



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Projekt dyplomowy

*Detekcja obiektu o kulistym kształcie w układzie
rekonfigurowalnym*

*Detection of a spherical shape object with in a reconfigurable
unit*

Autor:	<i>Konrad Wajda</i>
Kierunek studiów:	<i>Automatyka i Robotyka</i>
Opiekun pracy:	<i>Dr inż. Krzysztof Kołek</i>

Kraków, 2019

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy

Składam serdeczne podziękowania promotorowi dr inż. Krzysztofowi Kołkowi za poświęcony mi czas, ogromną pomoc oraz cierpliwość do mnie podczas tworzenia niniejszej pracy.

Dziękuję również recenzentowi, dr inż. Maciejowi Rosołowi za trud włożony w ocenę niniejszej pracy.

Spis treści

Rozdział 1 Wstęp.....	5
1.1 Cel pracy	5
1.2 Rozwinięcie problemu.....	5
1.3 Kolejne rozdziały	5
Rozdział 2 Osadzenie pracy w kontekście rozwoju techniki	7
2.1 Przetwarzanie obrazu	7
2.2 Układy rekonfigurowalne	7
2.3 Opis układu realizującego zagadnienie	8
2.4 Przetwarzanie i analiza obrazów w kontekście układów rekonfigurowalnych	11
Rozdział 3 Realizacja części sprzętowej pracy dyplomowej	15
3.1 Wstęp.....	15
3.2 Zmiana przestrzeni barw RGB -> YCbCr.....	16
3.3 Binaryzacja	18
3.4 Filtracja	19
3.5 Wyliczenie środka ciężkości	21
3.6 Moduł Rejestru 32-bitowego	25
3.7 Magistrala AXI	25
3.8 Dodanie układu Zynq oraz wygenerowanie pliku bitfile.....	27
Rozdział 4 Realizacja części programowej pracy dyplomowej	29
4.1 Wstęp.....	29
4.3 Połączenie TCP/IP.....	31
4.4 Mapowanie obszarów pamięci	32
4.7 Przetwarzanie plików BMP.....	33
4.5 API kamery	33
4.6 Aplikacja klienta	34
4.7 Aplikacja serwera	35
Rozdział 5 Przedstawienie wyników.....	37
5.1 Wstęp.....	37
5.2 Model programowy	37
5.3 Symulacje w środowisku Vivado	38
5.3 Logowanie danych w testowej aplikacji Serwera	39
5.4 Weryfikacja działania na podstawie prostych obrazów testowych	39
5.6 Konkluzje.....	41
Rozdział 6 Zakończenie	43
Bibliografia	45
Załączniki.....	47

Rozdział 1 Wstęp

1.1 Cel pracy

Celem niniejszego projektu dyplomowego jest opracowanie modułu dla układu FPGA wykrywającego położenie obiektu kulistego w obrazie wideo. W module należy wykorzystać charakterystyczny kolor/kształt obiektu. Opracowany moduł ma współpracować z magistralą AXI. Transfer klatek obrazu do układu FPGA oraz zwrot informacji o wykrytym położeniu ma następować za pomocą aplikacji dla procesora ARM poprzez łącze Ethernet. Układ ma zostać zaimplementowany na płycie UltraZed.

1.2 Rozwinięcie problemu

Podstawową kwestią która wymaga wyjaśnienia jest zdefiniowanie czym jest *położenie obiektu*. W niniejszej pracy przyjęto, iż jest to geometryczny środek ciężkości danego obiektu, który wyliczany jest przy pomocy wzorów bazujących na momentach geometrycznych ukazanych w rozdziale III – wzory (2),(3),(4),(5),(6) oraz (7). Położeniem początkowym jest lewy górny róg każdego obrazu którego współrzędne wynoszą $(0,0)$. *Położenie X* zdefiniowano jako położenie horyzontalne odnoszące się do numeru piksela reprezentującego geometryczny środek ciężkości. Analogicznie *położenie Y* odnosi się do położenia wertykalnego reprezentującego geometryczny środek ciężkości. Tak więc położenie obiektu jest parą liczb naturalnych, których wartości są ograniczone z dołu przez 0 oraz z góry przez rozdzielczość przetwarzanego obrazu w odpowiednim wymiarze.

Realizację pracy można podzielić na dwie części – część sprzętową oraz programową. Pierwsza z nich obejmuje utworzone moduły FPGA, ich wzajemne relacje i zależności jak również fizyczne połączenie obydwu części, druga zaś aplikacje przeznaczone na zarówno procesor ARM jak i komputer nadzorujący.

1.3 Kolejne rozdziały

- Rozdział II : Osadzenie pracy w kontekście rozwoju techniki – Rozdział poświęcono przetwarzaniu obrazów w kontekście układów rekonfigurowalnych, omówiono układ UltraZed na którym zaimplementowano pracę.
- Rozdział III: Realizacja części sprzętowej pracy dyplomowej – Rozdział poświęcono poszczególnym podmodułom FPGA wchodzącym w skład całego modułu realizującego wykrywanie środka ciężkości, omówiono sposób generacji pliku wynikowego oraz dokonania sprzętowego połączenia pomiędzy układem procesorów ARM a układem FPGA.
- Rozdział IV: Realizacja części programowej pracy dyplomowej – Rozdział poświęcono generowaniu systemu operacyjnego dla procesorów ARM, dwóm aplikacjom wchodzącym w skład pracy oraz omówiono ich najważniejsze funkcjonalności.
- Rozdział V: Wyniki – Rozdział poświęcono sposobom testowania stworzonego układu, ich wynikom oraz konkluzjom z nich wynikającym.

- Rozdział VI: Zakończenie – Rozdział poświęcono wskazaniu czy cel pracy został zrealizowany, możliwemu przyszłemu rozwojowi pracy jak również wnioskom płynącym z pracy z układami tego typu.
- Bibliografia: Spis wykorzystywanych źródeł informacji.
- Załączniki: Tabele zawierające zdjęcia wykorzystane przy testowaniu utworzonego układu.

Rozdział 2 Osadzenie pracy w kontekście rozwoju techniki

2.1 Przetwarzanie obrazu

Dwudziesty pierwszy wiek przyniósł ogromne zmiany w kontekście rozwoju technologii. Stale powiększająca się moc obliczeniowa nowoczesnych komputerów jak również coraz dalej idąca miniaturyzacja układów scalonych spowodowała, iż to co dawniej wydawało się niewykonalne ze względu na ograniczenia technologiczne, dzisiaj staje się coraz bardziej prawdopodobne do wykonania. Doprowadziło do coraz szybszego rozwoju tych dziedzin techniki które posiadają największe zapotrzebowanie na moc obliczeniową i szybkość działania, a które równocześnie zajmują strategiczną pozycję dla dalszego rozwoju ludzkości. Jedną z takich dziedzin jest przetwarzanie i analiza obrazu. Już teraz znajduje ona zastosowanie w ogromnej liczbie dziedzin naszego życia a zapotrzebowanie na jej wykorzystanie będzie tylko i wyłącznie wzrastać. Wykrywanie raka oraz innych schorzeń dających fizyczne objawy na ludzkim ciele, analiza zdjęć z satelit w celu odnalezienia interesujących nas obiektów czy, aby nie szukać daleko, nakładanie filtrów i elementów wirtualnych na obraz z aparatu w naszych telefonach to wszystko zasługa właśnie tej dziedziny techniki. Najbardziej dynamiczny rozwój przewiduje się jednak w dziedzinie motoryzacji. Nowoczesne samochody wykrywające i identyfikujące znaki drogowe albo przeszkody na swojej drodze w formie pieszych to właśnie efekt coraz większego zaawansowania tej dziedziny. Rozwój ten doprowadził jednak w pewnym momencie do zastoju. Problem okazał się być czas przetwarzania. Nowoczesne systemy powinny bowiem działać jak najszybciej, nawet lekkie opóźnienie w dostarczeniu danych mogłoby bowiem okazać się zgubne dla systemu, a w konsekwencji dla człowieka. Układy mikroprocesorowe, pomimo swojego coraz większego stopnia zaawansowania, nie potrafiły dokonać przetwarzania ciągłego strumienia danych w odpowiednim czasie. Wobec tego, konieczne okazało się znalezienie innego rozwiązania. Rozwiązaniem tym ukazały się układy rekonfigurowalne, a ściślej mówiąc, układy FPGA.

2.2 Układy rekonfigurowalne

Układem rekonfigurowalnym nazywamy rodzaj układu elektronicznego którego struktura może być rekonfigurowana co oznacza możliwość wielokrotnego zaprogramowania jej w taki sposób, aby pełniła różnorakie zadania zależne wyłącznie od projektanta jej systemu. Ich najpopularniejszą obecnie stosowaną odmianą są układy FPGA (*ang. Field Programmable Gate Array*), czyli Bezpośrednio Programowalna Macierz Bramek. Projektowanie układów tego typu najczęściej odbywa się przy pomocy języków opisu sprzętu, wśród których aktualnie największą popularnością cieszy się *Verilog* oraz *VHDL*. Wśród głównych elementów układów FPGA możemy wymienić (za [1]):

- CLB (*ang. Configurable Logic Block*) - Najważniejszy element każdego układu tego typu. To właśnie poprzez ich łączenie w odpowiedni sposób powstaje pożądany układ logiczny. Składają się z pomniejszych elementów, tzw. *slice* ów podłączonych matrycy przełączeń oraz szybkiej logiki przeniesienia. Te z kolei składają się z przerzutników,

LUT(ang. *Look Up Table*) oraz multiplexerów, których ilość zależna jest od rozpatrywanego układu.

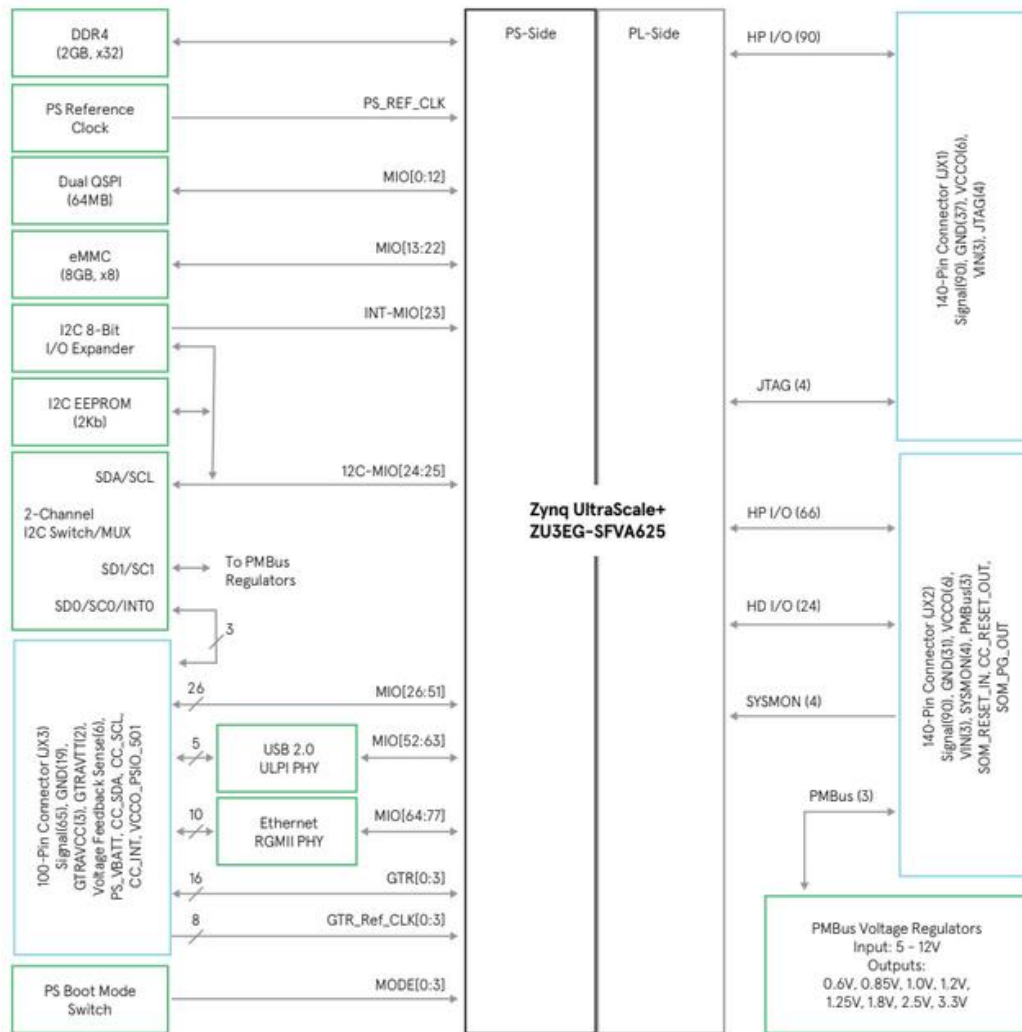
- CMT (ang. *Clock Managment Tiles*) - Bloki zarządzające sygnałem zegarowym. Pozwalają na generowanie sygnału taktującego o zadanej częstotliwości jak również równomierną propagację sygnału oraz tłumią zjawisko *Jitter*'a, czyli różnego rodzaju zakłócenia fazy zegara.
- BRAM (ang. *Block Random Access Memory*) - Pamięć blokowa, dedykowana dwuportowa pozwalająca na zapis oraz odczyt danych, która może być również skonfigurowana jako moduł FIFO (ang. *First-In,First-Out*). Jej pojemność jest zależna od rozpatrywanego układu.
- DSP48 - Jednostka arytmetyczno-logiczna (ang. *Arithmetic and Logical Unit*), pozwalająca na szybkie mnożenie liczb zawierająca również wbudowany akumulator.
- IOB (ang. *Input/Output Block*) - Bloki wejść-wyjść, podzielone na banki.

Najważniejszą cechą jak również największą zaletą układów rekonfigurowalnych jest ich równoległe działanie, a więc możliwość realizacji wielu różnych operacji jednocześnie. Wywodzi się to właśnie z ich budowy wewnętrznej i stanowi główny powód dla którego układy te są tak często wykorzystywane przy problemach wymagających obliczeń na dużej liczbie danych. Z uwagi na możliwość rekonfigurowania swojego wnętrza, a co za tym idzie, dużej dowolności w realizacji różnego rodzaju układów, cenione są również jako platformy do prototypowania różnego rodzaju układów scalonych, przed ich wypuszczeniem w krzemie.

Układy te mają jednak swoje negatywne strony, wśród których na w pierwszym szeregu wychodzi cena, która jest zdecydowanie wyższa niż tradycyjnych układów CPU jak również zmniejszona w stosunku do nich częstotliwość zegara taktującego. Z tego też powodu FPGA nie sprawdzają się najlepiej w przypadku algorytmów sekwencyjnych. Jednym z potencjalnych rozwiązań tego problemu jest połączenie zalet obydwu typów i wzajemne zredukowanie swoich negatywów poprzez układy zawierające w sobie zarówno FPGA jak i mikroprocesory. Tak powstały układy heterogeniczne SoC (ang. *System on Chip*). Są to kompletne układy scalone składające się z zarówno części rekonfigurowalnej jak i mikroprocesora oraz niezbędnych peryferiów w postaci GPIO (ang. *General Purpose Input/Output*), czy też pamięci.

2.3 Opis układu realizującego zadanie

Układem realizującym zadanie niniejszej pracy dyplomowej jest układ firmy Avnet, a konkretnie *UltraZed-EG SOM*. Jest to układ typu SOM(ang. *System-On-Module*), czyli rozwinięcie koncepcji układów SoC. W odróżnieniu od nich układy SOM posiadają również wbudowane dodatkowe komponenty w postaci chociażby regulatora napięcia, rezonatora itd., tworząc w ten sposób układ za pomocą którego można w pełni wykreować żądany produkt. Innymi słowy, SOM „obudowuje” SoC, tworząc z niego kompletne narzędzie do tworzenia systemów wbudowanych. Schemat układu przedstawiony został na *Rysunku 2.1*



Rysunek 2.1 Schemat układu UltraZed-EG-SOM [3]

Najważniejszym elementem tego układu jest MPSoC (ang. *Multiple System-on-Chip*), czyli system SoC zawierający więcej niż jeden mikroprocesor. W UltraZed jest to układ Xilinx Zynq® UltraScale+™ MPSoC, a konkretnie XCZUEG-1SFVA625. W Tabeli 2.1 przedstawiono jego najistotniejsze parametry.

Tabela 2.1 Parametry układu Xilinx Zynq® UltraScale+™ MPSoC

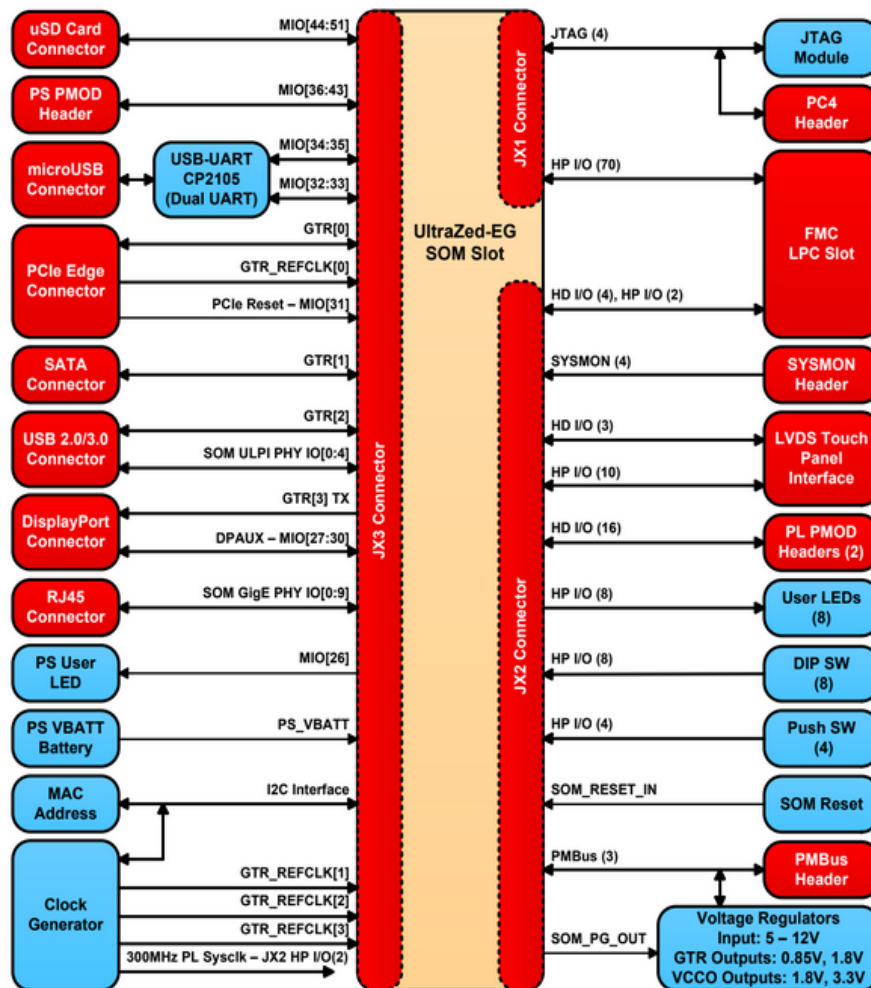
Układ mikroprocesorów	Quad-core ARM Cortex-A53
Liczba CLB (tysiące)	154
Pamięć wbudowana (Mb)	7.6
Liczba komórek DSP	240
Maksymalna liczba pinów I/O	252

Jak można zauważyć na schemacie, dodatkowe komponenty są dwojakiego rodzaju: przyporządkowane do warstwy PS (ang. *Processing System*), czyli części mikroprocesorowej bądź też do PL (ang. *Programmable Logic*), czyli części związanej z układem rekonfigurowalnym.

Z punktu widzenia realizowanego projektu pozostałymi ważnymi elementami układu są:

- warstwa Ethernetowa, konieczna do realizacji połączenia pomiędzy płytka a komputerem,
- wewnętrzny zegar, taktujący działanie całego układu,
- 4 DIP Switch`e, zezwalające na zmianę sposobu boot`owania całego systemu,
- sterowany przez warstwę PS interfejs JTAG pozwalający na zaprogramowanie układu.

Układ EG-SOM nie jest jednak dostosowany do pracy jako samodzielna płytka, czego powodem jest chociażby brak bezpośrednio dostępnych dla użytkownika wyprowadzeń peryferiów. W pracy z układem należy go zamontować na karcie wspomagającej (*ang. Carrier Card*), w tym wypadku był to układ *UltraZed™ PCIe Carrier Card* którego schemat przedstawiono na Rysunku 2.2

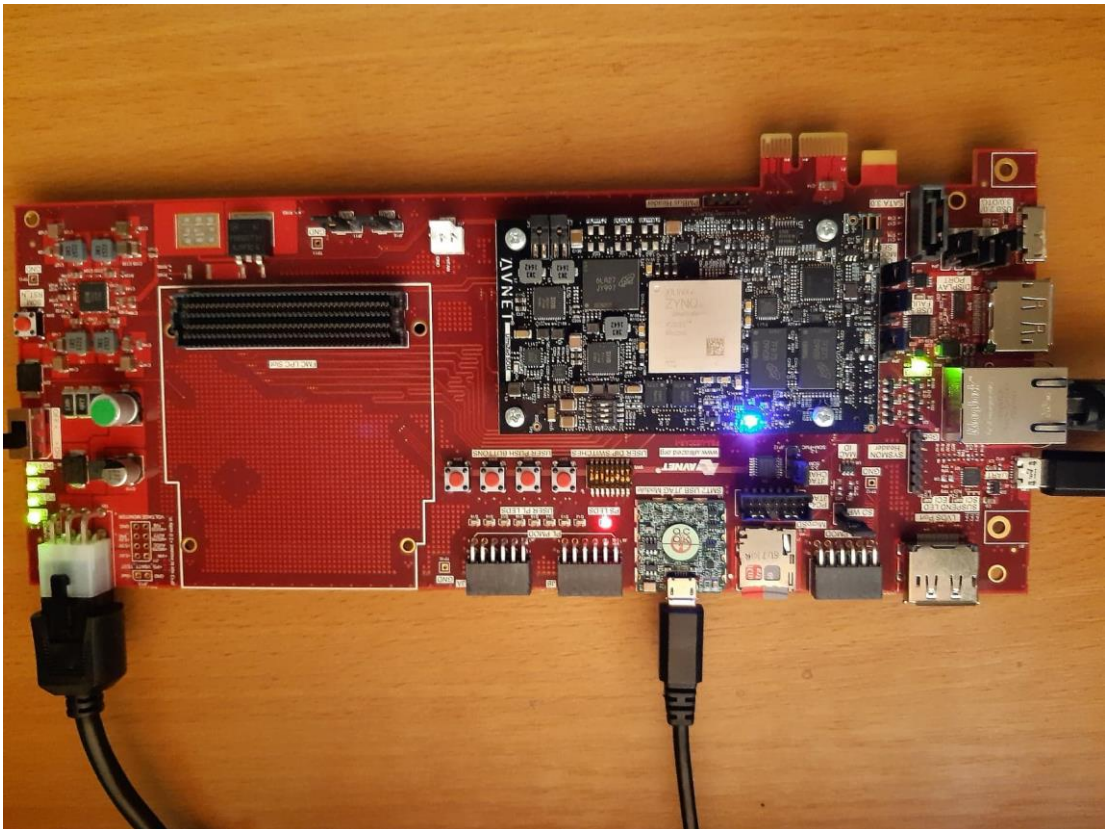


Rysunek 2.2 Schemat układu PCIe Carrier Card [2]

Połączenie pomiędzy układem SOM a kartą realizowane jest przy pomocy trzech konektorów TE 0.8mm FH, określanych jako *JX1*, *JX2* oraz *JX3*. Każde z nich odpowiedzialne jest za inne piny, co widoczne jest na powyższym schemacie. Pierwsze dwa z nich obsługują po 140 pinów, natomiast trzeci z nich 100.

Dzięki osadzeniu SOM na tej karcie, możliwe jest programowanie go przy użyciu JTAG, jak również przy użyciu plików konfiguracyjnych umieszczonych na specjalnej karcie typu

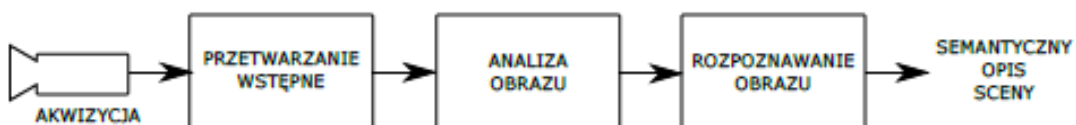
microSD dołączonej do zestawu. Karta wspomagająca umożliwia również połączenie się układu z komputerem poprzez UART, jak również poprzez Ethernet, co stanowi jeden z ważnych elementów całej pracy. Wprowadza również dodatkowe peryferia w postaci 4 przycisku typu *button*, 5 diod, z czego 4 z nich połączone są z warstwą PL (choć nic nie stoi na przeszkodzie by sterować nimi za pomocą mikroprocesorów) oraz jednej sterowanej z warstwy PS. Oddaje do dyspozycji użytkownika również 8 DIP Switch'y, przycisku *Reset* oraz, co bardzo istotne, gniazdo zasilające +12V DC. Na *Rysunku 2.3* przedstawiono zdjęcie całej płytki wraz z nałożonym układem SOM w czasie pracy.



Rysunek 2.3 Układ w czasie pracy

2.4 Przetwarzanie i analiza obrazów w kontekście układów rekonfigurowalnych

Typowy układ przetwarzania oraz analizy obrazów składa się z systemu dokonującego akwizycji danych, którym najczęściej jest kamera, elementu który realizuje wymagane obliczenia na dostarczonych danych oraz ewentualnie urządzenia wizualizującego dokonania np. monitora. Taki układ zwykle się nazywa potokowym systemem przetwarzania i analizy obrazów (za [1]). Ilustruje go *Rysunek 2.4*.

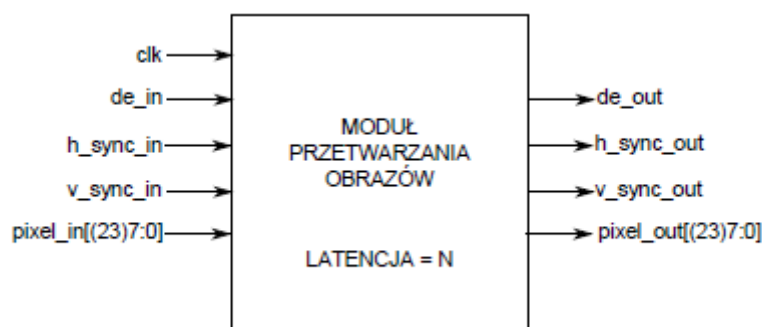


Rysunek 2.4. Schemat potokowego systemu przetwarzania i analizy obrazów [1]

Przymiotnik „potokowy” w nazwie systemu bierze swoją nazwę z przetwarzania potokowego, czyli jednego ze sposobów przetwarzania danych. Sposób działania jest następujący: Cały cykl pracy układu podzielony jest na oddzielne bloki, z których każdy wykonuje oddzielne zadanie. Dane przechodzą z jednego bloku do drugiego, aż do ostatniego, z którego to uzyskuje się dane wyjściowe całego systemu. Warto wspomnieć jeszcze o dwóch bardzo ważnych parametrach charakteryzujących każdy system przetwarzania potokowego, a mianowicie latencji (ang. latency) oraz przepustowości (ang. throughput). Często terminy te używane są przemiennie, i choć w części przypadków ich wartości są równe, nie oznaczają one jednak tego samego. Latencję definiujemy jako liczbę cykli zegara mijających od pojawienia się danych na wejściu do ukazania się wyników przetwarzania na wyjściu. Przepustowość zaś to liczba cykli zegara po której możliwe jest wprowadzenie kolejnej porcji danych do systemu. W przypadku pojedynczego modułu wartości te są sobie tożsame, jednak wraz ze wzrostem liczby modułów składowych systemu o różnych latencjach, przepustowość całości zaczyna coraz bardziej różnić się od latencji. Przy tworzeniu systemu potokowego projektant powinien dokonywać optymalizacji pod względem któregoś z tych parametrów, pamiętając, że poprawa jednego z nich może wpłynąć negatywnie na drugi.

Jak wspomniano wyżej, w celu uzyskania informacji z obrazu dokonuje się jego przetwarzania oraz analizy. Główna różnica pomiędzy tymi obiema operacjami jest bardzo prosta: w wyniku przetwarzania obrazu na wyjściu modułu otrzymujemy obraz, natomiast wynikiem działania analizy są dane, jak chociażby właśnie położenie interesującego nas obiektu w obrębie kadru kamery. Godny uwagi jest fakt, iż w przypadku systemu wizyjnego opracowanego na systemie FPGA obliczenia dokonywane są bezpośrednio na strumieniu pikseli otrzymywanych z systemu akwizycji, na każdym z dostarczonych pikseli.

Najczęściej stosowany interfejs modułu układu FPGA realizującego potokowy system przetwarzania wizyjnego ukazano na *Rysunku 2.5* (za [1]) :



Rysunek 2.5 Schemat typowego modułu do przetwarzania obrazów [1]

Sygnały wejściowe modułu wyglądają następująco:

- clk - Sygnał zegarowy, taktujący moduł.
- de_in - Bit charakteryzujący ważność piksela, czyli poprawność jego akwizycji.
- h_sync_in - Sygnał synchronizacji pionowej.
- v_sync_in - Sygnał synchronizacji poziomej.

- pixel_in - Właściwe dane piksela, najczęściej w formie 24 (w przypadku piksela kolorowego) bądź też 8 bitów (dla danych przedstawionych w odcieniach szarości).

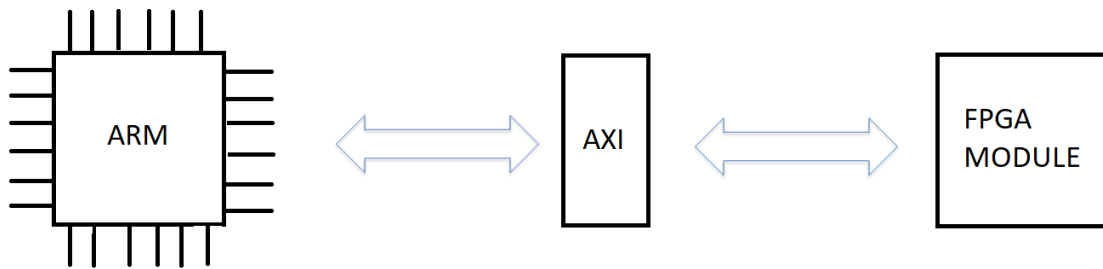
Jak można łatwo zauważyć, sygnały wyjściowe z modułu są bardzo zbliżone do danych wejściowych. Wynika to z faktu, iż wszystkie te dane poza samą wartością pikseli wychodzących są po prostu opóźnionymi sygnałami wchodzącymi. Opóźnienie to wynika z czasu przetwarzania danych zawartych we właściwych danych pikseli i zależy od złożoności algorytmu przetwarzającego, w przypadku prostych przetwarzań wynosi najczęściej kilka taktów zegara.

Zastanawiający może być jednak sam cel istnienia sygnałów kontrolnych, czyli *de*, *vsync* oraz *hsync*. Sygnały te wytwarzane są przez sam system akwizycji danych, czyli kamerę, i są one niezbędne do prawidłowego wyświetlania przez monitor otrzymanych danych. Sygnały synchronizacji pionowej jak i poziomej wywodzą się z pierwszych telewizorów analogowych, które działały na zasadzie „ostrzeliwania” ekranu wiązką elektronów. W celu poprawnego wyświetlania działko elektronowe poruszało się od lewej do prawej oraz z góry do dołu czym sterowały właśnie sygnały synchronizacji. Z kolei na czas powrotu działka na startową pozycję wygaszany był sygnał ważności bitu, co umożliwiało prawidłowe działanie całego systemu.

Rozdział 3 Realizacja części sprzętowej pracy dyplomowej

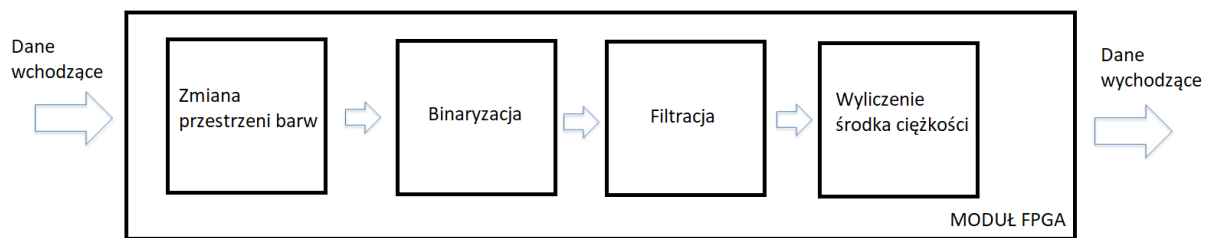
3.1 Wstęp

Jak wspomniano w rozdziale I, realizację tematyki pracy dyplomowej można podzielić na dwie części, tj. sprzętową oraz programową. W tym rozdziale opisana zostanie ta pierwsza, realizowana na rekonfigurowalnym układzie FPGA oraz jej połączenie z układem mikroprocesorów ARM. *Rysunek 3.1* pokazuje jej schemat.



Rysunek 3.1 Schemat połączeń układu

Najważniejszym elementem jest moduł FPGA widoczny z prawej strony schematu, realizujący obliczanie środka ciężkości. Składa się on z podmodułów, które to ukazane są na *Rysunku 3.2*.



Rysunek 3.4 Elementy składowe modułu FPGA

System składa się z czterech podmodułów w których przepływ danych następuje w sposób potokowy. Od lewej do prawej, każdy z modułów realizuje swoją funkcję:

- zmiana przestrzeni barw RGB na YCbCr,
- binaryzacja obrazu,
- filtracja obrazu,
- wyliczenie środka ciężkości.

Kolejne podrozdziały opisują każdy z tych modułów składowych po kolei jak również dodatkowy moduł pełniący pewnego rodzaju bufor danych dla modułu jak również jego kontroli. Przedostatni podrozdział poświęcony jest magistrali AXI-LITE realizującej wymianę danych pomiędzy układem FPGA a układem mikroprocesorów ARM Cortex A53, ostatni z kolei poświęcono sposobowi zbudowania całego *bitfile`a* włącznie z dodaniem układu PS. Podstawowa wersja modułu powstała w ramach zajęć z przedmiotu *Systemy rekonfigurowalne* i opisana jest w [1].

3.2 Zmiana przestrzeni barw RGB -> YCbCr

W celu poprawnego wykonania zadania binaryzacji, która to jest konieczna do prawidłowego przeprowadzenia wyliczenia środka ciężkości należy odnaleźć odpowiednie progi wartości. Dla obrazów przedstawionych w standardowej przestrzeni barw RGB (*ang. Red-Green-Blue*) jest to zadanie skomplikowane ze względu na konieczność odpowiedniego doboru wartości progowej dla każdej ze składowych z osobna. Pomocna w takich przypadkach wydaje się być zamiana przestrzeni barw, w której za kolor odpowiedzialne są 3 składowe na taką, w której jest on zależny od 2, w tym wypadku YCbCr. Typy danych występujące w tej przestrzeni są następujące:

- Y - składowa *luminancji*, czyli wartość informująca o poziomie natężenia światła,
- Cb - składowa różnicowa *chrominancji* Y-B, czyli różnica pomiędzy *luminancją* a składową reprezentującą barwę niebieską,
- Cr - składowa różnicowa *chrominancji* Y-R, czyli różnica pomiędzy *luminancją* a składową reprezentującą barwę czerwoną.

Zamiana tych przestrzeni barw odbywa się przy pomocy poniższych wzorów (za [1]):

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.16836 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \quad (1)$$

Jak można łatwo wywnioskować z postaci wzoru przekształcającego, do jej realizacji wymagane jest 18 operacji: 9 mnożeń oraz 8 dodawań.

Implementacja sprzętowa modułu

Interfejs modułu ukazano na *Rysunku 3.3*.

```
module rgb2ycbcr
(
    input rst,
    input clk,
    input clock_enable,
    input de_in,
    input v_sync_in,
    input h_sync_in,
    input [23:0] pixel_in,
    output [23:0] YCbCr,
    output de_out,
    output v_sync_out,
    output h_sync_out
)
```

Rysunek 3.3 Interfejs modułu zamiany przestrzeni barw

Jest to standardowy interfejs modułu wizyjnego, taki jak ukazano w Rozdziale II, uzupełniony dodatkowo o sygnał *clock_enable* służący do kontroli działania modułu, jak również sygnał *rst* dodający możliwość zresetowania stanu całego modułu stanem wysokim.

Wartości będące stałymi występującymi w (1) zaimplementowano jako 18-bitowe zmienne typu *wire signed*, przeznaczając 1 bit na znak oraz 17 na część ułamkową – z racji na fakt, iż wszystkie te wartości mieszczą się w przedziale (-1,1) nie przeznaczono żadnego bitu na część całkowitą.

Przykładowo: *wire signed [17:0] a1=18'b001001100100010111* implementuje wartość 0.298828125000000.

Z racji na fakt, iż w przypadku mnożenia czynniki powinny mieć równą liczbę bitów, wartości R,G oraz B rozszerzono poprzez konkatencję, czyli złączenie dwóch wektorów, tak aby również posiadały 18 bitów, przy czym wartości te były typu *unsigned*, czyli bez znaku, przy czym najpierw należało te wartości uzyskać z wektora pikseli wejściowych poprzez podział, np. *wire signed R={10'b0,pixel_in[23:16]}.*

Operacje mnożenia przeprowadzono przy użyciu 9 instancji modułu Xilinx *Multiplier*, przypisując otrzymane z nich wyniki do pomocniczych zmiennych *wire signed* o szerokości 35 bitów.

Następnie, dla każdej ze składowych wykonano 2 operacje dodawania przy użyciu modułów Xilinx *Adder*: wynik pierwszego mnożenia dodano do drugiego, natomiast wynik trzeciego mnożenia do wartości znajdującej w odpowiednim miejscu macierzy występującej po prawej stronie wzoru przekształcającego. Ponieważ interesujący nas wynik powinien być liczbą całkowitą, wobec czego wartości wchodzące do sumatorów zostały okrojone do 9 bitów- 1 bit na znak oraz 8 na część całkowitą z przedziału (0-255). Wyjścia tych modułów również posiadały 9 bitów, z racji na brak możliwości otrzymania w wyniku ich działania liczb o wartości co do wartości bezwzględnej większej aniżeli 255. Dla składowej Y wartość występująca w prawej macierzy wynosi 0, zdecydowano jednak na wdrożenie dla niej również drugiego dodawania, ze względu na poprawną synchronizację sygnałów w czasie.

Po wykonaniu wyżej wymienionych operacji dla każdej ze składowych barw otrzymano po dwie wartości, które następnie zsumowano używając kolejnego sumatora, otrzymując ostatecznie składowe Y,Cb oraz Cr, które to odpowiednio przypisano do sygnału wyjściowego *pixel_out* poprzez przypisanie ciągle asynchroniczne *assign*.

Bardzo ważnym elementem, nie tylko dla modułu zmiany barw, ale również dla każdego innego modułu realizującego operacje na pikselach, jest odpowiednie opóźnienie wejściowych sygnałów kontrolnych przed przepisaniem ich na wyjście. Wszystkie operacje realizowane na pikselach mają swoją latencję, wobec czego konieczne jest obliczenie latencji całej operacji na nich by wiedzieć o ile taktów zegara należy opóźnić te sygnały. Moduły utworzone przez firmę Xilinx posiadają opcję automatycznego doboru optymalnej wartości latencji wobec czego w celu obliczenia latencji całego modułu wystarczy zsumować poszczególne latencje na drodze od wejścia do wyjścia pikseli. W przypadku modułu zmiany barw wynosiła ona 9 (5 dla mnożenia oraz po 2 dla dodawania).

Samą operację opóźnienia o określoną liczbę taktów zegara przeprowadzono przy użyciu specjalnie do tego celu skonstruowanego modułu opóźniającego. Na *Rysunku 3.4* przedstawiono jego interfejs.

```
module delay1
#
(
    parameter N=8,
    parameter DELAY=7
)
(
    input [N-1:0] idata,
    input clk,
    output [N-1:0] odata,
    input ce,
    input rst
);
```

Rysunek 3.4 Interfejs modułu opóźniającego

Parametr N odpowiada szerokości danych wejściowych, natomiast $DELAY$ ilości taktów zegara o które dane wejściowe mają być opóźnione. Moduł ten przy użyciu funkcji języka Verilog *generate* implementuje $DELAY$ razy moduł opóźniający wartości wejściowe o szerokości N o jeden takt zegara. Zabezpieczony jest on również przy pomocy sygnału *reset* oraz *clock enable*.

3.3 Binaryzacja

Binaryzacją, inaczej progowaniem(*ang. thresholding*), nazywamy proces uzyskiwania z obrazu kolorowego bądź też w odcieniach szarości obrazu binarnego, czyli czarno-białego. W niniejszej pracy jest ona wykorzystywana w celu oddzielenia tła od właściwego obiektu w celu dalszego przetwarzania. Dla każdego z pikseli wchodzących do modułu dokonano porównania jego wartości C_b oraz C_r z przyjętymi wartościami i przypisano mu wartość wyjściową zgodnie z formułą:

$$Pixel = \begin{cases} 0, & C_b \in (T_a, T_b) \wedge C_r \in (T_c, T_d) \\ 255, & \text{w pozostałych przypadkach} \end{cases}$$

Gdzie T_a, T_b, T_c oraz T_d oznaczają odpowiednio progi dolne i górne dla składowych C_b oraz C_r .

Metodą inżynierską dowiedziono, iż dla warunków oświetleniowych obecnych w sali wartości te powinny wynosić odpowiednio:

- $T_a = T_c = 100$,
- $T_b = T_d = 160$.

Implementacja sprzętowa modułu

Implementacja modułu jest niezwykle prosta i opiera się na wykorzystaniu przypisania ciągłego asynchronicznego w następujący sposób:

```
assign bin = rst ? 0 : (clock_enable ? ((Cb > Ta && Cb < Tb && Cr > Tc && Cr < Td) ? 0:8'd255):prev_bin);
```

Progi dolne oraz górne zaimplementowano przy użyciu lokalnych parametrów *localparam*.

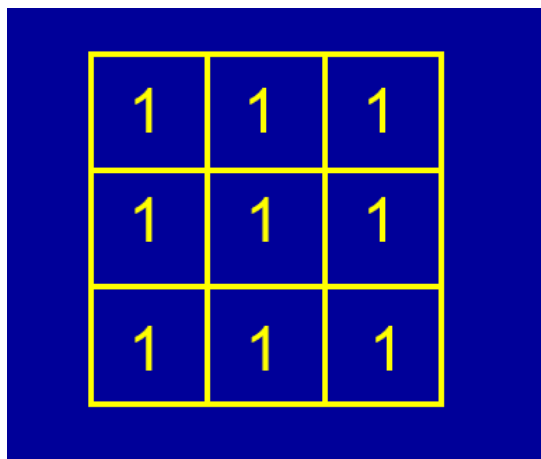
Wartość *prev_bin* jest obliczana jako przypisanie poprzedniej wartości sygnału *bin*.

Z racji iż nie są wykonywane żadne dodatkowe obliczenia, moduł nie wprowadza dodatkowej latencji, w związku z czym sygnały kontrolne wejściowe mogą zostać przepisane na wyjście bez użycia żadnych modułów opóźniających. Ponieważ wyjście tak jak wejście ma 24 bity szerokości, wartość *bin* przepisano na nie trzykrotnie poprzez konkatencję.

3.4 Filtracja

Można by przypuszczać, iż obraz po przejściu przez moduł binaryzujący powinien być jednolity, tj. zawierać jeden biały obiekt i czarne tło. Często jednak posiada czarne „dziury” będące efektem działania szumu. Obraz taki należy wobec tego poddać działaniu filtracji, tak aby ostatecznie można było przeprowadzić na nim operację wyliczenia środka ciężkości. Jedną ze stosowanych w tym celu metod jest wykorzystanie operacji morfologicznych.

Podstawowym pojęciem stosowanym w odniesieniu do nich jest *element strukturalny obrazu*, czyli pewnego rodzaju wycinek obrazu z wyróżnionym punktem centralnym. Przykładowy element strukturalny, dla operacji *erozji*, przedstawiono na Rysunku 3.5



Rysunek 3.5 Przykładowy element strukturalny [11]

Algorytm wykonania operacji morfologicznej na obrazie przedstawia się w sposób następujący:

- Element strukturalny przemieszczany jest po całym obrazie w taki sposób, aby analizowany piksel pokrywał się z punktem centralnym elementu strukturalnego.
- W każdym punkcie następuje porównanie czy konfiguracja pikseli otoczenia punktu jest zgodna ze wzorcem elementu strukturalnego.
- W przypadku wykrycia zgodności następuje jakaś operacja na tym pikselu, najczęściej jest to zmiana jego wartości.

Przykładowymi operacjami morfologicznymi jest *erozja* oraz *dylatacja*. Możemy je zdefiniować w sposób następujący (za [11]):

„Zakładamy, że obraz wyjściowy zawiera obszar X wyróżniający się pewną charakterystyczną cechą np.(jasnością). Figura przekształcona przez dylatację to zbiór punktów centralnych wszystkich elementów strukturalnych, których punkt mieści się we wnętrzu obszaru X . Miarą dylatacji jest wielkość elementu strukturalnego.

Figura X po wykonaniu operacji erozji (często określana krótko jako figura zerodowana) to *zbiór punktów centralnych wszystkich elementów strukturalnych, które w całości mieszczą się we wnętrzu obszaru X* ”.

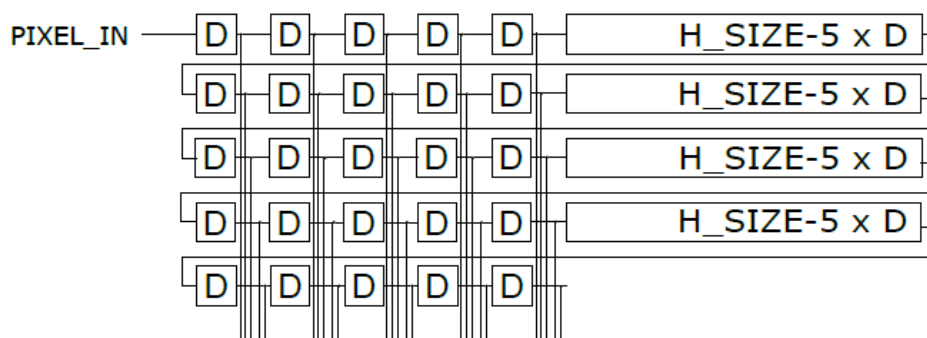
Operacje te, mimo, iż są korzystne z punktu widzenia filtracji obrazu binarnego, mają też pewną wadę, a mianowicie znacznie zmieniają pole przekształcanych obiektów – erozja poprzez zmniejszenie a dylatacja poprzez powiększenie. By wyeliminować zaistniały problem często dokonuje się tych operacji razem jedna po drugiej, a w zależności od kolejności otrzymując operację morfologicznego *otwarcia* bądź też *zamknięcia*.

W opisywanej pracy dyplomowej w celach filtracji obszaru binarnego zastosowano operację morfologicznego *zamknięcia*, a więc kombinację *dylatacji* oraz *erozji*, w tej właśnie kolejności.

Implementacja sprzętowa modułu

Realizowana operacja jest operacją kontekstową, czyli taką, która w przeciwieństwie do operacji punktowych, nadaje nową wartość rozpatrywanemu pikselowi nie tylko na podstawie jego samego, ale również pikselów w jego otoczeniu. Wiąże się z tym pewien problem, a mianowicie nie ma możliwości poprawnego obliczenia nowych wartości dla pikseli brzegowych obrazu. Możliwości obejścia tego problemu co najmniej kilka, w rozpatrywanej pracy zastosowano najprostszą z możliwych, a mianowicie przypisanie pikselom brzegowym na wyjściu wartości 0.

Kolejnym ważnym problemem z punktu widzenia sprzętowej implementacji jest fakt, iż przy przyłożeniu punktu centralnego do rozpatrywanego piksela należy znać również wartości jego sąsiadów, w tym wypadku 24, gdyż zastosowano maskę 5×5 . Należy więc wstrzymać działanie całego układu na czas obliczanie nowej wartości piksela. W tym celu wykorzystano 25 przerzutników opóźniających pojedynczy piksel o jeden takt zegara oraz 5 długi linii opóźniających zrealizowanych w postaci bloków pamięci BRAM ([1]), które przechowują H_SIZE-5 pikseli w każdej, gdzie wartość parametru H_SIZE powinna odzwierciedlać szerokość obrazu. Schemat takiego działania przedstawiono na Rysunku 3.6



Rysunek 3.6 Schemat linii opóźniających [1]

Ze względu na podobieństwo dla obydwu zastosowanych operacji morfologicznych, omówiona zostanie tylko *dylatacja*.

W celu implementacji 25 przerzutników utworzono 25 czterobitowych rejestrów. Szerokość wynikała z faktu, iż dla każdego piksela, oprócz jego binarnej wartości, istnieją również 3 sygnały kontrolne.

Tak utworzono rejestry od D11_1 do D55_1, dla każdego piksela w każdym wierszu i kolumnie elementu strukturalnego. Pierwsze piksele każdego wiersza otrzymywały swoją wartość od wyjścia odpowiedniej długiej linii opóźniającej (oprócz pierwszego piksela pierwszego wiersza, który to wartość otrzymywał bezpośrednio z danych wejściowych modułu), a ostatnie piksele każdego wiersza przepisywały swoją wartość na wejście odpowiedniej kolejnej linii opóźniającej (oprócz ostatniego piksela ostatniego wiersza).

W celu sprawdzenia czy rozpatrywany piksel nie jest pikselem brzegowym dla tak dobranego kontekstu 5x5, dokonywano koniunkcji na wszystkich wartościach sygnałów kontrolnych *de* pikseli rozpatrywanych w danym momencie i przypisywano do osobnej zmiennej *context_valid*. W przypadku gdy ta wartość wynosiła 0, rozpatrywanemu pikselowi przypisywano 0 bez względu na wartość jaką otrzymał w obliczeniu dylatacji dla niego. Obliczanie wartości samego piksela odbywało się poprzez działanie logiczne alternatywy na wszystkich wartościach pikseli kontekstu poza nim samym. Tak przetworzony piksel był następnie transportowany do części modułu zajmującej się działaniem *erozji*. Jak wspomniano wyżej, jego implementacja była praktycznie identyczna jak *dylatacji*, jedyną różnicą było to, iż w tym wypadku wartość piksela obliczana była jako koniunkcja, a nie alternatywa.

3.5 Wyliczenie środka ciężkości

Mając już dany przefiltrowany obraz binarny, można przeliczyć właściwą część zadania, a więc wyznaczyć środek ciężkości. Dokonano tego w oparciu o wzory bazujące na momentach geometrycznych ukazane poniżej (za [1]):

$$m_{00} = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} P_{ij} \quad (2)$$

$$m_{10} = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} i * P_{ij} \quad (3)$$

$$m_{01} = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} j * P_{ij} \quad (4)$$

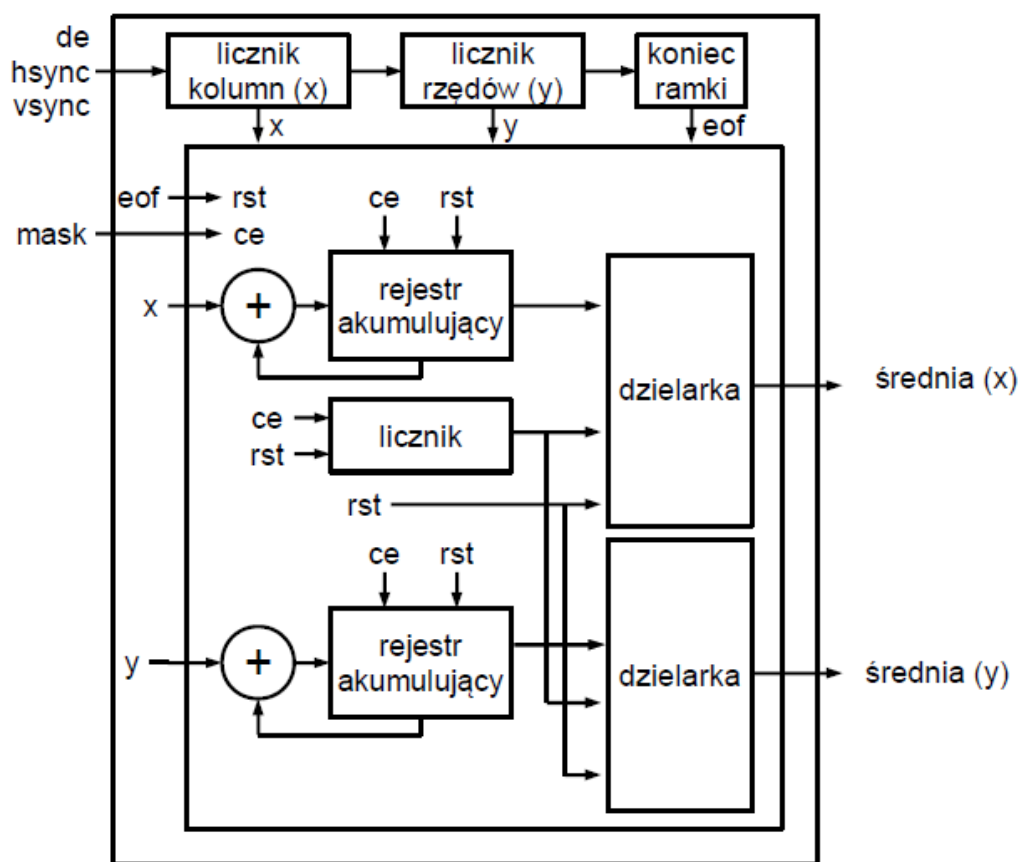
Gdzie N oraz M oznaczają wymiary obrazu, natomiast P_{ij} wartość binarną rozpatrywanego piksela. Znając te wartości, środek ciężkości rozpatrywanego obiektu możemy zdefiniować jako (za [1]):

$$x_{cen} = \frac{m_{10}}{m_{00}} \quad (5)$$

$$y_{cen} = \frac{m_{01}}{m_{00}} \quad (6)$$

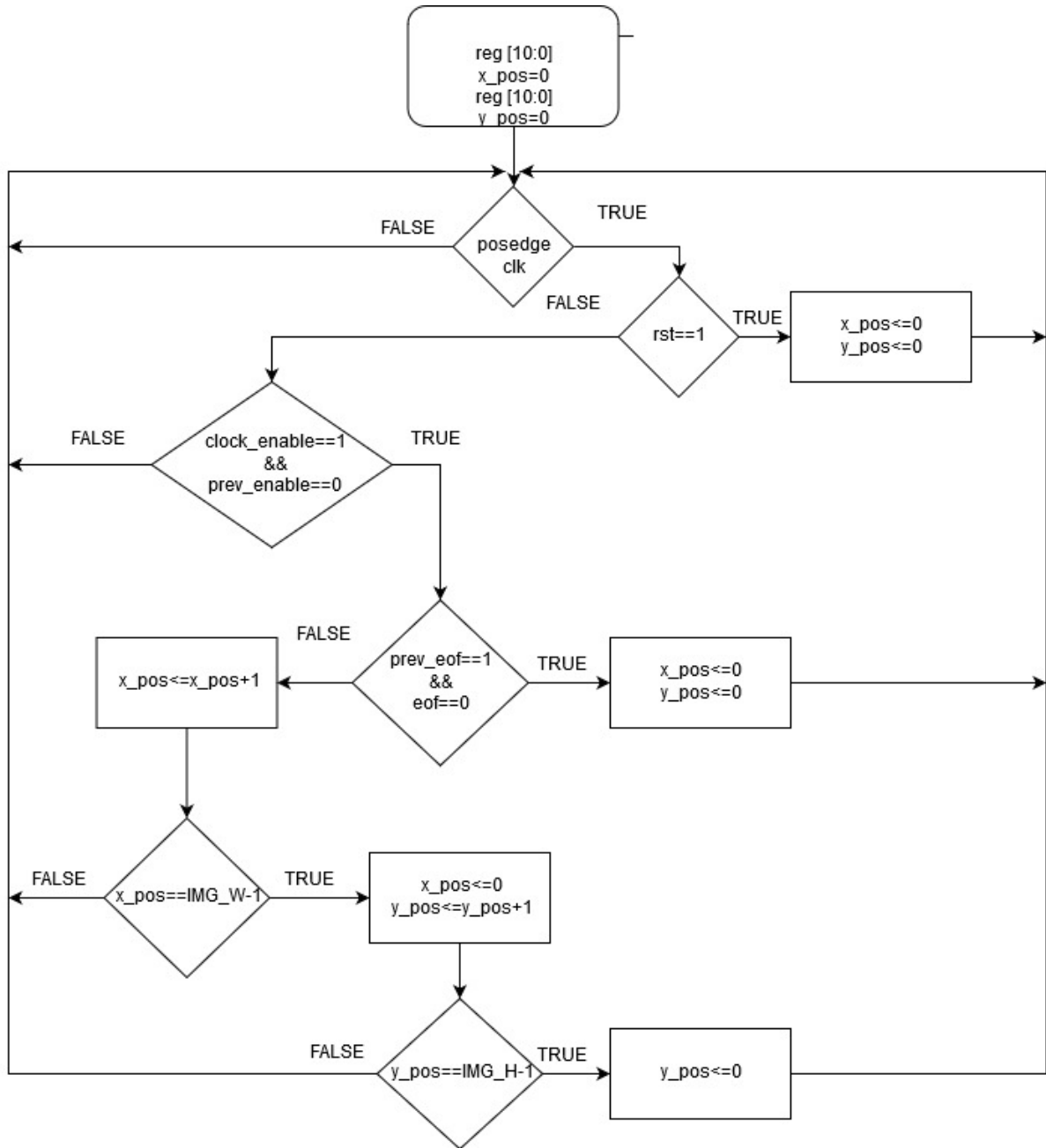
Implementacja sprzętowa modułu

Ogólny schemat modułu do wyznaczenia pożądanych wartości przedstawiono na *Rysunku 3.7*. Jednym z jego istotnych elementów są dwa rejestry akumulujące służące do przechowywania i dodawania do siebie kolejnych elementów w celu wyliczania wartości m_{01} oraz $m_{10} - (3)$ oraz (4). Zostały one zaimplementowane jako połączenie modułu Xilinx *Adder*, czyli zwykłego sumatora oraz modułu opóźniającego N bitów wejściowych o jeden takt zegara w celu poprawnej synchronizacji dodawania. Wyjście sumatora stanowi wejście do modułu opóźniającego, którego wyjście, oprócz wyprowadzenia na wyjście modułu, jest również sprzężeniem zwrotnym do jednego z wejść sumatora.



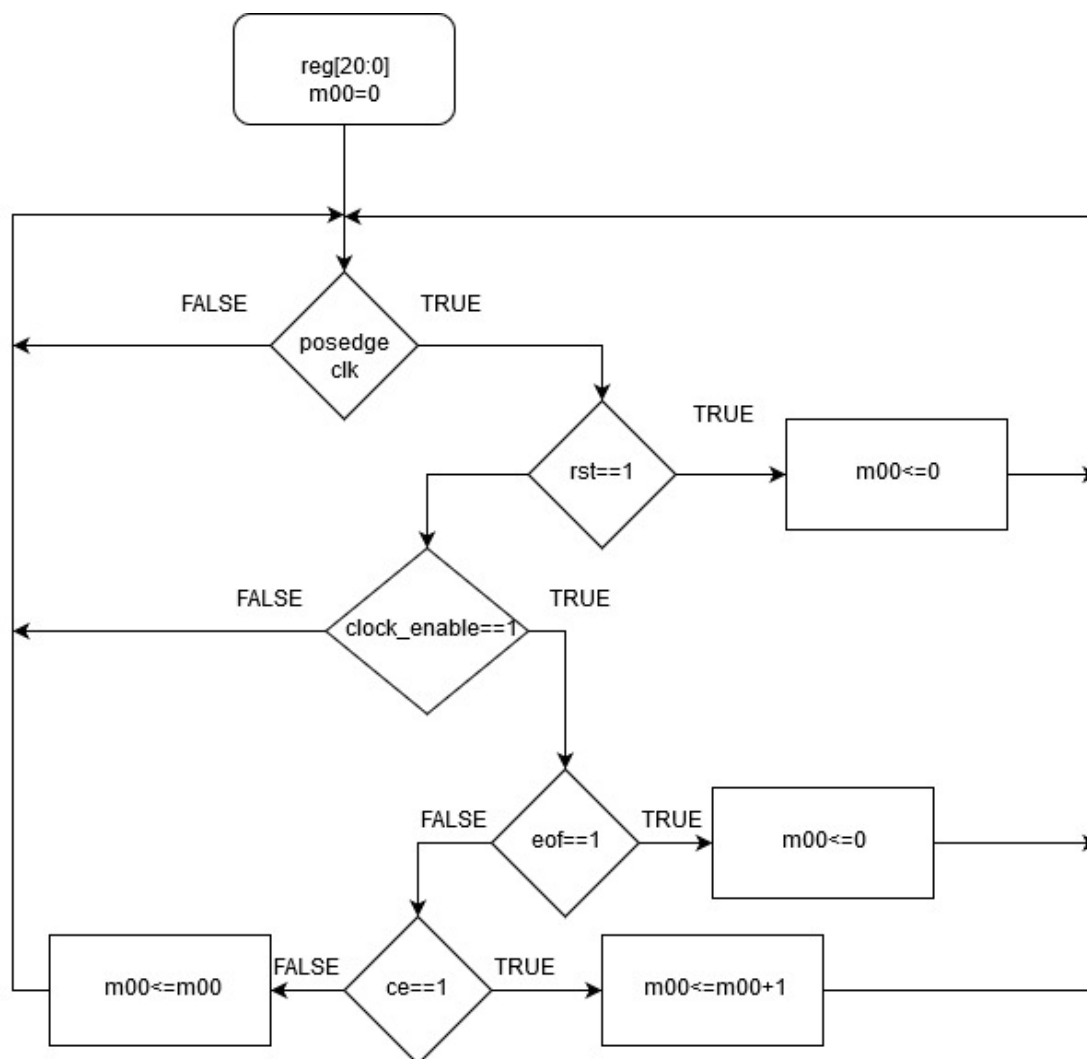
Rysunek 3.7 Schemat modułu wyliczającego środek ciężkości [1]

Na samym początku modułu zaimplementowano synchroniczny proces służący określaniu pozycji aktualnie rozpatrywanego piksela w całym module. Jego schemat blokowy ukazano na *Rysunku 3.8*. Wyznaczanie sygnału końca ramki *eof* odbywało się poprzez porównanie obecnej wartości sygnału synchronizacji pionowej *vsync* z jego poprzednią wartością, dzięki czemu można było wykryć zbocze narastające tego sygnału symbolizujące koniec nadawania obecnej ramki obrazu.



Rysunek 3.8 Schemat blokowy sekwencyjnego procesu wyznaczania pozycji

Jak wspomniano wyżej, wyznaczaniem wartości m_{01} oraz m_{10} zajmowały się akumulatory. Wartościami do sumowania były w nich odpowiednie pozycje x oraz y, natomiast sygnałem *clock enable* modułu był iloczyn logiczny wartości aktualnie rozpatrywanego piksela binarnego oraz wejściowego sygnału *clock_enable*. Wartość m_{00} – (2) wyliczano z kolei w procesie synchronicznym którego schemat blokowy przedstawiono na Rysunku 3.9.



Rysunek 3.9 Schemat blokowy procesu sekwencyjnego procesu wyznaczania m_{00}

Ostatnim, a zarazem najważniejszym krokiem obliczania środka ciężkości było zaimplementowanie dwóch dzieleni widocznych we wzorach na początku tego podrozdziału – (5) oraz (6). W celu ich wykonania zaimplementowano specjalną dzielarkę działającą na zasadzie maszyny stanów ([1]). Poza swoją oczywistą funkcją dzielenia, posiadała ona również wejście startujące oraz wyjście sygnalizujące zakończenie procesu dzielenia. Dzięki podłączeniu do tego wejścia sygnału końca ramki *eof* była pewność, iż dzielenie wykonywane jest na dokładnie tych wartościach na których powinno, natomiast dzięki sygnalizującemu wyjściu można było precyzyjnie określić moment w którym należało na wyjście całego modułu przepisać nowo obliczoną wartość środka ciężkości a kiedy „zatrzasnąć” poprzednią.

W ten sposób dokonano wyliczenia środka ciężkości obiektu w ramce. Wadą zaimplementowanego algorytmu jest fakt, obliczenia trwają niemalże całą ramkę obrazu, wobec czego nie ma zachowanej potokowości przetwarzania.

3.6 Moduł Rejestru 32-bitowego

W celu prawidłowej kontroli nad wartościami wejściowymi jak i koniecznymi sygnałami wyjściowymi modułu IP opisanego w powyższych podrozdziałach, utworzono dodatkowy moduł *Rej32b* współpracujący z modułem wyliczającym środek ciężkości.

Jego pierwszym zadaniem było przepisywanie z wewnętrznego 32 bitowego rejestru danych, uprzednio dostarczonych przez układ *PS*, wartości danego piksela na wejście modułu w zależności od wartości zmiennej *r_Enable*, której wartość obliczano w sposób następujący: Jeżeli układ *PS* wystawił sygnał *Start*, wtedy wartość ta zmieniała się na 1, zmiana na 0 następowała w dwóch możliwych przypadkach, a mianowicie gdy wystawiono sygnał *rst*, bądź też moduł odebrał obydwa pozytywne sygnały informujące o skończeniu procesu wyznaczania środka ciężkości dla danej ramki. W pozostałych przypadkach sygnał zachowywał swoją wartość z poprzedniego zbocza narastającego sygnału zegarowego.

Drugim ważnym zadaniem modułu było wystawianie długiego sygnału w przypadku gdy otrzymał on informację od obydwu dzielarek o skończonym procesie dzielenia. Ponieważ sygnały wystawiane przez niego trwały wyłącznie jeden takt zegara, wobec czego niemożliwym było aby taki sygnał był odczytany przez układ *PS*. Wobec tego, po otrzymaniu tych danych, moduł *Rej32b* wystawia na wyjście flagę *Done_xy*, która możliwa jest do „zgaszenia” poprzez zewnętrzny sygnał dostarczony przez *PS*, a mianowicie *put_done_down*. Pozwala to na odczyt przez układ wyliczonego środka ciężkości w dogodnym dla aplikacji momencie a następnie wygaszeniu go zewnętrznie.

3.7 Magistrala AXI

Wszystkie moduły omówione w poprzednich podrozdziałach następnie zaimplementowano w jednym module tworząc z nich system potokowy i robiąc z niego oddzielny IP Core. Taki moduł nie może jednakże bezpośrednio komunikować się z układem *PS* płytki Ultrazed, konieczne do tego jest zaimplementowanie pośrednika w postaci magistrali AXI4, w tym wypadku jej wersji LITE. Jest to jedna z trzech typów rodziny magistral AXI, zaraz obok AXI4 oraz AXI4-Stream. Główną różnicą pomiędzy wersją LITE a pełną wersją jest fakt, iż wersja LITE zezwala na szerokość interfejsu danych wyłącznie 32 bitowego, podczas gdy pełna wersja do 256. Jest to magistrala typu *Single Master -Single Slave*, jednakże jej implementacja w Xilinx IP pozwala na podłączenie więcej niż jednego *slave`a* do *mastera*. Zarówno AXI4 jak i jej wersja LITE składają się z 5 oddzielnych kanałów:

- *Read Address Channel*,
- *Write Address Channel*,
- *Read Data Channel*,
- *Write Data Channel*,
- *Write Response Channel*.

Dane mogą być przekazywane w obydwu kierunkach jednocześnie. Transfer odbywa się przy pomocy wzajemnej wymiany zapytań i potwierdzeń, tzw. *Handshake*.

Implementacja sprzętowa modułu

Implementacja sprzętowa w Vivado odbyła się w prosty sposób. Wybierając *Tools->Create and package New IP* wybrano *Create AXI4 Peripheral*. W nowo otwartym oknie dialogowym wybrano odpowiednie parametry tworzonego interfejsu i zatwierdzono zmiany, co spowodowało otwarcie nowego projektu Vivado z zaimplementowanymi dwoma plikami odpowiadającymi za realizację działania magistrali AXI. Posiadała ona 64 32-bitowe rejestry przeznaczone do przesyłu danych pomiędzy modulem FPGA a *masterem* AXI o domyślnych nazwach *slv_reg0*, *slv_reg1*, *slv_reg2* itd. Sygnałom do odczytu nadano osobne nazwy poprzez nadpisanie domyślnych przez nowo utworzone zmienne typu *wire*, natomiast te, które miały służyć do zapisu zostawiono bez zmian. Na *Rysunku 3.10* przedstawiono implementację końcowych modułów w instancji magistrali AXI. Zawierają one dodatkowe sygnały, które były pomocne przy testowaniu zachowania części *PL*. Na końcu z instancji AXI utworzono moduł *IP Core*.

```

Failure_0 Weight_Center
(
    .clock_enable(slv_reg30[0]),
    .rst(slv_reg26[0]),
    .clk(S_AXI_ACLK),
    .de_in(pixels[26]),
    .h_sync_in(pixels[24]),
    .v_sync_in(pixels[25]),
    .pixel_in(pixels[23:0]),
    .x(x_pos),
    .y(y_pos),
    .de_out(sync_out[2]),
    .h_sync_out(sync_out[1]),
    .v_sync_out(sync_out[0]),
    .pixel_out(pixel_out),
    .Done_x(X_done),
    .Done_y(Y_done),
    .Eof(Eof_check),
    .m00_out(m00),
    .m01_out(m01),
    .m10_out(m10),
    .pixel_in_centroid(pixel_ce),
    .de_muxes(de_muxes),
    .rgb_mux1(rgb1),
    .rgb_mux2(rgb2),
    .rgb_mux3(rgb3),
    .rgb_mux4(rgb4),
    .rgb_mux5(rgb5),
    .position_x(position_x),
    .position_y(position_y),
    .abstraction(abstraction)
);

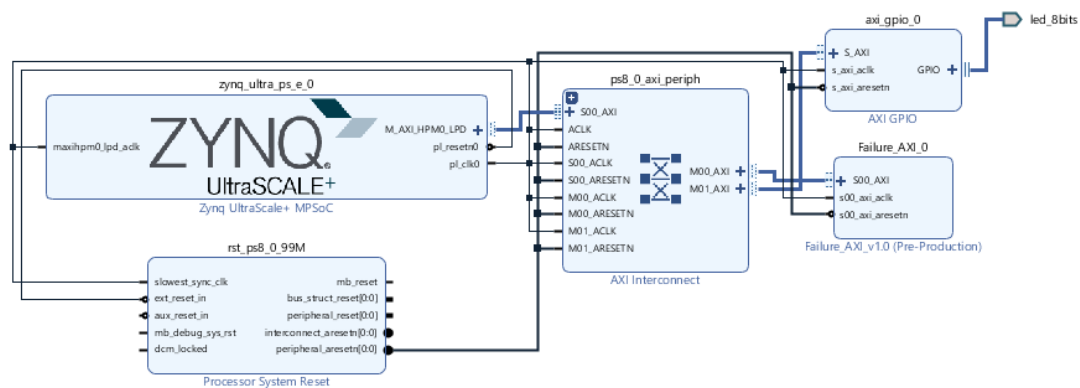
Rej32_0 Reg32b
(
    .rst(slv_reg26[0]),
    .clock_enable(slv_reg30[0]),
    .Clk(S_AXI_ACLK),
    .Done_x(X_done),
    .Done_y(Y_done),
    .Start(slv_reg1[0]),
    .Data_in(slv_reg0),
    .Enable(enable),
    .Data_out(pixels),
    .Done_xy(Done_xy),
    .AXI_check(Axi_check),
    .put_done_down(slv_reg25[0])
);

```

Rysunek 3.10 Implementacja końcowych modułów w instancji bloku implementującego magistralę AXI4-LITE

3.8 Dodanie układu Zynq oraz wygenerowanie pliku bitfile

Ostatnim krokiem w generowaniu części sprzętowej jest podłączenie utworzonego modułu do układu mikroprocesorów ARM, czyli do części *PS*. Oprogramowanie Vivado przychodzi tutaj z dużą pomocą użytkownikowi udostępniając opcje automatyzacji konfiguracji oraz niezbędnych podłączeń implementowanych modułów. Pracę rozpoczęto od utworzenia nowego projektu w którym stworzono *Block Design*. Poprzez wbudowaną bibliotekę dodano do niego *Zynq UltraScale+ MPSoC* i automatycznie dokonano jego konfiguracji. Warto wspomnieć, że użytkownik dostaje bardzo dużą swobodę w dostosowywaniu pod siebie konfiguracji *PS* co pozwala na chociażby zaoszczędzenie zasobów poprzez odłączenie peryferiów nieprzydatnych w danym projekcie. Poprzez zakładkę *Board* dodano do projektu chociażby obsługę 8 diod LED w ramach testów modułu. Po dodaniu układu Zynq do designu zaimportowano utworzony w ramach pracy moduł współpracujący z magistralą AXI i dokonano podłączenia tychże układów. Na *Rysunku 3.11* przedstawiono kompletny moduł realizujący część sprzętową niniejszej pracy inżynierskiej.



Rysunek 3.11 Część sprzętowa pracy dyplomowej

Widoczny w centralnej części układu blok *AXI Interconnect* symbolizuje blok pośredniczący w wymianie danych pomiędzy wszystkimi interfejsami AXI a układem *PS*. Zaobserwować można również blok odpowiedzialny za resetowanie całości. Z kolei w górnym prawym rogu widnieje układ AXI odpowiedzialny za komunikację z wyżej wspomnianymi diodami LED.

By zakończyć cały proces tworzenia układu wygenerowano plik *bitstream*, który to następnie posłużył do utworzenia odpowiedniej wersji części software'owej. Jednakże etap ten musi być poprzedzony dwoma innymi, a mianowicie *syntezą* oraz *implementacją*. Co ważne, żadnego tych etapów nie można przeprowadzić bezpośrednio na utworzonym projekcie, konieczne jest „opakowanie go” poprzez utworzenie *wrapper`a*.

Każdy z nich pełni swoją określoną funkcję:

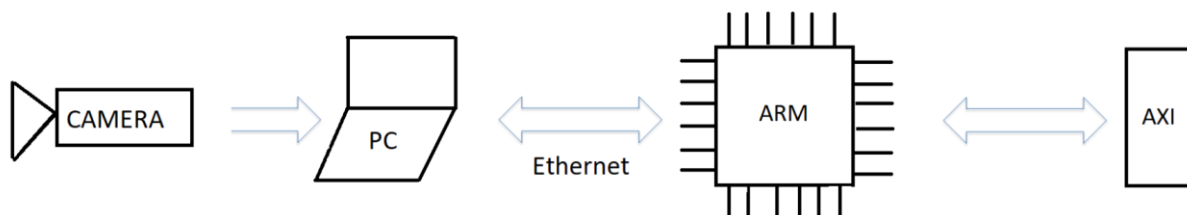
- Synteza - Przekształcenie poleceń języka HDL do architektury *netlisty* czyli modułów dostępnych w architekturze na którą tworzony jest projekt oraz połączeń pomiędzy nimi.
- Implementacja - Połączenie *netlisty* z ograniczeniami użytkownika (dodanymi poprzez plik *User Constraints*) oraz zmapowanie ich na konkretne elementy występujące w układzie rekonfigurowalnym i ich poszczególne miejsce na nim.

- Generowanie bitstream`u - Utworzenie pliku *bit*, który można następnie wgrać do układu FPGA otrzymując docelową architekturę.

Rozdział 4 Realizacja części programowej pracy dyplomowej

4.1 Wstęp

Poprzedni rozdział pracy dyplomowej dotyczył części sprzętowej. W tym rozdziale opisana zostanie druga część, a mianowicie część programowa. Pierwszy podrozdział dotyczy sposobu generacji systemu operacyjnego – *Petalinuxa*. Następne podrozdziały poświęcone są z kolei idei połączenia poprzez protokół TCP/IP, mapowaniu pamięci oraz aplikacjom realizującym je i rozszerzającym o wykonywanie pozostałych zadań związanych z pracą dyplomową. Rysunek 4.1 pokazuje jego schemat.



Rysunek 4.1 Schemat połączeń układu

Wszystkie aplikacje docelowo zaimplementowane na układzie *PS* napisano przy użyciu środowiska programistycznego *Xilinx SDK 2018.3*. Najważniejszym elementem każdego projektu jest w nim plik *hdf* opisujący wygenerowany układ w systemie rekonfigurowalnym. Otrzymuje się go poprzez opcję *export hardware* dostępną w używanym w poprzednim rozdziale oprogramowaniu *Vivado* po wygenerowaniu pliku *bitstream*. Tworząc nowy projekt w SDK programista ma do wyboru jedną z dwóch platform docelowych: *linux* oraz *bare metal*. Aplikacje tego drugiego typu wymagają dodatkowo wygenerowania osobnych plików *BSP* (ang. *Bare Support Package*) zawierających sterowniki dzięki którym możliwy będzie dostęp do peryferiów układu. Aplikacje na platformę *linux* nie wymagają ich, gdyż sam system operacyjny na który są one przeznaczone posiada już takowe wbudowane. Wszystkie napisane aplikacje w ramach pracy dyplomowej były tego właśnie typu. Jednakże, aby mogły one działać poprawnie koniecznym było wygenerowanie odpowiedniej wersji systemu operacyjnego, na którym mogłyby one działać, czyli mówiąc dokładniej, *Petalinuxa*.

4.2 Petalinux

Petalinux to system operacyjny typu Linux przeznaczony na systemy SoC zawierające układy FPGA firmy Xilinx. Do jego wygenerowania potrzebne są specjalne narzędzia programistyczne dostarczane przez producenta oprogramowania – *petalinux tools* oraz system operacyjny z zainstalowanym systemem operacyjnym typu Linux. Do celów pracy dyplomowej wykorzystano wirtualną maszynę z zainstalowanym systemem *Ubuntu* w wersji 16.04. W celu instalacji wszystkich niezbędnych aplikacji oraz samego systemu wykorzystano instrukcje zawarte w [6].

Pierwszą komendą, którą rozpoczęto pracę w celu wygenerowania własnej wersji systemu, było komenda linkująca odpowiedni plik ustawień:

```
$ source <path-to-installed-PetaLinux>/settings.sh
```

Następnie utworzono nowy projekt typu *PetaLinux*:

```
$ petalinux-create --type project --template <PLATFORM> --name <PROJECT_NAME>
```

Utworzony project należało następnie dopasować do potrzeb projektu poprzez zaimportowanie pliku *hdf*:

```
$ petalinux-config --get-hw-description=<path-to-directory containing hardware description-file>
```

Powyższa komenda wyzwała okno konfiguracyjne, pozwalające na wiele modyfikacji w celu dostosowania odpowiednich peryferiów, w przypadku opisywanego projektu pozostawiono domyślne opcje, usuwając jedynie domyślne logowanie do systemu poprzez *root*.

W celu zbudowania całego systemu wydano komendę:

```
$ petalinux-build
```

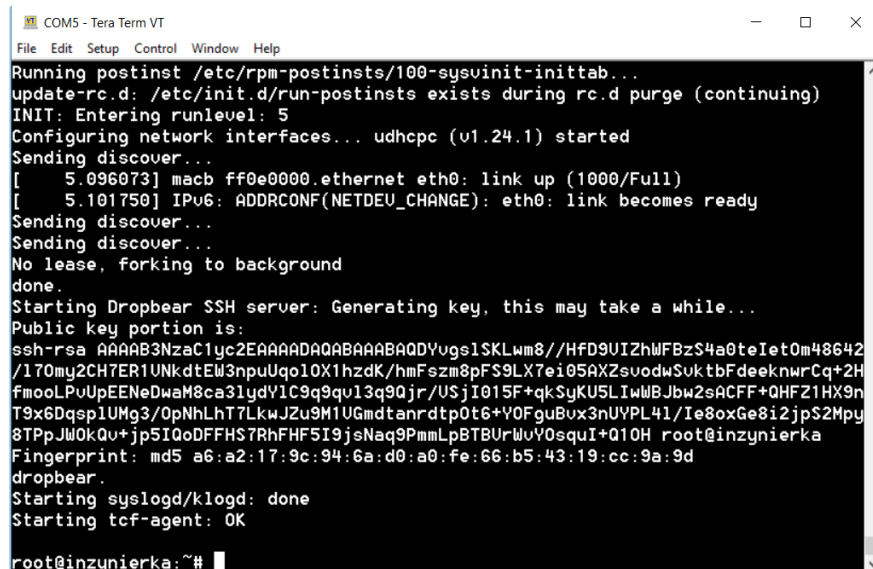
Po zbudowaniu całego systemu, wynikiem pracy był szereg plików, z których następnie należało wygenerować końcowy plik *BOOT.bin* poprzez komendę:

```
$ petalinux-package --boot --format BIN --fsbl images/linux/zynqmp_fsbl.elf --u-boot  
images/linux/u-boot.elf --pmufw images/linux/pmufw.elf --fpga images/linux/*.bit  
--force
```

Tak otrzymany plik załadowano na kartę SD wraz z *image.ub*, który to stanowi obraz samego systemu. Po włożeniu karty do układu, uruchomieniu układu jak i ustanowieniu połączenia poprzez UART ukazywał się widok widoczny na *Rysunku 4.2*

Połączenie zrealizowano poprzez oprogramowanie *Tera Term* z poniższymi parametrami:

- *Baud rate* – 115200
- *Data* – 8 bit
- *Parit* – none
- *Stop bits* – 1 bit
- *Flow control* – none.



```

COM5 - Tera Term VT
File Edit Setup Control Window Help
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
INIT: Entering runlevel: 5
Configuring network interfaces... udhcpc (v1.24.1) started
Sending discover...
[ 5.096073] macb ff0e0000.ethernet eth0: link up (1000/Full)
[ 5.101750] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
Sending discover...
Sending discover...
No lease, forking to background
done.
Starting Dropbear SSH server: Generating key, this may take a while...
Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDAQDYugs1SKLwm8/HfD9UIZhWFBzS4a0teIetOm48642
/170my2CH7ER1UNKdtEw3npuUqo10X1hzdK/hmFszm8pFS9LX7ei05AXZsuodwSuktbfdeeknrCq+2H
fmoolPvUpEEENeDwaM8ca31ydY1C9q9qu13q9Qjr/USjI015F+qkSyKUSLIwWBJbw2sACFF+QHFZ1HX9n
T9x6Dqsp1UMg3/OpNhLhT7LkwJZu9M1UGmdtanrdtp0t6+Y0FguBux3nUYPL41/Ie8oxGe8i2jpS2Mpy
8TPpJW0kQu+jp5IQoDFFHS7RhFHF5I9jsNaq9PmmLpBTBvUwV0sqUI+Q10H root@inzynierka
Fingerprint: md5 a6:a2:17:9c:94:6a:d0:a0:fe:66:b5:43:19:cc:9a:9d
dropbear.
Starting syslogd/klogd: done
Starting tcf-agent: OK
root@inzynierka:~#

```

Rysunek 4.2 Terminal wygenerowanego PetaLinuxa.

4.3 Połączenie TCP/IP

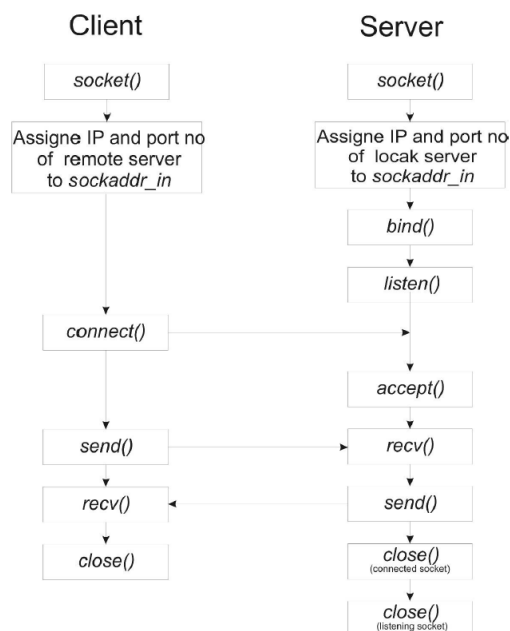
Do połączenia pomiędzy komputerem nadzorującym a układem *PS* wykorzystano protokół *TCP/IP* (ang. *Transmission Control Protocol/Internet Protocol*). Opiera się on na koncepcji gniazd sieciowych, tzw. *socketów*. Dla systemów Windows nazywają się one *Winsock*, natomiast dla systemów unix – *BSD Sockets*. Każde gniazdo zawiera informacje o adresie IP, swoim numerze portu oraz użytym protokole warstwy transportowej.

Połączenie poprzez *TCP/IP* wymaga dwóch aplikacji – *Serwera* oraz *Klienta*. W przeciwieństwie do drugiego równie popularnego protokołu – *TCP/UDP*, usługa ta tworzy wirtualne połączenie pomiędzy dwoma *socketami* oraz gwarantuje niezawodność transmisji jak również, co istotne z punktu widzenia realizowanego projektu, zachowanie kolejności przesyłanych danych.

Każde połączenie pomiędzy *Serwerem* a *Klientem* definiowane jest poprzez *kanal komunikacyjny*. Zawiera on informacje o protokole w jakim przesyłane są wzdłuż niego dane oraz adresy IP i numer portu obydwu stron. Przykładowy cykl pracy połączenia przy użyciu tego protokołu przedstawia *Rysunek 4.3*.

Klient zawsze jako pierwszy rozpoczyna komunikację, podczas gdy Serwer nasłuchuje na konkretnym porcie na nadchodzące wiadomości od Klienta. W tym celu Klient potrzebuje znać zarówno adres IP Serwera jak i numer portu na którym prowadzi on nasłuch. Następnie następuje wymiana żądanych informacji z potwierdzeniem po którym następuje zamknięcie kanału komunikacyjnego. Każda ze stron w toku komunikacji musi wykonać każde z poniższych zadań:

- Utworzenie gniazda
- Określenie jego parametrów
- Utworzenie kanału komunikacyjnego poprzez połączenie z drugim gniazdem
- Transmisja danych
- Zamknięcie gniazda co prowadzi do zamknięcia kanału komunikacyjnego.



Rysunek 4.3. Cykl pracy połączenia poprzez protokół TCP/IP [5]

Z punktu widzenia realizowanego projektu, bardzo ważnym aspektem jest fakt, iż zarówno biblioteki dla systemu Windows jak i PetaLinux posiadają wbudowane biblioteki implementujące szczegóły dotyczące tego jak i innych popularnych protokołów przesyłów danych, co zdecydowanie ułatwia pracę z nimi – tworząc aplikację typu *bare metal* konieczne byłoby między innymi samodzielne utworzenie stosu TCP/IP co stanowi bardzo czasochłonny proces.

4.4 Mapowanie obszarów pamięci

W celu dostępu do obszaru pamięci magistrali AXI odpowiedzialnej za komunikację z częścią *PL* układu konieczne jest skorzystanie z *mapowania obszaru pamięci*. Użytkownik nie posiada bowiem bezpośredniego dostępu do fizycznych adresów jądra systemu ze względów bezpieczeństwa. Konieczne jest więc zrzutowanie adresów fizycznych do pamięci wirtualnej by następnie poprzez działanie na niej móc działać na obszarze chronionym przez jądro systemu. Najważniejszą funkcją wykorzystywaną do tego celu jest *mmap()* dostarczana poprzez bibliotekę *<sys/mman.h>*. Przyjmuje ona szereg argumentów w postaci:

- *Addr* - Adres nowo zmapowanej pamięci, domyślnie *NULL*.
- *Length* - Rozmiar mapowanej pamięci w bajtach.
- *Prot* – Rodzaj ochrony mapowanej pamięci, w celu możliwości zarówno zapisu jak i odczytu należy ustawić *PROT_READ/PROT_WRITE*.
- *Flags* – Widoczność wprowadzonych zmian dla pozostałych procesów mapujących ten sam region pamięci.
- *Fd* – Uchwyt do otwieranego miejsca w pamięci – „*dev/mem*”.
- *Offset* – Adres mapowanej pamięci w otwieranym miejscu – musi stanowić wielokrotność rozmiaru strony pamięci, w tym wypadku 4096 bajtów.

Po skończeniu korzystania ze zmapowanego obszaru pamięci, należy go rozłączyć poprzez komendę *munmap()*.

Powyżej zaprezentowany sposób dostępu do pamięci fizycznej jest sposobem najprostszym, jednakże nieoptymalnym z punktu widzenia bezpieczeństwa systemu. Pozwala on bowiem na dostęp do każdego miejsca w pamięci, również tego zawierającego dane konieczne do poprawnego działania systemu, co w skrajnym przypadku może zagrozić jego poprawnemu działaniu.

4.7 Przetwarzanie plików BMP

W celu testowania poprawności działania układu przetwarzane były obrazy w formacie BMP o różnych rozmiarach uzyskane z kamery Basler AG. Utworzona została więc funkcja w języku C przyjmująca na wejście takowy obraz i zwracająca na wyjście wskaźnik do dwuwymiarowej tablicy alokowanej dynamicznie zawierającej poszczególne piksele obrazu – *ReadBMP2()*. Ze wczytywanego pliku BMP usuwano 54 bajtowy nagłówek zawierający informacje o tym obrazie, w tym m.in. o jego rozmiarach a następnie na tej podstawie alokowano dynamicznie dwuwymiarową tablicę o rozmiarze $[wysokość \cdot szerokość] \cdot [4]$. Wartość 4 wynikała z faktu, iż sztucznie do każdego piksela dodawano 3 sygnały kontrolne w celu poprawnego działania modułu FPGA i zachowania możliwości przetwarzania przez niego „surowego” ciągu danych z kamery innego typu. Ważnym punktem funkcji przetwarzającej był fakt, iż format BMP przechowuje informacje o poszczególnych pikselach w pasach, jednakże od dołu do góry, co implikowało konieczność odpowiedniego obrócenia otrzymanych wyników. Jeżeli dany piksel był ostatnim w rzędzie, otrzymał on dodatkowo flagę *hsync*, jeżeli ostatnim w ramce to flagę *vsync*. Równocześnie każdy otrzymywał flagę *de*.

Ciekawą kwestią dotyczącą postaci formatu BMP jest fakt, iż jeżeli szerokość przechowywanego obrazu nie jest liczbą podzielną przez 4 to ilość bajtów przypadających na daną linię pikseli zostaje automatycznie uzupełniona do tej podzielności. Zjawisko to zwane jest *padding* i wynika ono z historycznej budowy dawnych procesorów w których znacznie ułatwiało to wczytywanie plików zapisywanych w ten sposób.

Co więcej, format BMP przechowuje informacje jako *BGR*, a nie *RGB*, co implikowało konieczność odpowiedniego odwrócenia danych w każdym pikselu.

4.5 API kamery

Kamerą zastosowaną w pracy była kamera Basler AG. Jest to wysoce wyspecjalizowana kamera przemysłowa posiadająca możliwość obsługi przy pomocy oprogramowania *Pylon viewer*. Ściąganie potrzebnych danych z niej realizowane być może przy pomocy komend języka C opisanych w [13]. Dzięki możliwości ustalenia zarówno formatu wyjściowego jak i jego rozmiaru, zdjęciu uzyskiwane z niej były w formacie BMP o rozdzielczości 128 x 64 pikseli.

W celu uzyskania klatki obrazu z kamery konieczne jest wykonanie następujących kroków:

- Inicjalizacja – *PylonInitialize()*,

- Wyliczenie podpiętych urządzeń – *PylonEnumerateDevices()*
- Otwarcie dostępu do urządzenia – *PylonDeviceOpen()*,
- Utworzenie połączenia – *PylonDeviceGetStreamGrabber()*,
- Parametryzacja ilości i rozmiaru buforów do przechowania danych – *PylonStreamGrabberSetMaxNumBuffer()/PylonStreamGrabberSetaxBufferSize()*,
- Alokacja zasobów zgodnie z podanymi parametrami – *PylonStreamGrabberPrepareGrab()*,
- Rejestracja buforów i podpięcie ich do kolejki łączy – *PylonStreamGrabberRegisterBuffer()/pylonStreamGrabberQueueBuffer()*
- Uruchomienie akwizycji danych – *PylonDeviceExecuteCommandFeature*,
- Urochomienie kamery – *PylonGigEIssueActionCommand()*,
- Odczekanie określonego czasu na zapełnienie bufora – *PylonWaitObjectsAny()*,
- Odzyskanie zdjęcia – *PylonStreamGrabberRetrieveResult()*
- Wyświetlenie zdjęcia w oknie – *PylonImageWindowDisplayImageGrabResult()*,
- Zamknięcie całości – *PylonTerminate()*.

4.6 Aplikacja klienta

Aplikacja klienta, realizowana na komputerze klasy *PC*, napisana została w języku *C* w środowisku *CodeBlocks*. Zgodnie z informacjami zawartymi w podrozdziale 3 (*Rysunek 4.3*) na samym początku następuje utworzenie i inicjalizacja *Socket'a* właściwym adresem IP Serwera jak również numerem portu na którym tenże nasłuchuje po czym następuje nawiązanie pomiędzy nimi połączenia.

Aplikacja otwiera obraz który ma zostać przesłany, następnie odczytuje jej rozmiar i wysyła go do Serwera.

Dalej zostaje zaalokowana dynamiczna tablica zmiennych typu *char* o rozmiarze wysyłanego obrazu do której wpisywane są dane. Kolejne bajty zostają przesyłane poprzez protokół *TCP/IP* porcjami aż do końca pliku gdzie następuje zwolnienie zaalokowanej pamięci.

Po przesłaniu zdjęcia w formacie *BMP* Klient oczekuje na informację zwrotną ze strony Serwera w postaci dwóch liczb reprezentujących pozycję *X* oraz *Y* środka ciężkości piłeczki. Po ich otrzymaniu następuje „sprzątanie” nieistotnych informacji tj. zamknięcie pliku, zwolnienie zaalokowanej pamięci oraz przypisanie wartości *0* wszystkim zmiennym wykorzystywanym w toku operacji. Na samym końcu Klient zamyka połączenie z Serwerem i zwraca *0* z funkcji *Main* kończąc tym samym swoje działanie.

Aplikację można uruchomić z poziomu konsoli systemowej podając trzy argumenty:

1. Adres IP Serwera.
2. Adres Portu na którym nasłuchuje Serwer.
3. Ścieżkę dostępu do zdjęcia w formacie *BMP* na którym mają zostać dokonane obliczenia.

4.7 Aplikacja serwera

Jak wspomniano we wstępie do tego rozdziału, aplikacja Serwera, a więc przeznaczona na część *PS* rozpatrywanego układu, napisana została w środowisku *Xilinx SDK*, w języku C. Podobnie jak aplikacja Klienta, tak również aplikacja Serwera rozpoczyna swoje działanie od utworzenia *Socket`a* z wybranym przez użytkownika numerem portu jako argumentu. Następnie Serwer przechodzi w tryb nasłuchu oczekując próby nawiązania połączenia ze strony Klienta. Po jej nawiązaniu następuje przesłanie ze strony Klienta rozmiaru w bajtach obrazu który jest przez niego przetwarzany. Aplikacja odbiór właściwego obrazu rozpoczyna od otwarcia w trybie zapisania danych binarnych „wb” pliku o nazwie *Result.bmp*. Następnie odbiera od Klienta kolejne porcje danych aż ich ilość nie będzie równa odebranemu uprzednio rozmiarowi obrazu. Dalej następuje zamknięcie pliku oraz wywołanie funkcji *ReadBMP2* na nim otrzymując wskaźnik do dwuelementowej tablicy z poprawnie usytuowanymi wartościami kolejnych pikseli.

Kolejnym krokiem działania aplikacji było zmapowanie obszaru pamięci rozpoczynającego się pod adresem *0x80000000* o wielkości 256 bajtów, co odpowiada 64 istniejącym adresom 32 bitowych rejestrów utworzonych dla układu *PS*. Adres ten przypisany został utworzonej instancji magistrali AXI w czasie tworzenia *bitstream`u* i stanowi adres początkowy jej pamięci, której adres końcowy, tj. wysoki wynosi *0x800000FFF*. Do obsługi tego zmapowanego obszaru utworzono 31 zmiennych typu *volatile unsigned int*, każdy odpowiadający adresowi o 4 bajty wyższemu niż jego poprzednik.

Ukończywszy wszystkie powyższe kroki aplikacja przystępowała do przesyłu danych do części *PL* układu. Transmisja rozpoczynała się wystawienia a następnie zgaszenia sygnału *put_done_down* oraz *rst* w celu wyzerowania niepotrzebnych wartości które mogły pozostać na magistrali po poprzednim obiegu aplikacji. Wystawiała również sygnał *start* będący znakiem dla modułu *Rej32b* o rozpoczęciu nadawania danych. Następnie przy użyciu pętli *for* następowało wpisanie danych pojedynczego piksela w sposób następujący:

```
*val1=*picture[counter]<<24 /
  (*(picture[counter]+sizeof(unsigned char)))<<16 /
  (*(picture[counter]+2*sizeof(unsigned char)))<<8 /
  (*(picture[counter]+3*sizeof(unsigned char)));
```

W celu wpisania odpowiednich 4 wartości znajdujących się pod adresem wskazywanym przez *picture[counter]* dokonywało się przesunięć bitowych o odpowiednio 24, 16 oraz 8 bitów z równoczesnym wskazywaniem na kolejne adresy danego miejsca w pamięci.

Po wpisaniu danych na magistralę następowało ustawienie oraz wygaszenie sygnału *clock_enable* w celu możliwości spróbkowania danych przez moduł *PS* odpowiedzialny za znajdowanie środka ciężkości, a ściślej mówiąc, jego część zajmującą się ustalaniem kolejnych pozycji pikseli w obrazie. Z powodu drobnych problemów z synchronizacją poszczególnych sygnałów wprowadzono do pętli również opóźnienie w postaci funkcji *nanosleep()*.

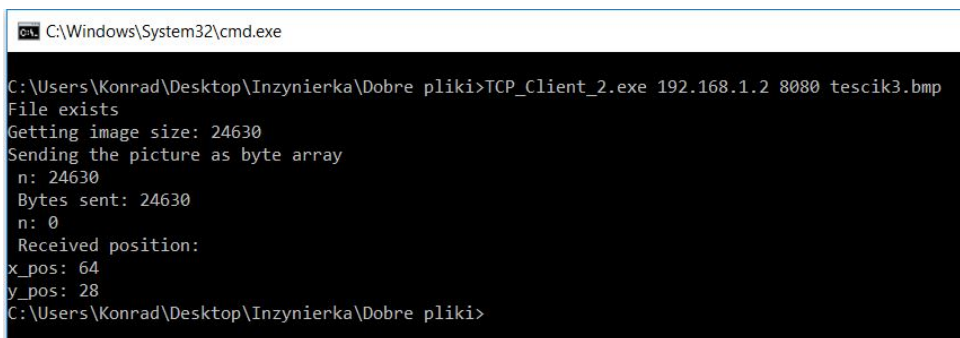
Gdy licznik wysłanych pikseli *counter* doszedł do wartości $[szerokość \cdot wysokość - 1]$ następowało wyjście z pętli. Aplikacja wystawiała wtedy sygnał *clock_enable* na 1 aby moduł FPGA mógł dokończyć wszystkie obliczenia związane z dzieleniem wartości m_{00} , m_{01} i m_{10} (2,3,4) zgodnie z (5,6).

Aplikacja wchodziła wtedy w stan bezczynności aż do otrzymania z modułu *Rej32b* flagi *Done_xy*, symbolizującej zakończenie obliczeń przez moduł FPGA. Zerowała wtedy sygnał *start* oraz *clock_enable* oraz wypisywała do konsoli odnalezione wartości *X* oraz *Y*.

Mając już dane poszukiwane wartości Serwer wysyłał je po kolei z powrotem do Klienta po czym przystępował do czyszczenia niepotrzebnych danych, z czego najważniejszym było zwolnienie pamięci zaalokowanej na wartość zwrótną z funkcji *ReadBMP2()*, gdyż w przeciwnym wypadku dochodziłoby do zjawiska wycieku pamięci co przy testach aplikacji skutkowało pojawieniem się w konsoli błędu *segmentation fault*.

Po wykonaniu wszystkich opisanych kroków aplikacja Serwera była gotowa do przyjęcia kolejnego połączenia z Klientem w celu przetworzenia kolejnej porcji danych w postaci zdjęcia w formacie BMP.

Na *Rysunku 4.4* oraz *4.5* ukazano przykładowy wygląd konsoli systemu *PetaLinux* oraz *Windows* w trakcie pracy obydwu aplikacji. Jak można zauważyć z kolejnych komend wyświetlanych w poszczególnych konsolach, droga informacji o obrazie przebiega w sposób opisany w dwóch powyższych podrozdziałach: Po nawiązaniu komunikacji z Serwerem Klient wysyła zdjęcie, następnie jest ono przetwarzane a końcowy wynik zostaje zwrócony do zleceniodawcy.



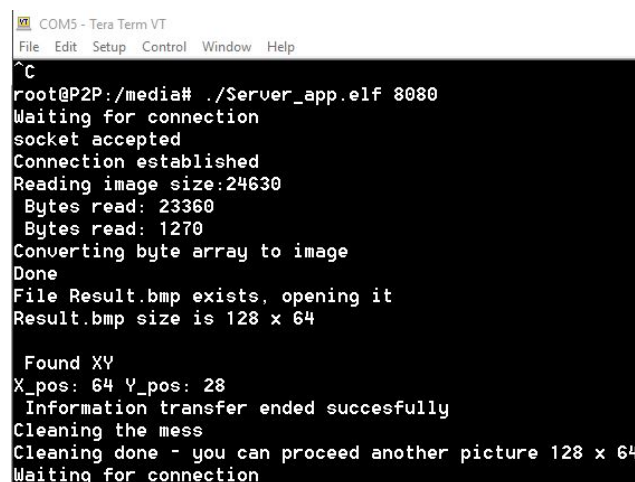
```

C:\Windows\System32\cmd.exe

C:\Users\Konrad\Desktop\Inzynierka\Dobre pliki>TCP_Client_2.exe 192.168.1.2 8080 tescik3.bmp
File exists
Getting image size: 24630
Sending the picture as byte array
n: 24630
Bytes sent: 24630
n: 0
Received position:
x_pos: 64
y_pos: 28
C:\Users\Konrad\Desktop\Inzynierka\Dobre pliki>

```

Rysunek 4.5 Konsola z programem Klienta



```

COM5 - Tera Term VT
File Edit Setup Control Window Help

^C
root@P2P:/media# ./Server_app.elf 8080
Waiting for connection
socket accepted
Connection established
Reading image size:24630
Bytes read: 23360
Bytes read: 1270
Converting byte array to image
Done
File Result.bmp exists, opening it
Result.bmp size is 128 x 64

Found XY
X_pos: 64 Y_pos: 28
Information transfer ended succesfully
Cleaning the mess
Cleaning done - you can proceed another picture 128 x 64
Waiting for connection

```

Rysunek 4.5 Konsola z programem Serwera

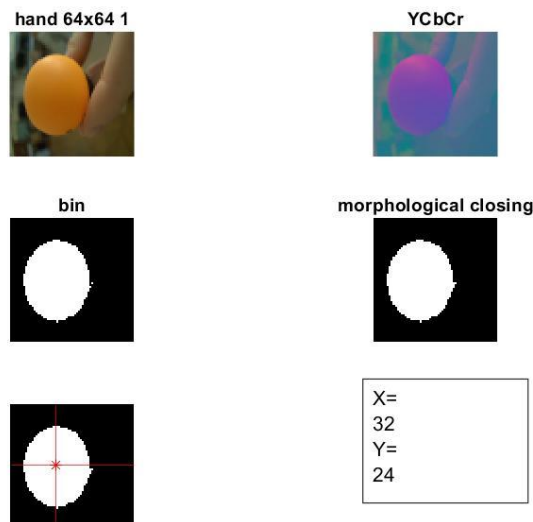
Rozdział 5 Przedstawienie wyników

5.1 Wstęp

W celu sprawdzenia poprawności działania opracowanych modułów utworzono model programowy w środowisku *Matlab*, jak również odpowiednie moduły testujące w środowisku Vivado przy pomocy komend języka *TCL*. Również w aplikacji Serwera zamieszczono odpowiednie funkcje języka C wypisujące uzyskane dane do zewnętrznego pliku *txt* w celu ich późniejszej weryfikacji. Wyniki otrzymane w czasie rzeczywistej pracy układu porównywano z nimi.

5.2 Model programowy

Utworzony w środowisku *Matlab* model programowy podzielony był na osobne bloki kodu odpowiadające poszczególnym podmodułom z modułu FPGA realizującym znalezienie środka ciężkości. Pobierał on zdjęcie w formacie PPM o wymiarach 64 x 64 pikseli, realizował zamianę przestrzeni barw z RGB na YCbCr, binaryzację, morfologiczne zamknięcie, znalezienie środka ciężkości wykorzystując wzory stosowane we właściwym module oraz wyrysowanie wyników i jego zwrócenie do nowego pliku. *Rysunek 5.1* ukazuje przykładowy wynik symulacji przeprowadzonej przy jego pomocy.

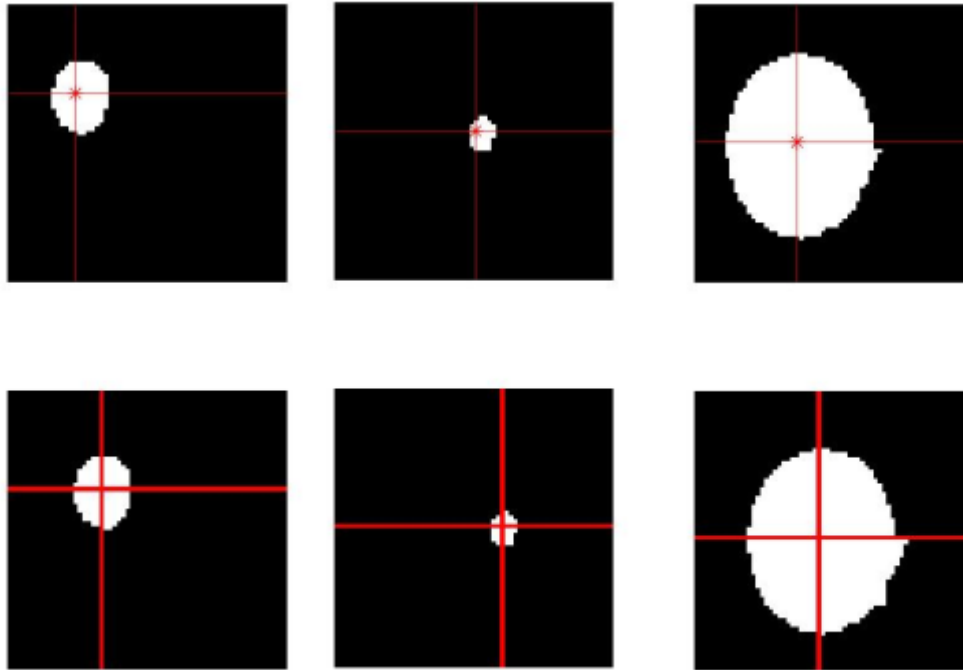


Rysunek 5.1 Wyniki symulacji dla jednego z testowych obrazów.

Czerwone linie przecinają się w odnalezionym punkcie środka ciężkości obiektu. W ramce obok niego widnieje pozycja tego punktu wyrażona w pikselach.

5.3 Symulacje w środowisku Vivado

W środowisku Vivado utworzono kilka symulacji w celu poprawnego przetestowania działania modułu. Pierwszy z nich przetwarzał zdjęcia w formacie *PPM* o wymiarze 64 x 64 pikseli i zwracał zdjęcie z zaznaczonym środkiem ciężkości w celu możliwości jego porównania z wynikami uzyskanymi z modelu programowego. Ten moduł testujący pochodził z materiałów dostarczonych studentom w ramach zajęć z przedmiotu *Systemy Rekonfigurowalne*. Na *Rysunku 5.2* ukazano wynik działania tego modułu testującego oraz modelu programowego w celach porównania.



Rysunek 5.2 Porównanie wyników modelu programowego oraz modułu testującego

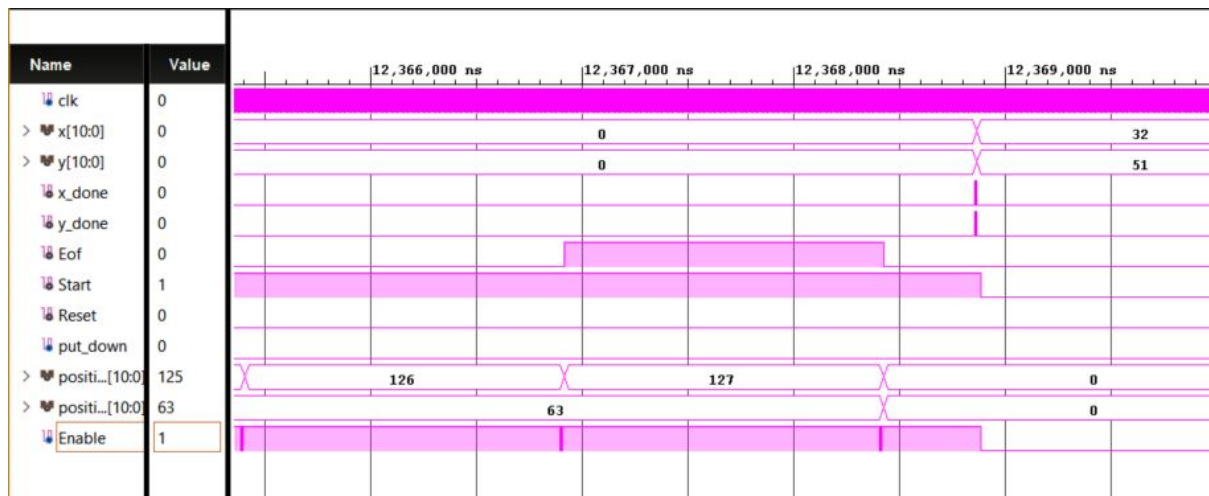
Górna część pokazuje wynik modelu programowego, dolna zaś modułu testującego. Udało się uzyskać prawie identyczne wyniki, różnice wynikać mogły z ewentualnych przesunięć spowodowanych synchronizacją sygnałów kontrolnych w poszczególnych podmodułach.

W trakcie pracy nad realizacją niniejszej pracy dyplomowej wprowadzone zostały modyfikacje do części *PS*, które na pewnym etapie uniemożliwiały już skorzystanie z poprzednich modułów testujących, które to okazały się niewystarczające dla weryfikacji poprawności działania układu na docelowej platformie, stąd też wynikała konieczność utworzenia nowych modułów testujących opisanych poniżej.

Drugi utworzony moduł testujący pobierał na wejście dane w postaci ciągu pikseli wytworzonych przez omawianą w *Podrozdziale 4* funkcję przetwarzającą zdjęcia w formacie *BMP*, a następnie wprowadzał ją na wejście modułu *FPGA*. Czynił tak dla dwóch obrazów w celu przetestowania działania ciągłego modułu.

Na *Rysunku 5.3* ukazano przebieg najważniejszych sygnałów części *PL* w końcowym etapie działania symulacji. Można zaobserwować, iż w momencie pojawienia się pozycji 127 na

liczniku horyzontalnym x_pos następuje wystawienie flagi Eof , czyli końca danej ramki, a chwilę później flag x_done oraz y_done symbolizujących koniec obliczeń – ukazuje się wtedy wartość X oraz Y pozycji.



Rysunek 5.3 Przebiegi sygnałów w symulacji Vivado

5.3 Logowanie danych w testowej aplikacji Serwera

Jednym z problemów które wynikły w toku realizacji zadania był fakt, iż poprawne wyniki symulacji nie implikowały poprawności zachowania się całości po wgraniu na płytkę. Inne taktowanie części PL oraz PS oraz różna długość trwania wykonywania poszczególnych komend języka C sprawiały, iż w zdecydowanej większości przypadków to, co wydawało się działać w słuszny sposób w symulacji dawało złe wyniki w rzeczywistości bądź też nie działało wcale. Wobec tego zdecydowano się na traktowanie wyników symulacji wyłącznie jako wskazówki, natomiast weryfikację poprawności przeprowadzano i zmiany w modułach wprowadzano poprzez konkluzje z wyników obserwacji sygnałów wypisywanych do zewnętrznego pliku *txt* w formie logów w aplikacji Serwera. Sygnały te wyprowadzano na zewnątrz poprzez magistralę AXI i są one widoczne jako sygnały nadmiarowe na *Rysunku 3.10*.

5.4 Weryfikacja działania na podstawie prostych obrazów testowych

Pierwsze testy układu przeprowadzono na prostych obrazach z zarysowaną jedynie piłeczką, tak jak ukazano na *Rysunku 5.5*. Piłeczkę po kolei umieszczano w różnych miejscach i przepuszczano przez układ uzyskując wartości przedstawione w *Tabeli 5.1*. Pozycję prawidłową uzyskiwano z modelu programowego. Otrzymane wyniki wraz z porównaniem z modelem ukazano w *Tabeli 5.1*



Rysunek 5.4 Przykład zdjęcia testowego- Simple1

Tabela 5.1 Porównanie wyników prostych obrazów testowych z modelem

Nazwa	Pozycja X z układu	Pozycja Y z układu	Pozycja X z modelu	Pozycja Y z modelu	ΔX	ΔY
<i>Simple1</i>	109	14	107	15	2	1
<i>Simple2</i>	23	22	20	23	3	1
<i>Simple3</i>	24	42	22	43	2	1
<i>Simple4</i>	110	46	108	47	2	1
<i>Simple5</i>	61	29	59	30	2	1
<i>Simple6</i>	101	26	99	26	2	0
<i>Simple7</i>	85	32	123	32	38	0
<i>Simple8</i>	7	61	5	61	2	0
<i>Simple9</i>	59	3	58	4	1	1
<i>Simple10</i>	6	21	4	22	2	1
<i>Simple11</i>	113	49	114	50	1	1

Jak można zauważyć, rozbieżności pomiędzy pozycją wykrytą układem a pozycją z modelu programowego w prawie wszystkich przypadkach testowych jest niewielka i z dużą dozą prawdopodobieństwa wynika z przesunięć następujących w układzie w ciągu przekazywania kolejnym podmodułom w układzie FPGA porcji danych. Możliwa jest tutaj również rola niedoskonałej synchronizacji pomiędzy układem *PS* a *PL* o której wspomniano w trakcie omawiania aplikacji Serwera w poprzednim rozdziale niniejszej pracy a którą starano się rozwiązać stosując opóźnienie głównej pętli dostarczającej dane.

Wynik uzyskany w obrazie *Simple7* jest jednak mocno odmienny od modelu programowego w pozycji horyzontalnej. Obraz ten przedstawia piłeczkę znajdującą się częściowo w kadrze z prawej strony.

Zdecydowano się więc na utworzenie przypadków testowych zbliżonych do *Simple7* by potwierdzić brak zgodności. Wyniki zebrano w Tabeli 5.2.

Uzyskane wyniki ukazują znaczące różnice, co sugeruje istnienie problemu z poprawnym wykrywaniem pozycji obiektu częściowo zakrytego z prawej strony kadru. Im bardziej dany obiekt jest wysunięty, tj. im więcej piłeczki widnieje w kadrze tym lepsze są uzyskiwane wyniki. Wydaje się, iż problemem tu występującym jest niepoprawne wyświetlanie ostatnich pozycji każdej linii, co może mieć związek z wykrywaniem ich pozycji przez moduł Centroid układu *PS*.

Tabela 5.2 Przypadki problematyczne

Nazwa	Pozycja X z układu	Pozycja Y z układu	Pozycja X z modelu	Pozycja Y z modelu	ΔX	ΔY
<i>Simple7_2</i>	54	18	125	18	71	0
<i>Simple7_3</i>	97	57	122	57	25	0
<i>Simple7_4</i>	105	22	118	22	13	0
<i>Simple7_5</i>	81	55	80	55	1	0
<i>Simple7_6</i>	107	26	117	26	10	0

5.5 Weryfikacja działania na podstawie obrazów uzyskanych z kamery

Kolejnym krokiem były testy przeprowadzone na obrazach uzyskanych z kamery, również o rozmiarze 128×64 . Przykładowy obraz ukazano na Rysunku 5.5 natomiast wyniki zebrano w Tabeli 5.2

Rysunek 5.5 Obraz testowy rzeczywisty – *Real4*

Tabela 5.3 porównanie wyników obrazów rzeczywistych z modelem

Nazwa	Pozycja X z układu	Pozycja Y z układu	Pozycja X z modelu	Pozycja Y z modelu	ΔX	ΔY
<i>Real1</i>	52	31	51	31	1	0
<i>Real2</i>	30	52	27	52	3	0
<i>Real3</i>	70	30	68	30	2	0
<i>Real4</i>	64	28	62	28	2	0
<i>Real5</i>	35	20	34	21	1	1

5.6 Konkluzje

Wyniki uzyskane z obrazów rzeczywistych zachowują się identycznie jak te z obrazów testowych. Występują bardzo niewielkie rozbieżności pomiędzy nimi a pozycją wyliczaną poprzez model programowy w Matlabie. Wśród głównych czynników powodujących brak idealności tych wyników wymienić można:

- Brak idealnej synchronizacji pomiędzy układem *PS* a *PL*.
- Wyliczanie konwersji z przestrzeni RGB na YCbCr przebiega w sposób zbliżony, choć nie identyczny, dla modelu programowego i modułu FPGA.

- Możliwy problem z wyświetlaniem ostatnich wartości pozycji horyzontalnych co dodatkowo może tłumaczyć niedostatecznie dobre wykrywanie pozycji obiektu znajdującego się częściowo poza kadrem z prawej strony.
- Możliwe opóźnienia wewnątrz układu *PS* spowodowane dostosowywaniem modułów do współpracy z układem *PL*.

Rozdział 6 Zakończenie

Należy przypomnieć, iż celem niniejszej pracy było stworzenie układu do detekcji pozycji obiektu kulistego w układzie rekonfigurowalnym. W tym celu należało stworzyć przy pomocy języka HDL Verilog modułu FPGA spełniającego to zadanie bazującego bądź to na charakterystycznym kolorze obiektu bądź to na jego kształcie. Moduł miał współpracować z magistralą AXI przy pomocy której dane miały być do niego dostarczane oraz zwracane z powrotem po wyliczeniu żądanych informacji. Transfer klatek obrazu do układu FPGA oraz zwrot informacji o wykrytym położeniu – środku ciężkości przedmiotu, miał następować przy użyciu aplikacji napisanej dla procesora ARM poprzez łączę Ethernet. Docelowo implementacja miała nastąpić na płycie UltraZed.

Analizując pracę, można stwierdzić iż założony cel został osiągnięty. Opracowano moduł FPGA wykrywający środek ciężkości zdjęć w formacie BMP o rozmiarze 128×64 pikseli. Przy pomocy magistrali AXI możliwy był transfer danych pomiędzy układem *PS* a *PL* w postaci pikseli obrazu oraz informacji o środku ciężkości. Aplikacja napisana na procesor ARM otrzymywała zdjęcia do przetwarzania od innej aplikacji, napisanej na komputer klasy PC, poprzez protokół TCP/IP implementując transfer w relacji Klient – Serwer. Przed wysłaniem zdjęć na układ FPGA aplikacja odpowiednio przetwarzała dane czyniąc je możliwymi do obróbki przez układ *PL*. Dane zwrotne z układu FPGA informujące o wyliczonym środku ciężkości Serwer przysyłał do Klienta występującego z żądaniem przetworzenia zdjęcia.

Utworzony układ działał w sposób zgodny z założeniami, wyliczając środek ciężkości obiektów wysyłanych z aplikacji Klienta z dobrym przybliżeniem. Brak idealności można wytłumaczyć co najmniej kilkoma czynnikami, wśród których najważniejszymi może być niedoskonałość modelu programowego, zmiany konieczne do wprowadzenia dla układu *PL* w celu dostosowania go do możliwości współpracy z układem *PS* czy chociażby brak idealnej synchronizacji sygnałów pomiędzy obydwojema typami układów. Defektem utworzonego układu jest niepoprawne wyliczanie środka ciężkości dla obiektów kulistych skrytych częściowo poza prawym kadrem – nie występuje on jednak już gdy obiekt widoczny jest już z prawej strony w całej okazałości.

Głównym wyzwaniem z jakim przyszło się zmierzyć w niniejszej pracy był brak doświadczenia z układami SoC. Obydwa typy zastosowanych układów, tj. FPGA i procesory ARM same w sobie nie są problematyczne w obsłudze i programowaniu, jednakże ich wzajemna synchronizacja może przysparzać problemów. Układy tego typu jak zastosowane w pracy zostały jednak pomyślane jako układy mające wspierać wzajemnie swoje działanie i rozszerzać je o nowe możliwości. Część związana z układem rekonfigurowalnym dzięki możliwości wykorzystania działań równoległych daje możliwość znacznego przyspieszenia działań w stosunku do układów procesorowych, jednakże nie byłaby ona w stanie działać na danych w takiej postaci w jakiej są one dostarczane od Klienta, czyli w postaci zdjęcia w formacie BMP. Tutaj właśnie nieodzowne okazuje się wsparcie aplikacji napisanej na procesory ARM, która może przerobić dane na postać znacznie bardziej dogodną dla układu FPGA jak również, dzięki

implementacji systemu operacyjnego bazującego na *Linuxie*, może realizować komunikację z Klientem poprzez protokół TCP/IP.

Utworzony układ posiada dalszą możliwość rozwoju, przede wszystkim poprzez lepszą synchronizację pomiędzy układem *PS* oraz *PL* jak również poprzez wyeliminowanie defektów związanych z prawą stroną kadru. Możliwe jest również znaczące przyspieszenie działania całości poprzez przemodelowanie układu tak, aby nie wymagał do swojego działania sygnałów synchronizacji, co jednak zmniejszyłoby uniwersalność takiego modułu. Zastosowana metoda wykrywania środka ciężkości opierająca się na wzorach opisanych w rozdziale *III*, również nie jest metodą jedyną możliwą do zastosowania. Zastosowanie innych metod, opierających się przykładowo nie o kolor a o kształt obiektu którego środek ciężkości chce się uzyskać mogłoby okazać się bardziej optymalnym rozwiązaniem niż zastosowane obecnie i jeszcze bardziej ukrócić czas potrzebny na przetworzenie danych. Zastosowanie takiego rozwiązania wiązałoby się jednak z niemożliwością uzyskania informacji o obiekcie zdeformowanym, czyli chociażby częściowo przyciętym przez kadr. Innym kierunkiem rozwoju który mógłby przyspieszyć działanie mogła by być rozbudowa modułu *Rej32b*, który pierwotnie miał być załącznikiem bufora FIFO do przechowywania danych z kolejnych przychodzących ramek obrazu. Ze względu na ograniczenia czasowe i problemy z synchronizacją układów nie został on dodany do końcowej wersji pracy.

Podsumowując uzyskane rezultaty wraz z przyjętymi założeniami można dojść do wniosku, iż cel został osiągnięty, nadal jednak istnieje szansa poprawy jego działania. Pytaniem jakie należy sobie zadać jest jednak czy bardziej istotną kwestią układu ma być jego uniwersalność czy też optymalność pod kątem zastosowania do jednego, konkretnego celu. Same układy SoC z pewnością stanowią doskonałe narzędzie programistyczne i ich zastosowanie w różnorodnego typu projektach może, dzięki odpowiednio wykwalifikowanym inżynierom, znacząco podnieść jakość efektów prac. Opanowanie ich w pełni wymaga jednak wysokich umiejętności i dużej dawki cierpliwości.

Bibliografia

- [1] Komorkiewicz M. Kryjak T.: *Systemy rekonfigurowalne: skrypt do ćwiczeń laboratoryjnych*, AGH 2018 [dostęp : 2019-11-25]
https://upel.agh.edu.pl/weaiib/pluginfile.php/43782/mod_resource/content/2/Skrypt_SR_22_02_2018.pdf [dostęp : 2019-11-25]
- [2] UltraZed™ PCIe Carrier Card Hardware User Guide
http://zedboard.org/sites/default/files/documentations/5265-AES-UG-ZU-PCIECC-G-UltraZed-EG-PCIe-v1_1.pdf [dostęp : 2019-11-25]
- [3] UltraZed-EG™ SOM
http://www.zedboard.org/sites/default/files/product_briefs/5043-PB-AES-ZU3EG-1-SOM-G-V3.pdf [dostęp : 2019-11-25]
- [4] PetaLinux Tools Documentation
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-petalinux-tools-reference-guide.pdf [dostęp : 2019-11-25]
- [5] Kołek K.: *Programowanie TCP/IP*, materiały wykładowe do przedmiotu *Podstawy komputerowych systemów sterowania*
https://upel.agh.edu.pl/weaiib/pluginfile.php/11144/mod_resource/content/3/TCPIP.pdf [dostęp : 2019-11-25]
- [6] Oracle® VM VirtualBox Installation Instructions for Windows 7 and Linux Virtual Machine Creation Targeting Avnet Development Boards
https://docs.avnet.com/amer/smart_channel/VirtualBox_Installation_Guide_2018_3_v1_12.pdf [dostęp : 2019-11-25]
- [7] AXI Reference Guide
https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf [dostęp : 2019-11-25]
- [8] 13_Improving_Performance Xilinx
Materiały szkoleniowe do praktyk HLS, UJ FAIS [dostęp : 2019-11-25]
- [9] Hajduk Z.: *Wprowadzenie do języka Verilog*, BTC, Legionowo 2009
Biblioteka AGH [dostęp : 2019-11-25]

[10] Gorgoń M.: *Środowisko programowo-sprzętowe do akwizycji, przetwarzania i wizualizacji złożonych sygnałów w oparciu o układy FPGA nowej generacji*, Automatyka, 2006, Tom 10, Zeszyt 3, 107 - 116 [dostęp : 2019-11-25]

Biblioteka AGH

[11] Pawlik P.: *Przekształcenia morfologiczne*, materiały wykładowe dla przedmiotu *Systemy wizyjne*

http://home.agh.edu.pl/~piotrus/sys_wiz/ [dostęp : 2019-11-25]

[12] Linux Article on Xilinx wiki: *Accessing BRAM in Linux*

<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842412/Accessing+BRAM+In+Linux#AccessingBRAMInLinux-3AccessingBRAMUsing/dev/mem> [dostęp : 2019-11-25]

[13] Pylon SDK SAMPLES MANUAL

<https://www.baslerweb.com/en/sales-support/downloads/document-downloads/pylon-sdk-samples-manual/> [dostęp : 2019-11-25]

[14] Gorgoń M.: Wykłady z przedmiotu *Systemy Rekonfigurowalne* [dostęp : 2019-11-25]

[15] YouTube channel ENGR TUTOR : *Vivado 2015.2 CUSTOM IP PART I, PART II, PART III* [dostęp : 2019-11-25]

<https://www.youtube.com/watch?v=qgnTkZ-InK8> [dostęp : 2019-11-25]

[16] StackOverFlow: *read pixel value in bmp file*

<https://stackoverflow.com/questions/9296059/read-pixel-value-in-bmp-file/43140660#43140660> [dostęp : 2019-11-25]

[17] GeekForGeeks: *Socket Programming in C/C++*

<https://www.geeksforgeeks.org/socket-programming-cc/> [dostęp : 2019-11-25]

[18] Microsoft: *Running the Winsock Client and Server Code Sample*

<https://docs.microsoft.com/en-us/windows/win32/winsock/finished-server-and-client-code> [dostęp : 2019-11-25]







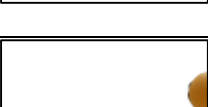
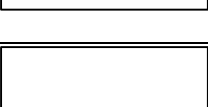

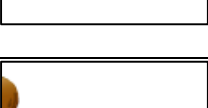
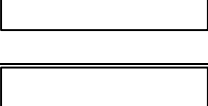
[19] YouTube channel Augmented Startups: *How to Blink an LED on Xilinx Zynq FPGA Devices- part I*


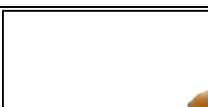
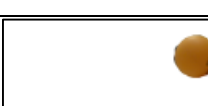
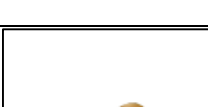
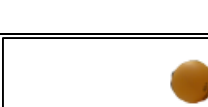

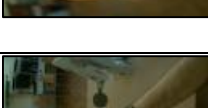

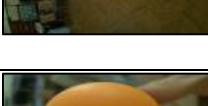
<https://www.youtube.com/watch?v=vvsOdj2ewkI> [dostęp : 2019-11-25]

[20] fpga4student: *Image processing on FPGA using Verilog HDL*

<https://www.fpga4student.com/2016/11/image-processing-on-fpga-verilog.html> [dostęp : 2019-11-25]

Załączniki

Nazwa	Zdjęcie
<i>Simple1</i>	
<i>Simple2</i>	
<i>Simple3</i>	
<i>Simple4</i>	
<i>Simple5</i>	
<i>Simple6</i>	
<i>Simple7</i>	
<i>Simple8</i>	
<i>Simple9</i>	
<i>Simple10</i>	
<i>Simple11</i>	

Nazwa	Zdjęcie
<i>Simple7_2</i>	
<i>Simple7_3</i>	
<i>Simple7_4</i>	
<i>Simple7_5</i>	
<i>Simple7_6</i>	
<i>Real1</i>	
<i>Real2</i>	
<i>Real3</i>	
<i>Real4</i>	
<i>Real5</i>	