ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

FPGA IMPLEMENTATION FOR ONSITE TARGET DETECTION WITH A LOW COST AND PORTABLE GROUND PENETRATING RADAR SYSTEM

SENIOR DESIGN PROJECT

Hakan DURAK

Ebubekir İNAL

uygundur
Berna Örs Yalçın

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JUNE, 2023

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

FPGA IMPLEMENTATION FOR ONSITE TARGET DETECTION WITH A LOW COST AND PORTABLE GROUND PENETRATING RADAR SYSTEM

SENIOR DESIGN PROJECT

Hakan DURAK
(040150245)

Ebubekir İNAL
(040170238)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JUNE, 2023

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

DÜŞÜK MALİYETLİ VE TAŞINABİLİR YER NÜFUZ EDEN RADAR SİSTEMİ İLE YERİNDE HEDEF TESPİTİ İÇİN FPGA UYGULAMASI

LİSANS BİTİRME TASARIM PROJESİ

Hakan DURAK
(040150245)

Ebubekir İNAL
(040170238)

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

JUNE, 2023

We are submitting the Senior Design Project Report entitled as “PROJECT TITLE”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Hakan DURAK .....
(040150245)

Ebubekir İNAL .....
(040170238)

FOREWORD

We express our sincere gratitude and appreciation to our esteemed advisor, Prof. Dr. Sıddıka Berna Örs Yalçın, for her invaluable guidance, unwavering support, and profound knowledge throughout the duration of our Senior Design Project. We would also like to extend our gratitude to the other assistants and staff members who have assisted us during the course of this project. Finally, we would like to acknowledge the Electronics and Communication Engineering Department of Istanbul Technical University for providing us with the necessary resources and facilities to carry out this project. We are grateful for the opportunities and guidance offered by the department, which have been crucial to our academic and professional development.

June, 2023

Hakan DURAK
Ebubekir İNAL

TABLE OF CONTENTS

Page

ABBREVIATIONS

AXI : Advanced Extensible Interface
ARM : Advanced RISC Machines
CPU : Central Processor Unit
DDR SDRAM: Double Data Rate Synchronous Dynamic Random-Access Memory
DMA : Direct Memory Access
FPGA : Field Programmable Gate Array
GPR : Ground Penetrating Radar
HDL : Hardware Description Language
HLS : High Level Synthesis
HP : High Performance
IP : Intellectual Property
ILA : Internal Logic Analyzer
PC : Personal Computer
PL : Programmable Logic
PS : Programmable System
RAM : Random Access Memory
RADAR : Radio Detection and Ranging RNMF
RNMF : Robust Nonnegative Matrix Factorization
UART : Universal Asynchronous Receiver Transmitter
VHDL : Very High –Speed Integrated Circuit Hardware Description Language

LIST OF TABLES

Page

LIST OF FIGURES

Page

FPGA IMPLEMENTATION FOR ONSITE TARGET DETECTION WITH A LOW COST AND PORTABLE GROUND PENETRATING RADAR SYSTEM

SUMMARY

Various methods have been tried to ensure transfer and processing performance in studies with high-dimensional datasets used. In case studies with processors, it has been observed that the processing speed is insufficient. The necessity of using FPGA for improvement has been emphasized and various studies have been carried out. FPGA provides advantages with its parallel working capability and high flexibility. It enables design perfectionism with the ability to run different processes together, low latency and different optimization options.For the study, ground penetrating radar will be used to detect an object in front of the obstacle. It is designed to enable the application of image recognition, by reprocessing data obtained in a computer environment. A matrix of images is data that has been transmitted from RADAR. A clutter removal algorithm and a RNMF algorithm are used for image processing.

It is intended to use Ground Penetrating Radar in this study for the detection of objects that are hidden by obstacles. The aim is to make it possible to use image detection by reprocessing received data in the electronic environment. A matrix of image data is transmitted to this detection from the RADAR system. Data access is provided independently of the processor by using Direct Memory Access (DMA), thereby reducing CPU load. This results in increased processing speed and reduced cycle times. By using custom hardware, it is possible to achieve maximum efficiency by completing the operation on your processor simultaneously in different blocks. This increases the performance of processing and reduces cycle timesThe time difference obtained in the custom thread enables you to make use of it over multiple loops, resulting in shorter processing times. In this study, different configurations and their results will be focused on transferring RADAR data to the FPGA system, applying the RNMF algorithm to the data and accelerating this process.

1. INTRODUCTION

1.1 Project Introduction and Followed Steps

This senior design project paper named "FPGA Implementation for On-Site Target Detection with a Low Cost and Portable Ground Penetrating Radar System." is within the scope of the TUBITAK 1001 Supporting Scientific and Technological Research Projects, and this project paper is continuation of the previously written and proven C based RNMF algorithm on ZedBoard Zynq-7000 Development Board. Throughout this project, it is aimed to obtain clutter-free target data with rnmf algorithm as fast as possible by using the capabilities of the Zynq-7000 SoC.

In the first part of the project, it is aimed that the radar data read from buried objects is simultaneously read via UART connection of FPGA board. For that purpose, two different approaches are presented by use of FPGA UART peripheral. Both approaches verified by observing sending input data on the TeraTerm terminal screen. Accurate target and clutter GPR images are observed on MATLAB screen after UART implementation. Therefore, it is ensured that the written UART based driver operates in algorithm flawlessly.

After the UART driver is implemented in the design accurately. Standard input data necessity is revealed. To that end, it is decided to send text document including input data from Intel processor-based machine to FPGA card in a single column format. As will be explained in more detail in the related chapter, the number of digits of the input data can be specified as well.

The second part of this project includes hardware implementation of the RNMF code snippets by writing HDL based IPs. There was a Custom IP design which was previously defined. However, this was not applicable for the implementation in the project due to no handshake protocol defined in the IP. So, a Verilog based handshake protocol was written for the IP and tested in the algorithm and verified it is correctness by observing target and clutter GPR images on MATLAB. After observing the operation duration with this newly created IP is long, it is deducted that memory-based

IPs are not sufficient to speed up the algorithm. Hence thorough research was done in the field of Custom IPs. FPGAs parallelism feature was performed by operating several AXI Lite based parallel Custom IPs at the same time. Besides, after getting satisfactory results this feature was also applied for the Stream based IPs. AXI Lite and AXI Stream based IPs are created and compared their features and concluded that stream-based IPs will improve the RNMF algorithm duration drastically.

After the IP characteristic is determined. The need for a DMA entity arose since data transaction between DDR memory and stream-based IPs are only available with the entity of DMA. In the third part of this project DMA properties and advantages are thoroughly researched and explained in the related chapter in this report. As a result, a great improvement has been occurred as the data is transferred directly between the IP and memory without passing through the processor.

As a last step of the project, Vitis HLS tool was explained in the last chapter in depth where you can produce stream-based IPs by writing C code. Optimization techniques such as pipelining was explained and applied to the newly created IPs. With the aim of achieving initiation interval as 1, accumulation and parallelism optimization techniques are introduced in the last chapter. Besides, optimization differences between Vitis HLS tools are introduced in the last chapter as well.

The last chapter of this report includes realistic constraints, conclusions, and recommendations where cost of the project, social, environmental, and economic impacts of the project and the last but not least the implications of the project and suggestions for the future is explained.

1.2 Purpose of Project

As previously defined in the first phase of the TUBITAK. Robust nonnegative matrix factorization (RNMF) model is chosen for detecting and classifying buried objects. For detection of buried objects, it is obligatory to separate the clutter in the radar image from the target object. In this senior design project, it is aimed to:

- Simultaneously reading input data from Radar machine to FPGA in floating-point number and making sure that the algorithm produces target and clutter data properly.

- Classifying code snippets with regard to arithmetic operations and as a result deciding which code snippets will be implemented on PL side of the FPGA for the purpose of speeding up the algorithm.

2. BASIC INFORMATION AND CONCEPTS

2.1 Xilinx Vivado, Vitis and Hls IDEs General Information

Xilinx Vivado Design Suite is a CAD software which enables electronics engineers to program FPGA cards in the most general sense. In doing so, engineers have options to use different HDL languages such as Verilog, VHDL or SystemVerilog. Verilog language is opted for this senior project since better grasping on hardware modelling. Vivado software also offers many no-charge IP cores which help engineers to design and debug their projects. As will be seen in the following chapters, many Vivado IPs were used throughout this senior project such as ILA, AXI Interconnect, DMA and MIG. Vivado Design Suite also enables engineers to synthesize, place and route and produce bitstream file the HDL code. Furthermore, Vivado tool produces a timing report which is extremely important for digital designs in terms of determining whether the design is working in time or not.

In this senior project Xilinx ZedBoard Zynq-7000 SoC FPGA and Artix-7 FPGA based Nexys 4 DDR boards were used. ZedBoard includes dual ARM Cortex-A9 hard processors on its PS side and MicroBlaze soft processor is utilized in the Nexys 4 DDR. Vivado offers Vitis tool for programming these processors. After building C based code on Vitis environment, an .elf file under debug file is created which includes assembly code waiting to be burning on instruction memory of hard or soft processor. Besides that, Vitis tool offers an optimization level which is detailed in the following chapters and thanks to this option it is able to speed up running duration of algorithm on processor.

High Level Synthesis (HLS) is a very advantageous tool itself. The primary reason is that the tool gives the opportunity to easily write very complex algorithms by synthesizing C/C++ functions into RTL. It also shows users the total latency of these

C/C++ based functions on FPGA card. In the analysis section, the tool allows users to easily inspect the code by pointing out where the code stalls too much. Moreover, by implementing various pragmas to the code it is possible to optimize and speed up the code drastically. As will be seen in the following chapters pipeline pragma is used in all hardware due to the fact that pipelining enables parallel execution in PL side of the FPGA.

2.2 FPGA Introductions Used in Project

Xilinx ZedBoard Zynq-7000 SoC Development Board is mainly used in this senior project. However, Artix-7 FPGA based Nexys 4 DDR board helped us to understand the concept of the RNMF algorithm in MicroBlaze soft processor. Due to the oscillator frequency in Nexys 4 DDR is 100 MHz in peripheral and processor itself some improvements are easily observed as will be seen in the following chapters contrary to ZedBoard where hard processor oscillator is about 666 MHz and improvements is defeated to this high frequency.

2.2.1 Xilinx ZedBoard Zynq-7000 SoC

RNMF algorithm previously performed on MATLAB in C language and after making sure that it is working as it should this C based algorithm is intended to move a mobile device. SoC devices are the best option for algorithms like RNMF where such a high number of iterations as 10000 and arithmetic operations. This is because SoC devices are integrated circuits which include both PS and PL sections within. The PS section includes a powerful CPU, memory interfaces, pins, analog/digital converters, and I/O peripherals such as UART, USB, ENET, SPI. PL sections contain LUTs, and Flip-Flops as every FPGA board contains. Hence the presence of these two sections on the SoC devices at the same time provides great opportunity for the user. Zynq-7000 SoC architecture can be seen in Figure 1.

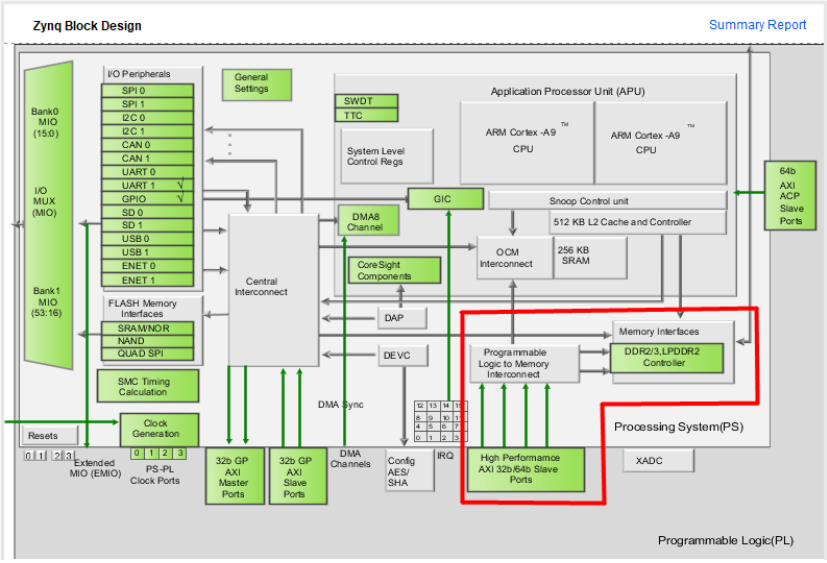


Figure 2.1: Zynq-7000 Architecture

As will be seen on the following chapters, algorithm can be optimized and accelerated by converting intensive arithmetic operation code snippets to hardware in the PL section and by keeping code snippets that contain lots of memory transition in the PS section. Additionally since the algorithm deals with 2 different array big arrays as X[46848] and target[46848] there is a need for a high capacity memory.

As a result of all these explanations, utilizing ZedBoard whose features are listed below, has great advantages in this senior project.

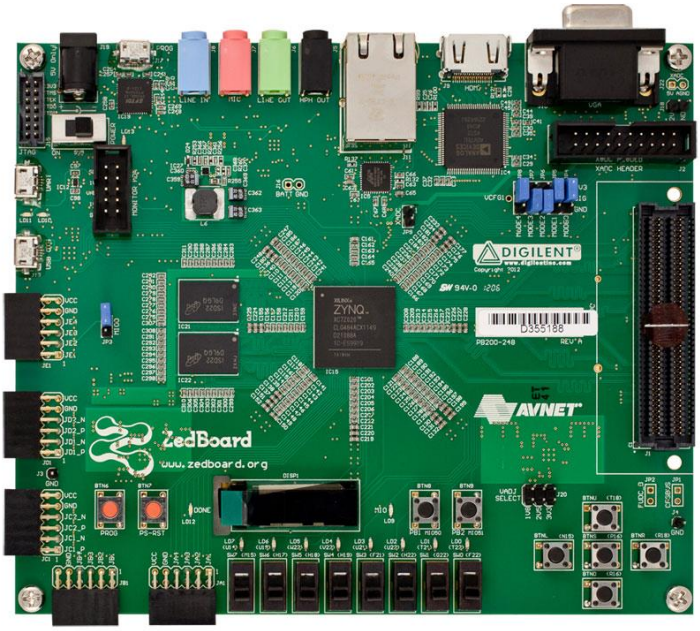


Figure 2.2: Xilinx ZedBoard Zynq-7000 SoC

• Dual-core ARM Cortex™-A9 processor

• DDR3 512 MB

• On-board USB-JTAG Programming

• USB OTG 2.0 and USB-UART

2.2.2 Nexys 4 DDR

Nexys 4 DDR is a pure programmable logic FPGA device including Artix-7. The biggest advantage of this FPGA board is that it contains a large DDR2 memory namely 128 MB. Since it is not contained in any hard processor some hardware utilization improvements will be more visible in this device as will be explained in target data utilization via DMA concept.

2.3 FPGA Configurations

FPGA boards can be configured in many different settings according to intended use likewise Zynq-7000 SoC devices use a multi-stage boot process that supports both non-secure and secure boot [1].

By default, ZedBoard uses SD Card configuration mode [1]. Configuration modes can be switched with MIO[8:2] boot mode pins according to the table below.

Table 2.1: ZedBoard Configuration Modes

	MIO[6]	MIO[5]	MIO[4]	MIO[3]	MIO[2]
Xilinx TRM→	Boot_Mode[4]	Boot_Mode[0]	Boot_Mode[2]	Boot_Mode[1]	Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

As can be seen from the table, the MIO[8:7] pins are used to set the I/O bank voltages, they are fixed and cannot be changed. JP8, JP9 and JP10 jumpers on the board are set to GND which designates JTAG mode as seen on the Figure 1 for the reason that many different designs and codes will be written and tested throughout the project.

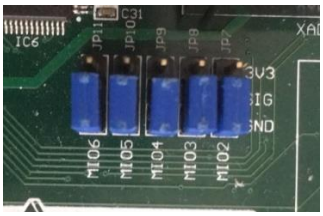


Figure 2.3: Jumper settings

3. DATA ACQUISITION

To bring in instant target detection feature to the project, communication between the radar machine and the FPGA had to be established. For that reason, the UART module, which is already available in the Zynq-7000 SoC, is used. Transmission speed is left as default 11520 bauds (bit/s) rate. In this section, two different methods where each uses different C libraries, are introduced.

3.1 Data Transfer by Using Standard Input Output Functions

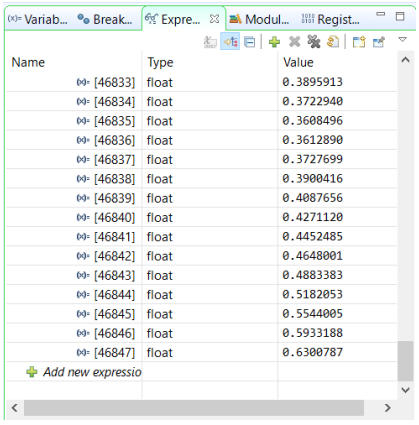
In order to obtain data from radar device dynamically, Following C code, which includes the simple standard input and output functions as printf() and scanf(), has been written.

```c
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include "xil_printf.h"

#define N 256*183 //46848
float X[N]; //giris verisi

int main()
{
    init_platform();
    xil_printf("Veri alimi basladi");   //bilgilendirme

    char str[12] = {};
    float val;
    for(int i=0 ; i<N ; i++){
        scanf("%s", str);
        val = strtof(str, NULL);
        X[i] = val;
    }
    xil_printf("Veri alimi sonlandi");  //bilgilendirme

    cleanup_platform();
    return 0;
}
```

Figure 3.1 : Simple data transferring C code.

As seen on the line 14, size of the input data is consisting of 12 digits, of which 1 is considered as decimal dot, 1 is considered as whole number and the rest is considered a decimal number. Moreover, it should be pay attention that data are saved as float data type after converting from char data type with the use of strtof() function for the reason that variables required for the RNMF algorithm to work must be float data type.

Throughout the project, X array denotes input data from radar device. Accuracy of the data received on the input X array was verified on the Vitis debug interface as seen Figure 3.2.



Name	Type	Value
[46833]	float	0.3895913
[46834]	float	0.3722940
[46835]	float	0.3608496
[46836]	float	0.3612890
[46837]	float	0.3727699
[46838]	float	0.3900416
[46839]	float	0.4087656
[46840]	float	0.4271120
[46841]	float	0.4452485
[46842]	float	0.4648001
[46843]	float	0.4883383
[46844]	float	0.5182053
[46845]	float	0.5544005
[46846]	float	0.5933188
[46847]	float	0.6300787
Add new expressio		

Figure 3.2 : Accuracy of the obtaining data

3.2 Data Transfer by Using UART Protocol

Since the RNMF algorithm requires a lot of data, and the FPGA card utilizing USB-to-UART Bridge is likely to fail in data acquisition while getting big data, it has been decided to use UART built-in functions in the project [2]. For this purpose, codes shown in below was written.

```
1   #ifndef UART_H
2   #define UART_H
3
4   /* Include Files */
5   #include <stdio.h>
6   #include "platform.h"
7   #include "xil_printf.h"
8   #include <stdlib.h>
9   #include "xuartps.h"
10  #include "xstatus.h"
11
12  #define STRSIZE 12
13  #define N 256*183 //46848
14  float X[N]; //giriş verisi
15
16  /* Function Declarations */
17  extern int uart(void);
18
19  #endif
20
```

Figure 3.3 : UART header file

```c
1   #include "platform.h"
2   #include "xil_printf.h"
3   #include "xuartps.h"
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include "xstatus.h"
7   #include "uart.h"
8
9   XUartPs_Config *Config;
10  XUartPs Uart_PS;
11
12  int uart(void)
13  {
14      xil_printf("Veri alimi basladi\n"); //bilgilendirme
15      /*****************************************************************************/
16          int status;
17          /*
18           * Initialize the UART driver so that it's ready to use.
19           * Look up the configuration in the config table, then initialize it.
20           */
21          Config = XUartPs_LookupConfig(XPAR_XUARTPS_0_DEVICE_ID);
22          if(NULL == Config){
23              return XST_FAILURE;
24          }
25          status = XUartPs_CfgInitialize(&Uart_PS, Config, Config->BaseAddress);
26          if(status!= XST_SUCCESS){
27              return XST_FAILURE;
28          }
29          /* Check hardware build. */
30          status = XUartPs_SelfTest(&Uart_PS);
31          if (status != XST_SUCCESS) {
32              return XST_FAILURE;
33          }
34
35          XUartPs_SetBaudRate(&Uart_PS, 115200);
36      /*****************************************************************************/
37          char tempStr[STRSIZE];
38          u8 tempChar;
39          int i = 0;
40
41          while(i != N){
42              tempChar = XUartPs_RecvByte(XPAR_PS7_UART_1_BASEADDR);
43              for (int j = 0; j < STRSIZE; j++) {
44                  tempStr[j] = tempChar;
45                  tempChar = XUartPs_RecvByte(XPAR_PS7_UART_1_BASEADDR);
46              }
47              X[i] = strtof(tempStr, NULL);
48              while (tempChar != '\n') {
49                  tempChar = XUartPs_RecvByte(XPAR_PS7_UART_1_BASEADDR);
50              }
51              i++;
52          }
53          xil_printf("Veri alimi sonlandi\n");    //bilgilendirme
54          return XST_SUCCESS;
55  }
```

Figure 3.4 : UART C file

On the lines between the 15th and 36th, the UART driver on the Zynq has been initialized and self-tested to ensure that it is working correctly. Lastly, the baud rate between the host computer and the FPGA is left as 115200 bauds.

XUartPs_RecvByte() functions that return 8-bit unsigned integer namely, u8 are used for every input byte [3]. These bytes are recorded on tempStr array in every STRSIZE loop in between the lines 43 and 46, and this loop size is selected as 12 for the same reason as described in the section 3.1. Ability to change STRSIZE value gives algorithm freedom to choose any desired input size.

As it seen on the line 47, strtof() function is used again since the RNMF algorithm works float data type. Finally, while loop in between 48-50 lines was written to repeat all these actions in each new received input row. Validation of the code was realized by writing a simple code as Figure 3.5.

```
13    int status;
14    status = uart();
15    if (status != XST_SUCCESS) {
16        xil_printf("Veri alimi hatali\r\n");
17    }
18    for(int i=0; i<N ; i++)
19    {
20        printf("%f ", X[i]);
21        printf("%d ", i);
22    }
```

Figure 3.5: UART validation code

Terminal screen of the code output is shown in Figure 3.6 and as seen on the last row; all 46848-input data is kept under X array.

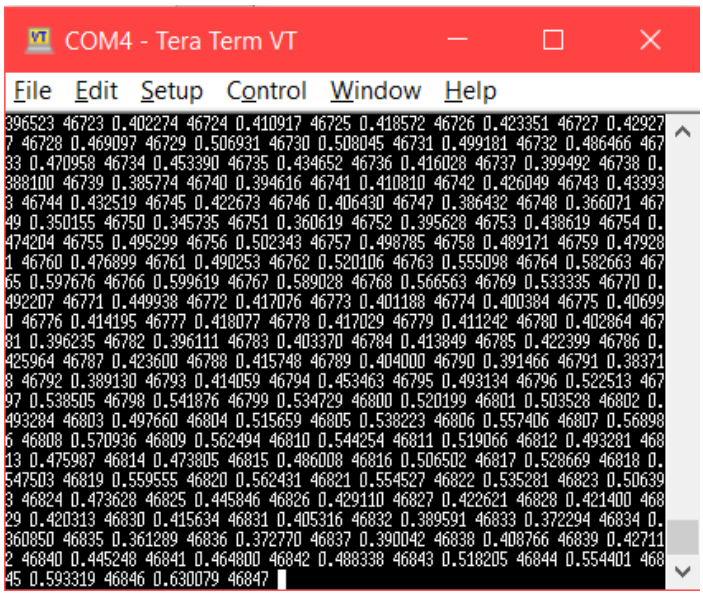


Figure 3.6: TeraTerm output screen

4. SERIAL INPUT TEXT DATA SIZE ADJUSTMENT

It has been observed that the matrix sizes of Ground Penetrating Radar (GPR) images from radar varies. However, RNMF C code is written in a format where input and output data image sizes are standardized as 256 rows and 183 columns. Hence, there is a need a standardization for the data coming from radar device as well. To this respect, in order for these matrices to work in the RNMF algorithm, they must be

converted an appropriate [256,183] form. This process was done by using MATLAB tool.

The MATLAB code that will convert each incoming data into the appropriate form is shown in Figure 4.1.

```
data = importdata('normalized_GPR_data.txt');
normalized = imresize(data, [256 183]);
```

data	4001x17 double
normalized	256x183 double

Figure 4.1: Reshaping incoming data to appropriate format

With the help of this code, the GPR picture of the incoming data can be converted to intended 256 x 183 matrix size as the following Figure 4.2.

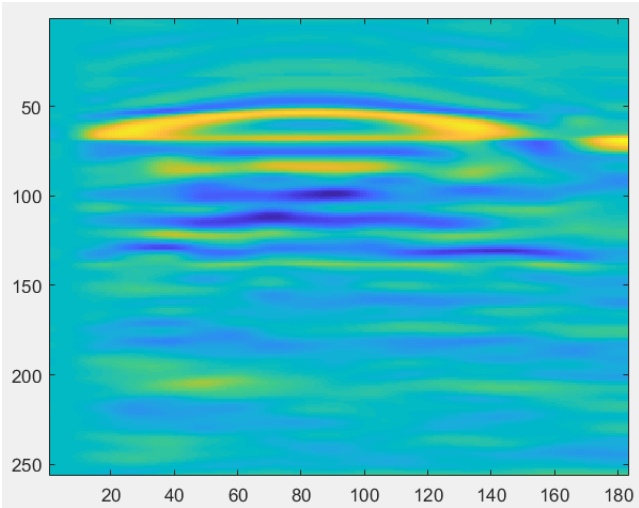


Figure 4.2: GPR image of sample incoming data

FPGA development cards use UART serial communication protocol. Hence, proposed standardization is in need of a .txt file in which, the incoming data is sorted serially from top to bottom in a single column. The code realizing this operation is shown in Figure 4.3.

```
data = importdata('normalized_GPR_data.txt');
normalized = imresize(data, [256 183]);
normalized_tek_sutun = normalized(:);
dlmwrite('normalized_GPR_data_tek_sutun.txt',normalized_tek_sutun, 'precision', '%.10f');
```

data	4001x17 double
normalized	256x183 double
normalized_tek_sutun	46848x1 double

Figure 4.3: Serial input text data adjustment

Sample radar data can be obtained in a .txt file as shown in Figure 4.4.

```
0.4931376037
0.4921420249
0.4911084520
0.4911223263
0.4920091157
0.4927064185
0.4930951403
0.4932757978
0.4933390536
```

Figure 4.4: Sample radar data in .txt file

It is important to note that, as described in 3.2 UART section, value of the input data size namely, STRSIZE can be set as seen on the line 4 in Figure 4.3. It is set as 10 precision and throughout the project value was not changed.

5. RNMF APPLICATIONS

Since the data acquisition work is accomplished, in this section Robust Nonnegative Matrix Factorization (RNMF) algorithm applications is introduced on FPGA development cards.

Firstly, 64-bit static algorithm was run on FPGA card and its running time were observed. Then the algorithm was converted to 32 bits by changing 64-bit double data type into 32-bit float data type. Programmable logic (PL) part of the ZedBoard is not required to run the RNMF algorithm, the reason for this the ARM Cortex-A9 hard processor and PS UART on the Processing system (PS) are sufficient for the operation. Therefore, no block design was made on VIVADO and no bitstream file was created.

5.1 64-Bit Static RNMF Algorithm

After the setup of the algorithm is done on VITIS, the math library required for the RNMF algorithm to operate is included in the project from the Properties section as shown in the Figure 5.1 below.

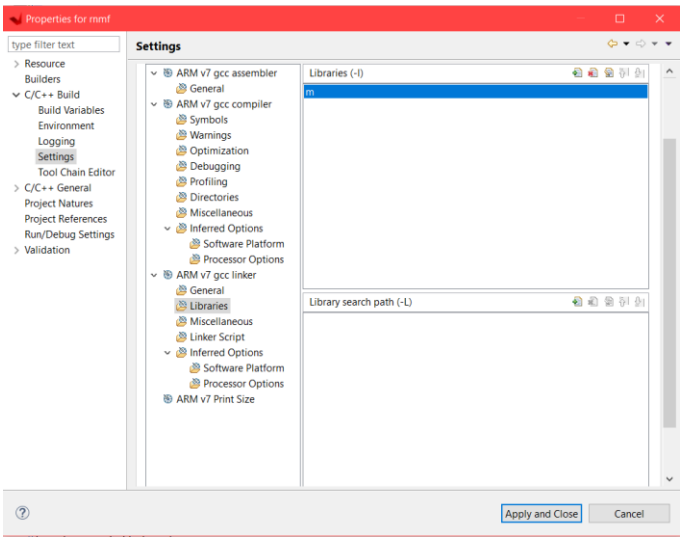


Figure 5.1: Math library inclusion to the project

After building the project and making connection of the FPGA with the PC. TeraTerm tool was used to observe the target and clutter data which are the RNMF algorithm output products. TeraTerm screen on Figure 5.2 shows that RNMF algorithm operates on the Zynq-7000 SoC as it should. Log output is also saved as in the Figure 5.3 to have knowledge of the operation duration and to create GPR images of target and clutter data.

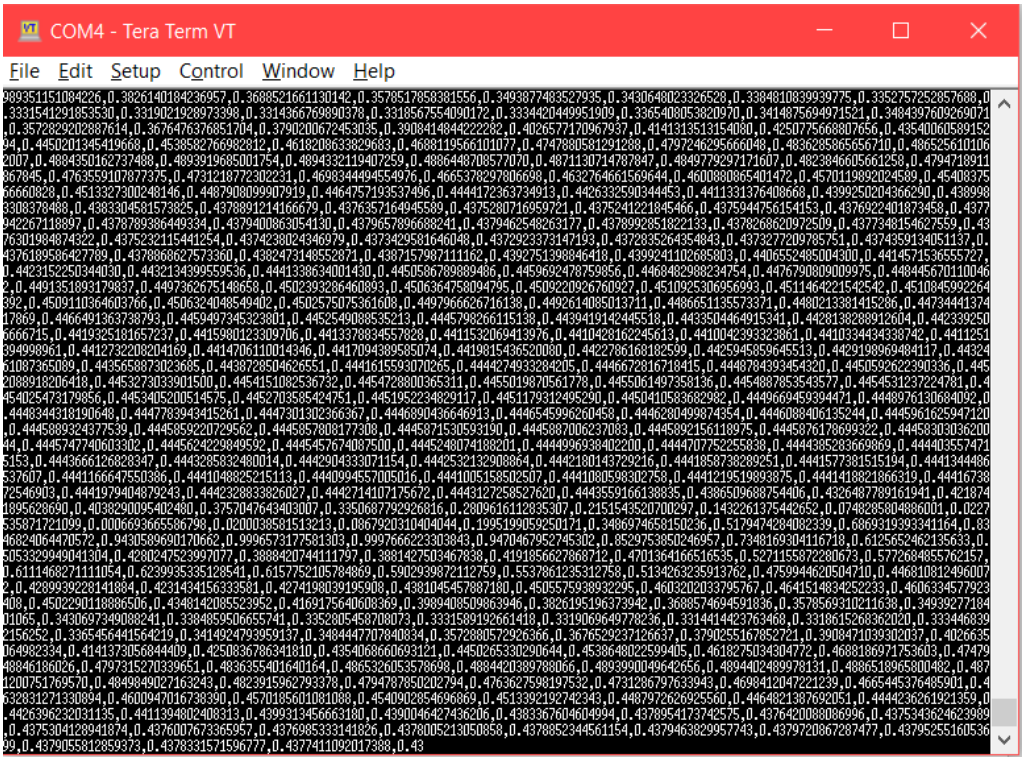


Figure 5.2: 64-bit RNMF algorithm TeraTerm screen

```
teraterm - Not Defteri
Dosya  Düzen  Biçim  Görünüm  Yardım
Output took 002_677_462_932 clock cycles.
Output took 04'34''.
TARGET DATA
0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.00000
0,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,
0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000
000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,-0.00000000000000
00000000000000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,-0.0000
0.0000000000000000,-0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000
00000,-0.0000000000000000,-0.0000000000000000,0.0000000000000000,0.0000000000000000,0.00000000000000
000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.00000000000
000000,-0.0000000000000000,-0.0000000000000000,-0.0000000000000000,0.0000000000000000,0.00000000000000
000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000
,0.0000566483687996,0.0000763878175118,0.0000777221681389,0.0000499064306396,0.0000000000000000,0.0000
0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.0000000000000000,0.00000
St 1, Stn 1   100%   Windows (CRLF)   UTF-8
```

Figure 5.3: 64-bit RNMF log output

Operation duration of the RNMF algorithm was measured as 4 minutes and 34 seconds.

The accuracy of the operation can be checked by drawing the GPR pictures on MATLAB both for the target and the clutter data separately. This code is seen Figure 5.4 below.

```
data = importdata('Raw_GPR_data.txt');
raw_data = imresize(data, [256 183]);
subplot(1,3,1)
imagesc(raw_data);
title('Raw GPR Image Collected from Target Object')
target_data = importdata('target.txt');
target_matrix = reshape(target_data, 256, 183);
clutter_data = importdata('clutter.txt');
clutter_matrix = reshape(clutter_data, 256, 183);
subplot(1,3,2)
imagesc(target_matrix);
title('Target Object Removed Clutter Image')
subplot(1,3,3)
imagesc(clutter_matrix);
title('Clutter (Object Removed) Image')
```

Figure 5.4: MATLAB code for drawing GPR image of data

As seen on the Figure 5.5 64-bit RNMF algorithm works as it should. However, 4’34’’ is quite long duration for a quick target detection operation and needs to be reduced.

Figure 5.5: 64-bit RNMF algorithm GPR images

5.2 32-Bit Static RNMF Algorithm

To speed up the operation duration algorithm is converted to 32-bit by converting all 64-bit double data type to 32-bit float data type.

As it seen from log by dividing each register data width into two, it is possible to decrease operation duration about 42 seconds.

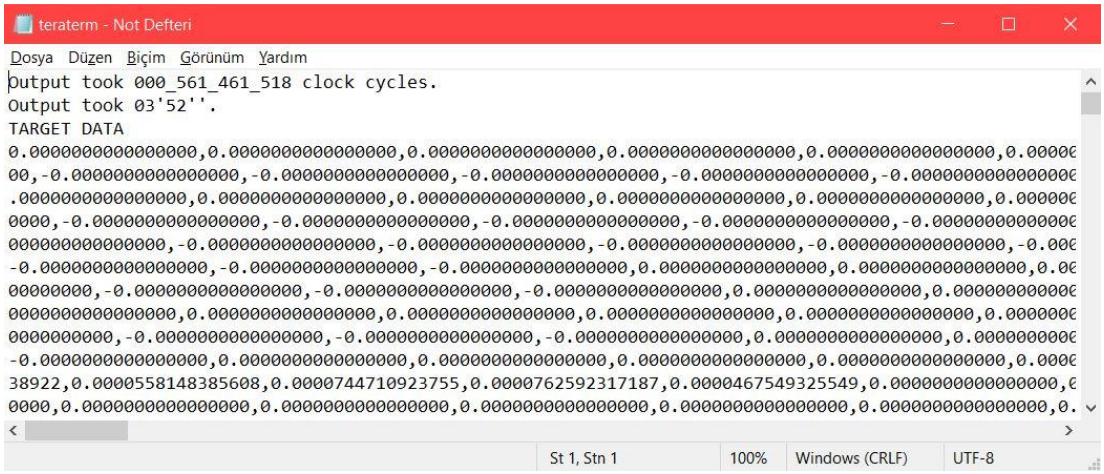


Figure 5.6: 32-bit RNMF log

As it seen on Figure 5.7 GPR images of output target and clutter data, it is possible to say decreasing data width does not impact RNMF operation drastically.

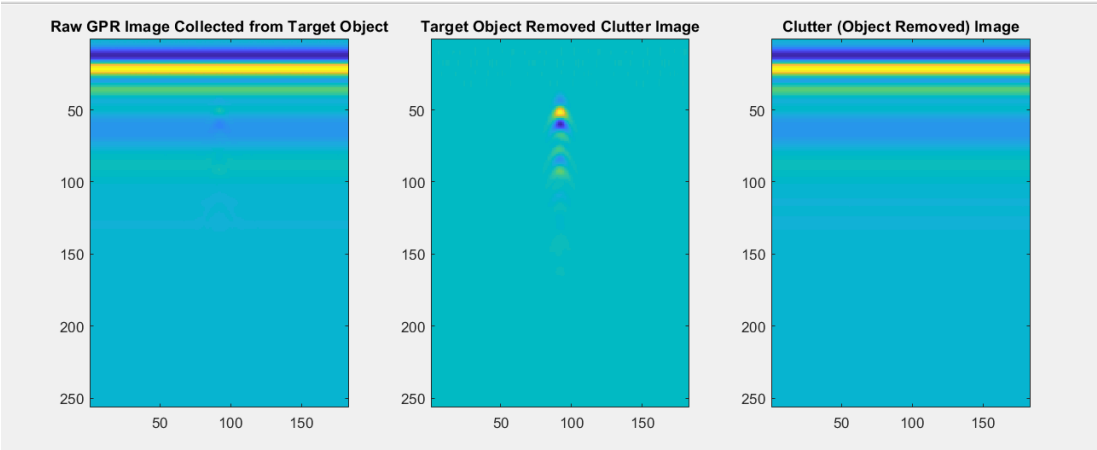


Figure 5.7: 32-bit RNMF algorithm GPR images

5.3 32-Bit Dynamic RNMF Algorithm

The “uart.h”and “uart.c” files which are introduced in data acquisition section, are added to the project to be able to bring in instant target detection feature to the project for as shown in the Figure 5.8.

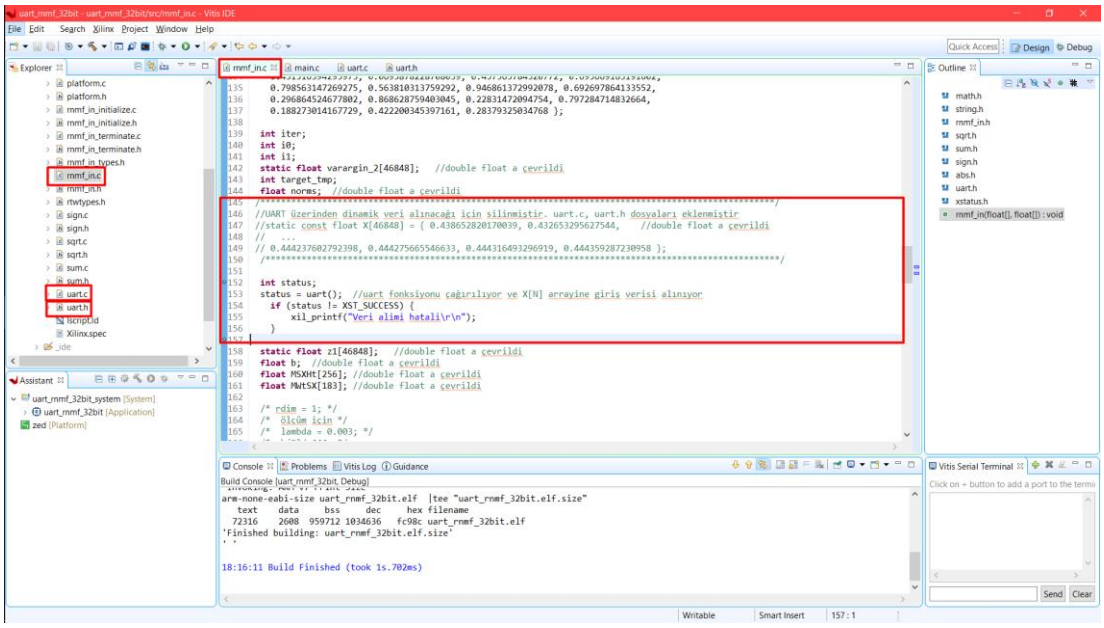


Figure 5.8: 32-bit dynamic RNMF algorithm

TeraTerm screen on Figure 5.9 shows that communication between PC and FPGA board is healthy.

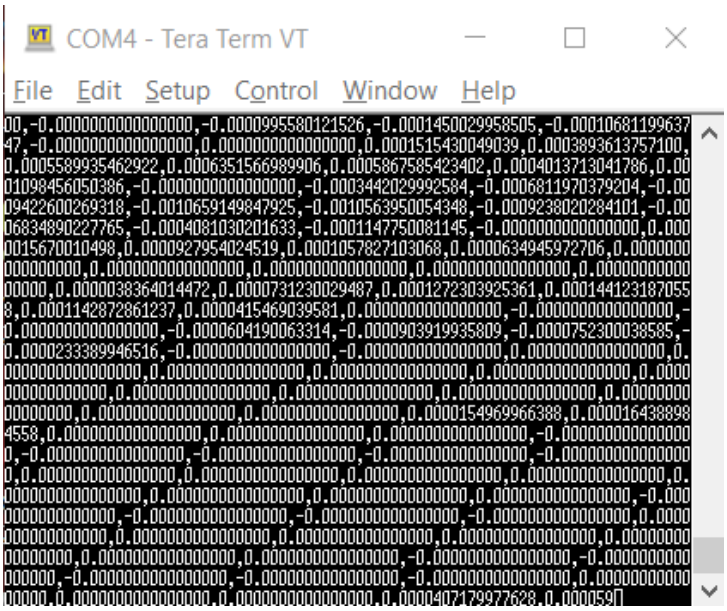


Figure 5.9: 32-bit dynamic RNMF terminal screen

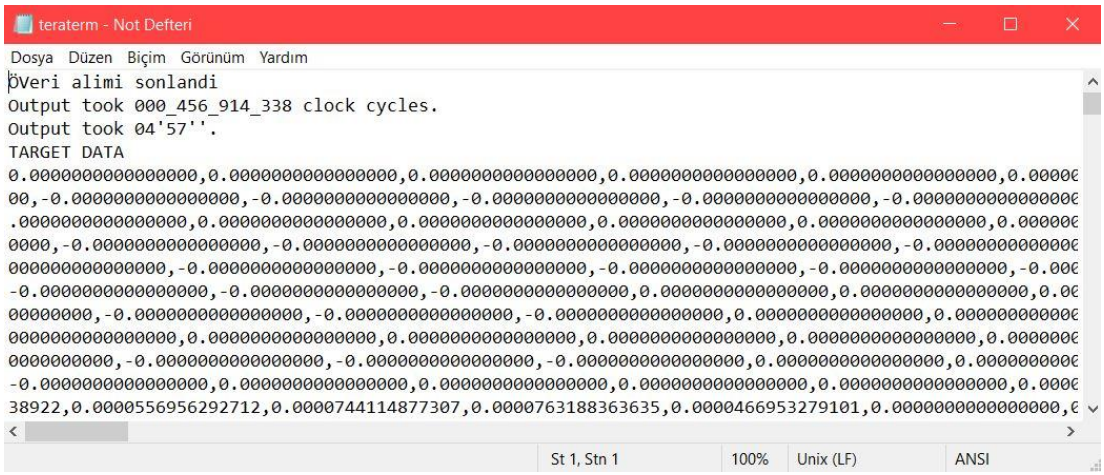


Figure 5.10: 32-bit dynamic rnmf log output

As it seen on Figure 5.10 operation duration of dynamic algorithm is observed longer than static algorithm. The reason for this increase is the data acquisition from radar device. This duration can be decreased by increasing bauds rate on the communication line. By looking at the Figure 5.11, it can be validated that the UART driver that created works as it should.

Figure 5.11: 32-bit dynamic RNMF algorithm GPR images

6. HARDWARE IMPLEMENTATION OF RNMF ALGORITHM

6.1 Advance eXtensible Interface Communication Bus Protocol

Advanced eXtensible Interface Bus Protocol (AXI) was first released in 1996 by ARM as part of AMBA (Advanced Microcontroller Bus Architecture). Shortly after its appearance it became standard not just for microcontrollers but also for SoC and ASIC parts. Now AXI standard basically defines how functional blocks or IPs communicate with each other. Xilinx also adopted AXI protocol for their IP cores for the reason that it provides so many improved features such as high throughput and performance as well as allowing burst transactions. In 2011, ARM released the last version of their bus protocol which is called AXI4 (AMBA 4.0). Zynq device that we use in this senior project also uses AXI interfaces to PS and PL communicate with each other.

There are two types of AXI interfaces:

- Stream
  - AXI4-Stream: For high-speed streaming data.
- Memory Mapped
  - AXI4(Full AXI4): For high-performance memory-mapped requirements.
  - AXI4-Lite: For simple, low-throughput memory-mapped communication.

Every AXI interface contains two essential IPs called Master and Slave as seen in Figure 1 and as seen in this Figure 1 there are five independent channels between Master and Slave.

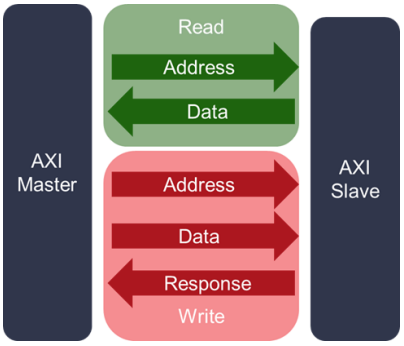


Figure 6.1: AXI Master and Slave IPs

AXI Master is responsible for initiating write and read transactions and AXI Slave is responsible for responding to these transactions initiated by the AXI Master.

6.1.1 AXI Memory Mapped Interface

Memory mapped denoting an address which is specified by the master IP within the transaction. It is possible write to or read from Slave IP in this type of interface[4].

For the AXI4-Lite only a single beat data transfer to a specified address per transaction is possible.That is why it has a very poor performance and is not preferred for advanced applications.

However, resulting from the fact that Full-AXI4 allows to sending up continuous beats up to 256 data in other words burst, Full-AXI4 offers better performance.

6.1.1.1 AXI Memory Mapped Interface Channel Signals

Inside of each 5 independent channels there are 3 main signals called ready, valid and data which build the channel. These channel signals can be seen in Figure 2. Channels can contain additional signals according to the user's request. These additional signals can be exemplified as len, size, id and cache. It is important to note that all these main signals are synchronous to the rising edge of the clock.

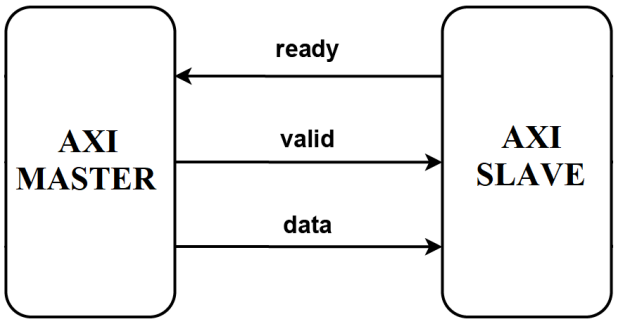


Figure 6.2: AXI Memory Mapped Interface Channel Signals

- Ready: This flag notifies Master IP that Slave IP is ready to receive data.
- Valid: This flag notifies Slave IP that Master IP transferring valid data.
- Data: As the name indicates, it contains the data, and its size can be varied according to specification.
- Len: This signal, which is being sent form Master IP to Slave IP, indicates the length of the burst operation.
- Size: This signal notifies Slave IP how long the transferred widths for each beat of data will be.
- Id: This signal is used to distinguish the transactions from each other by assigning different ids to Master IPs. As will be seen on the future project designs AXI Interconnect communicate with different IPs by using this signal.
- Cache: By setting cache capability, Master IP can respond to transaction in a faster time with the data it has already on its own buffer or cache. It is used to speed up the transactions in the AXI Interconnect.

6.1.2 AXI Stream Interface

This interface is used in a block where a dedicated process is perpetually performed on the data and sends out this output data to its slave. In this interface, Master IP does not need to provide an address to the Slave IP for transferring data. Furthermore, direction of the data always from Master IP to Slave IP. Basically, in AXIS interface Master IP is always and only writing to its Slave IP and no address for this data transfer is required[5].

6.1.2.1 AXI Stream Interface Channel Signals

In this One Write Channel, there are 1 more signal named last is contained as well as 3 main signals. This last flag is responsible for indicating last bit on the input data stream. This flag are shown is Figure 3.

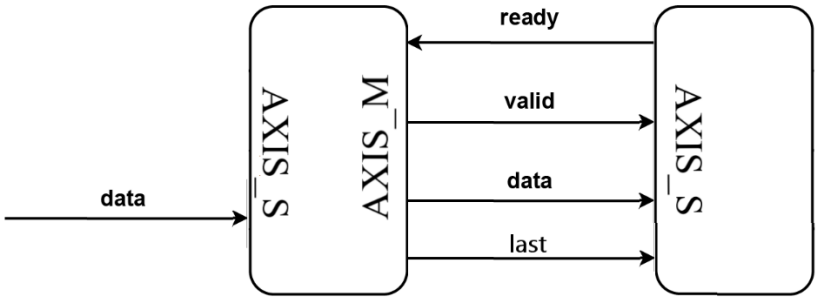


Figure 6.3: AXI Stream Interface Channel Signals

- Valid: This flag notifies Slave IP that Master IP is whether transferring last data in the stream or not.

6.2 Updating Custom IP Writing Handshake Protocol

As can be seen on line 11930 of the "rnmf_in.c" file, nested for loop was used to obtain the target data on the algorithm. Since there are too many iterations; algorithm spends lots of time in this nested loop.

```
11926  for (iter = 0; iter < 10000; iter++) {
11927    for (i0 = 0; i0 < 256; i0++) {
11928      for (i1 = 0; i1 < 183; i1++) {
11929        target_tmp = i0 + (i1 << 8);
11930        target[target_tmp] = X[target_tmp] - W[i0] * H[i1];
11931      }
11932    }
```

Figure 6.4: Time wasting 3 nested loops

By utilizing PL section of the FPGA, operation duration can be speed up for the reason that the FPGA has so many DSPs in it. The steps shown for the custom IP design have been followed.

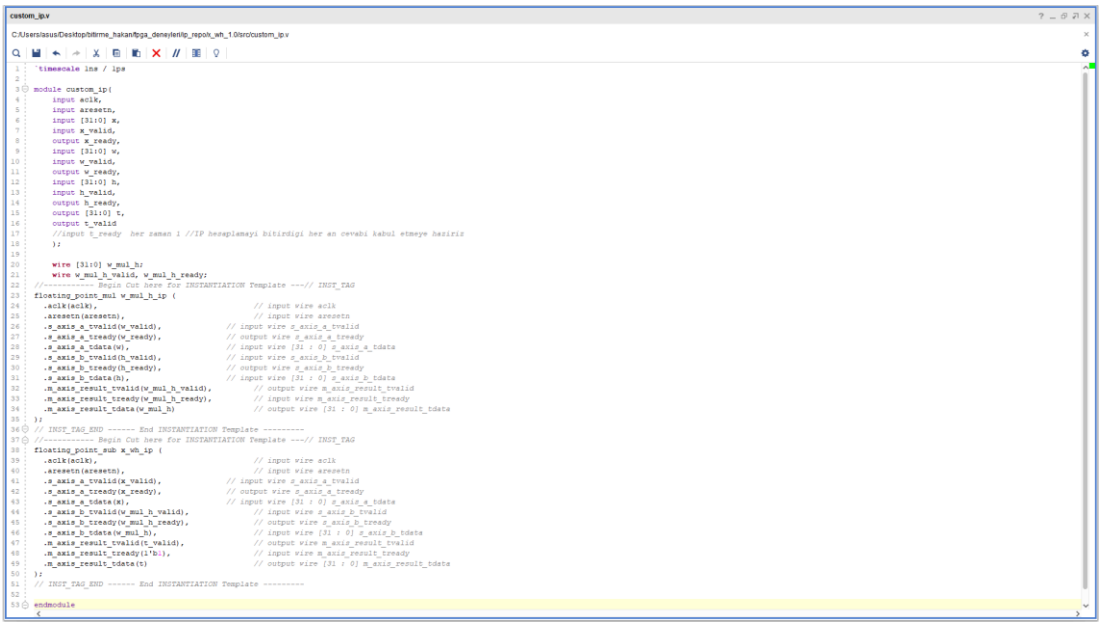
This IP contains 5 registers and has a Lite interface. This IP considered Slave in respect to the ARM CORTEX-A9 processor in the PS section.

Separate floating-point IPs have been added to the project for multiplying and subtracting. Since algorithm will be working on 32-bit float variables, single floating-

point precision is chosen as precision of inputs. Then, active low reset pin is added. Latency has not been changed to not violate positive WNS value.

Then, a new Verilog file is created from Add Sources ⇨ Add or create design sources. Multiplication and subtraction blocks have been added to the newly created Verilog file by instantiating from the floating_point_0.veo file. For this, the following sequence is followed. IP Sources → floating_point_0 → Instantiation Template.

The created Verilog file can be seen on Figure 6.5 below.



custom_ip.v

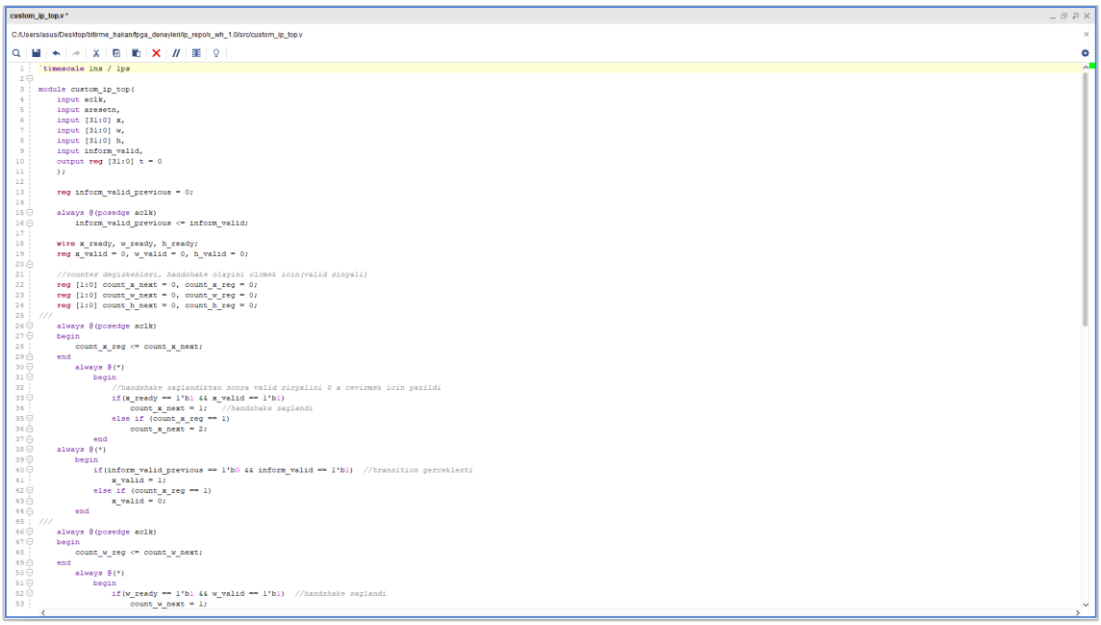
C:/Users/asus/Desktop/bitirme_hakan/fpga_deneylerI/ip_repo/k_wh_1.0/src/custom_ip.v

```
1   `timescale 1ns / 1ps
2
3   module custom_ip(
4       input aclk,
5       input aresetn,
6       input [31:0] x,
7       input x_valid,
8       output x_ready,
9       input [31:0] w,
10      input w_valid,
11      output w_ready,
12      input [31:0] h,
13      input h_valid,
14      output h_ready,
15      output [31:0] t,
16      output t_valid
17      //input t_ready  her zaman 1 //IP hesaplamayı bitirdiği her an cevabı kabul etmeye hazırız
18      );
19
20      wire [31:0] w_mul_h;
21      wire w_mul_h_valid, w_mul_h_ready;
22  //----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
23  floating_point_mul w_mul_h_ip (
24    .aclk(aclk),                                  // input wire aclk
25    .aresetn(aresetn),                            // input wire aresetn
26    .s_axis_a_tvalid(w_valid),              // input wire s_axis_a_tvalid
27    .s_axis_a_tready(w_ready),              // output wire s_axis_a_tready
28    .s_axis_a_tdata(w),                     // input wire [31 : 0] s_axis_a_tdata
29    .s_axis_b_tvalid(h_valid),              // input wire s_axis_b_tvalid
30    .s_axis_b_tready(h_ready),              // output wire s_axis_b_tready
31    .s_axis_b_tdata(h),                     // input wire [31 : 0] s_axis_b_tdata
32    .m_axis_result_tvalid(w_mul_h_valid),         // output wire m_axis_result_tvalid
33    .m_axis_result_tready(w_mul_h_ready),         // input wire m_axis_result_tready
34    .m_axis_result_tdata(w_mul_h)                 // output wire [31 : 0] m_axis_result_tdata
35  );
36  // INST_TAG_END ------ End INSTANTIATION Template ---------
37  //----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
38  floating_point_sub x_wh_ip (
39    .aclk(aclk),                                  // input wire aclk
40    .aresetn(aresetn),                            // input wire aresetn
41    .s_axis_a_tvalid(x_valid),              // input wire s_axis_a_tvalid
42    .s_axis_a_tready(x_ready),              // output wire s_axis_a_tready
43    .s_axis_a_tdata(x),                     // input wire [31 : 0] s_axis_a_tdata
44    .s_axis_b_tvalid(w_mul_h_valid),              // input wire s_axis_b_tvalid
45    .s_axis_b_tready(w_mul_h_ready),              // output wire s_axis_b_tready
46    .s_axis_b_tdata(w_mul_h),                     // input wire [31 : 0] s_axis_b_tdata
47    .m_axis_result_tvalid(t_valid),               // output wire m_axis_result_tvalid
48    .m_axis_result_tready(1'b1),                  // input wire m_axis_result_tready
49    .m_axis_result_tdata(t)                       // output wire [31 : 0] m_axis_result_tdata
50  );
51  // INST_TAG_END ------ End INSTANTIATION Template ---------
52
53  endmodule
```

Figure 6.5: custom_ip.v file

X, W, H are specified as input and T as output. Valid and ready signals are also defined for both inputs and output variables. Since floating_point_0_mul is a multiplication block, variable W is assigned to data A, variable H to data B and variable x_mul_h assigned to result data. Then, variable X is assigned to data A of floating_point_0_sub block, variable x_mul_h is assigned to data B, and variable T to result data. t_ready signal value is defined always 1 due to the fact that PS side is always ready to accept the answer whenever the IP finishes the computation.

Thereafter, custom_ip_top.v file created for the required handshake protocol between the IP and ARM CORTEX-A9 processor. Code is shown in Figure 6.6 below.



```
custom_ip_top.v *
C:/Users/asus/Desktop/bitirme_hakan/fpga_deneylerli/ip_repo/x_wh_1.0/src/custom_ip_top.v

1   `timescale 1ns / 1ps
2
3   module custom_ip_top(
4       input aclk,
5       input aresetn,
6       input [31:0] x,
7       input [31:0] w,
8       input [31:0] h,
9       input inform_valid,
10      output reg [31:0] t = 0
11      );
12
13      reg inform_valid_previous = 0;
14
15      always @(posedge aclk)
16          inform_valid_previous <= inform_valid;
17
18      wire x_ready, w_ready, h_ready;
19      reg x_valid = 0, w_valid = 0, h_valid = 0;
20
21      //counter degiskenleri, handshake olayini olmek icin(valid sinyali)
22      reg [1:0] count_x_next = 0, count_x_reg = 0;
23      reg [1:0] count_w_next = 0, count_w_reg = 0;
24      reg [1:0] count_h_next = 0, count_h_reg = 0;
25  ///
26      always @(posedge aclk)
27      begin
28          count_x_reg <= count_x_next;
29      end
30          always @(*)
31              begin
32                  //handshake saglandiktan sonra valid sinyalini 0 a cevirmek icin yazildi
33                  if(x_ready == 1'b1 && x_valid == 1'b1)
34                      count_x_next = 1;   //handshake saglandi
35                  else if (count_x_reg == 1)
36                      count_x_next = 2;
37              end
38      always @(*)
39          begin
40              if(inform_valid_previous == 1'b0 && inform_valid == 1'b1)  //transition gerceklesti
41                  x_valid = 1;
42              else if (count_x_reg == 1)
43                  x_valid = 0;
44          end
45  ///
46      always @(posedge aclk)
47      begin
48          count_w_reg <= count_w_next;
49      end
50          always @(*)
51              begin
52                  if(w_ready == 1'b1 && w_valid == 1'b1)  //handshake saglandi
53                      count_w_next = 1;
```

Figure 6.6: custom_ip_top.v file

As it seen on the always block in below after transition on signal inform_valid occurred, that means that block is ready to receive new X, W, H, input signals.

```
38      always @(*)
39          begin
40              if(inform_valid_previous == 1'b0 && inform_valid == 1'b1)  //transition gerceklesti
41                  x_valid = 1;
42              else if (count_x_reg == 1)
43                  x_valid = 0;
44          end
```

Figure 6.7: always block in custom_ip_top.v

On another block as it seen in below after ready and valid signals are both equals to 1 that means that handshake is done, and we can process and receive another inputs to the newly created custom IP to calculate the equation T = X – W*H.

```
30          always @(*)
31              begin
32                  //handshake saglandiktan sonra valid sinyalini 0 a cevirmek icin yazildi
33                  if(x_ready == 1'b1 && x_valid == 1'b1)
34                      count_x_next = 1;   //handshake saglandi
35                  else if (count_x_reg == 1)
36                      count_x_next = 2;
37              end
```

Figure 6.8: always block in custom_ip_top.v

These always blocks are needed for every input namely, X, W and H.

Lastly, a testbench file is written as seen on Figure 6.9 to test the accuracy of the handshake top module.

```
`timescale ins / ips

module testbench();

    reg aclk = 0;
    always #5 aclk =~ aclk;

    reg inform_valid = 0, aresetn = 0;
    reg [31:0] x = 0, w = 0, h = 0;
    wire [31:0] t;

    initial
        begin
            #100 aresetn = 1;

            #20 x = 32'h3ee0971a;
                w = 32'h3e4ab8de;
                h = 32'h3e4ab8de;
                inform_valid = 1;

            #20 inform_valid = 0;

            #300 $stop;
        end

        custom_ip_top m0(
                aclk,
                aresetn,
                x,
                w,
                h,
                inform_valid,
                t
            );
endmodule
```

Figure 6.9: Custom IP testbench file

Top module is verified on the simulation screen on Figure 6.10. This result shows that the handshake mechanism works on the custom IP as it should and this IP can be utilized in the RNMF algorithm.

Figure 6.10: Simulation output

The variable T, which holds the result of all these multiplication and subtraction operations, is assigned to the 4th register of the custom IP block, and the inform_valid variable required for the custom IP to work is assigned to the 3rd register. The zeroth register of the block is assigned to the X variable, the 1st register to the W variable, and the 2nd register to the H variable. These register assignments are important for data exchange between PS and PL components.

All these are shown in the VIVADO interface as in the Figures 6.11 and 6.12 below.

```
    // Add user logic here
    wire [31:0] X, W, H;
    wire inform_valid;

    assign X = slv_reg0;
    assign W = slv_reg1;
    assign H = slv_reg2;
    assign inform_valid = slv_reg3;

    custom_ip_top c_ip(
        .aclk(S_AXI_ACLK),
        .aresetn(S_AXI_ARESETN),
        .x(X),
        .w(W),
        .h(H),
        .inform_valid(inform_valid), //IP çalismasi icin bilgilendiriyor
        .t(T)
        );
    // User logic ends
```

Figure 6.11: IP internal register assignments

```
374 /*****************************************************************************/
375 wire [C_S_AXI_DATA_WIDTH-1:0] T;
376 /*****************************************************************************/
377     // Implement memory mapped register select and read logic generation
378     // Slave register read enable is asserted when valid address is available
379     // and the slave is ready to accept the read address.
380     assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
381     always @(*)
382     begin
383           // Address decoding for reading registers
384           case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
385             3'h0   : reg_data_out <= slv_reg0;
386             3'h1   : reg_data_out <= slv_reg1;
387             3'h2   : reg_data_out <= slv_reg2;
388             3'h3   : reg_data_out <= slv_reg3;
389             3'h4   : reg_data_out <= T;
390             default : reg_data_out <= 0;
391           endcase
392     end
```

Figure 6.12: IP internal register assignments continious

Finally, on component.xml tab merge changes are done. Thus, IP is made ready by clicking Re-Package IP from the Review and Package section.

After all of these IP creation process, it is time to implement this IP in the RNMF algorithm. For this project, both the ARM processor on the PS part and the programmable logics on the PL part of the ZedBoard will be needed. For that matter it is needed to create .xsa file on VIVADO.

The block design of the hardware that will be uploaded to ZedBoard has been created as it appears on the Figure 6.12.

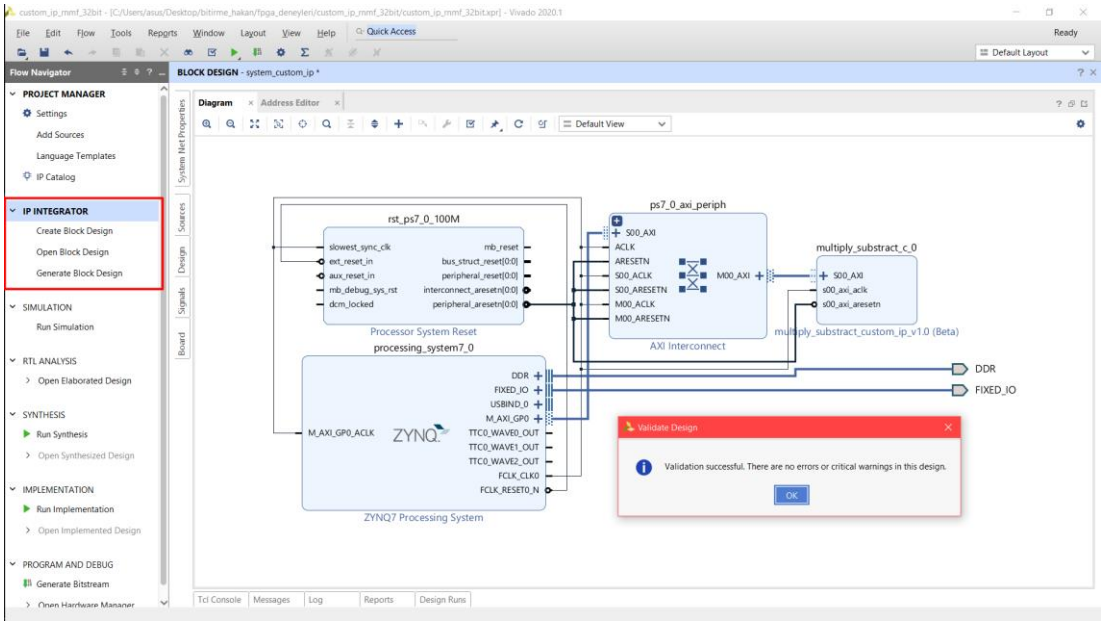


Figure 6.13: Block design of the hardware

After creating the .xsa file, it is imported to the VITIS tool. That file is selected for creating the platform which the application will run inside.

“custom_ip.h”and “custom_ip.c” files were created as in the Figure 6.14 and 6.15 in order to access the registers of the custom IP created before.

```
1 #include "custom_ip.h"
```

Figure 6.14: custom_ip.c file

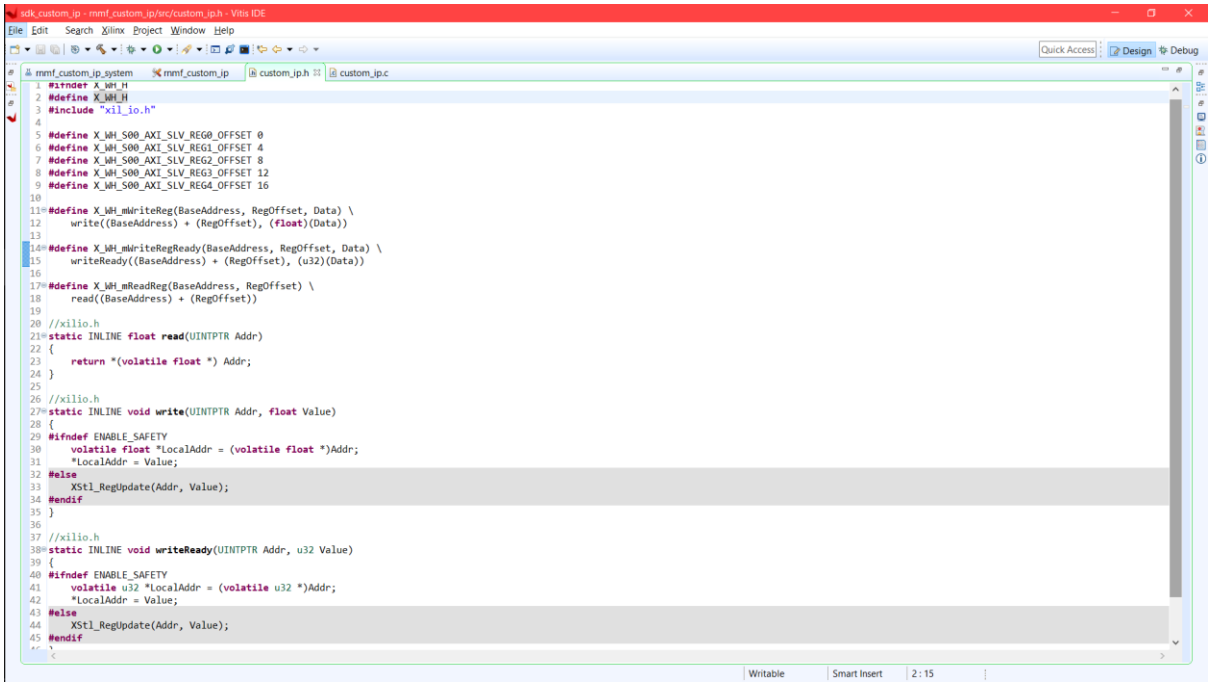


Figure 6.15: custom_ip.h file

Updated “rnmf_in.c” file can be seen on Figure 6.16. Since the IP is triggered with the transition of the inform_valid signal. This signal value changed before reading T value every time from 0 to 1 in order to read result value from IP register 4 which represents the T signal.

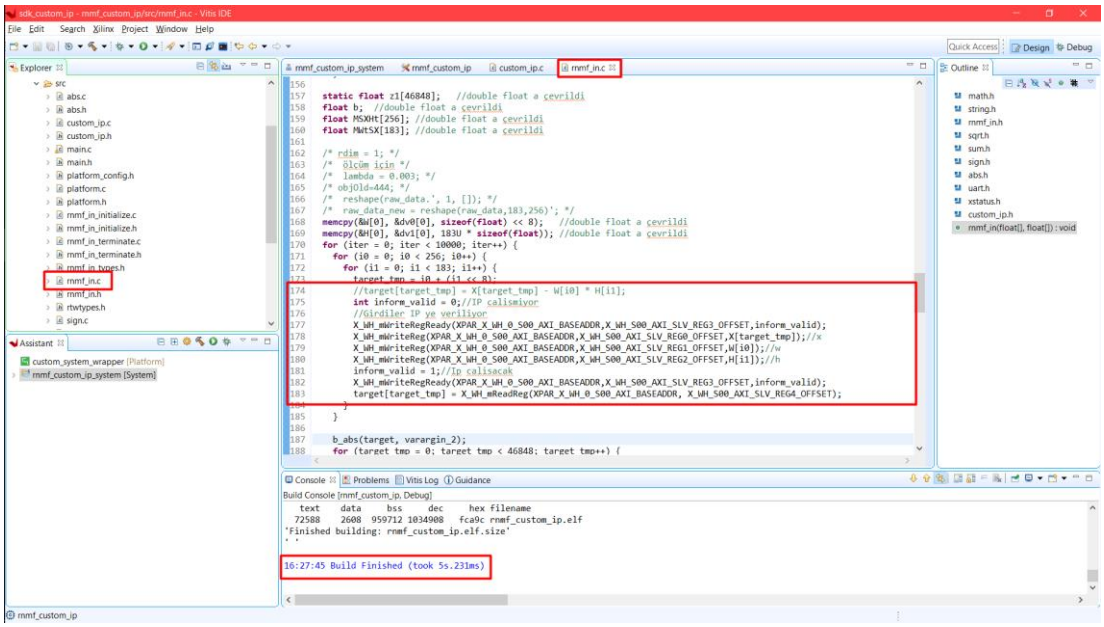


Figure 6.16: updated rnmf_in.c file

After building the project and making connection of the FPGA with the PC. Log operation is made on the TeraTerm to observe and draw the output GPR images of the algorithm. The output of target and clutter data is seen on the terminal screen as shown in the Figure 6.17.

Figure 6.17: Terminal screen

The running time of the algorithm was measured as 21 minutes. As it seen the algorithm took so long. This can be explained as follows: As a result of adding custom IP, number of operations which were one beforehand are increased to 6 and also because the AXI communication requires 4 clock cycles to read and write the registers. For this reason, the algorithm time is extended. However, since the FPGA can implement functions parallel. By using parallel IPs this failure can be overcome.

As can be seen from the MATLAB outputs on Figure 6.18, the custom IP is working correctly.

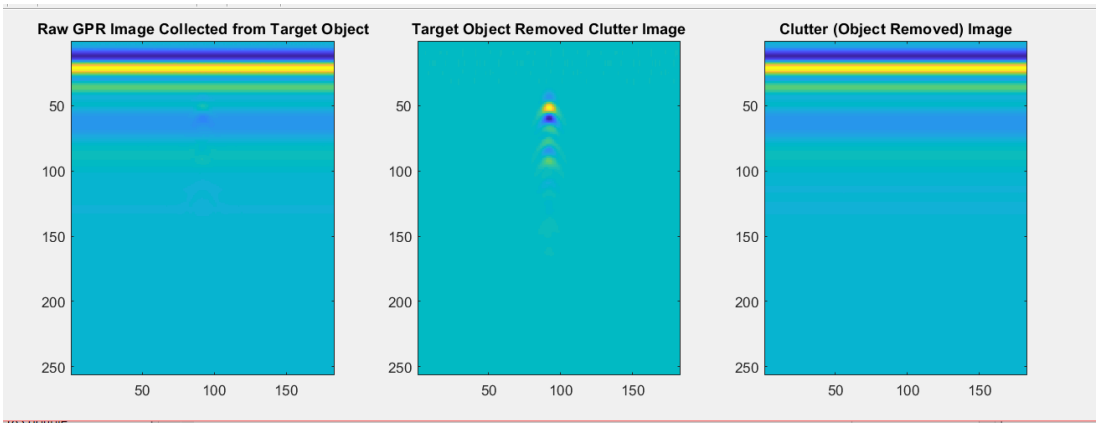


Figure 6.18: GPR images

7. PARALLEL PROCESSING OF FPGA

7.1 Parallelism On Nexys 4DDR

One of the great things that FPGA boards have is that they can process data parallelly i.e., Custom IPs that created at the PL part yields their operation results in one clock cycle. Thus, throughput can increase drastically. As seen on the former section utilizing one and only Custom IP do not have ability to speed up the RNMF algorithm. Aim is that utilizing any number of Custom IP in PL part parallelly to speed up the process.

```
3  void target_loop(float X[GPR_SIZE], float W[W_SIZE], float H[H_SIZE], float target[GPR_SIZE]){
4      int i0, i1;
5      int target_tmp;
6      loop_256: for (i0 = 0; i0 < 256; i0++) {
7        loop_183: for (i1 = 0; i1 < 183; i1++) {
8          target_tmp = i0 + (i1 << 8);
9          target[target_tmp] = X[target_tmp] - W[i0] * H[i1];
10       }
11     }
12 }
```

Figure 7.1: Aimed nested loop function

Aimed nested function block on Figure 7.1 is analyzed via Vivado HLS tool as seen on the Figure 7.2 with the aim of making sure that the parallelism can legitimately reduce the clock cycle of the operation process.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.263 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
562689	562689	5.627 ms	5.627 ms	562689	562689	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_256	562688	562688	2198	-	-	256	no
+ loop_183	2196	2196	12	-	-	183	no

Figure 7.2: Necessary clock cycle of the nested function to give correct results

Primer loop which is named as loop_256 needs 2198 clock cycle to operate. So, the original nested loop function needs 256*2198=562688 clock cycle and 1 for the input signal reception. Totally 5625689 clock cycle is needed to whole process to complete.

Different algorithms are tested. It is decided that utilizing 16 Custom IPs parallelly gives satisfying results. It should also be noted that utilizing more Custom IPs parallelly gives better operation results. However, there is a tradeoff between speed and the utilization of the resources of the FPGA as FFs and LUTs.

A sample code is given on Figure 7.3 and performance estimates utilizing 16 parallel target code line parallelly is given on Figure 7.4.

```
3  void target_loop(float X[GPR_SIZE], float W[W_SIZE], float H[H_SIZE], float target[GPR_SIZE]){
4      float W_temp;
5      int i0, i1, i2;
6      int target_tmp;
7      loop_256: for(i0 = 0 ; i0 < 256 ; i0++){
8          W_temp = W[i0];
9          loop_183: for(i1 = 0 ; i1 < 183 ; i1 += 3){
10             target_tmp = i0 + (i1 << 8);
11             loop_3: for(i2 = 0 ; i2 < 3 ; i2++){
12                 target[target_tmp + (i2 << 8)] = X[target_tmp + (i2 << 8)] - W_temp * H[i1 + i2];
13                 i2++;
14                 target[target_tmp + (i2 << 8)] = X[target_tmp + (i2 << 8)] - W_temp * H[i1 + i2];
15                 i2++;
16                 target[target_tmp + (i2 << 8)] = X[target_tmp + (i2 << 8)] - W_temp * H[i1 + i2];
17                 i2++;
18             }
19         }
20     }
21 }
```

Figure 7.3: A sample code utilizing 4 parallel line

By using parallelism, it is possible to decrease 5625689 to 33025 as seen on the estimates.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	10.409 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
33025	33025	0.344 ms	0.344 ms	33025	33025	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_256	33024	33024	129	-	-	256	no
+ loop_183	126	126	42	-	-	3	no

Figure 7.4: Performance estimates for 16 parallel target code line

Proposed solution is applied to the design by connecting 16 Custom IP to the MicroBlaze soft processor on Nexys 4DDR as in Figure 7.5 and the written C code is changed accordingly.

```
float H_temp;
int start, t_valid, i2;
for(i1 = 0 ; i1 < 183 ; i1++){
    H_temp = H[i1];
    for(i0 = 0 ; i0 < 256 ; i0 += 16){
        target_tmp = i0 + (i1 << 8);
        for(i2 = 0 ; i2 < 16 ; i2++){
            /*                          TARGET_IP(0)                        */
            start = 1;//IP calisiyor
            TARGET_IP_H_WriteRegStart(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG0_OFFSET, start);
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG1_OFFSET, X[target_tmp + i2]);  //x
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG2_OFFSET, W[i0 + i2]);          //w
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG3_OFFSET, H_temp);              //h
            start = 0;//Ip durdu
            TARGET_IP_H_WriteRegStart(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG0_OFFSET, start);          //start
            t_valid = TARGET_IP_H_ReadRegT_Valid(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG4_OFFSET);      //t_valid
            while(t_valid != 0){  //wait until the valid signal
                t_valid = TARGET_IP_H_ReadRegT_Valid(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG4_OFFSET);  //t_valid
            }
            target[target_tmp + i2] = TARGET_IP_H_ReadReg(XPAR_TARGET_IP_TARGET_IP_V2_0_S_AXI_BASEADDR, SLV_REG5_OFFSET);
            i2++;
            /*                          TARGET_IP(1)                        */
            start = 1;//IP calisiyor
            TARGET_IP_H_WriteRegStart(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG0_OFFSET, start);
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG1_OFFSET, X[target_tmp + i2]);  //x
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG2_OFFSET, W[i0 + i2]);          //w
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG3_OFFSET, H_temp);              //h
            start = 0;//Ip durdu
            TARGET_IP_H_WriteRegStart(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG0_OFFSET, start);          //start
            t_valid = TARGET_IP_H_ReadRegT_Valid(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG4_OFFSET);      //t_valid
            while(t_valid != 0){  //wait until the valid signal
                t_valid = TARGET_IP_H_ReadRegT_Valid(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG4_OFFSET);  //t_valid
            }
            target[target_tmp + i2] = TARGET_IP_H_ReadReg(XPAR_TARGET_IP_TARGET_IP_V2_1_S_AXI_BASEADDR, SLV_REG5_OFFSET);
            i2++;
            /*                          TARGET_IP(2)                        */
            start = 1;//IP calisiyor
            TARGET_IP_H_WriteRegStart(XPAR_TARGET_IP_TARGET_IP_V2_2_S_AXI_BASEADDR, SLV_REG0_OFFSET, start);
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_2_S_AXI_BASEADDR, SLV_REG1_OFFSET, X[target_tmp + i2]);  //x
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_2_S_AXI_BASEADDR, SLV_REG2_OFFSET, W[i0 + i2]);          //w
            TARGET_IP_H_WriteReg(XPAR_TARGET_IP_TARGET_IP_V2_2_S_AXI_BASEADDR, SLV_REG3_OFFSET, H_temp);              //h
            start = 0;//Ip durdu
            TARGET_IP_H_WriteRegStart(XPAR_TARGET_IP_TARGET_IP_V2_2_S_AXI_BASEADDR, SLV_REG0_OFFSET, start);          //start
            t_valid = TARGET_IP_H_ReadRegT_Valid(XPAR_TARGET_IP_TARGET_IP_V2_2_S_AXI_BASEADDR, SLV_REG4_OFFSET);      //t_valid
            while(t_valid != 0){  //wait until the valid signal
```

Figure 7.5: Altered C code utilizing 16 parallel IPs

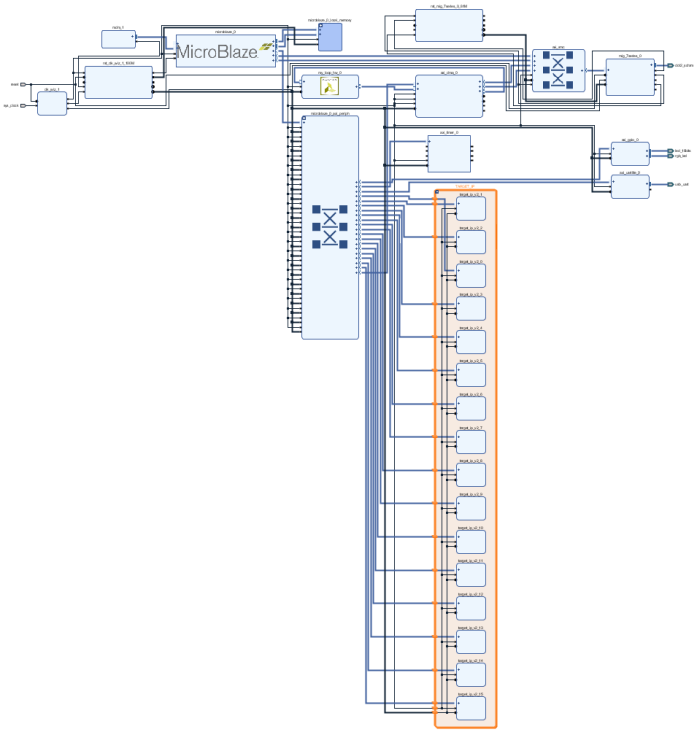


Figure 7.6: Proposed design on Nexys 4 DDR

Altered C code is operated on the proposed design and the resulting GPR images is obtained as seen on Figure 7.7 correctly.

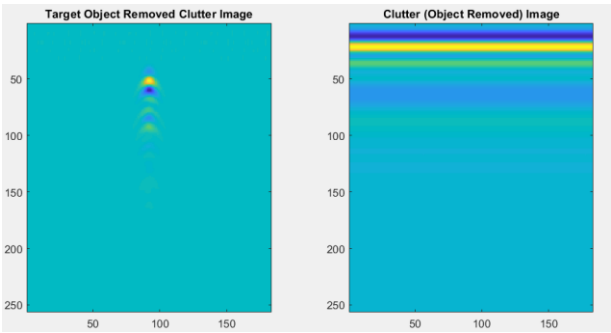


Figure 7.7: GPR images obtained with utilized parallel IPs

Operation results for a different proposal design is given as Table 6.1. But it should be noted that since the whole operation takes up too long on Nexys 4 DDR because of the low throughput of the card. These proposal operations tested for 2 iterations.

Table 7.1: Operation times for different proposals on Nexys 4 DDR

Proposals	Original C Code	Altered 16 parallel C code	1 IP	4 parallel IP	8 parallel IP	16 parallel IP
Time	0.4450”	0.4237”	0.5548”	0.5375”	0.5336”	0.5333”

As seen on the Table 7.1 best result is achieved on the altered code with 16 parallel line which is run on the design using 64 Kbyte cache memory. This result can be interpreted as that parallelism can speed up the operation duration. However rapid acquiring of data can increase up the performance of the algorithm as seen on the altered 16 parallel C code which runs on the design using cache memory. This result gives idea of using the DMA block on the design which enables the system to acquire data on DDR directly to the custom IPs on the PL section instead transferring to processor on the PS.

7.2 Parallelism On Zynq-7000 SoC

Same parallelism idea is tested on the ZedBoard as seen on Figure 7.8 and the operation results for different proposals is given on Table 7.2.

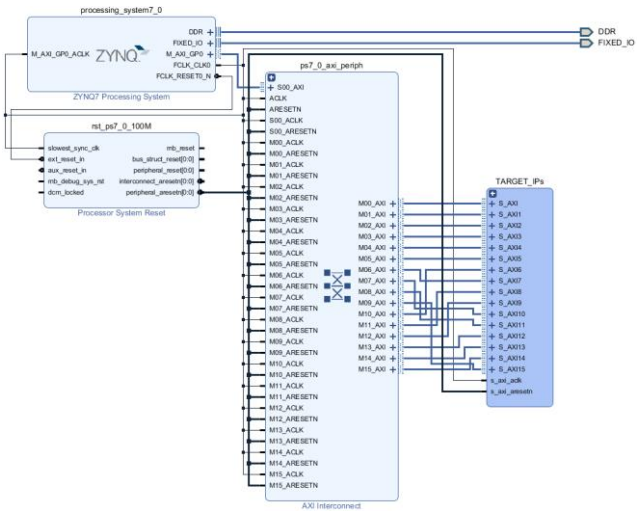


Figure 7.8: Zynq-7000 SoC design utilizing 16 parallel IP

As seen on the Table 7.2 it is not a good practice to run code on a memory based custom IPs even if parallelism is utilized. The reason is that ARM CORTEX-A9 processor in the PS section of the Zynq-700 SoC is operate in high frequency, 666 MHz to be exact. Operating in 100 MHz PL section does not improve the performance

the system on the contrary it slows down the process because of the transmitting signals between PS and PL. As a conclusion stream-based IP and utilizing DMA can yield a higher performance.

Table 7.2: Operation times for different proposals on Zynq-7000 SoC

	Altered Codes		Parallelism			
Optimization Levels	for(i0=0;i0<256;i0++) for(i1=0;i1<183;i1++)	for(i1=0;i1<183;i1++) for(i0=0;i0<256;i0+=16) for(i2=0;i2<16;i2++)	1 IP	4 IPs	8 IPs	16 IPs
-O0	3'54"	3'42"	16'26"	16'14"	16'12"	16'10"
-O3	1'15"	1'16"	12'10"	12'10"	12'12"	12'12"

8. DIRECT MODULE ACCESS (DMA) UTILIZATION ON FPGA

8.1 DMA Utilization on Nexys 4DDR

RNMF Algorithm with single AXI Lite interfaced custom IPs which performs T = X – W * H calculation completed process in 20.1205 seconds for 20 iterations. It is obvious that the duration needs to be improved. Also, as it concluded from the former section there is a need for special module to acquire data quicker. That module is called Direct Module Access (DMA). What DMA simply does is to transmit data from memory (DDR2 SDRAM) to intellectual property (IP) for each and every entry of data in each clock cycle. Thus, data does not need to be processed in the processor. In that way without any primary functions get involved i.e., fetch, decode, execute, and write back it is possible to calculate output of operation in a clock cycle.

DMAs in the whole project is configured as simple mode since the whole data in the memory is consecutive. It is also learned that for non-consecutive data DMA offers Scatter Gather Mode as seen on the Figure 8.1. Width of Buffer Length Register set as 26. This value represents the maximum value DMA can transmit which is 2^26 = 67,108,864 byte.

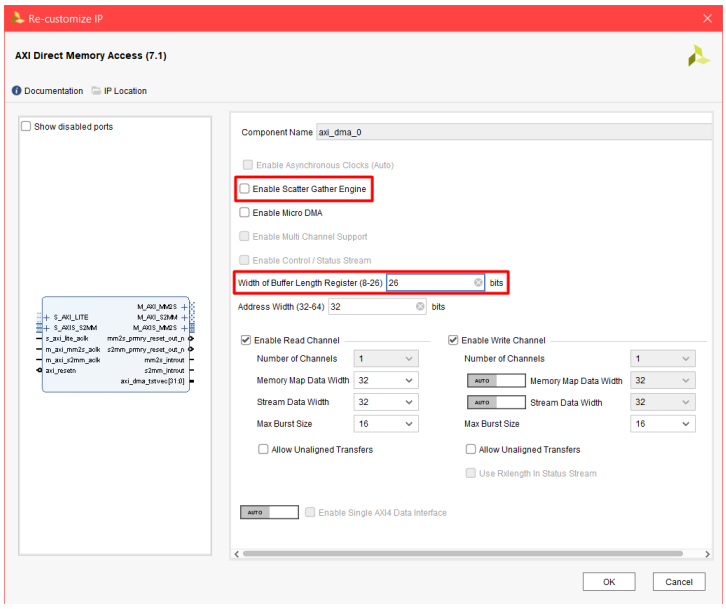


Figure 8.1: DMA Configurations

S_AXI_LITE port on DMA module is used for configuration i.e., flags, registers. Two more slave ports are added to the AXI SmartConnect module named as S02_AXI and S03_AXI. These ports allow DMA to read data from memory map. M_AXI_MM2S (AXI4 Memory Map Read) and M_AXI_S2MM (AXI4 Memory Map Write) ports on DMA module are responsible for this reading and writing operation respectively. Mentioned ports can be seen on Figure 8.2.

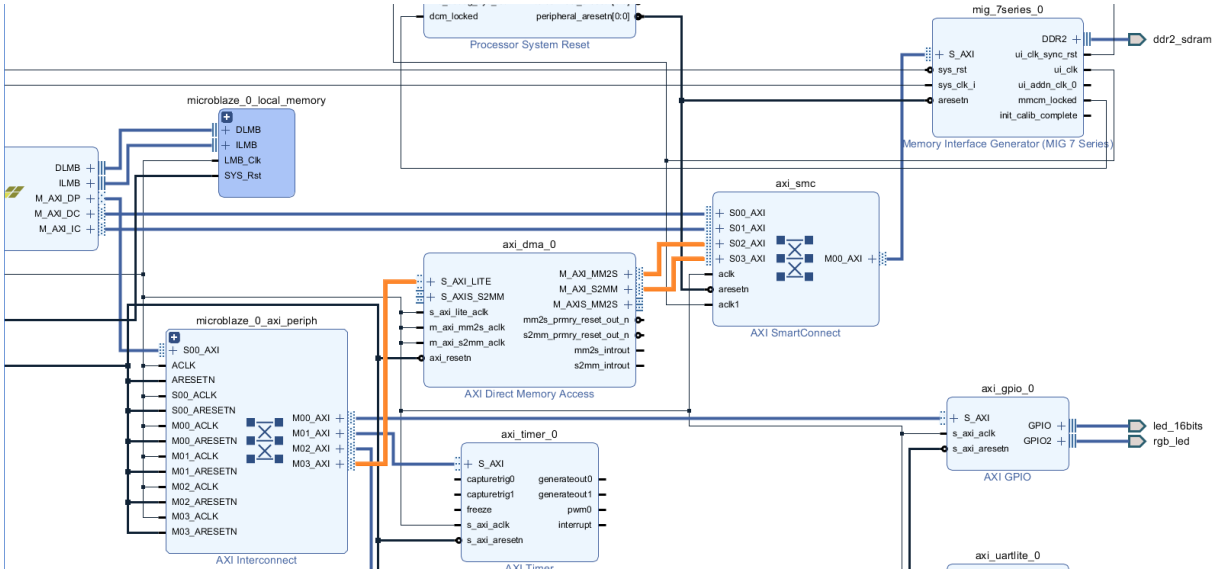
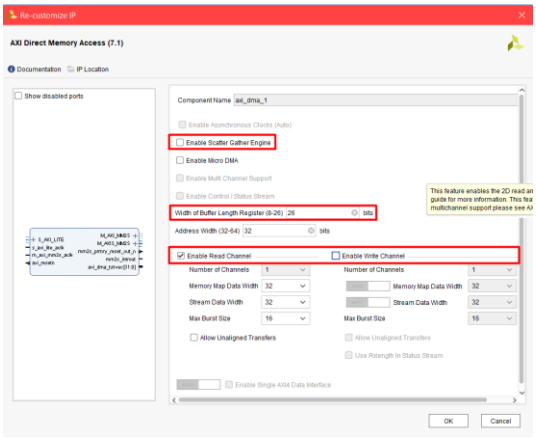


Figure 8.2: DMA ports

It can be also seen from the Figure 8.2 that DMA module is compatible with the AXI Stream interface to communicate with other IPs. This interface enables transmitting data in every clock cycle. AXI4 Stream Slave (S2MM) and AXI4 Stream Master

(MM2S) are the ports that allow reading from and writing to peripheral IPs, respectively. Additional two more DMA module which only performs reading form memory, is added to the design for the reason that 3 total namely X, W and H data are needed to calculate target data. Selecting only reading operation on DMA configuration interface can be seen on Figure 3.



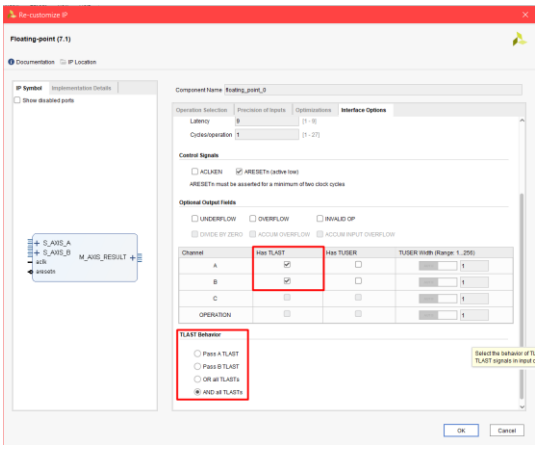


Figure 8.3: Selecting reading operation on DMA **Figure 8.4:** Tlast signal

To calculate the target data AXI Stream Interfaced Floating Point IPs are used on the Vivado Repository. Optional TLAST flag are added to the transmitted data to make data transaction over DMA reliable. TLAST flag designates the last bit of the transmitted data frame. Hence this flag is used to indicate the of the transaction. Created IP on the design can be seen on the Figure 8.5.

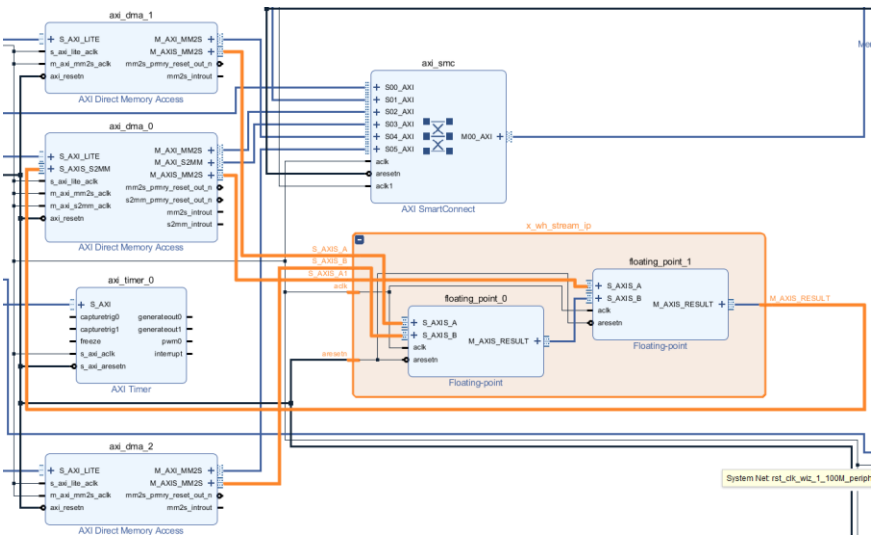


Figure 8.5: Design

To ensure that reading and writing from peripheral IPs are working correctly, debug signals are added to the ports (S2MM) and (MM2S), respectively. With helping of the

ILA (Internal Logic Analyzer) block it is easier to observe internal signals. ILA and the debug signals are seen on the Figure 8.6.

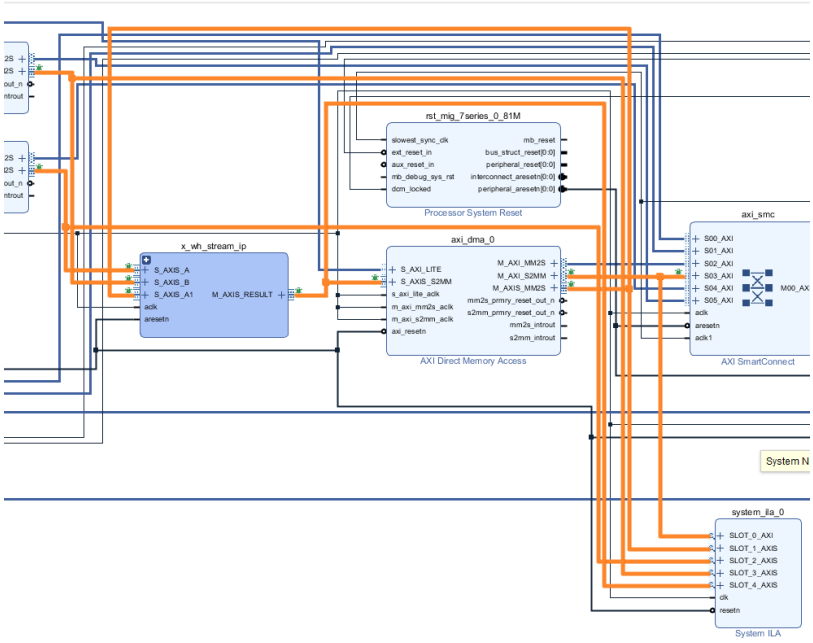


Figure 8.6: ILA and debug signals

Original code snippet that aimed to utilize with the DMA module is seen on the Figure 8.7. As it seen on the code X, W and H arrays hold 46848, 256 and 183 data in the arrays respectively. X and target arrays in this original code holds each 256 data in its columns starting from up to bottom. This behavior continues for each column from left side of the matrix to the right. This is shown in Figure 8.8.

```
for (i0 = 0; i0 < 256; i0++) {  //row
  for (i1 = 0; i1 < 183; i1++) {    //column
    target_tmp = i0 + (i1 << 8);
    target[target_tmp] = X[target_tmp] - W[i0] * H[i1];
  }
}
```

Figure 8.7: Original code

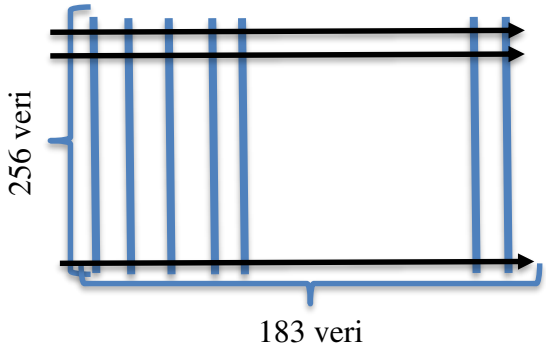


Figure 8.8: X[46848] and target[46848] array algorithm

It can be also shown that target array data values are acquired starting from beginning of the row to the end of the row as stated as black arrow on Figure 8.8.

Since the original code expects continuity for the W and H arrays, as in the array X, two new arrays named W_Temp and H_Temp are created. As in the original code does, W_Temp array repeats whole 256 W data over 183 times continuously. And H_Temp array repeats each H data value 256 times one by one. These arrays are shown on Figure 9.

```
for (iter = 0; iter < 1000; iter++) {
    /////////////Temporary Arrays///////////////
      int temp = 0;
      for(int col = 0 ; col < 183 ; col++){
          for(int row = 0 ; row < 256 ; row++){
              W_temp[temp] = W[row];
              temp++;
          }
      }
      temp = 0;
      for(int col = 0 ; col < 183 ; col++){
          for(int row = 0 ; row < 256 ; row++){
              H_temp[temp] = H[col];
              temp++;
          }
      }
```

Figure 8.9: Newly created temporary W and H arrays

On VITIS SDK DMA configurations are made as seen on the Figure 8.10. This configuration enables transmission of data.

```
/*************************************************************************/
  //DMA_0 = X, T
  //DMA_1 = W
  //DMA_2 = H
  XAxiDma my_DMA_XT, my_DMA_W, my_DMA_H;
  XAxiDma_Config *my_DMA_config_XT, *my_DMA_config_W, *my_DMA_config_H;
  my_DMA_config_XT = XAxiDma_LookupConfigBaseAddr(XPAR_AXI_DMA_0_BASEADDR);
  my_DMA_config_W = XAxiDma_LookupConfigBaseAddr(XPAR_AXI_DMA_1_BASEADDR);
  my_DMA_config_H = XAxiDma_LookupConfigBaseAddr(XPAR_AXI_DMA_2_BASEADDR);
  XAxiDma_CfgInitialize(&my_DMA_XT, my_DMA_config_XT);
  XAxiDma_CfgInitialize(&my_DMA_W, my_DMA_config_W);
  XAxiDma_CfgInitialize(&my_DMA_H, my_DMA_config_H);
/*************************************************************************/
```

Figure 8.10: DMA Configurations

Necessary codes to utilize DMA in the design properly is shown in Figure 8.11. These codes can be explained as below.

Since MicroBlaze soft processor in the design uses a 64 Kbyte cache memory, deleting data in cache and making sure that the data that is wanted to transmit resides in the DDR2 SDRAM. Then DMA to Device (DDR to IP) and the Device to DMA (IP to DDR) data transmission codes are added. Afterwards, it is waited for the transmitting process to complete successfully. With the last code it is made sure that the target data values are resides in DDR.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //writes back target, X, W and H from cache to DDR2 SDRAM
    //make sure that transmitted and received data arrays are stored in the DDR memory
    Xil_DCacheFlushRange((UINTPTR)target, sizeof(float) * 46848);
    Xil_DCacheFlushRange((UINTPTR)X, sizeof(float) * 46848);
    Xil_DCacheFlushRange((UINTPTR)W_temp, sizeof(float) * 46848);
    Xil_DCacheFlushRange((UINTPTR)H_temp, sizeof(float) * 46848);

    //Copying float data(X, W, H) from DDR2 SDRAM to the IP
    XAxiDma_SimpleTransfer(&my_DMA_XT, (UINTPTR)X, 46848 * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
    XAxiDma_SimpleTransfer(&my_DMA_W, (UINTPTR)W_temp, 46848 * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
    XAxiDma_SimpleTransfer(&my_DMA_H, (UINTPTR)H_temp, 46848 * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
    //Copying float data(T) from IP to the DDR2 SDRAM
    XAxiDma_SimpleTransfer(&my_DMA_XT, (UINTPTR)target, 46848 * sizeof(float), XAXIDMA_DEVICE_TO_DMA);

    while (XAxiDma_Busy(&my_DMA_W,XAXIDMA_DMA_TO_DEVICE)){/* Wait */};
    while (XAxiDma_Busy(&my_DMA_H,XAXIDMA_DMA_TO_DEVICE)){/* Wait */};
    while ((XAxiDma_Busy(&my_DMA_XT,XAXIDMA_DEVICE_TO_DMA)) || (XAxiDma_Busy(&my_DMA_XT,XAXIDMA_DMA_TO_DEVICE))){/* Wait */};
    //invalidate the cache for the received data's memory range
    Xil_DCacheInvalidateRange((UINTPTR)target, 46848 * sizeof(float));
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Figure 8.11: DMA utilization codes

ILA Trigger Setup is made as seen on the Figure 8.12. ILA block triggers when the t_valid flag on the W array is set.

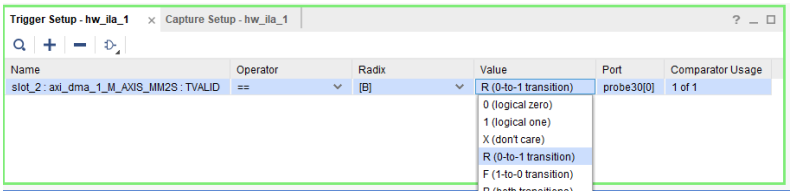


Figure 8.12: Trigger Setup

ILA observations are shown in Figure 8.13. It is seen that X, W and H array transmits data from DDR to IP according to the plan. X array transmits 46848 data sequentially, W array transmits 256 data in a repeating session 183 times and H array transmits 183 data repetitively 256 times for each of the value. In Figure 8.13, 200 data inputs are observed. Therefore, it is logical that H array seems to remain constant.

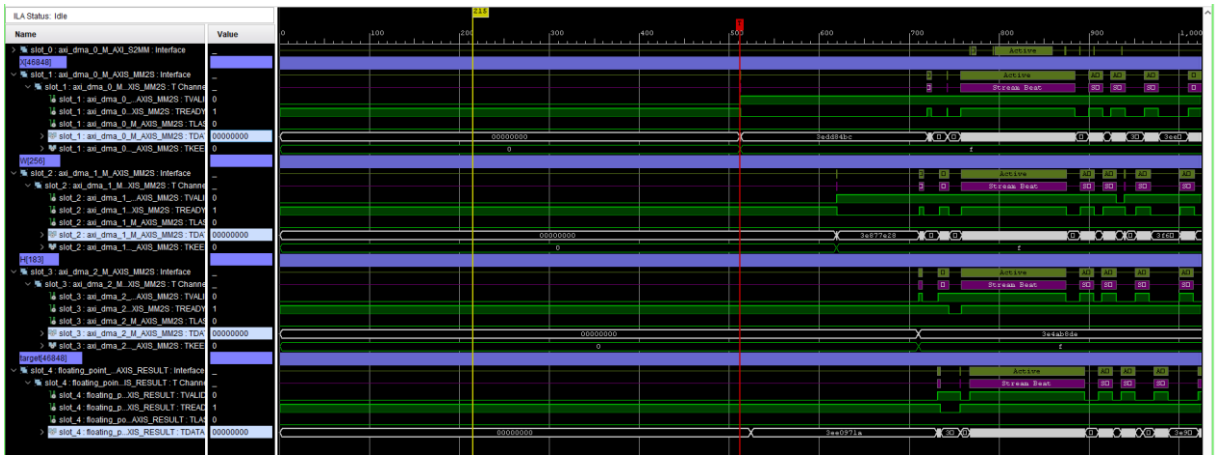


Figure 8.13: ILA observations

The target[46848] array and the data written to DDR over DMA is shown in Figure 8.14 and in Figure 8.15 respectively. From the two figures it can be said that the DMA is working according to plan.

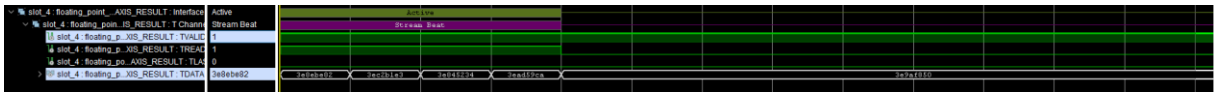


Figure 8.14: target[46848] array

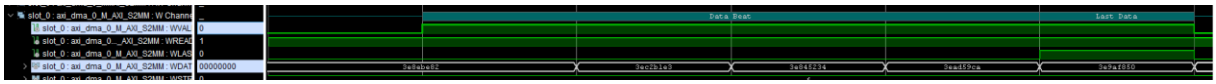


Figure 8.15: Data written DDR over DMA

Lastly the whole Rnmf algorithm is run. Target and Clutter data are obtained in less time successfully on Figure 8.16.

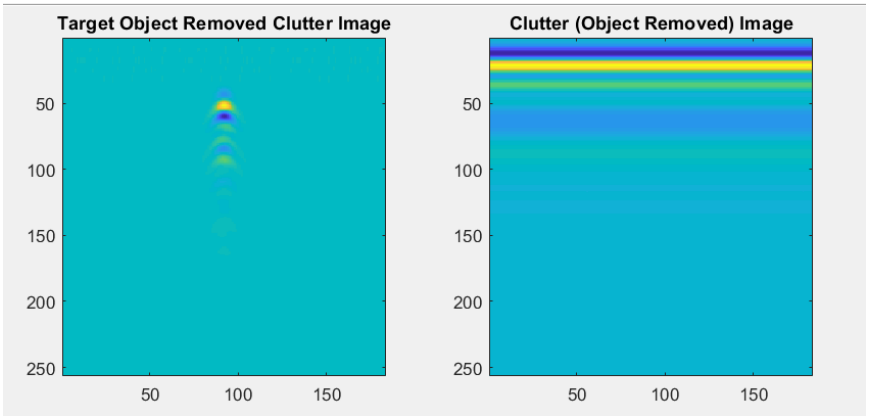


Figure 8.16: Resulting GPR images.

8.2 DMA Utilization on Zynq-7000 SoC

To utilize DMA module as well in ZedBoard, design shown in Figure 8.17 is built. Design procedure is almost identical except for the DMA structure.

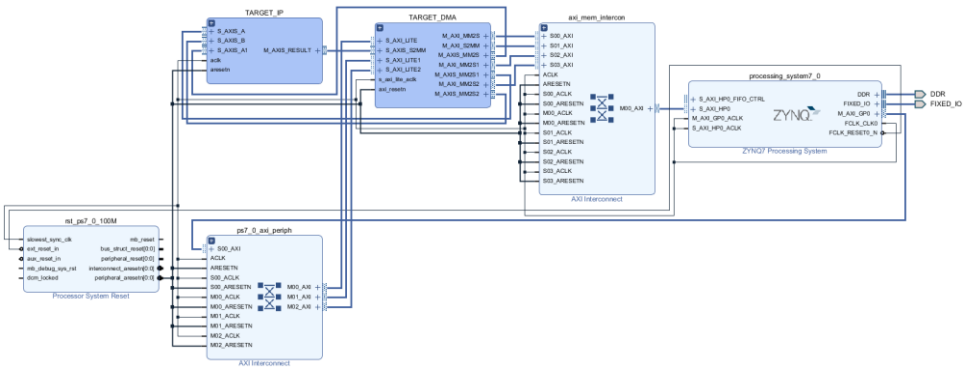


Figure 8.17: Design on ZedBoard

While configuring DMAs in the design, Max Burst Size is chosen as maximum 256 bits, as shown in Figure 18. This value determines the size of the packet in each transmission of the DMA block using AXI Stream interface. To increase the

performance of the system this chosen as maximum. By this means it is aimed that the high throughput can be achieved.

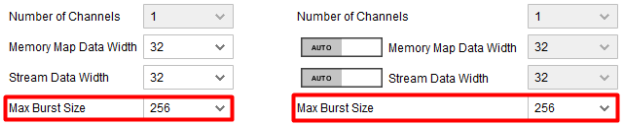


Figure 8.18: DMA Max Burst Size configuration

Zynq architecture offers High Performance (HP) Slave Ports to allow the IPs in the Programmable logic (PL) side of the FPGA to reach in a fast manner directly to the data in the DDR memory without visiting the processor. This can be seen on the Zynq internal design as shown in Figure 8.19.

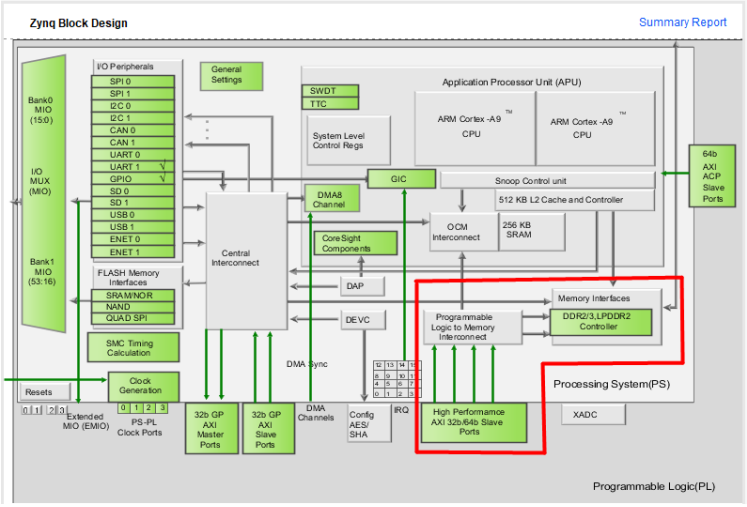


Figure 8.19: HP Slave Ports on Zynq Architecture

PL side of the FPGA operates in 100 MHz. Data length is 32 bits. Thus, the bandwidth on HP line is 100 MHz * 32 bits, 400MBps.

Table 8.1 shows the operating durations for different optimization selections. As it seen from the table aimed performance is not met in the ZedBoard. The reason for this is that ARM CORTEX-A9 processor in the Zynq architecture is already operates in very high frequencies namely 666 MHz.

Table 8.1: Operation durations on ZedBoard

Optimizations	Without DMA	With DMA
-O0	3’54”	4’29”
-O3	1’15”	1’44”

9. CREATING HARDWARE VIA VIVADO HLS

9.1 HLS Utilization on Nexys 4DDR

VITIS High-Level Synthesis enables creating hardware by means of writing pure C based codes. Code snippets in “rnmf_in.c” file will be tried to be converting into hardwares by utilizing Vitis HLS. Original code snippet that is going to be hardware on Vitis HLS can be seen on Figure 9.1, 9.2, and 9.3. b_abs() function prints the absolute value of target[46848] array data to the varargin_2[46848] array before the first loop. After necessary operations are performed in the first loop, array z1[46848] is obtained at the end of the loop. b_sign() function detects the sign of the data of the target array before the second loop starts. Newly obtained array is multiplied with the z1[46848] array to produce the output target[46848] array.

```
void b_abs(const float x[46848], float y[46848])    //double float a çevrildi
{
  int k;
  for (k = 0; k < 46848; k++) {
      //y[k] = fabs(x[k]);  //double float a çevrildi
      y[k] = fabsf(x[k]);
  }
}
```

Figure 9.1: b_abs() function

```
void b_sign(float x[46848]) //double float a çevrildi
{
  int k;
  float b_x;    //double float a çevrildi
  for (k = 0; k < 46848; k++) {
    b_x = x[k];
    if (x[k] < 0.0) {
      b_x = -1.0;
    } else if (x[k] > 0.0) {
      b_x = 1.0;
    } else {
      if (x[k] == 0.0) {
        b_x = 0.0;
      }
    }
    x[k] = b_x;
  }
}
```

Figure 9.2: b_sign() function

```
b_abs(target, varargin_2);
for (target_tmp = 0; target_tmp < 46848; target_tmp++) {
  norms = varargin_2[target_tmp] - 0.00015;
  varargin_2[target_tmp] -= 0.00015;
  z1[target_tmp] = fmaxf(0.0, norms);   //double float a çevrildi
  //z1[target_tmp] = fmax(0.0, norms);
}

b_sign(target);
for (i0 = 0; i0 < 46848; i0++) {
  target[i0] *= z1[i0];
}
```

Figure 9.3: Code snippet is going to be hardware

Original loop can be simplified as shown in Figure 9.4 and 9.5. Simplified function has 2 loops and two functions. Thus, provides higher performance.

```
my_loop.cpp   my_loop.h   my_loop_tb.cpp
1 #ifndef _MY_LOOP_
2 #define _MY_LOOP_
3 #define INPUT_SIZE 46848 //256*183 = 46848
4
5 void my_loop_hw(float target_in[INPUT_SIZE], float target_out[INPUT_SIZE]);
6 #endif
```

Figure 9.4: Simplified functions header code

```
my_loop.cpp   my_loop.h   my_loop_tb.cpp
1 #include "my_loop.h"
2 #include <math.h> //fabsf(), fmaxf()
3
4 //HLS kullanarak yaratilacak olan IP
5 void my_loop_hw(float target_in[INPUT_SIZE], float target_out[INPUT_SIZE]){
6     float a_x; //z1[target_tmp] ile esdeger
7     float b_x; //b_sign(target[target_tmp]) ile esdeger
8     float norms;
9     int target_tmp;
10    my_loop: for (target_tmp = 0; target_tmp < INPUT_SIZE; target_tmp++) {
11      norms = fabsf(target_in[target_tmp]); //norms = varargin_2[target_tmp] - 0.00015;
12      a_x = fmaxf(0.0, norms);  //z1[target_tmp] = fmaxf(0.0, norms);
13
14      //b_sign(target) ile esdeger
15      if (target_in[target_tmp] < 0.0) {b_x = -1.0;}
16      else if (target_in[target_tmp] > 0.0) {b_x = 1.0;}
17      else {if (target_in[target_tmp] == 0.0) {b_x = 0.0;}}
18
19      target_out[target_tmp] = b_x * a_x;   //target[i0] *= z1[i0] ile esdeger
20    }
21 }
```

Figure 9.5: Simplified C++ functions

A new testbench file is written in order to make sure that newly created simplified code gives the same result as original code. In this testbench file, my_loop_hw() represents the hardware function, while loop_sw() represents the code that is currently running correctly in the software. These both functions are run with the same input signals and the output signals are compared. In this way accuracy of the simplified code is verified. Testbench file can be shown as Figure 9.6.

```c
#include <stdio.h>
#include "my_loop.h"
#include <math.h>

//Software icin test edilecek fonksiyonlar
void b_abs(float x[46848], float y[46848]);
void b_sign(float x[46848]);
void loop_sw(float target_in[46848], float target_out[46848]);
//Yeni yazilan Hardware IP
void my_loop_hw(float target_in[INPUT_SIZE], float target_out[INPUT_SIZE]);

int main(){
    int fail_flag = 0;
    int err = 0;

    //ornek bir giris isareti olusturma
    float target_in[INPUT_SIZE] = {};
    int i0;
    for(i0 = 0 ; i0 < INPUT_SIZE ; i0++){
        target_in[i0] = 2*i0 + 0.159753;
    }

    //HW ve SW icin ayrı cıkıs isaretleri
    float target_out_hw[INPUT_SIZE];    //hardware
    float target_out_sw[INPUT_SIZE];    //software

    //HW ve SW icin deneniyor
    loop_sw(target_in, target_out_sw);  //hardware function
    my_loop_hw(target_in, target_out_hw);   //software function

    //Hata testi
    int i1;
    for(i1 = 0 ; i1 < INPUT_SIZE ; i1++){
        if(target_out_hw[i1] != target_out_sw[i1]){
            fail_flag = 1;
            err++;
        }
        if(fail_flag == 1){
            printf("Failed at [%d] | Hw:[%f] Sw:[%f]\n", i1, target_out_hw[i1], target_out_sw[i1]);
            fail_flag = 0;
        }
    }
    printf("Total error: %d\n", err);
    if(err != 0)
        printf("FAILED\n");
    else
        printf("PASSED\n");
    return fail_flag;
}
```

```c
void loop_sw(float target_in[46848], float target_out[46848]){
    float varargin_2[46848];
    float z1[46848];
    b_abs(target_in, varargin_2);
    int target_tmp;
    float norms;
    for (target_tmp = 0; target_tmp < 46848; target_tmp++) {
      norms = varargin_2[target_tmp] - 0.00015;
      varargin_2[target_tmp] -= 0.00015;
      z1[target_tmp] = fmaxf(0.0, norms);
    }
    b_sign(target_in);
    int i0;
    for (i0 = 0; i0 < 46848; i0++) {
        target_in[i0] *= z1[i0];
    }
    //buffer gorevi goruyor (cikis = giris)
    for(int i = 0 ; i < 46848 ; i++)
    {
        target_out[i] = target_in[i];
    }
}
void b_abs(float x[46848], float y[46848])
{
  int k;
  for (k = 0; k < 46848; k++) {
    y[k] = fabsf(x[k]);
  }
}
void b_sign(float x[46848])
{
  int k;
  float b_x;
  for (k = 0; k < 46848; k++) {
    b_x = x[k];
    if (x[k] < 0.0) {
      b_x = -1.0;
    } else if (x[k] > 0.0) {
      b_x = 1.0;
    } else {
      if (x[k] == 0.0) {
        b_x = 0.0;
      }
    }
    x[k] = b_x;
  }
}
```

Figure 9.6: Testbench file

Before running the testbench file, since HLS contains more than one function, the my_loop_hw() function to be converted to hardware was written in Project Settings → Synthesis → Top Function.

It can be seen in Figure 9.7 that the simplified code after Run C Simulation gives the same results as the code already used in the software.

```
Console   Errors   Warnings
Vivado HLS Console
INFO: [HLS 200-10] Setting target device to 'xc7a100t-csg324-1'
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
make: 'csim.exe' is up to date.
Total error: 0
PASSED
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Figure 9.7: Testbench output

Run C Synthesis is used to synthesize the function. The output obtained after the synthesis is given in Figure 9.8.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.718 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
374785	374785	3.748 ms	3.748 ms	374785	374785	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- my_loop	374784	374784	8	-	-	46848	no

Figure 9.8: Synthesis output

As can be seen, the hardware completes the operation in 8.718 nanoseconds at each clock. In this way, setup and hold time requirements are fulfilled. In other words, slack is positive. The hardware finishes its work in a total of 374785 clock cycles. As seen from the loop part, the amount of data in the array is 46848, each iteration takes 8 clock cycles, 1 clock cycle is spent to read the data from the memory. Operation duration = 374785 = 46848 ∗ 8 + 1 is calculated as can be seen from the pipeline part, pipeline is not used in this code.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	337	-
FIFO	-	-	-	-	-
Instance	-	3	341	350	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	56	-
Register	-	-	188	-	-
Total	0	3	529	743	0
Available	270	240	126800	63400	0
Utilization (%)	0	1	~0	1	0

Figure 9.9: Utilization

As can be seen in Figure 9.9, since the hardware uses multiplication on line 19, 3 floating point multipliers (DSP48E) have been added to the hardware. The hardware also uses 529 FF and 743 LUT.

In the HLS interface, the blocks in which the elapsed time in each iteration is spent can be observed in detail by clicking on the Analysis section[6].

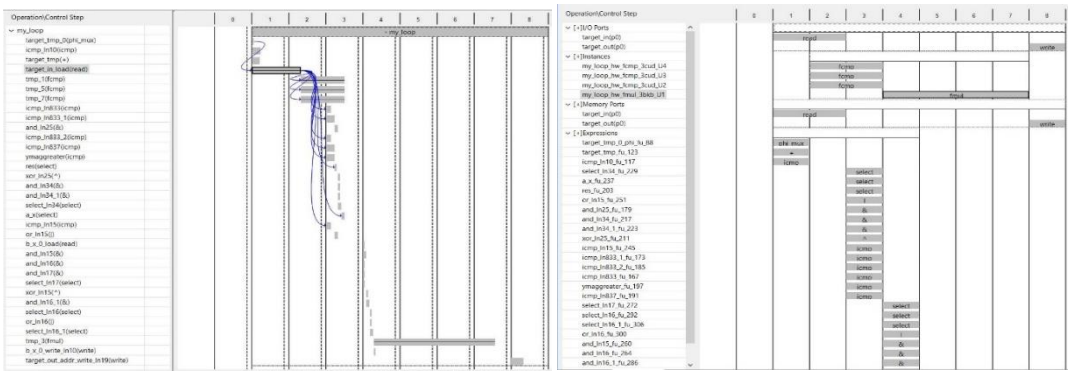


Figure 9.10: Analysis section

As a result of the synthesis, the Verilog code of the hardware is observed as shown in Figure 9.11.

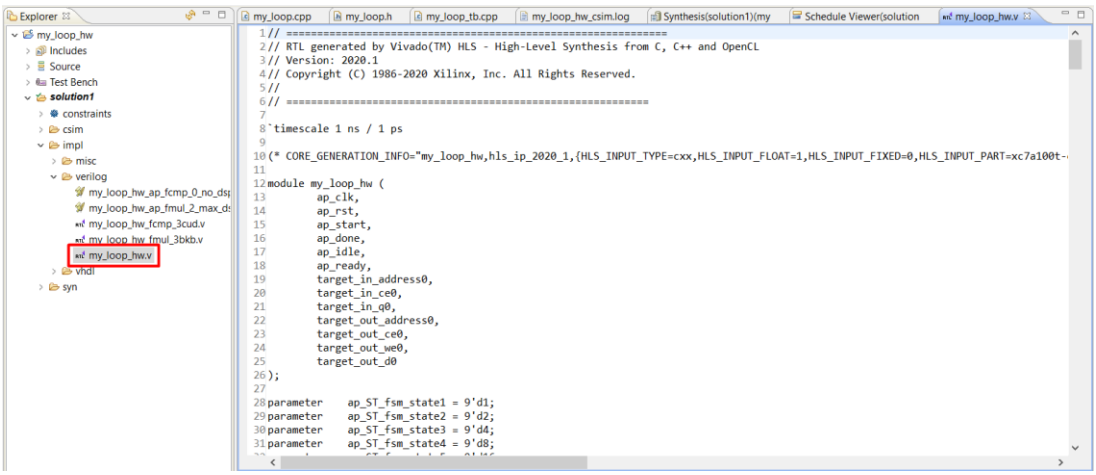


Figure 9.11: Verilog code of the hardware

To verify the functionality of the Verilog code written by HLS, Run C/RTL Cosimulation was performed. The correctness of the process was observed on the console screen as shown in Figure 9.12.

```
Console   Errors   Warnings   DRCs
Vivado HLS Console
## quit
INFO: [Common 17-206] Exiting xsim at Wed Apr  5 19:14:03 2023...
INFO: [COSIM 212-316] Starting C post checking ...
Total error: 0
PASSED
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater than 1 in RTL simulation. Otherwise, they will be marked as all NA. If user wants to calculat
Finished C/RTL cosimulation.
```

Figure 9.12: Successful Run C/RTL Cosimulation

9.2 AXI Stream Compatible Hardware on Nexys 4DDR

In order to speed up the RNMF algorithm and to read the data on the hardware created in each clock cycle, the hardware created was made compatible with AXI Stream. For this, input and output data were selected Interface → mode → axis from the VIVADO HLS Directive Editor window. In addition, Interface → mode → ap_ctrl_none was selected from the same window to cancel the control port. For faster and shorter running hardware, Directive → Pipeline was activated. The created pragmas can be seen in Figure 9.13.

```
5  void my_loop_hw(hls::stream<axis_data> &target_in, hls::stream<axis_data> &target_out){
6  #pragma HLS INTERFACE ap_ctrl_none port=return
7  #pragma HLS INTERFACE axis register both port=target_out
8  #pragma HLS INTERFACE axis register both port=target_in
9      float a_x; //z1[target_tmp] ile esdeger
10     float b_x; //b_sign(target[target_tmp]) ile esdeger
11     float norms;
12     int target_tmp;
13     axis_data local_read, local_write; //read and write from stream interface
14     my_loop: for (target_tmp = 0; target_tmp < INPUT_SIZE; target_tmp++) {
15 #pragma HLS PIPELINE
```

Figure 9.13: Pragmas

As can be seen in Figure 9.14, the interface of the hardware by performing C Synthesis has signals such as valid and ready that provide communication on the AXI Stream interface. However, as can be seen, the last signal, which is important in the use of DMA, could not be obtained.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	my_loop_hw	return value
ap_rst_n	in	1	ap_ctrl_none	my_loop_hw	return value
target_in_TDATA	in	32	axis	target_in	pointer
target_in_TVALID	in	1	axis	target_in	pointer
target_in_TREADY	out	1	axis	target_in	pointer
target_out_TDATA	out	32	axis	target_out	pointer
target_out_TVALID	out	1	axis	target_out	pointer
target_out_TREADY	in	1	axis	target_out	pointer

Figure 9.14: Hardware interface

To solve this problem, the header file has been reorganized and is shown in Figure 9.15. In order to enable DMA usage, target_in and target_out data are defined as axis_data format with last flag.

my_loop.cpp | my_loop.h | my_loop_tb.cpp | my_loop_hw_csim.log | my_loop_hw.v | Simulat

```
1  #ifndef _MY_LOOP_
2  #define _MY_LOOP_
3  #include <hls_stream.h> //axi stream header dosyası
4  #include "ap_int.h" //ap_int header dosyası
5  #define INPUT_SIZE 46848 //256*183 = 46848
6
7  struct axis_data {
8      float data; //veri
9      ap_uint<1> last;    //last bit
10 };
11
12 void my_loop_hw(hls::stream<axis_data> &target_in, hls::stream<axis_data> &target_out);
13 #endif
```

Figure 9.15: AXI Stream compatible version of the header file

The vectors in my_loop.cpp have been adapted to the stream interface. In addition, the last bit representing the last data has been assigned. The file is shown in Figure 9.16.

my_loop.cpp | my_loop.h | my_loop_tb.cpp | my_loop_hw_csim.log | my_loop_hw.v | Simulation(solutio

```
1  #include "my_loop.h"
2  #include <math.h> //fabsf(), fmaxf()
3
4  //HLS kullanarak yaratilacak olan IP
5  void my_loop_hw(hls::stream<axis_data> &target_in, hls::stream<axis_data> &target_out){
6  #pragma HLS INTERFACE ap_ctrl_none port=return
7  #pragma HLS INTERFACE axis register both port=target_out
8  #pragma HLS INTERFACE axis register both port=target_in
9      float a_x; //z1[target_tmp] ile esdeger
10     float b_x; //b_sign(target[target_tmp]) ile esdeger
11     float norms;
12     int target_tmp;
13     axis_data local_read, local_write; //read and write from stream interface
14     my_loop: for (target_tmp = 0; target_tmp < INPUT_SIZE; target_tmp++) {
15 #pragma HLS PIPELINE
16         local_read = target_in.read(); //okuma islemi
17         norms = fabsf(local_read.data) - 0.00015;   //norms = varargin_2[target_tmp] - 0.00015;
18         a_x = fmaxf(0.0, norms);    //z1[target_tmp] = fmaxf(0.0, norms);
19
20         //b_sign(target) ile esdeger
21         if (local_read.data < 0.0) {b_x = -1.0;}
22         else if (local_read.data > 0.0) {b_x = 1.0;}
23         else {if (local_read.data == 0.0) {b_x = 0.0;}}
24
25         local_write.data = b_x * a_x;   //target[i0] *= z1[i0] ile esdeger
26
27         if(target_tmp == (INPUT_SIZE-1)){   //last bit son degerde iken uretilir
28             local_write.last = (ap_uint<1>)1;
29         }else{
30             local_write.last = (ap_uint<1>)0;
31         }
32         target_out.write(local_write);
33     }
34 }
```

Figure 9.16: AXI Stream compatible hardware code

The adaptation process to the AXI Stream interface continued in the testbench code. The changes made are shown in Figure 9.17.

```c
#include <stdio.h>
#include "my_loop.h"
#include <math.h>

//Software icin test edilecek fonksiyonlar
void b_abs(float x[46848], float y[46848]);
void b_sign(float x[46848]);
void loop_sw(float target_in[46848], float target_out[46848]);
//Yeni yazilan Hardware IP
void my_loop_hw(hls::stream<axis_data> &target_in, hls::stream<axis_data> &target_out);

int main(){
    int fail_flag = 0;
    int err = 0;

    axis_data local_read, local_write; //axi stream arayuzu okuma yazma islemi
    hls::stream<axis_data> target_in_hw;    //donanıma ait stream yapisinda giris verisi
    hls::stream<axis_data> target_out_hw;   //donanıma ait stream yapisinda cikis verisi

    //ornek bir giris isareti olusturma
    float target_in_sw[INPUT_SIZE] = {};
    int i0;
    for(i0 = 0 ; i0 < INPUT_SIZE ; i0++){
        target_in_sw[i0] = 2*i0 + 0.159753;
    }
    int i2;
    for(i2 = 0 ; i2 < INPUT_SIZE ; i2++){
        local_read.data = target_in_sw[i2];
        if(i2 == (INPUT_SIZE-1)){   //last bit
            local_read.last = (ap_uint<1>)1;
        }else {
            local_read.last = (ap_uint<1>)0;
        }
        target_in_hw.write(local_read);
    }

    //HW ve SW icin ayrı cıkıs isaretleri
    float target_out_sw[INPUT_SIZE];    //software

    //HW ve SW icin deneniyor
    loop_sw(target_in_sw, target_out_sw);   //hardware function
    my_loop_hw(target_in_hw, target_out_hw);    //software function

    //Hata testi
    int i1;
    for(i1 = 0 ; i1 < INPUT_SIZE ; i1++){
        target_out_hw.read(local_write);
        if(local_write.data != target_out_sw[i1]){
            fail_flag = 1;
            err++;
        }
        if(fail_flag == 1){
            printf("Failed at [%d] | Hw:[%f] Sw:[%f]\n", i1, local_write.data, target_out_sw[i1]);
            fail_flag = 0;
        }
    }
    }
    printf("Total error: %d\n", err);
    if(err != 0)
        printf("FAILED\n");
    else
        printf("PASSED\n");
    return fail_flag;
}

void loop_sw(float target_in[46848], float target_out[46848]){
    float varargin_2[46848];
    float z1[46848];
    b_abs(target_in, varargin_2);
    int target_tmp;
    float norms;
    for (target_tmp = 0; target_tmp < 46848; target_tmp++) {
      norms = varargin_2[target_tmp] - 0.00015;
      varargin_2[target_tmp] -= 0.00015;
      z1[target_tmp] = fmaxf(0.0, norms);
    }
    b_sign(target_in);
    int i0;
    for (i0 = 0; i0 < 46848; i0++) {
        target_in[i0] *= z1[i0];
    }
    //buffer gorevi goruyor (cikis = giris)
    for(int i = 0 ; i < 46848 ; i++)
    {
        target_out[i] = target_in[i];
    }
}
void b_abs(float x[46848], float y[46848])
{
  int k;
  for (k = 0; k < 46848; k++) {
    y[k] = fabsf(x[k]);
  }
}
void b_sign(float x[46848])
{
  int k;
  float b_x;
  for (k = 0; k < 46848; k++) {
    b_x = x[k];
    if (x[k] < 0.0) {
      b_x = -1.0;
    } else if (x[k] > 0.0) {
      b_x = 1.0;
    } else {
      if (x[k] == 0.0) {
        b_x = 0.0;
      }
    }
    x[k] = b_x;
  }
}
```

Figure 9.17: AXI Stream compatible Testbench

C/RTL Cosimulation was run, and a successful result was obtained as shown in Figure 9.18.

```
Vivado HLS Console
run: Time (s): cpu = 00:00:02 ; elapsed = 00:00:11 . Memory (MB): peak
## quit
INFO: [Common 17-206] Exiting xsim at Wed Apr  5 23:39:24 2023...
INFO: [COSIM 212-316] Starting C post checking ...
Total error: 0
PASSED
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is
Finished C/RTL cosimulation.
```

Figure 9.18: Successful simulation result

Synthesis was performed and the last bit was obtained as seen in the new interface.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	my_loop_hw	return value
ap_rst_n	in	1	ap_ctrl_none	my_loop_hw	return value
target_in_TDATA	in	32	axis	target_in_V_data	pointer
target_in_TVALID	in	1	axis	target_in_V_last_V	pointer
target_in_TREADY	out	1	axis	target_in_V_last_V	pointer
target_in_TLAST	in	1	axis	target_in_V_last_V	pointer
target_out_TDATA	out	32	axis	target_out_V_data	pointer
target_out_TVALID	out	1	axis	target_out_V_last_V	pointer
target_out_TREADY	in	1	axis	target_out_V_last_V	pointer
target_out_TLAST	out	1	axis	target_out_V_last_V	pointer

Figure 9.19: Hardware interface

It can be said that the performance setup and hold time requirements in the synthesis result are complied with, and the hardware runs faster by reducing the processing time to 46863 clock cycles thanks to Pipeline. The outputs are shown in Figure 9.20.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.718 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
46863	46863	0.469 ms	0.469 ms	46863	46863	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- my_loop	46861	46861	15	1	1	46848	yes

Figure 9.20: Performance output

Finally, as shown in Figure 9.21, Export RTL is selected, and the hardware design is finalized.

```
Console  Errors  Warnings  DRCs
Vivado HLS Console
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'my_loop_hw_ap_fpext_0_no_dsp_32'...
WARNING: [IP_Flow 19-4832] The IP name 'my_loop_hw_ap_fptrunc_0_no_dsp_64' you have specified is long. Th
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'my_loop_hw_ap_fptrunc_0_no_dsp_64'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'my_loop_hw_ap_fptrunc_0_no_dsp_64'...
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'C:/Xilinx/Vivado/2020.1/data/ip'.
INFO: [Common 17-206] Exiting Vivado at Wed Apr  5 23:44:13 2023...
Finished export RTL.
```

Figure 9.21: Hardware design

9.3 Integration Of the Hardware With The RNMF Algorithm

The created IP was added to the design together with the DMA as shown in Figure 9.22.

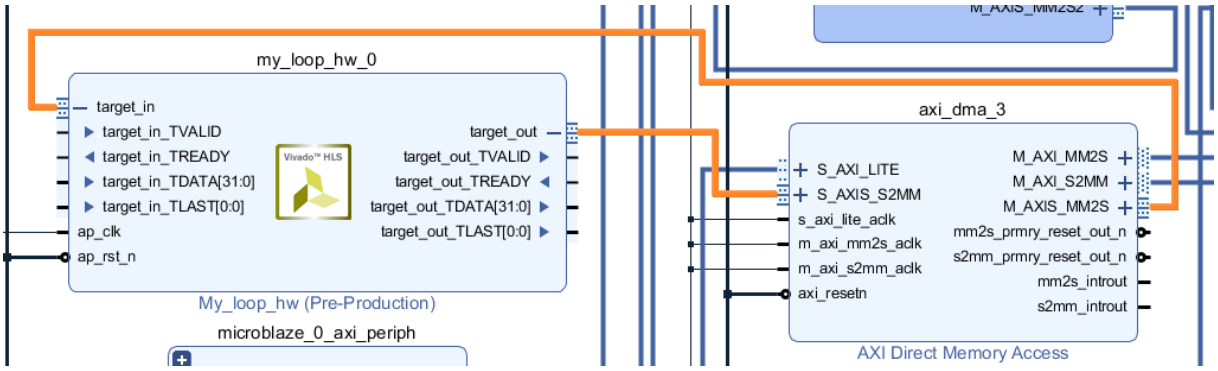


Figure 9.22: Hardware IP

In the design, it is aimed to speed up the RNMF algorithm by adding DMA and DMA of the written hardware to speed up the target data. The whole design can be seen in Figure 9.23.

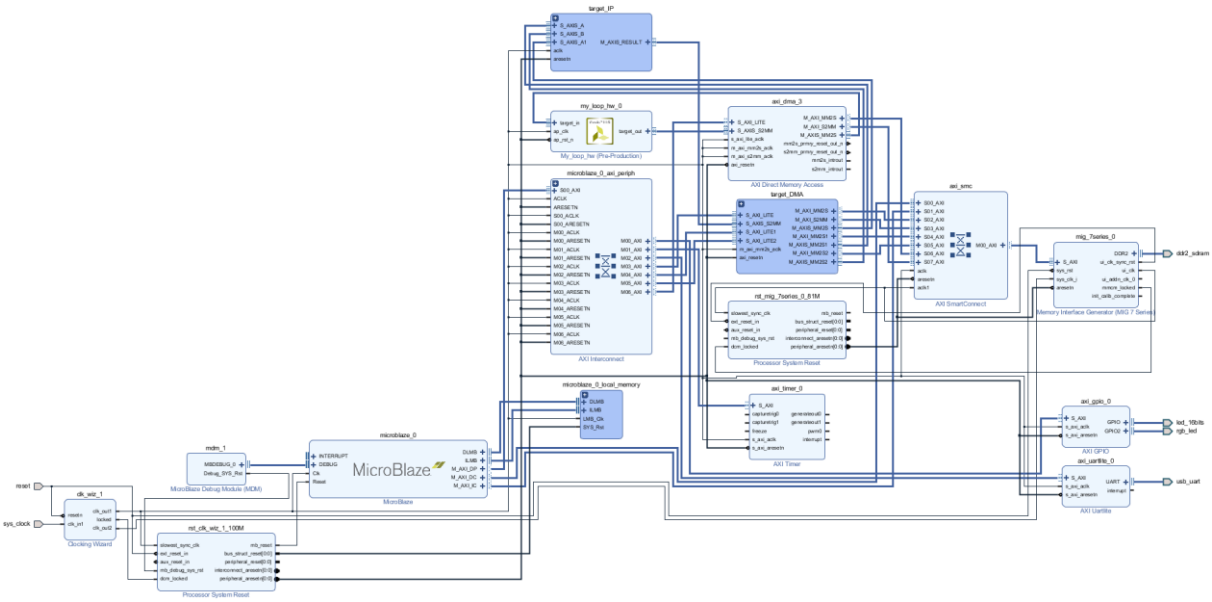


Figure 9.23: Full design

The DMA configuration and data transfer codes that will provide the connection between the manufactured hardware and DDR are shown in Figure 9.24 and Figure 9.25 in the rnmf_in.c file.

```
/*************************************************************************/
  XAxiDma my_DMA_LOOP;
  XAxiDma_Config *my_DMA_config_LOOP;
  my_DMA_config_LOOP = XAxiDma_LookupConfigBaseAddr(XPAR_AXI_DMA_3_BASEADDR);
  XAxiDma_CfgInitialize(&my_DMA_LOOP, my_DMA_config_LOOP);
/*************************************************************************/
```

Figure 9.24: DMA configuration codes

```
////////////////////////////////////////////////////////////////////////////////////////////////////
/*****************************************************************
    b_abs(target, varargin_2);
    for (target_tmp = 0; target_tmp < 46848; target_tmp++) {
      norms = varargin_2[target_tmp] - 0.00015;
      varargin_2[target_tmp] -= 0.00015;
      z1[target_tmp] = fmaxf(0.0, norms);   //double float a çevrildi
      //z1[target_tmp] = fmax(0.0, norms);
    }

    b_sign(target);
    for (i0 = 0; i0 < 46848; i0++) {
      target[i0] *= z1[i0];
    }
*****************************************************************/
////////////////////////////////////////////////////////////////////////////////////////////////////
    //writes back target from cache to DDR2 SDRAM
    //make sure that transmitted and received data arrays are stored in the DDR memory
    Xil_DCacheFlushRange((UINTPTR)target, sizeof(float) * 46848);
    //Copying float data(target) from DDR2 SDRAM to the IP
    XAxiDma_SimpleTransfer(&my_DMA_LOOP, (UINTPTR)target, 46848 * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
    //Overwrite float data(target) from IP to the DDR2 SDRAM
    XAxiDma_SimpleTransfer(&my_DMA_LOOP, (UINTPTR)target, 46848 * sizeof(float), XAXIDMA_DEVICE_TO_DMA);

    while ((XAxiDma_Busy(&my_DMA_LOOP,XAXIDMA_DEVICE_TO_DMA)) || (XAxiDma_Busy(&my_DMA_LOOP,XAXIDMA_DMA_TO_DEVICE))){/* Wait */};
    //invalidate the cache for the received data's memory range
    Xil_DCacheInvalidateRange((UINTPTR)target, 46848 * sizeof(float));
////////////////////////////////////////////////////////////////////////////////////////////////////
```

Figure 9.25 DMA data transfer codes

RNMF was set to the original iteration number of 10000 and tested. In the GPR images shown in Figure 9.28, it can be observed that the generated hardware works successfully.

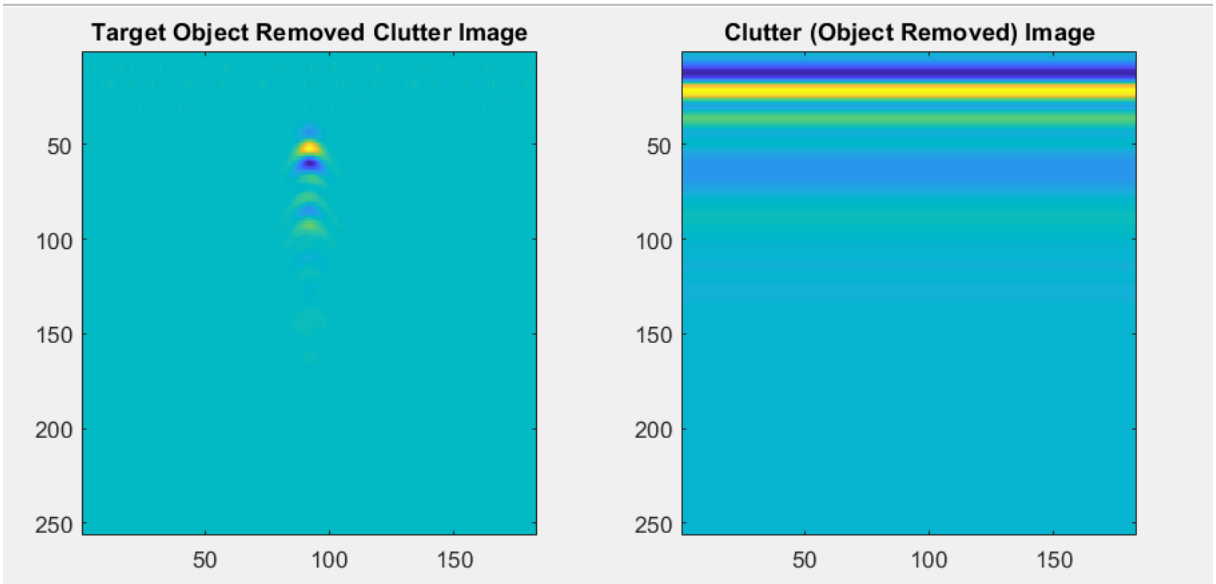


Figure 9.26: GPR images

Finally, the number of iterations was set to 2 and the speed was measured. The running time of the algorithm was 0.3531 seconds. This is considerably faster than the time before the hardware was created (1.5896 seconds), as expected.

9.4 HLS Hardware Test on Zynq-7000 SoC

Created code is tested on the ZedBoard. Design for this test can be seen on the Figure 9.29.

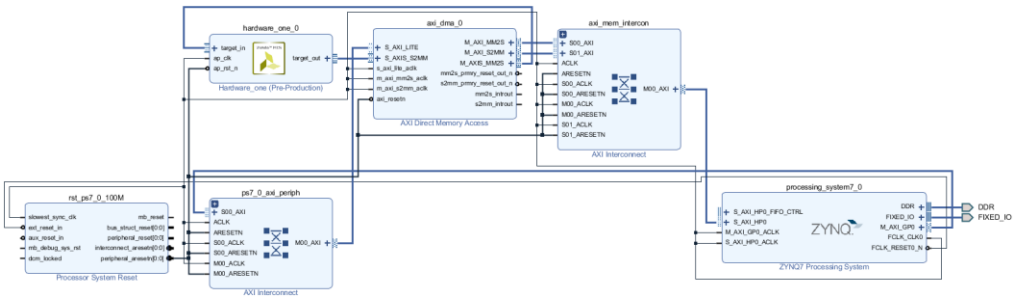


Figure 9.27: Design for ZedBoard

Operation durations are given as Table 9.1.

Table 9.1: Operation durations on ZedBoard

Optimization	Without Hardware	With Hardware
-O3	1’15”	00’38”

As it seen form the Table 9.1 newly created hardware improves performance of the RNMF algorithm drastically.

10. REALISTIC CONSTRAINTS AND CONCLUSIONS

10.1 Practical Application of this Project

The practical application of this project is its use as image processing in portable radar systems. It can be integrated into other systems where the RNMF algorithm is applied to the image data.

10.2 Realistic Constraints

Xilinx Nexys4 DDR, Zedboard FPGA devices and Vivado Software are paid. However, these were provided by the faculty laboratory.

10.3 Standards

The project was executed in adherence to the IEEE guidelines.

10.4 Health and Safety Concerns

This project does not include health and safety concerns.

10.5 Conclusion

The data delivered by the RADAR system has been effectively handled on the FPGA, and the image processing calculation has been essentially accelerated by the FPGA parallel operation capability and the DMA method. Capacity and design processes of Nexys4 DDR and Zedboard FPGA models are included. It can be foreseen that distinctive calculations other than RNMF will be quickened in FPGA systems with comparative approaches.

10.6 Future Work and Recommendations

This project focuses on the development of a solution using FPGA for object detection with GPR technology. The application area of the project covers various areas where the detection of objects hidden by obstacles is very important. We aim to improve the

overall performance of the system by addressing realistic constraints such as processing speed and data management through the use of FPGAs and custom hardware design. This project contributes to the development of image recognition capabilities and lays the foundation for further research and development in the field of GPR-based object detection.

REFERENCES

[1] ZedBoard_Configuration and Booting Guide

[2] ZedBoard User Guide

[3] https://xilinx.github.io/embeddedsw.github.io/uartps/doc/html/api/group__uartps__v3__11.html#ga60240486c69f6167ab13194ced5e8bb>

[4] https://support.xilinx.com/s/article/1053914?language=en_US

[5] AMBA® AXI™ and ACE™ Protocol Specification

[6] Vitis High-Level Synthesis User Guide