *IP* → *Repository* a następnie ikonę plusa i dodajemy do biblioteki ścieżkę do projektu z naszym IP



Rysunek 9. Wybranie płytki Zedboards z listy



Rysunek 10. Wybór folderu i pliku



Rysunek 11. Diagram projektu

core'em. Po kliknięciu OK biblioteka odświeży się i już można dodać blok do projektu. Otwórzmy block design i dodajmy IP Core, klikając w obrębie wyświetlonego diagramu prawym przyciskiem myszy i wybierając *Add IP…* Z listy która się pojawi wybieramy nasz blok i przeciągamy go gdzieś na diagram. Na górze pojawi się opcja *Run Connection Automation*, która pozwoli automatycznie utworzyć odpowiednie połączenia, w szczególności te dotyczące szyny danych AXI. Uruchamiamy automatyzację i klikamy OK, pozostawiając domyślne opcje. Jedyny sygnał, który się nie podłączy to sygnał przerwania. Aby móc go podłączyć, usuwamy połączenie pomiędzy portem wejściowym *ps_intr_13*, a blokiem przerwań (NIE usuwamy samego portu, ma pozostać niepodłączony) i podłączamy w to miejsce nasz sygnał przerwania. Na koniec fragment diagramu będzie wyglądał jak na **rysunku 11**. Zaznaczyłem na nim najważniejsze

w tym momencie linie, czyli te podłączone do nowego bloku. Klikamy przycisk *Validate*, a następnie, jeśli nie ma błędów, *Generate Bitstream*. Budowanie projektu znów potrwa kilkanaście minut.

W międzyczasie można przygotować DTS. Trzeba tam dodać informację o nowym bloku. Będzie on obsługiwany przez sterownik UIO, pozwalający na prostą obsługę bloków podłączonych do procesora przez szynę AXI. Zmiany trzeba wprowadzić, podobnie jak w poprzednich artykułach, w pliku zynq-zed-adv7511.dtsi. Na końcu, ale przed ostatnią zamykającą klamrą, a za wpisem o diodach LED, dopisujemy:

```
uio0@43C00000 {
    compatible = "generic-uio";
    reg = < 0x43C00000 0x1000 >;
    interrupts = < 0 57 0 >;
    interrupt-parent = <&intc>;
};
```

Z pewnych względów trzeba też dodać pewien parametr do argumentów wywołania programu:

```
uio_pdrv_genirq.of_id=generic-uio rootwait
```

Można do dodać w DTS w pliku zynq-zed.dtsi, lub w pliku uEnv.txt. Po zmianie przykładowa linijka argumentów wywołania będzie wyglądała tak:

```
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2
rw earlyprintk rootfstype=ext4 uio_pdrv_genirq.of_
id=generic-uio rootwait";
```

Pliki dts kompilujemy komendą (wykonaną z folderu, w którym są pliki DTS:

```
dtc -I dts -O dtb -o devicetree.dtb zynq-zed-adv7511.
dts
```

Plik devicetree.dtb kopiujemy w odpowiednie miejsce, zgodnie z wybraną metodą uruchamiania systemu. To samo należy zrobić z wygenerowanym ponownie plikiem BOOT.bin, lub samym bitstream'em, gdy ten się zbuduje.

Trzeci projekt

W trzecim projekcie obliczenia wykonywane są w FPGA. Projekt zakłada obecność odpowiedniego bloku przetwarzania i właściwego wpisu w DST, dodanego w poprzednim kroku. Sposób tworzenia projektu jest taki jak wcześniej. Po zbudowaniu projektu, uruchamiamy Linuksa, w razie potrzeby konfigurujemy sterownik Alsa i uruchamiamy projekt. Interfejs webowy jest ten sam. Te projekty różnią się tylko fragmentem kodu z pliku audio.c. W związku z ich długością opiszę tylko ten fragment kodu.

Listing 1. Wersja z obliczeniami w ARM

```
while (!audioHandle->threadTerminated) {
    /* Read num_frames frames from the PCM device  */
    /* pointed to by pcm_handle_capture to buffer capdata. */
    /* Returns the number of frames actually read. */
    while ((frames_read = snd_pcm_readi(pcm_handle_capture, capture_data,
                frames)) < 0) {
            snd_pcm_prepare(pcm_handle_capture);
            fprintf(stderr, "<<<<<<<<<< Capture Buffer Overrun >>>>>>>>>>\n");
    }
    //printf("Frames read: %d", frames_read);
    for (j = 0; j < frames_read; j++) {
        if (counter < wait_period) { // Ignoring first 3s
            counter++;
        } else {
            left_sample = raw_to_double(&capture_data[j * 8]);
            right_sample = raw_to_double(&capture_data[j * 8 + 4]);
        }
            clock_gettime(CLOCK_REALTIME, &requestStart);
            //Left sample processing:
            l_in_buff_ptr = (l_in_buff_ptr + 1) % 3;
            left_in_buf[l_in_buff_ptr] = left_sample;
            pthread_mutex_lock(&(audioHandle->lock));
            tmp_value[0] = fData->b0 * left_in_buf[l_in_buff_ptr];
            tmp_value[1] = fData->b1 * left_in_buf[(l_in_buff_ptr + 2) % 3];
            tmp_value[2] = fData->b2 * left_in_buf[(l_in_buff_ptr + 1) % 3];
            tmp_value[3] = -fData->a1 * left_out_buf[l_out_buff_ptr];
            tmp_value[4] = -fData->a2 * left_out_buf[(l_out_buff_ptr + 2) % 3];
            left_sample = 0;
            for (i = 0; i < 5; i++) left_sample += tmp_value[i];
            left_sample /= fData->a0;
            l_out_buff_ptr = (l_out_buff_ptr + 1) % 3;
            left_out_buf[l_out_buff_ptr] = left_sample;
// Right sample processing:
            r_in_buff_ptr = (r_in_buff_ptr + 1) % 3;
            right_in_buf[r_in_buff_ptr] = right_sample;
            right_sample = 0;
            tmp_value[0] = fData->b0 * right_in_buf[r_in_buff_ptr];
            tmp_value[1] = fData->b1 * right_in_buf[(r_in_buff_ptr + 2) % 3];
            tmp_value[2] = fData->b2 * right_in_buf[(r_in_buff_ptr + 1) % 3];
            tmp_value[3] = -fData->a1 * right_out_buf[r_out_buff_ptr];
            tmp_value[4] = -fData->a2 * right_out_buf[(r_out_buff_ptr + 2) % 3];
            for (i = 0; i < 5; i++)
                    right_sample += tmp_value[i];
            right_sample /= fData->a0;
            pthread_mutex_unlock(&(audioHandle->lock));
            r_out_buff_ptr = (r_out_buff_ptr + 1) % 3;
            right_out_buf[r_out_buff_ptr] = right_sample;
            double_to_raw(left_sample, &playback_data[j * 8]);
            double_to_raw(right_sample, &playback_data[j * 8 + 4]);
            clock_gettime(CLOCK_REALTIME, &requestEnd);
            accum += (requestEnd.tv_sec - requestStart.tv_sec)
                            + (requestEnd.tv_nsec - requestStart.tv_nsec) * BILLION;
            if (timer_counter < 10000) {
                    timer_counter++;
            }
            else{
                    timer_counter = 0;
                    accum *= 100;
                    printf( "\rProcessing time: %lfus cutoff freq: %d gain: %d", accum , fData->frequency, fData->dbGain);
                    fflush(stdout);
                    accum = 0;
            }
    }
    while ((pcmreturn = snd_pcm_writei(pcm_handle_playback, playback_data,
                frames_read)) < 0) {
            snd_pcm_prepare(pcm_handle_playback);
            fprintf(stderr, "<<<<<<<<<< Playback Buffer Underrun >>>>>>>>>>\n");
    }
}
```

Listing 2. Wersja z obliczeniami w FPGA

```
while (!audioHandle->threadTerminated) {
    /* Read num_frames frames from the PCM device  */
    /* pointed to by pcm_handle_capture to buffer capdata. */
    /* Returns the number of frames actually read. */
    while ((frames_read = snd_pcm_readi(pcm_handle_capture, capture_data,
                    frames)) < 0) {
            snd_pcm_prepare(pcm_handle_capture);
            fprintf(stderr, "<<<<<<<<<< Capture Buffer Overrun >>>>>>>>>>\n");
    }
//                  printf("Frames read: %d", frames_read);
    clock_gettime(CLOCK_REALTIME, &requestStart);
    pthread_mutex_lock(&(audioHandle->lock));
    if (counter < wait_period) { // Ignoring first 3s
            counter++;
    } else {
            for (j = 0; j < frames_read; j++) {
                    compBlock[j + CH1_IN_FS] = raw_to_int(&capture_data[j * 8]);
                    compBlock[j + CH2_IN_FS] = raw_to_int(&capture_data[j * 8 + 4]);
            }
            compBlock[CONTROL_REGISTER_ADDRESS] = startCountingMessage; //(unsigned int) ((framesToRead << 2) | (1 << 1));
            // Arm interrupt...
            write(fd, &interr, sizeof(interr));
            // ...and wait for it. This will block, until interrupt comes.
            read(fd, &interr, sizeof(interr));
            for (j = 0; j < frames_read; j++) {
                    //printf("%X ", compBlock[j + CH1_OUT_FS]);
                    int_to_raw(compBlock[j + CH1_OUT_FS], &playback_data[j * 8]);
                    //printf("%X ", compBlock[j + 32]);
                    int_to_raw(compBlock[j + CH2_OUT_FS],
                                    &playback_data[j * 8 + 4]);
            }
            //printf("\n");
    }
    pthread_mutex_unlock(&(audioHandle->lock));
    clock_gettime(CLOCK_REALTIME, &requestEnd);
    accum = (requestEnd.tv_sec - requestStart.tv_sec)
                    + (requestEnd.tv_nsec - requestStart.tv_nsec) / BILLION;
    if (timer_counter < 100) {
            timer_counter++;
    } else {
            timer_counter = 0;
            accum *= 1E6;
            printf( "\rProcessing time: %lfus cutoff freq: %d gain: %d", accum , fData->frequency, fData->dbGain);
            fflush(stdout);
            accum = 0;
    }
    while ((pcmreturn = snd_pcm_writei(pcm_handle_playback, playback_data,
                    frames_read)) < 0) {
            snd_pcm_prepare(pcm_handle_playback);
            fprintf(stderr, "<<<<<<<<<< Playback Buffer Underrun >>>>>>>>>>\n");
    }
}
```

Listing 1 przedstawia wersję z obliczeniami w ARM, a **listing 2** z obliczeniami w FPGA. Oba fragmenty zaczynają się od odczytania 32 ramek, z których każda składa się z dwóch próbek (prawej i lewej). Zaraz potem w pierwszym listingu (linijki 27-56) widać odpowiednie obliczenia. W drugim zaś najpierw trzeba skonwertować próbki na odpowiedni format, a następnie wysłać do bloku w FPGA. Odpowiadają za to linijki 19-23 listingu 2. Po wysłaniu próbek program daje znać sterownikowi UIO, że oczekuje na przerwanie i blokuje się, aż do momentu, w którym się ono pojawi (linijki 26 i 28 listingu 2). Na koniec w obu programach próbki zapisywane są do odpowiedniego bufora i przekazywane do sterownika Alsa.

W obu programach obliczenia współczynników filtrów wykonywane są przez procesor ARM. W przypadku drugiego programu jest to typowe podejście polegające na delegowaniu złożonych, ale rzadko wykonywanych obliczeń na procesor, zaś prostszych, ale krytycznych czasowo i wykonywanych w czasie rzeczywistym – na układ FPGA.

Mając takie dwa projekty można się pokusić o sprawdzenie, która metoda jest bardziej efektywna. Oba programy na standardowe wyjście wypisują czas, jaki zajęły im obliczenia dla bieżącego zestawu próbek. Intuicyjnie można się spodziewać, że szybciej wykonają się obliczenia w FPGA. Istotnie, analiza kodu VHDL wykazuje, że wykonują się one w 5 cyklach zegara 100 MHz, czyli w 1,6 µs – krócej niż ok. 2,5 µs potrzebnych na obliczenia w ARM. Jest to niezły wynik, zważywszy na to, że projekt nie jest zoptymalizowany, np. próbki nie są obliczane potokowo. Zmierzony przez program czas to jednak aż ok. 60 µs. Wynika to z faktu, ze potrzebny jest narzut czasowy na przesłanie próbek w obie strony przez magistralę AXI. Ta magistrala ze swojej natury pozwala na szybkie przesyłanie danych, lecz sterownik UIO jest bardzo prosty i wykonuje tę czynność z małą wydajnością. Aby zwiększyć wydajność najlepiej by było wykorzystać funkcję DMA procesora, albo zaprojektować cały system tak, aby procesor w ogóle nie brał udziału w przesyłaniu próbek. Jest to jednak zadanie nietrywialne, wymagające między innymi przerobienia lub zaprojektowania na nowo bloku z FPGA obsługującego układ ADAU1761, jak również opracowania sposobu przesyłania próbek pomiędzy blokami. To zadanie było częścią mojej pracy magisterskiej.

Podsumowanie

Tym razem udało się uruchomić na płytce przetwarzanie sygnałów audio. Wykorzystaliśmy dużo z poprzednich umiejętności, np. tworzenie i budowanie projektu w XSDK. Dodatkowo potrzebny był dedykowany IP core w FPGA i ponowne zbudowanie projektu w Vivado. Wykorzystany też został system przerwań i sterownik UIO, pozwalający na prostą obsługę własnych bloków spod Linuksa. Kolejną częścią projektu mogłoby być na przykład dodanie następnego bloku przetwarzania, albo obliczenie FFT i pokazanie wyniku w postaci graficznej na wyświetlaczu OLED.

Stanisław Aleksiński