AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA AUTOMATYKI I INŻYNIERII BIOMEDYCZNEJ

Praca dyplomowa magisterska

Implementacja algorytmu śledzenia obiektów w heterogenicznym układzie Zynq

Implementation of an object tracking algorithm in heterogeneous Zynq device

Autor: *Tomasz Meresiński*
Kierunek studiów: *Automatyka i Robotyka*
Opiekun pracy: *dr inż. Tomasz Kryjak*

Kraków, 2016

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 zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram 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.

Serdecznie dziękuję wszystkim osobom wspierającym mnie w pisaniu niniejszej pracy

Streszczenie

Śledzenie obiektów jest bardzo ważnym zadaniem realizowanym w wielu systemach wizyjnych. Bardzo często algorytmy te implementuje się z wykorzystaniem rekonfigurowalnych układów FPGA. Ostatnio jednak, coraz popularniejsze stają się układy hybrydowe – łączące logikę programowalną oraz szybki procesor. Przykładem może być Zynq – *system-on-a-chip* firmy *Xilinx* składający się z układu FPGA Artix-7 oraz procesora ARM Cortex-A9.

Podstawowym celem pracy była implementacja odpowiednio dobranego algorytmu śledzenia z wykorzystaniem tego układu. Przeanalizowano trzy metody rozwiązywania takiego zadania – KLT, *mean shift* oraz filtr cząsteczkowy zwracając baczną uwagę na możliwość wykonania podziału na część sprzętową oraz programową. Wybrany został algorytm filtru cząsteczkowego, który najlepiej wpisywał się w koncepcję implementacji sprzętowo-programowej. W pracy omówiony został proces podziału algorytmu oraz jego implementacja za pomocą *Vivado Design Suite*.

Dodatkowo, przedstawione zostały podstawowe techniki,jakimi należy się posługiwać w celu komunikacji pomiędzy komponentami układu Zynq, czyli przerwania oraz interfejs AXI4.

Abstract

Object tracking is one of the key features in many machine vision applications. Usually, those algorithms are implemented in FPGA circuits. Recently, so-called systems on a chip (SoC) integrating programmable logic and a fast processor core are getting more and more popular. Zynq produced by Xilinix is a good example of a such device.

The main goal of this master thesis was to implement one of well-known object tracking algorithms for that platform. Three methods were analysed – KLT, mean shift and particle filter. The last one presented the biggest opportunities from dividing it into a software and hardware part. Therefore it was used in this project. The majority of this work contains a description developing the system with Vivado Design Suite.

Additionally, two methods of communication between FPGA and processor in Zynq: AXI4 and interrupts were also described.

Spis treści

1. Wstęp

Śledzenie obiektów jest jednym z podstawowych zadań, jakie są wykonywane w systemach analizy obrazu. Dla wielu osób może być niejasne, czym się to różni od ciągłego wykrywania tego samego obiektu. Mianowicie, podczas śledzenia dostajemy dodatkową informację, że dokładnie ten sam obiekt, który był widoczny wiele klatek obrazu wcześniej w pewnym miejscu, obecnie znajduje się tutaj. Stwarza to na przykład możliwość wyznaczenia trajektorii obiektów, co w przypadku samego wykrywania jest niemożliwe. Warto też dodać, że metody śledzenia charakteryzują się zwykle niższą złożonością obliczeniową niż metody detekcji obiektów.

Metody śledzenia znajdują zastosowanie w:

– automatycznych systemach monitoringu,

– interfejsach człowiek – komputer (na przykład rozpoznaniu gestów),

– monitorowaniu i analizie ruchu obiektów (na przykład mrówek w mrowisku albo ruchu ulicznego),

– automatycznym sterowaniu samochodami – niezbędne jest tam wykrywanie poruszających się przeszkód oraz analiza trajektorii ich ruchu,

– kompresji obrazu.

Często jako obiekty mogą więc występować:

– człowiek (jako postać),

– ręka,

– samochód,

– poruszające się przeszkody.

Złożoność większości typowych algorytmów śledzenia nie jest bardzo duża – były skutecznie testowane już w latach 90. XX wieku. Pewne trudności może jednak sprawiać ich uruchamianie w czasie rzeczywistym, co z uwagi na rosnące standardowe rozdzielczości obrazu oraz liczbę klatek na sekundę wymaga coraz większych zasobów obliczeniowych. Dodatkowo, coraz częściej wykorzystywane są obrazy z wielu kamer równocześnie, co stwarza możliwość określania pozycji obiektu w trójwymiarze, ale jest przy tym bardziej złożone. W przypadku chęci stworzenia w pełni automatycznego układu niezbędne jest dodanie do niego wykrywania obiektów. Dzięki temu nie trzeba ręcznie wprowadzać, co ma być przez algorytm śledzone, powoduje jednak kolejne zwiększenie wymagań sprzętowych.

Ze względu na różnorodność algorytmów śledzenia, do realizacji można użyć praktycznie dowolnej platformy używanej do zastosowań wizyjnych. Możliwe do wykorzystania są na przykład standardowe, używane w komputerach stacjonarnych lub przenośnych, procesory. W przypadku chęci stworzenia układu wbudowanego, można

zastosować procesory ARM (ang. *Advanced RISC Machine*). Projekt na takiej platformie będzie najprawdopodobniej najszybszy w utworzeniu, jednak może nie być w stanie obsłużyć niektórych algorytmów w czasie rzeczywistym. Popularne do takiego zastosowania są także procesory DSP (ang. *Digital Signal Processor*). Są to układy wyspecjalizowane do zadań cyfrowego przetwarzania sygnałów, co objawia się przez możliwość bardzo szybkiego wykonywania niektórych operacji.

Śledzenie obiektów można z powodzeniem wykonywać także na platformach sprzętowych, na przykład w układach FPGA (ang. *Field Programmable Gate Array*). Takie podejście pozwala na wydajne uruchamianie algorytmów silnie równoległych albo z architekturą potokową.

Często jednak niektórych części algorytmów nie da się efektywnie w taki sposób zaprojektować. Jednym z możliwych rozwiązań tego problemu jest wytworzenie w strukturze układu *softprocessora*, na przykład *Microblaze'a* firmy *Xilinx*, co pozwala na łatwiejsze implementowania i wykonywanie sekwencyjnych fragmentów algorytmów.

Zynq jest to nowy *system-on-a-chip* (SoC) firmy Xilinx, który łączy w sobie układ FPGA oraz dwurdzeniowy procesor ARM. Otrzymuje się więc rozwiązanie, które stwarza możliwość łatwego łączenia zalet logiki programowalnej oraz przetwarzania sekwencyjnego. Rozwiązanie to jest także szybsze niż konkurencyjne zawierające *softprocessor*.

Połączenie to zaciekawiło autora na tyle, że zdecydował się uruchomić w układzie Zynq wybrany algorytm śledzenia. Pierwszym z celów był więc wybór takiej metody, która pozwala efektywnie wykorzystać zarówno cześć FPGA, jak i procesor znajdujący się w układzie. Przeprowadzono więc analizę kilku metod wybranych na podstawie popularności oraz faktu istnienia implementacji sprzętowo-programowych.

Podczas analizy okazało się, że najlepiej cechy te odzwierciedla filtr cząsteczkowy (ang. particle filter), w którym podstawowe obliczenia wykonywane są za pomocą dużej liczby cząstek, które można realizować w sposób równoległy. Za to procedura analizy danych od nich otrzymywanych jest już typowo sekwencyjna. Podstawowym celem była więc implementacja tak zaprojektowanego filtru cząsteczkowego w układzie Zynq oraz analiza stworzonego rozwiązania.

Układ pracy

Praca rozpoczyna się rozdziałem opisującym trzy popularne algorytmy śledzenia obiektów – KLT, *mean shift* oraz filtry cząsteczkowe.

Tematem rozdziału drugiego jest omówienie budowy układów Zynq. Opisano więc w nim budowę wszystkich jego części składowych – logiki reprogramowalnej FPGA (PL) oraz procesorów ARM (PS). Dodatkowa część została poświęcona magistrali AXI4, która jest podstawowym sposobem komunikacji pomiędzy PL, a PS.

W rozdziale trzecim przedstawiono analizę opisanych w pierwszym rozdziale algorytmów pod kątem możliwości wykorzystania zalet układów Zynq podczas ich implementacji. Zamieszczono tam także omówienie kilku zaprezentowanych już w literaturze projektów dotyczących śledzenia.

Rozdział czwarty zawiera szczegółowe omówienie architektury zrealizowanego projektu. Wyraźnie zaznaczono podział poszczególnych zadań pomiędzy część sprzętową oraz programową.

W rozdziale piątym znajduje się krótka analiza uzyskanych wyników. Pokazane jest również, jak niektóre parametry wpływają na jakość śledzenia. Praca kończy się podsumowaniem oraz wskazaniem potencjalnych kierunków rozwoju aplikacji.

2. Przegląd wybranych algorytmów śledzenia obiektów

Intuicyjnie rzecz ujmując, śledzenie wybranego obiektu polega na określaniu jego pozycji w kolejnych ramkach obrazu. W literaturze przedmiotu można znaleźć wiele różnych algorytmów służących do rozwiązania tego zadania, jednak każda z tych metod posiada pewne dodatkowe założenia w stosunku do śledzonego obiektu. Na początku tego rozdziału zostaną zaprezentowane podstawowe cechy opisujące obiekt, a następnie pokazane będą trzy wybrane algorytmy śledzenia: KLT, *mean shift* oraz filtr cząsteczkowy.

2.1. Obiekt

2.1.1. Reprezentacja

Pierwszym pytaniem, jakie projektant systemu śledzenia musi sobie zadać, to jaki będzie najlepszy sposób reprezentacji obiektu. Najpopularniejsze z nich to [1] (rysunek 2.1):

– Proste kształty geometryczne (elipsa, prostokąt),

– Punkt,

– Kontur,

– Sylwetka,

– Model łączony, złożony z kilku różnych połączonych ze sobą reprezentacji.

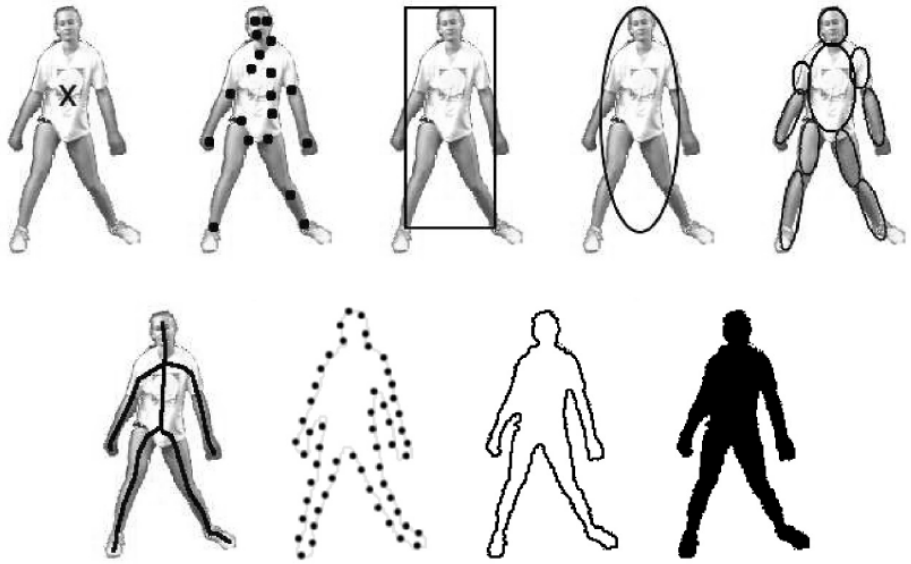
Najbardziej ogólną metodą jest śledzenie za pomocą prostych kształtów geometrycznych. Pomimo prostej reprezentacji, jest to zwykle bardzo dobre przybliżenie naszego celu. Dzięki niewielkiemu skomplikowaniu, algorytmy wykorzystujące tę reprezentację są stosunkowo szybkie.

Pozostałe możliwości pozwalają za to lepiej wykorzystywać pewne cechy charakterystyczne, jak dokładny kształt albo wielkość.

2.1.2. Kryterium śledzenia

Wybór kryterium śledzenia (ang. *tracking feature*) sprowadza się do odszukania pewnej cechy, dzięki której będzie możliwe odróżnienie obiektu od tła. Tak samo jak przy wyborze reprezentacji, odpowiedni dobór cechy pozwala skuteczniej wykorzystać wiedzę o obiekcie, albo przeciwnie, nie ograniczać się tylko do konkretnej gamy obiektów. Najczęściej spotyka się następujące kryteria: [1]

– Kolor,



Rys. 2.1. Różne sposoby reprezentacji obiektu (źródło [1])

– Krawędzie,

– Przepływ optyczny – pole opisujące przemieszenia poszczególnych pikseli w obrazie,

– Tekstura – zbiór bardziej skomplikowanych cech charakterystycznych pewnego obiektu.

W tym przypadku, najbardziej ogólnym wyborem jest kolor obiektu, ponieważ bardzo rzadko jest on zmienny. Pewnym utrudnieniem tutaj mogą być różnice w „postrzeganiu" koloru przez kamerę, związane na przykład ze zmianą oświetlenia. Z tego powodu często stosuje się tutaj inne przestrzenie barw niż RGB, na przykład YCbCr ([2]) albo HSV (ang. *Hue Saturation Value*) ([3]), a matematyczną reprezentacją będzie odpowiedni histogram.

2.2. Opis wybranych algorytmów

2.2.1. KLT

KLT to jeden ze starszych algorytmów śledzenia obiektów. Jego podstawowa wersja wykorzystuje reprezentację punktową, a kryterium śledzenia jest przepływ optyczny. Całkowicie ignorowany jest kolor obiektu, ponieważ algorytm operuje na obrazie w skali szarości. Nazwa pochodzi od nazwisk jego twórców: Takeo Kanade, Bruce'a Lucasa oraz Carlo Tomasi. Jego pierwsza wersja została zaprezentowana w pracy [4] z roku 1981, a pewne ulepszenia zostały opisane w artykule[5] z roku 1991.

Skrócony opis algorytmu

Podstawowym zadaniem jest znalezienie takich przesunięć ξ oraz η, aby spełniona była następująca równość (przepływ optyczny):

$$I\left(x, y, t+\tau\right) = I\left(x-\xi, y-\eta, t\right) \tag{2.1}$$

gdzie:

x – położenie punktu na osi x

y – położenie punktu na osi y

t – czas

τ – odstęp czasu

ξ – przesunięcie obiektu wzdłuż osi x

η – przesunięcie obiektu wzdłuż osi y

$I(x, y, t)$ – pewna funkcja (reprezentująca natężenie światła w punkcie (x, y) w czasie t)

W dalszej części przyjmijmy $\boldsymbol{d} = (\xi, \eta)$, $\boldsymbol{x} = (x, y)$, $J(\boldsymbol{x}) = I(x, y, t + \tau)$.

Oczywistym jest, że w rzeczywistych przypadkach równość ta nie zostanie nigdy spełniona – natężenie światła będzie się zmieniało przykładowo ze względu na zróżnicowany poziom oświetlenia. Z tego powodu należy przeprowadzić poszukiwania takiego $\boldsymbol{d}$ w pewnym obszarze $\mathcal{W}$, aby zminimalizować błąd dany wzorem:

$$\epsilon = \int_{\mathcal{W}} [I(\boldsymbol{x} - \boldsymbol{d}) - J(\boldsymbol{x})]^2 d\boldsymbol{x} \tag{2.2}$$

Ze względu na fakt, że dwie kolejne ramki obrazu niewiele się różnią od siebie, można rozwinąć funkcję $I(\boldsymbol{x} - \boldsymbol{d})$ za pomocą szeregu Taylora:

$$I(\boldsymbol{x} - \boldsymbol{d}) = I(\boldsymbol{x}) - \boldsymbol{g} \cdot \boldsymbol{d}$$

gdzie:

$\boldsymbol{g}$ – gradient funkcji I w punkcie $\boldsymbol{x}$

Po pewnych przekształceniach można otrzymać, że wartość $\boldsymbol{d}$ minimalizująca równanie (2.2) jest rozwiązaniem następującego równania liniowego:

$$G\boldsymbol{d} = \boldsymbol{e} \tag{2.3}$$

gdzie:

G – macierz dana wzorem: $G = \int_{\mathcal{W}} \boldsymbol{g}\boldsymbol{g}^T d\boldsymbol{x}$

$\boldsymbol{e}$ – wektor dany wzorem: $\boldsymbol{e} = \int_{\mathcal{W}} (I - J)\, \boldsymbol{g} d\boldsymbol{x}$

Wyliczony w ten sposób punkt $\boldsymbol{x} + \boldsymbol{d}$ jest szukaną przez nas pozycją obiektu w kolejnej ramce obrazu. Przedstawione powyżej wyprowadzenie sprawdza się dla ciągłej w przestrzeni funkcji natężenia światła. Jednak w przypadku komputerowego śledzenia obiektów nie jesteśmy w stanie takiej funkcji otrzymać. Z tego powodu wykorzystuje się dyskretną wersję tego równania, w której całki są zamieniane na odpowiadające im sumy.

Pełny algorytm prezentuje się więc następująco:

1. Konwersja obrazu do skali szarości (na przykład obliczenie składowej Y z przestrzeni YCbCr).
2. Określenie wielkości okna $\mathcal{W}$, zazwyczaj kwadrat o długości boku równym kilka pikseli.
3. Obliczenie gradientu $\boldsymbol{g}$ w każdym punkcie okna.
4. Wyznaczenie macierzy G oraz wektora $\boldsymbol{e}$.
5. Rozwiązanie równania liniowego (2.2), aby otrzymać przesunięcie obiektu $\boldsymbol{d}$.
6. Wyznaczenie nowej pozycji obiektu równej $\boldsymbol{x} + \boldsymbol{d}$.

2.2.2. *Mean shift*

Mean shift jest to nieparametryczny algorytm służący do znajdywania maksimum funkcji gęstości prawdopodobieństwa. Ze względu na swoją wydajność, znalazł swoje miejsce jako jeden z najpopularniejszych algorytmów w wizyjnym śledzeniu obiektów. Pierwszy opis jego użycia w tym celu znajduje się w pracy [6]. Wykorzystano tam reprezentację za pomocą elipsy, a śledzoną cechą był kolor obiektu (dokładniej jego histogram).

Skrócony opis algorytmu

Pierwszym punktem algorytmu jest uzyskanie matematycznego opisu śledzonego obiektu, co należy zrobić w pierwszej ramce obrazu. Wykorzystuje się do tego model celu dany wzorem:

$$\hat{\boldsymbol{q}} = \{\hat{q}_u\}_{u=1\cdots m} \tag{2.4}$$

gdzie:

m – rozmiar histogramu

Model ten musi spełniać założenia funkcji gęstości prawdopodobieństwa, więc:

$$\sum_{u=1}^{m} \hat{q}_u = 1 \tag{2.5}$$

Niech $\{\boldsymbol{x}_i^*\}_{i=1\ldots n}$ będą lokalizacjami wszystkich pikseli znajdującymi się wewnątrz śledzonego obiektu, wcześniej znormalizowanymi do koła jednostkowego o środku w punkcie $(0,0)$. Normalizacja ta jest niezbędna, aby uniezależnić wszystkie dalsze obliczenia od kształtu oraz pozycji obiektu. Wtedy poszczególne składowe modelu można obliczyć za pomocą wzoru:

$$\hat{q}_u = C \sum_{i=1}^{n} k\left(\|\boldsymbol{x}_i^\star\|^2\right) \delta\left[b\left(\boldsymbol{x}_i^\star\right) - u\right] \tag{2.6}$$

gdzie:

δ – delta Kroneckera

$k(x)$ – profil wybranego jądra (funkcja, która daje większą wagę punktom znajdującym się bliżej środka obiektu)

$b(\boldsymbol{x})$ – funkcja zwracająca, w której części histogramu znajduje się punkt $\boldsymbol{x}$

C – stała normalizująca tak, aby prawdziwa była zależność (2.5).

Model celu jest więc w rzeczywistości trochę zmodyfikowanym histogramem, do którego z większą wagą wliczane są punkty znajdujące się blisko środka obiektu.

W kolejnych ramkach obrazu należy wyznaczać kandydatów do celu danych wzorem:

$$\hat{\boldsymbol{p}}(\boldsymbol{y}) = \{\hat{p}_u(\boldsymbol{y})\}_{u=1\ldots m} \tag{2.7}$$

gdzie:

$\boldsymbol{y}$ – znormalizowana pozycja kandydata

Podobnie, jak w przypadku modelu, suma wszystkich składowych musi się równać 1:

$$\sum_{u=1}^{m} \hat{p}_u = 1 \tag{2.8}$$

Poszczególne części składowe kandydata do celu dane są wzorem:

$$\hat{p}_u(\boldsymbol{u}) = C_h \sum_{i=1}^{n_h} k\left(\| \frac{\boldsymbol{y} - \boldsymbol{x}_i}{h} \|^2 \right) \delta\left[b(\boldsymbol{x}_i) - u \right] \tag{2.9}$$

gdzie:

h – skala obiektu

C_h – stała normalizująca

Funkcja jądra może być praktycznie dowolna, jednak ze względów praktycznych najczęściej wykorzystuje się jądro z profilem Epanechnikova dane wzorem:

$$k(x) = \begin{cases} 1 - x & \text{jeśli } x \leq 1 \\ 0 & \text{w przeciwnych wypadkach} \end{cases} \tag{2.10}$$

Jakość kandydata określana jest za pomocą jego odległości od modelu:

$$d(\boldsymbol{y}) = \sqrt{1 - \rho\left[\hat{\boldsymbol{p}}(y), \hat{\boldsymbol{q}} \right]} \tag{2.11}$$

gdzie:

$$\hat{\rho}(\boldsymbol{y}) = \rho\left[\hat{\boldsymbol{p}}(y), \hat{\boldsymbol{q}} \right] \tag{2.12}$$

to współczynnik Bhattacharyya pomiędzy $\boldsymbol{p}$ oraz $\boldsymbol{q}$.

Obiekt w kolejnej ramce obrazu znajduje się więc w takim punkcie $\boldsymbol{y}$, gdzie współczynnik Bhattacharyya jest największy. W tym celu wykorzystuje się właściwy algorytm optymalizacji *mean shift*.

Bez wchodzenia w szczegóły wyprowadzeń, metoda ta przedstawia się następująco:

1. Ustalenie punktu początkowego $\hat{\boldsymbol{y}}_0 = \hat{\boldsymbol{x}}$
2. Obliczenie wag poszczególnych części histogramu zgodnie ze wzorem:
$$w_i = \sum_{u=1}^{m} \sqrt{\frac{\hat{q}_u}{\hat{p}_u(\hat{\boldsymbol{y}}_0)}} \delta\left[b(\boldsymbol{x}_i) - u \right] \tag{2.13}$$
3. Odszukanie kolejnej lokalizacji dla kandydata zgodnie ze wzorem:
$$\hat{\boldsymbol{y}}_1 = \frac{\sum_{i=1}^{n_h} \boldsymbol{x}_i w_i}{\sum_{i=1}^{n_h} w_i} \tag{2.14}$$
4. Zakończenie poszukiwań, jeśli odległość pomiędzy punktami $\hat{\boldsymbol{y}}_1$ oraz $\hat{\boldsymbol{y}}_0$ jest mniejsza niż ustalony wcześniej próg ϵ. Wartość ϵ ustala się zazwyczaj tak, by punkty znajdowały się wewnątrz tego samego piksela.
5. W przeciwnym wypadku podstawienie $\hat{\boldsymbol{y}}_0 = \hat{\boldsymbol{y}}_1$ oraz powrót do kroku 2.

Aby dodatkowo uzyskiwać informację o zmieniającym się rozmiarze obiektu, przeprowadza się powyższy algorytm trzykrotnie dla różnych wartości współczynnika h. Niech h_{prev} będzie skalą w poprzedniej ramce. Wtedy zazwyczaj przeszukuje się następujące skale: $0.9h_{prev}$, h_{prev} oraz $1.1h_{prev}$. Niech h_{opt} będzie wynikiem, dla którego osiągnięto najwyższy współczynnik podobieństwa modelu do kandydata. Wtedy za skalę w obecnej ramce uznaje się wartość: $h_{new} = 0.1h_{opt} + 0.9h_{prev}$. Tak przeprowadzane obliczenia rozmiaru pozwalają na wykorzystywanie mniejszych zasobów obliczeniowych.

2.2.3. Filtry cząsteczkowe

Filtry cząsteczkowe (ang. *particle filters*), inaczej sekwencyjne metody Monte Carlo, reprezentują kolejną grupę ogólniejszych algorytmów, które znalazły swoje zastosowanie w dziedzinie śledzenia obiektów. W przeciwieństwie do poprzednio opisywanych, są one oparte na prawdopodobieństwie znajdowania się obiektu w danym punkcie. Dodatkowo, wykorzystują one losowe rozmieszczanie swoich cząsteczek zgodnie z pewnym rozkładem prawdopodobieństwa, więc są niedeterministyczne. Wykorzystywany tutaj algorytm ma swoje źródło w pracy [7]. Obiekt jest w nim reprezentowany za pomocą pewnego dowolnego, prostego kształtu (elipsa bądź prostokąt), a śledzoną cechą jest kolor.

Skrócony opis algorytmu

Załóżmy, że śledzony obiekt porusza się zgodnie z pewnym równaniem stanu i jest obserwowany za pomocą niezależnego procesu. Przyjmijmy, że $\boldsymbol{x_t}$ reprezentuje ukryty, prawdziwy stan obiektu w chwili t, a $\boldsymbol{y_t}$ to ten obserwowany w chwili t. Algorytm opiera się o wyprowadzoną z twierdzenia Bayesa zależność:

$$p\left(\boldsymbol{x}_{t+1}|\boldsymbol{y}_{0:t+1}\right) \propto p\left(\boldsymbol{y}_{t+1}|\boldsymbol{x}_{t+1}\right) \int_{\boldsymbol{x}_t} p\left(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t\right) p\left(\boldsymbol{x}_t|\boldsymbol{y}_{0:t}\right) d\boldsymbol{x}_t \tag{2.15}$$

gdzie: zapis $\boldsymbol{x}_{0:t}$ oznacza wektor $(\boldsymbol{x}_0, \cdots, \boldsymbol{x}_t)$. Lewa strona tego równania to rozkład prawdopodobieństwa, że szukany obiekt znajduje się w miejscu $\boldsymbol{x}_{t+1}$ wiedząc, że pomiary dotychczas wskazywały $\boldsymbol{y}_{0:t+1}$. Wartość oczekiwana tego rozkładu reprezentuje najbardziej prawdopodobną pozycję, którą przyjmuje się jako wynik w zadaniu śledzenia.

Podstawą do rozwiązania powyższego równania w niniejszym algorytmie jest metoda Monte Carlo, w której skomplikowane problemy przybliżane są za pomocą skończonego zbioru M losowo rozmieszczonych cząsteczek $\{x_t^m\}_{m=1\cdots M}$. W tym przypadku, przybliżane jest prawdopodobieństwo $p\left(\boldsymbol{x}_{t+1}|\boldsymbol{y}_{0:t+1}\right)$, a cząsteczki reprezentują możliwe lokalizacje śledzonego obiektu. Aby możliwe było skorzystanie ze wzoru (2.15) należy przekształcić elementy z prawej strony równania na operacje na cząsteczkach.

Rozkład $p\left(\boldsymbol{x}_t|\boldsymbol{y}_{0:t}\right)$ jest wynikiem działania algorytmu w poprzedniej ramce obrazu, więc jego przybliżenie w każdej ramce obrazu poza pierwszą jest dostępne. W pierwszej ramce należy przyjąć z góry pewien odpowiednio dobrany rozkład – zazwyczaj jednostajny albo normalny. W każdej iteracji algorytmu występuje losowanie pozycji cząsteczek zgodnie z tym rozkładem – w pierwszej rozmieszcza się je w całej przestrzeni rozwiązań, a w kolejnych wybierane są z już istniejących.

Prawdopodobieństwo $p\left(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t\right)$ reprezentowane jest w źródłowej pracy za pomocą ruchu cząsteczek zgodnie z równaniem stanu obiektu danym wzorem:

$$\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t + B\boldsymbol{x}_{t-1} + v_t \tag{2.16}$$

gdzie: $v_t \sim \mathcal{N}\left(0, C\right)$ jest to pewna losowa wartość wygenerowana z rozkładu normalnego o wartości oczekiwanej równej 0. Jako stan obiektu przyjmuje się zazwyczaj jego lokalizację, czasami wraz z rozmiarem. Wartości macierzy A, B ustalone są tak, aby przedstawiały model ruchu o stałej prędkości, a więc $A = 2I$ oraz $B = -I$, a C można traktować jako parametr algorytmu.

Nie jest to jedyna używana metoda realizacji tego etapu. Przykładowo w pracy [8] proponowanym rozwiązaniem jest ruch cząstek zgodny z normalnym rozkładem prawdopodobieństwa. Takie podejście oznacza, że nie przewidujemy, że ruch obiektu będzie nadal trwał w tym samym kierunku, tylko zakładamy możliwość nagłej jego zmiany.

Kolejną czynnością jest przypisanie każdej cząsteczce wagi, która reprezentuje w ten sposób prawdopodobieństwo $p\left(\boldsymbol{y}_{t+1}|\boldsymbol{x}_{t+1}\right)$. Jest ono zazwyczaj określane za pomocą wzoru:

$$w_i = ke^{-\Lambda D^2} \tag{2.17}$$

gdzie:

- k – stała normalizująca sumę wag do 1,
- Λ – pewien ustalony parametr,
- D – dystans.

Najczęściej stosowanym w tym algorytmie dystansem jest podobieństwo histogramów w cząsteczkach z bazowym, czyli tym obliczonym w pierwszej ramce obrazu. W celu jego obliczenia można przykładowo zastosować wykorzystywany w algorytmie *mean shift* współczynnik Bhattacharrya – wzór (2.12).

Obliczone wagi wraz z lokalizacją cząsteczek przybliżają szukany we wzorze (2.15) rozkład prawdopodobieństwa. W ostatnim kroku algorytmu wagi wykorzystywane są więc do obliczenia najbardziej prawdopodobnej pozycji śledzonego obiektu zgodnie ze wzorem:

$$\boldsymbol{x}_{t+1} = \sum_{i=1}^{M} w_{t+1}^i \boldsymbol{x}_{t+1}^i \tag{2.18}$$

oraz jako prawdopodobieństwo wylosowania cząsteczki podczas etapu wyboru.

Ogólnie rzecz ujmując, cząsteczka jest obiektem reprezentującym pewną lokalizację w przestrzeni rozwiązań. W rozpatrywanym przez nas przypadku będą to współrzędne na obrazie wraz z wysokością oraz szerokością. Dodatkowo, każdej z nich można przypisać wartość pewnej funkcji, która potem przekształcana jest w wagę. Podczas śledzenia obiektu za pomocą opisywanego algorytmu będzie to podobieństwo aktualnego histogramu oraz bazowego.

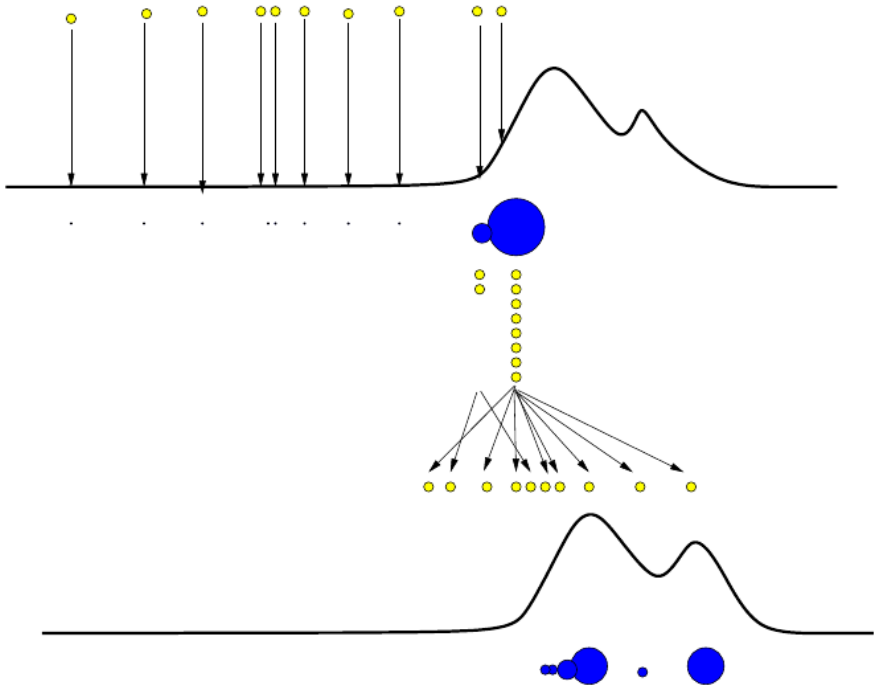
Podsumowując, w pierwszej ramce obrazu do wykonania są dwa zadania:

1. Obliczenie histogramu śledzonego obiektu.
2. Rozmieszczenie zgodnie z przewidywanym rozkładem prawdopodobieństwa cząsteczek w pobliżu obiektu. Najczęściej stosuje się rozkład jednostajny albo normalny.

W kolejnych ramkach obrazu algorytm prezentuje się następująco:

1. Przewidywanie: przesunięcie cząsteczek zgodnie z równaniem stanu – wzór (2.16).
2. Obliczenie histogramu w lokalizacji każdej cząsteczki ze zbioru.
3. Ważenie: obliczenie wagi każdej z cząsteczek zgodnie ze wzorem (2.17).
4. Wyjście: obliczenie najbardziej prawdopodobnego stanu śledzonego obiektu za pomocą wzoru (2.18)
5. Wybór: wybranie M cząstek zgodnie z prawdopodobieństwem określonym poprzez obliczone wagi.

Graficznie etapy te zaprezentowano na rysunku 2.2, gdzie za pomocą filtru cząsteczkowego poszukiwane są maksima zmiennej w czasie funkcji. Pierwszy etap przedstawia ważenie cząsteczek. Największe wagi zostały przypisane tym, przy których wartość funkcji była największa, co zostało pokazane przy pomocy wielkości niebieskich cząsteczek. Następnie następował wybór, a więc losowanie cząsteczek z prawdopodobieństwem określonym przez



Rys. 2.2. Działanie filtru cząsteczkowego (źródło [9])

wagi. Na rysunku można zauważyć, że wylosowane podczas tego etapu zostały tylko dwie z nich – największa osiem razy, a mniejsza dwa. Nowy zbiór cząsteczek został następnie losowo rozrzucony w swoim otoczeniu, co prezentuje etap przewidywania. Można zauważyć, że już po jednym etapie przestano rozpatrywać obszary cechujące się małą wartością funkcji, a większość cząsteczek znajdowała się w najbliższym otoczeniu maksimów.

3. Platforma sprzętowa ZYBO

W tym rozdziale krótko omówiono budowę platformy sprzętowej, na której zrealizowano opisywany projekt. Jest to płytka uruchomieniowa Digilent ZYBO oparta o układ Zynq-7000 firmy Xilinx, a dokładniej o jego najmniejszy model - Z-7010. Zynq jest to rodzina SoC (ang. *System-on-a-chip*), w której układy złożone są z dwóch głównych części: pierwszą jest logika reprogramowalna, czyli układ FPGA, a drugą tzw. system procesorowy (PS – ang. *processing system*), którego podstawowym elementem jest procesor ARM wzbogacony o układy wspierające komunikację. Możliwość wykorzystania takiego połączenia była podstawowym kryterium doboru algorytmu śledzenia, więc bardzo ważnym jest poznać cechy charakterystyczne jego poszczególnych składowych.

3.1. FPGA – PL

Pierwszą częścią składową układów z rodziny Zynq jest układ FPGA, czyli logika programowalna (ang. *programmable logic*). Z punktu widzenia projektanta, jest to bardzo duża macierz bloków programowalnych, których konfiguracja może być wielokrotnie zmieniana. Takie rozwiązanie pozwala na projektowanie bardzo szybkich układów logicznych pobierających przy tym mało mocy. Wielokrotna konfiguracja pozwala na zmniejszenie czasu projektowania systemu w porównaniu z konkurencyjnymi układami ASIC (ang. *Application Specific Integrated Circuit*), gdzie zmiana każdego elementu wymaga produkcji całego układu na nowo [10].

W dalszej części zostaną przedstawione pewne szczegóły budowy układów FPGA firmy Xilinx zastosowanych w wykorzystywanym w projekcie układzie Zynq, które odpowiadają rodzinie układów Artix-7.

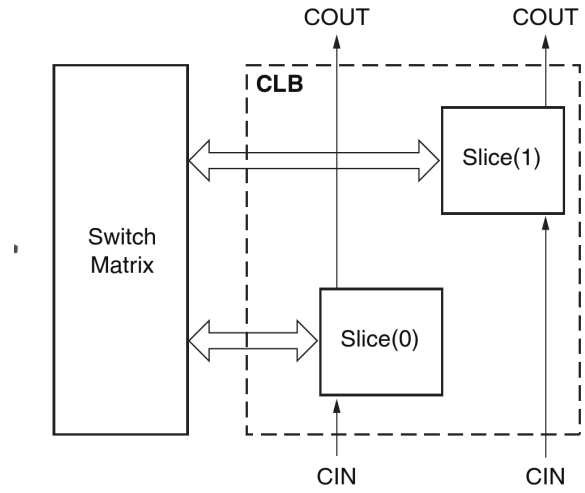
3.1.1. Budowa układów FPGA

Podstawowym komponentem układów FPGA jest konfigurowalny blok logiczny – CLB (ang. *Configurable Logic Block*) [11]. Jego schemat przedstawiony jest na rysunku 3.1. Jak można zauważyć, składa się on z 2 slice'ów połączonych z matrycą przełączeń (Switch Matrix), za pomocą której CLB mogą komunikować się między sobą. Sygnały CIN oraz COUT łączą ze sobą slice'y w dwóch sąsiednich blokach, a wykorzystywane są głównie przy implementacji operacji arytmetycznych. Można także zauważyć, że oba slice'y występujące wewnątrz jednego CLB są ze sobą niepołączone.

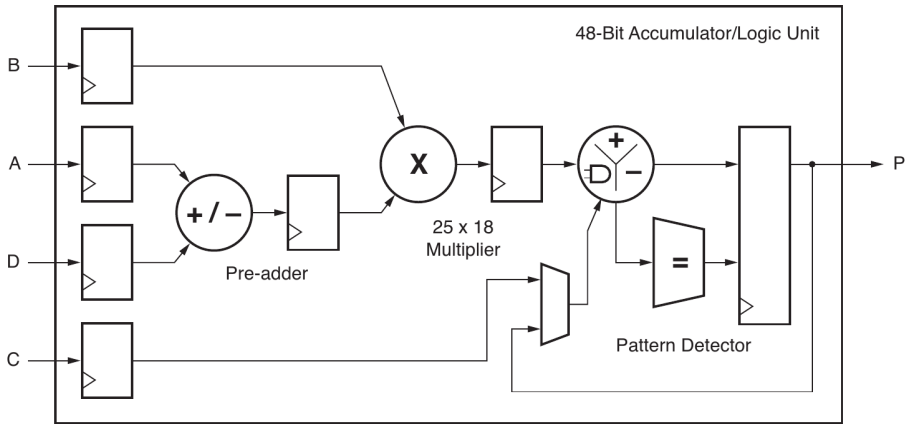
Idąc dalej, każdy slice składa się z następujących elementów [11]:

– Czterech generatorów funkcyjnych – każdy został zrealizowany jako sześciowejściowy LUT (ang. *look-up table*) z dwoma wyjściami.

– Multiplekserów – służą do łączenia ze sobą generatorów funkcyjnych w celu realizacji funkcji posiadających więcej niż sześć wejść.



Rys. 3.1. Budowa CLB. Źródło: *www.xilinx.com*



Rys. 3.2. Układ DSP48E1. Źródło: *www.xilinx.com*

– Ośmiu przerzutników typu D, z których cztery mogą być zamiennie wykorzystywane jako zatrzaski.

Dodatkowo, w niektórych slice'ach (SLIECEM) generatory funkcyjne można skonfigurować jako synchroniczną pamięć RAM albo rejestr przesuwny. Drugi typ – SLICEL nie ma takiej możliwości.

Pomiędzy matrycą bloków logicznych a fizycznymi sygnałami znajdują się dodatkowo bloki wejścia/wyjścia (IOB) [12]. Ich zadaniem jest konwersja sygnału w określonym standardzie na jeden bit danych, przekazywany później do logiki układu. Bardzo ważną, z punktu widzenia analizy obrazu, funkcjonalnością jest możliwość obsługi sygnałów różnicowych, którego przykładem jest wykorzystywany w HDMI (ang. *High Definition Multimedia Interface*) standard TMDS (ang. *Transition Minimized Differential Signaling*).

Każdy układ FPGA zawiera dodatkowo kilka innych, wyspecjalizowanych komponentów. Szeroko wykorzystywanym w tym projekcie zasobem jest DSP48E1 [13]. Jest to moduł za pomocą którego można zrealizować szybkie operacje arytmetyczne o dosyć dużych szerokościach słowa. Jego schemat zaprezentowano na rysunku 3.2.

Do zadań wymagających większych zasobów szybko dostępnej pamięci można wykorzystywać układy Block RAM [14]. Są to rozmieszczone wewnątrz logiki FPGA układy zawierające 36Kb dwuportowej pamięci. Każdy z nich może być skonfigurowany jako 1 obiekt z właśnie takim rozmiarem pamięci, albo dwa posiadające jej dwa

razy mniej. Dzięki takiemu ich rozmieszczeniu oraz rozmiarowi znajdują się pośrodku, pomiędzy dużą, ale oddaloną pamięcią zewnętrzną (zwykle DDR – ang. *Double Data Rate*) oraz małym, ale położonym wewnątrz CLB Distibuted RAMem.

3.1.2. Konfiguracja układów FPGA

Logikę w układach FPGA programuje się zwyczajowo za pomocą języka VHDL albo Verilog. Można także wykorzystywać języki programowania wyższego poziomu, na przykład C albo C++, a następnie za pomocą jednego z dostępnych narzędzi skompilować taki kod do VHDL albo Veriloga. Jest to przykład stosowania syntezy wysokiego poziomu (HLS – ang. *High Level Synthesis*), dzięki której działający algorytm może powstać szybciej, jednak będzie on mniej optymalny niż napisany za pomocą standardowych narzędzi przez doświadczonego projektanta.

Układy FPGA są konfigurowane za pomocą binarnego pliku konfiguracyjnego, generowanego na podstawie stworzonego kodu przez środowisko programistyczne. W przypadku Vivado Design Suite (narzędzie do programowania układów logicznych firmy Xilinx) proces generacji pliku konfiguracyjnego przedstawia się następująco [15]:

1. Synteza – proces przetwarzania kodu napisanego w jednym z dostępnych języków programowania (VHDL, Verilog, SystemVerilog) na netlistę, czyli opis algorytmu na poziomie bramek logicznych. Taka reprezentacja pozwala już na przeprowadzanie niektórych analiz oraz symulacji, a także jest wygodniejsza dla kolejnych etapów generacji pliku wynikowego.

2. Implementacja – proces, w którym netlista uzyskana w poprzednim etapie jest przetwarzana na projekt, w którym cała logika jest już rozmieszczona oraz połączona wewnątrz układu. Oprócz netlisty, w implementacji brany jest także pod uwagę plik ograniczeń użytkownika (User Constraints), w którym opisane są zależności czasowe oraz lokacyjne niektórych elementów.

3. Generacja pliku konfiguracyjnego

Dodatkowo, Vivado Design Suite posiada wbudowany symulator, dzięki któremu można sprawdzić poprawność działania algorytmu bez konieczności programowania, a nawet posiadania docelowego sprzętu. W celu generowania sekwencji testowych można wykorzystywać te same języki programowania co w przypadku projektowania układu, więc nie ma przy tym konieczności nauki dodatkowych technologii.

3.2. Procesor – PS

Sercem drugiej części układu Zynq jest dwurdzeniowy procesor ARM Cortex-A9 – *processing system* (PS). Jego zastosowanie w tym układzie jest odpowiedzią na bardzo częste wykorzystywanie softprocesorów, czyli procesorów zrealizowanych wewnątrz układu FPGA [16]. Jednym z popularniejszych układów tego typu jest Microblaze firmy Xilinx, który może być implementowany w każdym jej układzie. Rozwiązanie zastosowane w Zynq pozwala na posiadanie wewnątrz układu bardziej wydajnego procesora oraz pełniejsze wykorzystanie logiki programowalnej, gdyż nie są zajmowane dodatkowe bloki logiczne na ew. implementację softprocesora.

3.2.1. Architektura

ARM jest to rodzina procesorów typu RISC (ang. *Reduced Instruction Set Computing*) projektowana przez brytyjską firmę ARM Holdings [17]. Podejście polegające na zmniejszeniu oraz uproszczenie zestawu instrukcji (w przeciwieństwie do architektury x86 typu CISC – ang. *Complex Instruction Set Computing*) pozwoliło otrzymać mniej skomplikowany układ, co przekłada się na mniejsze zużycie mocy. Efekt ten spowodował bardzo dużą popularność układów tego typu w telefonach, tabletach oraz systemach wbudowanych, a więc wszędzie tam, gdzie wartość pobieranego prądu jest bardzo ważnym kryterium wyboru. Szacuje się, że procesory ARM posiadają obecnie 75% rynku 32-bitowych procesorów dla systemów wbudowanych.

Firma ARM Holdings nie zajmuje się bezpośrednio produkcją procesorów, a jedynie sprzedaje prawa do swych technologii. Produkt, który dostają klienci jest mocno konfigurowalny, co znacznie poszerza grono możliwych zastosowań. Przykładowo, firma Xilinx projektując procesor do Zynq zdecydowała się na układ dwurdzeniowy (możliwa liczba rdzeni to od 1 do 4), a także na opcjonalne układy NEON (wektorowe instrukcje SIMD – ang. *Single Instruction, Multiple Data*) oraz FPU (ang. *Floating Point Unit*) (obliczenia zmiennoprzecinkowe).

Zestawy instrukcji

Ciekawą cechą wyróżniającą procesory z rodziny ARM jest możliwość stosowania kilku zestawów instrukcji oraz przełączania pomiędzy nimi [18]. Podstawowy zestaw – ARM składa się z ciągu 32-bitowych rozkazów o stałej długości, co pozwala na osiągnięcie bardzo prostego etapu dekodowania instrukcji. Są one zaprojektowane w architekturze zwanej *load/store*, w której rozkazy dzielą się na dwie kategorie: dostępu do pamięci oraz operacji arytmetycznych, które potrafią operować jedynie na rejestrach procesora. Oznacza to, że przykładowo dodanie do siebie dwóch komórek pamięci składa się z etapów: pobrania danych (ang. *load*), przeprowadzenia obliczeń oraz zapisania wyników (ang. *store*).

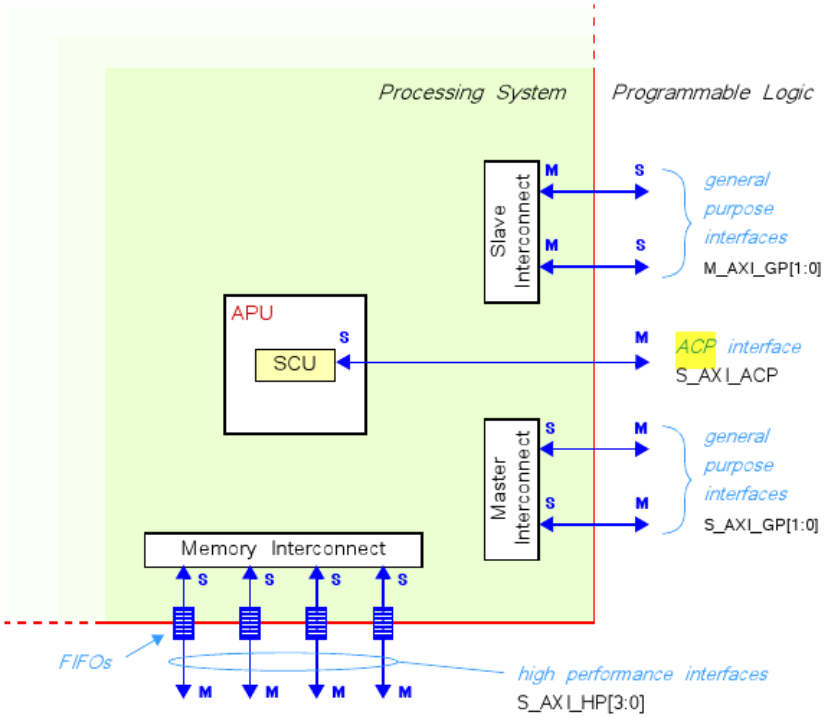
Fakt, że wszystkie instrukcje są 32-bitowe powoduje, że programy napisane dla tamtego zestawu zajmują dużo miejsca w pamięci. Problem ten rozwiązano za pomocą kolejnego zestawu zwanego *Thumb*. Reprezentuje on tylko część operacji z oryginalnego zbioru, jednak długość rozkazu to tylko 16 bitów. Większość jego instrukcji posiada bezpośredni odpowiednik w zestawie ARM, jednak niektóre z nich mają ograniczone funkcjonalności, przykładowo dostęp tylko do niektórych rejestrów albo części pamięci. Z tego powodu niektóre operacje muszą być zapisywane za pomocą większej liczby instrukcji, przez co program może stać się wolniejszy.

Powyższe problemy rozwiązane zostały podczas projektowania następcy powyższego zestawu, czyli *Thumb-2*. Rozszerza on swoją 16-bitową wersję o uzupełniający zbiór 32-bitowych instrukcji. Dzięki temu można osiągnąć podobną wydajność, co w zestawie ARM, przy wielkości programu porównywalnej do zestawu *Thumb*.

Procesory ARM posiadają także rzadziej wykorzystywane zestawy instrukcji. Na wzmiankę zasługuje tutaj zestaw *Jazelle*, który umożliwia bezpośrednie wykonywanie na procesorze niektórych instrukcji *Java Bytecode* [19].

3.3. Komunikacja w Zynq

Komunikacja wewnątrz Zynq opiera się o protokół AXI4 będący składową standardu ARM AMBA 3.0 [16]. Został on zaprojektowany do użytku wewnątrz mikrokontrolerów w 1996 roku i od tego czasu stał się standardem w komunikacji wewnątrz układów. Można wyróżnić trzy rodzaje połączeń w tym standardzie:



Rys. 3.3. Interfejsy AXI w Zynq (źródło [16])

1. AXI4 – inaczej AXI4 Full, oferuje zapis oraz odczyt danych z urządzenia z określonej komórki pamięci (o określonym adresie). Posiada tryb *burst*, w którym podawany jest jeden adres oraz długość transferu, a następnie przesyłane są po kolei wszystkie dane.

2. AXI4-Lite – wersja AXI pozbawiona trybu *burst*. Umożliwia więc jednoczesny odczyt oraz zapis tylko z jednej wybranej komórki pamięci.

3. AXI4-Stream – wersja z ciągłym przesyłem danych, w którym nie występuje pojęcie adresu w pamięci. Można powiedzieć, że jest to tryb *burst* o ciągłym działaniu. Podstawowym zastosowaniem jest przesył strumieni danych, na przykład obrazu albo dźwięku.

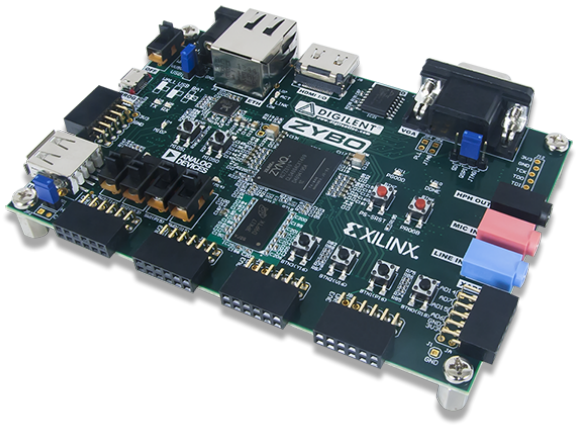
Komunikacja w standardzie AXI4 jest typu *master-slave*. *Master* inicjuje połączenia oraz transfery, a *slave* wykonuje jedynie odczyty oraz zapisy z określonych miejsc pamięci.

W układzie występują między innymi następujące interfejsy AXI4, zilustrowane na rysunku 3.3:

– Porty ogólnego przeznaczenia – bezpośrednie połączenia, które nie zapewniają buforowania danych. Posiadają szynę danych o szerokości 32 bitów. *Masterem* w dwóch z nich PS, a dwóch kolejnych PL.

– Porty wysokowydajne – cztery połączenia z buforem. Szerokość szyny danych wynosi 32 albo 64 bity. *Masterem* wszystkich czterech połączeń jest PL.

– Port ACP - 64 bitowe, asynchroniczne połączenie, którego celem jest zachowanie spójności pamięci podręcznej pomiędzy procesorem a logiką. Masterem połączenia jest PL.



Rys. 3.4. Zybo. Źródło: *www.digilentinc.com*

3.4. ZYBO

ZYBO jest płytką uruchomieniową zawierającą w sobie układ Z-7010 – najmniejszy z rodziny Xilinx Zynq-7000 [20]. Poza powyższym układem zawiera on pewną liczbę peryferiów zapewniających komunikację ze światem zewnętrznym w różnych standardach.

Najważniejsze z punktu widzenia niniejszej pracy to:

– Port HDMI, który może służyć zarówno jako wejściowy, jak i wyjściowy,

– Port VGA (ang. *Video Graphics Array*),

– Złącze USB (ang. *Universal Serial Bus*) wraz konwerterem USB - UART (ang. *universal asynchronous receiver/transmitter*).

Przykładami innych peryferiów są:

– Złącze 1G Ethernet,

– Wejścia oraz wyjścia audio,

– USB OTG (ang. *USB On-The-Go*),

– Slot kart MicroSD,

– 512MB pamięci DDR3.

Karta przedstawiona jest na rysunku 3.4.

4. Implementacja algorytmów śledzenia w układach FPGA oraz Zynq

Jak już zostało przedstawione w rozdziale 3, wykorzystywana w projekcie platforma sprzętowa jest dosyć specyficzna, więc bardzo ważną częścią pracy był taki dobór algorytmu, by jak najlepiej wykorzystać jej możliwości. W niniejszym rozdziale zaprezentowano analizą metod przedstawionych w rozdziale 2 w kontekście możliwej implementacji w układzie Zynq.

4.1. KLT

Dla przypomnienia, algorytm KLT opiera się na rozwiązywaniu układu równań liniowych (2.3) w każdej ramce obrazu. Poza tym podstawowym zadaniem, można także wyodrębnić pewne dodatkowe części:

– Obliczenie gradientu funkcji jasności dla danego punktu,

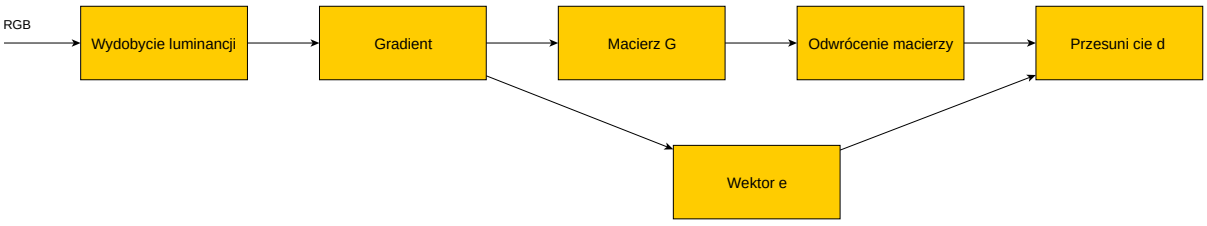
– Obliczenie macierzy G,

– Obliczenie wektora $\boldsymbol{e}$.

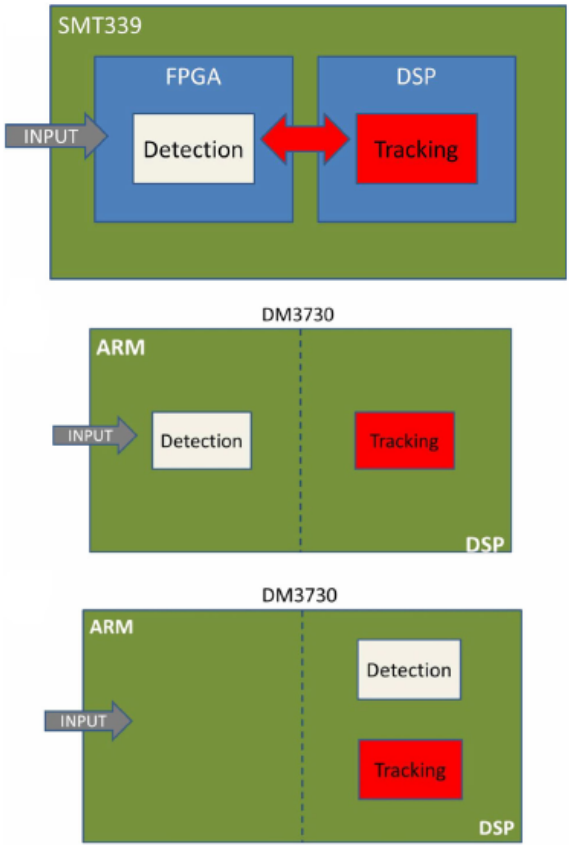
Wszystkie te etapy nadają się do realizacji w układzie FPGA. Przykładowy schemat przepływu danych w algorytmie został przedstawiony na rysunku 4.1. Dokładnie takie, potokowe podejście zostało zrealizowane w pracy [21].

Autor niniejszej pracy nie widzi dla tego przypadku żadnego sensownego wykorzystania systemu procesorowego dostępnego w układzie Zynq. Algorytm nie posiada części iteracyjnej, a umożliwienie śledzenia wielu obiektów wymaga powielenia istniejącej architektury. Zaprezentowane w cytowanej tutaj pracy rozwiązanie zmieściło się w 9176 LUT'ach, więc powinno dać się go zsyntezować także dla ZYBO, który posiada ich 17600.

Nieco inne podejście zostało jednak zaprezentowane w pracy [22]. Autorzy zrealizowali tam pełny układ śledzenia wraz z wykrywaniem obiektów. Testowane tam były różne podejścia dotyczące podziału takiego zadania



Rys. 4.1. Przepływ danych w algorytmie KLT



Rys. 4.2. Testowane w pracy [22] podziały zadań (źródło [22])

pomiędzy układami: FPGA, ARM oraz DSP (ang. *Digital Signal Processor*), co zostało przedstawione na rysunku 4.2. Wniosek z powyższej pracy był taki, że podział zadań z wykorzystaniem układu FPGA spowodował, że projekt działał najszybciej, osiągając ośmiokrotne przyspieszenie wykrywania obiektu względem DSP oraz 25x względem ARM.

Co prawda, zaprezentowany powyżej podział dzielił zadania pomiędzy układ FPGA oraz DSP, jednak nic nie stoi na przeszkodzie, by analogicznie postąpić w przypadku Zynq. Wadą w kontekście niniejszej pracy jest jednak fakt, że pomiędzy układami dzielone są całe algorytmy, a nie ich składowe. Podstawowym zamysłem autora było, aby odpowiednio rozmieścić algorytm śledzenia obrazu, a nie system jako całość.

4.2. *Mean shift*

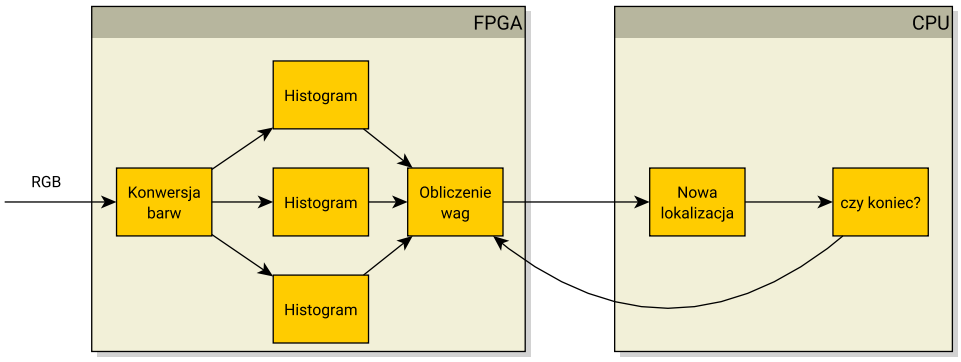
Algorytm *mean shift* znajduje się niejako po „przeciwnej stronie barykady" niż KLT. W przeciwieństwie do potokowego podejścia w poprzednim algorytmie jest to metoda typowo iteracyjna, co może sprawiać pewne problemy przy implementacji w układzie FPGA. Bardzo skutecznie wpasowuje się za to w architekturę działania procesorów.

Poza ogólną, iteracyjną sekwencją do wykonania są następujące zadania:

– Zmiana przestrzeni barw (RGB na YCbCr albo HSV),

– Obliczanie histogramów w sąsiadujących punktach,

– Obliczenie wag oraz wyjścia za pomocą wzorów (2.13) oraz (2.14).



Rys. 4.3. Schemat algorytmu *mean shift*.

Podejście polegające na obliczaniu histogramów oraz wag w układzie FPGA, a części iteracyjnej na softprocesorze Microblaze zostało przedstawione w artykule [23]. Autorzy doszli do wniosku, że wydzielenie dodatkowego modułu w logice programowalnej spowodowało przyspieszenie obliczeń wraz ze zmniejszeniem wykorzystania zasobów układu w porównaniu do „czysto programowego" rozwiązania.

Często stosowane są także pewne modyfikacje oryginalnego algorytmu, które ułatwiają realizację całości w logice programowalnej. Aby to umożliwić, dostosowywane są odpowiednie wzory, a w szczególności usuwana jest większość skomplikowanych i zajmujących dużo bloków logicznych dzieleń. Przykład takiego działania można znaleźć w pracy [24].

Obliczenia przeprowadzane są tam w przestrzeni kolorów LUV, a śledzone są obiekty z reprezentacją punktową. Pierwszym krokiem jest wybór z otoczenia aktualnego punktu wszystkich pikseli, których kolor w przestrzeni LUV jest podobny do koloru aktualnego piksela. W kolejnym kroku sumowane są określone wpływy z każdego z nich, aby ostatecznie uzyskać nowe przybliżenie koloru oraz przesunięcie. Niech $(x_c, y_c) = (x, y)$ będzie aktualną pozycję obiektu, $r\left(x, y\right)$ będzie kolorem w przestrzeni LUV w określonym punkcie, $h\left(p, q\right)$ będzie funkcją zwracającą 1 dla każdej pary kolorów p, q, które znajdują się odpowiednio blisko siebie oraz 0 w przeciwnym przypadku oraz $w_s = 2w + 1$ będzie szerokością rozpatrywanego okna. Przedstawione wyżej podejście może być opisane za pomocą wzorów:

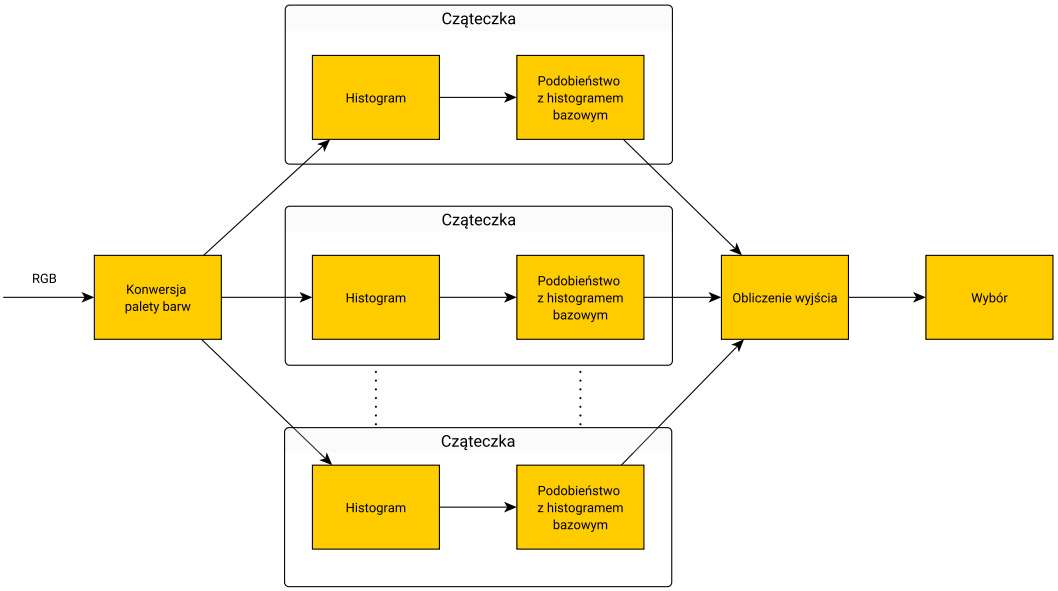
$$m = \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} h\left(r\left(x_c + dx, y_c + dy\right), r_c\right) \tag{4.1}$$

$$r_n = \frac{1}{m} \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} \left(r\left(x_c + dx, y_c + dy\right) - r_c\right) \cdot h\left(\left(r\left(x_c + dx, y_c + dy\right), r_c\right)\right) \tag{4.2}$$

$$dx_n = \frac{1}{m} \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} dx \cdot h\left(\left(r\left(x_c + dx, y_c + dy\right), r_c\right)\right) \tag{4.3}$$

$$dy_n = \frac{1}{m} \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} dy \cdot h\left(\left(r\left(x_c + dx, y_c + dy\right), r_c\right)\right) \tag{4.4}$$

Teraz w przypadku, gdy dx_n oraz dy_n są odpowiednio małe, na przykład w obrębie jednego piksela, przyjmujemy punkt (x_c, y_c) jako rozwiązanie zadania. W przeciwnym wypadku podstawiane jest $r_c = r_n$, $x_c = x_c + dx_n$ oraz $y_c = y_c + dy_n$ i powtarzane są wszystkie kroki algorytmu.



Rys. 4.4. Schemat filtru cząsteczkowego

4.3. Filtr cząsteczkowy

Filtr cząsteczkowy jest pod względem koniecznych do wykonania obliczeń bardzo zbliżony do algorytmu *mean shift* – podstawą do odszukiwania obiektu jest jego histogram. Sam algorytm można przedstawić w formie potokowej, a każde z koniecznych do wykonania obliczeń nie powinno sprawiać problemów w czasie implementacji w układzie FPGA. Taka architektura zaprezentowana została na schemacie 4.4. Dowód możliwości realizacji całości algorytmu w układzie FPGA można znaleźć w pracy [25]. Co prawda autorzy użyli histogram działający w skali szarości, jednak takie podejście nie zmieniło architektury rozwiązania.

Ciekawy sposób na jednoczesne wykorzystywanie procesora oraz logiki programowalnej przedstawione jest w pracy [26]. Obie części działają tam pod kontrolą autorskiego systemu operacyjnego, który jest w stanie dynamicznie przydzielać posiadanym zasobom odpowiednie wątki. Algorytm został podzielony na niezależne od siebie zadania, z których każde zostało zaimplementowane zarówno w sposób programowy, jak i sprzętowy. Następnie za pomocą adaptacji, czyli dynamicznemu, odpowiedniemu ustawianiu wątków, minimalizowany był czas obsługi ramki obrazu. Otrzymane wyniki świadczyły, że największy zysk można uzyskać poprzez umieszczenie w układzie FPGA obliczania wag cząsteczek, a liczenie histogramu było nieznacznie szybsze na procesorze. Nieopłacalna okazała się sprzętowa realizacja pozostałych etapów algorytmu.

Opierając się na tych danych, można przeprowadzić podział algorytmu na część programową oraz sprzętową bez wykorzystywania systemu operacyjnego oraz adaptacji. Autor proponuje umieścić w układzie FPGA obliczanie wag wszystkich cząsteczek wraz z odpowiadającym histogramem, a wybór oraz ruch cząsteczek zrealizować programowo. Taki podział zapewnia, że przysyłanie danych obrazowych pomiędzy logiką programowalną a procesorem jest niekonieczne.

Podejście to umożliwia także bardzo łatwe zorganizowanie śledzenia wielu obiektów. Wystarczy podczas implementacji stworzyć konfigurowalne przez procesor elementy, które służą tylko i wyłącznie do liczenia podobieństwa histogramów. W ten sposób można w dynamiczny sposób przydzielać cząsteczki poszczególnym śledzonym obiektom.

4.4. Posumowanie

Reasumując, każdy z przedstawionych w pracy algorytmów posiada swoją implementację zrealizowaną w pełni w układzie FPGA. W przypadku KLT, autor nie widzi jednak możliwości odpowiedniego włączenia do działania znajdującego się w Zynq procesora, więc można go wykluczyć z dalszych rozważań.

Zarówno w przypadku algorytmu *mean shift*, jak i filtrów cząsteczkowych, można odszukać zadowalający podział realizowanych zadań pomiędzy część programową oraz sprzętową. Jednak w przypadku *mean shifta* nie przynosi on dodatkowych korzyści w porównaniu z realizacją w samym układzie FPGA. Za to w filtrze cząsteczkowym można osiągnąć pewne dodatkowe możliwości, jak łatwe do realizacji śledzenie wielu obiektów jednocześnie oraz możliwość poprawy dokładności działania algorytmu. Autor do realizacji w omawianym projekcie wybrał więc filtr cząsteczkowy, a dokładna architektura tego rozwiązania przedstawiona będzie w kolejnym rozdziale.

5. Implementacja filtru cząsteczkowego w układzie Zynq

5.1. Tor wizyjny

Pierwszą, najbardziej widoczną dla użytkownika częścią projektu jest tor wizyjny. Znajduje się on w całości w układzie FPGA i składa się z części odpowiedzialnej za przechwytywanie obrazu ze złącza HDMI, modułu wizualizacji oraz z obsługi wyjściowego złącza VGA. Jego schemat wraz z komunikacją z pozostałymi częściami zaprezentowany jest na rysunku 5.1.

Dane o obrazie w sygnale HDMI przesyłane są w standardzie zwanym TMDS [27]. Składają się one z czterech równoległych linii transmisyjnych, w trzech przesyłane są kolory kolejnych pikseli, a w czwartej znajduje się zegar. Dane przesyłane są szeregowo bit po bicie w standardzie 8b/10b, czyli na każde 10 bitów 8 z nich reprezentuje kolor, a dwa pozostałe wykorzystywane są do przesyłania sygnałów synchronizacji. Użyty w pracy moduł do dekodowania sygnału HDMI ma swoje źródło w przykładach dostarczonych przez firmę *Digilent*, czyli producenta ZYBO.

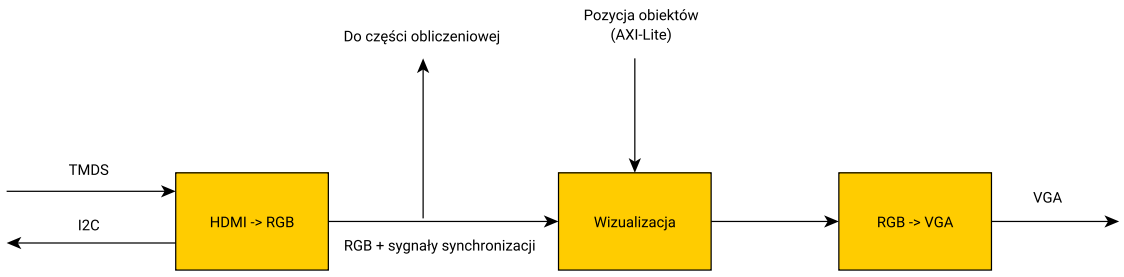
Moduł wizualizacji ma za zadanie wyrysować na obrazie aktualną pozycję śledzonych obiektów w postaci czerwonego prostokąta mającego go otaczać. Ze względu na to, że pozycja obiektu jest obliczana w CPU, informacja ta musi zostać tutaj w jakiś sposób dostarczona. Wykorzystywane jest w tym celu połączenie AXI-Lite, którego masterem jest system procesorowy. Połączenie zawiera cztery 32-bitowe rejestry, które zawierają kolejno:

– współrzędną x lewego górnego rogu obiektu,

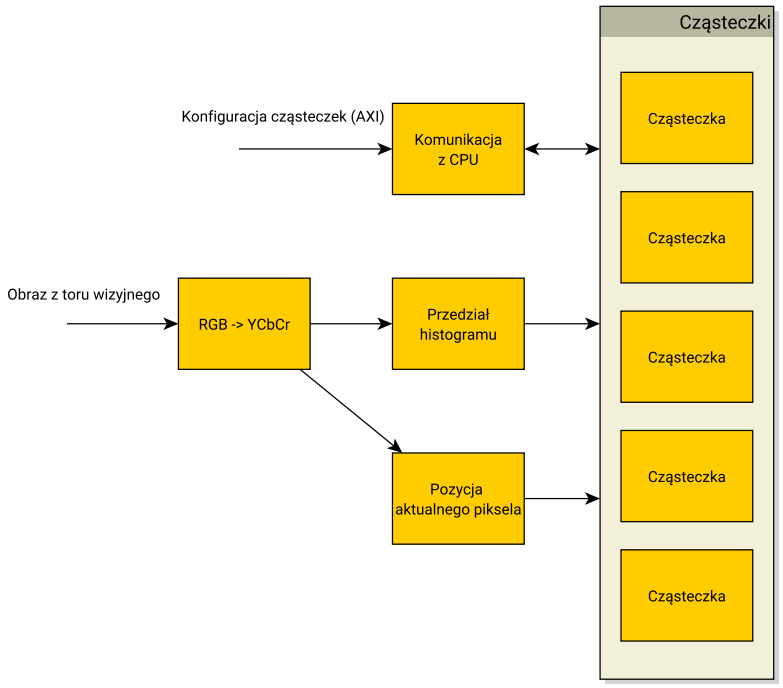
– współrzędną y lewego górnego rogu obiektu,

– współrzędną x prawego dolnego rogu obiektu,

– współrzędną y prawego dolnego rogu obiektu.



Rys. 5.1. Schemat toru wizyjnego



Rys. 5.2. Schemat części obliczeniowej

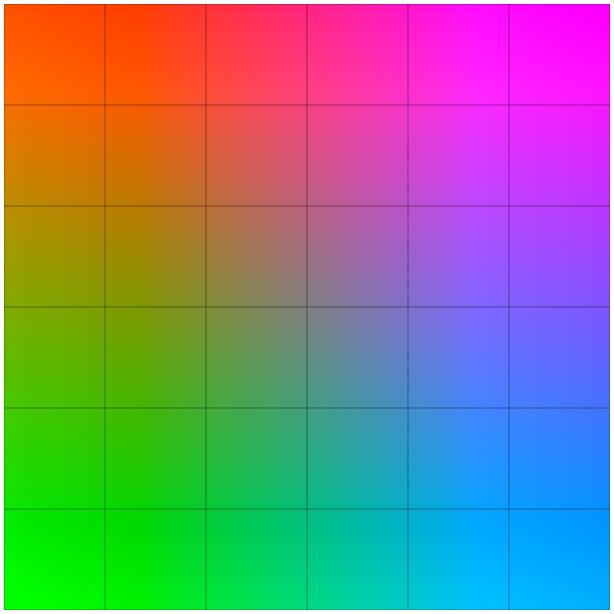
Dodatkową funkcjonalnością jest rysowanie siatki złożonej z kwadratów o boku długości 200 pikseli, których celem jest ułatwienie użytkownikowi korzystania z aplikacji (określanie początkowej pozycji obiektu).

VGA jest to analogowy sposób przesyłu obrazu, dawniej powszechnie wykorzystywany w komputerach stacjonarnych. Obecnie jest już praktycznie wyparty przez standardy cyfrowe takie jak HDMI, DVI (ang. *Digital Visual Interface*) albo *Display Port*. VGA należy do grupy standardów analogowych *component video*, czyli takich, gdzie składowe każdego koloru przesyłane są osobno. Poza danymi, przesyłane są także sygnały synchronizacji pionowej (*vsync*) oraz poziomej (*hsync*). Na płytce deweloperskiej ZYBO, składowe R oraz B mają rozdzielczość 5 bitów, a G 6 bitów, jednak w ogólności standard nie posiada takich ograniczeń. Podobnie jak w przypadku modułu HDMI, także i tutaj został użyty przykład od firmy *Digilent*.

5.2. Część obliczeniowa

Jest to druga, bardziej skomplikowana część realizowana w układzie FPGA. Ze względu na czytelność, wyłączone zostały z niej cząsteczki, które zostaną opisane później. Układ ten składa się więc z części, która jest przez nie wszystkie współdzielona. Jej schemat przedstawiony został na rysunku 5.2.

Pierwszym etapem rozpatrywanego potoku obliczeniowego jest moduł konwersji palety barw z RGB do YCbCr. Wewnętrznie, czynność ta składa się z mnożenia otrzymywanych wartości przez określoną, stałą macierz. Mnożeniem zajmują się wyspecjalizowane układy DSP48E1, ponieważ realizacja tego działania za pomocą elementóœ LUT zajmuje bardzo dużo miejsca w układzie. Wszystkie obliczenia realizowane są w standardzie stałoprzecinkowym ze znakiem, w którym na część całkowitą przeznaczono 8 bitów, a na część ułamkową 9. Jako wyjście podawany jest wynik bez części ułamkowej oraz bez znaku. Wykorzystany wzór to (wszystkie dane



Rys. 5.3. Przedziały histogramu na płaszczyźnie Cb-Cr

liczbowe są w postaci szesnastkowej):

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} \texttt{0x09917} & \texttt{0x12c8b} & \texttt{0x03a5e} \\ -\texttt{0x1a99b} & -\texttt{0x15665} & \texttt{0x10000} \\ \texttt{0x10000} & -\texttt{0x129a2} & -\texttt{0x1d65e} \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix} + \begin{bmatrix} \texttt{0} \\ \texttt{0x80} \\ \texttt{0x80} \end{bmatrix} \tag{5.1}$$

Obliczanie histogramów jest to kluczowe zadanie wykonywane przez każdą cząsteczkę. Każdy jego przedział reprezentuje jeden kwadrat na płaszczyźnie Cb-Cr, jak to zostało pokazane na rysunku 5.3.

Całość zadania otrzymywania histogramu w każdej z cząsteczek podzielona jest na dwie części. Pierwszy etap jest wspólny dla każdego elementu i z tego powodu znajduje się w części obliczeniowej. Określane jest w nim, w którym przedziale histogramu znajduje się aktualnie przetwarzany piksel. W drugim etapie każda cząsteczka na tej podstawie tworzy już właściwy histogram w odpowiadającym jej oknie.

Kod w języku Verilog odpowiedzialny za pierwszy etap przedstawiony jest na listingu 1. Warto zwrócić uwagę na format danych w porcie wyjściowym *currentBin*. Każdy jego bit odpowiada jednemu przedziałowi histogramu i można w nim odczytać, czy kolor aktualnego piksela się w nim znajduje czy nie.

Kolejnym zadaniem części obliczeniowej jest ciągłe wyznaczanie współrzędnych aktualnie rozpatrywanego piksela. Składowa *x* jest to wyjście z licznika zliczającego rosnące zbocza sygnału zegarowego, resetowanego za pomocą negacji sygnału *hsync*. Podobnie uzyskuje się składową *y* – wystarczy zliczać opadające zbocza sygnału *hsync* i zerować licznik za pomocą negacji *vsync*.

Wspólnym dla każdej z cząstek modułem jest także układ komunikacji z CPU. Jego zadaniem jest dostarczenie cząsteczkom parametrów okna w którym mają operować oraz histogramu bazowego. W zamian przekazują one wynik swoich obliczeń, czyli podobieństwo obliczonego histogramu z bazowym. Dodatkowo jedna z cząstek jest tak skonstruowana, że potrafi przekazać do procesora także obliczany w każdej iteracji histogram. Jest on wykorzystywany podczas w procesie inicjalizacji, podczas którego część programowa musi poznać bazowy histogram wykorzystywany w kolejnych ramkach obrazu. Wszystkie powyższe dane przesyłane są za pomocą połączenia

```
 module bin #(
        parameter COLORSIZE = 6
)
(
    input [7:0] y,
    input [7:0] cb,
    input [7:0] cr,
    output [COLORSIZE * COLORSIZE - 1:0] currentBin
);

genvar j;
generate
        for (j = 0; j < COLORSIZE * COLORSIZE; j = j + 1)
        begin: colorPart
                assign currentBin[j] = cb >= (j % COLORSIZE) * 256 / COLORSIZE &&
                        cb < (j % COLORSIZE + 1) * 256 / COLORSIZE &&
                        cr >= (j / COLORSIZE) * 256 / COLORSIZE &&
                        cr < (j / COLORSIZE + 1) * 256 / COLORSIZE;
        end
endgenerate

endmodule
)
```

Listing 1. Moduł określający przedział histogramu

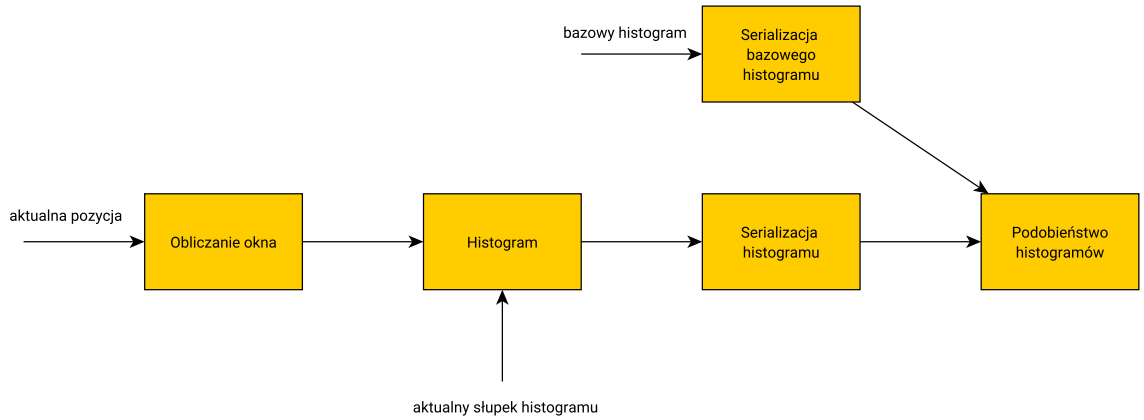
AXI, którego masterem jest system procesorowy. Pewną różnicą w stosunku do standardowej implementacji takiego połączenia jest rozdzielenie pamięci do odczytu oraz zapisu. Ułatwia to nieznacznie zarządzanie adresami pamięci przez programistę. W pamięci do zapisu znajdują się kolejno bez wyrównywania do najbliższego bajtu:

- współrzędna *x* lewego górnego rogu każdej z cząsteczek (11 bitów na cząsteczkę),
- współrzędna *y* lewego górnego rogu każdej z cząsteczek (11 bitów na cząsteczkę),
- współrzędna *x* prawego dolnego rogu każdej z cząsteczek (11 bitów na cząsteczkę),
- współrzędna *y* prawego dolnego rogu każdej z cząsteczek (11 bitów na cząsteczkę).
- histogram bazowy.

W pamięci do odczytu znajdują się:

- podobieństwo histogramów obliczone przez każdą kolejną cząsteczkę (32 bity na cząsteczkę),
- histogram obliczony przez pierwszą cząsteczkę.

Format przekazywanych histogramów zostanie pokazany w opisie cząsteczki.



Rys. 5.4. Cząsteczka

Ostatnią funkcjonalnością modułu jest generacja przerwań dla procesora oznaczających koniec przetwarzania ramki obrazu. Jest to znak, że w pamięci do odczytu zapisane już zostały wyniki obliczeń. Tutorial prezentujący obsługę AXI oraz przerwań w Zynq został zaprezentowany w dodatku A.

5.3. Cząsteczka

Jest to w rzeczywistości część układu obliczeniowego, ale wydzielona ze względu na czytelność prezentacji. Wyjściem każdej z cząsteczek jest podobieństwo histogramów: wyznaczonego w aktualnej ramce obrazu oraz bazowego, czyli otrzymanego w pierwszej ramce. Architektura realizacji tego zadania przedstawiona jest na rysunku 5.4.

Pierwsze dwa etapy głównego potoku mają za zadanie obliczyć histogram Cb-Cr w oknie przypisanym cząsteczce. Kolejne przedziały histogramu reprezentowane są przez liczbę pikseli posiadających odpowiedni kolor. Realizacja w języku Verilog przedstawiona jest na listingu 4. Jak można tam zauważyć, moduł ten składa się z jednego licznika na każdy przedział, których wyjścia łączone są w jedną wynikową tablicę. Port wejściowy *insideWindow* jest połączony z wyjściem pierwszego z modułów w potoku, w którym sprawdzane jest, czy współrzędne aktualnie rozpatrywanego piksela znajdują się wewnątrz okna.

Podobieństwo obliczane w cząsteczce to kwadrat współczynnika Bhattacharyya, czyli:

$$\rho\left(a,b\right) = \sum_{i=0}^{m} a_i b_i \tag{5.2}$$

gdzie: a oraz b są to histogramy o m przedziałach.

Równanie to można zrealizować jako akumulator z mnożeniem, co jest bezpośrednio wspierane przez układ DSP48E1. Podejście to wymaga jednak posiadania histogramu w postaci szeregowej, a wyjście z modułu histogramu jest w pełni równoległe. Problem ten został rozwiązany za pomocą dodatkowego modułu serializującego dane, przedstawionego na listingu 3. Składa się on z licznika podłączonego do sygnału zegarowego, którego wyjście wykorzystywane jest podczas przesunięcia bitowego wejścia.

```
module histogram #(
parameter HISTOGRAM_SIZE = 36
parameter HISTOGRAM_WIDTH = 6
)
(
input clk,
input [HISTOGRAM_SIZE - 1:0] currentBin,
input insideWindow,
input reset,
output[HISTOGRAM_SIZE * HISTOGRAM_WIDTH - 1: 0] histogram
);

genvar i;
generate
for (i = 0; i~< HISTOGRAM_SIZE; i~= i~+ 1)
begin: counters
wire useThisCounter;
assign useThisCounter = insideWindow && currentBin[i] == 1;

histogram_counter counter (
.CLK(clk),
.CE(useThisCounter),
.SCLR(reset),
.Q(histogram[(i + 1) * HISTOGRAM_WIDTH - 1: i~* HISTOGRAM_WIDTH])
);
end
endgenerate

endmodule)
```

Listing 2. Obliczanie histogramu

5.4. Część programowa

Część programowa, czyli ta uruchamiana na procesorze, ma pięć podstawowych zadań:

– Dokończenie liczenia wag,

– Obliczenie wyjścia (czyli pozycji śledzonego obiektu),

– Wybór cząsteczek,

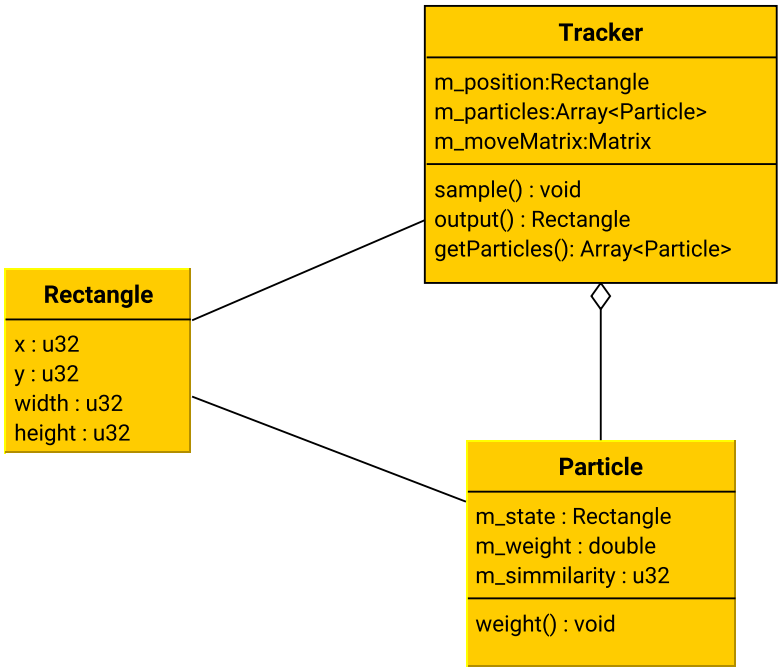
– Przesunięcie cząsteczek zgodnie z równaniem stanu,

– Komunikacja z komputerem (interfejs użytkownika).

```
  module serializer #(
        parameter HISTOGRAM_SIZE = 36,
        parameter HISTOGRAM_WIDTH = 6
)
(
    input clk,
    input enable,
    input reset,
    input [HISTOGRAM_SIZE * HISTOGRAM_WIDTH - 1:0] histogram,
    output [HISTOGRAM_WIDTH - 1:0] serialized
    );

reg clear;
wire [6:0]position;
serializer_counter counter (
      .CLK(clk),    // input wire CLK
      .CE(enable),      // input wire CE
      .SCLR(reset),  // input wire SCLR
      .Q(position)        // output wire [6 : 0] Q
    );

reg [HISTOGRAM_WIDTH - 1:0]result = 0;

always @(posedge clk)
begin
        result <= enable ? histogram >> (position * HISTOGRAM_WIDTH) : 0;
end

assign serialized = result;

always @(posedge clk)
begin
        clear = position > HISTOGRAM_SIZE;
end

endmodule
```

Listing 3. Serializacja histogramu

Pierwsze cztery zadania związane są z dokończeniem algorytmu śledzenia, którego pierwszy etap został wykonany w części sprzętowej. Odpowiednia procedura znajduje się w obsłudze przerwania pochodzącego od układu FPGA.



Rys. 5.5. Diagram klas części programowej

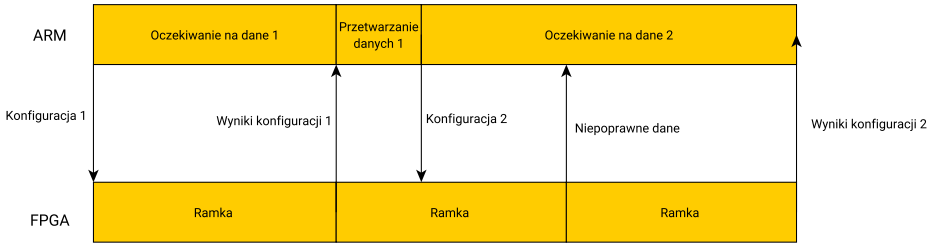
Po testach modelu programowego. autor doszedł do wniosku, że prezentowany tutaj algorytm, w którym wykonywane są wszystkie powyższe kroki nie działa w pełni poprawnie. Przyczyna leży najprawdopodobniej w tym, że wybór cząsteczek znajdujących się najbliżej obiektu oraz branie pod uwagę przesunięcia obiektu w równaniu ruchu wzajemnie się wzmacniają i w efekcie obiekt jest bardzo szybko gubiony. Autor znalazł dwa rozwiązania, które znacznie poprawiają sytuację. Pierwszym rozwiązaniem jest usunięcie z algorytmu etapu wyboru, jednak trudno jest po takiej operacji dalej nazywać metodę filtrem cząsteczkowym. Drugim wariantem jest zmiana procesu przewidywania na taki, w której występuje jedynie losowe rozmieszczenie cząsteczek, co przykładowo jest zaprezentowane w pracy [8]. Aby umożliwić testowanie obu wariantów, część programowa została tak skonstruowana, aby przełączanie pomiędzy nimi wymagało jedynie niewielkich zmian w kodzie programu.

Architektura części programowej opiera się o klasę *Tracker*, której podstawowym zadaniem jest zarządzanie elementami obliczeniowymi algorytmu, czyli cząsteczkami (klasa *Particle*). Diagram klas części programowej algorytmu przedstawiony jest na rysunku 5.5.

Komunikacja z użytkownikiem wykonywana jest w głównej pętli programu. Wykorzystywany jest występujący na płytce ZYBO konwerter USB UART. Po podłączeniu komputera wysyłane jest proste menu, gdzie można użytkownik może wpisać pozycję początkową obiektu. Jej otrzymanie uruchamia algorytm śledzenia, którego kroki z punktu widzenia programu są następujące:

1. Otrzymanie bazowego histogramu. Należy podać pierwszej cząsteczce początkową pozycję obiektu podaną przez użytkownika. Od momentu otrzymaniu drugiego przerwania niezbędne dane znajdują się w odpowiednim miejscu pamięci do odczytu części sprzętowej.

2. Wygenerowanie początkowego zbioru cząsteczek. Jak zostało już powiedziane w rozdziale 2, musi być on wylosowany na podstawie zadanego z góry rozkładu prawdopodobieństwa. Autor wybrał w tym celu rozkład jednostajny.



Rys. 5.6. Synchronizacja pomiędzy częścią programową i sprzętową

3. Wysłanie danych do układu FPGA. Do tego celu używany jest omówiony w części sprzętowej interfejs AXI. Wypełniane są tutaj odpowiednie komórki w pamięci otrzymanym w pierwszym punkcie histogramem bazowym oraz wylosowanymi w poprzednim etapie pozycjami cząsteczek.
4. Część sprzętowa algorytmu W tym momencie procesor czeka na otrzymanie od logiki programowalnej obliczonych podobieństw histogramów. Nastąpi to po otrzymaniu drugiego przerwania.
5. Aktualizacja wag zgodnie ze wzorem (2.17).
6. Obliczenie aktualnej pozycji obiektu – wzór (2.18).
7. Wysłanie do modułu wizualizacji za pomocą odpowiedniego połączenia AXI obliczonych w poprzednim kroku współrzędnych obiektu.
8. Opcjonalny etap wyboru.
9. Opcjonalne przesunięcie wybranych cząsteczek za pomocą równania stanu obiektu.
10. Powrót do punktu trzeciego.

Warty omówienia jest także sposób synchronizacji działań procesora oraz logiki programowalnej. Układ FPGA o zakończeniu swoich działań informuje część programową za pomocą przerwania, w której procedurze obsługi znajduje się całość kodu algorytmu. Standardowo, jeśli program znajduje się wewnątrz niej, przerwanie nie może zostać wywołane kolejny raz. W takim przypadku procesor nie zostanie powiadomiony o nowych danych podczas obsługi starych. Efekt ten jest w rozważanej aplikacji niepożądany. Dodatkowo, aby zapewnić sobie, że odczytywane dane odpowiadają aktualnie rozpatrywanej konfiguracji, musi być ignorowane co drugie przerwanie, co zostało zaprezentowane na rysunku 5.6.

6. Ewaluacja wykonanego systemu wizyjnego

6.1. Część sprzętowa

6.1.1. Konfiguracja modułu

Sprzętowa część algorytmu posiada trzy istotne parametry konfiguracyjne:

1. liczbę cząsteczek – tyle jednocześnie może być badanych potencjalnych lokalizacji obiektu,
2. liczbę przedziałów histogramu – tyle kolorów w obrazie jest rozróżnialne,
3. liczbę pikseli w przedziale – tyle pikseli jednego koloru może być brane pod uwagę. W dalszej części parametr ten będzie nazywany wielkością przedziału.

Oczywistym jest, że zwiększenie każdego z nich wpływa także na większe wykorzystanie zasobów układu FPGA. Ważnym zadaniem był więc taki ich dobór, aby maksymalnie wykorzystać dostępny sprzęt i jednocześnie każdy miał wystarczająco dużą wartość. Uzyskane wyniki przedstawione są w tabeli 6.1. Jak można tam zauważyć, nie są to duże wielkości, przykładowo w literaturze najczęściej mówi się o stu albo nawet dwustu cząsteczkach. Szczególnie małą wielkością charakteryzuje się jednak maksymalna możliwa liczba pikseli w przedziale histogramu, przez co praktyczne zastosowanie niniejszego projektu ogranicza się do śledzenia małych obiektów.

6.1.2. Wykorzystanie zasobów

Wykorzystanie zasobów sprzętowych przez moduł z powyżej wskazanymi parametrami przedstawione jest w tabeli 6.2. Jak można w niej zauważyć, zajęte są prawie wszystkie *slice'y* dostępne w układzie, także wszystkie próby zwiększenia wartości parametrów modułu kończyły się niepowodzeniem.

Możliwe jest także sprawdzenie, ile miejsca w układzie zajmuje pojedyncza cząsteczka, także po zmianie liczby przedziałów histogramu albo jego wielkości. Wyniki tych prób znajdują się w tabeli 6.3.

Za pomocą *Vivado Design Suite*, można także podjąć próbę oszacowania ilości energii zużywanej przez układ. Wyniki analizy zaprezentowano w tabeli 6.4. Można tam zauważyć, że zdecydowanie największa część mocy

Tabela 6.1. Wartości parametrów części sprzętowej

Parametr	Wartość
Liczba cząsteczek	25
Liczba przedziałów	36
Wielkość przedziału	63

Tabela 6.2. Wykorzystanie zasobów sprzętowych układu FPGA

Nazwa	Zajęte	Dostępne	Wykorzystanie
SLICE	4013	4400	91%
LUT	13763	17600	78%
FF	11430	35200	32%
DSP	31	80	39%

Tabela 6.3. Wykorzystanie zasobów przez jedną cząsteczkę dla różnych parametrów

Liczba przedziałów	Wielkość przedziału	SLICE	LUT	FF
36	63	167	425	278
49	63	307	949	486
36	127	217	655	316

zużywana jest przez system procesorowy (86%), a pozostałe składowe nie odgrywają dużej roli w wyniku. Łączna, szacowana moc zużywana przez układ to 1,912W, co porównując na przykład z *Raspberry Pi* zużywającym typowo 3,5 do 5W jest małą wartością.

Tabela 6.4. Analizy zużycia mocy przez układ

Składowa	Moc[W]
Zegary	0.034
Sygnały	0.027
Logika	0.016
DSP	0.013
MMCM	0.087
I/O	0.041
Processing System	1.556
Zużycie stałe	0.137

6.2. Część programowa

Poza samą poprawnością działania, ważnym kryterium jakości części programowej była szybkość wykonywania algorytmu, a co najważniejsze, aby umożliwiał on przeprowadzenie wszystkich obliczeń w czasie trwania jednej ramki obrazu. Nie jest to co prawda kryterium konieczne, ale dzięki temu zaprojektowany system jest w stanie osiągnąć swoją maksymalną wydajność.

Jest możliwym sprawdzenie, czy układ działa zgodnie z powyższymi założeniami, co jest szczególnie istotne wiedząc, że poza wykonaniem samego algorytmu, podczas tego czasu musi zostać wykonana komunikacja z układem FPGA za pomocą protokołu AXI. W tym celu testowo dodano do programu licznik sprawdzający, ile razy zostały wykonane obliczenia. Jego wartość wysyłano cyklicznie za pomocą protokołu UART do komputera. Tam uruchomiono prosty skrypt testowy wypisujący, ile razy na sekundę wykonywany był krok algorytmu. Kod tego testu przedstawiony jest poniżej:

```
#!/usr/bin/env python3
import serial
import time

ser = serial.Serial('/dev/ttyUSB1', 115200, bytesize=8, stopbits=1, parity='N')

startInterrupts = int(ser.readline())
start = time.time()

for i~in range(0, 20):
        ser.readline()

endInterrupts = int(ser.readline())
end = time.time()

duration = end - start
interrupts = endInterrupts - startInterrupts
rate = interrupts / duration

print("seconds: %f, interrupts: %d, ratio: %f" % (duration, interrupts, rate))
```

Wynikiem działania powyższego skryptu dla obrazu wejściowego o 60 klatkach na sekundę jest:

```
seconds: 70.030925, interrupts: 2100, ratio: 29.986752
```

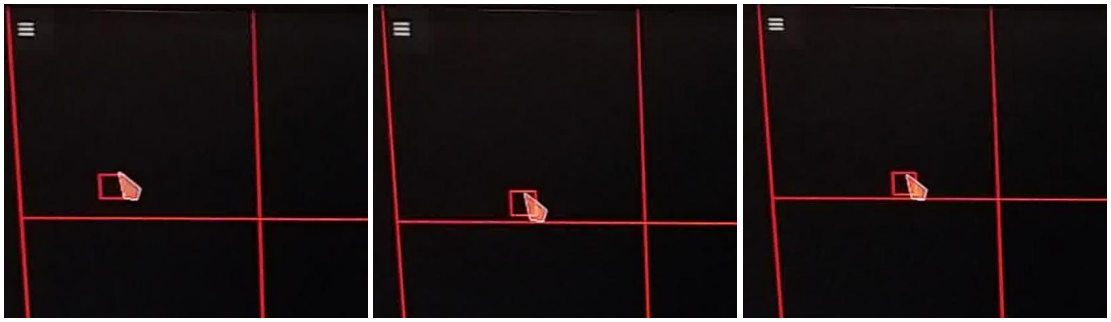
Jak można zauważyć, odstępstwo od założonego celu jest minimalne (rzędu 0,05%) co dowodzi, że implementacja części programowej jest wystarczająco szybka.

6.3. Prezentacja działania

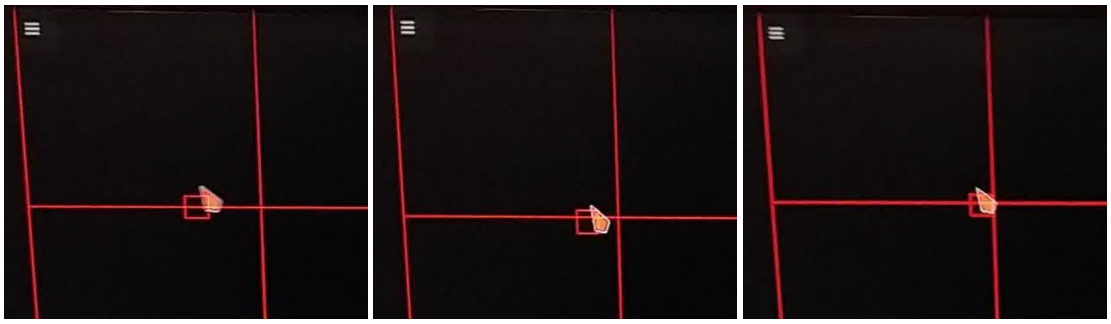
W tej części przedstawione zostaną wyniki działania dwóch wersji algorytmu dla różnych przypadków. Wszystkie testy wykonano w rozdzielczości 1024x768 przy 60 klatkach na sekundę.

6.3.1. Wersja ze zmienionym etapem przewidywania

Kilka sekwencji testowych prezentujących działanie tej wersji algorytmu przedstawiono na rysunkach: 6.1 oraz 6.2. Można na nich zauważyć, że śledzenie przebiegało poprawnie. Sporym ograniczeniem w przypadku tej wersji jest maksymalna szybkość poruszania się obiektu, która teoretycznie wynosi 150 pikseli na sekundę, jednak w praktyce znacznie mniej. Przyczyna leży w tym, że podczas jednej ramki obrazu, każda cząsteczka, a więc także w dłuższym okresie czasu obiekt, może przesunąć się o maksymalnie 5 pikseli. Sytuację można by poprawić, poprzez zwiększenie liczby cząsteczek – algorytm mógłby wtedy bezpiecznie bardziej je oddalać od obiektu bez obaw o jego zgubienie.

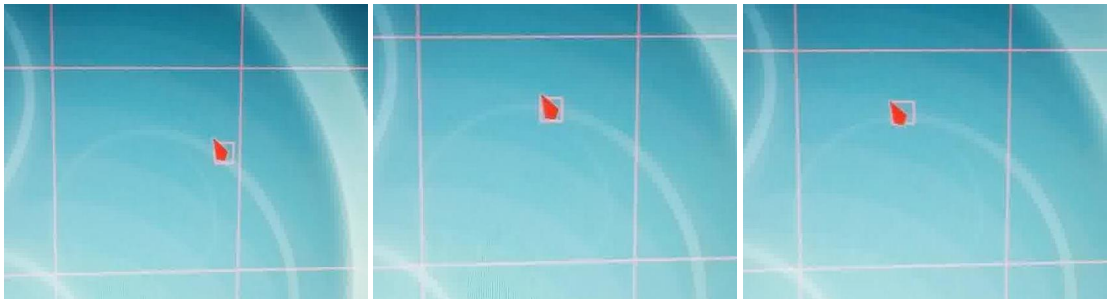


(a) Początek (b) Sekunda 1 (c) Sekunda 2

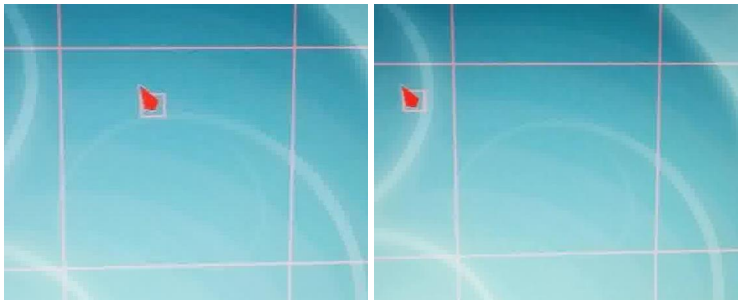


(d) Sekunda 3 (e) Sekunda 4 (f) Sekunda 5

Rys. 6.1. Śledzenie kursora myszy przy stałym tle

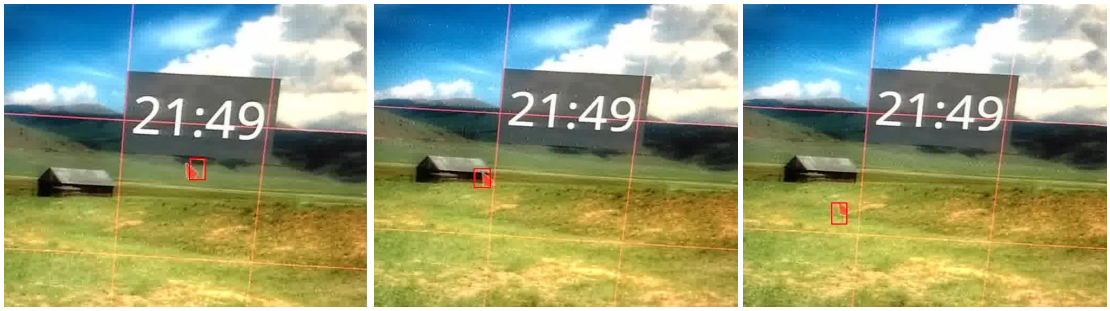


(a) Początek (b) Sekunda 1 (c) Sekunda 2

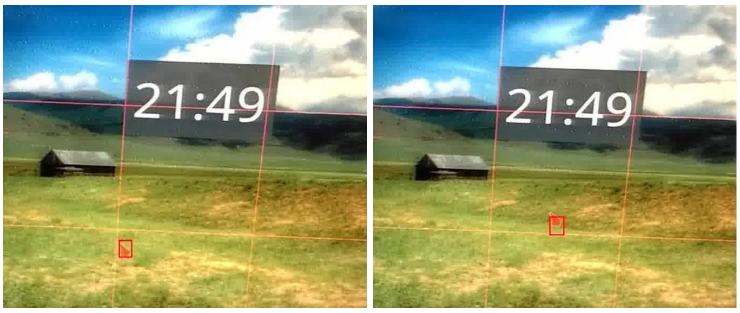


(d) Sekunda 3 (e) Sekunda 8

Rys. 6.2. Śledzenie kursora myszy przy zmiennym tle



(a) Początek (b) Sekunda 4 (c) Sekunda 8



(d) Sekunda 12 (e) Sekunda 20

Rys. 6.3. Śledzenie kursora myszy z tłem podobnym do początkowego

6.3.2. Wersja bez etapu wyboru

Jak już zostało powiedziane we wcześniejszym rozdziale, ta wariacja oryginalnego algorytmu nie jest filtrem cząsteczkowym. Gdy się uważniej spojrzy na jej konstrukcję, można zauważyć, że jest to chmura cząsteczek, zawsze pośrodku której znajduje się aktualna pozycja obiektu. Kilka sekwencji testowych można znaleźć na rysunkach 6.3 oraz 6.4.

Jak można zauważyć, w obu przypadkach śledzenie przebiegało poprawnie. Podczas obserwacji działania algorytmu został zauważony ciekawy efekt drgania wyniku, co można zaobserwować w sekwencji na rysunku 6.5 wykonaną podczas mniej niż jednej sekundy.

6.3.3. Porównanie

Oba testowane warianty posiadały swoje wady oraz zalety. Wersja bez etapu wyboru, dzięki wykorzystaniu dodatkowego przesunięcia w algorytmie, była w stanie śledzić szybciej poruszające się obiekty. Odbyło się to kosztem mniejszej stabilności działania oraz wyraźnymi drganiami wyjściowej pozycji. Należy jednak stwierdzić, że obie wariacje wykonywały swoje zadanie – były w stanie śledzić poruszający się obiekt.

6.4. Śledzenie wielu obiektów

Zostało już wcześniej wspomniane, że tak zaprojektowany układ powinien bezproblemowo umożliwiać śledzenie wielu obiektów jednocześnie. Niestety, wielkość dostępnego układu FPGA to uniemożliwiła, ponieważ dostępne zasoby sprzętowe nie pozwalają na umieszczenie większej liczby cząsteczek, co byłoby do tego wymagane.

(a) Początek (b) Sekunda 8

(c) Sekunda 15 (d) Sekunda 19

(e) Sekunda 21 (f) Sekunda 23

(g) Sekunda 26 (h) Sekunda 37

(i) Sekunda 57

Rys. 6.4. Śledzenie na rzeczywistym obrazie

Rys. 6.5. Przykład ilustrujący „drganie” pozycji śledzonego obiektu

Z tego powodu autor zrezygnował z umieszczania w kodzie projektu części to umożliwiającej, aby maksymalnie zwiększyć liczbę dostępnych cząstek opisujących pojedynczy obiekt. Jedyną niezbędną zmianą umożliwiającą śledzenie wielu obiektów jest dodanie możliwości podawania każdej z cząstek osobno bazowego histogramu – obecnie przesyłany jest tylko jeden, który jest następnie wszystkim przekazywany.

7. Podsumowanie

Zasadniczy cel niniejszej pracy został osiągnięty – system wizyjny zdolny do śledzenia zadanych obiektów. Co więcej, udało się znaleźć algorytm, który bardzo dobrze wpasowuje się w architekturę układu Zynq – posiada wiele równolegle obliczanych cząsteczek zarządzanych przy pomocy sekwencyjnego bloku kodu.

Największym mankamentem jest to, że zaprezentowane metody dobrze radzą sobie jedynie dla małych, wolno poruszających się obiektów. Możliwym sposobem naprawy tego stanu rzeczy, mogłoby być zwiększenie układu FPGA wykorzystywanego w projekcie albo ewentualnie zmniejszenie liczby zadań tam wykonywanych. Najprawdopodobniej system działałby z podobną szybkością 30 klatek na sekundę także w przypadku, gdyby część programowa algorytmu dostawała całość histogramów, a nie tylko ich podobieństwo z bazowym.

Zaprezentowany projekt posiada wiele możliwości rozwoju w przyszłości. Ciekawym usprawnieniem mogłoby być uruchomienie śledzenia wielu obiektów jednocześnie. Jest to zadanie, które jest częstym wymaganiem w stosunku do tego typu systemów, na przykład w monitoringu przestrzeni publicznej.

Pełny układ śledzenia obiektów powinien posiadać jeszcze jeden moduł, a mianowicie wykrywanie obiektów. W obecnym projekcie pozycja początkowa była podawana ręcznie za pomocą protokołu UART, ale zdecydowanie większą wartość miałby układ, który potrafi takie zadanie wykonać automatycznie. Poza oczywistą zaletą płynącą z autonomiczności takiego projektu, system byłby także w stanie znacznie szybciej reagować na pojawienie się na ekranie nowej rzeczy. Pozwalałoby to także na przykład na wyrysowanie pełnej trajektorii obiektu, a ie dopiero od momentu zauważenia oraz wprowadzenia odpowiednich danych przez człowieka (tj. ręcznej inicjalizacji śledzenia).

Po napisaniu pracy autor uważa, że środowisko Zynq może być z powodzeniem wykorzystywane także w innych zastosowaniach, a niezastąpione jest w bardziej złożonych problemach dotychczasowo rozwiązywanych przy pomocy kilku pojedynczych układów. Możliwe jest także podejście odwrotne – na przykład rozszerzenie istniejących projektów programowych o akceleratory obliczeń w układzie FPGA. Połączenie takie jest już obecnie coraz częściej stosowane w serwerach centrów obliczeniowych, gdzie niezbędna jest szybka obróbka bardzo dużej ilości danych. Możliwe, że w przyszłości takie podejście projektowe będzie stosowane nawet w komputerach osobistych ze względu na coraz większe wymagania w stosunku do mocy obliczeniowej oraz coraz baczniejsze przypatrywanie się zużyciu energii przez urządzenia domowe.

Projektowanie w środowisku Zynq może początkowo stwarzać pewne trudności, ponieważ trzeba dokładnie na początku dokładnie określić, co ma być częścią programową, a co sprzętową, jednak efekty są warte poniesionych wysiłków.

Bibliografia

[1] Alper Yilmaz, Omar Javed i Mubarak Shah. „Object Tracking: A Survey". W: *ACM Comput. Surv.* 38.4 (grud. 2006). ISSN: 0360-0300.

[2] Christopher T. Johnston, Kim T Gribbon i Donald G. Bailey. „FPGA based Remote Object Tracking for Real-time Control". W: *st International Conference on Sensing Technology*. 2005, s. 66–72.

[3] P. Pérez i in. „Color-based probabilistic tracking". W: *In Proc. ECCV*. 2002, s. 661–675.

[4] Bruce D. Lucas i Takeo Kanade. „An Iterative Image Registration Technique with an Application to Stereo Vision". W: *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'81. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1981, s. 674–679.

[5] Carlo Tomasi i Takeo Kanade. *Detection and Tracking of Point Features*. Spraw. tech. International Journal of Computer Vision, 1991.

[6] Dorin Comaniciu, Visvanathan Ramesh i Peter Meer. „Kernel-Based Object Tracking". W: *IEEE Trans. Pattern Anal. Mach. Intell.* 25.5 (maj 2003), s. 564–575. ISSN: 0162-8828.

[7] P. Pérez i in. „Color-based probabilistic tracking". W: *In Proc. ECCV*. 2002, s. 661–675.

[8] Peihua Li, Tianwen Zhang i Arthur EC Pece. „Visual contour tracking based on particle filters". W: *Image and Vision Computing* 21.1 (2003), s. 111–123.

[9] Neil Gordon. *Beyond the Kalman filter : particle filters for tracking applications*. 2004.

[10] *Zynq-7000 All Programmable SoC Overview*. 2016.

[11] *7 Series FPGAs Configurable Logic Block User Guide*. 2014.

[12] *7 Series FPGAs SelectIO Resources User Guide*. 2015.

[13] *7 Series FPGAs DSP48E1 Slice User Guide*. 2014.

[14] *7 Series FPGAs Memory Resources User Guide*. 2014.

[15] *Vivado Design Suite User Guide: Design Flows Overview*. 2016.

[16] Louise H. Crockett i in. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. UK: Strathclyde Academic Media, 2014. ISBN: 099297870X, 9780992978709.

[17] *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. 2012.

[18] *Cortex A9, Technical Reference Manual*. 2012.

[19] *Jazelle. https://www.arm.com/products/processors/technologies/jazelle.php*. Dostęp: 25 września 2016r.

[20] *ZYBO FPGA Board Reference Manual*. 2016.

[21] Jason Schlessman i in. „Hardware/Software Co-Design of an FPGA-based Embedded Tracking System". W: *Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*. CVPRW '06. Washington, DC, USA: IEEE Computer Society, 2006, s. 123–. ISBN: 0-7695-2646-2.

[22] Matteo Tomasi, Shrinivas Pundlik i Gang Luo. „FPGA–DSP co-processing for feature tracking in smart video sensors". W: *Journal of Real-Time Image Processing* (2014).

[23] Manoj Pandey i in. „Real time histogram computation in kernel based tracking system". W: *Advanced Electronic Systems (ICAES), 2013 International Conference on*. IEEE. 2013, s. 171–174.

[24] Dang Ba Khac Trieu i Tsutomu Maruyama. „An Implementation of the Mean Shift Filter on FPGA." W: *FPL*. IEEE Computer Society, 2011, s. 219–224. ISBN: 978-1-4577-1484-9.

[25] Jung Uk Cho i in. „Multiple objects tracking circuit using particle filters with multiple features". W: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE. 2007, s. 4639–4644.

[26] Markus Happe, Enno Lübbers i Marco Platzner. „A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking". W: *Journal of real-time image processing* 8.1 (2013), s. 95–110.

[27] *High-Definition Multimedia Interface, Specification Version 1.3a*. 2006.

A. Tutoriale

Dodatek ten zawiera kilka prostych tutoriali, które wprowadzają nowe technologie używane podczas tworzenia projektów dla Zynq.

A.1. Utworzenie projektu

Podstawowym środowiskiem pracy przy Zynq jest Vivado Design Suite. Autor podczas tworzenia niniejszego tutoriala korzystał z wersji 2016.2, więc wszelkie szczegółowe instrukcje odnoszą się właśnie do tej wersji. Inne wydania tego środowiska mogą zawierać pewne różnice, jednak ogólna idea powinna pozostać niezmieniona.

Jedną z rzeczy podawaną podczas tworzenia projektu, jest model układu albo płytki deweloperskiej. Niestety, domyślnie Vivado nie posiada skonfigurowanej płytki Zybo. Krótka instrukcja, jak ją dodać do środowiska przedstawiona jest pod adresem *https://reference.digilentinc.com/reference/software/vivado/board-files?redirect=1id=vivado/boardfiles2015*.

Po uruchomieniu programu i wybraniu opcji *Create New Project* należy przejść przez kreator pamiętając, aby na odpowiednim ekranie wybrać płytkę Zybo (rysunek A.1).

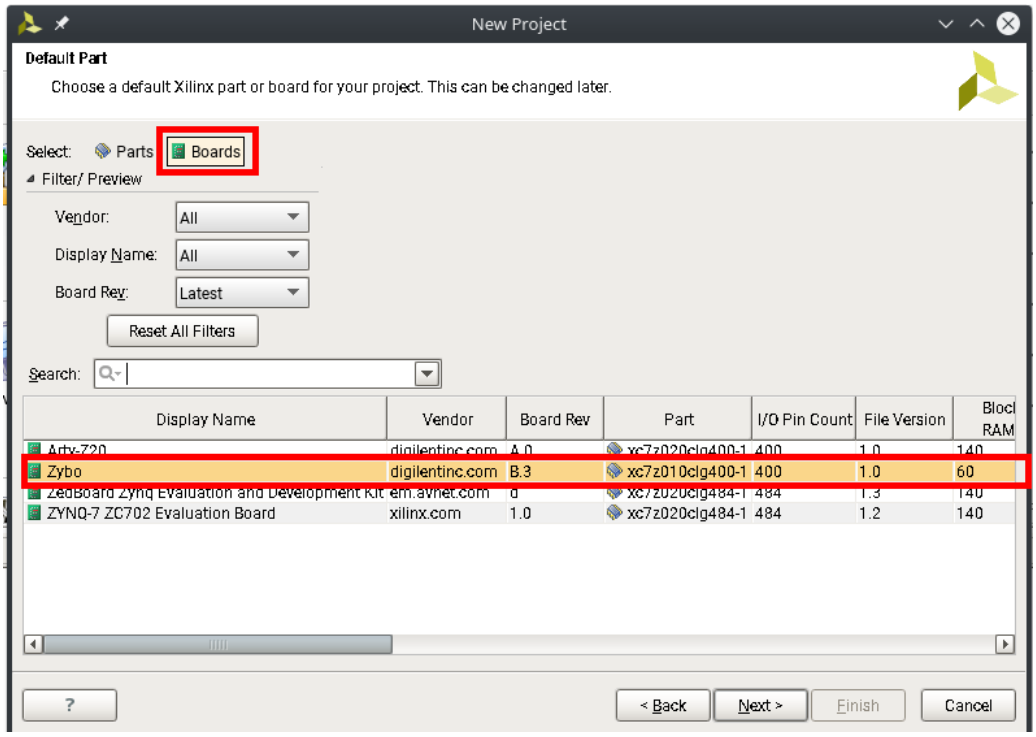
A.2. Komunikacja logiki programowalnej z procesorem

Projektowanie dla Zynq składa się zwykle z trzech etapów: projektowania logiki (FPGA), stworzenia części programowej (ARM) oraz dodania do projektu komunikacji pomiędzy tymi częściami. Dwa pierwsze etapy powinny być każdemu czytelnikowi znane, jednak sposoby komunikacji pomiędzy nimi najprawdopodobniej są pewną nowością.

W efekcie realizacji niniejszej instrukcji powstanie prosty program zapalający diody na płytce oraz odczytujący stan przełączników. Dzięki temu, że projekt ten nie posiada skomplikowanej logiki, będzie można łatwiej w ten sposób poznać tajniki komunikacji między opisywanymi układami.

W celu projektowania programów dla układów Zynq nie stosuje się zazwyczaj modułów pisanych ręcznie w Verilogu, a wykorzystuje się diagramy blokowe (*Block Design*). Jest to graficzna metoda budowy logiki za pomocą łączenia ze sobą gotowych modułów IP. Można oczywiście dodawać do biblioteki własne moduły i to właśnie będzie pierwszym celem tutoriala.

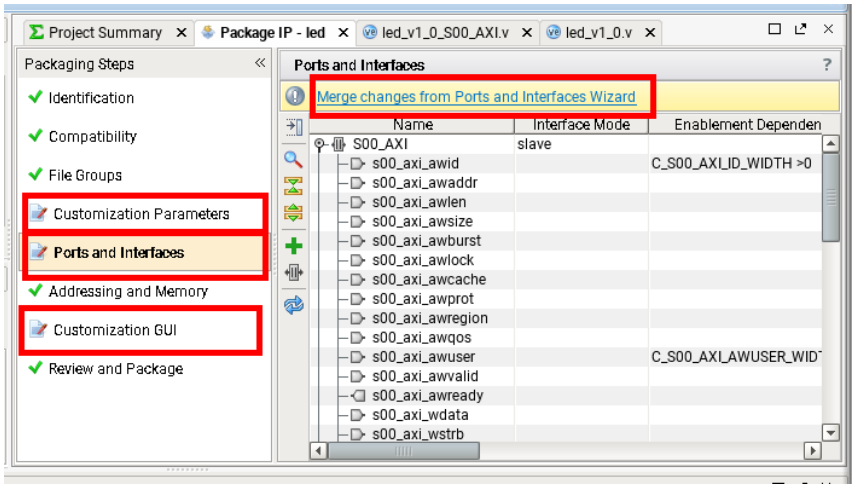
Moduł taki można utworzyć wybierając z menu głównego Tools->Create and Package IP... Otwiera się wtedy kreator tworzenia nowego projektu, w którym bardzo ważnym jest wybranie, że chcemy utworzyć nowe peryferium AXI, a także wybór typu interfejsu Full oraz ustalenie, że chcemy utworzony moduł edytować.



Rys. A.1. Tworzenie projektu

Po przejściu kreatora, otwiera się nowe okno projektu wraz z dwoma wstępnie uzupełnionymi plikami zawierającymi standardową implementację interfejsu AXI4. Autor proponuje zrobić jednak pewną modyfikację w domyślnie utworzonym kodzie. Standardowo, układ posiada jedną pamięć służącą do odczytu oraz zapisu danych, jednak dla naszego zastosowania jest to zupełnie nieprzydatna funkcjonalność. Łatwiej jest, gdy za pomocą zapisu przekazuje się parametry, a odczytuje się tylko i wyłącznie wyniki działań. Aby osiągnąć taki cel należy w pliku o nazwie NAZWA_PROJEKTU_v1_0_S00_AXI.v zmienić kod za komentarzem “Example code to access user logic memory region” na następujący:

```
//Memory
reg [2 ** C_S_AXI_ADDR_WIDTH - 1 : 0] ramWrite = 0;
reg [2 ** C_S_AXI_ADDR_WIDTH - 1 : 0] ramRead = 0;

//Write to memory
always @(posedge S_AXI_ACLK)
begin
if (axi_wready && S_AXI_WVALID)
begin
 ramWrite[axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] * 32 +: 32] <= S_AXI_WDATA;
end
end

//Read from memory
always @(posedge S_AXI_ACLK)
```



Rys. A.2. Tworzenie paczki IP

```
begin
if (axi_arv_arr_flag && axi_rvalid)
begin
 axi_rdata <= ramRead[axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] * 32 +: 32];
end
end
```

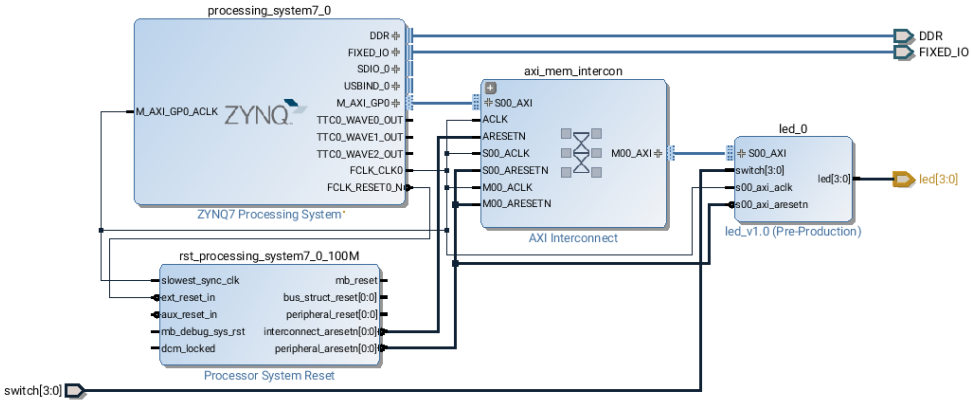
Dodatkowo należy dodać prostą logikę do odczytywania przełączników oraz zapalania diod. W ramach kolejnych testów warto tam zmieniać adresy aby sprawdzić, czy także będą poprawnie działać. Wcześniej utworzony kod należy więc uzupełnić za pomocą poniższego:

```
 assign led = ramWrite[3:0];

 always @(posedge S_AXI_ACLK)
 begin
        ramRead[3:0] = switch[3:0];
 end
```

Niezbędne porty należy teraz dodać w dwóch miejscach – w aktualnie edytowanym pliku oraz w pliku nadrzędnym o nazwie NAZWA_PROJEKTU_v1_0.v. W tym drugim należy także dodać odpowiednie połączenie, aby dane te były przekazywane do modułu podrzędnego. Realizację tego zadania autor pozostawia jako ćwiczenie.

Ostatnim etapem tworzenia modułu IP jest jego paczkowanie. Wykonuje się to w domyślnie otwartej zakładce *Package IP – NAZWA*. Specjalnie zaznaczone są tam sekcje, w których należy wykonać jakąś czynność, jak to jest przedstawione na rysunku A.2. Proces kończy się za pomocą wybrania *Re-Package IP* i zamknięcia projektu.

Posiadając już własny pakiet IP, należy odpowiednio go umieścić wewnątrz diagramu blokowego, który wpierw należy utworzyć za pomocą opcji *Create Block Design*. Dostajemy wtedy możliwość dodawania do niego modułów IP. Na liście dostępnych pakietów należy odszukać utworzony przez nas moduł, a następnie nanieść go na diagram. Kolejnym blokiem, który należy dodać jest *ZYNQ7 Processing System* reprezentujący procesor znajdujący się w układzie oraz umożliwiający jego konfigurację. Po jego dodaniu dostaje się możliwość skorzystania z opcji *Run Block Automation* oraz *Run Connection Automation*, dzięki którym wszystkie pozostałe niezbędne połączenia zostaną wykonane za nas automatycznie. Teraz aby dostać działający system potrzebujemy jeszcze dodać



Rys. A.3. Diagram blokowy układu

odpowiednie porty obsługujące diody oraz przełączniki. Można to zrobić wybierając opcję *Create Port* z menu kontekstowego wyjścia modułu. W efekcie powinniśmy dostać diagram podobny do tego na rysunku A.3.

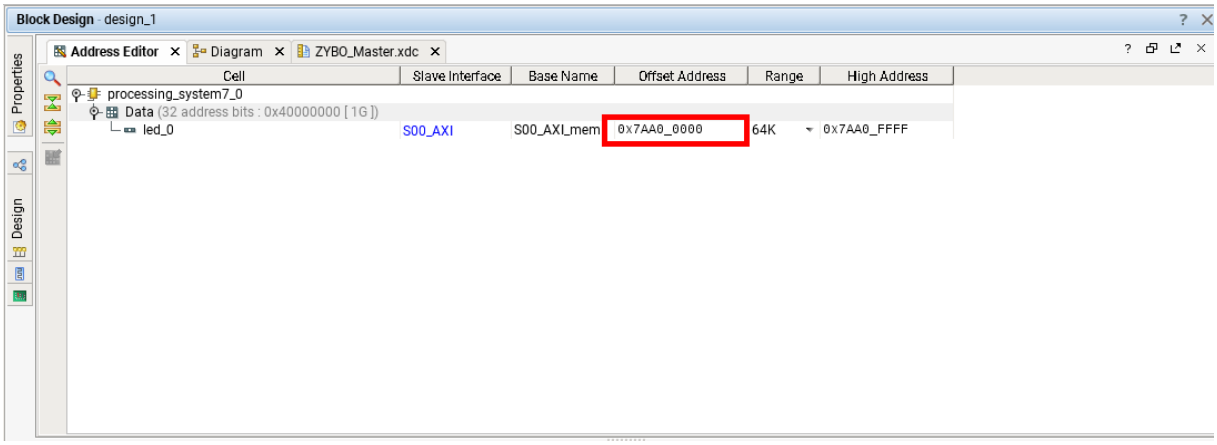
Jak wiadomo ze standardowego projektowania układów FPGA, aby porty mogły być reprezentowane przez pewne fizyczne złącze niezbędne jest umieszczenie odpowiedniego wpisu w pliku ograniczeń użytkownika. Domyślny plik dla płytki deweloperskiej Zybo z zakomentowanymi wszystkimi portami dostępny jest do ściągnięcia na stronie internetowej *https://github.com/Digilent/ZYBO/tree/master/Resources/XDC*. Należy dodać go teraz do projektu oraz odkomentować wszystkie wpisy odpowiedzialne za diody oraz przełączniki. Aby projekt z diagramu blokowego poprawnie się implementował, należy dodatkowo utworzyć jego HDL Wrapper. W tym celu trzeba odszukać plik diagramu blokowego wśród wszystkich plików źródłowych, a następnie wybrać opcję Create HDL Wrapper. Warto zostawić tam opcję automatycznej jego aktualizacji.

Część uruchamiana w logice programowalnej jest już praktycznie gotowa – wystarczy tylko wygenerować bitstream za pomocą opcji Generate Bistream i trochę poczekać.

W tym momencie nadszedł czas by zrealizować programową część tutoriala. Pierwszym krokiem będzie utworzenie odpowiedniego projektu. Z menu Vivado wybieramy File->Export->Export Hardware zaznaczając dodatkową opcję Include Bitstream. Następnie uruchamiamy SDK za pomocą File->Launch SDK. Uruchamia się wtedy program Xilinx SDK będący zmodyfikowaną wersją znanego IDE Eclipse.

Nowy projekt tworzymy za pomocą File->New->Application Project. Autor poleca w tym miejscu wybrać jako język programowania C++, ponieważ w dalszych projektach właśnie on będzie wykorzystywany. Bardzo ważną czynnością jest odszukanie adresu w pamięci, za pomocą którego należy komunikować się z utworzonym modułem. Powinien się on znajdować w pliku nagłówkowym xparameters.h pod nazwą XPAR_NAZWA_0_S00_AXI_BASEADDR, jednak przez pewne błędy w środowisku programistycznym nie zawsze wszystko poprawnie się generuje. Wtedy adresu należy szukać w Vivado w zakładce Address Editor (rysunek A.4).

Procedura zapisu za pomocą AXI do logiki programowalnej wykonywana jest za pomocą funkcji Xil_Out32() znajdującej się w pliku nagłówkowym xil_io.h, a odczytu za pomocą Xil_In32(). Prosty program, który będzie co pewien czas odczytywał wskazania na przełącznikach, a następnie zaświecał odpowiadające im diody zaprezentowany jest poniżej:



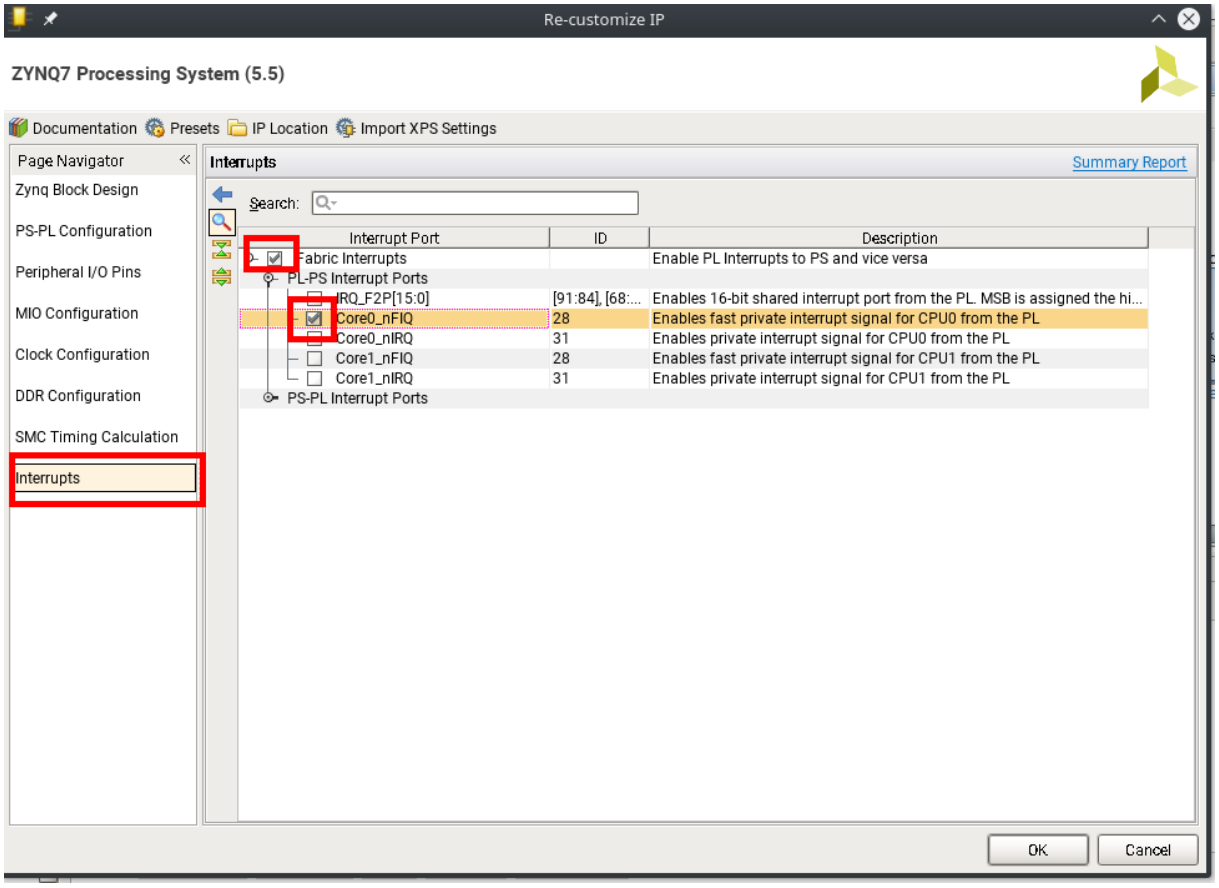
Rys. A.4. Okno edycji adresów w pamięci

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

//simple sleep implementation
void wait(unsigned steps) {
        volatile unsigned i;
        for (i = 0; i~< steps; i++);
}

int main()
{
        while (true) {
                //read switches
                u32 value = Xil_In32(XPAR_LED_0_S00_AXI_BASEADDR);
                //write to leds
                Xil_Out32(XPAR_LED_0_S00_AXI_BASEADDR, value);
                //wait for a~moment
                wait(10000000);
        }
}
```

Etap programowania jest dwuczęściowy – najpierw należy zaprogramować układ FPGA, a uruchomić aplikację na procesor. Logikę programuje się za pomocą opcji Xilinx Tools -> Program FPGA, a w celu wysłania programu na procesor klika się prawym klawiszem myszy na plik nazwa.elf znajdujący się w katalogu Binaries i wybiera się opcje Run As -> Launch On Hardware (GDB). Warto tutaj zaznaczyć, że można także zamiast Run As wybrać Debug As, aby uzyskać możliwość korzystania z wbudowanego debuggera.

A.2.1. Uruchomienie przerwań

W utworzonym na potrzeby niniejszej pracy projekcie dodatkowym sposobem komunikacji pomiędzy układem FPGA a procesorem są przerwania. Są to asynchroniczne sygnały, będące w stanie zmienić aktualnie wykonywany



Rys. A.5. Uruchomienie przerwań

na procesorze program na procedurę obsługi przerwania. W systemie procesorowym Zynq znajduje się układ zwany GIC (ang. *Generic Interrupt Controller*), którego zadaniem jest zbieranie możliwych sygnałów przerwań z różnych źródeł oraz zarządzanie nimi.

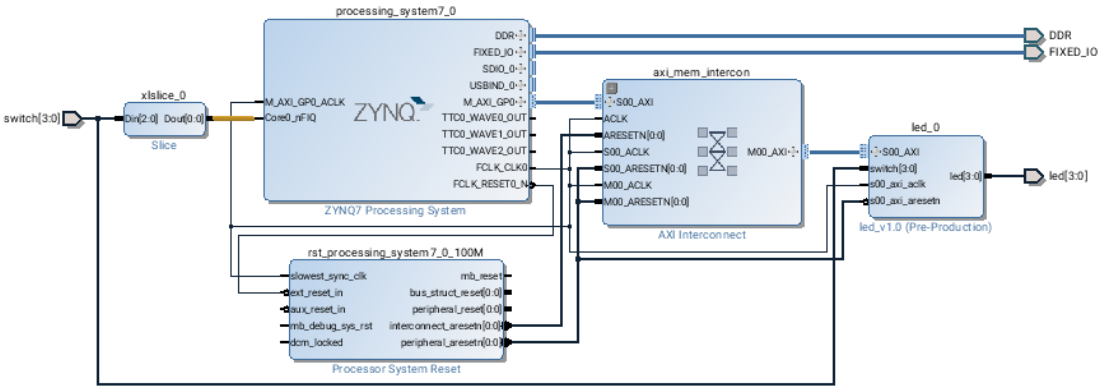
Proces uruchamiania przerwań z logiki programowalnej do procesora należy rozpocząć od odpowiedniego skonfigurowania modułu IP Zynq7 w logice programowalnej. Proces ten przedstawiony jest na rysunku A.5.

W celu testowania autor poleca połączyć wejście tego przerwania z jednym z przełączników, aby dało się je w łatwy sposób ręcznie wywoływać. Pewną trudnością może być fakt, że w układzie posiadamy port przełączników, który posiada szerokość 4 bitów, a wejście przerwania ma tylko 1 bit. Aby w diagramie blokowym podzielić sygnał na składowe należy użyć bloku IP o nazwie Slice. Połączony układ powinien wyglądać mniej więcej jak ten przedstawiony na rysunku A.6.

Po wygenerowaniu bitstreamu i jego wyeksportowaniu do SDK pozostaje odpowiednio obsłużyć przerwania na procesorze. Proponowany program testowy będzie zliczał liczbę wywołanych przerwań, a następnie wysyłał odpowiednią wartość do logiki programowalnej aby wyświetliła ją za pomocą diod.

Proces inicjalizacji przerwań składa się z czterech etapów:

1. pobrania konfiguracji GIC,
2. inicjalizacji GIC,
3. rejestracji procedury obsługi przerwania,
4. włączenia przerwań.



Rys. A.6. Diagram blokowy po uruchomieniu przerwań

Proces ten wraz z odpowiednimi komentarzami znajduje się poniżej:

```
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_io.h"
#include "xparameters.h"

unsigned interrupts;
void handler() {
interrupts++;

Xil_Out32(XPAR_LED_0_S00_AXI_BASEADDR, interrupts);

//Wyczyszczenie flagi przerwan
IntIDFull = XScuGic_CPUReadReg((XScuGic *)data, XSCUGIC_INT_ACK_OFFSET);
   XScuGic_CPUWriteReg((XScuGic *)data, XSCUGIC_EOI_OFFSET, IntIDFull);
}

void initializeInterrupts() {
s32 status;
//wygenrowanie konfiguracji GIC (ukladu obslugi przerwan)
XScuGic_Config *config = XScuGic_LookupConfig(DEVICE_ID);

//inicjalizacja przerwan
Xil_ExceptionInit();
//inicjalizacja GIC
status = XScuGic_CfgInitialize(&xscuInstance, config, config->CpuBaseAddress);
if (status != XST_SUCCESS) {
 xil_printf("Error CfgInitialize: %d\n", status);
 return 1;
}
```

```
 //Rejestracja procedury obslugi przerwania
 Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_FIQ_INT,
              handler,
              &xscuInstance);

 //Uruchomienie przerwan
 Xil_ExceptionEnableMask(XIL_EXCEPTION_ALL);
}

 }

 int main() {
 initializeInterrupts();
 while (true);
 }
```

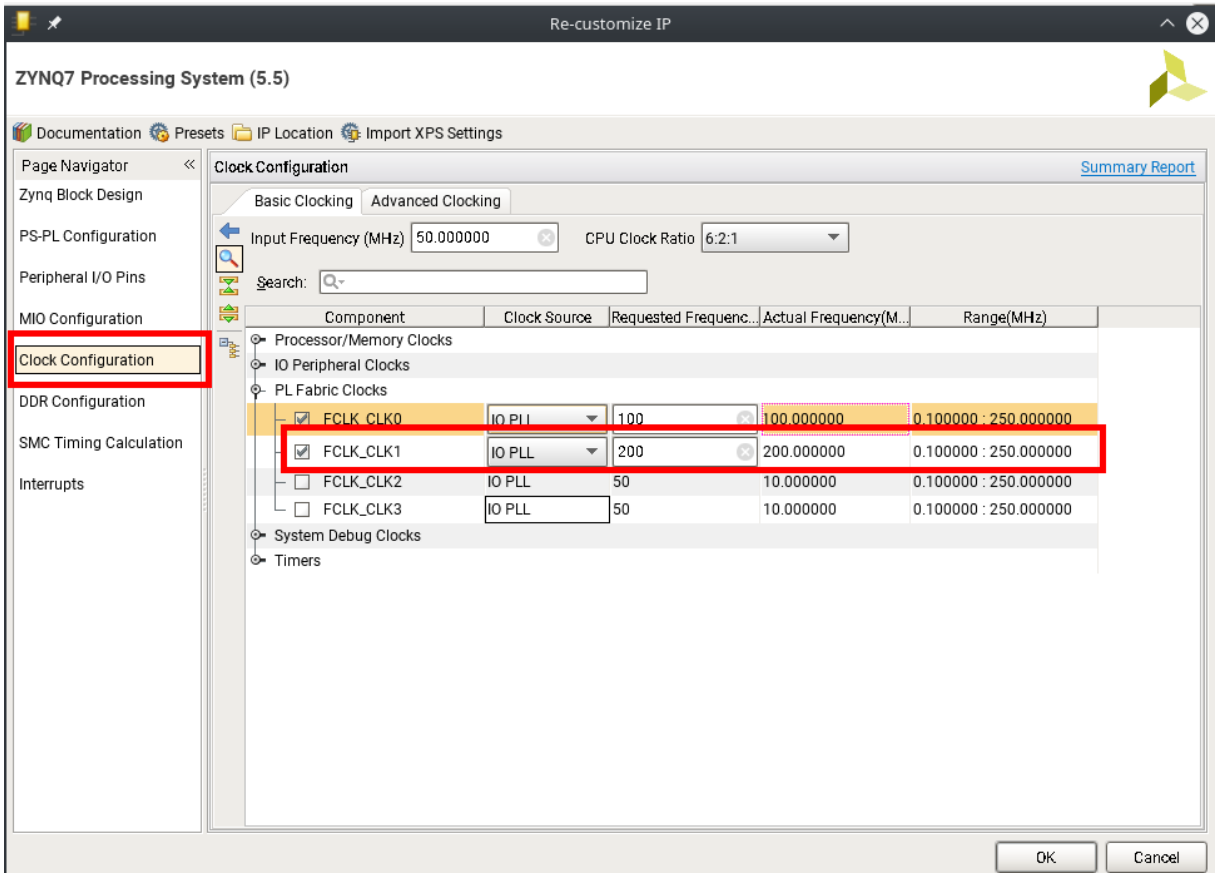
A.3. Tor wizyjny

Jedną z kluczowych części projektu było uruchomienie toru wizyjnego, a więc odczytywania obrazu z sygnału HDMI, dodanie wizualizacji, a następnie obsłużenie wyjścia VGA.

W przypadku obsługi zarówno HDMI, jak i VGA, warto wykorzystać gotowy przykład od firmy Digilent, w którym jest to wykonane. Znajduje się on na stronie internetowej *https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-hdmi-demo/start*. W przykładzie znajduje się o wiele więcej funkcjonalności niż jest wymagane w rozważanym projekcie, dlatego warto zwrócić uwagę jedynie na bloki IP dvi2rgb oraz rgb2vga wraz z ich obsługą. Najważniejszą rzeczą, którą należy tam zrobić jest podłączenie i konfiguracja zegara o częstotliwości 200Hz generowanego w Processing Systemie układu Zynq. To, jak należy to zrobić przedstawione jest na rysunku A.7.

Moduł wizualizacji to prosty moduł posiadający możliwość rysowania prostokąta w zadanym przez procesor miejscu. Informację o tym, gdzie ma się on znajdować przekazywana jest za pomocą interfejsu AXI-Lite z czterema rejestrami, w których zapisane są współrzędne dwóch rogów prostokąta. Dodatkową funkcjonalnością jest rysowanie siatki 200x200 pikseli w celu ułatwienia użytkownikowi lokalizacji obiektów na ekranie. Kod odpowiadający za rysowanie przedstawiony jest poniżej, a całość kodu tego modułu można znaleźć na dołączonej płycie CD.

```
assign r_out = (x >= left && x <= right && (y == top || y == bottom)) ||
       (y >= top && y <= bottom && (x == left || x == right)) ||
       x % 200 == 0 || y % 200 == 0
       ? 8'hff : r;
```



Rys. A.7. Konfiguracja zegara dla układu HDMI

```
module histogram #(
parameter HISTOGRAM_SIZE = 36
parameter HISTOGRAM_WIDTH = 6
)
(
input clk,
input [HISTOGRAM_SIZE - 1:0] currentBin,
input insideWindow,
input reset,
output[HISTOGRAM_SIZE * HISTOGRAM_WIDTH - 1: 0] histogram
);

genvar i;
generate
for (i = 0; i~< HISTOGRAM_SIZE; i~= i~+ 1)
begin: counters
wire useThisCounter;
assign useThisCounter = insideWindow && currentBin[i] == 1;

histogram_counter counter (
.CLK(clk),
.CE(useThisCounter),
.SCLR(reset),
.Q(histogram[(i + 1) * HISTOGRAM_WIDTH - 1: i~* HISTOGRAM_WIDTH])
);
end
endgenerate

endmodule)
```

Listing 4. Obliczanie histogramu

B. Zawartość płyty CD

Dołączona do pracy płyta CD zawiera:

– treść pracy w formacie PDF,

– kod źródłowy projektu