Politechnika Poznańska
Wydział Elektryczny
Instytut Automatyki i Inżynierii Informatycznej

Praca dyplomowa inżynierska

FORMAT PRACY DYPLOMOWEJ (PROJEKT ZESPOŁOWY)

Tomasz Kostur
Grzegorz Zachar

Promotor
dr hab. inż. (Marek Kraft) profesor nadzwyczajny

Poznań, 2015 r.

Tutaaj znajdzie się karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Podziękowania:

W tym miejscu można wstawić podziękowania.
Imię i Nazwisko autora 1

Lub nie wstawiać żadnych.
Imię i Nazwisko autora 2

Spis treści

Streszczenie

Obiektem badań pracy jest układ scalony Xillinx Zynq 7000. Jest on przedstawicielem nowego typu układów scalonych łączących w sobie sekwęcyjny mikroprocesor z programowalnym układem fpga.

Praca przedstawia sposób projektowania oprogramowania tego typu układów w połączeniu z systemem operacyjnym linux. Aplikacją realizowaną na układzie jest prosty algorytm stereowizyjny. Praca zawiera porówania różnego rodzaju implementacji algorytmu, realizowanego przy biblioteki OpenCV własnej logiki programowalnej oraz dostarczonego przez xillinx DMA implementowanego w fpga. Praca opisuje wady i zalety używania systemu linux, licznie napotkane problemy programistyczne lub błędy oprogramowania. Praca wskazóje na wiele możliwości optymalizacji aplikacji oraz przedstawia benchmark uzyskanych wyników

Abstract

In this thesis example format of Institute of Control and Information Engineering thesis is presented.

Rozdział 1

Wstęp

Autor: Mateusz Kowalski

1.1 Zawartość wstępu

1.1.1 Opis zagadnienia poruszonego w pracy

Pierwszy rozdział pracy powinien zawierać opis zagadnienia poruszonego w pracy.

1.1.2 Rys historyczny i aktualne zastosowania

Można również opisać historię badañ danego tematu, oraz jego aktualnych zastosowañ.

1.1.3 Autorstwo pracy

Autor: Rafał Kabaciñski

W przypadku pracy zespołowej pod tytułem każdego rozdziału powinno znaleźć się imię i nazwisko autora danego rozdziału. Autorem rozdziału może być tylko jedna osoba, ale podrozdziały mogą mieć innego autora niż cały rozdział.

1.2 Cel pracy

Wstęp powienien zawierać również rozdział opisujący cele projektu opisywanego w pracy jak na przykład:

- zapoznanie się z informacjami na temat istniejących rozwiązañ,
- stworzenie funkcjonalnego urządzenia,
- sprawdzenie wybranych rozwiązañ konstrukcyjnych.

1.3 Zawartość pracy

Ostatecznie wstęp musi zawierać przewodnik opisujący co znajduje się w dalszych rozdziałach pracy. Na przykład: Ogólny zarys stanu wiedzy na temat przygotowania i składu prac dyplomowych przedstawiono w rozdziale drugim. W rozdziale trzecim wprowadzono podstawowe informacje na temat korzystania z tego formatu pracy. Rozdział czwarty stanowi podsumowanie pracy.

Rozdział 2

Stan wiedzy

Oprócz rozdziału opisującego rys historyczny (1.1.2) konieczne jest również opisanie stanu wiedzy na temat danego zagadnienia. Powinno ono siê znaleźć w rozdziale drugim.

2.1 Informacje dotyczące redakcji prac dyplomowych

Informacje dotyczące zasad redakcji prac dyplomowych można znaleźć na stronie `www.cie.put.poznan.pl`.

Rozdział 3

Treść pracy

3.1 Używanie pakietu LaTeX

Informacje o tym jak używać pakietu LaTeXmożna znaleźć w [1] i [2].

3.2 Podział pracy na osobne pliki

Aby ułatwić pracę nad dokumentem, zwłaszcza w przypadku większej liczby zespołów warto go podzielić na większą ilość plików, na przykład według rozdziałów. W takim przypadku poszczególne rozdziały będą napisane w osobnych plikach tex, wstawianych do dokumentu głównego poleceniem $\backslash input$, na przykład:

```
%\input{01-Wstep.tex}
%\input{02-Stanwiedzy.tex}
%\input{03-Tresc.tex}
%\input{04-Wnioski.tex}
```

3.3 Wzory matematyczne w LaTeX

Przykładowy wzór: odwrotna transformata Fouriera:

- Ciągła:

$$f(x) = \mathcal{F}^{-1}\{\hat{f}(\xi)\} = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i x \xi} dx \tag{3.1}$$

- Dyskretna:

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} F_{DFT}(k) e^{j\frac{2\pi}{N}nk} dx \tag{3.2}$$

Przykładowa macierz: elementarna macierz obrotu punktu wokół osi x o kąt α:

$$RotX(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{bmatrix}$$

3.4 Tabele

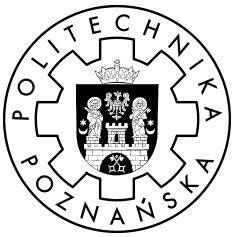
Tabela 3.1 zawiera przykładowe wyniki dwóch sprawdzianów.

Tabela 3.1: Przykladowa tabela

Lp.	nr indeksu	kolokwium	
		I	II
1	32453	4,0	5,0
2	42546	3,5	4,0
3	32546	2,0	3,0

3.5 Rysunki

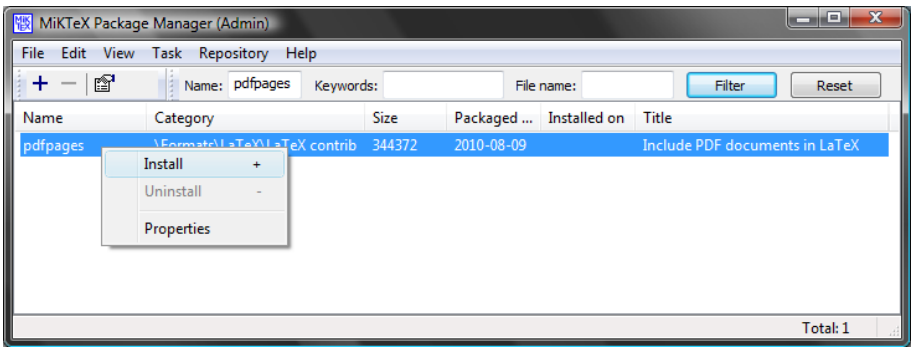
Rysunek 3.1 zawiera logo Politechniki Poznańskiej.



Rysunek 3.1: Logo Politechniki Poznańskiej; uwaga: w podpisach rysunków nie ma kropek na koñcu zdania; jeżeli zdań jest więcej należy oddzielać je średnikami i zaczynać z małej litery

3.6 Dodawanie pakietów

W przypadku użycia pakietu MiKTeX, aby zainstalować dodatkowe pakiety należy włączyć *Package Manager*, w katalogu *Maintenance (Admin)*. Następnie w pole *Name* wpisać nazwę brakującego pakietu i nacisnąć przycisk *Filter*. Nazwę wybranego pakietu należy kliknąć prawym przyciskiem myszy i nacisnąć *Install*, jak na rysunku 3.2.



Rysunek 3.2: Instalacja dodatkowych pakietów

3.7 Pakiet BibTEX

Pakiet BibTEXsłuży do zarządzania bibliografią. Pozycje bibliograficzne zapisywane są w plikach tekstowych z rozszerzeniem *.bib*, a następnie wywołanie poleceniem *\cite*. Pliki bib muszą mieć odpowiednią strukturę, którą można poznać na stronie `http://pl.wikipedia.org/wiki/BibTeX`, lub `http://en.wikipedia.org/wiki/BibTeX`.
Bibliografię tworzy się wywołując w dokumencie polecenie *\bibliography*. Po skompilowaniu dokumentu zawierającego odwołania do bibliografii, należy skompilować bibliografię za pomocą osobnego przycisku, a następnie znów skompilować dokument. Kompiluje się jednak zawsze tylko z widoku głównego dokumentu. BibTEXsam uporządkuje bibliografię według podanego stylu, zamieszczając tylko te pozycje które zostały zacytowane. Styl bibliografii ustawiany jest poleceniem *\bibliographystyle* przed wywołaniem pierwszego cytowania. W tym dokumencie użyto stylu plplain.

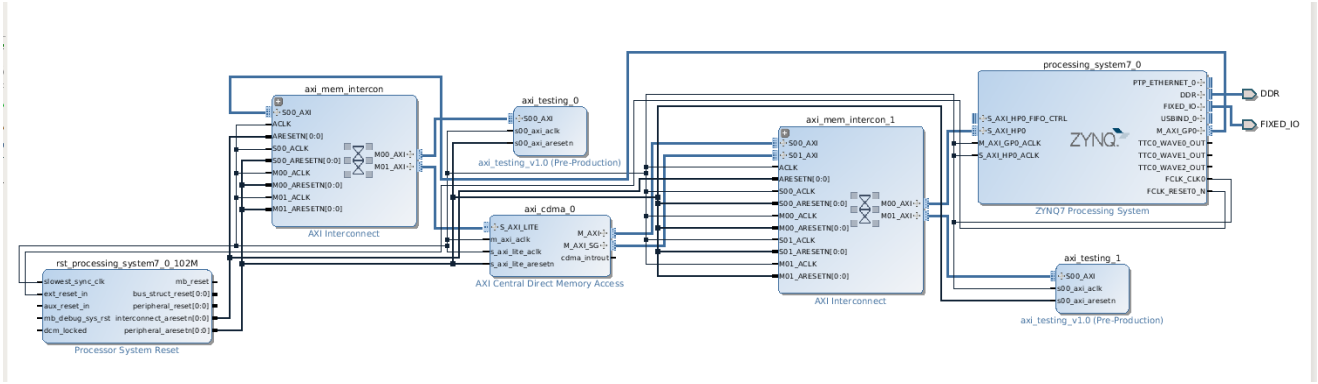
Rozdział 4

Logika Programowalna

Autor: Tomasz Kostur

4.1 Struktura projektowania komponentów logiki programowalnej

Środowisko Xillinx Vivado udostępnia w swoim API jako górną warstwę abstrakcji edytor schematu blokowego.Schemat blokowy jest czytelnym sposobem na szybkie zapoznanie się z projektem i jego komponentami. Komponentami schematu blokowego są bloczki IP (Intelectual Property). W skład bloczków IP wchodzą przede wszystkim urządzenia dostępne poprzez magistralę AXI w jej trzech rodzajach (AXI-lite, AXI-full, AXI-stream), bloczek symbolizujący processor *Zynq7 Processing System*, oraz specjalne bloczki generujące sygnały reset lub obsługujące wybraną część magistrali AXI *AXI Interconnect*. Rysunek 4.1 przedstawia docelowy schemat blokowy projektu.



Rysunek 4.1: Docelowy schemat połączeń komponentów w block design

Docelowy schemat blokowy przedstawia dwie alternatywne instancje jednego ze zbudowanych urządzeń IP. (moduł porównujący opisany w rozdziale 4.3 lub moduł zwracający dysparycje opisany w rozdziale 4.4) Jedna z instancji jest dostępna jako element podlegający bloczkowi procesora, dróga jest dostępna jako slave bloczka *AXI Central DMA* (opisany w rozdziale 4.2). Połączenia pomiędzy bloczkami master-slave są realizowane poprzez bloczek pośredniczący *AXI Interconnect*. Bloczek *AXI Interconnect* realizuje połączenia magistrali na zasadzie: każdy podłączony układ master ma dostęp do każdego podłączonego układu slave.

Podsumowując schemat połączeń:

- Bloczek symbolizujący procesor jest masterem dla bloczków *AXI Central Dma* oraz jednej z instancji bloczka przetwarzającego. Procesor może w ten sposób zapisywać i odczytywać dane z bloczka przetwarzającego oraz zapisywać i odczytywać rejestry konfiguracyjne CDMA.
- Bloczek *AXI Centrall Dma* jest masterem dla drugiej z instancji bloczka przetwarzającego, oraz masterem dla portu procesora *AXI High Performance*. Dzięki połączeniu do dodatkowego portu slave procesora Centrall DMA może zapisywać i odczytywać dane z pamięci RAM procesora.

4.2 Moduł AXI Centrall DMA

Moduł DMA (Direct Memory Access) jest urządzeniem przenoszącym dane z jednego miejsca w pamięci do drugiego. Dzięki DMA można znacznie odciążyć procesor z wielu zadań kopiowania dużych ilości danych w pamięci. Docelowym zadaniem DMA w aplikacji jest całkowite zautomatyzowanie przesyłania danych z pamięci ram procesora do układu fpga i z powrotem. W środowisku Vivado mamy do dyspozycji dwie wersje tego urządzenia

- *AXI DMA* łączące urządzenia *memory mapped*, z urządzeniami z magistralą *AXI stream*. Bloczek potrafiprzesyłać dane oraz konwertować jeden typ danych na drugi.
- *AXI Central Dma* wersja urządzenia służąca do przekazywania danych pomiędzy podległymi mu urządzeniami z magistralą mapowaną jako pamięć (*AXI-lite, AXI-full, AXI High Performance* procesora)

W aplikacji użyty został *AXI Central Dma*
Dwa główne tryby pracy Central DMA to *simple mode* oraz *scattes gatter*

4.2.1 Central DMA - simple mode

Zaprogramowanie zadania przeniesienia danych przez DMA sprowadza się do zapisania w pamięci fizycznej odpowiednich rejestrów konfigurujących CDMA. Adres rejestru startowego jest automatycznie generowany przez Vivado. Kolejne rejestry konfiguracyjne następują po sobie zgodnie z 32-bitowym adresowaniem pamięci (a więc co 4 bajty danych).

W tableli 4.1 przedstawiono poszczególne rejestry CDMA oraz ich krótkie wyjaśnienie. Kolumna *offset* oznacza odległość w bajtach od początkowego rejestru CDMA. Dla CDMA *simple mode* konieczne jest zapisanie rejestrów: kontrolnego *CDMACR*, statusu *CDMASR*, adresu danych do przekopiowania *„SA”*, adresu docelowego *„DA”*, ilości bajtów do przekopiowania *„RTT”*

Po zapisaniu wszystkich rejestrów następuje start pojedyńczego transferu.

4.2.2 Central DMA - scatter gather

Podczas gdy jest niezbędnym wykonanie wielu transferów o różnych adresach source i destinadion, warto jest skorzystać z opcji *„scatter gather”*. Opcja ta umożliwia korzystanie ze specjalne struktury zapisywanej w pamięci, najczęściej ramu procesora, definiującej kolejne zadania CDMA.

Zadania CDMA w opcji *„scatter gather”* są definiowane przez deskryptory. Deskryptory zawierają dane adresów początkowych transferów, docelowych, oraz liczbę

Tabela 4.1: Tabela niezbędnych rejestrów używanych w CDMA

offset	nazwa rejestru	wyjaśnienie
0x00	CDMACR	Central DMA Controll Register. jest odpowiedzialny za wybór trubu pracy CDMA, ustawianie systemu przerwań, resetowanie urządzenia
0x04	CDMASR	Central DMA Status Register. Odpowiedzialny za rozpoznawanie błędów, zaistniałych przerwań, statusu transferu.
0x08	CURDESC_PNTR	Adres pamięci fizycznej aktualnie przetwarzanego deskryptora w *„scatter gather”*
0x10	TAILDESC_PNTR	Adres pamięci fizycznej odtatniego deskryptora w *„scatter gather”*. Zapisanie wartości tego rejestru rozpoczyna serie transferów, osiągnięcie wartości TAILDESC_PTNR przez CURDESC_PNTR kończy serie transferów i ustawia bit w CDMASR sygnalizujący stan *„iddle”*
0x18	SA	Source Address. Adres od którego rozpoczyna się seria danych do transferu w *„simple mode”*
0x20	DA	Destination Address. Początkowy adres miejsca w pamięci do którego dana mają zostać przekopiowane w *„simple mode”*
0x28	BTT	Bytes to Transfer. Liczba bitów jaka ma być przekopiowana w *„simple mode”*.

bitów, podobnie jak do jest w *„simple mode”*. Podanto każdy deskryptor zawiera początkowy adres następnego deskryptora, oraz rejestr statusu wykonania zadania. Struktura pojedyńczego deskryptora zadania CDMA przedstawiona jest w tabeli **??**.

Tabela 4.2: Tabela struktury pojedyńczego deskryptora zadania CDMA

offset	nazwa rejestru	wyjaśnienie
0x00	NEXTDESC_PNTR	Adres fizycznej pamięci następnego deskryptora
0x08	SA	Adres fizyczny początku danych do przekopiowania (*„ Source Address”*)
0x10	DA	Adres fizyczny miejsce docelowego danych (*„ Destination Address”*)
0x18	CONTROL	Ilość bajtów do przekopiowania
0x1C	STATUS	Resestr zapisywany przez CDMA, oznaczający status wykonania zadania definiowanego przez deskryptor. Morze wskazywać na poprawne wykonane zadanie lub wystąpienie błędu.

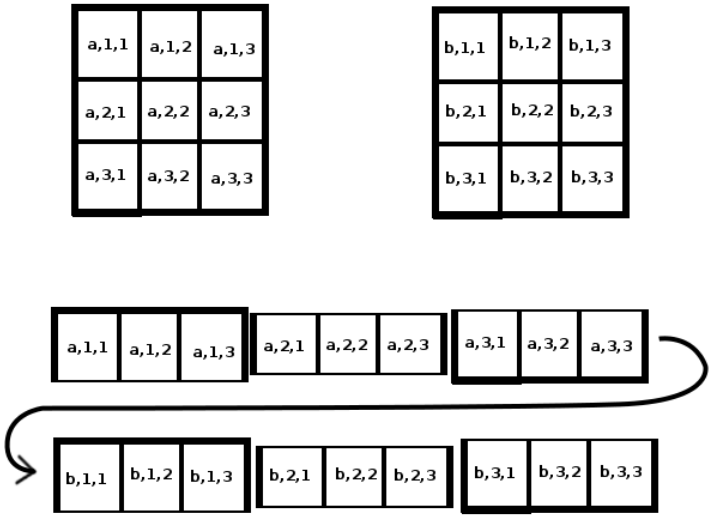
Przykładowa implementacja struktury deskryptorów w pamięci fizycznej została przedstawiona na listingu 4.11.

4.3 Działanie modułu porównującego

Pierwszym modułem jaki wykonano w języku verilog, jest moduł porównujący ze sobą dwie maski obrazu. Wielkość maski jest podawana w sparametryzowany sposób, jako argument syntezy modułu. (Po zakończeniu procesu syntezy i wygenerowaniu bitstream-u) nie ma możliwości zmiany rozmiaru maski.

Docelowy interfejs AXI-full udostępnia wymianę danych pomiędzy procesorem, a logiką programowalną na zasadzie zapisywania i odczytywania do pamięci. W związku z tym wszelkie abrstrkcje przekazywane jako argumenty do dalszego przetwarzania w module muszą zostać przekazane jako jednowymiarowy wektor danych.

Odpowiednie argumenty mają więc odpowiedni przedział adresów. W napisanych urządzeniach przyjęto standardową konwencje zapisywania macierzy od prawej do lewej z góry do dołu. Kolejne macierze, lub dodatkowe przekazywane dane następują bezpośrednio po sobie. Na rysunku 4.2 przedstawiono sposób "wypłaszczania"macierzy.



Rysunek 4.2: Sposób w jaki abstrakcja macierzy zostaje zapisana do pamięci modułu, jako jednowymiarowy wektor danych

Listing 4.1 przedstawia fragment test-bench-u przypisujący z pojedynczego wektora danych *"data vector"* do jakiego mamy dostęp poprzez interfejs AXI-full do etykiet oznaczających abstrakcje macierzy w celu dalszego ich przetwarzania jako argumenty zadania. Takie oznaczenie jest niezbędne dla czytelności kodu i realizacji zadania, nie jest natomiast konieczne oznaczenie macierzy jako tablicy dwuwymiarowej. W dalszej części kodu macierze są przetwarzane jako dwa pojedyncze wektory danych.

Listing 4.1: fragment test bench w języku verilog przedstawiający sposób przypisania

```
1
2  parameter MASK_SIZE = 3;
3  parameter matrix_cells = MASK_SIZE * MASK_SIZE;
4
5      wire [8-1 : 0] mask_left [matrix_cells -1 : 0];
6      wire [8-1 : 0] mask_right [matrix_cells -1 : 0];
7      genvar i;
8      generate
9          for (i=0 ; i<matrix_cells ; i=i+1)begin
10                 assign mask_left[i] = data_vector[i];
11                 assign mask_right[i] = data_vector[i +
                       matrix_cells];
12         end
13     endgenerate
```

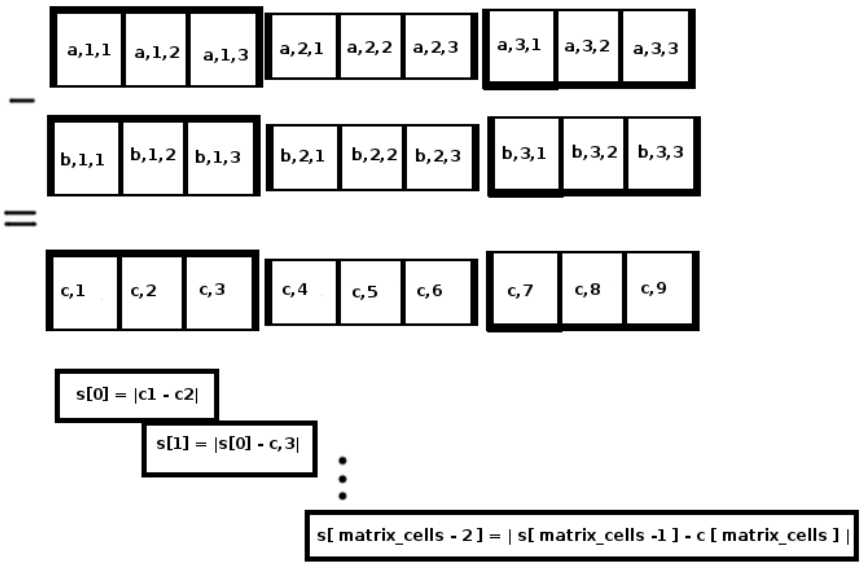
Sposób w jaki porównywane są macierze to odjęcie od siebie, z wartością bezwzględną odpowiadających sobie pikseli, a następnie zsumowanie wszystkich różnic (równanie 4.1). Im mniejszy wynik tym lepsze dopasowanie macierzy.

$$E = \sum_{i=0}^{matrix\_cells} |M_1[i] - M_2[i]| \tag{4.1}$$

Porównywanie zrealizowano na układzie kombinacyjnym dlatego równolegle w czasie dokonują się wszystkie odejmowania. Suma wszystkich odejmowań jest natomiast zrealizowana następująco:

- pierwszy rejestr wektora *śummation_steps"* jest sumą pierwszego odejmowania z drugim.
- każdy następny element wektora *śummation_steps"* jest sumą poprzedniego elementu tego wektora z następnym odejmowaniem.
- ilość elementów wektora *śummation_steps"* jest o jeden mniejsza niż ilość elementów wektora reprezentującego macierz. Ostatni element wektora jest sumą wszystkich odejmowań

Rysunek 4.3 obrazuje ułożenie poszczególnych etykiet połączeń wektora *śummation_steps"* w układzie.



Rysunek 4.3: Sposób w jaki ułożone są etykiety wektorów połączeń realizujące dopasowywanie dwóch macierzy

Listing **??** przedstawia realizację zadania w języku Verilog. Odejmowanie z wartością bezwzględną zostało zrealizowane przez multiplekser dołączony do każdego odejmowania. Multiplekser definiuje kolejność odejmowania od siebie zmiennych, tak aby zawsze mniejsza była odejmowana od większej.

Listing 4.2: Realizacja zadania match-owania macierzy w języku Verilog

```
1 //combinational per element substraction (
    diffrence_per_element is result)
```

```
2  wire [8-1 : 0] diffrence_per_element [matrix_cells -1 : 0];
3  genvar i;
4  generate
5          for ( i=0; i<matrix_cells; i = i+1)begin
6                  assign diffrence_per_element[i] = (mask_left
                       [i]>mask_right[i])? mask_left[i] -
                       mask_right[i] : mask_right[i] - mask_left
                       [i];
7          end
8  endgenerate
9  //combinational sum
10
11 parameter summation_steps_bits = 11 ;// 11 bits for max
       MAX_SIZE = 15
12 wire [summation_steps_bits -1 : 0] summation_steps [
       matrix_cells-2 : 0];
13 generate
14         assign summation_steps[0] = diffrence_per_element[0]
               + diffrence_per_element[1];
15         for (i=0; i<matrix_cells-2; i=i+1) begin
16                 assign summation_steps[i+1] =
                       summation_steps[i] +
                       diffrence_per_element[i+2];
17         end
18 endgenerate
19 wire [summation_steps_bits-1 : 0] diffrence_sum;
20 assign diffrence_sum = summation_steps[matrix_cells - 2];
```

Wynikiem jest zmienna *"diffrence_sum"*. Kody z listingów tego rozdziału są załączone w kod generowany przez Vivado. Dalsze przypisania do magistrali AXI-full są wytłumaczone w rozdziale 4.5.1

4.4 Działanie modułu zwracającego dysparycje

Kolejnym krokiem rozszerzającym jest wykonanie modułu zwracającego wartość dysparycji.

Z powodu na większą złożoność, aby umożliwić sprawne debugowanie kod realizujący zadanie spakowano w dyrektywę *"module"* języka Verilog, zamiast załączać go w całości do kodu magistrali AXI-full tak jak to zrobiono w podrozdziale 4.3

Ograniczeniem dyrektywy *"module"* jest brak możliwości przekazywania argumentów jako tablic wielowymiarowych lub wektorów. Jedyną możliwością przekazywania argumentów są pojedyncze wektory bitów(listing 4.3). Za każdym razem więc chcąc przekazać dane do modułu należy je "wypłaszczyć"do takich wektorów(listing 4.4), wewnątrz modułu należy je z powrotem oznaczyć etykietami połączeń jako tablice lub odpowiednio inne typy danych (listing 4.5).

Listing 4.3: wywoływanie kodu modułu śtereo_solver"urządzenia w języku Verilog

```
1  stereo_solver #
2  (
```

```
3           .MASK_SIZE(MASK_SIZE),
4           .MATCH_WIDE(MATCH_WIDE),
5           .POSITION_BITS(8)
6   )
7   solver_inst(
8           .flattern_mask(flattern_mask),
9           .flattern_match_array(flattern_match_array),
10          .mask_position(mask_position),
11          .match_position(match_position),
12          .DISSPARITION(dissparition)
13  );
```

Listing 4.4: oznaczenie części wektora danych "data_vector"jako poszczególne zmienne oraz wypłaczczanie wektorów danych do wektorów bitów

```
1
2   wire [8-1 : 0] mask [matrix_cells-1 : 0];
3   wire [8-1 : 0] match_array [MASK_SIZE*MATCH_WIDE-1 : 0];
4
5   generate
6           for(i=0; i<matrix_cells; i=i+1)begin
7                   assign mask[i] = data_vector[i][8-1 : 0];
8           end
9           for(i=0; i<MASK_SIZE*MATCH_WIDE; i=i+1)begin
10                  assign match_array[i] = data_vector[i+
                        matrix_cells][8-1 : 0];
11          end
12          for(i=0; i<MASK_SIZE*MASK_SIZE; i=i+1)begin
13                  assign flattern_mask[i*8 +: 8] = mask[i];
14          end
15
16          for(i=0; i<MASK_SIZE*MATCH_WIDE; i=i+1)begin
17                  assign flattern_match_array[i*8 +: 8] =
                        match_array [i];
18          end
19          assign mask_position = data_vector [matrix_cells +
                MASK_SIZE*MATCH_WIDE];
20          assign match_position = data_vector[matrix_cells +
                MASK_SIZE*MATCH_WIDE+1];
21
22  endgenerate
```

Listing 4.5: odzyskiwanie struktury wektorów danych z wektora bitów

```
1
2   wire [MASK_SIZE*MASK_SIZE-1 : 0] maskA [MATRIX_CELLS-1 :
        0];
3   wire [MASK_SIZE*MASK_SIZE-1 : 0] maskB [MATRIX_CELLS-1 :
        0];
4   //wypelnianie tablicy z wektora bitow
```

```
5  genvar i;
6  generate
7          for(i=0; i<MATRIX_CELLS; i=i+1) begin
8                  assign maskA[i] = flattern_maskA [i*8 +: 8];
9                  assign maskB[i] = flattern_maskB [i*8 +: 8];
10         end
11 endgenerate
```

Listing 4.6: deklaracja modułu realizującego dopasowywanie macierzy

```
1
2  module matcher #
3  (
4          parameter MASK_SIZE = 3,
5          parameter MATRIX_CELLS = 9,
6          parameter SUMMATION_STEPS_BITS = 11 //11 for max
               MASK_SIZE = 15;
7  )
8  (
9          input wire  [8*MATRIX_CELLS-1 : 0] flattern_maskA,
10         input wire  [8*MATRIX_CELLS-1 : 0] flattern_maskB,
11         output wire [SUMMATION_STEPS_BITS-1 : 0] match
12 );
```

Aby zachować wspomnianą wcześniej wygodę przy debugowaniu oraz pewność poprawnego działania w dyrektywę *"module"* spakowano również kod z dopasowywujący macierze z podrozdziału 4.3. Deklaracja modułu jest widoczna na listingu 4.7.

Zadania jakie więc należy wykonać w module zwracającym dysparycję to:

- Pobranie maski dopasowania.
- Pobranie przestrzeni dopasowania macierzy.
- Wygenerowanie wektora porównań maski z przestrzenią porównań.
- Wybór najlepszego dopasowania z wektora porównań.
- Obliczenie dysparycji na podstawie danych: położenia maski, połorzenia przestrzeni porównań, numeru najlepszego porównania.

Wektor porównań został stworzony dzięki dyrektywie *"generate"* która w sparametryzowany sposób umożliwia wygenerowanie odpowiedniej ilości modółów porównujących do syntezy. Wygenerowanie wektora porównań z użyciem modułu porównującego (deklaracja listing 4.7) przedstawione jest na listingu **??**

Listing 4.7: wygenerowywanie wektora porównań macierzy z przestrzenią porównań

```
1
2  genvar i,j,k;
3  parameter matches_vector_cells = MATCH_WIDE-(MASK_SIZE-1);
4  // deklaracja wektora kolejnych porownan
```

```
5  wire [13-1 : 0] matches_vector [matches_vector_cells-1 : 0];
       //13 od 255*25 = 6000 ponad wiec 2^13
6  // 13 bitow dla maksymalnej maski 5x5
7  generate
8  // petla generujaca kolejne wyniki porownan
9       for(i=0; i<matches_vector_cells; i=i+1)begin
10          wire [8-1 : 0] match_mask [MASK_SIZE-1 : 0][
                MASK_SIZE-1 : 0];
11 // petla wskazujaca wlasciwa maske do porownania z posrod
       pikseli przestrzeni porownan
12          for(j=0; j<MASK_SIZE; j=j+1)begin
13              for(k=0; k<MASK_SIZE; k=k+1)begin
14                  assign match_mask[j][k] =
                        flattern_match_array[((k+i)*8+j*
                        MATCH_WIDE*8)  +: 8];
15              end
16          end
17 //wyplaszczanie maski do wektora bitow w celu przekazania do
       modulu porownujacego
18          wire [8*mask_cells-1 : 0] flattern_match_mask;
19          for (j=0; j<MASK_SIZE; j=j+1) begin
20              for(k=0; k<MASK_SIZE; k=k+1)begin
21                  assign flattern_match_mask[j*MASK_SIZE*8+k*8
                        +: 8] = match_mask[j][k];
22              end
23          end
24 // przekazanie maski oraz maski wycietej z przestrzeni
       porownan do modulu porownujacego
25          matcher #
26          (
27              .MATRIX_CELLS(mask_cells),
28              .SUMMATION_STEPS_BITS(11)
29          )
30          gen_matcher (
31              .flattern_maskA(flattern_mask),
32              .flattern_maskB(flattern_match_mask),
33 // wypelnienie odpowiedniej komorki wektora porownan
34              .match(matches_vector[i])
35          );
36      end
37 endgenerate
```

Aby stworzyć strukturę która w sparapetryzowany sposób wybierała najmniejszy z pośród elementów wektora dopasowań, będąc układem kombinacyjnym postanowiono użyć następującego sposobu.

- Stworzono wektor *śort_vec"* którego początkowa część jest tożsama z wektorem kolejnych porównań

- każdy następny element wektora *śort_vec"* jest minimalną wartością z dwóch kolejno nastpujących po sobie komórek tego wektora. Dwie wykorzystane komórki do porównań nie są już wykorzystywane w następnych porównaniach dlatego podczas gdy krok inkrementacji komórki wpisywanej jest równy 1, to inkrementacja numerów komórek do wyboru wynosi 2. Wyjaśnia to rysunek 4.5.

- W ten sposób po odpowiednio dużej ilości porównań końcowy element wektora *śort_vec"* będzie najmniejszą jego wartością. Każda część wektora będąca początkiem pobierania do porównania dwóch wyników poprzednich porównań jest nazywana w dalszej części ŝtopniem porównań".

- kolejne stopnie porównań wektora mają wymiar malejącego ciągu geometrycznego o ilorazie

$$q = \frac{1}{2}$$

dlatego długość całego wektora i wynikająca z niej potrzebna ilość iteracji jest częścią całkowitą z sumy nieskończonego ciągu geometrycznego o pierwszym wyrazie równym długości wektora porównań.

$$sort\_vec\_cells = \left\lceil \frac{matches\_vector\_cells}{1 - \frac{1}{2}} \right\rceil \tag{4.2}$$

- Jeśli dla ułatwienia tak dobrać wielkość maski i przestrzeni porównań aby wektor porównań miał by wielkość równą jakiejś potęgi dwójki wówczas mamy pewność, że wszystkie składniki ciągu są liczbami całkowitymi, a długość wektora *sortvec* upraszczała by się do:

$$sort\_vec\_cells = \frac{matches\_vector\_cells}{1 - \frac{1}{2}} - \frac{1/2}{1 - \frac{1}{2}} \tag{4.3}$$

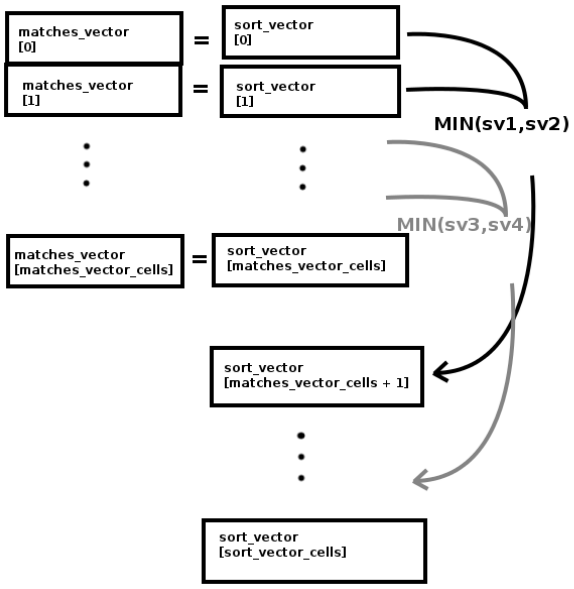
$$sort\_vec\_cells = matches\_vector\_cells * 2 - 1 \tag{4.4}$$

- dołączając równoległy wektor wypełniony kolejnymi liczbami i wykonując na nim identyczne operacje przenosin wartości co w *sort_vec* na jego końcy otrzymujemy numer (jego odległość od początku *sort_vec*) najlepszego dopasowania.

Sposób w jaki zaimplementowano powyższy algorytm w języku Verilog przedstawiono na listingu 4.8

Listing 4.8: kombinacyjne wybieranie najmniejszego elementu wektora w języku Verilog

```
1
2 parameter sort_vec_cells = matches_vector_cells*2-1;
3 wire [8-1 : 0] sort_vec [sort_vec_cells-1 : 0];
4 wire [8-1 : 0] sort_vec_cell [sort_vec_cells-1 : 0];
5
6 generate
7     for(i=0; i<matches_vector_cells; i=i+1)begin
8         assign sort_vec[i] = matches_vector[i];
9         assign sort_vec_cell[i] = i;
```



Rysunek 4.4: Sposób w jaki w całkowicie sekwęcyjny sposów uzyskano minimum z wektora

```
10      end
11      for(i=0; i<sort_vec_cells-1; i=i+2)begin
12          localparam push_cell = i/2 + matches_vector_cells;
13          assign sort_vec[push_cell] = (sort_vec[i] < sort_vec
                [i+1])? sort_vec[i] : sort_vec[i+1];
14          assign sort_vec_cell[push_cell] = (sort_vec[i] <
                sort_vec[i+1])? sort_vec_cell[i] : sort_vec_cell[
                i+1];
15
16      end
17  endgenerate
18
19  wire [8-1 : 0] min_of_vector;
20  wire [8-1 : 0] min_of_vector_cell;
21  assign min_of_vector = sort_vec [sort_vec_cells-1];
22  assign min_of_vector_cell = sort_vec_cell [sort_vec_cells
        -1];
```

4.5 Działanie generycznego kodu magistrali axi

W Xillinx Vivado tworzenie bloczka IP jest ułatwione dzięki automatycznemu generowaniu części kodu logiki programowalnej implementującego bloczek z wybraną przez nas ilością i rodzajem obsługiwanych magistrali AXI. Bloczek składa się z urządzenia nadrzędnego zbierającego z sobie wszystkie urządzenia magistrali, oraz z plików poszczególnych urządzeń. Ilość oraz rodzaj obsługiwanych urządzeń magistralowych jest dowolna. W topowym module powinno unikać się wszelkich funkcjonalnych operacji logicznych.

Magistralami możliwymi do wyboru są:

- AXI-lite, charakteryzująca się „mało kosztownym" rodzajem implementacji, przekazywane dane są tutaj poprzez generowane rejestry o nazwie *„slv_regX"* gdzie X oznacza numer konkretnego rejestru. Magistrala nie jest szybka i słurzy przede wszystkim do przekazywania zmiennych konfiguracyjnych urządzeń.
- AXI-stream. W przeciwieństwie do AXI-lite i AXI-full nie jest ona magistralą realizowaną na zasadzie modelowania pamięci. Axi-stream jest szybką magistralą umożliwiającą łączenie ze sobą pojedyńcze magistrale w szeregu na zasadzie master -¿ slave. Nie jest natomiast możliwe stworzenie sieci urządzeń działających na zasadzie klient server, z tego powodu m.in nie można podłączyć jej bezpośrednio do procesora. Możliwe jest konwertowanie wersji *stream* do *memory mapped*.
- AXI-full jest szybszą wersją magistrali AXI-lite. Sposób w jaki odczytujemy przekazane do niej dane jest dokładniej przedstawiony w rozdziale 4.5.1.

W aplikacji została użyta magistrala AXI-full ze względu na dostęp do przykładów, oraz możliwość bezpośredniego połączenia z bloczkiem *„ Zynq Processing System 7"*.

4.5.1 AXI full

Pośród wielu linijek wygenerowanego automatycznie, istnieje przykład implementacji pamięci obsługiwanej przez AXI-full. Przykład ten posłurzył do stworzenia własnego wektora danych przystosowanego do obsługi napisanych w rozdziałach 4.3 oraz 4.4 modółów.

Listing 4.9: fragment kodu łączący wygenerowany kod magistrali AXI-full z własnym modyłem

```
1  // sygnaly pozwolenia na zapis/odczyt do wektora danych
2  // data_vector  z magistrali
3  wire dvec_dren;
4  wire dvec_wren;
5  assign dvec_wren = axi_wready && S_AXI_WVALID;
6  assign dvec_rden = axi_arv_arr_flag;
7  // proces zapisania do wektora danych z magistrali
8  always @ (posedge S_AXI_ACLK) begin
9      if (dvec_wren && S_AXI_WSTRB[0]) begin
10         data_vector[mem_address] <= S_AXI_WDATA[8-1:0];
11     end
12 end
13 // wyplaszczanie wektora danych do wektora bitow w celu
       przekazania
14 // danych do kolejnego modulu
15 wire [8-1 : 0] mask [matrix_cells-1 : 0];
16 wire [8-1 : 0] match_array [MASK_SIZE*MATCH_WIDE-1 : 0];
17
18 generate
19     for(i=0; i<matrix_cells; i=i+1)begin
20         assign mask[i] = data_vector[i][8-1 : 0];
21     end
22     for(i=0; i<MASK_SIZE*MATCH_WIDE; i=i+1)begin
```

```
23          assign match_array[i] = data_vector[i+matrix_cells
                ][8-1 : 0];
24      end
25      for(i=0; i<MASK_SIZE*MASK_SIZE; i=i+1)begin
            assign flattern_mask[i*8 +: 8] = mask[i];
27      end
28
29      for(i=0; i<MASK_SIZE*MATCH_WIDE; i=i+1)begin
30          assign flattern_match_array[i*8 +: 8] = match_array
                [i];
31      end
32      assign mask_position = data_vector [matrix_cells +
            MASK_SIZE*MATCH_WIDE];
33      assign match_position = data_vector[matrix_cells +
            MASK_SIZE*MATCH_WIDE+1];
34
35  endgenerate
36  // odczyt z wektora do magistrali
37  always @ (posedge S_AXI_ACLK) begin
38      data_vector[matrix_cells + MASK_SIZE*MATCH_WIDE + 2] <=
            dissparition;
39  end
40  // przekazanie danych do modlu
41  stereo_solver #
42      (
43      .MASK_SIZE(MASK_SIZE),
44      .MATCH_WIDE(MATCH_WIDE),
45      .POSITION_BITS(8)//(POSITION_BITS)
46      )
47      solver_inst(
48      .flattern_mask(flattern_mask),
49      .flattern_match_array(flattern_match_array),
50      .mask_position(mask_position),
51      .match_position(match_position),
52      .DISSPARITION(dissparition)
53      );
```

W uzyskanym rozwiązaniu zapisane dane pod odpowiednimi adresami są odpowiadającymi im przekazywanymi zmiennymi. Cokolwiek natomiast zostaje odczytane z magistrali jest wynikiem zwracany przez jeden z sekwęcyjnych modółów opisanych w rozdziałach 4.3 i 4.4. Wynik jest zwracany na wszystkich adresach przestrzeni adresowej urządzenia AXI.

4.6 Narzędzia debugowania

Dużą częścią pracy jest opanowanie oraz napisanie kodów testowych. W czasie tworzenia pracy opanowano 3 rodzaje debugowania są to odpowiednio:

- Bechawioralny symulator offline

- Debugowanie poprzez interfejs JTAG działającej logiki na płytce rozwojowej
- Debugowanie poprzez dostarczony program Xillinx Microprocessor Debugger

W niniejszym rozdziale zostały przedstawione najważniejsze informacje odnośnie używania oraz ograniczeń poszczególnych sposobów weryfikacji programu.

4.6.1 Behawioralny symulator offline

Jest to wstępny i najprostszy sposób debuggowania kodu logiki programowalnej. Mając kod dowolnego modułu (dyrektywa *"module"* w języku Veriog) tworzymy testowy moduł nadrzędny generujący sygnały wejściowe do testowanego modułu. Przykładowy "test bench"pokazany w listingu 4.10.

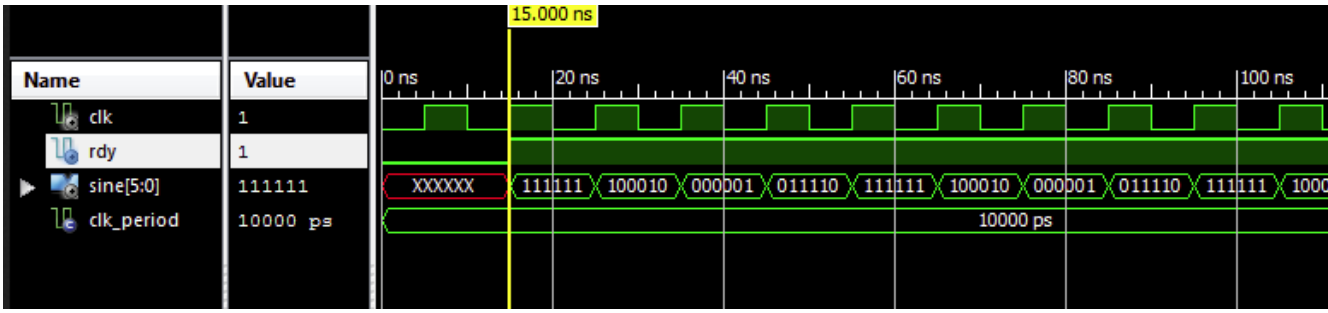
Listing 4.10: Przykładowy fragment kody Verilog użytego jako test bench

```
1
2  `timescale 1 ns / 1 ps
3
4  module stereo_solver_test_bench
5      (
6      );
7  // clock generation for test bench
8      reg clk = 0;
9
10     always begin
11     #5 clk = !clk;
12     end
13
14
15     reg test_reg = 8;
16
17     always @ (posedge clk) begin
18         test_reg <= test_reg -1;
19     end
20 endmodule
```

Jako wyjście z test bench-u otrzymujemy wektory przebiegów w czasie. Rysunek **??**) przedstawia interfejs jaki daje do dyspozycji Xillinx Vivado. Dla testów przy użyciu symulatora język Verilog jak i VHDL udostępniają dodatkowe funkcje nie mające wpływu na syntezę m.in

- dodatkowe stany logiczne: X-nieokreślony, Z-stan wysokiej impedancji
- dyrektywa *initial* wykonująca się tylko raz an początku wykonywania testu
- dyrektywy opóźnień czasowych. Niezbędne przy modelowaniu sygnałów, np sygnału zegarowego, lecz niemożliwe do syntezy bez połączenia z oscylatorem.

Mimo wielu zalet niestety taka symulacja offline nie zawsze jest całkowicie zgodna z modułem stworzonym w procesie syntezy. Symulator używanego środowiska bywał zawodny w następujących przypadkach:



Rysunek 4.5: Sposób w jaki w całkowicie sekwęcyjny sposów uzyskano minimum z wektora

- przypadków w których kluczowymi są różnice pomiędzy zmiennymi typu *reg* a *wire* i odpowiednimi ich operatorami przypisania = oraz ¡=
- Przypisywanie stałych wartości do pewnych elementów wektora
- Rożnych sposobów multipleksowania sygnałów

Zazwyczaj skutkowało to tym, że dobrze symulowany działający moduł, nie był możliwy do syntezy lub implementacji.

4.6.2 Debugowanie poprzez JTAG

Po zakończeniu procesu syntezy istnieje możliwość wskazania niektórych sygnałów i stworzenia do nich wyprowadzeń (*constraints*) do interfejsu JTAG. Po powtórnym procesie syntezy, a następnie implementacji i załadowaniu bitstream-u, wybieramy ilość próbek z jaką mają być pobrane zaznaczone sygnały i sygnały wyzwalania, działające podobnie jak sygnały wyzwalania w oscyloskopie. Wyjściowym API są tutaj przebiegi sygnałów w identycznym interfejsie jaki oferuje symulacja offline **??**

Słabą stroną tego sposobu jest fakt, że nawet przy wyłączonej opcji optymalizacji syntezy, zsyntezowane sygnały i moduły mogą się znacznie różnić od tych jakie zawarliśmy w kodzie. W procesie syntezy wiele kodu zostaje przekształconego lub usuniętego.

4.6.3 Xillinx Microprocessor Debug

Ostatecznym sposobem debugowania logiki programowalnej jest program XMD (Xillinx Microprocessor Debugger). Jest to program udostępniający interfejs konsolowy. Nieoceniony przy łączeniu logiki programowalnej z systemem operacyjnym. Najbardziej przydatnymi funkcjami programu są:

- Możliwość programowania bitsream-u logiki programowalnej podczas pracy procesora
- Możliwość sterowania procesorem poleceniami m.in *rst con stop ...*
- Możliwość swobodnego odczytywania rejestrów pamięci fizycznej, zarówno ram-u jak i pamięci z przestrzeni addresów dostępnych dla processora magistrali AXI. Komendy *mrd* -czytanie z pamięci, oraz *mwr* zapisywanie do pamięci

Program można dowolnie oskryptować używając składni *tcl shell*. Przykładowy skrypt programu używany do testowania DMA w listingu 4.11

Listing 4.11: fragment skryptu tcl wykorzystywany do testowania DMA

```
1  proc dma_sg_test {} {
2  #dane do przekopiowania
3          mwr 0x10000000 10
4          mwr 0x10000004 2
5          mwr 0x10000008 3
6          mwr 0x1000000C 4
7  # teraz deskryptory
8          mwr 0x12000000 0x12000040
9          mwr 0x12000008 0x10000000
10         mwr 0x12000010 0x76000000
11         mwr 0x12000018 4
12
13         mwr 0x12000040 0x12000080
14         mwr 0x12000048 0x10000004
15         mwr 0x12000050 0x76000004
16         mwr 0x12000058 4
17
18 #dwa zapisane do targetu to teraz odczytajmy
19         mwr 0x12000080 0x12000000
20         mwr 0x12000088 0x76000000
21         mwr 0x12000090 0x11000000
22         mwr 0x12000098 4
23
24 # ok no to teraz zapiszmy cdma controll register i inne
25         mwr 0x4E200000 0x00010002
26         mwr 0x4E200000 0x0001000A
27         mwr 0x4E200008 0x12000000
28         mwr 0x4E200010 0x12000080
29 #        mwr 0x4E200000 0x
30
31 #        mrd 0x11000000 3
32
33 }
34
35 proc d_reset {} {
36         rst
37         ps7_init
38         ps7_post_config
39         fpga -f design_1_wrapper.bit
40         clear
41         desc_clear
42 }
```

4.7 Napotkane problemy i możliwości optymalizacji

Poważnymi problemami jakie pojawiły się z związku z logiką programowalną są:

- Nawet w „czystym" nie modyfikowanym kodzie magistrali AXI-full nie udawało się, przy użyciu CDMA przesłać więcej niż 4 bajty danych w obrębie jednego zadania kopiowania pamięci DMA.
- W obrębie jednej tablicy desktyptorów zadań CDMA w *„statter gather"* mode nie udawało się użyć tego samego adresu fizycznego jako „source" i „destination". W przeciwnym razie moduł CDMA zawieszał się nie reagując na żadne komendy, w tym „soft reset".

Jednym ze sposobów działania aplikacji jest:

- Zaalokowanie przy użyciu procesora pamięci fizycznej i wpisanie do nich prawego i lewego obrazka.
- stworzenia tablicy deskryptorów CDMA oraz uruchomienie CDMA
- odczytanie wyniku w postaci całego obrazu dysparycji, obliczonego w logice programowalnej i przeniesionego do ram-u procesora za pomocą CDMA

W ten sposób niemal całkowicie odciążony procesor, miał by zwolniony czas na programy działające w systemie, komunikacje itp. Jednak konsekwencją wymienionych niedoskonałości jest konieczność resetowania zadanej tablicy deskryptorów zanim wystąpi przypadek w którym jeden z *„destination address"* staje się *„source address"*. W programie problem ten rozwiązano odpowiednio dzieląc pełną tablicę deskryptorów i zadając ją segmentami do CDMA w taki sposób aby omawiany błąd nigdy nie wystąpił. Nie jest to jednak rozwiązanie optymalne ponieważ koniecznym jest ciągłe monitorowanie rejestru statusu CDMA.

Być może innym rozwiązaniem tego problemu mogło by się okazać taka implementacja urządzenia AXI w której dane wejściowe mogą być odczytane, a wynik podawany jest jako odrębny dares przestrzeni adresowej modułu IP.

Konsekwencją błędu w którym poprzez jedno zadanie CDMA jesteśmy w stanie przesłać jedynie 4 bajty, co w stworzonej implementacji oznacza zaledwie jeden piksel z obrazu danych wejściowych, jest po raz kolejny spowolnienie przepływu danych, ale też bardzo duże wymagane rozmiary pamięci na dane oraz deskryptory.

Każdy piksel jest przechowywany jako zmienna 32-bitowa co przy dwóch obrazach o rozdzielczości 800x600 daje

$$3 * 800 * 600 * 4 = 5.76Mb \tag{4.5}$$

danych.

Liczba deskryptorów dla urządzenia zwracającego dsparycje to:

$$descriptor\_number = (mask\_size^2 + mask\_size * match\_wide + 3) * result\_width * result\_height$$
$$result\_height = picture\_height - mask_size - 1$$
$$result\_width = picture\_width - mask_size - 1$$

Dodając do tego konieczność, że minimalna odległość od siebie deskryptorów to 64 bajty. Przy rozmiarze maski 3x3, przestrzeni porównań 66x3 musieli byśmy zaalokować

$$(3 * 3 + 3 * 66 + 3) * (800 - 2) * (600 - 2) * 64 \approx 6.4Gb \tag{4.10}$$

danych.

Rozdział 5

Podsumowanie

W podsumowaniu należy pokrótce opisać sposoby i efekty realizacji celów pracy przedstawionych w rozdziale 1.2. Oprócz tego powinny siê tu znaleść wnioski wynikające z wyników pracy, oraz dalsze kierunki rozwoju zagadnienia.

Dodatek A

Opis zawartości płyty DVD

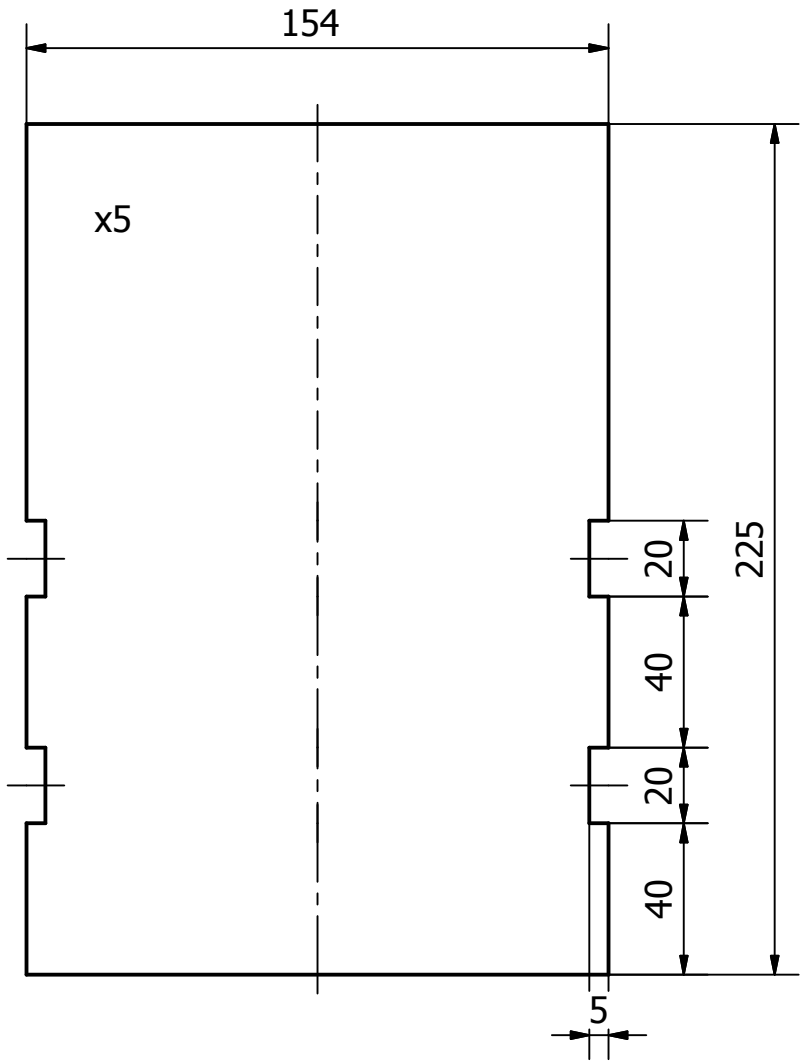
Zawartość płyty DVD lub CD dołączonej do pracy powinna zostać opisana w dodatku. Na płycie powinna znaleźć się cyfrową kopia pracy w formacie pdf.

A.1 Przykładowy podrozdział dodatku

Dodatki mogą zawierać własne rozdziały, które nie ingerują w pozostałą strukturę dokumentu.

Dodatek B

Rysunki techniczne



Konstruował	Data	Podpis		Podziałka	
Mateusz Kowalski	24.06.2010			1:2	
		PODPORA DŁONI			
					Nr Rysunku 01.05

Spis tablic

Spis rysunków

Literatura

[1] Wikipedia o LaTeX. [online]. Dostępny w Internecie: `http://pl.wikipedia.org/wiki/LaTeX`.

[2] Helmut Kopka, PatrickW. Daly. *A Guide to LaTeX: Document Preparation for Beginners and Advanced Users*. Addison-Wesley Professional, wydanie 4th, 2004.

Instytut Automatyki i Inżynierii Informatycznej, Wydział Elektryczny
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Tomasz Kostur \and Grzegorz Zachar",
    title = "Format pracy dyplomowej (projekt zespołowy)",
    school = "Poznan University of Technology",
    address = "Poznań, Poland",
    year = "2015",
}
```