



A G H

Akademia Górnictwo-Hutnicza im. Stanisława Staszica w Krakowie

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA AUTOMATYKI I INŻYNIERII BIOMEDYCZNEJ

Praca dyplomowa magisterska

*Wykorzystanie systemu operacyjnego Linux we wbudowanych systemach
wizyjnych zrealizowanych na platformie Zynq.*

*The use of the Linux operating system in embedded vision systems
implemented on the Zynq platform.*

Autor: Wojciech Gumiąż
Kierunek studiów: Automatyka i Robotyka
Opiekun pracy: dr inż. Tomasz Kryjak

Kraków, 2017

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształcła taki utwór, artystyczne wykonanie, fonogram, videogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Streszczenie

Celem pracy było uruchomienie i konfiguracja systemu operacyjnego PetaLinux na platformie Zynq. Zasadniczą uwagę poświęcono funkcjonalnościom, które mogą zostać wykorzystane do realizacji wbudowanych systemów wizyjnych. W ramach pracy przeprowadzono analizę wybranych funkcji systemu operacyjnego i zrealizowano algorytm generacji tła. Opracowano także sposób komunikacji pomiędzy modułami logiki programowalnej i procesorem obliczeniowym z wykorzystaniem interfejsu AXI. Użyto funkcjonalności systemu operacyjnego do analizy i prezentacji wyników działania aplikacji. Metody omówione w pracy pozwalają na budowę zaawansowanych aplikacji wykorzystujących możliwości logiki programowalnej i procesora ARM oraz integrację obu metod w jednym projekcie.

Abstract

The thesis discusses issues related to running and configuring a PetaLinux operating system on the Zynq platform. The research was focused of functionalities which can be incorporated into embedded vision systems. Analysis of selected PetaLinux features was conducted and a background generation algorithm was implemented as a result. Possibilities of communication between programmable logic modules and CPU through AXI interface were analysed. The operating system features were used to perform video analysis and results presentation. Methods discussed in this thesis enable users to create complex applications integrating programmable logic and ARM processor features in one project.

Spis treści

1. Wstęp.....	5
1.1. Cel pracy.....	7
1.2. Zawartość pracy	7
2. Platforma Zynq-7000.....	9
2.1. Systemy operacyjne dostępne dla platformy Zynq	12
3. Przegląd wybranych funkcjonalności platformy Zynq i systemu operacyjnego PetaLinux.....	19
3.1. Obsługa SSH.....	19
3.2. FPU i technologia NEON	20
3.3. Protokół AXI	22
3.4. Obliczenia równoległe	26
3.5. OpenCV	29
3.6. Integracja algorytmów w FPGA i CPU	30
3.7. Przerwania systemowe.....	32
4. System wizyjny zrealizowany na platformie Zynq z systemem operacyjnym PetaLinux.....	35
4.1. Moduł wyznaczania różnicy kolejnych ramek w sekwencji obrazów.....	37
4.2. Moduł generacji tła.....	39
4.3. Integracja z systemem PetaLinux	45
5. Proces konfiguracji modułów logiki programowalnej i systemu operacyjnego PetaLinux.....	51
5.1. Instalacja środowiska PetaLinux	51
5.2. Podstawowa konfiguracja projektu.....	52
5.3. Konfiguracja modułu wykorzystującego interfejs AXI	56
5.4. Konfiguracja modułu AXI VDMA	61
5.5. Obliczenia równoległe	66
5.6. OpenAMP	66

5.7. Biblioteka OpenCV	68
5.8. Wykorzystanie mechanizmu przerwań systemowych	71
5.9. Konfiguracja algorytmu generacji tła.....	77
6. Podsumowanie	87
Dodatki.....	93
A. Spis zawartości płyty CD	93
B. Aplikacja w architekturze NEON	94
C. Konwersja danych pomiędzy VDMA i OpenCV	96

1. Wstęp

Przetwarzanie obrazów i ich sekwencji stanowi pole rozległych badań naukowych i przemysłowych. W ich ramach, projektowane są algorytmy umożliwiające akwizycję, modyfikowanie, analizę, rozpoznawanie treści i prezentację obrazów. Często motywacją badań jest próba naśladowania zjawisk związanych z narządem wzroku człowieka i dążenie do uzyskania takiego opisu sposobu jego działania, aby umożliwić wykonanie zbliżonego do niego algorytmu przy użyciu układów elektronicznych. Odmiennym zagadnieniem jest poszukiwanie możliwości realizacji przetwarzania obrazów w taki sposób, aby uzyskać informacje niewidoczne dla ludzkiego oka, w oparciu o parametry obrazu o niewielkiej zmienności. Temat ten obejmuje analizę obrazów w celu wykrycia możliwych modyfikacji obrazu oryginalnego czy algorytmy wykorzystujące analizę widmową sygnału.

Techniki przetwarzania obrazów opierają się zwykle na analizie i redukcji informacji zawartej w sekwencji pikseli w taki sposób, aby uzyskać obraz wynikowy, na którym uwypuklone będą kluczowe z punktu widzenia algorytmu własności. Wynikiem działania procedury może być również zbiór cech opisujących badane zjawiska.

Zdefiniować można szereg operacji składających się na proces przetwarzania obrazu [1].

- Akwizycja – przygotowanie cyfrowej reprezentacji obrazu, „czytelnej” dla układu obliczeniowego.
- Przetwarzanie – proces modyfikacji danych wejściowych w celu przystosowania do obróbki algorytmicznej, wykorzystujący, między innymi, operacje skalowania, zmiany przestrzeni barw czy usuwania zakłóceń (filtracji).
- Analiza – redukcja informacji wizualnej w celu uzyskania opisu jakościowego lub ilościowego badanych cech i eliminacja zbędnych z perspektywy rozpatrywanego zadania informacji.
- Rozpoznawanie – proces uzyskiwania informacji wynikowych na podstawie wektora cech.

Techniki przetwarzania obrazów, a zwłaszcza ich sekwencji, znajdują zastosowanie w coraz większej liczbie dziedzin nauki i techniki. Jedną z nich jest przede wszystkim rozwijana w ostatnich latach budowa systemów ADAS (*ang. Advanced driver-assistance systems*). Ich działanie, poza sygnałami wizualnymi, wymaga użycia sygnałów o innych charakterystykach, między innymi czujników

optycznych oraz systemów *LIDAR* (*ang.* Light Detection and Ranging). Celem projektowania zaawansowanych systemów wsparcia kierowcy jest stopniowe zwiększenie autonomii pojazdów i ograniczenie zaangażowania kierowcy. W szerszej perspektywie, rozwój systemów ADAS może pozwolić na zaprojektowanie pojazdów w pełni autonomicznych, pozwalających na transport osób i towarów bez udziału kierowców. Dane z czujników wizyjnych mogą być przetwarzane w celu uzyskania informacji na temat lokalizacji i przebiegu jezdni, innych uczestników ruchu, oznakowania czy potencjalnych zagrożeń. Opracowanie współcześnie stosowanych technik znaleźć można w pracach [2, 3].

Inny zbiór technik wykorzystywany jest w celu detekcji i rozpoznawania twarzy oraz badania emocji. Zagadnienie to znajduje zastosowanie w ramach projektowania nie tylko systemów przemysłowych, ale jest również powszechnie stosowane w oprogramowaniu współcześnie dostępnych aparatów cyfrowych czy w ramach serwisów społecznościowych. Metody te mogą również pozwolić na budowę systemów weryfikacji użytkownika bez konieczności zdefiniowania hasła dostępu. Znajdują także zastosowanie w interfejsach przystosowanych do pracy z osobami niepełnosprawnymi. Analizę wykorzystywanych w tym celu algorytmów znaleźć można w pracy [4].

Współcześnie, coraz większe znaczenie mają również systemy śledzenia osób i analizy ich zachowań w celu wykrycia działań niepożądanych. Motywując to zwiększeniem bezpieczeństwa, badane są takie zagadnienia jak detekcja porzuconych bagażów, obecność osób nieuprawnionych w ustalonych strefach czy śledzenie ruchu i re-identyfikacja przy użyciu wielu kamer. Potrzeba automatyzacji wynika ze złożoności projektowanych systemów, które zasięgiem obejmować mogą całe aglomeracje i pozwalać na obserwację zachowań tysięcy osób. Z tego powodu, praktycznie niemożliwe jest zapewnienie odpowiedniej liczby operatorów – tj. takiej, która w pełni pozwoliłaby na wykorzystanie i analizę pozyskanych informacji w czasie rzeczywistym. Omawiane systemy mogą działać niezależnie lub stanowić jeden z elementów zintegrowanego oprogramowania, wykorzystującego dane z wielu źródeł [5, 6, 7].

Równolegle do rozwoju algorytmów wizyjnych, badane są techniki implementacji pozwalające na wykorzystanie ich w systemach uruchamianych na układach elektronicznych różnego typu. Algorytmy wizyjne projektowane są z myślą o uruchamianiu na powszechnie stosowanych układach procesorowych w architekturach rodziny x86 lub ARM, mikrokontrolerach, układach ASIC (*ang.* Application Specific Integrated Circuit) i FPGA (*ang.* Field-Programmable Gate Array).

Pośród wymienionych platform wyróżnić można rodzinę Zynq [8], integrującą możliwości układów FPGA oraz procesorów ARM. Dzięki zastosowaniu logiki programowalnej, możliwe jest projektowanie algorytmów wizyjnych wykonywanych w sposób strumieniowy, zapewniając wysoką wydajność przy stosunkowo niskim zapotrzebowaniu na energię. Uzupełnieniem takiego układu jest procesor ARM, umożliwiający wykorzystanie algorytmów, które wymagają swobodnego dostępu do kontekstu obliczeniowego. Procesor sekwencyjny jest również, w porównaniu

do układów logicznych, lepiej przystosowany do wykonywania algorytmów zdominowanych przez instrukcje lub takich, których sprzętowa implementacja jest trudna lub niemożliwa.

Układy Zynq pozwalają wykorzystać zalety algorytmów projektowanych z myślą o implementacji przy użyciu języków HDL (*ang. Hardware Description Language*) oraz powszechnie stosowanych języków proceduralnych. W szczególności umożliwiają także na uruchomienie systemu operacyjnego, ze szczególnym uwzględnieniem systemu PetaLinux [9], dzięki czemu możliwy jest dostęp do szerokiego zbioru narzędzi związanych z powszechnie stosowanymi systemami operacyjnymi.

1.1. Cel pracy

Celem niniejszej pracy było uruchomienie oraz skonfigurowanie systemu PetaLinux na platformie Zynq, ze szczególnym uwzględnieniem funkcjonalności, które mogą zostać wykorzystane we wbudowanych systemach wizyjnych. W pierwszym etapie przeprowadzono analizę architektury układu oraz dostępnych systemów operacyjnych i systemów czasu rzeczywistego. Następnie, opracowano zagadnienia teoretyczne i praktyczne związane z funkcjonalnościami systemu, które mogą znaleźć zastosowanie we wbudowanych systemach wizyjnych. Ostatecznie, działanie komponentów zaprezentowano na przykładzie wybranego systemu wizyjnego.

1.2. Zawartość pracy

Praca podzielona została na pięć rozdziałów.

Rozdział 2. zawiera opis i analizę platformy Zynq-7000. Omówiono krótko specyfikację układu. Poruszono zagadnienia związane z dostępными systemami operacyjnymi, z uwzględnieniem zalet i wad każdego z proponowanych rozwiązań. Opisano również możliwość wykorzystania systemów czasu rzeczywistego.

Rozdział 3. zawiera analizę funkcjonalności układu, które mogą zostać wykorzystane w systemach wizyjnych. Zbadano możliwości wykorzystania systemu operacyjnego PetaLinux i jego integracji z układem reprogramowalnym. Opisano również protokół AXI, ze szczególnym uwzględnieniem modułów AXI DMA (*ang. Direct Memory Access*) oraz VDMA (*ang. Video DMA*).

W rozdziale 4. zaprezentowano system wizyjny wykorzystujący omawiane funkcjonalności, którego zadaniem jest generacja tła i segmentacja obiektów pierwszoplanowych. Zaproponowano metody integracji rozwiązań implementowanych w obu częściach układu, wskazano ograniczenia i potencjalne kierunki rozwoju.

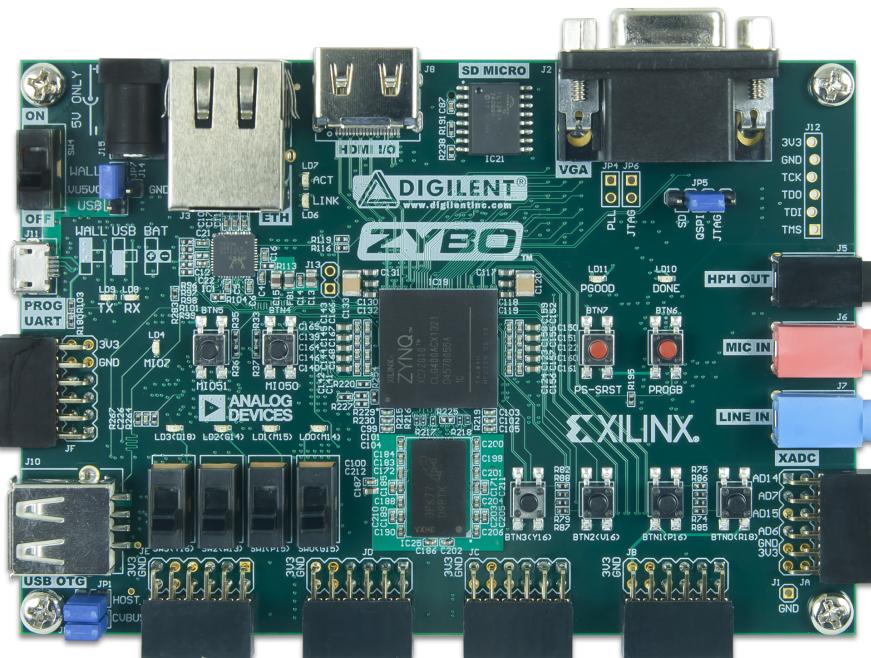
Rozdział 5. zawiera zbiór instrukcji związanych z konfiguracją funkcjonalności omawianych w poprzednich rozdziałach, na przykładzie platformy uruchomieniowej ZYBO. Zaprezentowano w nim kroki wymagane do poprawnej konfiguracji wykorzystywanych elementów systemu oraz wskazano metody umożliwiające weryfikację poprawności działania.

Pracę kończy rozdział zawierający krótkie podsumowanie wykonywanych zadań oraz wskazanie dalszych kierunków rozwoju.

2. Platforma Zynq-7000

W trakcie realizacji pracy wykorzystywano kartę uruchomieniową ZYBO wyposażoną w układ SoC (*ang.* System-on-a-chip) Zynq-7000. SoC to układy scalone integrujące szereg układów elektronicznych, takich jak mikroprocesor, układy koprocesujące, interfejsy wejścia i wyjścia, czy pamięci. Są one powszechnie wykorzystywane do projektowania systemów wbudowanych. Zynq-7000 to rodzina układów heterogenicznych, których centralną część stanowi dwurdzeniowy procesor o architekturze ARM w wersji Cortex-A9 (*PS, ang.* Processing System), współpracujący z działającym równolegle układem FPGA (*PL, ang.* Programmable Logic), opartym na architekturze Artix-7 lub Kintex-7 [10].

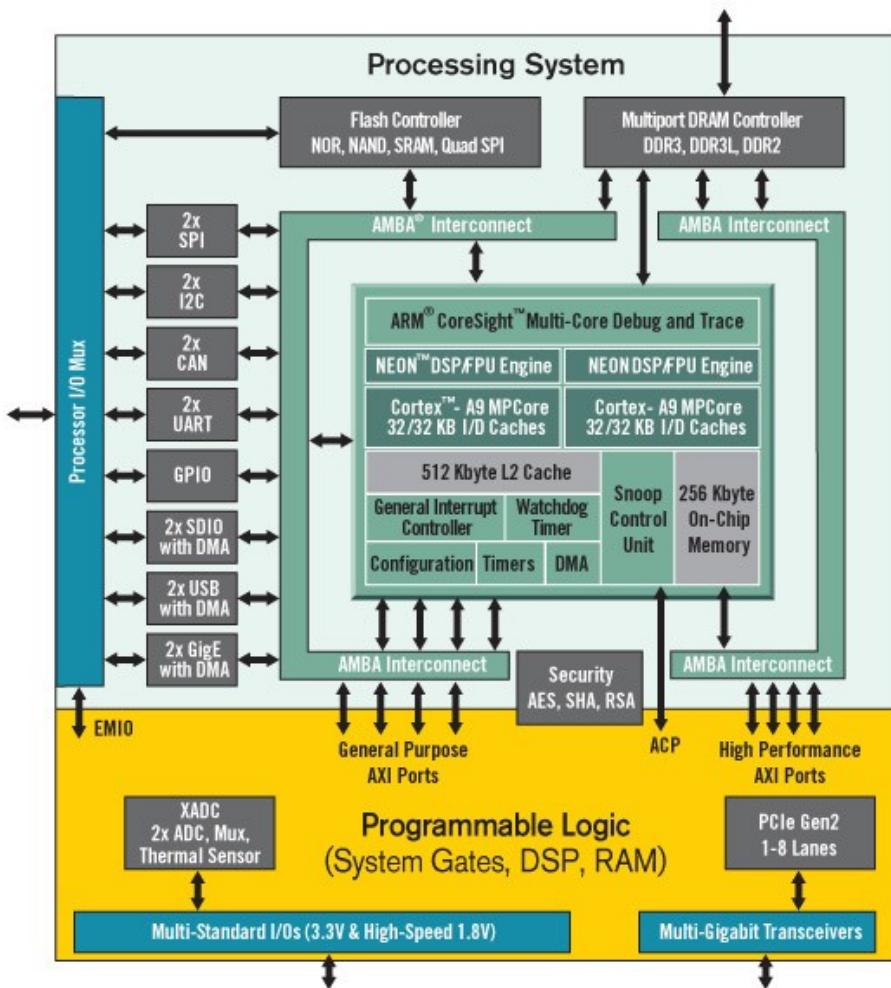
Karta ZYBO wyposażona jest ponadto w 512 MB pamięci RAM, złącza HDMI i VGA do transmisji obrazu, gniazda Jack do przesyłu sygnału dźwiękowego, gniazdo USB oraz slot pamięci MicroSD. Komunikacja sieciowa jest możliwa dzięki implementacji stosu TCP/IP i obecności gniazda RJ-45 [11]. Kartę przedstawiono na rysunku 2.1. Elementy układu podpisane zostały nadrukami.



Rys. 2.1. Karta Digilent ZYBO. (Źródło: [8])

Układy rodziny Zynq-7000 są stosowane w aplikacjach systemów wsparcia kierowców, systemach wizyjnych wysokich rozdzielczości, cyfrowego przetwarzania sygnałów czy kryptograficznych. W pracy [12] zaproponowano wykorzystanie zalet elementów FPGA i CPU do projektowania systemów wsparcia kierowców ADAS, co pozwoliło na redukcję czasu odpowiedzi systemu i zapotrzebowania na energię. W innej pracy [13] zbadano możliwość wykorzystania układu do transmisji sygnału o rozdzielczości 4K (3840×2160 pikseli) przy możliwie niewielkim zużyciu zasobów i energii. Wśród zagadnień kryptograficznych realizowanych przy użyciu omawianej rodziny układu wymienić można algorytmy generowania liczb pseudolosowych o wysokiej rozdzielczości [14].

Na rysunku 2.2 przedstawiono schemat omawianej architektury.



Rys. 2.2. Schemat architektury Zynq-7000. (Źródło: [8])

Schemat przedstawia podział układu na części *PL* – oznaczonej kolorem żółtym, oraz *PS* – na zielono. Architektura części programowej zbliżona jest do powszechnie stosowanych układów FPGA serii 7 firmy Xilinx. Wyposażono ją jednak w zbiór portów umożliwiających wydajną komunikację z procesorem. Ponadto, konfiguracja tej części wykonywana jest na starcie przez procesor lub przy użyciu interfejsu JTAG i układ nie zawiera elementów pozwalających na

wykorzystanie logiki programowalnej w pełni niezależnie. Część procesorowa wyposażona jest w szereg interfejsów, w tym kontroler pamięci DDR3, interfejs komunikacji AMBA oraz zbiór interfejsów peryferyjnych.

Procesor wyposażony jest w koprocessor arytmetyczny (*FPU*) – wsparcie obliczeń na liczbach zmiennoprzecinkowych oraz *NEON* – wsparcie obsługi architektury SIMD (*ang. Single Instruction, Multiple Data*), pozwalającej na przetwarzanie wielu strumieni danych przy użyciu jednego strumienia instrukcji. Zagadnienia te szerzej opisane zostały w rozdziale 3.2.

Układ wyposażony jest w kontroler pamięci DDR, obsługujący żądania dostępu ze strony zarówno procesora, jak i logiki programowalnej. Pamięć jest wspólnie dzielona między obiema częściami. Pozwala to na wymianę danych, do czego wykorzystywany jest standard AXI. Zastosowanie znajduje również mechanizm DMA, pozwalający na przeprowadzanie operacji z użyciem pamięci bez udziału procesora. Interfejs pozwala na transmisję pojedynczych słów danych, umożliwiając konfigurację parametrów pracy modułów algorytmicznych, jak i na transmisję o wysokiej przepustowości. Pozwala to, dzięki zastosowaniu modułu VDMA, na przesyłanie obrazu o rozdzielczości HD z częstotliwością osiągającą wartości 680 klatek na sekundę [15]. Interfejs AXI opisano szerzej w rozdziale 3.3.

Dzięki zastosowaniu kontrolera przerwań (*General Interrupt Controller*) możliwe jest wykorzystanie modułów logiki programowalnej komunikujących się z procesorem z wykorzystaniem techniki zgłaszania żądań. Zagadnienie to opisano w rozdziale 3.7. Ponadto, dostępne są powszechnie spotykane układy zegarowe (*Timers*) i *Watchdog*, odpowiedzialny za przerwanie pracy procesora w przypadku wykrycia błędного wykonania programu. Kontekst pamięciowy synchronizowany jest pomiędzy oboma rdzeniami procesora dzięki modułowi *Snoop Control Unit*.

Układ logiki programowalnej należeć może do rodzin Artix-7 lub Kintex-7 i zbudowany jest z typowych elementów wykorzystywanych w FPGA o różnej liczebności, związanej z klasą układu:

- CLB (*ang. Configurable Logic Blocks*) – w tym tablice Look-up (*LUT*) – 14400 – 277400 elementów, przerzutniki (*FF*) – 28800 – 554800 elementów. *LUT* mogą pracować w trybie pamięci RAM zawierającej szesnaście jednobitowych komórek, kolejki FIFO o długości szesnastu elementów lub generatora czteroargumentowej funkcji logicznej. Przerzutniki *FF* pozwalają na synchronizację zegarową elementów obliczeniowych.
- Pamięci *Block RAM* – od 1,8Mb do 26,5Mb (50 – 755 elementów). *BRAM* to moduły pamięci konfigurowalnej, wspierające operacje odczytu i zapisu. Stanowić mogą alternatywę dla zewnętrznych modułów pamięciowych.
- Elementy *DSP* – wykorzystywane zwykle do implementacji operacji dodawania i mnożenia – 66 – 2020 elementów.
- Bloki *IOB* – umożliwiające budowę interfejsów wejściowych i wyjściowych.

- Inne – w tym interfejs JTAG, PCI Express czy konwertery analogowo-cyfrowe.

2.1. Systemy operacyjne dostępne dla platformy Zynq

Centralny element architektury stanowi dwurdzeniowy procesor ARM. Jest on odpowiedzialny za przeprowadzenie konfiguracji logiki reprogramowalnej. Ponadto pozwala na wykonanie dowolnego programu użytkownika. Powszechnie stosowana jest konfiguracja *bare-metal*, w której procesor wykonuje program zaprojektowany w pełni przez użytkownika. Pozwala to na uzyskanie możliwie największej kontroli nad pracą układu, ogranicza jednak możliwości wykorzystania pełni zasobów procesora oraz utrudnia projektowanie rozbudowanych aplikacji. Brak systemu operacyjnego ogranicza możliwość wykorzystania komunikacji sieciowej na etapie wykonania programu. Możliwości przechowywania wyników i logów aplikacji są niewielkie. Ponadto, użycie zewnętrznych bibliotek, w tym związanych z przetwarzaniem obrazów, takich jak OpenCV, jest niemożliwe. W efekcie, wykorzystanie konfiguracji pozbawionej systemu operacyjnego nie jest możliwe w przypadku aplikacji wymagających nadzoru bez fizycznego dostępu do układu czy przechowywania wyników.

W niniejszej pracy badano możliwość wykorzystania systemu operacyjnego Linux na przykładzie aplikacji systemów wizyjnych. Dzięki zastosowaniu Linuxa, możliwe staje się budowanie programów składających się z wielu modułów działających niezależnie. System ten wspiera obsługę sieci, co pozwala na wykorzystanie narzędzi komunikacji sieciowej, jak SSH [16], do konfiguracji i nadzorowania działania aplikacji, co opisano w rozdziale 3.1. Ponadto, możliwe jest wykorzystanie powszechnie dostępnych bibliotek, ułatwiających rozwój aplikacji w krótkim czasie. Zagadnienie to badano na przykładzie biblioteki OpenCV [17], udostępniającej narzędzia przetwarzania obrazów, co opisano w rozdziale 3.5.

Zbadano możliwość wykorzystania systemu PetaLinux, rozwijanego przez organizację Xilinx, jak i podstawowej wersji systemu, opartej wyłącznie na źródle jądra oraz dystrybucji bazującej na Ubuntu Core. Ponadto, rozpatrzoneno możliwość użycia systemu czasu rzeczywistego do wykonania zadań obliczeniowych z zachowaniem reżimu czasowego.

2.1.1. PetaLinux

Firma Xilinx zapewnia dostęp do zbioru narzędzi *PetaLinux Tools* [9] umożliwiających przeprowadzenie procesu konfiguracji, budowania i uruchamiania systemu Linux na platformie Zynq. Dzięki zintegrowaniu koniecznych narzędzi w jednym pakiecie, proces ten jest w dużej części zautomatyzowany i ogranicza interakcję z programistą, zapewniając przy tym możliwość dowolnej konfiguracji systemu. Celem wykorzystania omawianego pakietu narzędzi jest zbudowanie systemu operacyjnego gotowego do uruchomienia i umożliwiającego szybką rekonfigurację zarówno elementów logiki reprogramowalnej jak i samego systemu operacyjnego.

Pakiet wymaga dostarczenia zewnętrznych zależności, w tym narzędzi umożliwiających budowanie systemu – kompilatora, generatora parserów, systemu budowania – oraz zbioru narzędzi programistycznych i konfiguracyjnych. W przypadku dystrybucji Debian, zależności mogą być zainstalowane poleceniem:

```
apt-get install tofrodos iproute2 gawk gcc git make net-tools libncurses5-dev tftpd  
zlib1g-dev libssl-dev flex bison libselinux1
```

W przypadku pracy na systemie wspierającym architekturę 64-bitową, konieczne jest również zainstalowanie bibliotek programistycznych dla architektury 32-bitowej.

```
dpkg --add-architecture i386  
apt-get update  
apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386  
apt-get install libgtk2.0-0:i386 libxtst6:i386 gtk2-engines-murrine:i386 lib32stdc  
++6 libxt6:i386 libdbus-glib-1-2:i386 libasound2:i386
```

Praca z pakietem wymaga ponadto wykorzystania oprogramowania *Vivado Design Suite* [18] do zaprojektowania układu połączeń logiki reprogramowalnej oraz *Xilinx SDK* [19] do komplikacji programów uruchamianych w środowisku systemu operacyjnego układu. Pierwszym krokiem jest wykonanie projektu w pakiecie *Vivado*. Szczegóły procesu opisano w rozdziale 5.2.1. Wyeksportowany projekt Vivado jest konieczny do przeprowadzenia procesu budowania systemu. Proces konfiguracji projektu PetaLinux opisano w rozdziale 5.2.3.

Pakiet udostępnia możliwość dodania do budowanego systemu zbioru programów i bibliotek. Dostępne jest kilkaset pakietów oprogramowania oferowanych na zasadach wolnych licencji, w tym biblioteki do przetwarzania obrazów. Ponadto, pakiet umożliwia dodanie własnoręcznie zbudowanych aplikacji. Pozwala to na integrację etapu projektowania aplikacji oraz budowy i uruchamiania systemu operacyjnego w jednym procesie.

Projekt PetaLinux jest niezależny od projektu Vivado i może powstawać równolegle. Zmiany w strukturze modułów logiki reprogramowalnej wymagają ponownego zbudowania plików wynikowych systemu operacyjnego PetaLinux, jednak proces ten został wydzielony z oprogramowania Vivado. Pozwala to na wykorzystanie jednego projektu opisującego logikę współpracującego z aplikacjami *bare-metal* i systemowymi.

Proces budowania systemu jest czasochłonny, na etapie prototypowania aplikacji praktyczne jest zastosowanie oprogramowania pracującego w trybie *bare-metal*. Pozwala to na przeprowadzenie procesu debugowania aplikacji bezpośrednio z poziomu oprogramowania Vivado/SDK. Po upewnieniu się, że sprzętowa część algorytmu działa poprawnie, zaprojektować można aplikację systemową, odpowiedzialną za komunikację, monitorowanie i wykorzystanie wyników działania algorytmu w kompletnym projekcie.

2.1.2. Inne dystrybucje systemu

Niezależnie do analizy zastosowania PetaLinux, zbadano również inne możliwości konfiguracji systemu operacyjnego do zastosowania na platformie ZYBO. Wśród dostępnych opcji, rozpatrzono dystrybucję Ubuntu Core oraz budowę systemu Linux ze źródeł.

2.1.2.1. Budowa ze źródeł

Wykorzystanie pakietu PetaLinux związane jest z ograniczeniem dostępności projektu do środowisk, dla których ten pakiet narzędzi jest dostępny. W przypadku konieczności uruchomienia projektu na systemie nie wspieranym przez twórców oprogramowania lub potrzeby wprowadzenia dużych zmian w kodzie źródłowym systemu i konfiguracji, konieczne może być przeprowadzenie pełnego procesu budowania samodzielnie. Takie podejście pozwala również na pełne zrozumienie znaczenia kolejnych kroków procesu konfiguracji.

Proces budowy systemu składa się z kilku etapów.

1. Plik binarny zawierający konfigurację części oprogramowania wykorzystującej logikę programowaną dołączany jest w trakcie procesu budowania systemu operacyjnego.
2. Konieczne jest zbudowanie dwóch programów rozruchowych (*ang. bootloader*). Pierwszy z nich – FSBL (*ang. First Stage Boot Loader*) – odpowiada za przeprowadzenie procesu wstępnej konfiguracji procesora, kontrolera pamięci i uruchomienie drugiego programu rozruchowego. Na drugim etapie rozruchu wykorzystywany jest program U-Boot. Jego zadaniem jest przygotowanie środowiska do uruchomienia właściwego systemu operacyjnego.
3. Kolejny krok wymaga zbudowania struktury drzewa urządzeń (*ang. device tree*). Pozwala ona na zdefiniowanie i konfigurację urządzeń połączonych z procesorem, dzięki czemu mogą być one obsłużone przez system operacyjny. W przypadku układu Zynq, wykorzystanie tej struktury pozwala na konfigurację i komunikację z elementami układu FPGA.
4. Po przeprowadzeniu wstępnej konfiguracji elementów systemu, możliwe jest wykonanie procesu konfiguracji, budowania i przygotowania wynikowych plików binarnych.

Opisany proces jest skomplikowany i wymaga specjalistycznej wiedzy. Ze względu na konieczność integracji wielu niezależnych modułów, takich jak programy rozruchowe czy *device tree*, jego przebieg może ulegać zmianom w kolejnych wersjach systemu. Dostępne są obszerne opracowania tematu, zawierające precyzyjny opis kolejnych wymaganych kroków [20, 21, 22].

2.1.2.2. Ubuntu Core

Ubuntu Core to dystrybucja systemu Linux dedykowana do zastosowań w urządzeniach tzw. internetu rzeczy (*IoT – ang. Internet of Things*). Dystrybucja ta oparta jest na podstawowej wersji systemu Ubuntu, przystosowana do uruchomienia na urządzeniach o ograniczonej mocy obliczeniowej.

Dzięki wykorzystaniu Ubuntu Core, możliwy jest dostęp do repozytorium oprogramowania udostępnianego przez dystrybucję. W przeciwieństwie do dystrybucji PetaLinux, instalowane oprogramowanie może być aktualizowane w trakcie pracy systemu. Cechą ta może być istotna w przypadku aplikacji działających przez długi czas, gdy aktualizacja oprogramowania jest korzystna ze względu na znalezione błędy lub poprawę bezpieczeństwa w kolejnej wersji. System Ubuntu Core może być zbudowany i uruchomiony na karcie ZYBO przy użyciu dedykowanego narzędzia [23].

2.1.3. RTOS

System operacyjny czasu rzeczywistego (*RTOS, ang. Real Time Operating System*) to system operacyjny, którego zadaniem jest obsługa aplikacji przy zachowaniu założenia o nieprzekroczeniu maksymalnego dopuszczalnego czasu odpowiedzi programu. Pozwala to na projektowanie aplikacji, w których czas odpowiedzi ma kluczowe znaczenie, w tym systemów sterowania lub krytycznych aplikacji wizyjnych. Dzięki zastosowaniu dwurdzeniowego procesora w układzie Zynq, rozważyć można zaprojektowanie rozwiązań, w których jeden z rdzeni odpowiada za wykonanie programu Linuxa, a drugi – aplikacji lub systemu czasu rzeczywistego.

Rozpatrzone możliwość uruchomienia systemu operacyjnego PetaLinux i jego współpracę z aplikacjami czasu rzeczywistego na przykładzie OpenAMP [24]. OpenAMP zapewnia interfejs umożliwiający komunikację pomiędzy programami działającymi w systemie Linux oraz aplikacjami czasu rzeczywistego, wykorzystując do tego narzędzia dostępne już w systemie. Z punktu widzenia klasycznego systemu operacyjnego, program działający w systemie czasu rzeczywistego jest zewnętrznym zasobem, który oczekuje na zlecenie wykonania konkretnego zadania i wysyła odpowiedź. Dzięki wykorzystaniu systemu czasu rzeczywistego FreeRTOS [25], aplikacje mogą mieć dostęp do zasobów systemowych, w tym pamięci i interfejsów komunikacji.

System czasu rzeczywistego może być wykorzystany do obliczeń o krytycznym znaczeniu. W przypadku wykorzystania klasycznego systemu operacyjnego, nie jest możliwe zagwarantowanie wykonania dowolnego zadania w określonym czasie. W trakcie działania aplikacji, system może zadecydować o jej czasowym zatrzymaniu i udostępnieniu zasobów innemu z oczekujących zadań. Aplikacja działająca w czasie rzeczywistym pozwala uniknąć tego zjawiska.

System PetaLinux oferuje dostęp do modułów `RPMmsg`, `remoteproc`, `virtIO`, które są wymagane do zapewnienia komunikacji z systemem czasu rzeczywistego. Udostępnione zostały również aplikacje testowe, które pozwalają na sprawdzenie poprawności działania konfiguracji. Użycie systemu czasu rzeczywistego wymaga zmian projektowych, w tym konfiguracji dwóch instancji konsoli do komunikacji szeregowej i zdefiniowania struktury drzewa urządzeń określającej obszar pamięci dla obu systemów operacyjnych. Po zbudowaniu poprawnie skonfigurowanego systemu i jego uruchomieniu, przetestowanie działania aplikacji wymaga użycia poniższych poleceń.

```
modprobe zynq_remoteproc firmware=image_echo_test
modprobe rpmsg_user_dev_driver
```

```
echo _test
```

W rezultacie uruchomione zostaną moduły odpowiedzialne za obsługę systemu czasu rzeczywistego i przeprowadzony będzie test komunikacji. Proces konfiguracji pozwalający na zastosowanie OpenAMP przedstawiono w rozdziale 5.6. Wykorzystanie systemów czasu rzeczywistego wykracza poza zakres niniejszej pracy, a zagadnienie jest obiektem obszernych opracowań [26, 27].

Podsumowanie

Zarówno wykorzystanie pakietu PetaLinux Tools, jak i obu pozostałych metod pozwala na zbudowanie w pełni funkcjonalnej dystrybucji systemu Linux i uruchomienie jej na układzie Zynq. Każda z metod wiąże się z innymi ograniczeniami i udostępnia inne możliwości. W przypadku użycia narzędzi PetaLinux, użytkownik uzyskuje dostęp do ograniczonego zbioru dodatkowych aplikacji, niewielkiej w porównaniu do repozytoriów udostępnianych w dystrybucji Ubuntu Core. Ponadto aktualizacja oprogramowania może wymagać ponownego zbudowania systemu lub nie być możliwa bez aktualizacji pakietu narzędzi.

Dystrybucja Ubuntu Core zapewnia dostęp do aktualizacji samego systemu, pozwalając na zachowanie bezpieczeństwa działania i dostęp do poprawek kodu oprogramowania. Może to być kluczowe w przypadku wykorzystania układu Zynq do działania przez długi czas z oczekiwana niezawodnością.

W przypadku konieczności dostosowania kodu systemu operacyjnego do własnych potrzeb, praktyczne staje się natomiast wykorzystanie technik budowy systemu bezpośrednio ze źródeł. Ogranicza to jednak możliwości instalacji dodatkowego oprogramowania i wymaga dobrej znajomości zagadnień związanych z działaniem systemu Linux. Proces budowy systemu operacyjnego może być trudny do zautomatyzowania ze względu na konieczność integracji wielu elementów. Zmiany w implementacji każdego z nich mogą wpłynąć na przebieg zadania i wymagać modyfikacji procedur budowania.

Pakiet PetaLinux pozwala jednak na największą integrację z oprogramowaniem Vivado, co ułatwia prototypowanie aplikacji. Dzięki udostępnieniu repozytorium oprogramowania oraz braku konieczności ingerencji użytkownika w proces budowania systemu, wykorzystanie go jest najlepszym rozwiązaniem w większości projektów. Z tego powodu, w niniejszej pracy zdecydowano się na użycie tego rozwiązania na dalszym etapie projektu.

Zastosowanie systemu czasu rzeczywistego współpracującego z innym systemem operacyjnym pozwala na wykonanie krytycznych sekcji kodu z zachowaniem ograniczeń czasowych. Pamiętać należy jednak, że wiąże się to z ograniczeniem maksymalnej wydajności operacji wykonywanych przez klasyczny system operacyjny.

Firma Xilinx zrezygnowała ze wsparcia dla systemu FreeRTOS i podobnych i zdecydowano się na oparcie na bibliotece OpenAMP do realizacji zadań wykonywanych w czasie rzeczywistym.

W okresie powstawania pracy, literatura omawiająca integrację biblioteki z systemem operacyjnym dla kart innych producentów nie była powszechnie dostępna. Z tego powodu, realizacja omawianych zadań w przypadku karty ZYBO była poważnie utrudniona. Udostępnione materiały nie uwzględniały możliwości użycia biblioteki na karcie ZYBO. Ponadto, środowisko SDK nie wspierało budowy aplikacji czasu rzeczywistego na wykorzystywaną kartę i wymagało użycia rozwiązań dedykowanych innym kartom rodziny Zynq-7000.

3. Przegląd wybranych funkcjonalności platformy Zynq i systemu operacyjnego PetaLinux

W ramach pracy przeprowadzono analizę wybranych funkcjonalności układów rodziny Zynq działających pod kontrolą systemu operacyjnego PetaLinux, które mogą zostać zastosowane podczas realizacji wbudowanych systemów przetwarzania obrazów. Wybrane zagadnienia przedstawiono w poniższych podrozdziałach.

3.1. Obsługa SSH

Po połączeniu karty z uruchomionym systemem PetaLinux z komputerem przez interfejs USB, możliwe jest otworzenie konsoli komunikacji przy użyciu protokołu transmisji szeregowej. Komunikacja odbywa się z prędkością 115200 bodów, ośmioma bitami danych, jednym bitem stopu i bez bitu parzystości.

Komunikacja przy użyciu transmisji szeregowej jest jednak ograniczona do sytuacji, w których możliwy jest bezpośredni dostęp do układu. Ponadto, nie zapewnia wysokiej przepustowości transmisji czy możliwości przesyłu plików. Z tych powodów, korzystne staje się wykorzystanie protokołu SSH (*ang. Secure Shell*) do nawiązania komunikacji sieciowej. SSH jest najczęściej stosowanym protokołem bezpiecznej komunikacji, wspieranym przez zdecydowaną większość dystrybucji systemu Linux i nie wymagającym dodatkowej konfiguracji na etapie budowania systemu. Dzięki zastosowaniu technik szyfrowania połączenia, zapewnia mechanizm nawiązywania kryptograficznie bezpiecznej komunikacji między dwoma urządzeniami. Wśród najczęściej wykorzystywanych funkcji protokołu wymienić można udostępnienie zdalnej konsoli poleceń, przesyłanie plików czy tunelowanie połączeń.

Połączenie odbywa się przy użyciu poniższego polecenia.

```
ssh root@adres-ip-urządzenia
```

Domyślne hasło administratora w przypadku dystrybucji PetaLinux to **root**. Może być ono zmienione na etapie konfiguracji systemu z wykorzystaniem poniższych poleceń.

```
petalinux-config -c rootfs  
Petalinux RootFS Settings -> Root password
```

Aby uprościć proces logowania, wykorzystać można mechanizm wymiany kluczy, zapewniany przez protokół. Weryfikacja obu stron połączenia przy użyciu kluczy wykorzystuje techniki kryptografii asymetrycznej. Użytkownik posiada parę związków ze sobą kluczy kryptograficznych, umownie nazywanych kluczem publicznym i prywatnym. Wiadomość zaszyfrowana przy użyciu klucza publicznego może zostać odszyfrowana wyłącznie z użyciem klucza prywatnego. Użytkownik udowodnić może swoją tożsamość przez przesłanie oryginału otrzymanej wiadomości, zaszyfrowanej przy użyciu klucza publicznego. Korzystając z tej zależności, klucz publiczny może być powszechnie znany i wykorzystywany do budowania kryptograficznie bezpiecznych wiadomości, pod warunkiem zachowania klucza prywatnego w tajemnicy.

Algorytmy kryptografii asymetrycznej wykorzystywane są przez narzędzie SSH na etapie weryfikacji tożsamości obu stron komunikacji. Po nawiązaniu połączenia, komunikacja zabezpieczana jest wybranym algorytmem symetrycznym. Wykorzystanie mechanizmu kluczy wymaga użycia poniższego polecenia.

```
ssh-copy-id -i ~/.ssh/id_rsa.pub root@adres-ip-urzadzenia
```

Umożliwia to logowanie w przyszłości bez konieczności podania hasła użytkownika. Skonfigurowany w opisany sposób protokół daje dostęp do pełnego zbioru narzędzi, w tym zdalnej obsługi konsoli użytkownika, przesyłania plików, tunelowania portów czy zdalnego montowania systemów plików.

3.2. FPU i technologia NEON

Układ Zynq wyposażony jest w koprocessor arytmetyczny oraz wspiera polecenia wykorzystujące technologię NEON [28]. Elementy te pozwalają na zwiększenie wydajności projektowanych aplikacji w przypadku, gdy wykonywane operacje wymagają przeprowadzania obliczeń na liczbach zmiennoprzecinkowych lub działań wektorowych.

Koprocessor arytmetyczny, FPU (*ang.* Floating-point unit), to układ działający we współpracy z jednostką procesora, dedykowany do wykonywania obliczeń na liczbach zmiennoprzecinkowych. Wykorzystanie dedykowanego układu pozwala na zwiększenie szybkości wykonywania operacji arytmetycznych, pierwiastkowania i przesunięć bitowych. W przypadku braku układu FPU, konieczne jest symulowanie jego działania dla jednej operacji na liczbach zmiennoprzecinkowych przez wykonywanie większej liczby operacji na typach całkowitych, co wiąże się ze spadkiem wydajności.

Technologia NEON pozwala na rozszerzenie puli rozkazów procesora ARM o polecenia wykorzystujące architekturę SIMD zdefiniowaną przez taksonomię Flynn'a [29]. SIMD (*ang.* Single Instruction stream, Multiple Data streams) to klasa systemów, które pozwalają na przetwarzanie wielu strumieni danych na podstawie jednego strumienia instrukcji. Zastosowania tej architektury obejmują zagadnienia, w których dla wielu wartości wejściowych konieczne jest wykonanie

tej samej operacji. Cechę tę posiada wiele operacji związanych z przetwarzaniem sygnałów i obrazów, w tym wyznaczanie wartości szybkiej transformaty Fouriera, implementacje filtrów FIR i IIR czy operacje skalowania, rotacji i filtracji uśredniającej obrazu.

Rozpatrzone możliwość wykorzystania architektury NEON w zagadnieniach przetwarzania sygnałów. Działanie testowano na podstawie programu wyznaczającego wartość iloczynu skalarnego dwóch wektorów zadanej długości. Porównano trzy implementacje algorytmu, którego kod źródłowy zawarto w dodatku B. Wykorzystano implementację bazową oraz stosującą polecenia dostępne w architekturze NEON i porównano wyniki z implementacją zaprojektowaną w asemblerze.

Implementacja w architekturze NEON wykorzystuje dedykowane funkcje, udostępnione w bibliotece `arm_neon.h`, które mają na celu maksymalne zwiększenie wydajności aplikacji. W przypadku pozostałych implementacji, stosowane są polecenia wykonywane na koprocesorze VFP (ang. Vector Floating-Point). VFP to układ niezależny od FPU, pozwalający na wykonanie jednej instrukcji dla wektora danych wejściowych. Układ ten nie należy do rodziny SIMD i wykonuje instrukcje sekwencyjnie, w przeciwieństwie do architektury NEON.

Tabela 3.1. Wyniki testu wydajnościowego.

Bez optymalizacji			
Implementacja	min [s]	max [s]	średnio [s]
Bazowa	0,4266	0,4339	0,4296
NEON	0,1103	0,1108	0,1105
ASM	0,4082	0,4086	0,4083
Z optymalizacjami			
Bazowa	0,1080	0,1152	0,1092
NEON	0,1088	0,1147	0,1090
ASM	0,1087	0,1144	0,1089

Eksperyment przeprowadzono z wykorzystaniem karty ZYBO działającej pod kontrolą systemu PetaLinux. Rozpatrzone przeprowadzenie procesu komplikacji z wyłączenymi optymalizacjami kompilatora (flaga `-O0`) oraz z włączonymi wszystkimi optymalizacjami (`-O3`). Wykorzystane polecień NEON wymaga użycia odpowiadających im parametrów komplikacji. Poniżej przedstawiono polecenie komplikacji testowej implementacji wykorzystującej NEON.

```
arm-linux-gnueabihf-gcc -Wall -O3 -mcpu=cortex-a9 -mfpu=neon -ftree-vectorize -mvectorize-with-neon-quad -mfloat-abi=hard -ffast-math -funsafe-math-optimizations -g -c -o "src/main.o" "../src/main.c"
```

Wyniki testów wydajności zebrane w tabeli 3.1. Aby zminimalizować wpływ systemu operacyjnego na przebieg eksperymentu, każdy etap wykonywany był tysiąc razy, a na bazie wyników wyznaczono wartości maksymalnego i minimalnego czasu wykonania, a także wartość średnią.

Dane wejściowe algorytmu były wspólne dla każdej implementacji i generowane w sposób pseudolosowy na etapie wykonania programu, co uniemożliwiało zastosowanie przez kompilator części technik optymalizacyjnych, które mogłyby mieć niepożądany wpływ na wiarygodność testu. Czas wymagany do wygenerowania danych wejściowych nie został uwzględniony w zebranych wynikach.

W sytuacji, gdy wyłączono optymalizację na etapie komplikacji, zauważalny jest znaczny wzrost wydajności w przypadku wykorzystania instrukcji udostępnianych przez architekturę NEON. Pozwala ona na niemal czterokrotne zwiększenie szybkości działania programu względem pozostałych implementacji. Różnica ta zanika w przypadku wykorzystania możliwości optymalizacji kodu programu na etapie komplikacji. Użyty kompilator – `arm-linux-gnueabihf-gcc` w wersji 5.2.1 – wykorzystał w przypadku implementacji bazowej instrukcje VFP. Dzięki temu, również różnica czasu wykonania procedury bazowej względem implementacji w asemblerze jest niewielka.

Wyniki pozwalają wnioskować o słuszności wykorzystania instrukcji udostępnianych przez architekturę NEON ze względu na możliwy wzrost wydajności. Istotna jest jednak weryfikacja wyników i potwierdzenie poprawy działania aplikacji. W przypadku, gdy różnice między programami są niewielkie, użycie instrukcji NEON może być niekorzystne ze względu na zwiększoną latencję wykonania rozkazów – w przypadku aplikacji o niewielkiej liczbie instrukcji warunkowych i skoków, porównywalne wyniki może dać wykorzystanie VFP. Dzięki możliwości wykorzystania na etapie komplikacji technik optymalizacji szybkości działania aplikacji, projektowanie wersji algorytmów dedykowanych używanej architekturze w asemblerze nie jest konieczne w zdecydowanej większości przypadków.

3.3. Protokół AXI

Protokół AXI (*ang. Advanced eXtensible Interface*) zdefiniowany został w specyfikacji AMBA (*ang. Advanced Microcontroller Bus Architecture*) 3. W kolejnej wersji dokumentu sprecyzowano standard w najnowszej wersji – AXI4 [30]. Protokół wykorzystywany jest do komunikacji pomiędzy elementami układu lub modułami zbudowanymi wewnątrz logiki reprogramowalnej i jest dedykowany systemom o dużej wydajności i pracującym z wysoką częstotliwością.

Specyfikacja definiuje trzy typy interfejsu:

- AXI4 – wykorzystujące technikę MMIO (*ang. Memory-Mapped Input/Output*) do odwzorowania rejestrów w przestrzeni adresowej pamięci RAM i dedykowanej aplikacjom wymagającym dużej wydajności komunikacji.
- AXI4-Lite – uproszczona wersja protokołu, wykorzystująca MMIO i dedykowana aplikacjom o mniejszych rozbudowanych wymaganiach komunikacyjnych.
- AXI4-Stream – wersja przepływowowa protokołu, nie wykorzystująca technik MMIO.

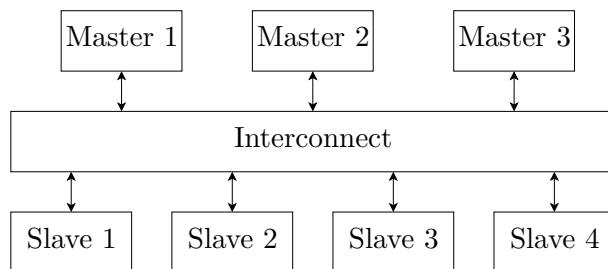
Interfejsy wykorzystujące technikę MMIO stosowane są powszechnie w zadaniach konfiguracji modułów aplikacji czy przesyłania informacji, takich jak ramka sygnału wizyjnego do pamięci procesora. Dzięki reprezentacji stanu elementów logiki reprogramowalnej w postaci komórek pamięci operacyjnej procesora, możliwa jest jednolita analiza działania całego systemu.

Interfejs w wersji *Stream* wykorzystywany jest natomiast do przesyłania sygnału pomiędzy kolejnymi elementami układu, na przykład transmisji kolejnych pikseli obrazu pomiędzy kolejnymi składowymi algorytmu przetwarzania obrazu. Proces przesyłania danych w takiej formie charakteryzuje się większą wydajnością, analiza działania aplikacji jest jednak utrudniona ze względu na brak reprezentacji przesyłanych danych w pamięci.

Możliwe jest również połączenie obu typów interfejsu wewnętrz jednego elementu. Technika ta wykorzystana została w przypadku elementu AXI VDMA, umożliwiając manipulowanie ramkami obrazu wizyjnego przesyłanymi przy użyciu interfejsu *Stream* dzięki buforowaniu w pamięci RAM. Zagadnienie to szerzej opisano w rozdziale 3.3.3. Podobne techniki wykorzystano również w przypadku interfejsu Ethernet DMA, umożliwiającego komunikację przy użyciu protokołu Ethernet.

3.3.1. Przebieg transakcji

Transakcja komunikacyjna odbywa się pomiędzy dwoma urządzeniami – *master* i *slave*, jednak dzięki zastosowaniu elementów AXI-Interconnect możliwe jest połączenie wielu urządzeń, co przedstawiono na schemacie 3.1.



Rys. 3.1. Schemat połączenia Interconnect w protokole AXI.

Komunikacja odbywa się przy użyciu pięciu niezależnych kanałów:

- Read Address
- Write Address
- Read Data
- Write Data
- Write Response

Każdy kanał zawiera zbiór sygnałów wykorzystywanych w trakcie wymiany danych.

Transmisja rozpoczyna się od wykorzystania sygnałów *valid* i *ready*. Urządzenie źródłowe wymusza stan wysoki sygnału *valid* i oczekuje na zmianę wartości sygnału *ready* urządzenia docelowego na stan wysoki. W chwili, gdy oba sygnały znajdują się w tym stanie, właściwe dane mogą zostać przesłane z urządzenia źródłowego do docelowego. Pozwala to na przekazanie takich danych jak adres odczytu/zapisu do pamięci, odczytywanych lub zapisywanych danych i potwierdzenia zapisu. Proces nawiązania transakcji odbywa się niezależnie dla każdego wykorzystywanego kanału.

Procedura odczytu danych składa się z dwóch etapów:

1. Zdefiniowanie przez urządzenie *master* adresu i parametrów transmisji danych na kanale *Read Address*.
2. Przesłanie przez urządzenie *slave* jednej lub więcej wartości danych na kanale *Read Data*.

Natomiast procedura zapisu wymaga trzech etapów:

1. Zdefiniowanie przez urządzenie *master* adresu i parametrów transmisji danych na kanale *Write Address*.
2. Przesłanie przez urządzenie *master* jednej lub więcej wartości danych na kanale *Write Data*.
3. Przesłanie przez urządzenie *slave* odpowiedzi na kanale *Write Response*.

Protokół pozwala ponadto na przesłanie do 256 wartości danych w trakcie jednej transmisji dzięki technice *burst*, a transakcje odczytu i zapisu danych mogą odbywać się równolegle. Przepływ danych w interfejsie AXI4-Stream odbywa się wyłącznie w jednym kierunku i nie jest możliwy odczyt danych przesyłanych wcześniej przez urządzenie *master* do *slave*. Procedura ta jest podobna do transakcji zapisu, jest jednak rozszerzona o możliwość dzielenia operacji na kilka mniejszych i łączenia wielu transakcji w jedną.

3.3.2. AXI DMA

DMA (*ang. Direct Memory Access*) to technika często stosowana w przypadku konieczności wykonywania operacji na pamięci RAM urządzenia z dużą szybkością. Wykorzystanie kontrolera DMA pozwala przeprowadzać operacje odczytu i zapisu do pamięci operacyjnej bez konieczności użycia głównej jednostki procesora. Dzięki temu, procesor odpowiada wyłącznie za skonfigurowanie kontrolera DMA i może wykonywać inne operacje w trakcie transmisji danych. Ponadto, stosowanie kontrolera DMA pozwala zwykle na uzyskanie wyższej przepustowości komunikacji z pamięcią i zmniejszenie zużycia energii. Kontroler DMA może również przeprowadzać podstawowe operacje konwersji sygnałów, na przykład, w przypadku sygnału wizjynego, konwersję

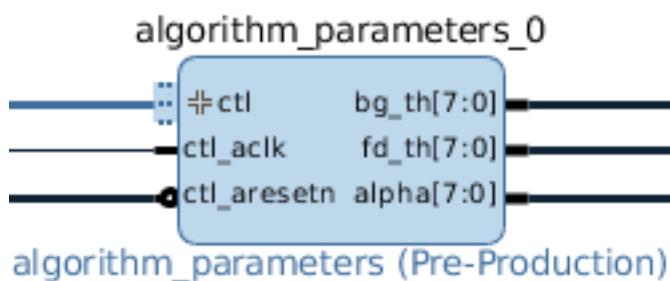
sygnałów synchronizacji obrazu – kontroler odpowiada za odpowiednie wyrównanie (*ang. alignment*) danych w pamięci, tak by zachować odstępy poprawnej długości pomiędzy kolejnymi liniami obrazu.

DMA pozwala na przesłanie wielu wartości danych w ramach jednej transakcji w trybie *burst*. *Master* przesyła wyłącznie adres pierwszego bajta danych, a kolejne adresy wyznaczane są w trakcie operacji przez urządzenie *slave*. Wyznaczany adres może być zwiększany, w przypadku, gdy operacja wykonywana jest w pamięci, bądź mieć stałą wartość, co ma miejsce w przypadku zapisu lub odczytu z kolejki FIFO (*ang. First In, First Out*). Interfejs pozwala również na ograniczenie dostępnej przestrzeni adresowej, w efekcie czego wartość adresu po przekroczeniu górnej granicy zakresu przyjmuje ponownie najniższą dopuszczalną wartość. Własność ta może być wykorzystana do projektowania linii buforujących.

Protokoły transmisji danych wykorzystywać mogą kolejność bajtów od najmniej znaczącego lub odwrotną. Należy o tym pamiętać na etapie projektowania modułów odpowiedzialnych za proces komunikacji. Protokół AXI DMA wykorzystuje kolejność bitów, w której najmniej znaczący bajt umieszczony jest jako pierwszy.

Dzięki zastosowaniu techniki DMA możliwa jest konfiguracja parametrów pracy algorytmu zaprojektowanego w układzie logiki reprogramowalnej oraz obserwacja jego działania na etapie wykonania z poziomu procesora ARM. Moduł algorytmiczny może udostępniać rejesty konfiguracyjne w przestrzeni adresowej pamięci procesora, a wyniki działania programu mogą być przesyłane do pamięci operacyjnej. W szerszej perspektywie, pozwala to na udostępnienie interfejsu użytkownika, umożliwiającego nadzór nad pracą algorytmu, na przykład z poziomu konsoli dostępnej przez *ssh* lub w formie interfejsu strony internetowej. Możliwe jest również przesyłanie powiadomień z elementów logiki programowej do procesora ARM w celu wymuszenia reakcji na osiągnięty stan programu, na przykład przesłanie informacji o ukończeniu iteracji algorytmu dla aktualnej ramki obrazu. Można w tym celu wykorzystać mechanizm przerwań systemowych, co szerzej opisano w rozdziale 3.7.

Mechanizm DMA zbadano na przykładzie projektu modułu umożliwiającego modyfikację parametrów oraz odczyt aktualnego stanu parametrów. Schemat strukturalny modułu przedstawiono na rysunku 3.2.



Rys. 3.2. Graficzna reprezentacja modułu AXI DMA w programie Vivado.

Moduł wyposażony jest w interfejs AXI, podpisany `ctl` oraz związane z nim sygnały, zegarowy – `ctl_aclk` oraz reset – `ctl_aresetn`. Sygnały wyjściowe pozwalają na odczyt zdefiniowanych parametrów z poziomu innych modułów logiki reprogramowalnej. Dzięki wydzieleniu modułu odpowiedzialnego za konfigurację algorytmu z częścią wykonującą obliczenia algorytmiczne, możliwe jest uproszczenie kodu języka opisu sprzętu związanego z każdym z modułów oraz zwiększenie czytelności schematu. Jeden moduł konfiguracyjny może być związany z kilkoma, działającymi niezależnie, modułami algorytmicznymi. Ponadto, zmiany w strukturze algorytmu są uproszczone. Proces projektowania oraz komunikacji z modulem przedstawiono w rozdziale 5.3.

3.3.3. AXI Video DMA

Interfejs AXI VDMA pozwala na wykorzystanie techniki DMA w przypadku aplikacji przetwarzających sygnał wizyjny. Mechanizm *Video DMA* oparty został na wykorzystaniu protokołu AXI w wersji Stream oraz Memory Mapped w połączeniu z techniką DMA do buforowania sygnału wizyjnego. Sygnał wizyjny przesyłany jest do modułu przy użyciu protokołu strumieniowego, gdzie następnie jest buforowany i zapisywany do komórek zewnętrznej pamięci RAM. Przechowywany obraz może być odczytany z poziomu procesora ARM. Moduł wspiera również komunikację w drugą stronę, pozwalając na odczyt obrazu z pamięci i przesłanie go dalej w postaci strumienia. Połączenie tych technik pozwala na wykorzystanie modułu do buforowania obrazu lub w celu rozdzielenia zadań algorytmicznych pomiędzy FPGA i CPU.

Moduł VDMA pozwala na zdefiniowanie do trzydziestu dwóch buforów ramek obrazu. Operacje mogą być wykonywane cyklicznie na każdym buforze lub stale na jednym z nich. Pozwala to na wielokrotną transmisję jednej klatki obrazu. Dane w buforze reprezentują kompletne ramki obrazu w ciągłym fragmencie pamięci, umożliwiając swobodny dostęp do poszczególnych pikseli. Struktura danych jest identyczna do tablicy zawierającej wartości kolejnych pikseli w obrazie.

Powszechnie wykorzystywanym zastosowaniem modułu jest mechanizm potrójnego buforowania, umożliwiający zmianę częstotliwości taktowania zegara sygnału wizyjnego. Zapis i odczyt danych może odbywać się niezależnie z tego samego lub różnych buforów. Dzięki zastosowaniu trzech buforów, zagwarantować można, że zapis i odczyt danych zawsze odbywa się z niezależnych obszarów pamięci, co pozwala uniknąć zjawiska nadpisania przechowywanych danych przed ich wyświetleniem. W niniejszej pracy rozpatrzono możliwość wykorzystania modułu VDMA w celu obsługi algorytmów wymagających kontekstu w postaci dwóch kolejnych ramek obrazu. Proces konfiguracji modułu przedstawiono w rozdziale 5.4.

3.4. Obliczenia równoległe

Ze względu na wykorzystanie w układzie Zynq procesora ARM o dwóch rdzeniach, możliwe jest rozważenie zagadnienia zwiększania szybkości wykonania algorytmu przez zróżwnoleglenie

obliczeń w dwóch wątkach. Stosując prawo Amdahla, wykazać można, że maksymalne przyspieszenie, jakie można uzyskać w systemie wieloprocesorowym jest proporcjonalne do liczby elementów obliczeniowych. Zależność ta zachodzi po warunkiem, że całe zadanie może być realizowane w sposób równoległy. W przypadku omawianego procesora, spodziewać się można korzyści nie przekraczających dwukrotnego zwiększenia szybkości wykonania algorytmu.

Zagadnienia związane z obliczeniami równoległymi stanowią obszar aktualnych badań, których efekty pozwoliły na zaprojektowanie zbioru bibliotek ułatwiających wykorzystanie własności systemów wieloprocesorowych w praktyce. W ramach pracy rozważono możliwości wykorzystania wątków natywnych oraz bibliotek Intel TBB(*ang.* Threading Building Blocks) i OpenMP do budowy aplikacji wielowątkowych.

3.4.1. Wątki natywne

Użycie wątków natywnych wymaga wykorzystania bibliotek systemowych – w przypadku aplikacji w języku C++ działającej w systemie PetaLinux zastosować można biblioteki `<thread>` lub `<pthread.h>` wchodzące w skład bibliotek standardowych [31]. Ich wykorzystanie pozwala na możliwie najbardziej efektywne użycie zasobów maszyny obliczeniowej. Wymaga to jednak dużych umiejętności programisty oraz dobrej znajomości architektury docelowej oraz wykonywanego zadania. Ponadto, zastosowanie biblioteki `<pthread.h>` wymaga zgodności systemu docelowego ze standardem POSIX, natomiast w przypadku `<thread>`, konieczne jest przeprowadzenie procesu komplikacji kompilatorem zgodnym ze standardem C++11. Założenia te mogą okazać się problematyczne w przypadku konieczności migracji aplikacji na system nie spełniający opisanych wymagań.

Stosowanie wątków natywnych pozwala na budowę wielowątkowych aplikacji działających heterogenicznie. Jest to najprostszy sposób na zbudowanie programu, w którym kilka wątków odpowiada za kilka różnych zadań. Na przykład, jeden wątek może być odpowiedzialny za przeprowadzenie obliczeń algorytmicznych, drugi za obsługę interfejsu użytkownika i przygotowanie danych wejściowych do właściwych obliczeń algorytmicznych, a kolejny – za niekrytyczne operacje po zakończeniu pracy algorytmu, takie jak przesłanie wyników do bazy danych.

3.4.2. Biblioteka Intel Threading Building Blocks

Biblioteka Intel Threading Building Blocks stanowi zbiór narzędzi rozszerzających standard języka C++ o elementy związane z obliczeniami równoległymi. Składają się na to implementacje algorytmów równoległych, struktury danych przeznaczone do wykorzystania w systemach wielowątkowych oraz implementacje operacji atomowych i algorytmów wzajemnego wykluczania [32].

Użycie biblioteki opiera się na zastosowaniu jej elementów na etapie powstawania aplikacji. Z tego względu, podobnie jak w przypadku wątków natywnych, konieczne jest zaprojektowanie

aplikacji w sposób możliwie najlepiej wykorzystujący zalety biblioteki. Refaktoryzacja kodu istniejącego programu w taki sposób, by zastosować *TBB* może być utrudniona i ostatecznie nie pozwolić na uzyskanie zadowalających wyników.

Główna zaleta stosowania *TBB* jest większa skalowalność wynikowych rozwiązań. W przypadku zastosowania wątków natywnych, konieczne jest zaprojektowanie aplikacji w sposób umożliwiający wykorzystanie innej liczby wątków, gdy będzie to konieczne. W przypadku zastosowania dodatkowej biblioteki, stanowi ona warstwę abstrakcji pomiędzy programistą a warstwą obliczeniową, dzięki czemu proces możliwie najlepszej integracji aplikacji z platformą docelową odbywać się może przy niewielkiej interakcji ze strony projektanta. Biblioteka *TBB* zgodna jest z ideą programowania generycznego, paradygmatu powszechnie stosowanego w aplikacjach projektowanych w języku C++. Jej stosowanie stanowi naturalne rozszerzenie możliwości tego języka i nie wymaga szerokiej wiedzy na temat architektury systemu docelowego.

3.4.3. Biblioteka OpenMP

Biblioteka OpenMP to interfejs programowania aplikacji pozwalający na budowanie wieloplatformowych programów wykonywanych równolegle. Rozwiązanie to jest dedykowane aplikacjom powstającym w językach C i C++ [33]. OpenMP składa się z dyrektyw kompilatora i zbioru bibliotek, które pozwalają kształtować zachowanie programu na etapie wykonania. Ze względu na wykorzystanie dyrektyw kompilacji, możliwa jest integracja biblioteki z istniejącą aplikacją, nie wymagając przy tym modyfikowania właściwego kodu programu. Wymaga to wyłącznie znajomości aplikacji w stopniu umożliwiającym identyfikację obszarów, których równoległe wykonanie pozwoli na osiągnięcie największych zysków, obserwowanych w formie przyśpieszenia działania programu.

Analogicznie jak w przypadku biblioteki *TBB*, zastosowanie OpenMP ma na celu zapewnienie skalowalności aplikacji i dodanie warstwy abstrakcji pomiędzy kod programu, a operacje wykonywane na wątkach obliczeniowych. Obie biblioteki wyposażone są również w algorytmy równoważenia obciążenia.

Wykorzystanie biblioteki OpenMP wymaga zastosowania kompilatora wspierającego dyrektywy wchodzące w jej skład. Ze względu na specyfikę stosowania części interfejsu biblioteki – w formie komentarzy do właściwego kodu aplikacji – możliwa jest komplikacja programu kompilatorem nie wspierającym jej. Wynikowa aplikacja nie będzie korzystać z zalet przetwarzania równoległego, jednak powinna pozwalać na uzyskanie poprawnych wyników algorytmu.

Podsumowanie

Trzy opisane rozwiązania zapewniają dostęp do różnych możliwości i obarczone są różnym kosztem stosowania. Z tego względu, nie jest możliwy jednoznaczny wybór najlepszego rozwiązania dla aplikacji wykonujących obliczenia równoległe. Często słuszne może okazać się wykorzystanie więcej niż jednej biblioteki w aplikacji, wykorzystując ją do nadzoru nad zadaniami różnego typu. Część rozwiązań wymaga wsparcia kompilatora, część ograniczona jest do pewnej grupy systemów lub architektur. Wybór podejścia do obliczeń równoległych powinien stanowić etap projektowania aplikacji, a decyzja powinna uwzględniać szereg zagadnień.

W ramach pracy nie omówiono kilku innych popularnych rozwiązań związanych z obliczeniami wielowątkowymi, w tym *OpenCL* oraz *MPI*, ze względu na ograniczone możliwości ich wykorzystania w systemie PetaLinux działającym na platformie ZYBO. Badane zagadnienie stanowi temat obszernych dyskusji, których wyniki odnaleźć można w publikowanych pracach [34, 35]. W rozdziale 5.5 opisano kroki wymagane do zastosowania omawianych bibliotek w aplikacjach uruchamianych w systemie PetaLinux.

3.5. OpenCV

Algorytmy wizyjne znajdują zastosowanie w wielu aplikacjach realizowanych w ramach projektów związanych z szerokim spektrum dziedzin techniki. Ze względu na swoją popularność, część rozwiązań została zintegrowana w zbiorze bibliotek OpenCV [17]. Zbiór ten zawiera wydajne implementacje najczęściej stosowanych algorytmów, dedykowanych uruchamianiu na układach CPU oraz GPU (*ang. Graphics Processing Unit*). Biblioteka jest szeroko stosowana ze względu na wysoką wydajność, stabilność działania i możliwość przenoszenia rozwiązań pomiędzy platformami o różnych architekturach.

Zastosowanie zewnętrznej biblioteki pozwala, w porównaniu do autorskiej implementacji algorytmu, na ograniczenie czasu wymaganego do zbudowania działającego prototypu. Ponadto, gotowe rozwiązania zapewniają zwykle większą stabilność i wystarczającą wydajność w większości przypadków. Kluczowym ograniczeniem względem autorskiej implementacji jest brak możliwości dostosowania algorytmu do rozpatrywanego przypadku. Może to wiązać się z koniecznością sprowadzenia danych wejściowych do struktury wspieranej przez bibliotekę, co może prowadzić do ograniczenia wydajności działania aplikacji. Wśród algorytmów dostępnych w bibliotece znajdują się procedury detekcji i rozpoznawania twarzy, klasyfikacji zachowań, śledzenia obiektów, czy identyfikacji obrazów podobnych.

Wykorzystanie biblioteki OpenCV w aplikacjach *bare-metal* może być niemożliwe ze względu na liczbę zależności, które należy dostarczyć do poprawnego jej działania. Dostępność systemu operacyjnego, takiego jak PetaLinux, pozwala na dołączenie do systemu plików bibliotek, wybierając je z puli prekompilowanych zasobów, dostępnych w pakiecie PetaLinux lub dostarczając

zewnętrzny zbiór bibliotek, przygotowany przez użytkownika. Możliwości te zbadano w pracy na przykładzie biblioteki OpenCV w wersjach 2.4 oraz 3.1.

Pakiet PetaLinux oferuje dostęp do prekompilowanej biblioteki w wersji 3.1. Wydanie to wciąż znajduje się w początkowej fazie rozwoju i nie zostało w pełni zaadaptowane w części środowisk. Z tego powodu, wersja 2.4 biblioteki wciąż znajduje zastosowanie. OpenCV w wersji 2.4 nie jest jednak oficjalnie dostępne w środowisku PetaLinux. Jego użycie wymaga zbudowania biblioteki na bazie kodu źródłowego wraz z zależnościami i dodanie plików wynikowych do zbioru bibliotek dostępnych w projekcie PetaLinux. Przebieg procesu dla obu przypadków opisano w rozdziale 5.7.

3.6. Integracja algorytmów w FPGA i CPU

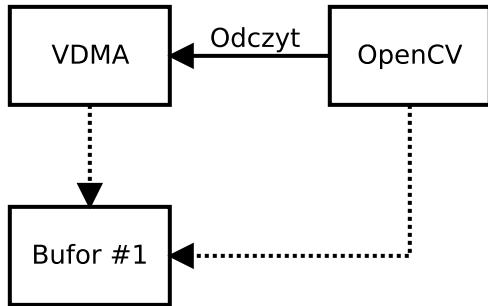
Zagadnienie integracji rozwiązań algorytmicznych wewnętrz logiki programowalnej z obliczeniami prowadzonymi przez procesor ARM może pozwolić na zwiększenie wydajności działania aplikacji względem algorytmów realizowanych wyłącznie w jednym z tych układów. Podział algorytmu na sekwencję etapów, realizowanych w logice reprogramowalnej lub w systemie procesorowym pozwala wykorzystać zalety obu rozwiązań. Układy FPGA pozwalają na użycie zbioru algorytmów w technice strumieniowej, co umożliwia dokonać przetwarzania obrazów w czasie rzeczywistym, z zachowaniem przyjętych opóźnień.

Implementacja części algorytmów w sposób umożliwiający przetwarzanie strumieniowe może być niemożliwa. Procedury wymagające dużego kontekstu na etapie wykonania lub dostępu do danych historycznych projektowane są w celu uruchamiania na układach CPU o swobodnym dostępie do pamięci operacyjnej. Ponadto, realizacja algorytmów zdominowanych przez instrukcje lub wymagających zastosowania wyrażeń warunkowych i pętli może być nieefektywna w porównaniu do implementacji na CPU. Przykładami takich procedur mogą być śledzenie zmiennej liczby obiektów w kadrze czy stosunkowo proste operacje rotacji lub odbicia obrazu.

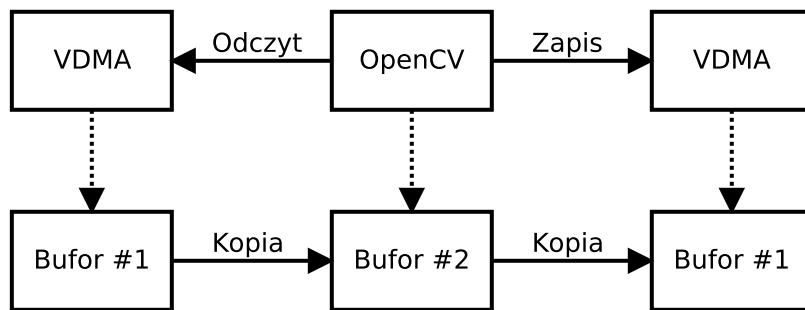
Podział algorytmu wizyjnego na etapy wykonywane naprzemiennie w logice programowalnej i procesorze ARM wymaga transmisji danych pomiędzy dwoma modułami. Realizacja tego zadania w obu kierunkach może być dokonana przy użyciu elementów AXI VDMA. W ramach pracy, zbadano możliwość wykorzystania modułu VDMA do transmisji danych wizyjnych z poziomu FPGA do procesora ARM, w celu ich odczytu i prezentacji z wykorzystaniem komunikacji sieciowej. Zbadano również możliwość konwersji danych do struktur dostępnych w bibliotece OpenCV, co umożliwia wykorzystanie biblioteki do realizacji algorytmów wizyjnych. Procedury odczytu oraz zapisu danych przedstawiono w dodatku C.

Sprawdzono szybkość działania procedur transmisji danych pomiędzy dwoma elementami obliczeniowymi dla sygnału wizyjnego o rozdzielczości HD (1280×720 pikseli), przyjmując rozmiar piksela równy trzydziestu dwóm bitom w cyklu dziesięciu tysięcy iteracji odczytu i zapisu.

Wyniki zebrano w tabeli 3.2. Schemat komunikacji i wykorzystania pamięci w obu przypadkach przedstawiono na rysunkach 3.3 i 3.4.



Rys. 3.3. Wymiana danych bez użycia procedur kopowania.



Rys. 3.4. Wymiana danych z wykorzystaniem procedur kopowania.

Tabela 3.2. Wyniki testu wydajnościowego transmisji danych.

	Bez kopiowania		Z kopiowaniem	
	Łącznie [s]	Średnio [s]	Łącznie [s]	Średnio [s]
Odczyt	0,13	0,000013	151,34	0,015
Zapis	—	—	121,56	0,012

Biblioteka OpenCV pozwala na użycie zewnętrznych źródeł danych do budowy struktur obrazu, a w konsekwencji pominąć etap kopiowania pamięci udostępnianej przez sterownik VDMA. W takim przypadku, czas odczytu pełnej ramki obrazu nie przekracza kilkunastu mikrosekund. Należy jednak pamiętać, że stosując tę technikę, zagwarantować trzeba, że przetwarzanie obrazu zakończy się do chwili rozpoczęcia odczytu danych z wykorzystywanego bufora przez moduł VDMA. W przypadku zastosowania układu trzech buforów i sygnału wejściowego o częstotliwości sześćdziesięciu klatek na sekundę, daje to okno przetwarzania o długości szesnastu milisekund. Przekroczenie tej wartości może wpłynąć na błędą pracę algorytmu ze względu na nadpisanie danych przez moduł VDMA.

Z tego względu, rozważyć należy skopiowane danych wejściowych do nowego bufora na etapie odczytu oraz ponowne skopiowanie danych do bufora źródłowego w trakcie zapisu. Procedura odczytu wymaga średnio piętnastu milisekund, natomiast zapis zajmuje nie więcej niż dwanaście

milisekund. Zastosowanie tej techniki nie pozwala na przetwarzanie sygnału wizyjnego w czasie rzeczywistym, gwarantuje jednak nienaruszalność danych. W połączeniu z zastosowaniem modułu VDMA o większej liczbie buforów, pozwala na integrację algorytmu o dużej złożoności obliczeniowej pomiędzy elementami obliczeniowymi obu typów. Jednakże, w celu zachowania limitów czasowych, konieczne może być ograniczenie przetwarzania danych do części klatek obrazu i pominięcie pozostałych. Nie powinno to jednak wpłynąć na zachowanie algorytmu wewnątrz logiki programowalnej.

3.7. Przerwania systemowe

Przerwanie to sygnał wysyłany przez urządzenie lub program, które ma na celu przekazanie do procesora informacji o zdarzeniu, które wymaga natychmiastowej obsługi. Przerwania podzielić można na maskowalne i niemaskowalne. Klasa przerwań maskowalnych może być zdefiniowana jako ignorowana przez właściwe ustawienie rejestrów kontrolera przerwań. Druga klasa przerwań związana jest zwykle ze zdarzeniami o krytycznym znaczeniu, tak jak zdarzenia związane z działaniem zegarów czy układu *watchdog*¹, więc ich wystąpienie nie może zostać zignorowane przez układ procesora.

Wykorzystanie przerwań pozwala na zaprojektowanie interfejsu aplikacji współpracującej ze zbiorem urządzeń peryferyjnych, takich jak czujniki, przyciski, czy klawiatury. W przypadku układu Zynq, rolę urządzeń wysyłających sygnał przerwania przyjąć mogą również elementy logiki programowalnej, takie jak układy VDMA, zegary, czy moduły zaprojektowane przez użytkownika.

Układ Zynq wyposażony jest w moduł GIC (*ang. Generic Interrupt Controller*), który pełni rolę kontrolera przerwań, odpowiedzialnego za obsługę zdarzeń. *GIC* obsługuje przerwania z kilku źródeł:

- przerwania programowe – zbiór nie więcej niż szesnastu zdarzeń, które pozwalały na wywołanie procedury obsługi przerwania bezpośrednio z kodu aplikacji. Zachowanie to pozwala na komunikację z systemem operacyjnym i jest zwykle wykorzystywane do wywoływania operacji wejścia lub wyjścia.

Innym zastosowaniem przerwań programowych jest wysłanie sygnału *yield*, który pozwala na dobrowolne wywłaszczenie obecnie aktywnego procesu przez układ planisty systemowego².

¹Moduł odpowiedzialny za wykrycie błędного działania systemu i wymuszający restart procesora w takiej sytuacji.

²Proces systemowy odpowiedzialny za wybór procesów do wykonania przez procesor w danej chwili. Wybrane zadania wykonywane są przez jednostkę przez ograniczony czas, po którym mogą zostać zatrzymane na rzecz innych oczekiwanych procesów.

- przerwania systemowe – współdzielone i prywatne – zdarzenia zgłasiane przez sprzęt, na przykład klawiatury, moduły DMA czy układy pamięci. Dla każdego urządzenia lub zbioru zdefiniowana jest unikalna linia przerwania, a do kategorii zdarzenia przypisany jest identyfikator. Obsługa przerwania przez system operacyjny polega na wywołaniu agenta zdarzeń związanego z wyemitowanym identyfikatorem.

Przerwania współdzielone pozwalają na komunikację pomiędzy procesorem a urządzeniami peryferyjnymi oraz układem FPGA, mogą być definiowane przez użytkownika i być obsługiwane przez dowolny rdzeń procesora. Zdarzenia prywatne to przerwania definiowane niezależnie dla każdego rdzenia i pozwalają na obsługę zdarzeń zegarowych czy watchdoga.

W trakcie obsługi przerwania z dowolnego źródła, wykonanie kodu aplikacji jest wstrzymany na czas wywołania kodu agenta obsługi zdarzenia. Stan rejestrów zapisywany jest na stosie i wykonywany jest kod odpowiedzialny za obsługę przerwania. Po zakończeniu wykonania, deklarowanego przez wywołanie instrukcji procesora *RETI*, przywracany jest zapamiętany stan aplikacji i wznowiane jest jej wykonanie.

Przerwania to zdarzenia wywoływane asynchronicznie do normalnego działania aplikacji i mogące pochodzić z wielu źródeł, co może prowadzić do sytuacji, w której kilka linii przerwań sygnalizuje stan alarmu jednocześnie, co może powodować problemy z właściwą obsługą zdarzeń. Procesor ARM pozwala na priorytetyzowanie przerwań. Przyznanie pierwszeństwa pewnemu zbiorowi krytycznych zdarzeń umożliwia rozwiązywanie problemu kolejności wykonania w przypadku, gdy dwa zdarzenia o różnym priorytecie zostaną zgłoszone w tym samym czasie, a także na przerwanie obsługi przerwania o niskim priorytecie na rzecz wywołania agenta krytycznego zdarzenia. Ze względu na charakter przerwań, część zdarzeń może mieć krytyczne znaczenie dla poprawnego wykonania aplikacji, a brak ich obsługi może prowadzić do zatrzymania działania procesora lub przeprowadzenia procesu restartu.

Procedury obsługi przerwań mogą być definiowane zarówno w aplikacjach typu *bare-metal* jak i w systemach operacyjnych, takich jak PetaLinux. Obsługa przerwań w aplikacji *bare-metal* wymaga wykorzystania kontrolera do zarejestrowania agenta zdarzeń w formie funkcji aplikacji.

W przypadku systemu operacyjnego, konieczne jest wykorzystanie sterownika sprzętowego. W skład PetaLinux wchodzą sterowniki dedykowane zbiorowi modułów definiowanych w logice reprogramowalnej. Obsługa zdarzenia w niestandardowym module, zaprojektowanym przez użytkownika, wymagać może napisania dedykowanego sterownika urządzenia. Zagadnienie to wykracza poza zakres niniejszej pracy i stanowi rozbudowany proces, wymagający szerokiej wiedzy na temat działania systemów operacyjnych. Temat ten omawiany jest w pracach [36, 37].

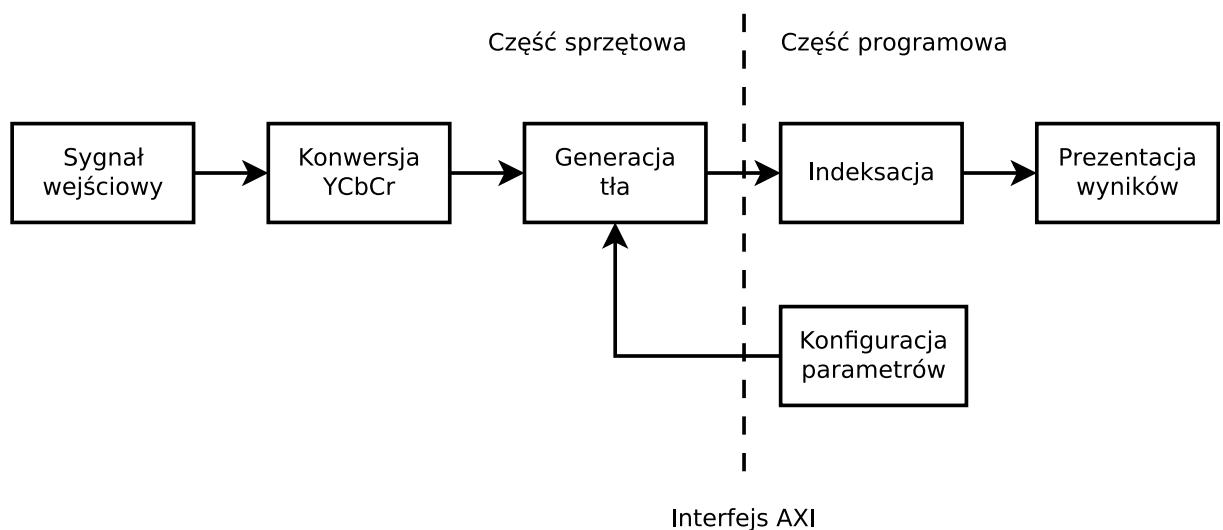
W ramach pracy, zbadano możliwość wykorzystania przerwania emitowanego przez moduł AXI Timer, pozwalającego na wykonywanie operacji odliczania czasu oraz przerwań modułu AXI VDMA, pozwalających na przesłanie notyfikacji związanych ze zdarzeniami odczytu lub zapisu kolejnych ramek obrazu na właściwy im kanał.

Moduł AXI Timer wykorzystać można do odliczania czasu pomiędzy kolejnymi wykonaniami procedury, która powinna być wywoływanie cyklicznie, w regularnych odstępach czasu. Działanie modułu AXI Timer może być warunkowane przez sygnały innych modułów logiki. Pozwala to na zaprojektowanie aplikacji, w której pewne zadanie wykonywane jest cyklicznie, pod warunkiem wystąpienia zdefiniowanego zdarzenia na poziomie logiki, a kontekst tego zdarzenia nie musi być znany z poziomu aplikacji wykonywanej przez procesor – na przykład, wykonanie przez aplikację operacji analizy danych uzyskiwanych przez algorytm wizyjny cyklicznie, pod warunkiem, że moduł algorytmu wykonywany przez układ logiczny ustawił sygnał spójności wyników. Podobne zachowanie uzyskać można korzystając z funkcji procesora i związanych z nim zegarów, co ogranicza jednak możliwości konfiguracji działania procedury do kontekstu znanego z poziomu CPU.

Przerwania definiowane przez moduł AXI VDMA pozwalają na zgłoszenie zdarzenia po wykonaniu procesu odczytu lub zapisu określonej liczby ramek obrazu lub po upływie określonego czasu od uzyskania sygnałów synchronizacji obrazu. Wykorzystanie tych mechanizmów pozwala na zaprojektowanie procedur aplikacji, które powinny być wykonane co określoną liczbę klatek sygnału wizyjnego. W szczególnym przypadku, mechanizm ten pozwala na wykonanie przez procesor dla każdej ramki obrazu operacji algorytmicznych, których implementacja sprzętowa byłaby trudna lub niemożliwa. Mechanizm ten pozwala również na przeprowadzenie operacji końcowej analizy wyników algorytmu wizyjnego, zaimplementowanego w logice programowalnej. Proces konfiguracji projektu wykorzystującego mechanizm przerwań opisano w rozdziale 5.8.

4. System wizyjny zrealizowany na platformie Zynq z systemem operacyjnym PetaLinux

W ramach pracy zbadano możliwość realizacji systemów wizyjnych wykorzystujących możliwości obliczeniowej logiki programowalnej oraz procesora ARM i integrujących obie architektury w jednym procesie algorytmicznym. Zaproponowano moduł segmentacji obiektów pierwszoplanowych oparty o tzw. generację i modelowanie tła. Dane wejściowe pochodząły z kamery video połączonej z układem przy użyciu interfejsu HDMI. Sygnał wizyjny poddawany był przetwarzaniu i analizie przez elementy logiki programowalnej, a wyniki przesyłane były przy użyciu mechanizmu AXI VDMA do układu CPU. Proces rozpoznawania miał na celu indeksację obiektów pierwszoplanowych, z wykorzystaniem biblioteki OpenCV i procedury `cv::connectedComponents`. Wykorzystano ponadto mechanizm AXI DMA do konfiguracji parametrów algorytmu logiki programowalnej oraz wykorzystano narzędzia systemu operacyjnego PetaLinux do obsługi tego procesu. Zaproponowano również mechanizm synchronizacji dwóch kolejnych ramek sygnału przy użyciu modułu AXI VDMA. Algorytm i jego podział na część sprzętową i programową przedstawiono na schemacie 4.1.



Rys. 4.1. Schemat algorytmu i podział na część sprzętową i programową.

W przypadku algorytmów przetwarzania sekwencji obrazów, na etapie analizy jednej klatki wykorzystać można informacje uzyskane w trakcie obliczeń dla poprzednich ramek. Rozszerzenie kontekstu o parametry historyczne pozwala na projektowanie bardziej zaawansowanych systemów, zwykle wymaga jednak wykorzystania modułów zewnętrznej pamięci w celu przechowywania danych historycznych.

Wśród algorytmów wymagających kontekstu związanego z więcej niż jedną ramką obrazu wyróżnić można między innymi:

- segmentację obiektów pierwszoplanowych – umożliwia podział obrazu na elementy tła oraz znajdujące się na pierwszym planie, pozwalając zwykle ograniczyć obszar analizy do fragmentów obrazu, z którymi związane są obiekty pierwszoplanowe,
- śledzenie obiektów i analiza zachowań – indeksacja to proces przypisanie etykiet do obiektów i wyznaczenie zbioru niezależnych elementów obrazu. Pozwala to śledzić ruch każdego z obiektów oraz analizę zachowań. Badanie zachowań, na przykładzie przechodniów, może umożliwić detekcję obecności osób w strefach nieuprawnionych czy wykrycie osób o nieokreślonych motywacjach działań i w ten sposób przyczynić się do zwiększenia bezpieczeństwa w przestrzeni publicznej,
- wyliczanie przepływu optycznego – pozwala na analizę ruchu obiektów znajdujących się w kadrze, umożliwiając estymację odległości czy parametrów ruchu.

Implementacja wymienionych typów algorytmów w architekturze potokowej może być utrudniona lub niemożliwa bez użycia zewnętrznego elementu pamięciowego. Jedna ramka obrazu kolorowego o rozdzielcości 1280×720 pikseli ma rozmiar 2,8MB. Układ Zynq dostępny na karcie ZYBO wyposażony jest w bloki pamięci BRAM o łącznej pojemności 2,1 MB. Zatem nawet w najprostszych systemach, wymagających buforowania wyłącznie jednej ramki obrazu nie jest możliwa realizacja tego zadania bez użycia pamięci zewnętrznej.

Ponadto, końcowa analiza wyników algorytmu w architekturze FPGA jest odgórnie ograniczona do przewidywanych parametrów działania systemu, a w efekcie mniej elastyczna – na przykład, zagadnienie śledzenia obiektów może być ograniczone do maksymalnej zadanej liczby niezależnych elementów. Proces analizy wymagać może budowania rozbudowanych maszyn stanu, których realizacja może być podatna na błędy, a dodanie nowych funkcjonalności utrudnione. Ze względu na te ograniczenia, korzystny może okazać się podział algorytmu na niezależne etapy, wykonywane przez moduły sprzętowe zrealizowane w logice programowalnej lub procedury uruchamiane na ARM.

Klasyczny sekwencyjny element obliczeniowy pozwala na adaptację algorytmu do zmieniających się w czasie parametrów obrazu – na przykład śledzenie zmieniającej się liczby obiektów pierwszoplanowych. Umożliwiają to mechanizmy dynamicznej alokacji pamięci czy obsługi wyrażeń warunkowych i pętli, właściwie niedostępne w przypadku implementacji sprzętowych.

Ponadto, użycie systemu operacyjnego pozwala wykorzystać zaawansowane możliwości prezentacji i przechowywania wyników działania algorytmu wizyjnego, na przykład prezentację danych wyjściowych przy użyciu interfejsu sieciowego lub zapis do bazy danych. W poniższym rozdziale zaproponowano metody wykorzystania platformy Zynq na przykładzie wybranych elementów systemów wizyjnych.

4.1. Moduł wyznaczania różnicy kolejnych ramek w sekwencji obrazów

Wyznaczenie różnicy pomiędzy dwoma kolejnymi ramkami strumienia wizyjnego wymaga obliczenia dla każdego piksela wartości różnicy, opisanej formułą (4.1).

$$d^i(x, y) = |p^i(x, y) - p^{i-1}(x, y)| \quad (4.1)$$

gdzie:

x, y – współrzędne piksela,

i – indeks ramki w sekwencji obrazów,

$p^i(x, y)$ – wartość w i -tej ramce dla piksela o współrzędnych (x, y) ,

$d^i(x, y)$ – wyznaczana wartość różnicy.

W omawianym przypadku, sygnał źródłowy i wynikowy mają postać obrazu przedstawionego w odcieniach szarości. Zastosować można również inne metryki odległości pomiędzy pikselami, na przykład metrykę euklidesową, opisaną formułą (4.2), która w przypadku sygnałów o jednym kanale jest równoznaczna formule (4.1), ale może być również zastosowana dla obrazów kolorowych.

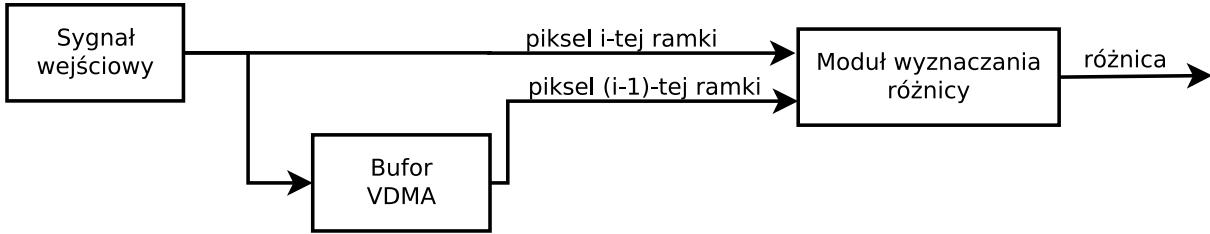
$$d_e^i(x, y) = \sqrt{(p^i(x, y) - p^{i-1}(x, y))^2} \quad (4.2)$$

Wyznaczanie różnicy obrazów składających się z więcej niż jednego kanału jest możliwe, jednak interpretacja graficzna wyników może być nieczytelna. Obraz kolorowy poddać można redukcji do jednego kanału przed wykonaniem kroku wyznaczania różnicy. Zagadnienie obliczania różnicy dwóch kolejnych obrazów w sekwencji może stanowić przykład algorytmu, którego realizacja w systemach potokowych, pomimo niskiej złożoności obliczeniowej, może być utrudniona, ze względu na konieczność wykorzystania zewnętrznego modułu pamięci W praktycznych realizacjach, konieczne jest wykorzystanie modułów pamięci operacyjnej w celu zapamiętania poprzedniej ramki obrazu.

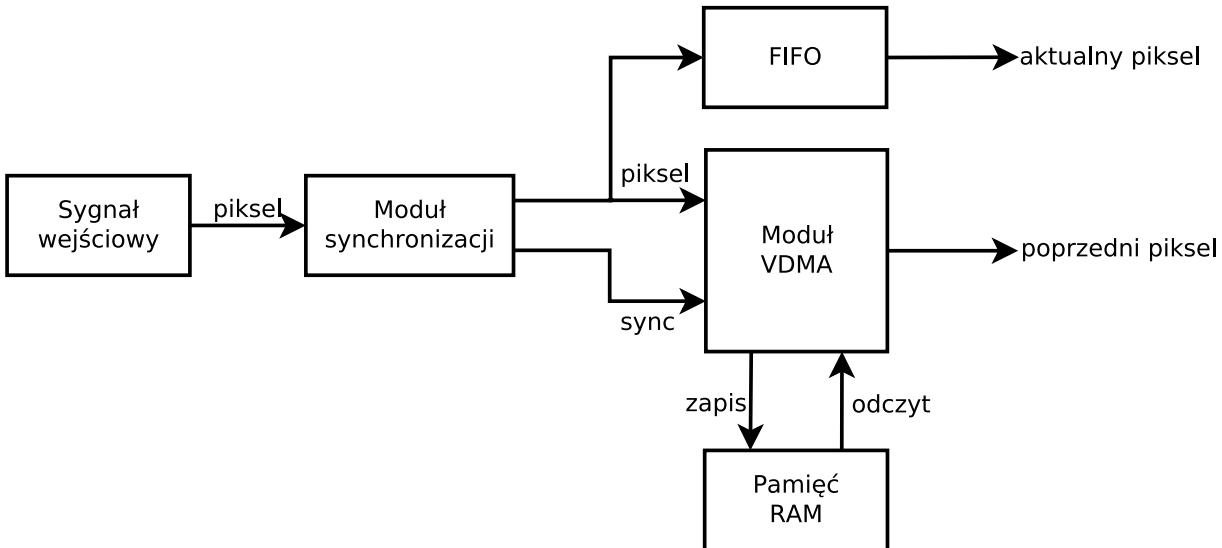
Architekturę strumieniową realizującą omawiane zadanie przedstawiono na schemacie 4.2.

Wykorzystano moduł AXI VDMA w roli bufora sygnału, opóźniającego dane o pełen cykl strumieniowania ramki obrazu. Schemat elementu buforującego przedstawiono na schemacie 4.3.

Realizacja bufora wymagała zaprojektowania mechanizmu synchronizacji dwóch niezależnych klatek sygnału wizyjnego. W tym celu wykorzystano moduł kolejki FIFO dla protokołu



Rys. 4.2. Schemat architektury obliczającej różnicę sekwencji obrazów.



Rys. 4.3. Schemat elementu buforującego ramkę obrazu.

AXI4-Stream oraz dedykowany element synchronizujący kanał odczytu z bufora VDMA z sygnałem rozpoczęcia nowej ramki obrazu strumienia wejściowego.

Założono, że algorytm będzie wykorzystywany w systemach wizyjnych czasu rzeczywistego, działających w architekturze potokowej. Aplikację przystosowano do działania z sygnałem wizyjnym o dowolnej rozdzielczości i próbkowaniu, składającym się z jednego lub wielu kanałów obrazu. Zastosowano kolejkę FIFO o długości 128 elementów oraz linie buforujące związane z modelem VDMA o tej samej długości.

W celu weryfikacji działania elementu wyznaczającego różnicę sekwencji obrazów zaprojektowano strukturę rozszerzoną o elementy umożliwiające komunikację przy użyciu protokołu AXI oraz przepływ sygnału wizyjnego. Zaprojektowano aplikację umożliwiającą konfigurację modułu w trybie *bare-metal* oraz przy współpracy systemu PetaLinux.

Sprawdzono działanie aplikacji dla sygnału wizyjnego o rozdzielczości 1280×720 pikseli i częstotliwości sześćdziesięciu ramek na sekundę. Szacowane zapotrzebowanie wynikowego systemu na energię elektryczną nie powinno przekroczyć $1,86W$. Właściwa energia wymagana do prowadzenia operacji obliczeniowych nie przekracza wartości $1,723W$, w tym $1,559W$ (90%) to energia wymagana do obsługi układu ARM.

Tabela 4.1. Wykorzystanie zasobów przez aplikację wyznaczającą różnicę kolejnych ramek obrazu.

Rodzaj zasobu	Użycie	Dostępne	Procent użycia
FF	3059	17600	17,38%
LUT 6	5721	17600	32,51%
SLICE	2550	4400	57,95%
DSP 48	0	80	0%
BRAM	6	60	10%

W tabeli 4.1 przedstawiono zapotrzebowanie na zasoby FPGA układu Zynq. Moduł przeznaczony jest do pracy z częstotliwością $200MHz$, co pozwala na analizę sygnału wideo o rozdzielcości 1920×1080 pikseli i częstotliwości obrazu $60Hz$. Proces konfiguracji modułu VDMA wykorzystywanego do buforowania ramki obrazu przedstawiono w rozdziale 5.4. Wynik działania modelu programowego zaprezentowano na rysunku 4.4. Projekty Vivado i PetaLinux związane z omawianą aplikacją dodano jako załączniki do pracy oraz udostępniono w repozytorium [38].



Rys. 4.4. Ramka obrazu i wyznaczona różnica ramek.

4.2. Moduł generacji tła

Poza najprostszymi przypadkami analizy ruchu, informacja uzyskana w wyniku odejmowania ramek sekwencji wizyjnej nie jest wystarczająca do analizy strumienia obrazów. Zagadnienie to może być jednak elementem składowym bardziej rozbudowanych algorytmów, na przykład modułów realizujących algorytm generacji i modelowania tła, czy segmentacji obiektów pierwszoplanowych, gdzie pozwala określić zbiór obiektów będących w ruchu.

Generacja tła to zadanie ekstrakcji elementów *tła* badanego obrazu, a więc takich, które stanowią stały, niezmienny element sceny. Dzięki wydzieleniu obiektów tła, pozostałe elementy

obrazu klasyfikowane są jako obiekty pierwszoplanowe. Zwykle, uważa się za nie elementy będące w ruchu. Uwzględnić jednak należy również obiekty, których ruch jest niejednostajny, na przykład zatrzymujący się piesi lub pojazdy na skrzyżowaniu.

Bardziej zaawansowane metody generacji tła uwzględniają ponadto dodatkowe warunki klasyfikacji obiektów do dwóch z omawianych grup:

- cienie – choć mogą być związane zarówno z elementami tła jak i pierwszoplanowymi, oceniane jest zwykle, by nie były uwzględniane w grupie obiektów wymagających analizy,
- ruchome elementy tła – występujące na przykład pod wpływem wiatru ruchy roślin czy deszcz nie powinny być traktowane jako obiekty pierwszoplanowe,
- obiekty o niejednostajnym ruchu – algorytm powinien klasyfikować poprawnie obiekty pierwszoplanowe, które pojawiają się na scenie a następnie zatrzymują, nie traktując ich jako elementy tła,
- obiekty wzajemnie przesłaniające się – elementy pierwszego planu mogą, w wyniku ruchu, zostać zasłonięte z perspektywy kamery przez elementy tła. Nie powinno to wpływać na zmianę klasyfikacji obiektów z obu grup.
- warunki oświetleniowe – możliwość zmiany warunków oświetleniowych może wymagać ciągłej korekty parametrów generowanego modelu tła. Uwzględnić należy zarówno zmiany długookresowe, wynikające na przykład z cyklu dobowego, jak i krótkookresowe, wynikające z nagłych zmian, takich jak włączenie lub wyłączenie sztucznego oświetlenia sceny.

Zagadnienie modelowania tła nie jest trywialne i wymaga metod uwzględniających część lub wszystkie z wymienionych powyżej ograniczeń. Opracowanie dostępnej literatury poruszającej ten temat znaleźć można w pracy [39]. W ramach niniejszego opracowania zdecydowano się na realizację generacji tła przy pomocy metody średniej bieżącej, opisanej zależnością (4.3). Wartość modelu tła wyznaczana jest niezależnie dla każdej składowej obrazu.

$$b^i(x, y) = \alpha p^i(x, y) + (1 - \alpha)b^{i-1}(x, y) \quad (4.3)$$

gdzie:

x, y – współrzędne piksela,

i – indeks ramki w sekwencji obrazów,

$p^i(x, y)$ – wartość w i -tej ramce dla piksela o współrzędnych (x, y) ,

$b^i(x, y)$ – wartość w i -tej ramce dla piksela modelu tła o współrzędnych (x, y) ,

α – współczynnik bezwładności tła z przedziału $(0, 1]$.

Wadą przedstawionej metody jest jej wrażliwość na krótkookresowe zmiany oświetlenia. Jednym ze sposobów eliminacji zakłóceń pojawiających się cyklicznie – na przykład drgań liści

pod wpływem wiatru – jest wykorzystanie wielu niezależnie wyznaczanych modeli tła. Stosując kilka modeli, budować można warianty dopasowane do najczęściej występujących przypadków, uporządkowanych według prawdopodobieństwa wystąpienia. Przy takim podejściu, obliczenia prowadzone są dla każdego modelu tła niezależnie, wartość nie jest jednak aktualizowana w sytuacji, gdy stan piksela nie jest zbliżony do oczekiwanej, związanego z wybranym modelem. Jednym z dostępnych rozwiązań jest algorytm GMM (*ang.* Gaussian Mixture Model), wykorzystujący kilka rozkładów prawdopodobieństwa Gaussa. W niniejszej pracy nie zdecydowano się na realizację opisanej powyżej metody eliminacji zakłóceń, uzasadniając to zachowaniem czytelności i prostoty implementacji.

Algorytm dostosowano do pracy z sygnałem opisanym w przestrzeni barw *YCbCr*. Procedura generacji tła odbywa się niezależnie dla każdej składowej sygnału. Przyjęto, że aktualizacja wartości modelu tła powinna mieć miejsce wyłącznie w przypadku, jeśli aktualnie badany piksel może być uznany za element tła. W tym celu wprowadzono dwa warunki wykonania obliczeń:

1. Warunek ruchu.

Aktualizacja powinna mieć miejsce wyłącznie w przypadku, jeśli wartość piksela nie uległa zmianie większej niż dopuszczalna względem poprzedniej ramki obrazu. W przeciwnym razie przyjąć można, że nastąpił ruch elementu i piksel nie należy do tła. Zależność opisano wzorem (4.4).

$$d_Y^i(x, y) > T_{fd} \quad (4.4)$$

gdzie:

$d_Y^i(x, y)$ – różnica ramek dla kanału Y obrazu, opisana wzorem (4.1),
 T_{fd} – próg ruchu z zakresu $[0, 255]$, zwykle nie przekraczający 30.

Większe wartości współczynnika T_{fd} pozwalają dokonać aktualizacji modelu tła dla elementów o coraz większej różnicy względem poprzedniej ramki obrazu.

2. Warunek tła.

Aktualizacja powinna mieć miejsce wyłącznie w przypadku, jeśli piksel został sklasyfikowany jako element tła na bazie aktualnego modelu. Zależność opisano równaniem (4.5).

$$w_Y m_Y^i(x, y) + w_{Cb} m_{Cb}^i(x, y) + w_{Cr} m_{Cr}^i(x, y) > T_{bg} \quad (4.5)$$

gdzie:

$m^i(x, y)$ – zmiana wartości piksela względem tła, opisana zależnością (4.6),
 w_k – współczynnik wagi związany z k -tym kanałem
 T_{bg} – współczynnik bezwładności przynależności do tła z zakresu $[0, 255]$.

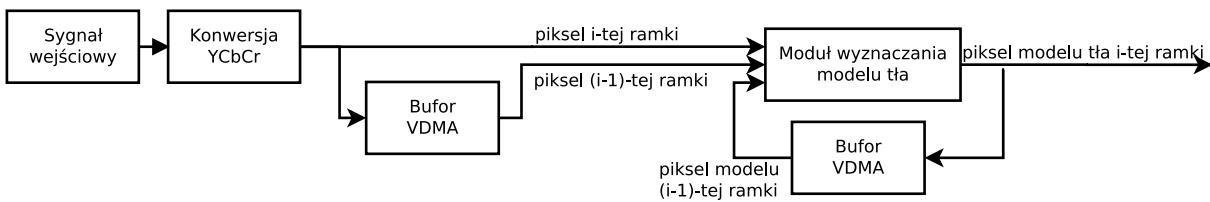
$$m_k^i(x, y) = |p_k^i(x, y) - b_k^{i-1}(x, y)| \quad (4.6)$$

gdzie:

k – indeks składowej barwnej sygnału.

Większe wartości parametru T_{bg} pozwalają na aktualizację modelu tła w sytuacji, gdy różnica piksela względem aktualnego modelu tła jest znaczna. Jego wartość nie przekracza jednak zwykle 30.

W trakcie eksperymentów przyjęto wartości współczynników wag przynależności do tła odpowiednio: $w_Y = 1, w_{Cb} = 2, w_{Cr} = 2$. Aktualizacja modelu tła powinna mieć miejsce wyłącznie w sytuacji, gdy spełnione są oba warunki przedstawione powyżej. Schemat blokowy algorytmu przedstawiono na rysunku 4.5.



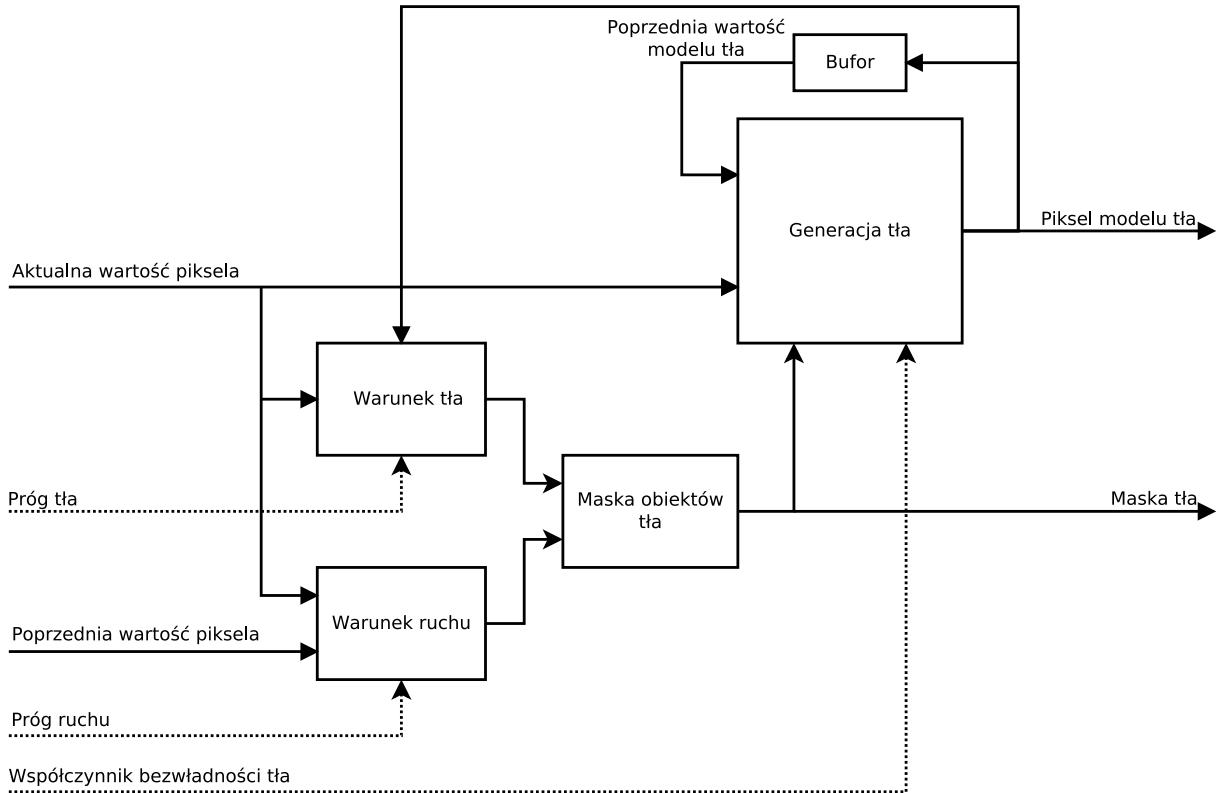
Rys. 4.5. Schemat architektury wyliczającej model tła.

Działanie modułu generacji tła przedstawiono na schemacie 4.6. Na wejściu, moduł otrzymuje aktualną wartość piksela w przestrzeni barw $YCbCr$ oraz poprzednią wartość piksela i modelu tła. Ponadto, przekazywane są parametry opisujące działanie algorytmu: α , T_{bg} oraz T_{fd} . Moduł odpowiedzialny jest za wyznaczenie maski obiektów tła oraz nowej wartości modelu, które stanowią wyjścia elementu obliczeniowego.

Algorytm wymaga wykorzystania dwóch buforów AXI VDMA. Jeden z nich przeznaczony jest do buforowania ramki obrazu wejściowego, natomiast drugi przechowuje aktualny model tła. Alternatywą jest zastosowanie wspólnego bufora i przechowywanie w nim dwóch scalonych sygnałów. Rozwiązanie to pozwala ograniczyć zapotrzebowanie na zasoby logiczne, może jednak wiązać się z trudnościami w synchronizacji wielu strumieni wizyjnych.

Algorytm zintegrowano z układem umożliwiającym komunikację z procesorem ARM, co pozwala na transmisję uzyskanego modelu tła i jego dalszą analizę. Wykorzystano w tym celu trzeci moduł AXI VDMA. W praktycznych zastosowaniach moduł ten może okazać się zbędny, ze względu na to, że w jednym z pozostałych modułów VDMA przechowywany jest model tła dla poprzedniej klatki obrazu. Opóźnienie jednego cyklu nie powinno wpływać negatywnie na jakość działania aplikacji. Niezależny moduł VDMA pozwala jednak na przesyłanie wyników również w przypadku, gdy algorytm generacji tła nie stanowi ostatniego etapu obliczeń.

Ze względu na duże zapotrzebowanie algorytmu na elementy obliczeniowe logiki reprogramowalnej, zdecydowano się ograniczyć rozmiar kolejek FIFO do 64 elementów. Szacowane zapotrzebowanie wynikowego systemu na energię elektryczną nie powinno przekroczyć $1,936W$.



Rys. 4.6. Algorytm wyznaczania modelu tła.

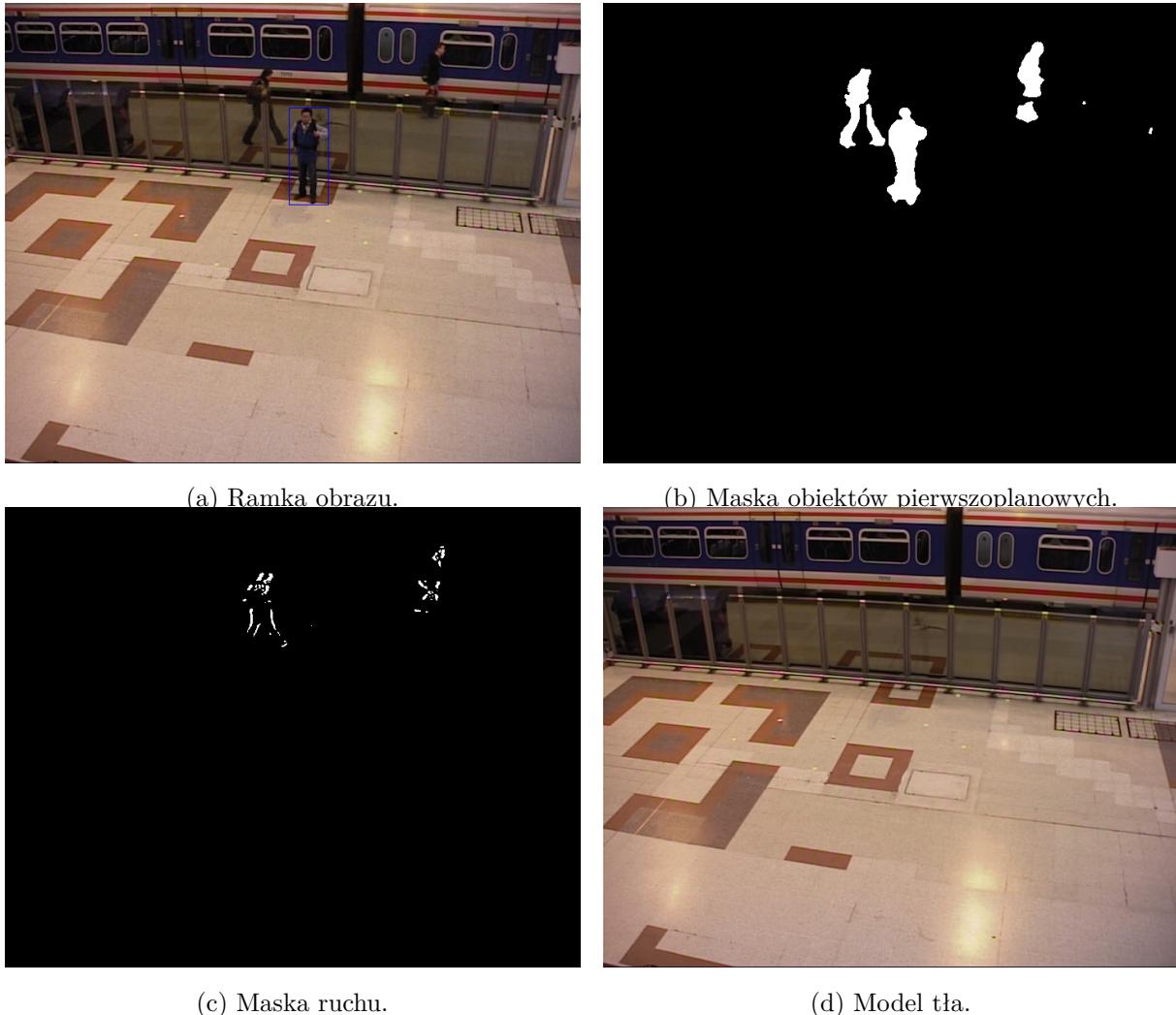
Właściwa energia wymagana do przeprowadzania operacji obliczeniowych nie przekracza wartości $1,797W$, w tym $1,565W$ (87%) to energia wymagana do obsługi układu ARM.

Tabela 4.2. Wykorzystanie zasobów przez aplikację.

Rodzaj zasobu	Użycie	Dostępne	Procent użycia
FF	6938	17600	39,42%
LUT 6	13670	17600	77,67%
SLICE	4400	4400	100%
DSP 48	15	80	18,75%
BRAM	12	60	20%

W tabeli 4.2 przedstawiono zapotrzebowanie na zasoby FPGA układu Zynq. Proporcjonalnie duże zużycie zasobów wynika z konieczności wykorzystania wielu modułów AXI VDMA oraz innych elementów wykorzystujących interfejs AXI. Może ono być ograniczone przez wykorzystanie jednego modułu do obsługi buforowania zarówno ramki obrazu, jak i modelu tła. W przypadku złożonych aplikacji, konieczne może okazać się użycie układu z większą liczbą dostępnych zasobów obliczeniowych. Moduł przeznaczony jest do pracy z częstotliwością $200MHz$, co pozwala na analizę sygnału wideo o rozdzielcości 1920×1080 pikseli i częstotliwości obrazu $60Hz$. Wynik działania modelu programowego przedstawiono na rysunku 4.7. Projekty Vivado

i PetaLinux związane z omawianą aplikacją dodano jako załączniki do pracy oraz udostępniono w repozytorium [38]. Proces konfiguracji aplikacji przedstawiono w rozdziale 5.9.



Rys. 4.7. Ramka obrazu wejściowego oraz wyznaczane przez model programowy obrazy masek i tła.

Dzięki zastosowaniu maski ruchu, obiekty poruszające się nie wpływają na stan modelu tła. Ponadto, dzięki badaniu warunku przynależności do tła, osoba znajdująca się na pierwszym planie i nie będąca w ruchu również nie wpływa negatywnie na wynik działania algorytmu generacji tła.

Napotkane trudności implementacyjne

W trakcie realizacji projektu napotkano na szereg trudności. Środowisko Vivado nie udostępnia narzędzi wymaganych do zaprojektowania aplikacji wizyjnych wykorzystujących mechanizm buforowania pełnych ramek obrazu. W konsekwencji, proces ten wymaga użycia kilku niezależnych elementów manipulujących strumieniami AXI, co prowadzi do skomplikowania projektu i wprowadza opóźnienia. Ponadto, omawiany algorytm wymaga zapewnienia synchronizacji kilku

sygnałów wizyjnych na kolejnych etapach przetwarzania. Biblioteka AXI nie oferuje modułu dedykowanego temu zadaniu, przez co konieczne jest użycie elementów **AXI4-Stream Combiner** oraz **AXI4-Stream Broadcaster**, których działanie wymaga modyfikacji parametrów wykorzystywanych strumieni. Użycie modułów VDMA do buforowania ramek wymaga zapewnienia sygnału, względem którego synchronizowana jest chwila rozpoczęcia transmisji klatki obrazu.

Ze względu na konieczność przeprowadzenia procesu *handshake*¹ przed przesłaniem danych pomiędzy modułami, komunikacja przy użyciu protokołu AXI jest dwukierunkowa. Wystąpienie opóźnień w transmisji lub brak synchronizacji prowadzić może do zakleszczenia (*ang. deadlock*) procesu komunikacji, co może skutkować utratą przesyłanych danych.

Zjawisko to zaobserwowano w trakcie realizacji modułu generacji tła. Ze względu na opóźnienia wymagane do poprawnej pracy algorytmu, komunikacja z modułem VDMA wykorzystywanym do przechowywania modelu tła narażona była na błędy i, w efekcie, prowadziła do niepoprawnego działania całego modułu. Zbadano kilka możliwości alternatywnej realizacji komunikacji pomiędzy modułami, jednak nie osiągnięto efektów gwarantujących poprawność działania całej aplikacji. Opisanych problemów nie zaobserwowano w sytuacji, gdy moduł działał bez opóźnień. Projekt zawierający moduły wprowadzające opóźnienia do komunikacji z użyciem protokołu AXI może wymagać użycia dedykowanego elementu odpowiedzialnego za proces komunikacji. Zadania tego nie zrealizowano w ramach niniejszej pracy.

Ze względu na przedstawione ograniczenia i zaobserwowaną w pracy modułu niestabilność, nie zaprezentowano kompletnego systemu wizyjnego implementującego algorytm generacji tła. Zrealizowane jego składowe – moduły wyznaczania różnicy klatek obrazu, buforowania i synchronizacji dwóch kolejnych ramek, czy moduł generacji tła, a także aplikacja systemowa, pozwalająca na monitorowanie i konfigurację modułów algorytmicznych z poziomu PetaLinux – stanowią mogące przykłady realizacji projektów wykorzystujących funkcjonalności platformy Zynq.

4.3. Integracja z systemem PetaLinux

Równolegle do procesu projektowania elementów logiki programowalnej, rozwijana była aplikacja przeznaczona do pracy pod kontrolą systemu PetaLinux, udostępniająca funkcjonalności monitorowania oraz modyfikowania stanu algorytmu. Na etapie prototypowania, elementy logiki reprogramowalnej kontrolowane były przez aplikację działającą w trybie *bare-metal*, bez wsparcia dla systemu operacyjnego. Po zakończeniu tego etapu, możliwe stało się zaprojektowanie aplikacji działającej pod kontrolą systemu PetaLinux, umożliwiającej wykorzystanie zaawansowanych funkcji systemu. Założono, że projektowana aplikacja powinna spełniać szereg wymagań:

¹Proces wymiany informacji pomiędzy dwoma urządzeniami, który ma na celu ustalenie parametrów transmisji i zadeklarowanie gotowości do komunikacji. W przypadku interfejsu AXI-Stream, wykorzystuje sygnały **valid** oraz **ready** i ma na celu przekazanie informacji o poprawności danych przez moduł *master* oraz o gotowości do ich odczytu przez moduł *slave*.

- Konfiguracja modułów AXI i algorytmu.

Podstawowym zadaniem aplikacji powinno być przeprowadzenie wstępnej konfiguracji modułów, wykorzystując w tym celu interfejs AXI. Proces ten powinien mieć miejsce na etapie uruchamiania aplikacji. Aplikacja powinna być też odpowiedzialna za przeprowadzenie procesu konfiguracji parametrów wykonywanego algorytmu wizyjnego. Ponadto, działanie algorytmu nie powinno zostać przerwane w razie wyłączenia programu.

- Konfiguracja aplikacji przy użyciu argumentów wiersza poleceń.

Konfiguracja parametrów działania aplikacji, w tym rozmiar przetwarzanych obrazów i parametry algorytmu powinny być konfigurowane przy użyciu argumentów wiersza poleceń.

- Monitorowanie działania algorytmu.

Aplikacja powinna udostępniać opcję monitorowania stanu elementów algorytmu, ze szczególnym uwzględnieniem modułów AXI VDMA, odpowiedzialnych za buforowanie danych oraz komunikację z procesorem.

- Zapis wyników pracy algorytmu.

Program powinien być odpowiedzialny za zapis wyników działania algorytmu, na przykład w formie obrazów przechowywanych w pamięci.

- Wykorzystanie komunikacji sieciowej.

Uruchomienie aplikacji nie powinno wymagać fizycznego dostępu do układu Zynq. Docelowym narzędziem komunikacji jest protokół SSH. Ponadto, aplikacja powinna udostępniać interfejs wykorzystujący protokół HTTP, umożliwiający weryfikację stanu aplikacji przy użyciu przeglądarki internetowej.

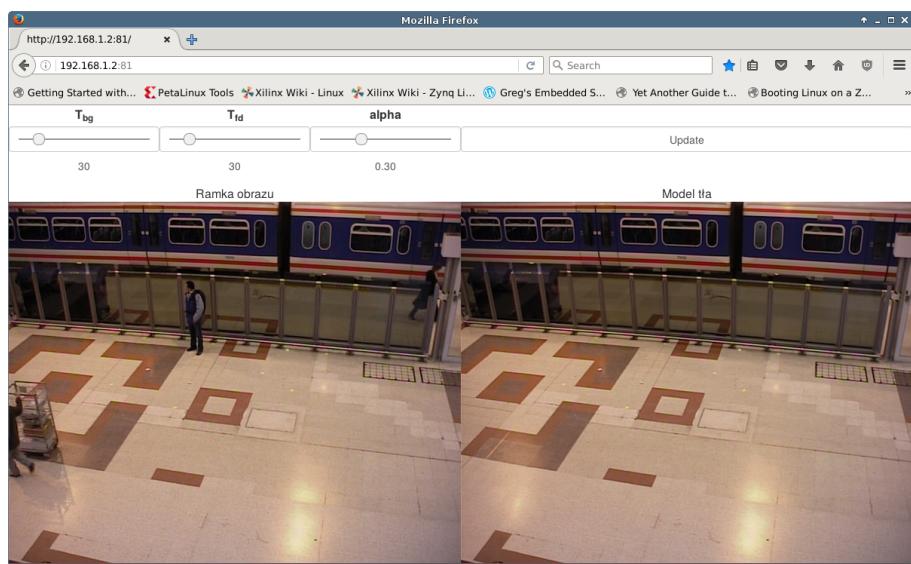
- Kompatybilność z procedurami biblioteki OpenCV

Aplikacja powinna zapewniać zgodność z technikami programowania wykorzystywanyimi przez bibliotekę OpenCV w stopniu umożliwiającym użycie algorytmów biblioteki ze strukturami danych wykorzystywanyimi przez program.

Zaprojektowano aplikację w języku C, spełniającą przedstawione wymagania. Program odpowiedzialny jest za konfigurację elementów logiki reprogramowalnej na podstawie wartości przekazanych przy użyciu argumentów wiersza poleceń. Aplikacja wykonuje operacje monitorowania działania algorytmu i zapisu informacji logu do pliku. Ponadto, umożliwia cykliczny zapis obrazów będących wynikiem działania algorytmu do plików graficznych.

Proces obsługi aplikacji opiera się na wykorzystaniu protokołu SSH, program udostępnia również interfejs HTTP, umożliwiający uzyskanie aktualnych wyników działania algorytmu. Zbadano możliwość wykorzystania aplikacji w roli elementu obliczeniowego, odpowiedzialnego za przeprowadzenie części operacji algorytmicznych. Zaproponowano moduł indeksacji obiektów na bazie generowanego modelu tła. W tym celu wykorzystano procedurę `cv::connectedComponents` dostępną w bibliotece OpenCV.

Ze względu na ograniczenia sprzętowe, moduł indeksacji nie był w stanie spełnić wymagań pracy w czasie rzeczywistym dla sygnału wizyjnego o częstotliwości $60Hz$ i rozdzielczości 1280×720 pikseli – jego wydajność nie przekraczała piętnastu ramek na sekundę. Na rysunku 4.8 przedstawiono widok interfejsu *www* udostępnianego przez aplikację, przedstawiający aktualny stan i wyniki działania programu. Widok wygenerowany został na bazie wyników wyznaczonych przez model programowy. Interfejs wyświetla model tła oraz najnowszą ramkę obrazu, a także pokazuje wartości parametrów algorytmu i umożliwia ich modyfikację w trakcie działania aplikacji. Związane z omawianą aplikacją projektu dodano jako załączniki do pracy oraz udostępniono w repozytorium [38].



Rys. 4.8. Interfejs www aplikacji.

Podsumowanie

Zaproponowane rozwiązania projektowe pozwalają na wykorzystanie części funkcji systemu operacyjnego, które badane były w ramach pracy. Szczególnie istotnym zagadnieniem jest komunikacja pomiędzy elementami zaprojektowanymi w dwóch architekturach. Dzięki wykorzystaniu transmisji danych, możliwe jest zaprojektowanie algorytmów podzielonych na moduły wykonywane naprzemiennie przez obie części układu, wykorzystując atuty obu architektur do możliwie maksymalnego zwiększenia wydajności pełnego algorytmu.

Ponadto, wykorzystanie systemu operacyjnego pozwala na realizację zadań zwykle niemożliwych w przypadku projektów aplikacji realizowanych wyłącznie przy użyciu elementów logiki reprogramowalnej lub sterowanych przez aplikację *bare-metal*. Program działający pod kontrolą systemu operacyjnego umożliwia prowadzenie zadań konfiguracji, kontroli i monitorowania działania aplikacji z wykorzystaniem komunikacji sieciowej.

Wykorzystanie systemu operacyjnego pozwala również na implementację aplikacji w dowolnym języku programowania. Dzięki temu, stosując dedykowane rozwiązania programistyczne, projektowanie aplikacji o rozbudowanych możliwościach zajmuje mniej czasu.

W trakcie realizacji projektu napotkano szereg ograniczeń.

- Liczba elementów obliczeniowych wykorzystywanego układu Zynq nie pozwala na realizację rozbudowanych rozwiązań algorytmicznych przy użyciu zaproponowanych technik. Ze względu na duże zapotrzebowanie na elementy logiki przez moduły AXI VDMA, buforowanie pełnych ramek obrazu jest kosztowne. W przypadku bardziej rozbudowanych algorytmów, konieczne może być wykorzystanie układu o większych możliwościach lub zastosowanie technik optymalizacji zużycia zasobów.
- Procesor ARM dostępny w układzie Zynq nie pozwala na realizację algorytmów wizyjnych o dużej złożoności obliczeniowej w czasie rzeczywistym. Próba wykorzystania rozwiązań biblioteki OpenCV do indeksacji obiektów pierwszoplanowych nie spełniała ograniczeń czasowych dla sygnału o częstotliwości $60Hz$.

W przypadku bardziej złożonych algorytmów, konieczne może być wykorzystanie układu o większej wydajności. Innym rozwiązaniem może być użycie protokołów sieciowych do transmisji danych do elementu obliczeniowego oferującego wydajność wystarczającą do realizacji zadań obliczeniowych. Dla zbioru algorytmów, których realizacja strumieniowa jest znana, możliwe jest również przeniesienie zadań obliczeniowych do elementów logiki reprogramowalnej. W przypadku, gdy żadne z zaproponowanych rozwiązań nie jest możliwe, konieczne jest ograniczenie częstotliwości działania algorytmu do poziomu, dla którego układ obliczeniowy będzie spełniać ograniczenia czasowe.

- Proces budowy systemu operacyjnego na bazie projektu sprzętowego jest złożony i wymaga dużych nakładów czasowych. Z tego powodu, na etapie projektowania połączeń logiki reprogramowalnej, wykorzystanie aplikacji typu *bare-metal* pozwala skrócić okres prototypowania.

W konsekwencji, konieczne może być zaprojektowanie dwóch aplikacji związanych z projektem: aplikacji *bare-metal*, wykorzystywanej na etapie prototypu oraz programu działającego pod obsługą systemu operacyjnego, projektowanego po ukończeniu implementacji sprzętowej.

Z tego powodu, aplikacje systemu operacyjnego nie pozwalają w pełni zastąpić programów *bare-metal* i powinny być traktowane jako metoda rozbudowy możliwości projektu.

- Proces komunikacji przy użyciu protokołu AXI narażony jest na błędy związane z koniecznością synchronizacji strumieni wizyjnych oraz opóźnienia wprowadzane przez moduły algorytmiczne. Prowadzi to do niestabilnego działania algorytmu.

5. Proces konfiguracji modułów logiki programowalnej i systemu operacyjnego PetaLinux

Użycie funkcjonalności opisywanych w niniejszej pracy wymaga przeprowadzenia konfiguracji wykorzystywanych modułów logicznych oraz systemu operacyjnego. W poniższych podrozdziałach zebrano informacje związane z poruszanymi zagadnieniami. Przedstawiono proces konfiguracji bazowego projektu Vivado i wykorzystania go na etapie budowy systemu PetaLinux, a także omówiono proces użycia modułów AXI i wybranych funkcjonalności systemu PetaLinux – obliczeń równoległych, biblioteki OpenCV, mechanizmu przerwań systemowych. Przedstawiono również proces konfiguracji modułu generacji tła zaproponowanego w rozdziale 4.

Omawiane zagadnienia wymagają użycia oprogramowania Vivado [18] i zintegrowanego z nim środowiska programistycznego Xilinx SDK [19]. Ponadto, konfiguracja systemu operacyjnego odbywa się przy użyciu narzędzi pakietu PetaLinux Tools [9].

5.1. Instalacja środowiska PetaLinux

Zbiór narzędzi PetaLinux może zostać pobrany ze strony producenta [9]. Pamiętać należy, że pakiet współpracuje ze środowiskiem Vivado wyłącznie w tej samej wersji.

PetaLinux wymaga zainstalowania zbioru kilkudziesięciu bibliotek i programów w ramach zależności. Pełen zbiór znaleźć można w opracowaniu [40]. W przypadku systemu Debian, większość bibliotek jest do niego dołączona, a brakujące pakiety mogą zostać dodane przy użyciu polecenia:

```
sudo apt-get install tofrodos iproute2 gawk gcc git make net-tools \
libncurses5-dev tftpd zlib1g-dev libssl-dev flex bison libselinux1
```

Ponadto, w przypadku systemu w wersji *64-bit*, konieczna jest również instalacja bibliotek *32-bit*:

```
sudo dpkg --add-architecture i386
sudo apt-get update

sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 \
libgtk2.0-0:i386 libxtst6:i386 gtk2-engines-murrine:i386 \
```

```
lib32stdc++6 libxt6:i386 libdbus-glib-1-2:i386 libasound2:i386
```

Proces instalacji przeprowadzić można przy użyciu polecień:

```
PETALINUX_DIR=ściezka-instalacji  
mkdir -p $PETALINUX_DIR  
.:/petalinux-v2016.3-final-installer.run $PETALINUX_DIR
```

Użycie narzędzi pakietu wymaga wcześniejszego wykonania skryptu przygotowującego środowisko po każdym uruchomieniu konsoli:

```
source $PETALINUX_DIR/settings.sh
```

Proces ten można uprościć, dodając powyższy wpis do pliku konfiguracyjnego powłoki *bash* – *.bashrc* wewnątrz katalogu domowego użytkownika.

5.2. Podstawowa konfiguracja projektu

Wykorzystana w projekcie karta – Digilent ZYBO – nie jest bezpośrednio wspierana przez środowisko Vivado. Wynika z tego konieczność sprecyzowania parametrów układu na etapie tworzenia projektu. W trakcie modyfikowania projektu, w przypadku dodania modułów wykorzystujących interfejs wejścia/wyjścia, konieczna jest również konfiguracja parametrów interfejsu. Aby uprościć proces projektowania, zalecane jest skonfigurowanie obsługi układu przed utworzeniem projektu. Proces ten opisano w dokumentacji producenta [41].

5.2.1. Vivado

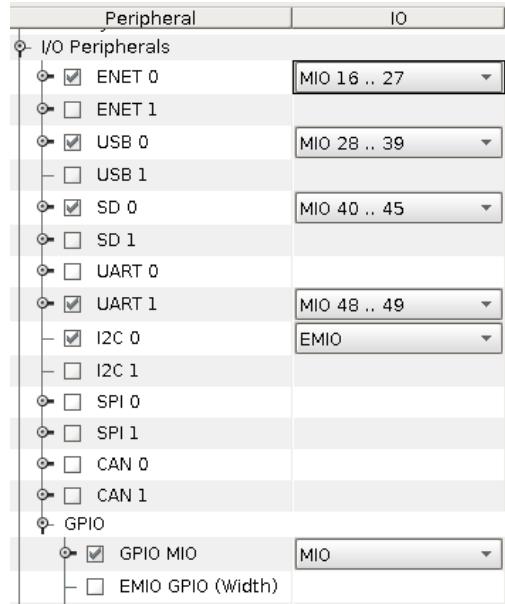
Utworzyć należy projektu typu „*RTL Project*”, z odznaczoną opcją „*Do not specify sources at this time*”. W kroku „*Add Constraints*” dodać należy plik konfiguracyjny dla wybranego układu. W przypadku ZYBO, plik ten jest dostępny na stronie producenta. W kolejnym kroku możliwe jest skonfigurowanie parametrów układu. Wykorzystać do tego należy zakładkę „*Boards*” i wybrać wykorzystywany model.

Po utworzeniu projektu, skonfigurować należy właściwą przestrzeń roboczą dla projektu, wykorzystując do tego opcję „*IP Integrator -> Create Block Design*”. Do nowo utworzonej przestrzeni dodać należy moduł IP reprezentujący procesor ZYNQ – „*ZYNQ7 Processing System*”. W kolejnych krokach należy dokonać konfiguracji modułu procesora, klikając dwukrotnie na moduł.

- Kanały interfejsu AXI mogą być konfigurowane przez zakładkę „*PS-PL Configuration*”. Możliwa jest aktywacja kanałów ogólnego przeznaczenia (*GP*) oraz wysokiej wydajności (*HP*).

- Interfejsy komunikacji konfigurowane mogą być z poziomu zakładki „*MIO Configuration*”. Zalecane jest aktywowanie interfejsów *ENET 0*, *SD 0* i *UART 1* ze względu na ich wykorzystanie na dalszym etapie pracy.

Przykład konfiguracji interfejsów wejścia/wyjścia przedstawiono na rysunku 5.1.

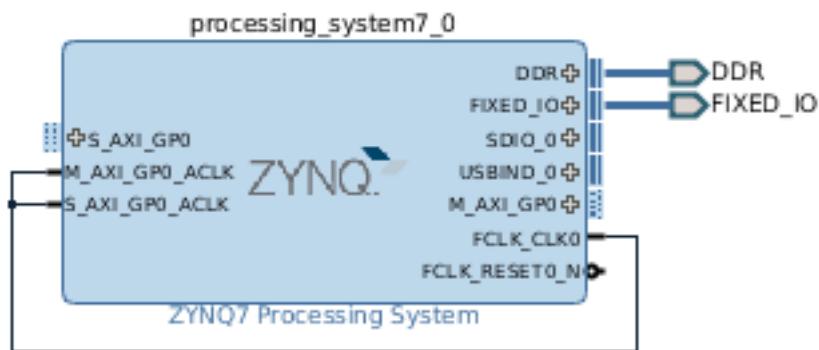


Rys. 5.1. Okno konfiguracji interfejsów wejścia i wyjścia.

- Parametry sygnałów zegarowych dostępnych z poziomu układów logiki reprogramowalnej modyfikować można w zakładce „*Clock Configuration/PL Fabric Clocks*”. W projekcie wymagającym obsługi AXI VDMA wykorzystano trzy sygnały zegarowe:
 - bazowy, o częstotliwości 100MHz,
 - wykorzystywany do komunikacji interfejsem AXI, o częstotliwości 140MHz,
 - zegar obsługi sekwencji wizyjnej, o częstotliwości 200MHz, umożliwiający współpracę ze strumieniem wideo o częstotliwości 60Hz i rozdzielcości co 1920×1080 pikseli.
- Częstotliwość pracy procesora oraz pamięci zmieniać można w zakładce „*Clock Configuration/Processor/Memory Clocks*”. Zdefiniować należy częstotliwość pracy CPU równą 650MHz oraz DRR równą 525MHz.

Po ukończeniu etapu konfiguracji procesora i powrocie do głównego okna programu, należy użyć opcji „*Run Block Automation*”. Utworzono zostaną połączenia interfejsów pamięci DDR oraz **FIXED_I0**. W przypadku zdefiniowania interfejsów AXI, połączyć należy właściwe sygnały zegarowe. Przykład wynikowej konfiguracji projektu przedstawiono na rysunku 5.2.

Przedstawiona konfiguracja stanowi podstawę każdego projektu wykorzystującego moduł procesora Zynq. Po zakończeniu konfiguracji, wygenerować należy warstwę HDL, korzystając



Rys. 5.2. Okno projektu.

z opcji „*Create HDL Wrapper*” dostępnej po kliknięciu prawym przyciskiem myszy na utworzony wcześniej plik źródłowy. Skonfigurowany w ten sposób projekt może być budowany i uruchamiany na platformie ZYBO.

5.2.2. SDK

W celu utworzenia projektu aplikacji w SDK, konieczne jest wyeksportowanie plików opisujących projekt z poziomu Vivado, wykorzystując do tego opcję „*File/Export/Export Hardware*” z zaznaczoną opcją „*Include bitstream*”. W efekcie, dostępny powinien być projekt *nazwa_projektu_hw_platform_0*, zawierająca plik *nazwa_projektu.hdf*, zawierający konfigurację sprzętową, stanowiący podstawę każdego budowanego programu *bare-metal*.

W przypadku budowania aplikacji na platformę PetaLinux, na etapie tworzenia projektu, zmodyfikować należy pole „*OS Platform*” na wartość „*linux*”, „*Processor Type*” na „*ps7_cortexa9*” oraz wybrać właściwy język programowania. Tak zdefiniowana aplikacja nie może korzystać z bibliotek udostępnianych przez firmę *Xilinx* (na przykład zawierających procedury obsługi modułów VDMA czy kontrolera przerwań), ale mogą korzystać z pełnej biblioteki standardowej języka *C* oraz rozszerzeń *POSIX*, w tym operacji wejścia/wyjścia, komunikacji sieciowej czy bibliotek matematycznych.

W celu uruchomienia aplikacji systemowej na karcie ZYBO, przeprowadzić należy proces budowania i skopiować wynikowy plik z katalogu *Debug* lub *Release* do systemu plików systemu PetaLinux. Wykorzystać można do tego narzędzie SSH:

```
scp Debug/hello-world.elf root@adres-ip:~/
```

Aplikację uruchomić można przy użyciu konsoli użytkownika, również stosując narzędzie SSH.

5.2.3. PetaLinux

Utworzenie struktury katalogów projektu wykonywane jest przy użyciu poniższego polecenia.

```
petalinux-create -t project --template zynq --name nazwa-projektu
```

```
cd nazwa-projektu  
git init .
```

Powstała struktura zintegrowana jest z systemem kontroli wersji *git*, co pozwala zachować uporządkowanie danych wewnętrz projektu oraz wersjonowanie. Kolejnym krokiem jest zainportowanie projektu *Vivado*.

```
petalinux-config --get-hw-description=/sciezka/do/projektu/projekt.sdk/
```

Jeśli polecenie wywołane zostało po raz pierwszy dla danego projektu, uruchomione zostanie narzędzie konfiguracyjne, domyślne ustawienia są jednak poprawne. Konfiguracja projektu odbywa się przy użyciu polecenia **petalinux-config**. Skonfigurować należy metodę uruchamiania systemu – w omawianym przypadku, uruchomienie następuje na bazie plików znajdujących się na karcie SD.

```
petalinux-config  
Image Packaging Configuration -> Root filesystem type -> SD card
```

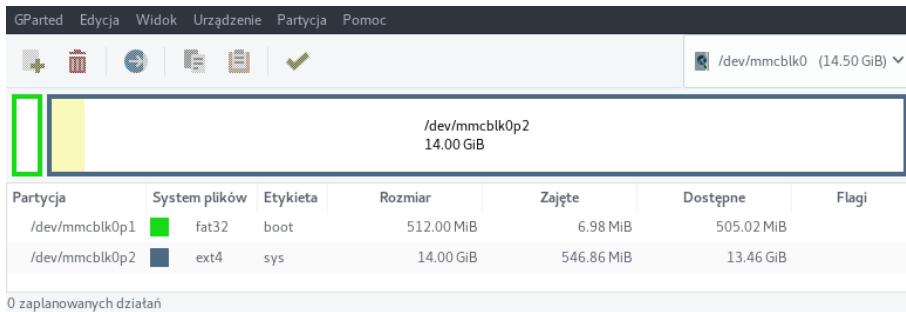
Należy również zmodyfikować argumenty przekazywane systemowi na etapie uruchamiania, umożliwiając wykorzystanie sterowników do modułów logiki reprogramowalnej.

```
petalinux-config  
Kernel Bootargs -> dezaktywować opcje Generate boot args automatically i zdefiniować  
własna wartości:  
console=ttyPS0,115200 earlyprintk uio_pdrv_genirq.of_id=generic-uio root=/dev/  
mmcblk0p2 rw rootwait
```

Następnie, przeprowadzić należy proces budowania systemu oraz wygenerować pliki wynikowe.

```
petalinux-build  
petalinux-package --boot --fsbl images/linux/zynq_fsbl.elf --fpga images/linux/  
system_wrapper.bit --u-boot --force  
petalinux-package --image -c rootfs --format initramfs
```

Uruchomienie systemu wymaga przygotowania karty SD – musi ona posiadać dwie partycje, pierwszą, z etykietą *boot* i systemem plików *fat32*, drugą – odpowiednio *sys* i *ext4*. Pierwsza z nich, zawierająca pliki wymagane na etapie inicjalizacji systemu, musi być poprzedzona 4 MB niezaallokowanej przestrzeni i mieć rozmiar co najmniej 40 MB. Druga partycja zawiera pliki systemowe, jej rozmiar powinien wynosić co najmniej kilkaset megabajtów. Proces formatowania przeprowadzić można przy użyciu narzędzia *gparted*. Na rysunku 5.3 przedstawiono efekt konfiguracji karty SD.



Rys. 5.3. Partycjonowane karty SD przy użyciu programu gparted.

Pliki wynikowe należy przenieść na kartę SD, korzystając z poniższych poleceń.

```
rm -rf /punkt-montowania/sys/*
cp images/linux/BOOT.BIN /punkt-montowania/boot/
cp images/linux/image.ub /punkt-montowania/boot/
cp images/linux/rootfs.cpio /punkt-montowania/sys/
cd /punkt-montowania/
pax -rvf rootfs.cpio
sync
cd -
```

Ze względu na mechanizm buforowania przez kontroler operacji zapisu danych, pamiętać należy o wywołaniu polecenia `sync`, zapewniającego zachowanie integralności danych. Karta SD pozwala na uruchomienie systemu operacyjnego na karcie i przechowywanie danych użytkownika pomiędzy startami układu. Dalsza praca z systemem odbywać się może przez protokoły komunikacji *SSH* lub *UART*.

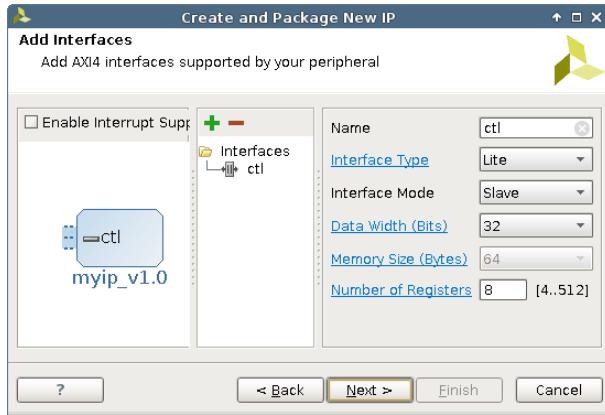
5.3. Konfiguracja modułu wykorzystującego interfejs AXI

Interfejs AXI stanowi podstawową metodę komunikacji pomiędzy modułami logiki programowej oraz elementami FPGA i CPU. Dzięki wykorzystaniu tego protokołu do obsługi projektowanych narzędzi, możliwe jest użycie zbioru elementów dostępnych w bibliotece Vivado i uproszczenie procesu projektowania procedur komunikacyjnych. Poniżej przedstawiono kroki konfiguracyjne wewnętrz narzędzi Vivado i PetaLinux.

5.3.1. Vivado

Oprogramowanie Vivado umożliwia zbudowanie modułu wykorzystującego protokół AXI przez użycie opcji „*Create and package new IP...*”, zawartej w menu *Tools*. Na ekranie wyboru zadania wybrać należy opcję „*Create a new AXI4 peripheral*”. Po zdefiniowaniu podstawowych danych związanych z modułem, takich jak jego nazwa i nazwisko autora, w kolejnym kroku możliwe będzie zdefiniowanie interfejsu modułu. Na tym etapie konfiguracji dodać należy wszystkie połączenia wykorzystujące interfejs AXI. W przypadku modułu konfiguracyjnego o podstawowej

strukturze, interfejs zawierać powinien jedno połączenie wykorzystujące protokół AXI w wersji *Lite*, działający w trybie *slave*, z oczekiwana liczbą rejestrów. Każdy register powinien być związany z jedną wartością, której konfiguracja ma być możliwa. Przykład konfiguracji przedstawiono na rysunku 5.4.



Rys. 5.4. Konfiguracja interfejsów modułu AXI DMA.

W omawianym przykładzie zdefiniowano interfejs AXI o nazwie *ctl*, związany z ośmioma rejestrami o długości trzydziestu dwóch bitów w pamięci. Zdefiniowanie interfejsów kończy proces podstawowej konfiguracji modułu. W kolejnym kroku należy wybrać opcję „Edit IP” w celu dostosowania kodu źródłowego modułu.

Po wygenerowaniu, z modułem powinien być związany jeden plik źródłowy, zwierający instrukcje odpowiadające za obsługę komunikacji przy użyciu interfejsu AXI. Do pliku dodać należy elementy odpowiedzialne za zdefiniowanie wyjść modułu oraz przypisanie im właściwych wartości. W celu zadeklarowania wyjść modułu, odpowiadające im wpisy należy umieścić po komentarzu „// Users to add ports here”. Przykład przedstawiono na listingu 5.1.

Listing 5.1. Definicja interfejsów wyjściowych modułu.

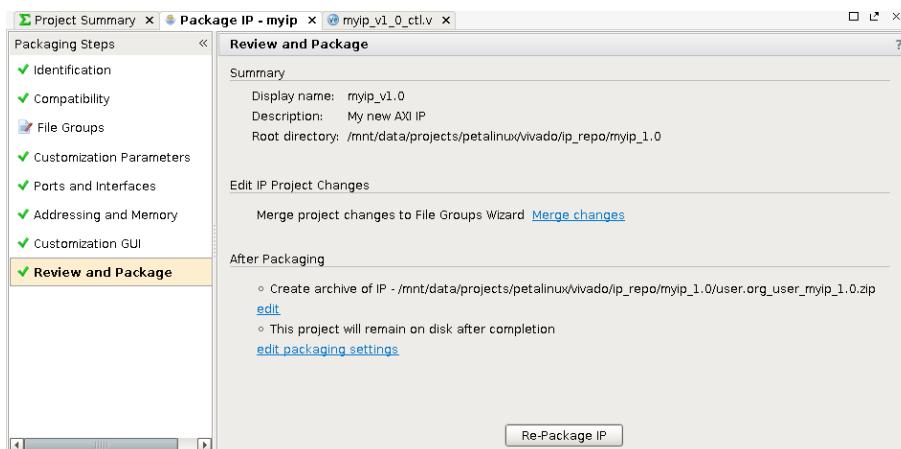
```
// Users to add ports here
output wire parameter_a,
output wire [7:0] parameter_b,
output wire [15:0] parameter_c,
output wire [31:0] parameter_d,
// User ports ends
```

Zdefiniowano cztery sygnały wyjściowe, o różnej liczbie bitów. Następnie, należy dokonać modyfikacji kodu odpowiedzialnego za powiązanie wartości parametrów z rejestrami modułu. Rejestry AXI zdefiniowane są poniżej linii „// Number of Slave Registers N”, gdzie *N* to liczba dostępnych rejestrów. Rejestry te mają nazwy *slv_regn*, gdzie *n* to indeks rejestru – nie jest zalecana modyfikacja tych nazw. Modyfikacji kodu należy dokonać poniżej linii „// Add user logic here”. Przykład przedstawiono na listingu 5.2.

Listing 5.2. Powiązanie wyjść z rejestrami modułu.

```
// Add user logic here
assign parameter_a = slv_reg0[0];
assign parameter_b = slv_reg1[7:0];
assign parameter_c = slv_reg2[15:0];
assign parameter_d = slv_reg3[31:0];
// User logic ends
```

Wartości parametrów powiązano bezpośrednio z danymi znajdującymi się w rejestrach. W rozbudowanych aplikacjach może być konieczne dodanie instrukcji modyfikujących wartości rejestrów przed przesłaniem ich na wyjście modułu. Po ukończeniu modyfikacji modułu, konieczne jest zapisanie zmian i wygenerowanie plików wynikowych. W tym celu należy wykorzystać okno „*Package IP*”, sekcję „*Review and Package*”. Widok narzędzia przedstawiono na rysunku 5.5.



Rys. 5.5. Okno finalizacji modyfikacji modułu.

Należy wybrać opcję „*Merge changes*”, umożliwiającą zintegrowanie wprowadzonych zmian z projektem bazowym. Następnie, można zakończyć edycję projektu przez wybór opcji „*Re-Packaging IP*”. Moduł będzie dostępny z poziomu interfejsu wyszukiwania modułów IP. Dodanie modułu do projektu wymaga zdefiniowania adresu pamięci z nim związanego. Wykorzystać do tego należy okno „*Address Editor*”, dostępną w z poziomu głównego okna projektu. W omawianym przykładzie, z modułem powiązano przestrzeń rozpoczęającą się od adresu 0x43000000 i długości 64K.

5.3.2. SDK

Konfiguracja wartości parametrów modułu opiera się na zapisie pod właściwe adresy pamięci. W przypadku pracy w trybie *bare-metal*, wykorzystać można instrukcję *Xil_Out32* z biblioteki *xil_io.h*. W przypadku pracy z systemem PetaLinux, wykorzystać należy biblioteki systemowe. Implementację *bare-metal* przedstawiono na listingu 5.3, natomiast systemową na listingach 5.4, 5.5 i 5.6.

Listing 5.3. Obsługa modułu w trybie bare-metal.

```
#include "xparameters.h"
#include "platform.h"
#include "xil_io.h"

#define PARAMETER_A_REGISTER 0
#define PARAMETER_B_REGISTER 4
#define PARAMETER_C_REGISTER 8
#define PARAMETER_D_REGISTER 12

#define BASEADDR XPAR_ALGORITHM_PARAMETERS_0_CTL_BASEADDR

int main()
{
    init_platform();

    Xil_Out32(BASEADDR + PARAMETER_A_REGISTER, 1);
    Xil_Out32(BASEADDR + PARAMETER_B_REGISTER, 25);
    Xil_Out32(BASEADDR + PARAMETER_C_REGISTER, 1 << 10);
    Xil_Out32(BASEADDR + PARAMETER_D_REGISTER, 1 << 30);

    while(1);
}
```

Listing 5.4. Obsługa modułu w trybie systemowym - `main.c`.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

#include "axi.h"

#define PARAMETER_A_REGISTER 0
#define PARAMETER_B_REGISTER 4
#define PARAMETER_C_REGISTER 8
#define PARAMETER_D_REGISTER 12

#define BASEADDR 0x43000000

typedef int memory_handle_t;

void setup_virtual_memory(struct axi_interface *interface, size_t length,
    memory_handle_t memory_handle, off_t base_addr) {
    interface->base_addr = base_addr;
    interface->virt_addr = (virt_address) mmap(NULL, length, PROT_READ | PROT_WRITE,
        MAP_SHARED, memory_handle, base_addr);
```

```

if (interface->virt_addr == MAP_FAILED) {
    perror("Failed to map virtual memory.");
    exit(1);
}

int main() {
    memory_handle_t memory_handle = open("/dev/mem", O_RDWR | O_SYNC);

    struct axi_interface* parameters = (struct axi_interface*) malloc(sizeof(struct
        axi_interface));
    if (parameters == NULL) {
        perror("Memory allocation failed.");
        exit(1);
    }
    setup_virtual_memory(parameters, 65535, memory_handle, BASEADDR);

    axi_write(parameters->virt_addr, PARAMETER_A_REGISTER, 1);
    axi_write(parameters->virt_addr, PARAMETER_B_REGISTER, 25);
    axi_write(parameters->virt_addr, PARAMETER_C_REGISTER, 1 << 10);
    axi_write(parameters->virt_addr, PARAMETER_D_REGISTER, 1 << 30);

    unsigned int parameter_a = axi_read(parameters->virt_addr, PARAMETER_A_REGISTER);

    while(1);
}

```

Listing 5.5. Obsługa modułu w trybie systemowym - `axi.h`.

```

typedef unsigned int* virt_address;

struct axi_interface {
    unsigned int base_addr;
    virt_address virt_addr;
};

void axi_write(virt_address virt_addr, int location, unsigned int value);
unsigned int axi_read(virt_address virt_addr, int location);

```

Listing 5.6. Obsługa modułu w trybie systemowym - `axi.c`.

```

#include "axi.h"

void axi_write(virt_address virt_addr, int location, unsigned int value) {
    virt_addr[location >> 2] = value;
}

unsigned int axi_read(virt_address virt_addr, int location) {

```

```
    return virt_addr[location >> 2];  
}
```

System Linux udostępnia zbiór procedur związanych z obsługą pamięci operacyjnej, w tym pozwalające na wirtualizację fizycznych adresów, co jest konieczne do obsługi urządzeń peryferyjnych z poziomu systemu operacyjnego. Adres fizyczny urządzenia zdefiniowano przez nazwę **BASEADDR**. Określono również przesunięcia adresów kolejnych rejestrów modułu, na przykład **PARAMETER_A_REGISTER**.

Procedura **setup_virtual_memory** przyjmuje jako argumenty wskaźnik do struktury interfejsu AXI, zawierającej informacje o adresach fizycznym i wirtualnym pamięci, rozmiarze przestrzeni adresowej w bajtach, a także uchwyt do kontrolera pamięci systemowej oraz adres fizyczny modułu. W wyniku konfiguracji obszar pamięci fizycznej mapowany jest w przestrzeni adresowej systemu, co pozwala na odczyt i zapis wartości. Aby uprościć mechanizm odczytu i zapisu do pamięci, udostępniono procedury **axi_write** i **axi_read**, których argumentami wywołań jest bazowy adres wirtualny modułu oraz przesunięcie wybranego rejestru.

Poprawność działania komunikacji przetestowano na przykładzie modułu liczącego, kontrolowanego z poziomu systemu operacyjnego. Moduł udostępniał rejestr kontrolny, którego najmłodszy bit odpowiadał za aktywację pracy licznika, oraz rejestr przechowujący zliczaną wartość. Zapisując i odczytując wartości rejestrów zweryfikowano, że komunikacja z modułem ma prawidłowy przebieg.

5.3.3. PetaLinux

W celu wykorzystania techniki DMA w aplikacji działającej w systemie PetaLinux, konieczna jest aktywacja właściwych parametrów konfiguracji na etapie budowania systemu. W tym celu wykonać należy polecenie:

```
petalinux-config -c kernel
```

i aktywować funkcjonalność DMA:

```
Device Drivers -> DMA Engine Support  
Device Drivers -> DMA Engine Support -> Xilinx AXI DMAS Engine
```

Włączenie sterowników DMA oraz zmodyfikowanie argumentów uruchomienia systemu operacyjnego, opisane w rozdziale 5.2.3, pozwala na wykorzystanie interfejsu AXI i techniki DMA do komunikacji z modułami logiki programowalnej.

5.4. Konfiguracja modułu AXI VDMA

Proces konfiguracji modułu VDMA składa się z kroków podobnych do opisanych w rozdziale 5.3, poświęconej modułom AXI DMA. Poniżej przedstawiono dodatkowe kroki, związane bezpośrednio z konfiguracją modułu VDMA.

5.4.1. Vivado

Do projektu dołączyć należy moduł *AXI Video Direct Memory Access*. Okno konfiguracji związane z nim pozwala na wybór obsługiwanych kanałów:

- write (*S2MM*) – kanał zapisu, pozwalający na transmisję danych z formatu strumieniowego do pamięci operacyjnej,
- read (*MM2S*) – kanał odczytu, umożliwiający konwersję danych przechowywanych w pamięci do strumienia.

Ustawienia pozwalają na wybór szerokości strumienia informacji dla jednego piksela, maksymalną liczbę buforów w pamięci oraz długość linii buforujących, związanych z oboma kanałami. Wartości wielkości strumienia danych oraz liczby buforów związane są ściśle z projektowanym algorytmem, natomiast długość linii buforujących może wpływać na stabilność działania systemu. Zwiększenie tej wartości może poprawić działanie algorytmu w przypadku, gdy operacje związane z pamięcią operacyjną wykonywane są z opóźnieniem.

Zakładka ustawień zaawansowanych pozwala na zdefiniowanie parametrów związanych ze sterowaniem kanałami transmisji. Wartość parametru „*Fsync Options*” w aplikacjach nie wymagających zewnętrznej synchronizacji powinna być zdefiniowana jako **tuser** dla kanału zapisu oraz **none** dla kanału odczytu, dzięki czemu sygnał synchronizacji modułu będzie związany z wejściowym strumieniem AXI. Część aplikacji może wymagać synchronizacji strumienia odczytu z drugim strumieniem danych, na przykład z inną ramką sygnału wizyjnego. W takiej sytuacji wykorzystać należy opcję synchronizacji **fsync**, a wejście układu **mm2s_fsync** połączyć z właściwym sygnałem synchronizacji.

W ramach pracy wykorzystywano również synchronizację pomiędzy kanałami przy użyciu parametru **GenLock**, o wartości **master** dla kanału zapisu i **slave** dla odczytu. Pozwalało to zachować przesunięcie o stałe, definiowanej z poziomu aplikacji, wartości pomiędzy buforami wykorzystywanymi przez oba kanały.

Ze względu na dużą wartość przepływu danych przez oba kanały, do komunikacji z procesorem wykorzystać należy połączenia o wysokiej wydajności. Kanały te można aktywować korzystając z opcji konfiguracyjnych modułu *ZYNQ7 Processing System: „PL-PS Configuration/HP Slave AXI Interface”*, i aktywując jeden lub wiele kanałów. Z modułem AXI VDMA powiązać należy sygnał zegarowy o częstotliwości nie mniejszej od wartości tak zwanego zegara piksela, związanego ze strumieniem wizyjnym na wejściu. Sygnał ten powinien być generowany przez układ ZYNQ, a nie powiązany bezpośrednio z zegarem strumienia obrazu.

5.4.2. SDK

Konfiguracja modułu VDMA wymaga zastosowania technik opisanych w rozdziale 5.3.2.

Proces uruchamiania transmisji dla modułu wymaga wykonania kroków zdefiniowanych przez producenta i opisanych w rozdziale „*Programming Sequence*” dokumentacji [15]. Kod programu odpowiedzialnego za konfigurację modułu AXI VDMA przedstawiono na listingu 5.7.

Listing 5.7. Obsługa modułu AXI VDMA w aplikacji *bare-metal*.

```
#include "xparameters.h"
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "sleep.h"
#include "xaxivdma.h"

#define S2MM_VDMACR 0x30
#define S2MM_VDMASR 0x34
#define S2MM_VSIZE 0xA0
#define S2MM_HSIZE 0xA4
#define S2MM_FRMDLY_STRIDE 0xA8
#define S2MM_START_ADDRESS 0xAC
#define PART_PTR_REG 0x28
#define MM2S_VDMACR 0x00
#define MM2S_VDMASR 0x04
#define MM2S_VSIZE 0x50
#define MM2S_HSIZE 0x54
#define MM2S_FRMDLY_STRIDE 0x58
#define MM2S_START_ADDRESS 0x5C
#define HSIZE_FULL 1980
#define VSIZE_FULL 750

#define HSIZE_ACTIVE 1280
#define VSIZE_ACTIVE 720

#define PIXEL_SIZE 3
static volatile u8 framebuffer1[VSIZE_ACTIVE*HSIZE_ACTIVE*PIXEL_SIZE] = {0};
static volatile u8 framebuffer2[VSIZE_ACTIVE*HSIZE_ACTIVE*PIXEL_SIZE] = {0};
static volatile u8 framebuffer3[VSIZE_ACTIVE*HSIZE_ACTIVE*PIXEL_SIZE] = {0};

XAxiVdma AxiVdmaFrameBuffering;

void debug_vdma(UINTPTR addr);

void init_vdma_buffer(UINTPTR addr, u32 fb1, u32 fb2, u32 fb3)
{
    // 1
    Xil_Out32(addr + MM2S_VDMACR,
    (255 << 16) | 0x8 | 0x80 | 0x2 | 0x1);
    Xil_Out32(addr + S2MM_VDMACR,
    (255 << 16) | 0x8 | 0x80 | 0x2 | 0x1);
```

```

// 2
Xil_Out32(addr + MM2S_START_ADDRESS, fb1);
Xil_Out32(addr + MM2S_START_ADDRESS + 4, fb2);
Xil_Out32(addr + MM2S_START_ADDRESS + 8, fb3);

Xil_Out32(addr + S2MM_START_ADDRESS, fb1);
Xil_Out32(addr + S2MM_START_ADDRESS + 4, fb2);
Xil_Out32(addr + S2MM_START_ADDRESS + 8, fb3);

// 3
Xil_Out32(addr + MM2S_FRMDLY_STRIDE, HSIZE_ACTIVE*PIXEL_SIZE);
Xil_Out32(addr + S2MM_FRMDLY_STRIDE, HSIZE_ACTIVE*PIXEL_SIZE | (2 << 24));

// 4
Xil_Out32(addr + MM2S_HSIZE, HSIZE_ACTIVE*PIXEL_SIZE);
Xil_Out32(addr + S2MM_HSIZE, HSIZE_ACTIVE*PIXEL_SIZE);

// 5
Xil_Out32(addr + S2MM_VSIZE, VSIZE_ACTIVE);
Xil_Out32(addr + MM2S_VSIZE, VSIZE_ACTIVE);
}

int main()
{
    init_platform();

    xil_printf("Config - vdma\r\n");
    // vdma frame buffering
    init_vdma_buffer(XPAR_FRAME_BUFFER_VDMA_PREVIOUS_FRAME_BASEADDR,
        (u32)&framebuffer1, (u32)&framebuffer2, (u32)&framebuffer3);

    xil_printf("Config - done\r\n");
    while(1) {
        xil_printf("XPAR_FRAME_BUFFER_VDMA_PREVIOUS_FRAME\r\n");
        debug_vdma(XPAR_FRAME_BUFFER_VDMA_PREVIOUS_FRAME_BASEADDR);
        sleep(1);
    }
    cleanup_platform();
    return 0;
}

```

Funkcja `init_vdma_buffer` jest odpowiedzialna za przeprowadzenie konfiguracji modułu VDMA. Jej argumenty stanowią adres elementu VDMA oraz adresy trzech buforów obrazu. Zagwarantować należy, by przestrzeń zarezerwowana na każdy bufor była wystarczająca do przechowania pełnej ramki obrazu. W przypadku wykorzystania modułu VDMA do transmisji

kontekstu związanego z każdym pikselem, zmodyfikować należy wartość `PIXEL_SIZE` do liczby bajtów zajmowanej przez jeden piksel.

Procedura `init_vdma_buffer` wykonuje szereg operacji związanych z uruchomieniem transmisji sygnału dla obu kanałów niezależnie. Przedstawione poniżej działania związane są z indeksami znajdującymi się wewnątrz komentarzy w listingu.

1. Konfiguracja przerwań i uruchomienie kanału VDMA. W prezentowanym przykładzie, oba kanały skonfigurowano do działania w trybie cyklicznym, korzystającym naprzemiennie ze wszystkich buforów obrazu, z aktywowanym trybem synchronizacji *Genlock* o wewnętrznym źródle sygnału.
2. Przypisanie adresów buforów obrazu. Mogą być one wspólne lub unikalne dla każdego kanału.
3. Zdefiniowanie parametrów obrazu wejściowego z sygnałami wygaszania i wzajemnego opóźnienia kanałów. W omawianym przykładzie, kanał odczytu zachowuje opóźnienie dwóch klatek obrazu względem kanału zapisu.
4. Przypisanie rozmiaru jednej linii obrazu, z uwzględnieniem wielkości piksela w bajtach, nie uwzględniając cykli wygaszania.
5. Zdefiniowanie liczby linii obrazu, nie uwzględniając cykli wygaszania.

Wpisanie wartości do rejestrów `S2MM_VSIZE` i `MM2S_VSIZE` powoduje rozpoczęcie transmisji sygnału.

5.4.3. Petalinux

Konfiguracja projektu zgodna jest z opisem dla modułów DMA, przedstawionym w rozdziale 5.3.3. W przypadku projektu aplikacji działającej pod kontrolą systemu operacyjnego, pamiętać należy, że konfiguracja buforów obrazu wymaga użycia adresów fizycznych, które mogą różnić się od adresów wirtualnych komórek pamięci.

Zdefiniować należy adresy buforów, odległe od siebie co najmniej o rozmiar jednej ramki sygnału wizyjnego. Zagwarantować należy nienaruszalność pamięci z perspektywy systemu operacyjnego. Efekt ten najprościej jest osiągnąć przez ograniczenie rozmiaru pamięci dostępnej dla systemu operacyjnego i zdefiniowanie adresów buforów poza tym zakresem. Wykorzystać do tego można argument `mem` przekazywany na etapie uruchamiania systemu, na przykład `mem=224M`. Proces dodawania argumentów uruchamiania systemu opisano w rozdziale 5.2.3.

Adres fizyczny pamięci nie może być odczytany bezpośrednio, w tym celu musi zostać powiązany z adresem wirtualnym. Odpowiedzialną za to procedurę `setup_virtual_memory` przedstawiono na listingu 5.4, przy czym parametr `base_addr` to adres fizyczny pierwszej komórki bufora.

5.5. Obliczenia równoległe

Użycie rozwiązań omawianych w rozdziale 3.4 wymaga aktywacji właściwych funkcji kompilacji. W przypadku zastosowania wątków natywnych lub biblioteki dostępnej w standardzie C++ wymagane jest aktywowanie przełączników:

```
gcc main.c -o main.out -pthread  
g++ main.cpp -o main.out -std=c++11 -pthread
```

Dla biblioteki TBB wymagane jest przeprowadzenie linkowania względem jej kodu źródłowego:

```
g++ main.cpp -o main.out -ltbb
```

Natomiast dla interfejsu OpenMP, konieczne jest użycie przełącznika:

```
g++ main.cpp -o main.out -fopenmp
```

Wykorzystanie omawianych rozwiązań wiąże się z użyciem dedykowanych procedur lub dyrektyw kompilatora. Etap projektowania aplikacji działającej pod kontrolą systemu operacyjnego PetaLinux nie różni się od budowania oprogramowania na inne platformy. Należy jednak pamiętać, że układ Zynq wyposażony jest w procesor ARM o dwóch rdzeniach, więc potencjalne korzyści zastosowania aplikacji wielowątkowej nie przekraczają dwukrotnego zwiększenia szybkości działania aplikacji. Sposób zastosowania bibliotek znaleźć można w dokumentacji każdego z narzędzi i literaturze cytowanej w związanym z tym zagadnieniem rozdziale.

5.6. OpenAMP

Informacje związane z procesem konfiguracji i uruchamiania systemu operacyjnego czasu rzeczywistego znaleźć można w opracowaniu [42]. Poniżej przedstawiono zwięzłe podsumowanie kroków koniecznych do wykonania w przypadku karty ZYBO.

5.6.1. Vivado

Uruchomienie systemu operacyjnego czasu rzeczywistego wymaga powiązania z nim portu szeregowego oraz zegara. Skonfigurować należy moduł procesora ZYNQ:

- Włączyć obsługę portu UART: „*Peripheral I/O Pins -> UART 0*”.
- Z systemem powiązać należy sygnał zegarowy: „*MIO Configuration -> Application Processor Unit -> Timer 1 (EMIO)*”.

5.6.2. PetaLinux

Uruchomienie systemu czasu rzeczywistego wymaga zbudowania projektu typu *Board Support Package* i przeprowadzenie procesu konfiguracji. Proces ten wymaga zewnętrznych zależności i może być trudny do wykonania dla kart uruchomieniowych, które nie są w pełni wspierane przez producenta PetaLinux, takich jak ZYBO. Zamierzony efekt można osiągnąć przez wykorzystanie zasobów dostępnych dla karty Zedboard lub w repozytorium projektu OpenAMP [43].

W przypadku wykorzystania plików BSP karty Zedboard, skopiować należy pliki `device-tree openamp.dts` oraz `openamp-overlay.dtsi` do ścieżki `subsystems/linux/configs/device-tree` wewnątrz projektu PetaLinux, a następnie zmodyfikować plik `system-top.dts`, dodając na końcu listy dyrektyw `/include/ wpis /include/ openamp-overlay.dtsi"`.

Następnie należy włączyć wymagane do obsługi systemu czasu rzeczywistego funkcjonalności PetaLinux.

```
petalinux-config -c kernel
-> Kernel Features
-> Memory split
-> 2G/2G user/kernel split
* High Memory Support
-> Device Drivers
-> Generic Driver Options
* Userspace firmware loading support
-> Remoteproc drivers
```

A także dodać do obrazu systemu aplikacje testowe i moduły:

```
petalinux-config -c rootfs
Apps
* echo_test
* mat_mul_demo
* proxy_app
Modules
* rpmsg_proxy_dev_driver
* rpmsg_user_dev_driver
```

Po uruchomieniu systemu operacyjnego, uruchomienie i test funkcjonalności może być wykonany przy użyciu poniższych poleceń.

```
modprobe zynq_remoteproc firmware=image_echo_test
modprobe rpmsg_user_dev_driver
echo_test
```

5.7. Biblioteka OpenCV

5.7.1. OpenCV 2

Biblioteka OpenCV w wersji 2.4 nie jest oficjalnie dostępna w pakiecie PetaLinux, może jednak zostać dołączona do systemu operacyjnego dzięki mechanizmowi aplikacji użytkownika. Należy zatem przeprowadzić proces komplikacji kodu źródłowego biblioteki wraz z zależnościami, ponieważ prekompilowane pliki na platformę ARM nie są publicznie dostępne. Poniżej przedstawiono proces instalacji zależności.

Listing 5.8. Definicje zmiennych środowiskowych.

```
export ARMPREFIX=ściezka/instalacji
export CCPREFIX=arm-linux-gnueabihf -
```

Zmienna CCPREFIX wskazuje na prefiks kompilatora zawartego w pakiecie PetaLinux, a zmienna ARMPREFIX wskazuje na ścieżkę, gdzie zainstalowane zostaną pliki wynikowe.

Listing 5.9. Kompilacja biblioteki *xVideo*.

```
wget http://downloads.xvid.org/downloads/xvidcore-1.3.3.tar.gz
tar -zxvf xvidcore-1.3.3.tar.gz
cd xvidcore/build/generic/
./configure --prefix=${ARMPREFIX} --host=arm-linux-gnueabihf --disable-assembly
make
make install
```

Listing 5.10. Kompilacja biblioteki *x264*.

```
git clone git://git.videolan.org/x264
cd x264
./configure --enable-shared --host=arm-linux-gnueabihf --disable-asm --prefix=${ARMPREFIX} --cross-prefix=${CCPREFIX}
make
make install
```

Listing 5.11. Kompilacja biblioteki *ffmpeg*.

```
git clone git://source.ffmpeg.org/ffmpeg.git
cd ffmpeg
git checkout release/2.6
./configure --enable-cross-compile --cross-prefix=${CCPREFIX} --target-os=linux \
--arch=arm --enable-shared --disable-static --enable-gpl --enable-nonfree \
--enable-ffmpeg --disable-ffplay --enable-ffserver --enable-swscale \
--enable-pthreads --disable-yasm --disable-stripping --enable-libx264 \
--disable-libxvid --prefix=${ARMPREFIX} --extra-cflags="-I\"${ARMPREFIX}\"/include" \
 \
--extra-ldflags="-L\"${ARMPREFIX}\"/lib"
```

```
make
make install
```

Kompilowane biblioteki zapewniają dostęp do procedur obsługi strumieni wideo oraz obrazów w najczęściej wykorzystywanych formatach. Po zainstalowaniu zależności, przystąpić można do pobrania i instalacji biblioteki OpenCV.

Listing 5.12. Pobieranie biblioteki OpenCV w wersji 2.4.10.

```
git clone https://github.com/Itseez/opencv.git
cd opencv
git checkout 2.4.10
```

Listing 5.13. Kompilacja biblioteki *OpenCV*.

```
mkdir build && cd build
cmake -DBUILD_DOCS=OFF -DBUILD_TESTS=OFF -DWITH_1394=OFF -DWITH_CUDA=OFF \
-DWITH_CUFFT=OFF -DWITH_EIGEN=OFF -DWITH_GSTREAMER=OFF -DWITH_GTK=OFF \
-DWITH_JASPER=OFF -DWITH_JPEG=OFF -DWITH_LIBV4L=OFF -DWITH_OPENEXR=OFF \
-DWITH_PNG=OFF -DWITH_PVAPI=OFF -DWITH_TIFF=OFF -DWITH_V4L=OFF \
-ENABLE_PRECOMPILED_HEADERS=OFF -DWITH_FFMPEG=ON \
-DCMAKE_SYSTEM_NAME=Linux -DCMAKE_SYSTEM_PROCESSOR=arm \
-DCMAKE_C_COMPILER=arm-linux-gnueabihf-gcc \
-DCMAKE_CXX_COMPILER=arm-linux-gnueabihf-g++ \
-DCMAKE_INSTALL_PREFIX=$ARMPREFIX \
-DCMAKE_FIND_ROOT_PATH=katalog/zawierający/narzędzia/kompilacji ../
make
make install
```

Aby zmniejszyć rozmiar biblioteki, a także skrócić proces instalacji, część modułów została dezaktywowana. Wartość zmiennej `CMAKE_FIND_ROOT_PATH` to ścieżka zawierająca strukturę katalogów wykorzystywanego kompilatora. W przypadku pakietu Petalinux w wersji 2016.3, właściwa ścieżka względem punktu instalacji pakietu to `Xilinx/Petalinux/tools/linux-i386/gcc-arm-linux-gnueabi/arm-linux-gnueabihf`.

Po zakończeniu procesu, pliki wynikowe znaleźć można w katalogu `$ARMPREFIX/lib`.

Pliki te mogą być dołączone do budowanego systemu operacyjnego jako dodatkowe zależności. W tym celu wykorzystać należy polecenie:

```
petalinux-create -t libs --template install --name opencv2
```

Utworzona zostanie struktura katalogów `components/libs/opencv2`, do której skopiować należy pliki wynikowe komplikacji biblioteki i jej zależności. Następnie, zmodyfikować należy plik `Makefile`, zgodnie z zawartymi w nim instrukcjami. W przypadku biblioteki OpenCV, wykorzystać można tekst generowany w wyniku wywołania polecenia:

```
for f in $(find . -type f -name "*.so*" -printf '%P\n'); \
do echo -e '\t$(TARGETINST) -d' $f /lib/$f; done
```

Aktywacja biblioteki wewnątrz projektu wymaga wywołania polecenia przedstawionego na listingu 5.14 i wyboru biblioteki w zakładce „*Libs*”.

Listing 5.14. Dołączenie biblioteki do projektu PetaLinux.

```
petalinux-config -c rootfs
```

Bibliotekę skompilowano i potwierdzono poprawność działania dla plików testowych dostępnych na stronie internetowej twórców. Porównano wyniki działania aplikacji modyfikującej obraz na wejściu i zapisującego wynik do pliku z programem uruchomionym na procesorze architektury *x86* i nie stwierdzono różnic.

5.7.2. OpenCV 3

Biblioteka OpenCV w wersji 3.1 dołączona jest do pakietu PetaLinux. W celu jej aktywacji, wykorzystać należy polecenie przedstawione na listingu 5.14 i wybrać biblioteki w zakładce „*Filesystem Packages/libs/opencv*”. Działanie biblioteki przetestowano na przykładzie programu dokonującego segmentacji obiektów pierwszoplanowych i ich indeksacji, przedstawionego na listingu 5.15.

Listing 5.15. Aplikacja indeksująca obiekty pierwszoplanowe.

```
#include <opencv2/opencv.hpp>
#include <iostream>

int main(int argc, char* argv[])
{
    cv::Mat input_image = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

    cv::Mat binary;
    cv::threshold(input_image, binary, 200, 255, 0);
    cv::imwrite("binary.png", binary);

    cv::Mat labels, stats, centroids;

    int num_labels = cv::connectedComponentsWithStats(binary, labels, stats, centroids
    );

    cv::imwrite("components.png", labels);

    for (int l = 1; l < num_labels; l++)
        std::cout << "#" << l << "(x,y) = (" << centroids.at<long double>(l, 0) << ", "
        << centroids.at<long double>(l, 1) << ")" << std::endl;

    return 0;
}
```

5.7.3. SDK

Wykorzystanie bibliotek w projekcie SDK wymaga wskazania katalogu ze źródłami oraz bibliotekami w ustawieniach projektu. W przypadku użycia biblioteki w wersji 3.1, wystarczające jest utworzenie aplikacji w języku C++ i typu *OpenCV Example Application*. Dla wersji 2.4, konieczne jest ręczne zmodyfikowanie parametrów komplikacji projektu, w sposób analogiczny do konfiguracji aplikacji wykorzystującej OpenCV i działającej na platformie x86, wskazując jednak na skompilowane wcześniej pliki dla platformy ARM.

Dokonać należy modyfikacji opcji projektu w ścieżce „*Tool Settings -> ARM v7 Linux g++ compiler -> Directories*” i dodać katalog `include` znajdujący się w strukturze plików: `ściezka/instalacji/include`. Zmodyfikować należy również opcje konfiguracji programu linkującego „*Tool Settings -> ARM v7 Linux g++ linker -> Libraries*”. Zdefiniować należy ścieżkę poszukiwania bibliotek `ściezka/instalacji/lib`, a także dodać wszystkie wykorzystywane moduły do listy używanych bibliotek, na przykład `opencv_core`, `opencv_imgproc`, `opencv_video`.

5.8. Wykorzystanie mechanizmu przerwań systemowych

Użycie mechanizmu przerwań systemowych wymaga zbudowania połączeń wewnętrz logiki programowalnej oraz konfiguracji agentów przerwań na poziomie aplikacji użytkownika. Poniżej opisano kroki wymagane do użycia omawianego mechanizmu w aplikacjach *bare-metal* oraz działających w systemie PetaLinux.

5.8.1. Vivado

Moduły wspierające mechanizm przerwań wyposażone są w dedykowane połączenia wyjściowe, wykorzystywane do transmisji sygnału przerwania. W przypadku modułu AXI Timer właściwe połączenie ma sygnaturę *interrupt*, natomiast w przypadku modułu AXI VDMA, sygnały przerwań dla kanałów odczytu i zapisu mają odpowiednio nazwy *mm2s_intout* oraz *s2mm_intout*.

Obsługa przerwań wymaga konfiguracji modułu procesora. Aktywować należy ścieżkę „*Fabric Interrupts -> PL-PS Interrupt Ports -> IRQ_F2P*” wewnątrz zakładki *Interrupts*. W rezultacie, dostępne będzie wejście procesora *IRQ_F2P* o szerokości do szesnastu linii. We wspomnianej zakładce ustawień aktywować można również inne połączenia przerwań, w tym szybkie przerwania w kierunku procesora oraz połączenia prowadzone od procesora do układów logiki, pozwalające na transmisję zdarzeń z interfejsów procesora, takich jak DMA, UART czy Ethernet.

Kanał *IRQ_F2P* pozwala na połączenie nie więcej niż szesnastu linii przerwań. W przypadku wykorzystania mechanizmu na platformie PetaLinux, pierwszym ośmiu liniom, zaczynając od najmłodszego bitu, przypisane będą identyfikatory przerwań w zakresie [61 – 68], natomiast pozostałym ośmiu – [84 – 91].

W przypadku konieczności zaprojektowania interfejsu wykorzystującego więcej niż szesnaście linii przerwań, konieczne jest zastosowanie układu dedykowanego obsługującego zdarzeń – „*AXI Interrupt Controller*”. Pozwala on na połączenie nie więcej niż trzydziestu dwóch linii przerwań do jednej linii na wejściu procesora i udostępnia interfejs umożliwiający identyfikację układu odpowiedzialnego za wysłanie sygnału przerwania. Zapewnia również mechanizmy priorytetyzacji i zagnieżdżania przerwań.

W sytuacji, gdy interfejs nie zawiera więcej niż szesnastu przerwań, wystarczające jest użycie modułu konkatenacji sygnałów zdarzeń do jednego wektora, którego wyjście połączone jest z wejściem *IRQ_F2P* procesora.

5.8.2. Aplikacja *bare-metal*

Wykorzystanie przerwań wymaga napisania procedury odpowiedzialnej za obsługę zdarzeń oraz zarejestrowanie jej jako agenta danego przerwania. Ponadto zwykle wymagane jest przeprowadzenie konfiguracji modułu w taki sposób, aby aktywować funkcję zgłaszania przerwań. Wymagane funkcje znaleźć można w plikach nagłówkowych *xparameters.h*, *xscugic.h*, *xil_exception.h*, oraz *xaxivdma.h* dla modułu AXI VDMA i *xtmrctr.h* dla AXI Timer.

Procedurę konfiguracji obsługi przerwań podzielić można na kilka etapów:

1. Zdefiniowanie agentów zdarzeń.

Konieczne jest zdefiniowane funkcji, które będą wywołane w przypadku wystąpienia przerwania. W najprostszym rozwiązaniu, ich celem jest akceptacja zdarzenia i przeprowadzenie konfiguracji modułu w taki sposób, aby umożliwić jego dalsze działanie – w przypadku modułu zegarowego jest to wykonanie restartu zegara. Moduł AXI VDMA nie wymaga żadnych kroków na etapie wywołania przerwania.

Ponadto, procedura jest odpowiedzialna za wykonanie obliczeń związanych z wystąpieniem przerwania.

Na listingu 5.16 przedstawiono funkcje agentów przerwań dla modułu zegara oraz obu kanałów AXI VDMA.

Listing 5.16. Procedury obsługi przerwań.

```
void Timer_InterruptHandler(void *data, u8 id) {
    // dodatkowe obliczenia

    // zerowanie przerwania
    XTmrCtr_Stop(data, id);
    XTmrCtr_Reset(data, id);
    XTmrCtr_Start(data, id);
}

void AxiRead_InterruptHandler(void *data, u32) {
```

```

    // dodatkowe obliczenia
}

void AxiWrite_InterruptHandler(void *data, u32) {
    // dodatkowe obliczenia
}

```

2. Konfiguracja modułów.

Oba omawiane moduły wymagają przeprowadzenia dodatkowych kroków konfiguracji. W przypadku modułu zegarowego, konieczne jest aktywacja obsługi przerwań w rejestrze kontrolnym – `TCSRn`, natomiast w przypadku modułu VDMA, parametryzacja odbywa się przez rejestrzy `MM2S_VDMACR` dla kanału zapisu oraz `S2MM_VDMACR` dla kanału odczytu.

Ponadto, konieczna jest rejestracja agentów przerwań dla obu modułów. Proces ten przedstawiono na listingu 5.17, zmienne `TimerInstancePtr` i `AxiVdmaInstancePtr` są wskaźnikami do wykorzystywanych struktur typu `XTmrCtr` i `XAxiVdma`.

Listing 5.17. Rejestracja agentów przerwań.

```

XAxiVdma_SetCallBack(AxiVdmaInstancePtr, XAXIVDMA_HANDLER_GENERAL,
    &AxiWrite_InterruptHandler, AxiVdmaInstancePtr, XAXIVDMA_WRITE);

XAxiVdma_SetCallBack(AxiVdmaInstancePtr, XAXIVDMA_HANDLER_GENERAL,
    &AxiRead_InterruptHandler, AxiVdmaInstancePtr, XAXIVDMA_READ);

XTmrCtr_SetHandler(TimerInstancePtr, Timer_InterruptHandler, TimerInstancePtr);

```

3. Konfiguracja kontrolera przerwań.

W ostatnim kroku następuje konfiguracja kontrolera zdarzeń. Procedurę przedstawiono na listingu 5.18.

Listing 5.18. Konfiguracja kontrolera przerwań.

```

XScuGic InterruptController;
XScuGic_Config *GicConfig;
int ScuGicInterrupt_Init(u16 DeviceId, XTmrCtr *TimerInstancePtr,
    XAxiVdma * AxiVdmaIntancePtr) {
    int Status;
    GicConfig = XScuGic_LookupConfig(DeviceId);
    if (NULL == GicConfig)
        return XST_FAILURE;

    // a
    Status = XScuGic_CfgInitialize(&InterruptController, GicConfig,
        GicConfig->CpuBaseAddress);
    if (Status != XST_SUCCESS)

```

```

    return XST_FAILURE;

Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
    (Xil_ExceptionHandler) XScuGic_InterruptHandler,
    &InterruptController);
Xil_ExceptionEnable();

// b
Status = XScuGic_Connect(&InterruptController,
    XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR,
    (Xil_ExceptionHandler) XTmrCtr_InterruptHandler,
    TimerInstancePtr);
if (Status != XST_SUCCESS)
    return XST_FAILURE;

Status = XScuGic_Connect(&InterruptController,
    XPAR_FABRIC_AXI_VDMA_RESULT_S2MM_INTROUT_INTR,
    (Xil_ExceptionHandler) (XAxiVdma_WriteIntrHandler),
    AxiVdmaInstancePtr);
if (Status != XST_SUCCESS)
    return XST_FAILURE;

// c
XScuGic_Enable(&InterruptController,
    XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR);

XScuGic_Enable(&InterruptController,
    XPAR_FABRIC_AXI_VDMA_RESULT_S2MM_INTROUT_INTR);
XScuGic_Enable(&InterruptController,
    XPAR_FABRIC_AXI_VDMA_RESULT_MM2S_INTROUT_INTR);
return XST_SUCCESS;
}

```

Wewnątrz procedury ma miejsce kilka etapów konfiguracji:

- (a) Uruchomienie kontrolera przerwań i rejestracja agenta zdarzeń, odpowiedzialnego za wstępna obsługę wszystkich zgłaszanych wyjątków.
- (b) Rejestracja wszystkich modułów logiki programowalnej, które połączone są z wejściem IRQ_F2P i których przerwania powinny być obsługiwane przez aplikację. Definiowane są również procedury odpowiedzialne za obsługę każdego zdarzenia.
- (c) Aktywacja kanałów obsługi przerwań. Wykonanie tego kroku rozpoczyna proces oczekiwania kontrolera na zdefiniowane przerwanie.

5.8.3. PetaLinux

Obsługa przerwań w systemie PetaLinux wymaga wykorzystania dedykowanych sterowników sprzętu i przeprowadzenia przy ich użyciu procesu konfiguracji. Pakiet PetaLinux udostępnia sterowniki do modułów AXI, które ich wymagają i w ramach niniejszej pracy ograniczono się do ich wykorzystania. W przypadku konieczności obsługi przerwania z niestandardowego modułu, konieczne może być dostarczenie dedykowanego mu sterownika, co wymaga specjalistycznej wiedzy z dziedziny działania systemów operacyjnych i komunikacji z urządzeniami peryferyjnymi.

Aby uzyskać dostęp do modułów zaimplementowanych w logice programowalnej, konieczna jest aktywacja tak zwanych modułów systemowych. Na etapie konfiguracji systemu operacyjnego aktywować należy poniższe opcje:

Listing 5.19. Konfiguracja modułów systemowych.

```
petalinux-config -c kernel

Device Drivers -> Userspace I/O drivers
Device Drivers -> Userspace I/O drivers -> Userspace I/O platform driver with
    generic IRQ handling
Device Drivers -> Userspace I/O drivers -> Userspace I/O platform driver with
    generic iqr and dynamic memory
```

Konieczna jest również znajomość identyfikatorów linii przerwań. Można je odczytać z poziomu SDK, po utworzeniu projektu *Board Support Package* dla wykorzystywanej konfiguracji sprzętowej. Identyfikatory linii przerwań zdefiniowane są w pliku `xparameters.h`, na przykład:

```
/* Definitions for Fabric interrupts connected to ps7_scugic_0 */
#define XPAR_FABRIC_AXI_VDMA_RESULT_MM2S_INTROUT_INTR 61
#define XPAR_FABRIC_AXI_VDMA_RESULT_S2MM_INTROUT_INTR 62
#define XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR 63
```

Wartości te mogą być również znalezione w strukturze *device tree*, generowanej przez pakiet PetaLinux na etapie parametryzacji, w której zdefiniowane są informacje o konfiguracji sprzętowej, wymagane do poprawnego działania systemu. Wymagane informacje znajdują się w pliku `subsystems/linux/configs/device-tree/pl.dtsi`. Na listingu poniżej przedstawiono fragment konfiguracji związany z modułem AXI Timer.

```
axi_timer_0: timer@42800000 {
    # ...
    compatible = "xlnx,xps-timer-1.00.a";
    interrupt-parent = <&intc>;
    interrupts = <0 31 4>;
    reg = <0x42800000 0x10000>;
    # ...
};
```

Kolejne wpisy w konfiguracji definiują informacje o sterowniku, który powinien być odpowiedzialny za obsługę modułu z poziomu procesora, modulem odpowiedzialnym za kontrolę przerwań oraz definicję zdarzeń. Ostatni wpis zawiera informację o adresie urządzenia w pamięci oraz rozmiarze tego zasobu. Definicja przerwania zawiera trzy elementy, z których kluczowa jest wartość 31. Ze względu na specyfikę formatu danych, w celu uzyskania właściwego identyfikatora przerwania, konieczne jest zwiększenie jej o 32. Uzyskany wynik – 63 – jest zgodny z definicją we wnętrzu pliku `xparameters.h`. W razie konieczności zaprojektowania dedykowanego sterownika sprzętu, wymagana jest wiedza na temat struktury `device tree` oraz zasad budowy oprogramowania tego typu. Informacje na ten temat znaleźć można we właściwych źródłach [37, 44].

Pakiet PetaLinux pozwala na dodanie do konfiguracji własnych modułów systemowych. W celu utworzenia struktury plików dla nowego modułu, wykorzystać można polecenie:

```
petalinux-create -t modules -n nazwa_modułu --enable
```

W wyniku działania polecenia utworzona zostanie struktura, którą następnie należy zmodyfikować dodając funkcjonalności sterownika. Skompilowany na etapie budowania projektu moduł znajduje się w ścieżce `/lib/modules/identyfikator-kernela/extrę` i może być uruchomiony poleceniem:

```
insmod nazwa_modułu.ko
```

Logowane przez moduł wiadomości mogą być odczytane przy użyciu polecenia `dmesg`. W celu weryfikacji poprawności konfiguracji przerwań systemowych, wykorzystać można interfejs `/proc/interrupts`. Wszystkie przerwania mogą być wypisane przy użyciu polecenia:

```
cat /proc/interrupts
```

W przypadku modułu AXI Timer, spodziewany jest wpis o treści:

```
63:          1          0  axi-timer  40
```

Potwierdza on obecność linii przerwania o identyfikatorze 63, związanej ze sterownikiem `axi-timer`, która została wywołana jeden raz w przypadku pierwszego rdzenia procesora.

Moduł systemowy odpowiedzialny jest za obsługę przerwań. W przeciwnieństwie do aplikacji `bare-metal`, nie jest możliwe bezpośrednie powiadomienie programu pracującego poza obszarem jądra systemu w chwili wystąpienia zdarzenia. Ograniczenie to wynika z dużej złożoności jądra Linuxa i konieczności obsługi przerwań w możliwie najkrótszym czasie. W przypadku wymogu wiedzy o wystąpieniu zdarzenia z poziomu aplikacji, wykorzystać można jedno z dwóch rozwiązań:

- Obsługa interfejsu UIO (ang. Userspace Input/Output) – moduł systemowy może umożliwić dostęp do urządzenia przez uchwyt wewnątrz interfejsu `/dev` systemu. Po otwarciu uchwytu przez program użytkownika, wykonać można operacje odczytu i zapisu do rejestrów urządzenia. Wystąpienie przerwania może zostać zaobserwowane przez cykliczny

odczyt rejestru odpowiedzialnego za obsługę analizowanego zdarzenia. Podobne wyniki osiągnąć można przy użyciu metod bezpośredniej komunikacji z urządzeniem dzięki znajomości jego adresu fizycznego, co opisano w rozdziale 5.3.2. Wadą obu metod jest konieczność cyklicznego odczytywania wartości rejestrów kontrolnych urządzenia (*polling*), co może ograniczać wydajność aplikacji.

- Wysłanie sygnału do programu użytkownika – moduł systemowy definiuje dedykowany sygnał i w trakcie obsługi przerwania wysyła powiadomienie o wystąpieniu zdarzenia do procesu programu użytkownika. Proces ten może przeprowadzić obsługę wystąpienia sygnału w sposób podobny do obsługi puli sygnałów systemowych. Metoda ta została opisana w pracy [45]. Konieczność znajomości identyfikatora procesu użytkownika wewnątrz procedury obsługi przerwań w module systemowym stanowi poważne ograniczenie. Konieczność przeprowadzania procesu konfiguracji przy każdym uruchomieniu aplikacji zmniejsza w dużym stopniu możliwości praktycznego wykorzystania tej metody.

Ze względu na ograniczenia obu proponowanych metod, wykorzystanie przerwań systemowych w aplikacjach uruchamianych pod kontrolą systemu operacyjnego może być utrudnione. W przypadku aplikacji, których podstawowym zadaniem jest obsługa przerwań pochodzących z modułów logiki programowej, zastosować można metodę *pollingu*. Jeśli jednak program odpowiedzialny jest za wykonanie również innych zadań, konieczne może być wykorzystanie sygnałów systemowych.

5.9. Konfiguracja algorytmu generacji tła

Algorytm generacji tła wykorzystuje funkcjonalności opisane dotychczas w rozdziale. Poniżej przedstawiono zagadnienia związane z integracją funkcjonalności w jednym projekcie.

5.9.1. Vivado

Wejściami modułu generacji tła są aktualna wartość piksela obrazu i jego wartość z poprzedniej ramki oraz piksel modelu tła wyznaczony dla ostatniej klatki. Ponadto, dostępne są sygnały wejściowe związane z parametrami pracy algorytmu. Wyjścia modułu stanowią zbiór trzech ramek obrazu, zawierający sygnały masek ruchu i przynależności do obiektów pierwszoplanowych oraz obliczany model tła.

Aby umożliwić konfigurację parametrów algorytmu zaprojektowano moduł **algorithm_parameters**. Udostępnia on wyjścia związane z każdym z parametrów i wykorzystuje interfejs AXI do komunikacji z procesorem. Sposób konfiguracji modułów tego typu opisano w rozdziale 5.3.

Buforowanie ramek obrazu wykorzystuje mechanizm AXI VDMA. Proces konfiguracji modułów tego typu opisano w rozdziale 5.4. Wykorzystano niezależne bufory do przechowywania ramki

obrazu wejściowego oraz modelu tła. Alternatywnym rozwiązaniem jest wykorzystanie jednego modułu do obsługi obu sygnałów. Aby osiągnąć możliwie maksymalną wydajność, z każdym modułem związano niezależne interfejsy **AXI_Interconnect** połączone z procesorem przy użyciu kanałów wysokiej wydajności. Trzeci moduł VDMA był odpowiedzialny za przesłanie wyników działania algorytmu do części procesorowej, w celu przeprowadzenia dalszego przetwarzania i prezentacji wyników.

Dla synchronizacji trzech strumieni wizyjnych wykorzystano moduł **AXI4-Stream Combiner**, umożliwiający połączenie wielu strumieni **AXI-Stream** w jeden, o większej szerokości. Aby zagwarantować stabilność działania aplikacji, z każdym sygnałem wejściowym modułu związano element **AXI4-Stream Data FIFO**, zapewniający kolejkę FIFO o długości sześćdziesięciu czterech elementów.

Przeciwny mechanizm zastosowano na wyjściu modułu generującego model tła – trzy ramki obrazu przesypane były w formie jednego strumienia o szerokości dziewięciu bajtów i dzielone na niezależnie strumienie przy użyciu modułu **AXI4-Stream Broadcaster**. Dzięki zastosowaniu opisywanych technik możliwa była synchronizacja sześciu niezależnych strumieni i utrzymanie stabilności działania projektu.

Konwersja obrazu z przestrzeni *RGB* do *YCbCr* odbywała się przy użyciu zaprojektowanego modułu **rgb2ycbcr**. Otrzymywał on na wejściu wektor o szerokości dwudziestu czterech elementów, zawierający kanały składowych *RGB*, a na wyjściu udostępniał wektor o tej samej szerokości, zawierający wartości składowych sygnału *YCbCr*. Operacja miała miejsce przed konwersją strumienia video do formy *AXI4-Stream*.

Operacje akwizycji sygnału z interfejsu HDMI i wysłania obrazu wynikowego do interfejsu VGA, a także do konwersji do i z postaci *AXI4-Stream* zrealizowano przy użyciu modułów dostępnych w bibliotece producenta karty ZYBO, której repozytorium dostępne jest pod adresem [46].

Wykorzystano również moduł **AXI GPIO** do obsługi kanału **hpd** związanego z wejściem HDMI. Sygnał **hpd** odpowiedzialny jest za zdefiniowanie gotowości kanału do odbioru transmisji.

5.9.2. SDK

Konfiguracja modułów VDMA opisana została w rozdziale 5.4 i w omawianym projekcie ma ten sam przebieg dla każdego z trzech elementów. Pamiętać należy o zapewnieniu niezależnych przestrzeni adresowych dla każdego zbioru buforów obrazu.

Moduł udostępniający parametry algorytmu zawiera trzy rejesty, związane z poszczególnymi wartościami. Wartości progów ruchu i tła mają charakter liczb całkowitych z zakresu od 0 do 255. Parametr bezwładności tła to liczba stałoprzecinkowa bez znaku, o jednym bicie części całkowitej i siedmiu bitach części ułamkowej. Moduł algorytmiczny współpracuje poprawnie z wartościami z zakresu od 0 do 1. Proces konfiguracji modułu przedstawiono na listingu 5.20.

Listing 5.20. Konfiguracja modułu parametryzującego generację tła.

```
#define ALPHA_REGISTER 0
#define BG_TH_REGISTER 4
#define FD_TH_REGISTER 8

// ...

Xil_Out32(XPAR_ALGORITHM_PARAMETERS_0_CTL_BASEADDR + ALPHA_REGISTER, 32); // 00100000 = 0.25
Xil_Out32(XPAR_ALGORITHM_PARAMETERS_0_CTL_BASEADDR + BG_TH_REGISTER, 25);
Xil_Out32(XPAR_ALGORITHM_PARAMETERS_0_CTL_BASEADDR + FD_TH_REGISTER, 30);
```

W prezentowanym przykładzie, zdefiniowano wartości parametrów: $\alpha = 0,25$, $T_{bg} = 25$, $T_{fd} = 30$.

Podobny przebieg ma procedura konfiguracji modułu odpowiedzialnego za sygnał `hpd`. Aby zachować zgodność ze specyfikacją interfejsu HDMI, aby zgłosić gotowość do odczytu, należy przeprowadzić następujące operacje:

1. Wymuszenie stanu niskiego kanału.
2. Zdefiniowanie kanału jako wejściowy.
3. Wymuszenie stanu wysokiego kanału.

Na listingu 5.21 przedstawiono kod związany z omawianymi operacjami.

Listing 5.21. Konfiguracja modułu sterującego kanałem `hpd`.

```
Xil_Out32(XPAR_GPIO_0_BASEADDR, 0);
Xil_Out32(XPAR_GPIO_0_BASEADDR + 4, 0);
Xil_Out32(XPAR_GPIO_0_BASEADDR, 0xFFFFFFFF);
```

Zalecana kolejność konfiguracji modułów to:

1. Moduł parametryzujący pracę algorytmu.
2. Moduł odpowiedzialny za obsługę kanału `hpd`.
3. Moduły AXI VDMA.

Ponadto, funkcjonalność aplikacji można rozszerzyć dzięki użyciu przerwań systemowych, których konfigurację opisano w rozdziale 5.8.

5.9.3. PetaLinux

Aplikacja PetaLinux związana z projektem odpowiedzialna jest za dwa niezależne zadania. Pierwszym z nich jest przeprowadzenie konfiguracji modułów logiki programowalnej oraz uruchomienie modułu algorytmicznego. Po zakończeniu tego etapu, aplikacja może zakończyć

działanie. W drugim przypadku, pracując w trybie ciągłym, program odpowiedzialny jest za monitorowanie stanu algorytmu i udostępnienie interfejsu sieciowego z wynikami.

Konfiguracja modułów obliczeniowych opiera się na założeniach zgodnych z przedstawionymi w rozdziale poświęconym programowi *bare-metal*. Elementy logiki, do komunikacji z którymi wykorzystano protokół AXI, zebrane są w strukturze `video_transmit`. Zawiera ona elementy `axi_vdma` umożliwiające współpracę z modułami VDMA oraz `axi_interface` wykorzystywane do komunikacji z pozostałymi modułami. Ponadto, przechowuje wartość uchwytu do pamięci fizycznej oraz strukturę konfiguracyjną, w której zebrano informacje związane z pracą algorytmu i parametry obrazu. Opis struktur znaleźć można na listingu 5.22.

Listing 5.22. Struktury konfiguracyjne aplikacji.

```
struct video_transmit {
    struct axi_vdma *vdma_frame_buffer;
    struct axi_vdma *vdma_background_buffer;
    struct axi_vdma *vdma_result_frame;
    struct axi_interface *hpd;
    struct axi_interface *parameters;

    struct video_config *config;

    memory_handle_t memory_handle;
};

struct vdma_framebuffer {
    vdma_buffer_addr physical_addr;
    vdma_buffer_addr virtual_addr;
};

typedef unsigned char* vdma_buffer_addr;

struct axi_vdma {
    struct axi_interface common;
    size_t num_framebuffers;
    size_t current_framebuffer_index;
    struct vdma_framebuffer *framebuffers;
};

struct axi_interface {
    unsigned int base_addr;
    virt_address virt_addr;

    char id[32];
};

typedef unsigned int* virt_address;
```

```

struct video_config {
    struct {
        int width;
        int height;
        size_t pixel_size;
        size_t framebuffer_length;
    } video;

    struct {
        unsigned char bg_th;
        unsigned char fd_th;
        double alpha;
    } algo;
};

typedef int memory_handle_t;

```

Proces konfiguracji podzielono na dwa etapy – przygotowywania środowiska aplikacji (**setup**) oraz właściwej konfiguracji modułów (**initialize**). Na listingu 5.23 przedstawiono fragment obu procesów, związany z konfiguracją modułu parametryzującego pracę algorytmu.

Listing 5.23. Konfiguracja modułu parametryzującego pracę algorytmu.

```

struct video_transmit* setup(struct application_config config) {
    struct video_transmit *video_transmit = (struct video_transmit*) malloc(sizeof(  

        struct video_transmit));
    video_transmit->memory_handle = open("/dev/mem", O_RDWR | O_SYNC);

    video_transmit->config = (struct video_config*) malloc(sizeof(struct video_config)  

        );
    if (video_transmit->config == NULL) {
        perror("config memory allocation failed.");
        exit(1);
    }
    init_config(video_transmit->config, config.image_width, config.image_height,  

        config.alpha, config.bg_th, config.fd_th);

    video_transmit->parameters = (struct axi_interface*) malloc(sizeof(struct  

        axi_interface));
    if (video_transmit->parameters == NULL) {
        perror("parameters memory allocation failed.");
        exit(1);
    }
    setup_virt_memory(video_transmit->parameters, 65535, video_transmit->memory_handle  

        , algorithm_parameters_addr);
    strcpy(video_transmit->parameters->id, "parameters");
}

```

```

// konfiguracja pozostałych zmodułów AXI
return video_transmit;
}

void initialize(struct video_transmit *video_transmit) {
    init_parameters(video_transmit->parameters, *video_transmit->config);

    // uruchomienie pozostałych modułów AXI
}

void update_parameters(struct axi_interface *interface, struct video_config config)
{
    unsigned char alpha_c = convert_alpha(config.algo.alpha);
    axi_write(interface->virt_addr, ALPHA_REGISTER, alpha_c);
    axi_write(interface->virt_addr, BG_REGISTER, config.algo.bg_th);
    axi_write(interface->virt_addr, FD_REGISTER, config.algo.fd_th);
}

static unsigned char convert_alpha(double alpha) {
    if (alpha >= 1)
        return 0b10000000;
    if (alpha <= 0)
        return 0;

    unsigned char result = 0;
    for (ssize_t bit = 6; bit >= 0; bit--)
        if ((char) (alpha * 128) >= (result | (1 << bit)))
            result |= 1 << bit;
    return result;
}

```

Procedura `setup` wykonuje operacje związane z alokacją pamięci oraz powiązaniem adresu fizycznego modułu logiki z adresem wirtualnym. Wewnątrz procedury `initialize` ma miejsce właściwa konfiguracja modułu, a więc, w przypadku elementu parametryzującego algorytm – zapis wartości parametrów do związań z nimi rejestrów. Ze względu na konieczność konwersji wartości parametru α z postaci liczby zmiennoprzecinkowej do zapisu stałopozycyjnego, zaprojektowano procedurę `convert_alpha`, która umożliwia poprawną konfigurację rejestrów modułu bez konieczności podania reprezentacji binarnej przez użytkownika. Wykorzystanie kolejnych modułów wymaga zadeklarowania ich wewnątrz struktury `video_transmit` oraz dodania wywołań związanych z nimi funkcji wewnątrz obu opisywanych procedur. Definicje procedur `setup_virt_memory` oraz `axi_write` znaleźć można w rozdziale 5.3.

Aplikacja udostępnia ponadto serwer `http` wykorzystywany do prezentacji wyników działania algorytmu oraz wykonuje cyklicznie procedurę monitorowania stanu modułów AXI. Procedury konfiguracyjne zaimplementowano w języku C, z wykorzystaniem bibliotek systemowych oraz

biblioteki `libpng`, wymaganej do zapisu wyników działania algorytmu w formie plików graficznych. Do projektu dołączono skrypt umożliwiający pobranie i budowę zewnętrznych zależności – `dependencies.sh`. Wykonać go można przy użyciu polecenia:

```
bash dependencies.sh
```

W wyniku wywołania zbudowana zostanie struktura katalogów zawierająca kod źródłowy oraz skompilowane pliki bibliotek. Dołączony do pracy projekt skonfigurowany jest w sposób wykorzystujący te pliki.

Aplikację zintegrowano z modułem C++ wykorzystującym funkcje biblioteki OpenCV w procesie indeksacji elementów na bazie maski obiektów pierwszoplanowych. Procedurę opisano w pracy [47], a użytą implementację przedstawiono na listingu 5.24. Procedura `label_buffer` umożliwia wykonanie procesu indeksacji na podstawie danych zawartych w buforze VDMA.

Listing 5.24. Moduł indeksacji obiektów pierwszoplanowych.

```
cv::Mat label(cv::Mat const foregroundMask) {
    cv::Mat labels, stats, centroids, result;
    int nccomps = cv::connectedComponentsWithStats(foregroundMask,
        labels, stats, centroids);

    std::vector<cv::Vec3b> colors(nccomps + 1);
    colors[0] = cv::Vec3b(0, 0, 0);

    for (int i = 0; i < nccomps; i++) {
        colors[i] = cv::Vec3b(rand() % 256, rand() % 256, rand() % 256);
        if (stats.at<int>(i - 1, cv::CC_STAT_AREA) < 100)
            colors[i] = cv::Vec3b(0, 0, 0);
    }

    result = cv::Mat::zeros(foregroundMask.size(), CV_8UC3);

    for (int y = 0; y < foregroundMask.rows; y++) {
        for (int x = 0; x < foregroundMask.cols; x++) {
            int label = labels.at<int>(y, x);
            result.at<cv::Vec3b>(y, x) = colors[label];
        }
    }

    return result;
}

void label_buffer(unsigned char *buffer_in, unsigned char *buffer_out, int width,
    int height) {
    cv::Mat input = cv::Mat(height, width, CV_8UC3, buffer_in);
    input = input.clone();
    cv::Mat result = label(input);
```

```
    std::memcpy(buffer_out, result.data, result.total() * result.elemSize());  
}
```

Uruchomienie aplikacji w systemie operacyjnym PetaLinux wymaga dołączenia do niego zbioru zależności:

```
petalinux-config -c rootfs  
-> Filesystem Packages  
-> base  
-> gcc-runtime-xilinx  
  * libstdc++6  
-> glibc-xilinx  
  * libc6  
  * libcidn1  
  * libthread-db1  
-> libs  
-> opencv  
  * opencv  
  * wykorzystywane żmody opencv  
-> zlib  
  * libz1  
-> libpng  
  * libpng16-16
```

Omwianą aplikację można dodać do obrazu systemu PetaLinux lub przesłać przy użyciu protokołu SSH.

Konfiguracja aplikacji wykorzystuje argumenty wiersza poleceń:

- **--image-width** liczba pikseli w linii obrazu,
- **--image-height** liczba linii w obrazie,
- **--algo-alpha** wartość parametru α ,
- **--algo-fd** wartość parametru T_{fd} ,
- **--algo-bg** wartość parametru T_{bg} ,
- **--mode** tryb pracy programu:
 - **setup** w przypadku procesu konfiguracji elementów logiki programowalnej,
 - **algo** dla pracy w trybie ciągłym.
- **--server-port** port serwera **http**,
- **--help** – wyświetlenie pełnego opisu konfiguracji aplikacji.

Wywołania dla obu trybów przedstawiono na listingu 5.25.

Listing 5.25. Sposoby uruchomienia aplikacji.

```
./background-model --mode setup --image-width 1280 --image-height 720 \
--algo-alpha 0.2 --algo-fd 30 --algo-bg 20

./background-model --mode algo --image-width 1280 --image-height 720 \
--server-port 8080
```


6. Podsumowanie

Platforma Zynq umożliwia realizację aplikacji wizyjnych z wykorzystaniem funkcjonalności układu FPGA i procesora ARM. Pozwala to na integrację strumieniowych algorytmów wizyjnych projektowanych z myślą o układach logiki programowej z oprogramowaniem pracującym na klasycznym procesorze obliczeniowym. Badany układ umożliwia również wykorzystanie systemu operacyjnego PetaLinux, powiększając zasób możliwości w porównaniu do aplikacji typu *bare-metal* o nowe funkcje – w tym komunikację sieciową, metody przechowywania danych, a także realizację zaawansowanych zadań algorytmicznych dzięki zastosowaniu zewnętrznych bibliotek.

Procesory architektury ARM są szeroko stosowane w systemach wbudowanych, i, dzięki swej popularności, zyskały wsparcie dla dużego zbioru narzędzi, w tym kompatybilność systemów operacyjnych wielu typów. W ramach pracy, zbadano możliwości wykorzystania i uruchomiono system PetaLinux oraz porównano jego funkcjonalności z systemami Ubuntu Core i podstawową wersją Linuxa. Ponadto, omówiono zagadnienie zastosowania systemu operacyjnego czasu rzeczywistego i uruchomiono aplikacje realizujące zadania w czasie rzeczywistym.

System PetaLinux oferuje dostęp do szeregu funkcjonalności, które mogą znaleźć zastosowanie w projektach związanych z przetwarzaniem obrazów i sekwencji wizyjnych. W pracy omówiono możliwości wykorzystania biblioteki OpenCV i przedstawiono proces projektowania aplikacji z jej wykorzystaniem. Zebrano również informacje teoretyczne związane z protokołem AXI, umożliwiającym komunikację pomiędzy elementami obliczeniowymi logiki programowej a programem pracującym pod kontrolą systemu operacyjnego. Protokół AXI pełni istotną rolę we współczesnych realizacjach zaawansowanych algorytmów wizyjnych.

Powszechnie spotykanymi ograniczeniami systemów wizyjnych realizowanych przy użyciu wyłącznie elementów logiki programowej są niewielka interaktywność i brak możliwości przechowywania wyników. System wizyjny realizowany w ramach pracy – segmentacja obiektów pierwszoplanowych – jest przykładem rozwiązania tych ograniczeń dzięki wykorzystaniu systemu PetaLinux. Algorytm podzielony został na dwa niezależne moduły – zrealizowany przy użyciu elementów logiki programowej moduł odpowiedzialny za generację tła, oraz pracujący pod kontrolą systemu operacyjnego program odpowiedzialny za indeksację obiektów pierwszoplanowych. Aplikacja umożliwiła również prezentację stanu oraz wyników pracy algorytmu w formie interfejsu *www*, a także udostępniała możliwość zmiany wartości parametrów algorytmu w trakcie działania.

W ramach realizacji projektu zaproponowano metodę buforowania pełnej ramki obrazu przy użyciu modułu AXI VDMA. Zaprojektowano również moduł wyznaczający różnicę dwóch kolejnych klatek obrazu, który może stanowić element składowy rozbudowanych systemów wizyjnych. W trakcie realizacji projektu napotkano na ograniczenia, w tym niewystarczającą liczbę zasobów logicznych oraz niewielką wydajność procedur algorytmicznych uruchamianych na procesorze ARM, w wyniku czego nie była możliwa realizacja pełnego algorytmu działającego w czasie rzeczywistym.

Zebrano również informacje związane z praktycznym wykorzystaniem omawianych funkcjonalności w projektach wizyjnych. Przedstawiono metody konfiguracji kolejnych modułów w ujęciu ogólnym, możliwe do wykorzystania w trakcie projektowania rozwiązań algorytmicznych na dowolny układ rodziny Zynq. Informacje te znaleźć można w rozdziale 5.

Materiał zebrany i przedstawiony w ramach niniejszej pracy może posłużyć za podstawę do realizacji zaawansowanych algorytmów wizyjnych z wykorzystaniem platformy Zynq i systemu operacyjnego PetaLinux. Przedstawione techniki mogą zostać wykorzystane do rozwoju istniejących, jak i projektowania nowych aplikacji.

System operacyjny PetaLinux pozwala na realizację projektów wizyjnych o większych, w porównaniu do aplikacji *bare-metal*, możliwościach. Dzięki wykorzystaniu interfejsu sieciowego, budować można interaktywne panele kontrolne, zawierające informacje o stanie algorytmu a także pozwalające na jego konfigurację w trakcie pracy. Dostęp do przestrzeni przechowywania danych pozwala również na archiwizację i prezentację wyników historycznych pracy algorytmu.

Zbiór przedstawionych w ramach pracy technik nie wyczerpuje możliwości badanego systemu operacyjnego. W ujęciu ogólnym, PetaLinux lub inny system operacyjny działający na układzie z rodziny Zynq pozwala na projektowanie aplikacji, które różnią się w znacznym stopniu od programów wykorzystywanych w systemach wbudowanych o ograniczonych możliwościach. Zastosowanie zaawansowanych technik projektowania aplikacji pozwala na uzyskanie efektu zbliżonego do oprogramowania używanego w życiu codziennym, na przykład dzięki wykorzystaniu protokołów komunikacji sieciowej i interfejsów graficznych.

Zaproponowany moduł generacji tła może być wykorzystywany w złożonych aplikacjach realizowanych na platformie Zynq. Konieczna jest jednak integracja rozwiązania z elementem odpowiedzialnym za komunikację przy użyciu protokołu AXI, aby wyeliminować ograniczenia napotkane w trakcie realizacji projektu, prowadzące do niestabilnego działania algorytmu. Ponadto, aplikacja odpowiedzialna za udostępnienie interfejsu sieciowego, a także komunikację z elementami logiki programowalnej i przetwarzanie obrazów przy użyciu biblioteki OpenCV może zostać w prosty sposób zintegrowana z innymi algorytmami wizyjnymi realizowanymi na platformie Zynq.

Bibliografia

- [1] Ryszard Tadeusiewicz i Przemysław Korohoda. *Komputerowa analiza i przetwarzanie obrazów*. Wydawnictwo Fundacji Postępu Telekomunikacji, 1997.
- [2] Klaus Bengler i in. „Three Decades of Driver Assistance Systems: Review and Future Perspectives”. W: *IEEE Intelligent Transportation Systems Magazine* (2014).
- [3] Gorka Velez i Oihana Otaegui. „Embedding vision-based advanced driver assistance systems: a survey”. W: *IET Intelligent Transport Systems* (2017).
- [4] J Anil i L Padma Suresh. „Literature survey on face and face expression recognition”. W: *Circuit, Power and Computing Technologies (ICCPCT), 2016 International Conference on* (2016).
- [5] K Sriram i R Havaldar. „Human detection and tracking in video surveillance system”. W: *Computational Intelligence and Computing Research (ICCIC), 2016 IEEE International Conference on* (2016).
- [6] Muddsser Hussain i in. „Multi-target tracking identification system under multi-camera surveillance system”. W: *Progress in Informatics and Computing (PIC), 2016 International Conference on* (2016).
- [7] Huiwen Gouo i in. „A novel approach for global abnormal event detection in multi-camera surveillance system”. W: *Information and Automation, 2015 IEEE International Conference on* (2015).
- [8] *Zybo Reference Manual*. Dostęp: 2017.07.02. URL: https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual#zynq_ap_soc_architecture.
- [9] *PetaLinux Tools*. Dostęp: 2017-07-04. URL: <https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html>.
- [10] *Zynq-7000 All Programmable SoC*. Dostęp: 2017-07-02. URL: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>.
- [11] *Zynq-7000 All Programmable SoC Data Sheet*. Dostęp: 2017-07-02. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

- [12] Guanwen Zhong i in. „Design of Multiple-Target Tracking System on Heterogeneous System-on-Chip Devices”. W: *IEEE Transactions on Vehicular Technology* (2016).
- [13] Maleen Abeydeera i in. „4K Real-Time HEVC Decoder on an FPGA”. W: *IEEE Transactions on Circuits and Systems for Video Technology* (2016).
- [14] Paweł Dąbal i Ryszard Pełka. „Fast pipelined pseudo-random number generator in programmable SoC device”. W: *Signals and Electronic Systems (ICSES), 2014 International Conference on* (2014).
- [15] Xilinx. *AXI Video Direct Memory Access v6.2*. Dostęp: 2017-07-04. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf.
- [16] *The SSH (Secure Shell) Remote Login Protocol*. Network Working Group.
- [17] *Open Source Computer Vision Library*. Dostęp: 2017-07-04. URL: <http://opencv.org/>.
- [18] *Vivado Design Suite*. Dostęp: 2017-07-04. URL: <https://www.xilinx.com/products/design-tools/vivado.html>.
- [19] *Xilinx Software Development Kit*. Dostęp: 2017-07-04. URL: <https://www.xilinx.com/products/design-tools/embedded-software/sdk.html>.
- [20] Jeremy Herbert. *Getting Started with the Linux Kernel and the Digilent Zynq*. Dostęp: 2017-08-11. URL: http://jeremyherbert.net/get/digilent_zybo_zynq_getting_started.
- [21] *Building stock (Xilinx) Linux For Zynq*. Dostęp: 2017-07-06. URL: <https://embeddedgreg.com/2014/04/08/step-4a-lets-build-stock-linux/>.
- [22] *Xilinx Wiki - Build Kernel*. Dostęp: 2017-07-06. URL: <http://www.wiki.xilinx.com/Build+Kernel>.
- [23] Peter Crosthwaite. *Ubuntu Core port for Digilent Zynq board*. Dostęp: 2017-08-23. URL: <https://github.com/pcrost/ubuntu-core-zynq>.
- [24] *OPEN ASYMMETRIC MULTI PROCESSING (OpenAMP)*. Dostęp: 2017-07-06. URL: <http://www.multicore-association.org/workgroup/oamp.php>.
- [25] *FreeRTOS*. Dostęp: 2017-07-06. URL: <http://www.freertos.org/>.
- [26] Adam Taylor's MicroZed Chronicles, Part 171: OpenAMP and PetaLinux Build. Dostęp: 2017-07-06. URL: <https://forums.xilinx.com/t5/Xcell-Daily-Blog/Adam-Taylor-s-MicroZed-Chronicles-Part-171-OpenAMP-and-PetaLinux/ba-p/748583>.
- [27] *OpenAMP Framework for Zynq Devices*. Dostęp: 2017-07-06. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1186-zynq-openamp-gsg.pdf.

- [28] *ARM NEON technology*. Dostęp: 2017-07-07. URL: <https://developer.arm.com/technologies/neon>.
- [29] Michael J. Flynn. „Some Computer Organizations and Their Effectiveness”. W: *IEEE TRANSACTIONS ON COMPUTERS* (1972).
- [30] *AXI™ and ACE™ Protocol Specification*. ARM.
- [31] Anthony Williams. *Język C++ i przetwarzanie współbieżne w akcji*. Helion, 2013.
- [32] James Reinders. *Intel Threading Building Blocks. Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2010.
- [33] *OpenMP Application Programming Interface*. Dostęp: 2017-07-20. OpenMP Architecture Review Board. URL: <http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf>.
- [34] *Choosing the right threading framework*. Dostęp: 2017-07-20. URL: <https://software.intel.com/en-us/articles/choosing-the-right-threading-framework>.
- [35] Philipp Kegel, Maraike Schellmann i Sergei Gorlatch. „Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores”. W: *European Conference on Parallel Processing* (2009).
- [36] Robert Love. *Jadro Linuksa. Przewodnik programisty*. Helion, 2014.
- [37] Jonathan Corbet, Alessandro Rubini i Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, 2005.
- [38] Wojciech Gumuła. *Repozytorium projektów Vivado i Petalinux*. Dostęp: 2017-08-23. URL: <https://github.com/wgml/magister>.
- [39] Tomasz Kryjak. „Implementacja zaawansowanych algorytmów przetwarzania, analizy i szyfrowania obrazów w układach reprogramowalnych”. Prac. dokt. Akademia Górnictwo-Hutnicza w Krakowie, 2012.
- [40] *Petalinux Tools Documentation*. Dostęp: 2017-08-30. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug1144-petalinux-tools-reference-guide.pdf.
- [41] *Vivado Version 2015.1 and Later Board File Installation*. Dostęp: 2017-07-16. URL: <https://reference.digilentinc.com/reference/software/vivado/board-files>.
- [42] *Zynq All Programmable SoC Linux-FreeRTOS AMP Guide*. Dostęp: 2017-08-31. URL: https://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_2/ug978-petalinux-zynq-amp.pdf.
- [43] *Open Asymmetric Multi Processing repository*. Dostęp: 2017-08-31. URL: <https://github.com/OpenAMP/open-amp>.

- [44] Xillibus. *A Tutorial on the Device Tree (Zynq)*. Dostęp: 2017-08-06. URL: <http://xillybus.com/tutorials/device-tree-zynq-1>.
- [45] Daniel Bovet i Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2002.
- [46] *Diligent Vivado library*. Dostęp: 2017-08-23. URL: <https://github.com/DiligentInc/vivado-library>.
- [47] Gary Bradski i Kaehler Adrian. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, 2016.
- [48] Haoliang Qin. *Boost Software Performance on Zynq-7000 AP SoC with NEON*. Dostęp: 2017-07-07. Xilinx. URL: <https://www.xilinx.com/support/documentation/application-notes/xapp1206-boost-sw-performance-zynq7soc-w-neon.pdf>.
- [49] *Dot Product with Neon Intrinsics*. Dostęp: 2017-07-07. URL: <https://stackoverflow.com/a/17442498>.

Dodatki

A. Spis zawartości płyty CD

Dołączona do pracy płyta CD zawiera pliki źródłowe omawianych projektów:

- **ip-repo** –moduły logiki programowalnej, wykorzystane do realizacji projektów wizyjnych:
 - **algorithm_parameters** – moduł konfiguracji parametrów algorytmu,
 - **background_model** – moduł algorytmu generacji tła,
 - **frame_difference** – moduł odejmowania dwóch ramek obrazu,
 - **frame_synchronizer** – moduł synchronizujący dwa strumienie *AXI-Stream*,
 - **rgb2ycbcr** – moduł odpowiedzialny za konwersję obrazu z przestrzeni barw *RGB* do *YCbCr*,
- **proj-background-model-petalinux** – projekt PetaLinux pozwalający na uruchomienie projektów odejmowania ramek i generacji tła,
- **proj-background-model-sdk** – aplikacje *bare-metal* oraz systemowa związane z algorytmem generacji tła,
- **proj-background-model-vivado** – projekt sprzętowy związany z algorytmem generacji tła,
- **proj-frame-difference-sdk** – aplikacje *bare-metal* oraz systemowa związane z algorytmem odejmowania ramek,
- **proj-frame-difference-vivado** – projekt sprzętowy związany z algorytmem odejmowania ramek,
- **proj-rtos-petalinux** – projekt PetaLinux pozwalający na uruchomienie systemu operacyjnego czasu rzeczywistego,
- **proj-frame-difference-vivado** – projekt sprzętowy pozwalający na uruchomienie systemu operacyjnego czasu rzeczywistego,
- **praca-diplomowa-w-gumula.pdf** – plik zawierający treść niniejszej pracy.

B. Aplikacja w architekturze NEON

Na listingach poniżej zaprezentowano implementację procedury wyznaczającej wartość iloczynu skalarnego dwóch wektorów o zadanej długości. Listing 1. zawiera implementację bazową, wykorzystującą podstawowe operacje dostępne w języku C. Kod listingu 2. wykorzystuje funkcjonalności modułu NEON w celu potencjalnego zmaksymalizowania wydajności operacji. Obie implementacje porównano z procedurą napisaną w asemblerze, wykorzystującą instrukcje koprocesora VFP, przedstawioną na listingu 3.

Sposób wykorzystania procedur w programie przedstawiono na listingu 4. Kompilacja programu wymaga aktywacji przełączników odpowiedzialnych za obsługę instrukcji NEON. Wykorzystano również techniki optymalizacji udostępniane w kompilatorze *gcc*. Polecenie kompilacji przedstawiono poniżej:

```
arm-linux-gnueabihf-gcc -Wall -O3 -mcpu=cortex-a9 -mfpu=neon -ftree-vectorize -mvectorize-with-neon-quad -mfloat-abi=hard -ffast-math -funsafe-math-optimizations -g -c -o "main.o" "main.c"
```

Listing 1. Implementacja bazowa.

```
float dot_product(float *first, float *second, unsigned int len) {
    float sum = 0.0;
    for (unsigned int i = 0; i < len; i++)
        sum += first[i] * second[i];
    return sum;
}
```

Listing 2. Implementacja w architekturze NEON. (Źródło: [48])

```
float dot_product_neon(float * restrict first, float * restrict second, unsigned int len) {
    float32x4_t vec1_q, vec2_q;
    float32x4_t sum_q = {0.0, 0.0, 0.0, 0.0};
    float32x2_t tmp[2];
    float result;
    for( int i=0; i<( len & ~3); i+=4 )
    {
        vec1_q=vld1q_f32(&first[i]);
        vec2_q=vld1q_f32(&second[i]);
        sum_q = vmlaq_f32(sum_q, vec1_q, vec2_q );
    }
    tmp[0] = vget_high_f32(sum_q);
    tmp[1] = vget_low_f32 (sum_q);
    tmp[0] = vpadd_f32(tmp[0], tmp[1]);
    tmp[0] = vpadd_f32(tmp[0], tmp[0]);
    result = vget_lane_f32(tmp[0], 0);
    return result;
}
```

```
}
```

Listing 3. Implementacja w asemblerze. (Źródło: [49])

```
float dot_product_asm(float * restrict first, float * restrict second, unsigned int
    len) {
    float net1D=0.0f;
    asm volatile (
        "vmov.f32 q8, #0.0"
        "1:"
        "subs %3, %3, #4"
        "vld1.f32 {d0,d1}, [%1]!"
        "vld1.f32 {d4,d5}, [%2]!"
        "vmla.f32 q8, q0, q2"
        "bgt 1b"
        "vpadd.f32 d0, d16, d17"
        "vadd.f32 %0, s0, s1"
        : "=w"(net1D)
        : "r"(first), "r"(second), "r"(len)
        : "q0", "q2", "q8");
    return net1D;
}
```

Listing 4. Główna procedura programu.

```
int main() {
    float first[] = {1.1, 2.2, 3.3};
    float second[] = {-1.1, -2.2, -3.3};

    float result_base = dot_product(first, second, 3);
    float result_neon = dot_product_neon(first, second, 3);
    float result_asm = dot_product_asm(first, second, 3);

    return 0;
}
```

C. Konwersja danych pomiędzy VDMA i OpenCV

Transmisja sygnału wizyjnego pomiędzy elementami logiki programowalnej a procesorem CPU jest możliwa dzięki wykorzystaniu modułu AXI VDMA. W celu odczytu obrazu z bufora lub zapisu danych do niego wykorzystano bibliotekę OpenCV. Dla zwiększenia czytelności kodu, usunięto elementy związane z konfiguracją modułu VDMA. Założono, że pierwszy bajt bufora obrazu VDMA znajduje się pod adresem `framebuffer_ptr`. Procedura `from_vdma` jest odpowiedzialna za zbudowanie macierzy typu `cv::Mat` na bazie danych znajdujących się w buforze. Funkcja `to_vdma` pełni przeciwną rolę i umożliwia zapis danych z macierzy do bufora VDMA.

Domyślnie, procedura odczytu nie kopiuje danych z komórek pamięci związanych z modulem VDMA, co pozwala na uzyskanie możliwie największej wydajności. W przypadku, gdy macierz jest modyfikowana lub przechowywana przez okres dłuższy niż czas transmisji jednej ramki obrazu, konieczne jest uzyskanie kopii danych – odpowiedzialne jest za to polecenie `image = image.clone();`.

Na listingu 5. zaprezentowano metodę konwersji sygnału wizyjnego pomiędzy elementami obliczeniowymi wykonanymi w architekturach FPGA i ARM.

Listing 5. Konwersja sygnału wizyjnego pomiędzy AXI VDMA i `cv::Mat`.

```
#include "opencv2/core/core.hpp"

cv::Mat const from_vdma(unsigned char *ptr, std::size_t width, std::size_t height,
    std::size_t bytes_per_pixel)
{
    return cv::Mat(height, width, CV_8UC(bytes_per_pixel), ptr);
}

void to_vdma(cv::Mat const &image, std::size_t bytes_per_pixel, unsigned char *ptr)
{
    assert(image.isContinuous());
    if (ptr != image.ptr())
        std::memcpy(ptr, image.ptr(),
            image.rows * image.cols * bytes_per_pixel);
}

int main(int, char**)
{
    const std::size_t width = 1280, height = 720, bytes_per_pixel = 4;
    unsigned char framebuffer_ptr[width * height * bytes_per_pixel];

    for(int i = 0; i < 10000; i++)
    {
        cv::Mat image = from_vdma(framebuffer_ptr,
            width, height, bytes_per_pixel);
    }
}
```

```
// opcjonalna kopia
image = image.clone();

algorithm(image);

to_vdma(image, bytes_per_pixel, framebuffer_ptr);

await_next_frame();
}

return 0;
}
```