UPPSALA
UNIVERSITET

IT 18 006
Examensarbete 30 hp
Mars 2018

Hardware Accelerator of Matrix Multiplication on FPGAs

Zhe Chen

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Hardware Accelerator of Matrix Multiplication on FPGAs

Zhe Chen

To solve the computational complexity and time-consuming problem of large matrix multiplication, this thesis design a hardware accelerator using parallel computation structure based on FPGA. After function simulation in ModelSim, matrix multiplication functional modules as a custom component used as a coprocessor in co-operation with Nios II CPU by Avalon bus interface. To analyze computation performance of the hardware accelerator, two software systems are designed for comparison. The results show that the hardware accelerator can improve the computational performance of matrix multiplication significantly.

Handledare: Philipp Rummer
Ämnesgranskare: Stefanos Kaxiras
Examinator: Arnold Neville Pears
IT 18 006
Tryckt av: Reprocentralen ITC

Declaration of Authorship

I, Zhe CHEN, declare that this thesis titled, "Hardware Accelerator of Matrix Multiplication on FPGAs" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Acknowledgements

My deepest gratitude goes first and foremost to Professor Stefanos Kaxiras, my supervisor, for his constant encouragement and guidance. He has walked me through all the stages of the writing of this thesis. Without his consistent and illuminating instruction, this thesis could not have reached its present form.
Second, I would like to express my heartfelt gratitude to my coordinator Professor Philipp Rümme, who led me into the world of Embedded Systems. I am also greatly indebted to the professors and teachers of the Uppsala University, who have instructed and helped me a lot in the past years.
Last my thanks would go to my beloved family for their loving considerations and great confidence in me all through these years. I also owe my sincere gratitude to my friends and my fellow classmates who gave me their help and time in listening to me and helping me work out my problems during the difficult moment of the thesis.

Contents

List of Figures

List of Abbreviations

MPC	**M**odel **P**redictive **C**ontrol
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HPC	**H**igh **P**erformance **C**omputing
EDA	**E**lectronic **D**esign **A**utomation
VLSI	**V**ery **L**arge **S**cale **I**ntegrated Circuit
SOC	**S**ystem **O**n **C**hip
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
HDL	**H**ardware **D**escription **L**anguage
STA	**S**tatic **T**iming **A**nalysis
SOPC	**S**ystem **O**n **P**rogrammable **C**hip
IDE	**I**ntegrated **D**evelopment **E**nvironment
RTOS	**R**eal **T**ime **O**perating **S**ystems
IP	**I**ntellectual **P**roperty
PIO	**P**arallel **I**nput **O**utput
IRQ	**I**nterrupt **R**equest
PLL	**P**hase **L**ocked **L**oop
DAE	**D**ecoulped **A**ccess **E**xecute
HAL	**H**ardware **A**bstraction **L**ayer
PLD	**P**rogrammable**L**ogic **D**evice
MATLAB	**MAT**rix **LAB**oratory
GPU	**G**raphics **P**rocessing **U**nit

Chapter 1

Introduction

Real-time matrix operations are currently the most common types of operations, and they include a large number of computation operations in process control, real-time image processing, real-time digital signal processing, and network control systems. The computation performance directly affects system performance. At present, most of the matrix operations are implemented by software. With the growth of the matrix dimension, the speed of the processing is becoming significantly slower. For example, in model predictive control (MPC) applications, especially in the embedded systems, matrix operations usually take the largest amount of computation, and these operations are offen highly time-consuming [1] [2]. In the automotive industry, an MPC system requires a high level of real-time calculation of the matrix. Real-time matrix calculation performance becomes the bottleneck of MPCs in fast system applications. Therefore, it is meaningful to study matrix operations in these context and the research would be done using a hardware accelerator that is based on field programmable gate array (FPGA).

In recent years, many scholars have conducted research to improve the computational performance of matrix operations. The research directions are mainly divided into two areas. One area involves starting from the side of multi-processor parallel computing in order to improve the computing speed of matrix operations; The other area is taken from the hardware implementation point of view, and it involves designing the hardware structure to achieve matrix parallel computing to improve computing performance.

This thesis focuses on the second area, namely in implementing the hardware accelerator for matrix multiplication based on a Nios II embedded soft core on FPGA, which achieves the parallel computing. The hardware accelerator mainly includes two functional modules—-task logic module and interface

module. The functional modules are custom components connected on an Avalon Bus to communicate with the Nios II processor. ModelSim is used for conducting the simulation. The result shows that the design is feasible and offers high computational performance.

Chapter 2

Background

2.1 Matrix Multiplication

Matrix operations are indispensable tools used for describing the mathematical relationships in many engineering problems. Matrix multiplication is the most basic mathematical tool in signal processing and image processing. This section focuses on the basic concepts of matrix multiplication and discusses a suitable way approach to FPGA implementation.

According to the definition, if A is an $\boldsymbol{n}$ x $\boldsymbol{m}$ matrix and B is an $\boldsymbol{m}$ x $\boldsymbol{p}$ matrix,

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix} \tag{2.1}$$

The matrix multiplication **AB** is defined to be the $\boldsymbol{n}$ x $\boldsymbol{p}$ matrix

$$\mathbf{AB} = \begin{pmatrix} AB_{11} & AB_{12} & \cdots & AB_{1p} \\ AB_{21} & AB_{22} & \cdots & AB_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ AB_{n1} & AB_{n2} & \cdots & AB_{np} \end{pmatrix} \tag{2.2}$$

where each i and j entry is given by multiplying the entries $\mathbf{A}_{ik}$ (i.e., across row i of **A**) by the entries $\mathbf{B}_{kj}$ (i.e., down column j of **B**).
For k = 1, 2, ..., m, and summing the result over k:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} \mathbf{A}_{ik}\mathbf{B}_{kj} \tag{2.3}$$

The computational complexity of this algorithm is $\boldsymbol{n}$ x $\boldsymbol{m}$ x $\boldsymbol{p}$, that is, $\mathbf{O}$ (n^3). Ofter times when the algorithm is implemented on a microprocessor or a single chip microcomputer, it is both serial and inefficient. Based on this algorithm, we implement a parallel computing method based on FPGA to improve the computing performance.

The parallel architecture of matrix multiplication includes five parts: (a) memory modules for storing matrices A and B respectively, (b) registers, (c) cache, (d) high performance computing (HPC) multiplier and (e) HPC accumulator [3]. The HPC multiplier and accumulator compute in parallel, which this configuration can greatly improve the speed of matrix multiplication. The structure of parallel matrix multiplication is shown in Figure 2.1.

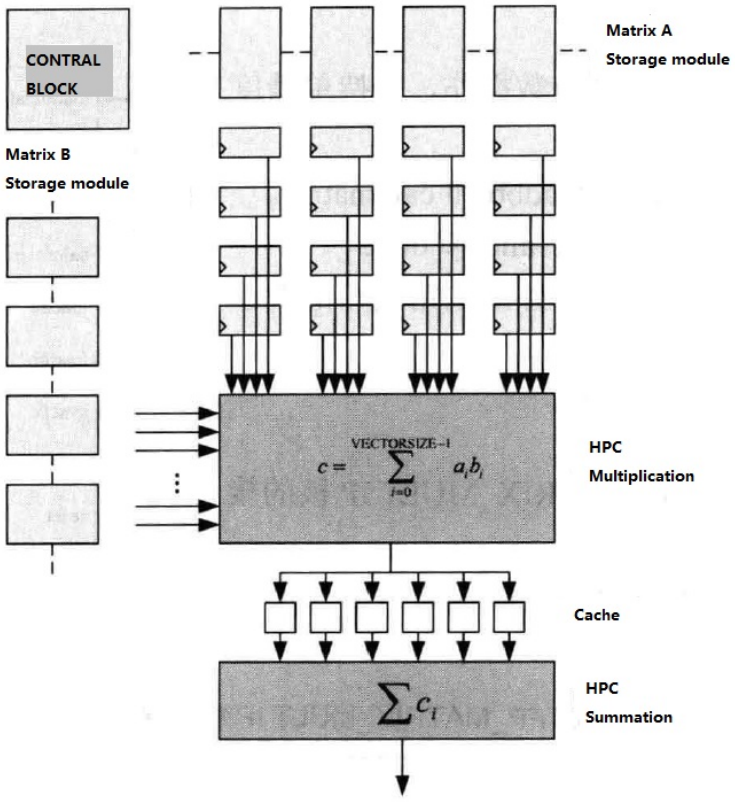

FIGURE 2.1: Parallel architecture of matrix multiplication

The matrix A is loaded from the memory block;the matrix is then stored in the registers and computed by the HPC multiplier with matrix B loaded from another memory block. After this, the output of the HPC multiplier is written to the Cache. When all elements of a row of the matrix are written to the Cache, the HPC adder calculates these elements as the output matrix. The results of the matrix are output row by row.

In order to compute the matrix in parallel, we need to decompose the rows of A and columns of B are divided into sub-rows and sub-columns respectively, each containing ***VECTOR_SIZE*** elements. For example, matrix A_{4x4}, B_{4x4}, ***VECTOR_SIZE*** = 2, the result matrix C_{4x4}:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{2.4}$$

From the above, we can obtain the following:

$$\begin{aligned} C_{11} &= A_{11} * B_{11} + A_{12} * B_{21} \\ C_{12} &= A_{11} * B_{12} + A_{12} * B_{22} \\ C_{21} &= A_{21} * B_{11} + A_{22} * B_{21} \\ C_{22} &= A_{21} * B_{12} + A_{22} * B_{22} \end{aligned}$$

where $A_{ij}, B_{ij}, C_{ij}(i, j = 1, 2)$ are all 2 x 2 matrices.

As we can see, the calculation changes from two -4 x 4 matrix multiplication to eight -2 x 2 matrix multiplication and four -2 x 2 matrix addition. The eight -2 x 2 matrix multiplication can be calculated in parallel.

2.2 FPGA

2.2.1 Introduction of FPGA

FPGA is a digital integrated circuit designed to be configured by a customer or a designer after manufacturing. Hence, "field - programmable" device, is one of the most popular programmable devices available today.

With the improvement of electronic design automation (EDA) technology and microelectronics technology, FPGAs can run at speeds approaching the GHz level. Coupled with the ease of design in parallel processing and large

data processing, FPGAs can be applied in a wide range of high-speed, real-time monitoring, and control fields. With its high integration and high reliability, FPGA can integrate the entire design system in the same chip–system-on-chip (SOC)–which greatly reduces its size.

2.2.2 Characteristic of Modern FPGA

Modern FPGA exhibits a number of characteristics:

- **Large scale and short period of development**
  With the continuous improvement of Very Large Scale Integrated Circuit (VLSI) technology for an integrated circuit chip, it would be possible for such a chip to integrate hundreds of millions of transistors.Moreover, the scale of integration within an FPGA is also increasing. With a larger chip size, it has a more powerful performance. It is also more suitable to achieve having a SOC. On the other hand, FPGA design is quite flexible. It can shorten the period of development.
- **Small investment in the development process**
  FPGA chips are tested before manufacturing. When an error is found, users can directly change the design. This can reduce the risk of investment and save money. In this way, many complex systems could be realized using FPGA. Even the design of an application specific integrated circuit (ASIC) also need to implement FPGA for verification as an essential step.
- **Good security**
  According to the requirements, the anti-reverse FPGA technology can be selected; this funching can protect the security of the system and the designer's intellectual property.

2.2.3 FPGA Design Flow

Hardware description language (HDL) is a specialized computer language; it enables a formal description of structure and behavior of digital circuit logic systems. Designers can use the HDL to describe their own design idea, and they can then compile the program with EDA tools in order to synthesize them into gate-level circuits. In this way, the design functions can finally be completed using FPGA.

Verilog HDL is a type of HDL, commonly used in design and verification of digital electronic system at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits.

With the growth of integrated circuit technology, it is difficult for a designer to independently design an ultra-large scale system. In order to reduce the risk caused by design errors, it is possible to use a hierarchical structure design idea called modular design to solve these problems. Modular design is a method that divides the total system into a number of modules.

The FPGA design includes the following main steps [4]:

- **Circuit design and input**
  This step describes using circuit logic to achieve certain functions. The common methods for this step are HDL and schematic design. The schematic design is based on determining the requirements of the chip selection and drawing the schematic to connect them all. It is used widely in the early years and the advantage of this approach is that it is easy to understand. However, in large-scale designs, this method has poor maintainability and a huge workload;both these shortcomings are not ideal conditions for construction and modification. Moreover, the main disadvantage is that: when the selected chip is upgraded, all the schematics have to change correspondingly. Therefore, the most commonly used method is the HDL, especially in large-scale projects. VHDL and Verilog are two widely used HDL; their common feature is the use of top-down design, which is made for modular design and easy modification. They also have good portability and universality because the design is not changed due to the changes of technology and structure on the chip.

- **Functional simulation**
  After the circuit design is completed, it is necessary to simulate the system using with a dedicated simulation tool to verify whether the circuit function meets the design requirements. Functional simulation is sometimes called pre-simulation. Some common simulation tools include ModelSim, VCS, NC-Verilog and NC-VHDL, and Active HDL/ VHDL/ Verilog HDL. Through the simulation, errors could be more easily found and modified in time in order to improve the reliability of the design.

- **Synthesis**
  Synthesis is used for translating the circuit design into a logical netlist

consisting of basic logic elements such as AND, NOT, RAM, and flip-flops. It could also optimize the logical connection according to the requirements (or rules). It will create **.edf** and **.edn** files to produce a layout for manufacturers.

- **Synthesis simulation**
  Synthesis simulation is used after synthesis in order to check whether the result is consistent with the original design. Use of the delay file from synthesis into synthetic simulation model can estimate the impact of the gate delay. Although the synthesis simulation is more accurate than the function simulation, it can only estimate the gate delay but not the wire delay. There is still a difference between the simulation results and the real situation after routing. The main purpose of this simulation is to check whether the result after synthesis is the same as the design input.

- **Implementation (routing)**
  The essence of the result after synthesis is a logic netlist composed of basic units–such as NAND gate, flip-flop, and RAM–which can still be very different with the real configuration of the chip. At this point, we use the FPGA software provided by the manufacturers. To do this, we select the type of chip, and the software configures the logic netlist into the specific FPGA device. This process is referred to as implementation. Altera divided implementation process into other sub-step–some main steps include translate, map, place and route. Since only the device manufacturer knows the internal structure of the device, we are restricted to the software provided by the device manufacturer.

- **Routing simulation**
  After implementation, timing simulation is the next step needed. The delay file from the implementation should be included into the design. The timing simulation includes both the gate delay and the wire delay. Compared with the previous simulation, the information on the delay is the most comprehensive and accurate, and this information can provide a clearer picture regarding the real situation of the chip. Moreover, routing simulation has more verification after the timing simulation in order to ensure the reliability of the design sometimes. For example, TimeQuest Timing Analyzer can be used for completing the static timing analysis (STA). Third-party verification tools can also be used for observing the connection and configuration within the chip.

In some high-speed designs, third-party board-level verification tools are also required for simulation and verification.

- **Debugging**
  The final step is to debug or write the generated configuration file to the chip for testing. The corresponding tool in Quartus II is SignalTap. If there are any problems found within the simulation or verification, the developer needs to return to the corresponding step to change or redesign the system.

2.2.4 Verilog HDL

Verilog HDL is a hardware description language that models digital systems with a variety of abstract design levels from the algorithmic level, and gate level to switch level [5]. The complexity of a modelled system can range from a simple gate to a complete electronic digital system. The system can be described hierarchically, and at the same time it follows the timing sequence.

In this thesis, Verilog is used as main HDL because it has more support available by third-party tools more. Moreover, the syntax structure is also simpler than other HDLs because Verilog has more in common with the C language than other languages. In addition, the simulation tool of Verilog HDL is easy to use, and the corresponding stimulus test module is easy to write.

Module is the most basic and important concept in Verilog. Each Verilog design system consists of several modules. The following are the basic features of the module:

- The module is a program that starts with the keyword module, and it ends with the keyword endmodule.
- The module represents the logical entity on the actual hardware circuit, and achieves it a specific function.
- The modules are run in parallel.
- The module is hierarchical: at a high-level module, a complex function can be achieved by calling the connected low-level modules. A top-level module is needed in order to complete the entire system

An example of a module implementing a 3-bit full adder is shown in Figure 2.2.

```
module adder(count,sum,a,b,cin)
/*
 * The port of the 3-bit full adder
 */
    input[2:0] a,b;
    input cin;
    output cout;
    output[2:0] sum;
/*
 * The function of the 3-bit full adder
 */
    assign {cout,sum} = a+b+cin;
endmodule
```

FIGURE 2.2: Design of 3-bit full adder using Verilog

From the example, it can be seen that everything between the two keywords module and endmodule defines the details of the module, and adder is given as the name of the module. The code after the first comment is the port declaration–which in this case involves two inputs and two outputs–and after the second comment the function given is to add.

The advantage of designing a complex digital logic system using Verilog is that the logic function of the circuit is easy to understand and it is convenient for the computer to analyze and process. In the design, the logic design and implementation can be divided into two separate phases in order to operate. In this way, the logic design is not dependent on the technology, and it can be reused for different chips. In addition, the idea of modular design makes the complex logic circuit design easy to be completed by different people.

2.2.5 System on a Programmable Chip (SOPC)

SOPC technology was first proposed by the Altera Company, and it is a SOC design solution based on FPGA. It integrates the processor, I/O port, and memory, and it requires functional blocks put into a single FPGA to implement a programmable SOC. SOPC technology is a new and comprehensive electronic design technology that is created in order to implement electronic systems as large and integrated as possible. This make it the important achievement in modern computer application technology and also an important development direction of modern processor applications. The core of SOPC design is the Nios II soft core processor; this design includes the hardware configuration, hardware design, hardware simulation, software design, and software debugging. The

typical SOPC System is shown in Figure 2.3, and it is mainly divided into three parts–the processor, the Avalon Bus, and customizable peripherals.

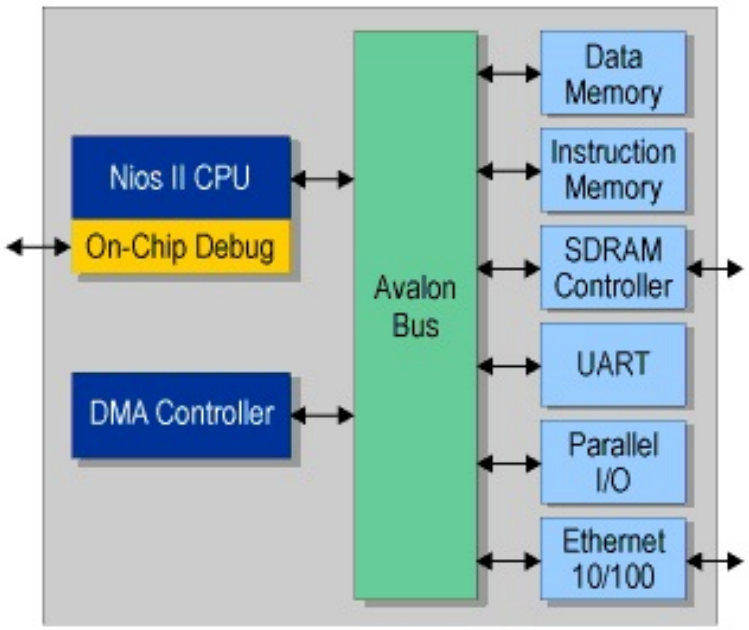

FIGURE 2.3: The typical SOPC System [6]

- **Processor**
  The processor here is a Nios II soft core. The Nios II processor is developed by Altera Company, and it is the second generation of on-chip programmable soft-core processors with 32-bit. The basic structure of a 32-bit processor consists of a 32-bit instruction size, 32-bit data and address paths, 32 general purpose registers, and 32 external interrupt sources. The Nios II processor has a complete software development kit, including a compiler, integrated development environment (IDE), JTAG debugger, real-time operating system (RTOS), and TCP / IP stack. Designers can easily create a customizable processor using the SOPC Builder in Altera's Quartus II software; they can also easily add the number of Nios II processors to their system according to its needs.

- **Avalon Bus**
  The Avalon bus, developed by ALTERA Company, is an interface of the on-chip processor and the on-chip peripherals in an FPGA-based SOPC. An Avalon bus interface can be divided into two categories-master and slave. The main difference between the two is the control of Avalon bus. The master interface has Avalon bus control, while

the slave interface is passive. The way of communication between the master and slave interfaces is shown in Figure 2.4.

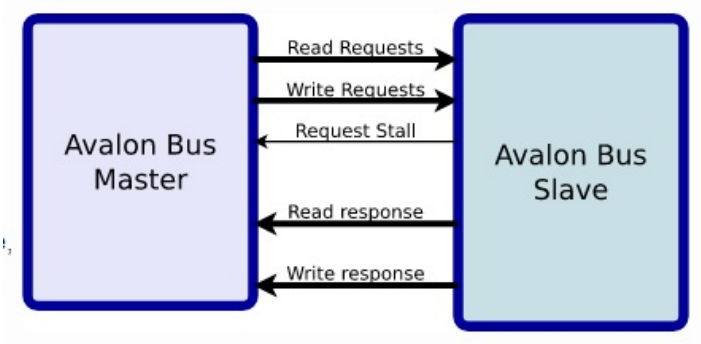

FIGURE 2.4: The way of communication between Avalon master and slave [7]

The Avalon bus is automatically generated and adjusted when peripherals are added or deleted in a SOPC builder. After this, the optimal structure of Avalon bus for peripheral configuration is created. If the users only use the customize peripherals that are already in the software–in other words, they meet Avalon bus specification–the users do not need to know the details of the Avalon bus specification. However, for the users designing their own peripherals, peripherals must meet the Avalon bus specification. Otherwise, these cannot be integrated into the system.

- **Peripherals**
  Here we introduce several peripherals that are commonly used. The SOPC builder provides cores for each peripheral called intellectual property (IP) cores so that they can be easily integrated in the SOPC system [8].
  - Parallel I/O (PIO)
    The PIO core provides an interface between an Avalon slave port and the general-purpose I/O port. The I/O port connects either to on-chip user logic or to I/O pins that connect to devices external to the FPGA. Some examples of its use include controlling LEDs, acquiring data from switches, controlling display devices, and configuring and communicating with off-chip devices.

- SDRAM Controller
  The SDRAM controller core provides an Avalon interface to off-chip SDRAM. The SDRAM controller allows designers to create custom systems in an FPGA that connects easily to SDRAM chips. The core can access SDRAM subsystems with various data width (i.e. 8, 16, 32, or 64 bits), various memory sizes, and multiple chip selects.
- Timer
  The timer core is a timer for Avalon-based processor systems, such as a Nios II processor system. The timer provides the following features:
  - 32-bit and 64-bit counters;
  - controls to start, stop, and reset the timer;
  - two count modes, namely count down once and continuous count-down;
  - an option to enable or disable the interrupt request (IRQ) when the timer reaches zero;
  - optional watchdog timer feature resets the system if the timer ever reaches zero;
  - optional periodic pulse generator features that outputs a pulse when the timer reaches zero; and
  - compatible with 32-bit and 16-bit processors.
- Phase locked loop (PLL)
  The PLL core provides a means of accessing the dedicated on-chip PLL circuitry in FPGAs. The PLL core is a component wrapper around the Altera ALTPLL megafunction. The core takes a SOPC builder system clock as its input and generates PLL output clocks locked to that reference clock. PLL output clocks are made available in two ways–either as sources of system-wide clocks in your SOPC Builder system or as output signals on your SOPC Builder system module.
- System ID
  The system ID core provides the interface of read-only Avalon slave with a unique identifier. The Nios II processor uses the system ID core to verify that an executable program was compiled

targeting the actual hardware image configured in the target FPGA. If the expected ID in the executable does not match the system ID core in the FPGA, it is possible that the software would fail to execute correctly.

2.3 Decoupled Access-Execute (DAE) Architectures

DAE architecture is a type of architecture which separates the processing into two parts–namely access to memory to fetch the store results; and the data execution to produce the results [9]. Through architecturally decoupling data from execution, this can save the time of loading data. In other words, both an access data process and an executed data process can run at same time. The idea of DAE architectures is shown in Figure 2.5.

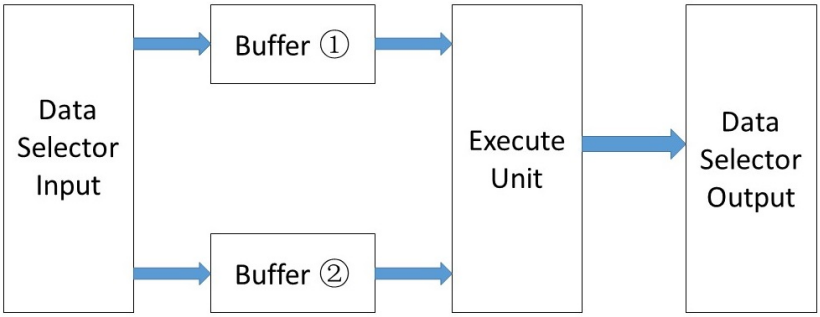

FIGURE 2.5: Decoupled access-execute architecture

The input data through the multiplexer allocated the data into one of two buffers. The buffer can be any type of storage module, such as DPRAM or FIFO.

- In the first stage the input data is allocated into buffer1 using the control of the input data selector
- In the second stage, the input data is allocated into buffer2 by switching the input data selector. At the same time, the data already inside buffer 1 is transferred into the execute unit for computing.
- In the third stage, the input data is allocated into buffer1 by switching the input data selector again. At the same time, the data already inside buffer2 is transferred into the execute unit for computing.

The above three stages are repeated.

The biggest advantage of this architecture is that the input and output multiplexers switch cooperatively, and they transfer the data to the execute unit for processing without stopping. In viewing the design as a whole, and looking at both ends of the design to see the data, it can be noted that both the input and output data are continuous. Therefore, this architecture is ideal for pipelined algorithms.

The disadvantage of this design is that the need of resources would increase. For example, two buffers are needed at least. If resources are limited, it would be necessary to balance the resource with speed.

This is quite useful especially when the execute process is very short, particularly if it is even shorter than the access process. In such a case, users can prepare the data for the execute process in order to save the waiting time.

Chapter 3

Design and Implementation Hardware Accelerator

In this thesis, the term hardware accelerator refers to implementing the user-customized logic function module using FPGA, and then using this to communicate with other modules through the Avalon bus. When numerous mathematical operations are calculated in hardware, they are faster and more efficient than in software. Therefore, we use a hardware accelerator to improve system performance.

More specifically, a matrix multiplication hardware accelerator is built in this thesis. The design of matrix multiplication hardware accelerator includes mainly two parts which are hardware and software. The hardware design includes the task logic function module and interface module which are all written in Verilog HDL. The software part consists of hardware abstraction layer (HAL) peripherals, which including corresponding C language header files and source files.

3.1 Development Tool

Two tools are used in this thesis, namely Quartus II which is used for development and ModelSim which is used for simulation.

3.1.1 Quartus II

Quartus II software is an integrated, proprietary development tool designed for Altera's FPGA chips, and it is the newest generation of more powerful

and more integrated EDA development software. Using Quartus II can complete the design process from generating the design entry, synthesis, simulation to downloading the design to hardware.A Quartus II integrated environment includes: a system-level design, an embedded software development, programmable logic device (PLD) design, synthesis, layout and wiring, verification and simulation. Quartus II can also directly assign Synplify Pro, ModelSim and other third-party EDA tools to complete the design of the synthesis and simulation tasks. In addition, it can be combined with an SOPC builder in order to achieve the development of an SOPC system.

Quartus II provides an easy-to-use graphical user interface to complete the entire design flow [10]. The general process of design flow is shown in Figure 3.1.

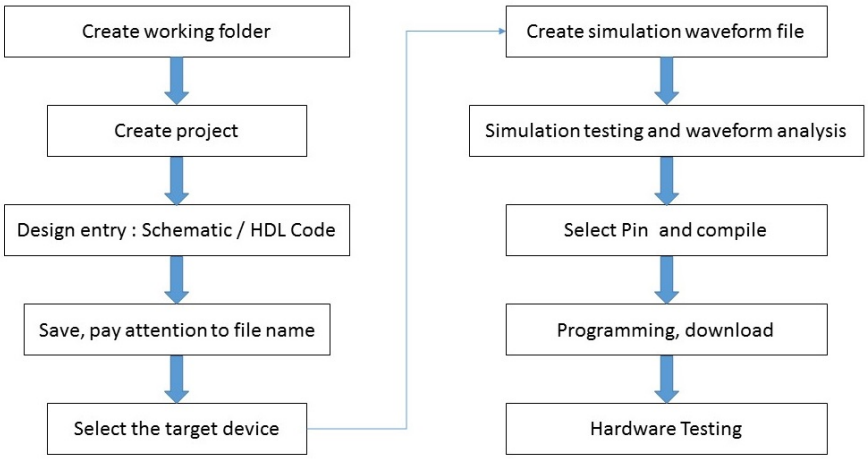

FIGURE 3.1: The general design flow of using Quartus II

3.1.2 ModelSim

Developed by Menter Company, ModelSim is one of the best HDL language simulation software.It provides an efficient simulation environment. and it is the only simulator that single-core supporting both VHDL and Verilog mixed simulations. It compiles the program and emulates it quickly because it uses optimization of the compiler technology, Tcl / Tk technology, and single core simulation technology [11] . The compiled code is platform-neutral and is

good for protecting the IP core. It also has an intelligent, user friendly, and easy-to-use graphical user interface, which is convenient for users to debug.

This thesis use the ModelSim Altera version, which is a special version for Altera device. It can be used directly from the Quartus II after users set the ModelSim Altera as the simulator.

3.2 Design of Hardware Accelerator

The matrix multiplication hardware accelerator includes two functional modules–TMATRIX Task Logic module and the Avalon Slave and Avalon Master interface module. The modules are made as custom components that hang onto the Avalon bus, namely as the Nios II CPU hardware accelerator. The structure of a matrix multiplication accelerator is shown in Figure 3.2.

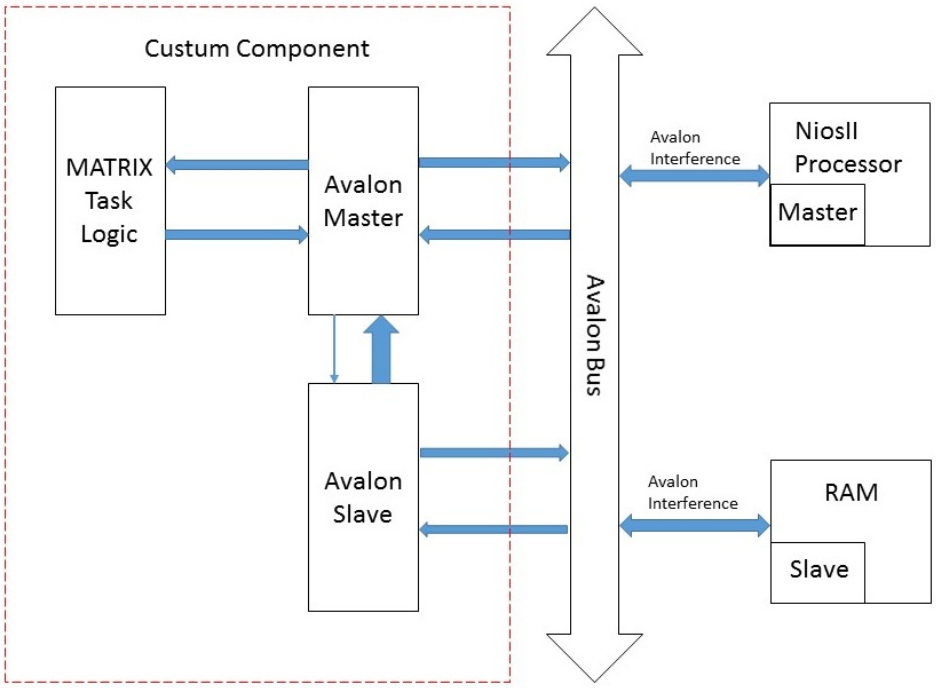

FIGURE 3.2: The structure of a matrix multiplication accelerator

3.2.1 MATRIX Task Logic

A task logic function module is used for achieving the matrix multiplication function. In this thesis, the task logic function module is design for two - 8 x 8 matrix multiplication. In order to accelerate this calculation process better, DAE design has been added to the design. As mentioned in the thesis background, we decompose an 8 x 8 matrix into sixteen -2 x 2 matrices, and thus the resulting matrix should also be sixteen -2 x 2 matrices. In this thesis, we used 16 sub-modules to calculation each 2 x 2 matrix of the result matrix. The function of each sub-module is the same because the output is decided by the different input. The hierarchy of the matrix task logic module is shown in Figure 3.3.

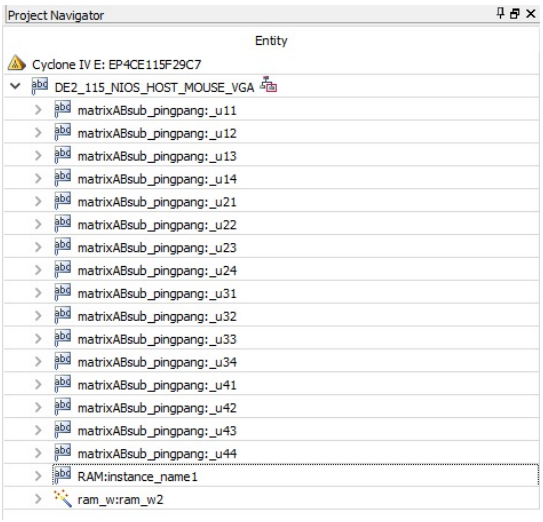

FIGURE 3.3: The hierarchy of the matrix task logic module

For example, the two matrices that were entered are A and B, and the result is matrix C. The size of A, B and C are 8 x 8. It is assumed that the inputs are the first two rows of A and the first two columns of B. For each sub-module, the design is shown in Figure 3.4.

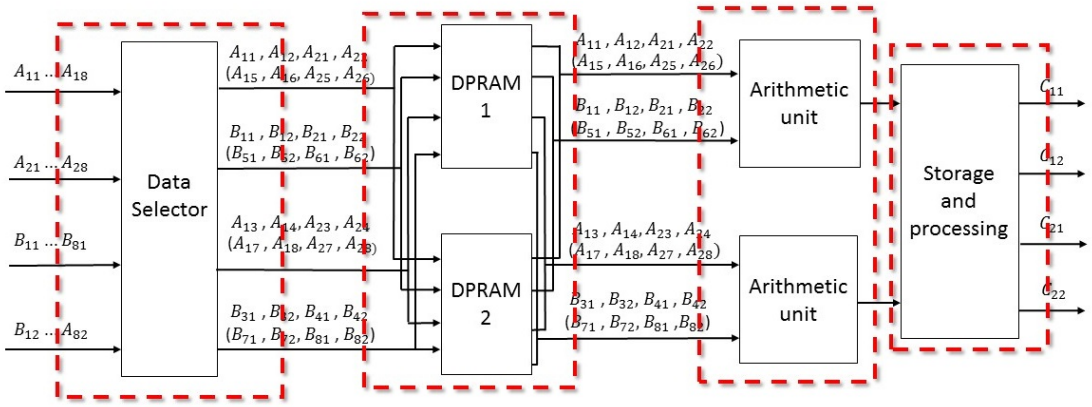

FIGURE 3.4: Design of the multiplication sub-module

The whole design is divided into four parts with the red dotted boxes. From left to right, these parts are data select, buffer, calculation, and processing and output respectively.

- **Data Select**
  The data selector chooses the suitable date to output to the next step– that is, an individual 2 x 2 small matrix. The input is the first two rows of A and the first two columns of B, both A and B can be divided into four -2 x 2 matrix, totalling eight matrices. The calculation process later will treat 2 x 2 matrix as a unit, and thus the data selector will output twice, each time with four matrices, including the two matrices from A and two from B.

- **Buffer**
  As mentioned above, two DPRAMs are used as the buffer to implement the DAE design. The input of the DPRAM is the output of the data selector–namely four -2 x 2 matrix or a value of 0. The output of the DPRAM is the data of the input or 0. These two DPRAMs work with each other using two Boolean signals: one control writes to the DPRAM, the other control reads from DPRAM. If the write signal is 0, the data is written to DPRAM1 and the value of 0 to DPRAM2. When the next clock comes, the write signal becomes 1, the data is then written to DPRAM2 and 0 to DPRAM1. The read signal is the same design as write. Thus, the total output to the next step is simply to add the output of the two DPRAMs and pass the data to the next step.

- **Calculation**
  This process is to calculate the multiplication of matrix. As seen from the previous step, the DPRAM passes four -2 X 2 matrices each time. Because of this, two arithmetic units are used here, and the function of these two units is the same. Each one is used to calculate the multiplication of two -2 x 2 matrix–one from A and one from B. The algorithm is shown in Figure 3.5, the output of the arithmetic unit is also a 2 x 2 matrix. The result is passed to the next step.

```
else begin
            r1  <= i_A11*i_B11+i_A12*i_B21;
            r2  <= i_A11*i_B12+i_A12*i_B22;
            r3  <= i_A21*i_B11+i_A22*i_B21;
            r4  <= i_A21*i_B12+i_A22*i_B22;
   end
assign o_C1 = r1[15:0];
assign o_C2 = r2[15:0];
assign o_C3 = r3[15:0];
assign o_C4 = r4[15:0];
```

FIGURE 3.5: Calculation algorithm of the 2x2 Matrix

- **Processing and Output**
  This process is used firstly for storing the preliminary results of the calculation part that is split into output twice, each time as two -2 x 2 matrices. The two are then added together as the final output because the resulting matrix C needs to add the matrices that were previously multiplied. The add algorithm is shown in Figure 3.6.

```
input[15:0]w_C1_11;  input[15:0]w_C2_11;  input[15:0]w_C3_11;  input[15:0]w_C4_11;
input[15:0]w_C1_12;  input[15:0]w_C2_12;  input[15:0]w_C3_12;  input[15:0]w_C4_12;
input[15:0]w_C1_21;  input[15:0]w_C2_21;  input[15:0]w_C3_21;  input[15:0]w_C4_21;
input[15:0]w_C1_22;  input[15:0]w_C2_22;  input[15:0]w_C3_22;  input[15:0]w_C4_22;

output[15:0]o_C11,o_C12,
            o_C21,o_C22;

assign o_C11=w_C1_11+w_C2_11+w_C3_11+w_C4_11;
assign o_C12=w_C1_12+w_C2_12+w_C3_12+w_C4_12;
assign o_C21=w_C1_21+w_C2_21+w_C3_21+w_C4_21;
assign o_C22=w_C1_22+w_C2_22+w_C3_22+w_C4_22;
```

FIGURE 3.6: Add Algorithm of the 2x2 Matrix

The real design of this submodule from Quartus II is shown in Figure 3.7.

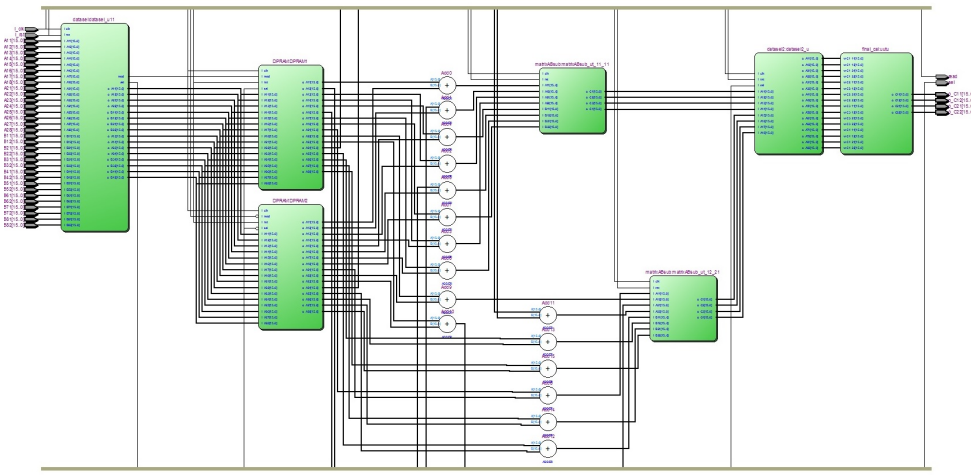

FIGURE 3.7: The design of submodule from Quartus II

After passing through this submodule, the output is the 2 x 2 matrix of the top left corner of the resulting matrix C. Thus, 16 same-function sub-modules are used for implementing the whole MATRIX task logic module to calculate the resulting matrix C. The MATRIX Task Logic module is a user-custom module which be connected on Avalon Bus.

3.2.2 Avalon Master and Slave

- The Avalon Master interface module is used for reading the data from RAM; it then passes the data to the MATRIX task logic module and calculate it, after which it writes the result back to RAM.

- The Avalon Slave interface module receives the address and control word sent from the Nios II CPU and it pass the information it to the Avalon Master. At the same time, the Avalon Slave also read the status word from MATRIX Task Logic in order to pass to the Nios II CPU.

  This part is implemented in the SOPC builder by connecting the corresponding line between each module. After that, the system generates the optimal structure automatically.

3.2.3 RAM

The RAM here is an on-chip memory that is used storing the matrix computing data, and it stores back the result data after. In this design, we used three RAM IP cores. One is called ram_a for storing the matrix A; one is called ram_b for storing the matrix B, and another is called ram_w for storing the resulting matrix. Both ram_a and ram_b are already storing the data by a memory initialization file generated by MATLAB. The logic of reading the data from RAM and writing back are shown in Figures 3.8 and 3.9.

```
integer j;
reg[5:0]men_addr[3600:1];
always @(posedge i_clk or posedge i_rst)
begin
    if(i_rst)
        begin
            for(j=1;j<=3600;j=j+1)
            men_addr[j]<=6'd0;
        end
    else begin
            men_addr[1]<=i_read_addr;
            for(j=2;j<=3600;j=j+1)
            men_addr[j]<=men_addr[j-1];
        end
end
```

```
//Read data from ram_a and ram_b
ram_a ram_a_u(
        .aclr   (i_rst),
        .address (men_addr[3600]),
        .clken  (1'b1),
        .clock  (i_clk),
        .rden   (1'b1),
        .q      (Aout)
        );
ram_b ram_b_u(
        .aclr   (i_rst),
        .address (men_addr[3600]),
        .clken  (1'b1),
        .clock  (i_clk),
        .rden   (1'b1),
        .q      (Bout)
        );
```

FIGURE 3.8: The logic of read data from RAM

```
Reg[5:0] o_Waddr;
always @(posedge CLOCK_50 or negedge reset_n)
begin
    if(~reset_n)
        begin
            o_Waddr <= 6'd0;
        end
    else begin
            if(cnt == 3'd0);
            o_Waddr <= o_Waddr + 6'd1;
        end
end
```

```
//Write data to ram_w
ram_w ram_w2(
        .aclr   (~reset_n),
        .address (o_Waddr),
        .clock  (CLOCK_50),
        .data   (o_sum),
        .wren   (o_En),
        .q      ()
        );
```

FIGURE 3.9: The logic of Write data back to RAM

3.2.4 Nios II Processor

A Nios II embedded processor soft core, sends instructions and reads MATRIX Task Logic working status. The software flow corresponding to

the hardware accelerator is shown in Figure 3.10.

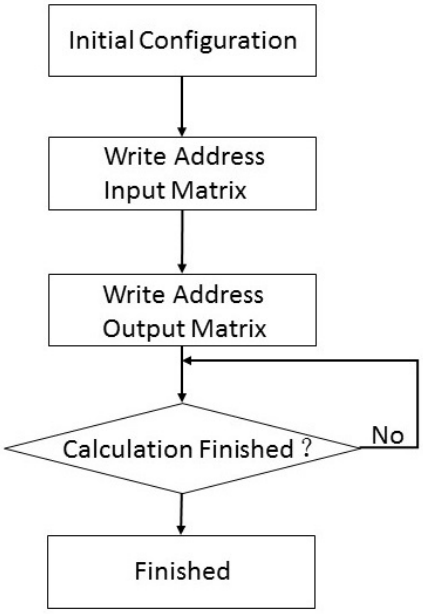

FIGURE 3.10: Software flow of the hardware accelerator

The software flow includes the following steps:

- Step 1: System initialization configuration and initialization of read and write SOPC core begins the process.
- Step 2: Nios II CPU writes the storage address of the input and output matrices in RAM to the hardware accelerator;
- Step 3: When the address changes, the hardware accelerator start;
- Step 4: When the hardware accelerator calculation is completed, an enable signal is generated to determine if the calculation is finished or not.

The important program need to be written here is the one that generates the address. The related software configurations are shown in Figure 3.11. After these programs are configured, the SOPC top-level module generates the address.

```
unsigned int dc_r16(unsigned char reg_no)
{
 unsigned int result;

 outport(dc_com, reg_no);
 uSDelay(10);
 result=inport(dc_data);

 return(result);
}

void dc_w16(unsigned char reg_no, unsigned int data2write)
{
 outport(dc_com,reg_no|0x80);
 uSDelay(10);
 outport(dc_data,data2write);
}
```

```
unsigned long dc_r32(unsigned char reg_no)
{
 unsigned int result_l,result_h;
 unsigned long result;

 outport(dc_com, reg_no);
 uSDelay(10);
 result_l=inport(dc_data);
 result_h=inport(dc_data);

 result = result_h;
 result = result<<16;
 result = result+result_l;

 return(result);
}

void dc_w32(unsigned char reg_no, unsigned long data2write)
{
 unsigned int low_word;
 unsigned int hi_word;

 low_word=(data2write)&0x0000FFFF;
 hi_word=((data2write)&0xFFFF0000)>>16;

 outport(dc_com,reg_no|0x80);
 uSDelay(10);
 outport(dc_data,low_word);
 outport(dc_data,hi_word);
}
```

FIGURE 3.11: The Nios II software configuration

The core of the whole hardware accelerator design is the finite state machine design. A finite state machine allows the various functional modules to work in sequence so that the entire system can correctly read and write the matrix data and calculate it. The finite state machine (matrix multiplication as an example) designed in this thesis is shown in Figure 3.12.

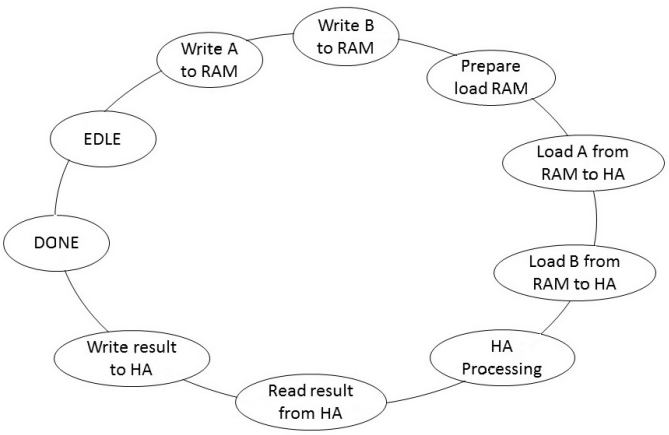

FIGURE 3.12: State transition diagram of finite state machine

In a total of ten states, each state is changed to the next based on the input conditions and the existing state. After this, the corresponding tasks are executed.

Chapter 4

Simulation and Analysis

4.1 ModelSim Simulation

After the design of the matrix hardware accelerator is finished, ModelSim simulated the system in order to verify that the designed function is correct. In this thesis, we took two -8 x 8 matrices as an example input to finish the simulation. The value of A and B are as follows:

$$A = \begin{pmatrix} 42 & 40 & 42 & 88 & 96 & 99 & 29 & 57 \\ 72 & 54 & 56 & 89 & 53 & 75 & 13 & 15 \\ 0 & 42 & 14 & 9 & 69 & 28 & 2 & 59 \\ 30 & 69 & 20 & 4 & 32 & 79 & 68 & 70 \\ 15 & 20 & 80 & 17 & 69 & 10 & 21 & 10 \\ 9 & 88 & 97 & 88 & 83 & 45 & 27 & 41 \\ 19 & 3 & 31 & 10 & 2 & 91 & 49 & 69 \\ 35 & 67 & 69 & 42 & 75 & 29 & 5 & 41 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 14 & 88 & 66 & 90 & 91 & 93 & 2 \\ 54 & 81 & 62 & 62 & 57 & 62 & 70 & 3 \\ 66 & 40 & 75 & 11 & 0 & 2 & 7 & 3 \\ 51 & 17 & 35 & 95 & 62 & 93 & 76 & 25 \\ 94 & 93 & 27 & 45 & 33 & 69 & 75 & 86 \\ 59 & 35 & 90 & 58 & 53 & 100 & 92 & 54 \\ 90 & 75 & 43 & 41 & 89 & 17 & 71 & 55 \\ 14 & 73 & 96 & 24 & 36 & 14 & 12 & 84 \end{pmatrix}$$

When simulating in ModelSim, the Verilog HDL files are added for the matrix multiplication module, and the Avalon Master and Avalon Slave interface module are added to ModelSim; after this, the system writes a test file, and it generates clock signal and control signal. The simulation results of matrix multiplication in ModelSim are shown in Figures 4.1 and 4.2.

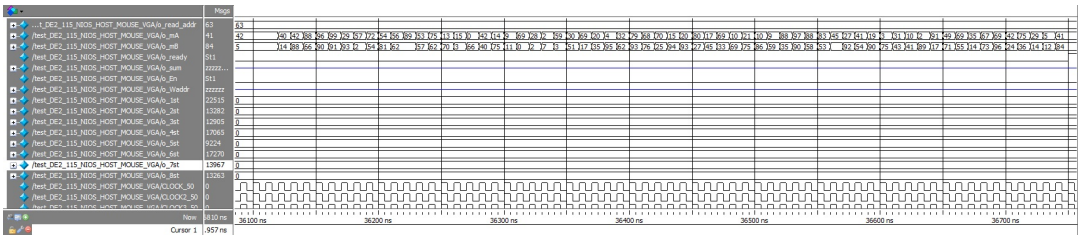

FIGURE 4.1: The loading simulation result in ModelSim

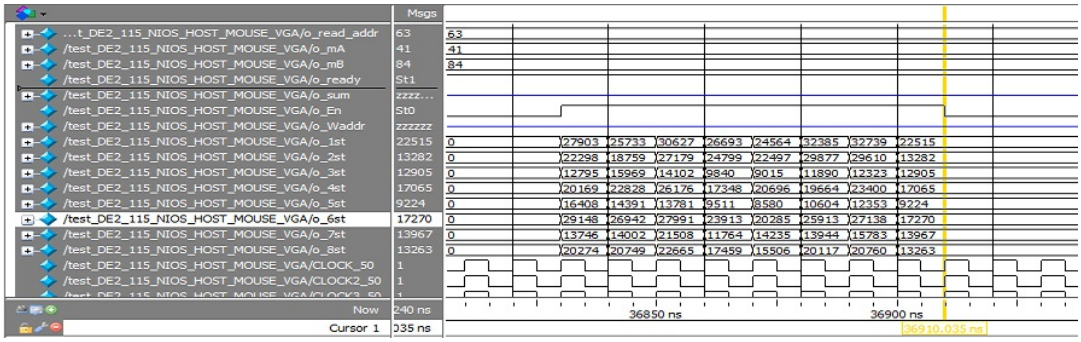

FIGURE 4.2: The calculation simulation result in ModelSim

In Figure 4.1, the loading result from RAM is exactly the same as the data stored in RAM, including the sequence. Thus,it is possible to conclude that the function of loading data from RAM is correct. In Figure 4.2, the simulation results of the hardware accelerator calculation for the matrix multiplication are also the same as the result calculated by the MATLAB software. Therefore, the correctness of the hardware accelerator designed in this thesis is verified.

4.2 Performance Analysis

Figure 4.2 also shows that the duration of loading data from RAM for the calculation and the result after calculation to store back to RAM is **3681** clock cycles. To analyze the computing performance of the designed hardware matrix multiplication accelerator, this performance is compared with other software matrix multiplication systems. Here, we used two ways to build the software matrix multiplication systems–one uses MATLAB and the other one uses a Nios II soft core.

4.2.1 Software matrix multiplication system using MATLAB (MATrix LABoratory)

MATLAB is a proprietary programming language developed by MathWorks. It is often used in matrix manipulations for plotting functions and data, implementation algorithms, and creating user interfaces and interfaces for programs written in other languages, including C, C++, Java, Fortran and Python. In this thesis, we wrote a program to calculate the matrix multiplication using C language in MATLAB. The code is shown in Figure 4.3.

```
Fcump = 2.6e9;    %CPU 2.9GHz
MTKL  = 10e4; %10000 times
Output= [];
for ij = 1:MTKL
   tic;
   if mod(ij,100)==1
     ij
     rng(ij);
   end
   %creat random A,B
   A      = round(100*rand(8,8));
   B      = round(100*rand(8,8));
   [Rr,Cr] = size(A);
   C      = A*B;
   %Output
   Output= C;
   times = toc;
   Times2(ij) = times;
end
 CLocknum = mean(Times2)*Fcump
```

FIGURE 4.3: Software code in MATLAB

Because the duration of each calculation was not the same, we used the average of 10000 times calculation as the output duration. Also, the frequency of the CPU is 2.9 GHz. Thus, the result obtained from the matrix multiplication using software in MATLAB is **108760** clock cycles.

4.2.2 Software matrix multiplication system using Nios II soft core

To further analyze the computing performance of the matrix multiplication hardware accelerator, we wrote a program to calculate the matrix multiplication using the Nios II software core. This core was chosen because

we used the same 32-bit Nios II soft core as the CPU in the hardware accelerator, and they share the same environment. The code is shown in Figure 4.4.

```
start = clock();
for(i=0;i<7;i++)
{
    for(j=0;j<7;j++)
    {
        C[i][j]+=A[i][1]*matrix2[1][j];
    }
}
finish = clock();
Total_time = finish – start;

return Total_time;
```

FIGURE 4.4: Software code using Nios II soft core

After successful compilation, we downloaded the software to a hardware that including the embedded Nios II soft core. It returned a result of 0.0031 second. The crystal on the board for the Nios II soft core is 80 MHz. The result obtained from the matrix multiplication using software in Nios II soft core is **248000** clock cycles. The summary of the matrix multiplication hardware accelerator performance analysis results is shown in Figure 4.5.

Function Name	8 x 8 Matrix Multiplication
Hardware Accelerator (clock cycles)	3681
Software using MATLAB(clock cycles)	108760
Software using Nios II soft core(clock cycles)	248000

FIGURE 4.5: The summary of performance analysis result

The calculation speed increased by 67 times when using a hardware accelerator to achieve matrix multiplication, which is many times more than the software in the same environment. Also, the speed is increased 30 times when using MATLAB with a better CPU. Therefore, using a hardware accelerator can achieve better computational performance. From the result of performance analysis, it can be concluded that the hardware accelerator based on the FPGA proposed in this thesis has good computing performance.

Chapter 5

Conclusion and Discussion

To overcome the computational complexity and time-consuming problem of large matrix multiplications, this thesis designed a hardware accelerator based on FPGA. First, according to the matrix algorithm analysis, we designed a parallel computing structure for matrix multiplication. Then, the structure was built as a custom component and hung on an Avalon bus as a hardware accelerator, working with Nios II processor. After that, ModelSim was used for functional simulation. In order to analyze the computing performance of the hardware accelerator, two software calculations were designed for comparison. The result shows that the matrix multiplication hardware accelerator based on FPGA designed in this thesis has high computational performance.

The design idea of this hardware accelerator is mostly like similar to graphics processing unit (GPU) for CPU. The GPU is a specialized electronic circuit designed to accelerate the processing of images for output to a display device. Because it specializes in processing the image, the GPU can process the image faster. We also use the GPU to reduce the CPU workload. However, the greatest difference between the design idea of GPU and our hardware accelerator is that the structure of GPU is also fixed, when users want to make improvements or modifications, they must do it with software. With our hardware accelerator based on FPGA, it is mapped with real digital circuit. The advantage of the hardware accelerator is that it is more flexible and faster. The disadvantage is that the development time is longer, and the algorithm is more difficult to achieve in hardware.

In this thesis, we built a hardware accelerator for two 8 x 8 matrix multiplication. For better application, it can be improved for any size of matrix. This can be down in two ways, namely in software and hardware. With software, we can treat this 8 x 8 matrix multiplication accelerator as a basic unit, and use the software code to block the large size of matrix into

8 x 8. With hardware, users need to change the hardware design, but the same idea can be used for rebuilding the system into 16 x 16 or a larger size. The advantage for the hardware method is that it is quicker than using the software method. The disadvantage is that it is complicated and need more hardware resources, such as multipliers and larger RAM.

Bibliography

[1] MACIEJOWSKI J M LING K V YUE S P. "A FPGA implementation of model predictive control". In: *American Control Conference* (2006) (cit. on p. 1).

[2] Minghua He and Keck-Voon Ling. "Model predictive control on a chip". In: *Control and Automation, 2005. ICCA'05. International Conference on*. Vol. 1. IEEE. 2005, pp. 528–532 (cit. on p. 1).

[3] "Floating-Point IP Cores User Guide". In: (). URL: https://www.altera.com/documentation/eis1410764818924.html (cit. on p. 4).

[4] Deming Chen, Jason Cong, Peichen Pan, et al. "FPGA design automation: A survey". In: *Foundations and Trends® in Electronic Design Automation* 1.3 (2006), pp. 195–330 (cit. on p. 7).

[5] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*. Vol. 1. Prentice Hall Professional, 2003 (cit. on p. 9).

[6] "Stratix GX Devices & Nios II Family of Embedded Processors". In: (). URL: https://www.altera.com/products/fpga/stratix-series/stratix/stratix-gx/features/sgx-stratixgx_nios.html (cit. on p. 11).

[7] "Building Formal Assumptions to Describe Wishbone Behaviour". In: (Nov. 2017). URL: http://zipcpu.com/zipcpu/2017/11/07/wb-formal.html (cit. on p. 12).

[8] "Quartus II Version 8.0 Handbook Volume 5: Embedded Peripherals". In: (May 2008). URL: http://www.johnloomis.org/NiosII/docs/n2cpu_nii5v3.pdf (cit. on p. 12).

[9] James E Smith. "Decoupled access/execute computer architectures". In: *ACM SIGARCH Computer Architecture News*. Vol. 10. 3. IEEE Computer Society Press. 1982, pp. 112–119 (cit. on p. 14).

[10] "Introduction to Quartus II Manual". In: (). URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/intro_to_quartus2.pdf (cit. on p. 17).

[11] "ModelSim -Intel FPGA". In: (). URL: https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.html (cit. on p. 17).