

UNIVERSITY OF CINCINNATI

Date: Nov 10, 2005

I, Shuguang Wu,

hereby submit this work as part of the requirements for the degree of:

Master of Science

in:

Computer Engineering

It is entitled:

FPGA Implementation of the FDTD Algorithm Using

Local SRAM

This work and its defense approved by:

Chair: Dr. Karen A. Tomko

Dr. Harold W. Carter

Dr. Wen-Ben Jone

FPGA Implementation of the FDTD Algorithm

Using Local SRAM

A thesis submitted to the

Division of Research and Advanced Studies
of
The University of Cincinnati

in partial fulfillment of requirements for the degree of

Master of Science

in the

Department of Electrical & Computer Engineering
and
Computer Science
of
The College of Engineering

November 2005

by

Shuguang Wu

B.E., Beijing University of Posts and Telecommunications, 1996
M.E., Beijing University of Posts and Telecommunications, 1999

Thesis Advisor and Committee Chair: **Dr. Karen A. Tomko**

Abstract

The Finite-Difference Time-Domain (FDTD) algorithm is a powerful tool to model electromagnetic phenomena. It is computation-intensive. Plenty of work has been done to implement this algorithm on FPGA and to improve the implementation performance.

This thesis presents an implementation of the FDTD algorithm on FPGA in two reconfigurable computing systems in order to explore the implementation feasibility and to improve the implementation performance. The two reconfigurable computing systems used in this thesis are a prototyping system in the ACL Lab at University of Cincinnati and a Cray XD1 system available at the Ohio Supercomputer Center. There are three major functional units in the FPGA: update engines calculating three equations in the FDTD algorithm, interface to host system, interface to local SRAMs. The local SRAMs are used to store input and output data for the FDTD algorithm. The purpose to use the local SRAMs is to reduce the data transfer between the host system and the FPGA. Host applications are developed to verify the FPGA implementation.

Acknowledgement

I'm extremely grateful to my advisor Dr. Karen A. Tomko for her precious guidance and encouragement during the work on my thesis. It is really great honor for me to work with her. And I'm very thankful for her kindness, support and co-operation.

I'm thankful to Dr. Harold W. Carter and Dr. Wen-Ben Jone for participating in my thesis committee.

SynplifyPro used in this project is provided by Synplicity through their University Program, Xilinx ISE is provided by Xilinx through Xilinx's University Program (XUP), and Modelsim is donated from Mentor Graphics as part of their Higher Education Program (HEP).

I would like to thank the members in our group for their help. They are: Sachin Gandhi, Robert Cully, Ashish Desai. It is great pleasure to work with them.

And I express my great gratitude to my parents and my sisters. Their love, encouragement and support accompany me forever.

Table of Contents

1	Introduction	1
1.1	Overview	1
1.2	Related Work	4
1.2.1	Reconfigurable Computing System	4
1.2.2	FPGA Implementation of Floating Point Algorithm	5
1.2.3	FPGA Implementation of FDTD Algorithm	7
1.3	Thesis Organization	8
2	Finite-Difference Time-Domain Algorithm for Electromagnetic Simulation	10
3	FPGA-Based Reconfigurable Computing Systems	13
3.1	Laboratory Prototype System	14
3.1.1	System Architecture	14
3.1.2	Simulation Environment	15
3.1.3	Host Programming Environment	17
3.2	Cray XD1 System	18
3.2.1	System Architecture	19
3.2.1.1	Cray XD1 Chassis	19
3.2.1.2	RapidArray Interconnect	20
3.2.2	Simulation Environment	21
3.2.3	Host Programming Environment	22
4	FDTD FPGA Implementation using Local SRAM	25

4.1	FPGA Implementation Flow	26
4.2	FPGA Implementation in the ACL Prototype System	28
4.2.1	VHDL Design	28
4.2.1.1	FPGA Function	28
4.2.1.2	FPGA Architecture	30
4.2.1.3	Data Organization in Local Memories	32
4.2.1.4	Components and Processes	33
4.2.2	Simulation	41
4.2.3	Synthesis	41
4.2.4	P&R	42
4.2.5	Host Programming	42
4.3	FPGA Implementation in the Cray XD1 System	43
4.3.1	VHDL Design	43
4.3.1.1	FPGA Function	44
4.3.1.2	FPGA Architecture	45
4.3.1.3	Data Organization in Local Memories	46
4.3.1.4	Components and Processes	48
4.3.2	Simulation	57
4.3.3	Synthesis	58
4.3.4	P&R	59
4.3.5	Host Programming	59
5	Theoretical Performance Analysis and Future Work	61
5.1	Performance Analysis in the Prototype System	62

5.2 Performance Analysis in the Cray XD1 System	63
5.3 Performance Comparison	65
5.4 Future Work in the Cray XD1 System	66
Bibliography	69

List of Figures

Figure 3-1 : Reconfigurable Computing System	13
Figure 3-2 : FIREBIRD TM /PCI block diagram	15
Figure 3-3 : Cray XD1 chassis	19
Figure 3-4 : Expansion Module	20
Figure 4-1 : FPGA Implementation Flow	26
Figure 4-2 : FPGA Architecture in the Prototype System	31
Figure 4-3 : Block Diagram of Magnetic update Hx	34
Figure 4-4 : Block Diagram of Magnetic update Hy	34
Figure 4-5 : Block Diagram of Electric update	35
Figure 4-6 : Structure of the Test Bench in the Prototype System	41
Figure 4-7 : FPGA Architecture in the Cray XD1 System	45
Figure 4-8 : Parameters in Register FPGA_reg	51
Figure 4-9 : Procedure for FDTD algorithm calculation.	53
Figure 4-10 : Read Addresses Generation Order for Hx, Hy values update	54
Figure 4-11 : Write Addresses Generation Order for Hx, Hy values update	55
Figure 4-12 : Read Addresses Generation Order for Ez values update	56
Figure 4-13 : Write Addresses Generation Order for Ez values update.	57
Figure 4-14 : Structure of the Test Bench in the Cray XD1.	58

Chapter 1

Introduction

1.1 Overview

This thesis presents an implementation of the Finite-Difference Time-Domain (FDTD) algorithm on FPGA in two reconfigurable computing systems. Local SRAMs are used to store input and output data for the FDTD algorithm. Host applications are developed to verify the implementation and to evaluate performance of the implementation.

The work done in this thesis is based on the work done by Gandhi. Gandhi implemented the FDTD algorithm in a reconfigurable computing system named Heterogeneous HPC computer (HHPC) [21]. His work focuses on speedup of three equations in the FDTD algorithm. In his thesis, three update engines are implemented to calculate three equations in the FDTD algorithm, and parallelism and pipelining are utilized to speed up these update engines. Input and output data are stored in host system memories. For each iteration of FDTD algorithm calculation, input and output data must be transferred between the host system and the FPGA. This slow data communications is performance bottleneck and greatly degrades the performance of the whole system.

In this thesis, the implementation focuses on speedup of the whole process to complete the FDTD algorithm. The whole process includes input and output data transfer between the host system and the FPGA and data update by the update engines. In order to realize this goal, the implementation is done in two reconfigurable computing systems

different from the HHPC system. Local SRAMs attached to the FPGA are used to store input and output data to reduce data transfer between the host system and the FPGA.

The two reconfigurable computing systems used in this thesis are a prototyping system in the ACL Lab at University of Cincinnati which is referred to as the Prototype System throughout this thesis and a Cray XD1 system available at the Ohio Supercomputer Center. The Prototype system consists of a PC and a reconfigurable computing board. In the Prototype system, FPGA and local SRAMs are located on the reconfigurable computing board, host applications run on the host PC. The Cray XD1 system consists of a number of Cray chassis connected by a switch fabric. Each chassis consists of one management processor and six compute blades. In the Cray XD1 system, host processors are located on the compute blades and host applications run on the host processors. FPGAs and local SRAMs are attached to the compute blades.

There are three major functional units in the FPGA: update engines calculating three equations in the FDTD algorithm, interface to the host system, interface to the local SRAMs. The update engines are implemented using floating point units. Two update engines are used to update magnetic field values, one is used to update electric field values. These update engines are pipelined. Furthermore, in the Cray XD1 system, the two update engines used to update magnetic field values are processed in parallel. Due to the pipelining and the parallelism, throughput and speed of the implementation improve. Interface to the host system interacts with the host system. The host system issues commands such as read and write to the FPGA through this interface. The FPGA processes the received commands and gives response to the host system through this

interface, too. Interface to the local SRAMs is used for the FPGA to read and write the local SRAMs.

In order to reduce the data transfer between the host system and the FPGA, input and output data for the FDTD algorithm are stored in the local SRAMs. In the Prototype system, local SRAMs are single-port memories. They can't be read and written simultaneously. Three local SRAMs are used to store input data and two are used to store output data for the FDTD algorithm. Input data should be transferred from the host system to the input SRAMs before the calculation of the FDTD algorithm begins. When one iteration of calculation completes, the results will be stored in the output SRAMs and the host system should read the results back from the output SRAMs. In the Cray XD1 system, there are four dual-port SRAMs. They can be read and written at the same time as long as the read address and the write address are not the same. These SRAMs are organized as ping-pong SRAMs. This ping-pong mechanism enables the FPGA to execute many iterations of calculation with little communication with the host system. Reduction of data transfer between the host system and the FPGA during intervals between calculation iterations greatly improves the performance of the whole system.

In this thesis, the main purpose of developing the host application is to verify the FPGA implementation. Though the FPGA design is already simulated and debugged after VHDL design completes, the FPGA design still needs to be verified to run correctly in the actual system. So host application needs to be developed to verify the function of the FPGA implementation. The host application loads the FPGA binary file into the FPGA, writes data to and reads data from the registers in the FPGA or the local SRAMs through the FPGA. It also starts the internal update engines in the FPGA and monitors the status

of the update engines. By this method, all the functional units of the FPGA can be verified.

The host application can also be augmented to evaluate the performance of the FPGA implementation. The experimental evaluation is the work of another member in our group. In this thesis, only theoretical analysis and evaluation of performance of the FPGA implementation is done.

1.2 Related Work

1.2.1 Reconfigurable Computing System

FPGA was first introduced in early 1990s. Since then, a lot of reconfigurable computing systems have been developed. C.E. Cox et al. developed a reconfigurable computing system named GANGLION [1]. GANGLION was a fast digital connectionist classifier. Its architecture was realized using a FPGA array on a VME card attached to a workstation. M. Wazlowski et al developed PRISM II [2]. PRISM II was a general purpose hardware platform. It mainly consisted of PRISM host processor and reconfigurable hardware platform. The compiler for PRISM II accepted a host application as input and produced hardware image and software image. The hardware image was used for programming the hardware platform. The Splash 2 system was developed by D. Buell et al. [3]. It consisted of a Sun workstation, an interface board, and Splash array boards. FPGAs on each array board were arranged in a linear array and were connected via a crossbar switch. The Splash 2 system was effective on applications such as text searching, sequence analysis and image processing. T.Tsutsui et al. developed YARDS [4]. YARDS comprised three cards: the main card, the MPU card, and the FPGA card.

There were a FPGA array mounted on the FPGA card. Several telecommunication applications were developed in the YARDS system.

Miyazaki, T categorized reconfigurable computing systems into three types: attached processors, coprocessors and special purpose machines [5]. He also investigated typical applications suitable for reconfigurable computing systems. Smith, M.C. et al. investigated the hardware architecture and configuration of reconfigurable computing system [6]. Software architecture of reconfigurable computing system was discussed, too. Fidanci, O.D. et al. overviewed hardware architecture and programming model of SRC-6ETM reconfigurable computers [7]. The SRC-6E could outperform a general-purpose microprocessor for computationally intensive algorithms whether or not the overhead due to configuration and data transfer was included. The paper presented by El-Araby, E. et al. was also based on the SRC-6ETM reconfigurable computers [8]. They put their concern on the DMA transfer between the host system and the FPGA. Theoretical model was built and analyzed for this performance bottleneck. Experimental work was done to verify the theoretical analysis.

The reconfigurable computing systems used in this thesis include the Prototype system and the Cray XD1 system. Based on the paper presented by Miyazaki, T [5], the Prototype system can be categorized into attached processors, the Cray XD1 system can be categorized into coprocessors.

1.2.2 FPGA Implementation of Floating Point Algorithm

The most commonly used format for floating point numbers is described in IEEE Std 754 [9]. Until the late 1990s, the resource and speed of FPGA was restricted and it

was difficult to implement floating point arithmetic on FPGA. Many people explored and analyzed the feasibility and performance of implementation of floating point algorithm on FPGA during this period. B. Fagin et al. implemented floating point adder and multiplier using FPGA and discussed the tradeoff between performance and area requirement [10]. N. Shirazi et al. designed and optimized floating point adder/subtractor, multiplier and divider to maximize speed and to minimize area [11]. L. Louca et al. implemented floating point adder and multiplier and investigated the area-speed tradeoff [12]. W. B. Ligon III et al. presented implementation of floating point addition and multiplication functional units and discussed the performance and device utilization of these units [13].

From the beginning of 2000s, the resource and speed of FPGA increases greatly and area is no longer the overriding concern for implementation of floating point algorithm. More efforts are focused on performance and optimization of the implementation during this period. A. Jaenicke et al. presented an approach for developing and optimizing parameterized floating point units [14]. These units could be customized to meet user constraints by varying the precision, rounding modes, or the number of pipeline stages. Jian Liang et al. presented a floating point unit generation tool for FPGAs [17]. This tool could be used to create a variety of floating point units based on throughput, latency, and area requirements. Pavle Belanovic also presented a parameterized floating point library for use with reconfigurable hardware [15][16]. This library was fully parameterized for format control, arithmetic operations and conversion to and from any fixed-point format.

The floating point library presented by Pavle Belanovic is chosen for the FPGA implementation of FDTD Algorithm in my thesis.

1.2.3 FPGA Implementation of FDTD Algorithm

The first paper about FPGA implementation of FDTD algorithm was presented by Schneider et al. [18]. In his paper, one-dimensional FDTD algorithm was implemented on FPGA using a pipelined bit-serial arithmetic architecture. And the implementation used integer calculation. A one-dimensional resonator was used to verify the implementation and to explore the hardware speed and costs. Durbano et al. presented the first FPGA implementation of three-dimensional FDTD algorithm using floating point arithmetic units [19]. In his paper, system architecture for the FPGA implementation was introduced and functionality of each module in the system architecture was described. The speed of the implementation was more than 5 times slower than software implementation at that time. Chen et al. implemented two-dimensional FDTD algorithm on FPGA using fixed point arithmetic [20]. In the implementation, magnetic field updating algorithm along the x-coordinate and along the y-coordinate were processed in parallel, and magnetic field updating algorithms and electric field updating algorithm were partially paralleled. The components implementing the magnetic field updating algorithms and the electric field updating algorithm were designed as pipelines. The on-board memories stored the magnetic field values and the electric field values and were organized in a swapping mechanism. Another implementation of two-dimensional FDTD algorithm on FPGA was done by Gandhi [21]. In his master thesis, floating point arithmetic was used, three update engines magnetic update H_x , magnetic update H_y and

electric update were implemented to realize the magnetic field updating algorithms and the electric field updating algorithm, and parallelism and pipelining were utilized to speed up the implementation. The electric field values and the magnetic field values were stored in host system memories. The slow data communications between the host system and the FPGA degraded the performance of the whole system.

In my thesis, the update engines designed by Gandhi are used, local SRAMs are used to store the electric field values and the magnetic field values.

1.3 Thesis Organization

Chapter 1 introduces the work I have done. In summary, I have improved the FPGA implementation in [21] by utilizing SRAMs attached to the FPGA to reduce host-FPGA communications. Also introduced is related work done in the field of reconfigurable computing, on FPGA implementation of floating point algorithm and on FPGA implementation of FDTD Algorithm. Chapter 2 introduces the Finite-Difference Time-Domain (FDTD) algorithm. The method to derive FDTD algorithm from Maxwell's equations is also explained in this chapter. Chapter 3 describes the reconfigurable computing systems used in this thesis: the Prototype system and the Cray XD1 system. The architecture of each system is introduced, the simulation environment and the host programming environment provided by each system are described. Chapter 4 presents in detail the FPGA implementation of FDTD algorithm in the Prototype system and the Cray XD1 system. The FPGA implementation is divided into five stages: VHDL design, simulation, synthesis, P&R, host programming. The work done on each stage is

explained in detail. Chapter 5 analyzes results and performance of the FPGA implementation done in this thesis and proposes some suggestions for the future work.

Chapter 2

Finite-Difference Time-Domain Algorithm for Electromagnetic Simulation

The Finite-Difference Time-Domain (FDTD) algorithm is a very powerful tool for the modeling of electromagnetic phenomena. This algorithm is a set of discretized finite difference equations derived from Maxwell's equations. It was first presented by Kane S. Yee.

Maxwell's equations in an isotropic medium are [22]:

$$\partial B / \partial t + \nabla \times E = 0 \quad (1a)$$

$$\partial D / \partial t - \nabla \times H = J \quad (1b)$$

$$B = \mu H \quad (1c)$$

$$D = \varepsilon E \quad (1d)$$

The definitions for the symbols in the equations (1a)--(1d) are:

B – Magnetic flux density

E – Electric field

D – Electric flux density

H – Magnetic field

μ – Magnetic permittivity

ε – Electric permittivity

Maxwell's equations are very powerful in solving electromagnetic problems, but are not suitable for processing by computer. In 1966, Kane S. Yee successfully presented a method to discretize Maxwell's equations and to derive the FDTD algorithm [23].

There are two assumptions for the derivation. One is that the boundary condition should be appropriate for a perfectly conducting surface. This assumption implies that the tangential components of the electric field vanish and the normal component of the magnetic field vanishes on the surface. The other assumption is that the space grid size must be such that over one increment the electromagnetic field does not change significantly [23].

Based on these two assumptions, Kane S. Yee discretized both the physical region and the time interval of the Maxwell's equations on uniform grids. Then he derived the finite difference equations for two modes of electromagnetic waves: Transverse electric wave (TE) and Transverse magnetic wave (TM). Later these equations are named as Finite-Difference Time-Domain (FDTD) algorithm. Here only FDTD algorithm for the TM is presented [23]:

$$H_x^{n+1/2}(i, j + \frac{1}{2}) = H_x^{n-1/2}(i, j + \frac{1}{2}) - \frac{1}{Z} \frac{\Delta r}{\Delta y} [E_z^n(i, j+1) - E_z^n(i, j)] \quad (2a)$$

$$H_y^{n+1/2}(i + \frac{1}{2}, j) = H_y^{n-1/2}(i + \frac{1}{2}, j) + \frac{1}{Z} \frac{\Delta r}{\Delta x} [E_z^n(i+1, j) - E_z^n(i, j)] \quad (2b)$$

$$E_z^{n+1}(i, j) = E_z^n(i, j) + Z \frac{\Delta r}{\Delta x} [H_y^{n+1/2}(i + \frac{1}{2}, j) - H_y^{n+1/2}(i - \frac{1}{2}, j)] \\ - Z \frac{\Delta r}{\Delta y} [H_x^{n+1/2}(i, j + \frac{1}{2}) - H_x^{n+1/2}(i, j - \frac{1}{2})] \quad (2c)$$

FDTD Algorithm for TM waves

The electromagnetic field values in the equations (2a)--(2c) are updated $\frac{1}{2}$ time step by $\frac{1}{2}$ time step from two parts: field values calculated in previous $\frac{1}{2}$ time step and field values in adjacent space cells. This characteristic makes it possible to implement the FDTD algorithm on parallel computers because only nearest-neighbor interactions are involved.

There are many methods to implement FDTD algorithm on parallel computers. Commonly in these methods, the discretized physical space is partitioned into regions and distributed to multiple processors. Data on the boundaries between regions is exchanged between processors, the electric field values are updated and stored in memory using previously stored magnetic field values, and then the magnetic field values are updated and stored in memory using the electric field values just calculated.

Chapter 3

FPGA-Based Reconfigurable Computing Systems

A reconfigurable computing system consists of a number of computing nodes connected by an interconnection network. Host processors and reconfigurable computing elements (FPGAs and CPLDs) are associated with some or all of the computing nodes. The general diagram of a reconfigurable computing system is depicted in figure 3-1:

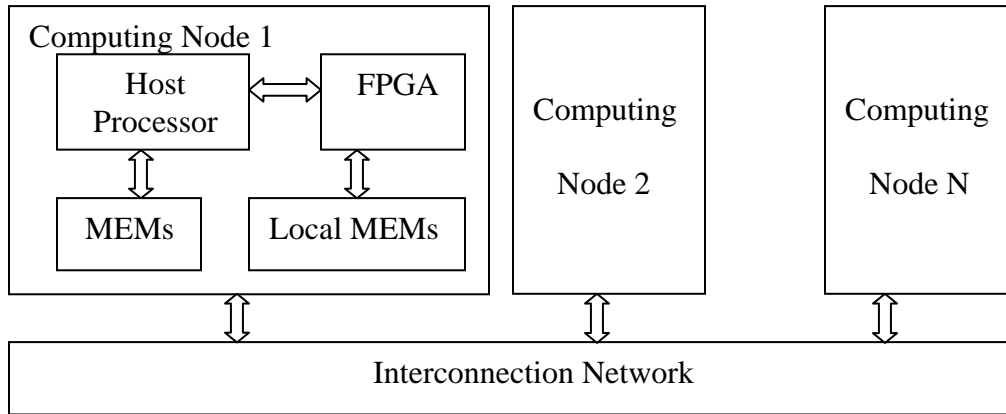


Figure 3-1 : Reconfigurable Computing System

In a reconfigurable computing system, a host application runs on one or several host processors, and some tasks in the host application are assigned to the FPGAs. The host processors and the FPGAs cooperate to execute the host application. Reconfigurable computing system is a good choice for host applications with intensive computation and parallelism.

In this thesis, FDTD algorithm is implemented in two reconfigurable computing systems: the Prototype system in the ACL Lab at University of Cincinnati and the Cray XD1 system. The Cray XD1 system has higher bandwidth, lower latency and more powerful memory architecture than the Prototype system.

3.1 Laboratory Prototype System

3.1.1 System Architecture

The Prototype system is an integration of a PC with a reconfigurable computing board inserted into one of its PCI slots. This system is built for reconfigurable computing research. The specification for the PC is:

- Intel® Xeon™ CPU 1.70 GHz
- 512 MB of RAM
- 28 GB IDE disk
- 3Com® Fast Ethernet Controller

The reconfigurable computing board in the Prototype system is an Annapolis Micro FIREBIRD™/PCI board. The FIREBIRD™/PCI features include [24]:

- one Processing Element PE0 that is a Xilinx VIRTEX-E FPGA
- PE0 can optionally be programmed from flash on power up
- Processing clocks up to 150MHz
- Five memory banks, containing 9 to 36 Mbytes of synchronous ZBT SRAM
- 5.4Gbytes/sec of memory bandwidth
- 66MHz/64bit PCI transactions (3.3V PCI signaling only)

The block diagram for the FIREBIRD™/PCI board is depicted in figure 3-2.

The local address data (LAD) bus in figure 3-2 is a single master, 64-bit, shared address/data bus used for communications between the FPGA and the host processor.

Every cycle on the LAD bus is initiated by the PCI Controller.

The user's design in the FPGA communicates with the host processor via the LAD bus. Any memory reads/writes by the host also go through the LAD bus and the FPGA.

The FPGA directly accesses the ZBT SRAM.

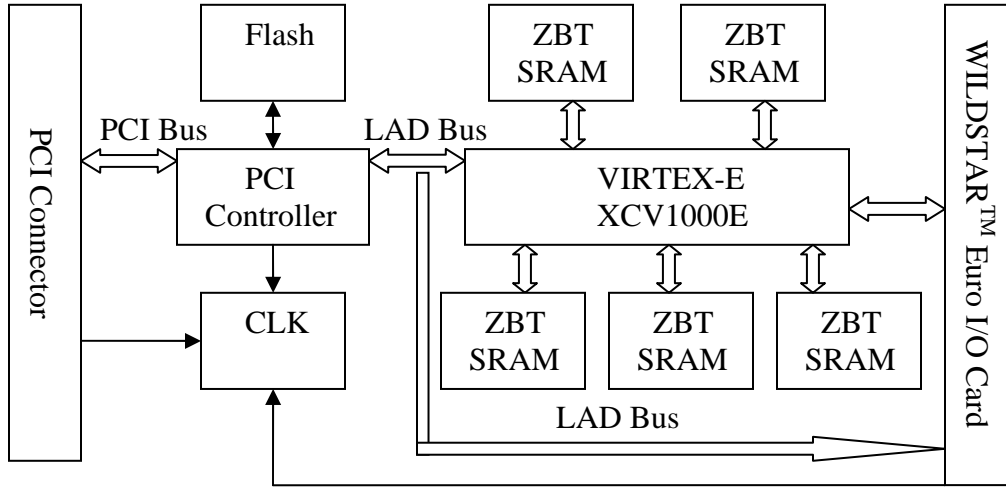


Figure 3-2 : FIREBIRD™/PCI block diagram [24]

3.1.2 Simulation Environment

The simulation environment includes VHDL models for the host system and the FIREBIRD™/PCI board. These VHDL models are provided by Annapolis Micro Systems, Inc. to give the designer an accurate test bed for the completed Processing Element (PE) design and to enable the designer to validate the PE design before the synthesis, placement, and routing steps [24].

The VHDL model for the host system is used to access the PE resources of the FIREBIRD™/PCI board via the PCI Controller. It can also be used to initialize and analyze the contents of any memory device on the FIREBIRD™/PCI board. This VHDL model is similar to the actual host system and it provides VHDL functions similar to the actual software Application Programming Interface (API) functions.

The VHDL model for the host system includes the following functions [24]:

- WS_Open : Sets up initial board configuration
- WS_ReadPeReg/WS_WritePeReg: Reads/writes register locations in the PE Register Space
- WS_ReadPeReg64/WS_WritePeReg64: Reads/writes register locations in the PE Register Space
- WS_DMARead/WS_DMAWrite: Reads/writes the PE Space using DMA
- WS_MClkSetConfig/WS_UClkSetConfig: Sets the configuration of the memory and PE/user clocks
- WS_WaitOnInterrupt : Waits for PE interrupt signal
- WS_QueryInterruptStatus : Returns current interrupt status of all PEs
- WS_ResetInterrupt : Clears pending interrupts

The VHDL model for the FIREBIRD™/PCI board includes [24]:

- PCI Controller Model
- On-board Memory Model
- I/O Card Model

The PCI controller model provides functions to handle LAD bus transactions, to configure the clocks, and to handle PE interrupts. These functions can all be accessed from the functions provided by the VHDL model for the host system.

The on-board memory model is used to inspect the memory contents from the VHDL simulator tool. This model can be configured to an “empty” architecture if no memories on the board are needed by the PE.

The I/O card model is empty. This model tri-states all of its signals to the FIREBIRD™/PCI board.

3.1.3 Host Programming Environment

The Host Programming environment consists of three layers [25]:

- User's host application
- WILDSTAR™ Application Programming Interface (API)
- WILDSTAR™ Device Driver

User's host application is programmed with 'C' language and runs on the host PC. Its essential function is to load the embedded PE application (provided as an FPGA binary file) into the PE on the FIREBIRD™/PCI board and to maintain the operating environment of the embedded PE application.

The overall host application is bound by the WILDSTAR™ API. To develop host application, at least one library and several include files must be included. Under Windows® NT™ and UNIX, a single library--the WILDSTAR™ API is included. The include files contain constants, data types, and prototypes necessary to interface to the WILDSTAR™ API.

The WILDSTAR™ API centralizes specific knowledge of the underlying system. It presents a generalized view of the hardware resources and control operations in the system.

The WILDSTAR™ API routines are organized to provide high-level operations by performing combinations of low-level WILDSTAR™ device driver functions. The routines are provided to accomplish clock control, PE control, register interfaces

read/write, DMA Operations, Interrupt Operations, Temperature/Power Monitoring Operations, LED Display Operations. The subset of the API functions available to the host application programmer and used for the work presented in this thesis is [25]:

- WS_Open : Open a FIREBIRD™/PCI board
- WS_Close : Close a FIREBIRD™/PCI board
- WS_GetPhysicalConfig : Get the configuration information from the ID PROM(s)
- WS_ProgramPe : Program a particular PE
- WS_DeProgramPe : Deprogram a particular PE
- WS_MClkSetConfig : Set the source and frequency for M clock
- WS_UClkSetConfig : Set the source and frequency for P clock
- WS_ReadPeReg : Read from PE register space
- WS_WritePeReg : Write to PE register space
- WS_DmaRead : Read a buffer using DMA
- WS_DmaWrite : Write a buffer using DMA
- WS_ResetInterrupt : Reset the specified interrupt sources on the board
- WS_QueryInterruptStatus : Check the status of pending interrupts on the board

The WILDSTAR™ device driver provides a low-level interface to the FIREBIRD™/PCI board's register space and a central location for all of the system's global resources. It performs all necessary steps to initialize, to access and to maintain the hardware.

3.2 Cray XD1 System

3.2.1 System Architecture

The Cray XD1 system is designed specifically for high performance computing. It consists of many Cray XD1 chassis interconnected with one high-speed switch fabric called the RapidArray Interconnect.

3.2.1.1 Cray XD1 Chassis

The Cray XD1 chassis contains a management processor and six compute blades.

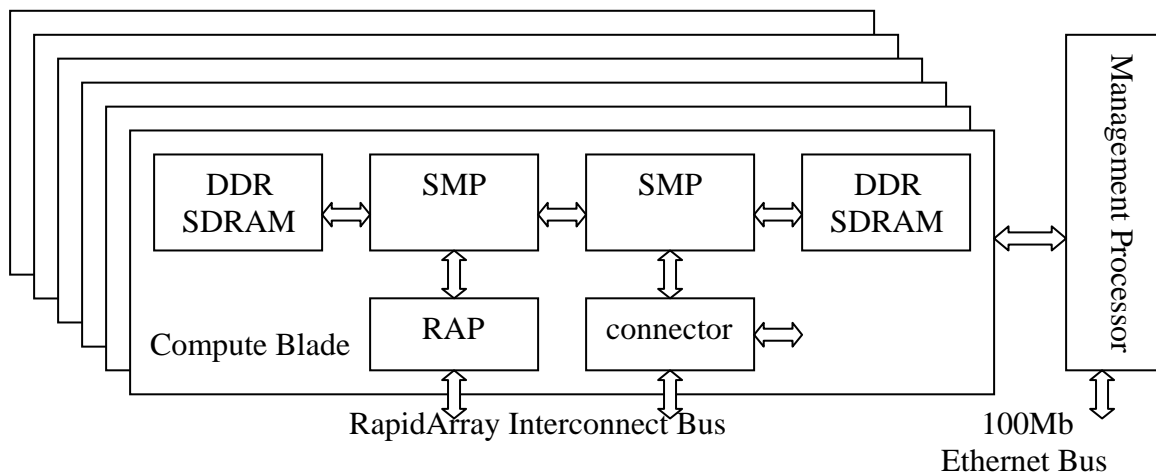


Figure 3-3 : Cray XD1 chassis [27]

Each compute blade includes the following components [27]:

- Two 64-bit AMD Opteron processors—configured as one two-way symmetric multiprocessors (SMP) that runs Linux operating system.
- Up to 16GB of DDR SDRAM per SMP
- one RapidArray processor (RAP)—provide high-bandwidth, low-latency interface to the RapidArray Interconnect
- A connector for an expansion module

The expansion module is an optional board that attaches to a compute blade. Each expansion module contains the following components [27]:

- An application acceleration processor (AAP FPGA)
- An RAP which provides two additional RapidArray links per compute blade
- Four quad-data-rate (QDR) II SRAMs for the AAP FPGA
- A programmable clock source for the AAP FPGA

The block diagram for the expansion module is as follows [27]:

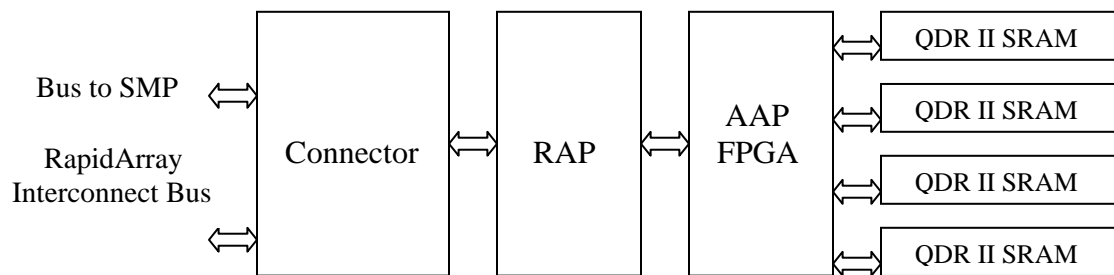


Figure 3-4 : Expansion Module [27]

The AAP FPGA belongs to the Xilinx Virtex-II Pro series. The resources in the AAP FPGA include 2 PowerPC processor blocks, 23616 logic slices, 232 18×18 bit multiplier blocks, 232 18 Kb Block RAMs.

The RAP connects the AAP FPGA to the local SMP and the RapidArray Interconnect. The QDR II SRAM provides local high-speed storage for the AAP FPGA. Each of the four QDR II SRAM circuits operates independently. The programmable clock sets the speed of the AAP FPGA for each design.

3.2.1.2 RapidArray Interconnect

The RapidArray Interconnect is a central communications construct in the Cray XD1 system. It connects processors and memories within a chassis and between chassis. This interconnect enables the system to avoid PCI-X bus bottlenecks and shared-resource contention.

“The RapidArray Interconnect is a 96-GB-per-second (maximum per chassis) nonblocking, embedded crossbar-switch fabric that connects the RAPs. Each chassis has either one or two RapidArray switch fabrics, each of which consists of RapidArray links and a 24-port internal switch.” [27]

3.2.2 Simulation Environment

The elements in the simulation environment are [31]:

- A VHDL test bench
- VHDL model for the RapidArray fabric
- VHDL model for the QDR II SRAM
- The stimulus file

The test bench instantiates the AAP FPGA, the fabric model and the QDR II SRAM model.

In the simulation, the fabric model processes commands such as read request and write request from a stimulus file. It will also process read and write requests generated by the AAP FPGA.

The QDR II SRAM model simulates a dual-port, 36-bit wide QDR II SRAM. This SRAM is synchronous and can be read and written at the same time.

The stimulus file is a text file containing stimulus commands which are inputs to the fabric model. The commands includes: [31]

- I : Initialize Link
- P <Text to print> : display messages on the simulation console
- D <delay value> : insert a time delay between requests
- R <addr> <expected data> <byte mask> <byte request> <size> : Read Request
- W <addr> <write data> <byte mask> <byte request> <size> : Write Request
- B <data> <byte mask> : Burst Request

where:

<Text to print> = Message for console

<delay value> = delay in user clock cycles

<addr> = 40 bits address in hex

<data> = 64 bits data in hex

<byte mask> = 8 bits data mask in hex (1 = enable, 0 = disable)

<byte request> = 4 bits in hex (1 = byte req, 0 = dw request)

<size> = size of read/write access in double words (32 bits) in hex

3.2.3 Host Programming Environment

The Host Programming environment consists of three layers [28]:

- host application on the SMP
- FPGA Application Programming Interface (API)
- RT core bus transactions

To communicate with the AAP FPGA, the host application first opens the AAP FPGA to get a file descriptor. Then using the file descriptor, the host application loads the converted logic file (provided as an FPGA binary file) into the AAP FPGA. After this, the host application can read and write registers in the AAP FPGA, read data from or write data to the QDR II SRAMs through the AAP FPGA. At last, the host application can reset the AAP FPGA and close the file descriptor.

The FPGA API provides the functions that the host application needs to use an AAP FPGA. These functions include [28]:

- `fpga_open` : Opening an FPGA
- `fpga_load` : Loading the converted logic file into the FPGA
- `fpga_reset` : Resetting an FPGA
- `fpga_start` : Releasing an FPGA from reset state
- `fpga_memmap` : Accessing FPGA locations from the host application
- `fpga_mem_sync` : Synchronizing accesses to FPGA locations
- `fpga_wrt_appif_val` : Writing individual FPGA locations
- `fpga_rd_appif_val` : Reading individual FPGA locations
- `fpga_set_ftrmem` : Accessing host application memory from an FPGA
- `fpga_status` : Checking the status of an FPGA
- `fpga_is_loaded` : Checking the programming state of an FPGA
- `fpga_unload` : Erasing an FPGA
- `fpga_close` : Closing an FPGA

There is a component referred to as the Rt_core (see 4.3.1.4) in the AAP FPGA. The FPGA API functions initiate appropriate RT core bus transactions to this component. The user logic in the AAP FPGA processes the bus transactions and responds appropriately. If user logic in the AAP FPGA needs to access the SMP memory through Rt_core, it sends a bus transaction to Rt_core, then the bus transaction is forwarded to hardware on the SMP, where it becomes a read or write transaction to the SMP DRAM.

Chapter 4

FDTD FPGA Implementation using Local SRAM

FPGA implementation of the FDTD algorithm has been done for each of the systems described in the previous chapter: the Prototype system and the Cray XD1 system. The design goal in the Prototype system is to explore implementation feasibility, while the goal in the Cray XD1 system is to improve implementation performance.

In both systems, floating point units are used to implement the update engines and the update engines are pipelined, input and output data for the FDTD algorithm are stored in the local SRAMs to reduce the data transfer between the host system and the FPGA.

In the Prototype system, the two magnetic update engines are processed serially, each SRAM is single-port with random delay, and only one iteration of FDTD algorithm calculation can be supported. While in the Cray XD1 system, the two magnetic update engines are processed in parallel, each SRAM is dual-port with fixed delay, the data in the SRAMs in the Cray XD1 system are organized as ping-pong buffers, and specified number of iterations of FDTD algorithm calculation can be supported. So the design in the Cray XD1 system has higher throughput, higher memory access speed, and less data flow between the host system and the FPGA.

The FPGA implementation flow in the Prototype system is almost the same as that in the Cray XD1 system. But the detailed steps are different in these two systems. In this thesis, the FPGA implementation flow and the detailed steps in the flow are explained for both systems.

4.1 FPGA Implementation Flow

The FPGA implementation flow in this thesis has the following five basic steps:

- VHDL Design—Create VHDL source code for the target FPGA
- Simulation—Simulate the VHDL source code
- Synthesis—Translate the VHDL source code into a gate-level netlist
- P&R—Place and route the gate-level netlist on the target FPGA
- Host Programming—Program host application, C/C++ with API calls

The FPGA implementation flow is depicted in figure 4-1:

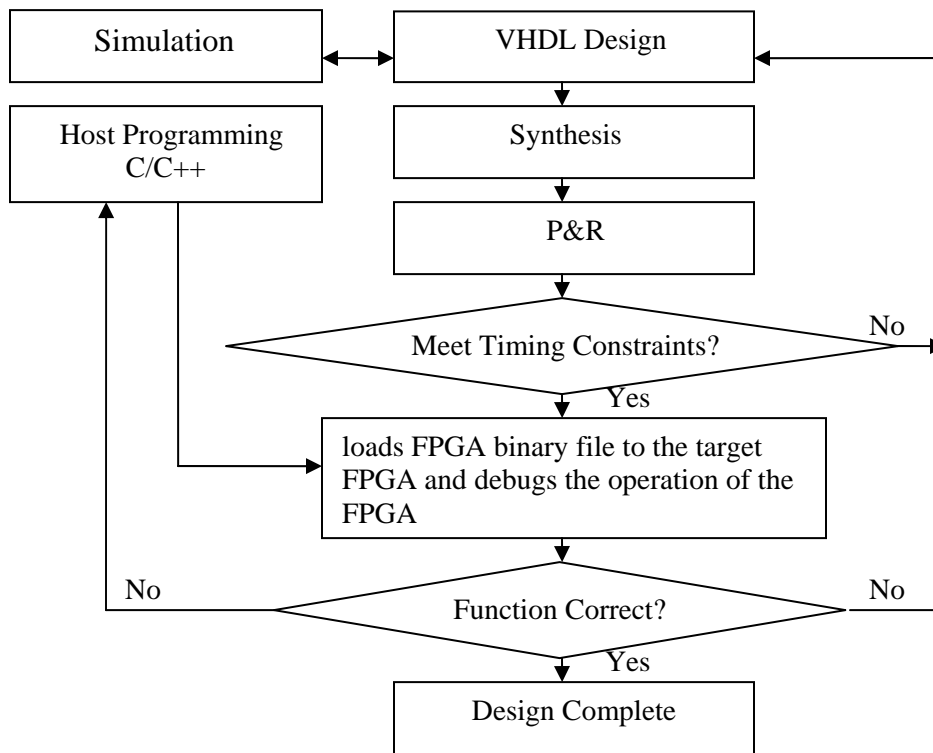


Figure 4-1 : FPGA Implementation Flow

VHDL can specify a hardware design in terms of familiar programming constructs such as conditional statements, loops, and function calls. It provides a flexible and powerful way to generate efficient logic. VHDL code can be written with many text

editors. The target FPGA in the Prototype system is the Processing Element PE0 located on the FIREBIRDTM/PCI board (see 3.1.1), and the target FPGA in the Cray XD1 system is AAP FPGA located on the expansion module board (see 3.2.1.1).

The simulation is done in the simulation environment described in chapter 3 (see 3.1.2 and 3.2.2). In the simulation, VHDL source code and the VHDL models provided in the simulation environment that manipulate the design inputs and monitor the outputs are combined and compiled by the simulation tool. Then the simulation process starts, waveforms for all the signals in the VHDL source code and the VHDL models are derived, checked and verified. The simulation tool used is ModelSim VHDL simulator.

The synthesis process translates the VHDL source code into gate-level elements such as AND gates, OR gates, and flip-flops. The output of the synthesis process is a gate-level netlist. In the Prototype system, the synthesis tool used is Synplify Pro. In the XD1 Cray system, the synthesis tool used is Xilinx XST.

P&R is the process of translating, mapping, placing, routing, and generating an FPGA binary file for the VHDL source code. It assigns logic in the gate-level netlist to specific physical resources of the target FPGA. A set of user-determined timing and placement constraints guides the P&R process. These constraints are in the user constraint file (UCF). The target FPGA type is Xilinx VIRTEX-E XCV1000E in the Prototype system and Xilinx Virtex-II Pro XC2VP50 in the Cray XD1 system. The P&R tool used is part of the Xilinx ISE design suite.

The host application is programmed using the APIs provided in the host programming environment described in chapter 3 (see 3.1.3 and 3.2.3). It loads the FPGA binary file to the target FPGA and debugs the operation of the FPGA.

4.2 FPGA Implementation in the ACL Prototype System

4.2.1 VHDL Design

In this section, the function of the VHDL source code for the FPGA is explained and the FPGA architecture for the VHDL source code is drawn. Next data organization in local memories is explained. At last components and processes in the VHDL source code are described in detail.

4.2.1.1 FPGA Function

The FPGA can be used as a bridge to take data from the host system and to write it to local memories, to read data from local memories and to write it to the host system. It can also be used as FDTD algorithm update engines.

If the FPGA is used as a bridge, configuration data should be written into internal registers in the FPGA first. The configuration data indicates the local memories access type (read or write), the start address of the accessed local memories and the length of the data. Then data transfer between the host system and the local memories starts. Data are transferred through the FPGA in DMA mode.

If the FPGA is used as FDTD algorithm update engines, it can only complete one iteration of FDTD algorithm calculation. The host system must write input data to the local memories before each iteration of calculation and read output data back from the local memories after each iteration of calculation. The FPGA can be configured to operate in two modes: UpdateHxHy mode and UpdateEz mode. In UpdateHxHy mode, update engine magnetic update Hx and magnetic update Hy are enabled to execute one

iteration of calculation. In UpdateEz mode, update engine electric update is enabled to execute one iteration of calculation.

The operation sequence in UpdateHxHy mode is as follows:

- 1) $H_{x_{t-1}}$ values are read from local memory MEM0; $H_{y_{t-1}}$ values are read from local memory MEM1; Ez values are read from local memory MEM2. These read operations are done in parallel
- 2) $H_{x_{t-1}}$ values, $H_{y_{t-1}}$ values, Ez values are fed to internal FIFOs in the FPGA in parallel
- 3) $H_{x_{t-1}}$ values, and Ez values in internal FIFOs are fed to the enabled update engine magnetic update Hx; $H_{y_{t-1}}$ values, and Ez values in internal FIFOs are fed to the enabled update engine magnetic update Hy.
- 4) When the result data H_{x_t} values and H_{y_t} values from update engine magnetic update Hx and magnetic update Hy are ready, they are pushed into another two FIFOs in the FPGA
- 5) H_{x_t} values are pulled out of the FIFO and written to local memory MEM3, H_{y_t} values are pulled out of the FIFO and written to local memory MEM4.


Similarly, the operation sequence in UpdateEz mode is as follows:

- 1) Hx values are read from local memory MEM0; Hy values are read from local memory MEM1; Ez_{t-1} values are read from local memory MEM2. These read operations are done in parallel
- 2) Hx values, Hy values and Ez_{t-1} values are fed to internal FIFOs in the FPGA in parallel


- 3) Hx values, Hy values and Ez_{t-1} values in internal FIFOs are fed to the enabled update engine electric update
- 4) When the result data Ez_t values from update engine electric update are ready, they are pushed into another FIFO in the FPGA
- 5) Ez_t values are pulled out of the FIFO and written to local memory MEM3


4.2.1.2 FPGA Architecture


FPGA architecture is drawn in Figure 4-2. Definitions for the symbols used in figure 4-2 are:

: *external interface*

: *internal bus*

: *internal signal*

: *component in the VHDL source code*

: *process in the VHDL source code*

In the FPGA architecture, components with prefix “LAD” are related to the LAD bus. The host system communicates with the FPGA through these components. With the aid of these components, the host system can access internal registers in the FPGA and the local memories.

The memory interface in the FPGA consists of components with prefix “Mem”. This memory interface multiplexes the read/write requests from the host system and the update engines in the FPGA. The host system and the update engines can’t access the same memory at the same time.

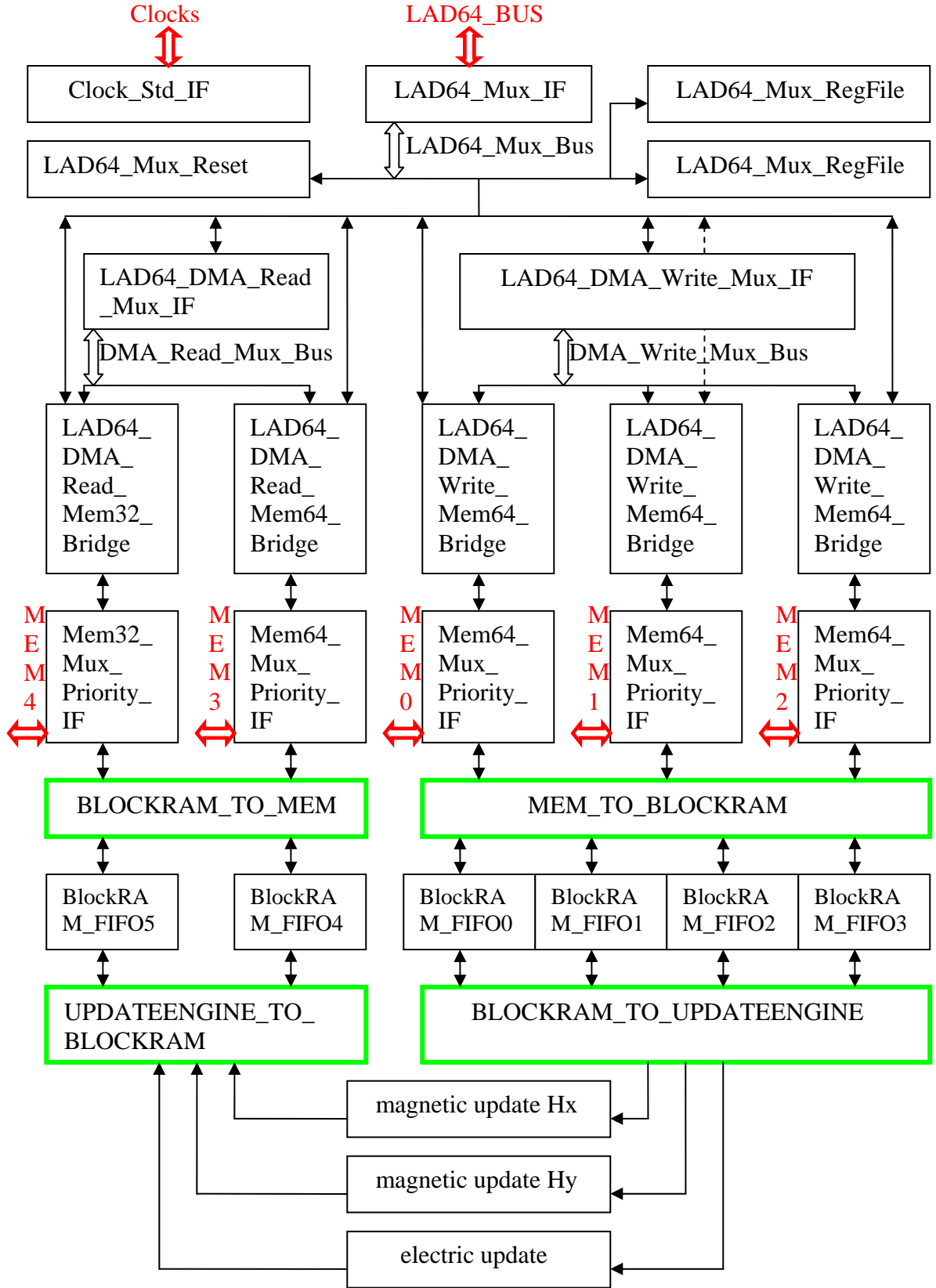


Figure 4-2 : FPGA Architecture in the Prototype System

Three update engines are used for FDTD algorithm calculation. Four processes are used for the update engines to access the local memories.

Because the local memories have random delay, internal FIFOs are used when the update engines access the local memories. This increases the complexity and decreases the performance of the FPGA design.

4.2.1.3 Data Organization in Local Memories

There are five independent local memories on the FIREBIRDTM/PCI board (see 3.1.1). MEM0 to MEM3 are 64-bit wide, MEM4 is 32-bit wide. MEM0 to MEM2 are used to store source data, MEM3 to MEM4 are used to store destination data. In detail, MEM0 stores $H_{x,t-1}$ values, MEM1 stores $H_{y,t-1}$ values, MEM2 stores $E_{z,t-1}$ values, MEM3 stores $H_{x,t}$ values or $E_{z,t}$ values, MEM4 stores $H_{y,t}$ values.

Assume the electromagnetic field processed by the FPGA is a Grid_Row×Grid_Column matrix. Then Hx values, Hy values and Ez values in this matrix can be organized as two-dimensional data arrays. The range for Hx values is from $H_x[0][0]$ to $H_x[\text{Grid_Row}-1][\text{Grid_Column}-1]$, the range for Hy values is from $H_y[0][0]$ to $H_y[\text{Grid_Row}-1][\text{Grid_Column}-1]$, the range for Ez values is from $E_z[0][0]$ to $E_z[\text{Grid_Row}-1][\text{Grid_Column}-1]$.

$H_{x,t-1}[i][j]$ in the electromagnetic field matrix is stored in MEM0 at address $i \times \text{Grid_Column} + j$, $H_{y,t-1}[i][j]$ is stored in MEM1 at address $i \times \text{Grid_Column} + j$, $E_{z,t-1}[i][j]$ is stored in MEM2 at address $(i-1) \times \text{Grid_Column} + (j-1)$, $H_{x,t}[i][j]$ is stored in MEM3 at address $i \times \text{Grid_Column} + j$, $H_{y,t}[i][j]$ is stored in MEM4 at address $i \times \text{Grid_Column} + j$, $E_{z,t}[i][j]$ is stored in MEM3 at address $(i-1) \times \text{Grid_Column} + (j-1)$.

4.2.1.4 Components and Processes

Components in the FPGA architecture can be divided into four groups:

- update engines
- components related to LAD bus
- components related to memory interface
- internal FIFO

Processes in the FPGA architecture include:

- MEM_TO_BLOCKRAM
- BLOCKRAM_TO_UPDATEENGINE
- UPDATEENGINE_TO_BLOCKRAM
- BLOCKRAM_TO_MEM

There are three update engines in the FPGA. They are used to update magnetic field and electric field values. To improve speed, these update engines are fully pipelined.

- *magnetic update H_x* : updates magnetic field values along the x-coordinate. It

implements the following equation:

$$H_{x_t}[i][j] = H_{x_{t-1}}[i][j] - dtmudy \times (E_z[i+1][j+1] - E_z[i+1][j]) \quad (3a)$$

This equation is transformation of equation (2a) in the FDTD algorithm in chapter 2.

H_{x_t} : the magnetic field value along the x-coordinate

$H_{x_{t-1}}$: the magnetic field value along the x-coordinate from the previous time step

E_z : the electric field value along the z-coordinate

$dtmudy$: constant that includes time step, grid spacing, and magnetic permeability

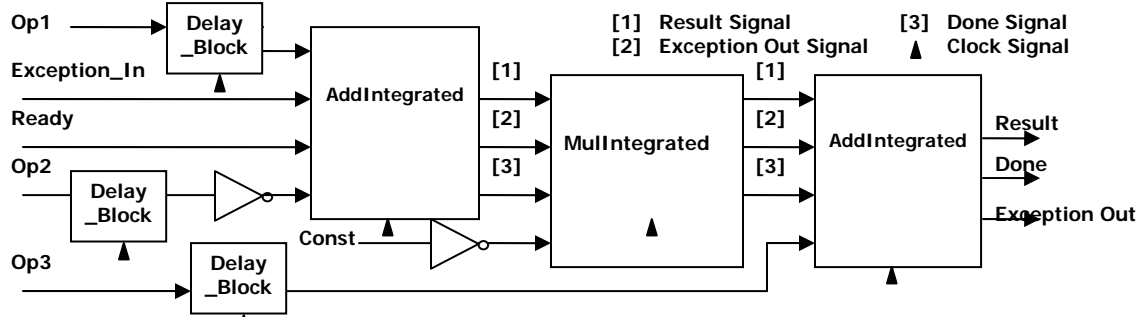


Figure 4-3 Block Diagram of Magnetic update H_x [21]

- *magnetic update H_y* : updates magnetic field values along the y-coordinate. It

implements the following equation:

$$H_{y_t}[i][j] = H_{y_{t-1}}[i][j] + dtmudx \times (E_z[i+1][j+1] - E_z[i][j+1]) \quad (3b)$$

This equation is transformation of equation (2b) in the FDTD algorithm in chapter 2.

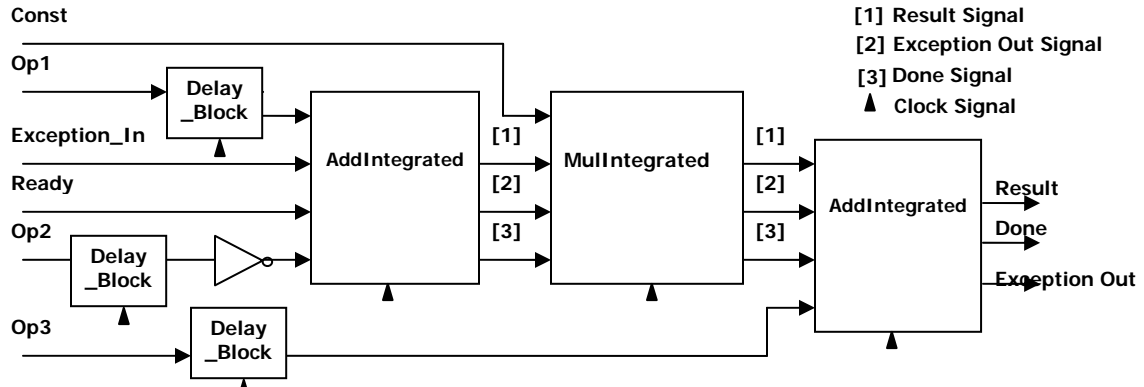


Figure 4-4 Block Diagram of Magnetic update H_y [21]

- *electric update* : updates electric field values along the z-coordinate. It implements

the following equation:

$$E_{z_t}[i][j] = E_{z_{t-1}}[i][j] + dt\epsilon_s dx \times (H_y[i][j-1] - H_y[i-1][j-1]) - dt\epsilon_s dy \times (H_x[i-1][j] - H_x[i-1][j-1]) \quad (3c)$$

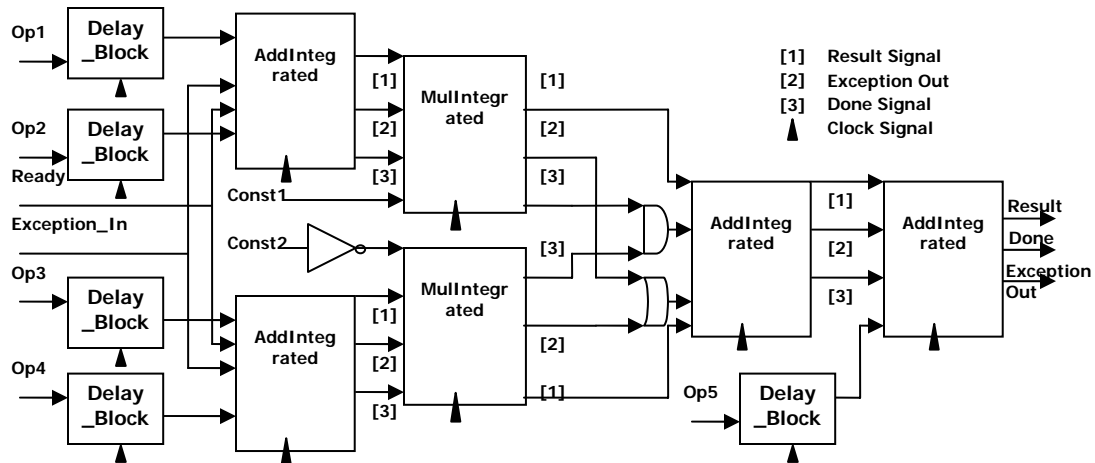


Figure 4-5 Block Diagram of Electric update [21]

The FPGA communicates with the host system through the LAD bus. Components related to LAD bus in the FPGA include [24]:

- *LAD64_Mux_IF* : connects internal components in PE0 to the PCI Controller on the FIREBIRD™/PCI board. It has master port LAD64_BUS and clients port LAD64_Mux_Bus. LAD64_BUS is a local address data (LAD) bus. It is a single master, 64-bit, shared address/data bus. Every cycle on it is initiated by the PCI Controller and may last from four to hundreds of clock cycles. LAD64_Mux_Bus is clients bus, the internal components in PE0 such as LAD64_Mux_Reset and LAD64_Mux_RegFile are connected to it
- *LAD64_Mux_Reset* : provides an LAD accessible reset unit for PE0. It encapsulates a VIRTEX_STARTUP block. When the host application writes a '1' to its control address, it will generate a reset pulse on the global reset line of PE0

- *LAD64_Mux_RegFile* : provides an LAD accessible register file on PE0. Each register in the file is 64-bits. Configuration data such as version number and update type are written into the register file in this component

- *LAD64_DMA_Read_Mux_IF* : used with component

LAD64_DMA_Read_Mem64_Bridge and *LAD64_DMA_Read_Mem32_Bridge* to read data from the local memory on the FIREBIRD™ /PCI board. It receives DMA data from *LAD64_DMA_Read_Mem64_Bridge* and *LAD64_DMA_Read_Mem32_Bridge* and transmits the received DMA data on the LAD bus which is connected to the host system. It also controls the DMA read status

- *LAD64_DMA_Write_Mux_IF* : used with component

LAD64_DMA_Write_Mem64_Bridge to write data to the local memory. It receives data from the LAD bus which is connected to the host system, then it transmits the received data on DMA bus which is connected to *LAD64_DMA_Write_Mem64_Bridge*. It also controls the DMA write status

- *LAD64_DMA_Read_Mem64_Bridge* : acts as a pre-fetch unit for the

LAD64_DMA_Read_Mux_IF. It reads data from the 64-bit Mem bus which is connected to the *Mem64_Mux_Priority_IF* and transmits the received data to the DMA bus which is connected to the *LAD64_DMA_Read_Mux_IF*. It has ports LAD, Mem and DMA. Port LAD is connected to the *LAD64_Mux_IF* and receives control data from the host system. *LAD64_DMA_Read_Mem64_Bridge* contains a register file *CRegfile* which is used to control the start and stop memory addresses. It also contains a status record which can be used to determine if a DMA read is being performed and what percentage of the DMA read has been completed

- *LAD64_DMA_Read_Mem32_Bridge* : acts as a pre-fetch unit for the

LAD64_DMA_Read_Mux_IF. It reads data from the 32-bit Mem bus which is connected to the *Mem32_Mux_Priority_IF* and transmits the received data to the DMA bus which is connected to the *LAD64_DMA_Read_Mux_IF*. It has ports LAD, Mem and DMA. Port LAD is connected to the *LAD64_Mux_IF* and receives control data from the host system. *LAD64_DMA_Read_Mem32_Bridge* contains a register file *CRegfile* which is used to control the start and stop memory addresses. It also contains a status record which can be used to determine if a DMA read is being performed and what percentage of the DMA read has been completed

- *LAD64_DMA_Write_Mem64_Bridge* : used with *LAD64_DMA_Write_Mux_IF* to take data from the host system and write it to the 64-bit local memory. It reads data from DMA bus connected to *LAD64_DMA_Write_Mux_IF* and transmits the received data to the Mem bus connected to *Mem64_Mux_Priority_IF*. It has ports LAD, Mem and DMA. Port LAD is connected to the *LAD64_Mux_IF* and receives control data from the host system. *LAD64_DMA_Write_Mem64_Bridge* contains a register file *CRegfile* which is used to control the start and stop memory addresses. It also contains a status record which can be used to determine if a DMA write is being performed and what percentage of the DMA write has been completed

The FPGA accesses the local memories through a 64-bit Memory Standard Interface and a 32-bit Memory Standard Interface. Components related to these memory interfaces in the FPGA include [24]:

- *Mem64_Mux_Priority_IF* : a 64-bit memory server in the PE0. It multiplexes memory ports among different components that are clients contained in the PE0 and is

capable of providing 64-bit memory access to these clients. For example, for the local memory MEM0 in figure 4-2, Mem64_Mux_Priority_IF provides memory access to clients LAD64_DMA_Write_Mem64_Bridge and MEM_TO_BLOCKRAM. Each client must be assigned a unique element of the client vector in order to be connected to the 64-bit memory server

- *Mem32_Mux_Priority_IF* : a 32-bit memory server in the PE0. It multiplexes memory ports among different components that are clients contained in the PE0 and is capable of providing 32-bit memory access to these clients

Internal FIFO used in the FPGA is component *BlockRAM_FIFO*. This is a 256x32 FIFO. 32-bit data can be pushed into it and pulled out of it. If the FIFO is almost full, Full Flag is set. If the FIFO is almost empty, then Empty Flag is set.

There are four processes in the FPGA. These processes control the memory access of the update engines.

- *MEM_TO_BLOCKRAM* : reads data from three 64-bit local memories (MEM0 to MEM2) and feeds the data to four BLOCKRAM_FIFOs (BLOCKRAM_FIFO0 to BLOCKRAM_FIFO3). Local memory MEM0 is connected to BLOCKRAM_FIFO0, Local memory MEM1 is connected to BLOCKRAM_FIFO1, Local memory MEM2 is connected to BLOCKRAM_FIFO2 and BLOCKRAM_FIFO3. If PE0 is in UpdateHxHy mode or UpdateEz mode and the four BLOCKRAM_FIFOs have spare space, read requests are issued to the three local memories. When the output data from the three local memories are ready, they are pushed into the four BLOCKRAM_FIFOs. If PE0 is in UpdateHxHy mode, the pseudo-code to generate read addresses to the three local memories is:

for update engine magnetic update H_x:

for i from 0 to Grid_Row-1

for j from 1 to Grid_Column-1

generate read address $i \times \text{Grid_Column} + j - 1$ for MEM0

generate read address $i \times \text{Grid_Column} + j - 1$ for MEM2

generate read address $i \times \text{Grid_Column} + j$ for MEM2

for update engine magnetic update H_y:

for i from 1 to Grid_Row-1

for j from 0 to Grid_Column-1

generate read address $(i - 1) \times \text{Grid_Column} + j$ for MEM1

generate read address $(i - 1) \times \text{Grid_Column} + j$ for MEM2

generate read address $i \times \text{Grid_Column} + j$ for MEM2

If PE0 is in UpdateEz mode, the pseudo-code to generate read addresses to the three local memories is:

for update engine electric update

for i from 1 to Grid_Row-1

for j from 1 to Grid_Column-1

generate read address $(i - 1) \times \text{Grid_Column} + j - 1$ for MEM0

generate read address $(i - 1) \times \text{Grid_Column} + j$ for MEM0

generate read address $(i - 1) \times \text{Grid_Column} + j - 1$ for MEM1

generate read address $i \times \text{Grid_Column} + j - 1$ for MEM1

generate read address $(i - 1) \times \text{Grid_Column} + j - 1$ for MEM2

- *BLOCKRAM_TO_UPDATEENGINE* : reads data from four BLOCKRAM_FIFOs (BLOCKRAM_FIFO0 to BLOCKRAM_FIFO3) and feed the data to update engines. It monitors the status of the four BLOCKRAM_FIFOs and read data from them. Then it enables update engines magnetic update Hx and magnetic update Hy and feeds data to them if PE0 is in UpdateHxHy mode or enables update engine electric update and feeds data to it if PE0 is in UpdateEz mode.
- *UPDATEENGINE_TO_BLOCKRAM* : reads data from update engines and writes it to BlockRAM_FIFOs (BLOCKRAM_FIFO4 to BLOCKRAM_FIFO5). If PE0 is in UpdateHxHy mode, result of update engine magnetic update Hx is pushed to BlockRAM_FIFO4 and result of update engine magnetic update Hy is pushed to BlockRAM_FIFO5. If PE0 is in UpdateEz mode, result of update engine electric update is pushed to BlockRAM_FIFO4.
- *BLOCKRAM_TO_MEM* : reads data from BlockRAM_FIFOs (BLOCKRAM_FIFO4 to BLOCKRAM_FIFO5) and writes the data into two local memories (MEM3, MEM4). If PE0 is in UpdateHxHy mode, data from BLOCKRAM_FIFO4 is written to MEM3 and data from BLOCKRAM_FIFO5 is written to MEM4. If PE0 is in UpdateEz mode, data from BLOCKRAM_FIFO4 is written to MEM3. If PE0 is in UpdateHxHy mode, the pseudo-code to generate write addresses to the two local memories is:

for i from 0 to Grid_Row-1

for j from 1 to Grid_Column-1

generate write address $i \times \text{Grid_Column} + j - 1$ for MEM3

for i from 1 to Grid_Row-1

for j from 0 to Grid_Column-1

generate write address $(i - 1) \times \text{Grid_Column} + j$ for MEM4

If PE0 is in UpdateEz mode, the pseudo-code to generate write addresses to the three local memories is:

for i from 1 to Grid_Row-1

for j from 1 to Grid_Column-1

generate write address $(i - 1) \times \text{Grid_Column} + j - 1$ for MEM3

4.2.2 Simulation

The simulation is done in the simulation environment described in chapter 3 (see 3.1.2). The test bench includes and instantiates the FPGA, VHDL model for the host system and the FIREBIRD™/PCI board. The operation sequence in the test bench is the same as the operation sequence in the host application described in 4.2.5.

Structure of the test bench is depicted in figure 4-6.

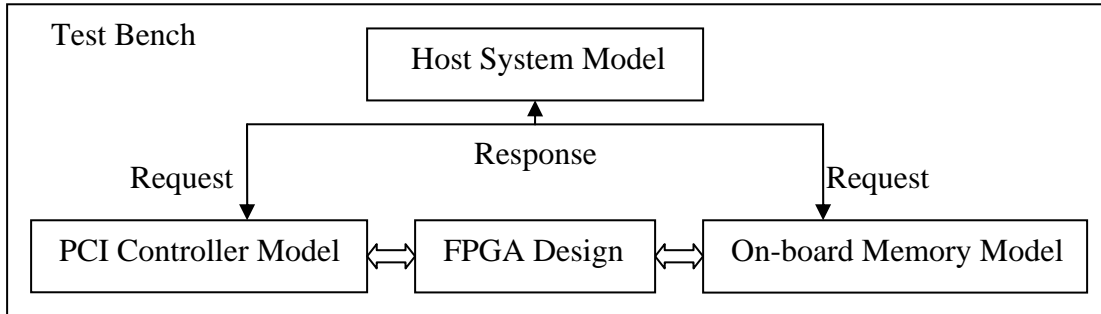


Figure 4-6 Structure of the Test Bench in the Prototype System

4.2.3 Synthesis

The VHDL source code for the FPGA is synthesized with Synplify Pro. The output of synthesis is an EDIF file.

4.2.4 P&R

The EDIF file is placed and routed on the target FPGA Xilinx VIRTEX-E XCV1000E. The output file is a FPGA binary file for the VHDL source code.

4.2.5 Host Programming

The host application is programmed with 'C' language in the host programming environment described in chapter 3 (see 3.1.3). In order to interact with PE0, it calls WILDSTAR™ API functions.

The operation sequence in the host application is as follows:

- 1) Open the FIREBIRD™/PCI board and check configuration information
- 2) Configure the clocks on the board
- 3) Download FPGA binary file into XCV1000E
- 4) Reset FPGA, update engines and interrupt signal
- 5) Write $H_{x_{t-1}}$ values in DMA mode to local memory MEM0
- 6) Write $H_{y_{t-1}}$ values in DMA mode to local memory MEM1
- 7) Write E_z values in DMA mode to local memory MEM2
- 8) Enable update engines magnetic update Hx and magnetic update Hy
- 9) Check interrupt signal. If interrupt signal is set, the update process ends
- 10) Read H_{x_t} values in DMA mode from local memory MEM3
- 11) Read H_{y_t} values in DMA mode from local memory MEM3
- 12) Write Hx values in DMA mode to local memory MEM0
- 13) Write Hy values in DMA mode to local memory MEM1
- 14) Write $E_{z_{t-1}}$ values in DMA mode to local memory MEM2

- 15) Enable update engines electric update
- 16) Check interrupt signal. If interrupt signal is set, the update process ends
- 17) Read Ez_t values in DMA mode from local memory MEM3
- 18) Unload FPGA binary file from XCV1000E
- 19) Close the FIREBIRD™/PCI board

The Prototype System is very simple. There are only one host processor and one FPGA in the system. So thread-level parallelism in the FDTD algorithm can't be exploited. Local SRAMs attached to the FPGA are single-port with random delay and low speed. These low-performance SRAMs can't support parallelism among the update engines and can only support one iteration of FDTD algorithm calculation.

4.3 FPGA Implementation in the Cray XD1 System

In order to overcome the limitations of the Prototype System and to improve the performance, the FPGA implementation is done in another reconfigurable system named Cray XD1 System. In the Cray XD1 system, there are lots of host processors and FPGAs. The FDTD algorithm can be partitioned and distributed to the host processors and the FPGAs, thus thread-level parallelism can be exploited. Furthermore, local SRAMs in the Cray XD1 system are dual-port with fixed delay and high speed. These SRAMs can support parallelism among the update engines and specified number of iterations of FDTD algorithm calculation. So performance of the FPGA design in the Cray XD1 system improves greatly compared with that in the Prototype system.

4.3.1 VHDL Design

In the Cray XD1 system, the FPGA which is referred to as the AAP FPGA is located on the expansion module board (see 3.2.1.1). In this section, the function of the VHDL source code for the FPGA is explained and the FPGA architecture for the VHDL source code is drawn. Then data organization in local memories is explained. At last components and processes in the VHDL source code are described in detail.

4.3.1.1 FPGA Function

The FPGA can be used as a bridge to take data from the host system and to write it to local memories, to read data from local memories and to write it to the host system. It can also be used for the FDTD algorithm update engines.

If the FPGA is used as a bridge, local memories QDR II SRAM 1 to QDR II SRAM 4 are mapped to the address space of the host system. Read commands from this address space and write commands to this address space will be delivered to the FPGA. The FPGA then reads data from the local memories and forwards them to the host system or write the received data to the local memories.

If the FPGA is used for the FDTD algorithm update engines, it must be initialized first. The initialization data are stored in two internal registers `FPGA_reg` and `Update_counter`. These registers are described in 4.3.1.4. After configuration the FPGA will execute specified number of iterations of FDTD algorithm calculation. Throughout this thesis, “one iteration of FDTD algorithm calculation” means magnetic field values update by magnetic update H_x and magnetic update H_y at one time step, or electric field update by electric update at one time step. The exact operation sequence is described in 4.3.1.4 (see component `Qdr_fdt` in 4.3.1.4).

4.3.1.2 FPGA Architecture

FPGA architecture which supports both memory bridge and FDTD operation is drawn in figure 4-7.

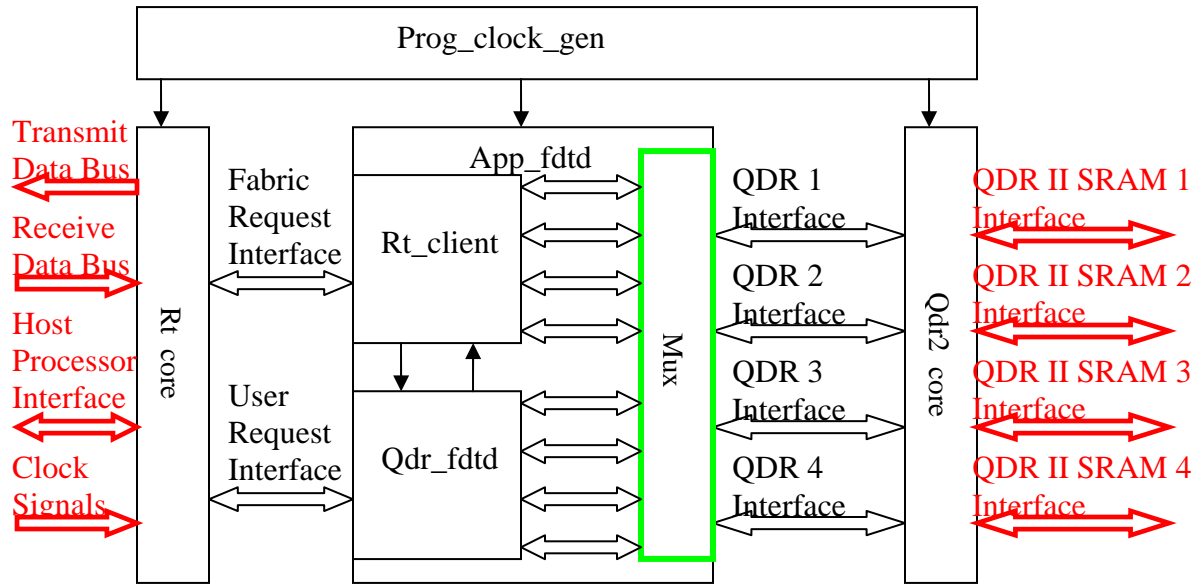


Figure 4-7 FPGA Architecture in the Cray XD1 System

Definitions for the symbols in figure 4-7 are as follows:

↔ : external interface

↔ : internal bus

→ : internal signal

□ : component in the VHDL source code

□ : process in the VHDL source code

The FPGA interacts with the host system using the component Rt_core. The Rt_core delivers requests from the host system to component Rt_client, it also delivers response from Rt_client to the host system.

In the FPGA architecture, the `Rt_client` acts as an agent for the host system to access the local memories and to control component `Qdr_fdttd`.

`Qdr_fdttd` contains three update engines. If enabled, `Qdr_fdttd` can execute a specified number of iterations of FDTD algorithm calculation without interference from the host system. The magnetic update H_x and the magnetic update H_y in `Qdr_fdttd` run in parallel. One H_x result and one H_y result can be produced per clock cycle. During the calculation, `Qdr_fdttd` accesses the local memories exclusively. It reads input data from the local memories and stores intermediate results and output data to the local memories.

`Qdr2_core` is used for the FPGA to access the local memories. Memory access requests from `Rt_client` and from `Qdr_fdttd` are multiplexed by component `Mux`. These requests are then processed by `Qdr2_core`. `Rt_client` and `Qdr_fdttd` can't access the same SRAM at the same time.

The local memories have fixed delay, so no internal FIFOs are needed for the FPGA to access the local memories. This simplifies the FPGA design and improves the memory access speed.

4.3.1.3 Data Organization in Local Memories

There are four independent local memories (QDR II SRAM 1 to QDR II SRAM 4) on the expansion module board (see 3.2.1.1). The QDR II SRAMs are 36-bit wide high speed, low latency memories. Each QDR II SRAM has 2 ports hence is capable of sustaining simultaneous, single clock, read and write accesses at clock speeds of up to 199 MHz [31]. QDR II SRAM 1 is used to store $H_{x_{t-1}}$ values and H_{x_t} values, QDR II

SRAM 2 is used to store $H_{y_{t-1}}$ values and H_{y_t} values, QDR II SRAM 3 and 4 are used to store $E_{z_{t-1}}$ values and E_{z_t} values.

The QDR II SRAMs are dual-port and the data stored in them is organized as ping-pong buffers. The ping-pong mechanism works like this: for cycle N , if the input data are stored in the lower (upper) parts of the SRAMs, then output data are stored in the upper (lower) parts of the SRAMs. Then during the next cycle $N+1$, the input data are read from the upper (lower) parts of the SRAMs and the output data are stored in the lower (upper) parts of the SRAMs. This process can continue until the specified number of calculation iterations is reached. This ping-pong mechanism enables the FPGA to execute many iterations of calculation with little communication with the host system. Reduction of data transfer between the host system and the FPGA during intervals between calculation iterations greatly improves the performance of the whole system.

The $H_{x_{t-1}}$ values and H_{x_t} values are stored in QDR II SRAM 1 in row major order, The $H_{y_{t-1}}$ values and H_{y_t} values are stored in QDR II SRAM 2 in column major order, the $E_{z_{t-1}}$ values and E_{z_t} values are stored in QDR II SRAM 3 in row major order and stored in QDR II SRAM 4 in column major order. Assume the electromagnetic field processed by the FPGA is a $\text{Grid_Row} \times \text{Grid_Column}$ matrix, the row major order means value $V[i][j]$ in the matrix is stored at the address $i \times \text{Grid_Column} + j$ or $\text{Grid_Row} \times \text{Grid_Column} + i \times \text{Grid_Column} + j$ in the QDR II SRAM, the column major order means value $V[i][j]$ in the matrix is stored at the address $j \times \text{Grid_Row} + i$ or $\text{Grid_Row} \times \text{Grid_Column} + j \times \text{Grid_Row} + i$ in the QDR II SRAM.

The row major order and the column order are adopted so that Qdr_fdttd can utilize these orders to improve throughput of the update engines. Utilizing these orders and

parallelism between magnetic update Hx and magnetic update Hy, Qdr_fdttd can produce one Hx result and one Hy result per clock cycle or one Ez result every two clock cycles.

4.3.1.4 Components and Processes

The components in the FPGA architecture include:

- Prog_clock_gen
- Rt_core
- Qdr2_core
- Rt_client
- Qdr_fdttd
- Mux

Component Prog_clock_gen generates the programmable global clocks.

Component Rt_core is provided by Cray Inc. It connects internal components in the FPGA to external devices such as SMPs and other AAP FPGAs connected to the RapidArray Interconnect (see 3.2.1.2) in the Cray XD1 system. It allows internal components to access the external devices, and it also allows the external devices to access internal registers and local memories of the FPGA.

Component Rt_core is connected to the external devices through ports Transmit Data Bus, Receive Data Bus, Host Processor Interface, Clock Signals, and it provides Fabric Request Interface and User Request Interface to internal components in the FPGA. Ports Transmit Data Bus and Receive Data Bus are used to transfer data between Rt-core and the external devices. The host Processor Interface is provide to the external devices to set the frequency of the programmable global clocks and to reset the FPGA. The

Fabric Request Interface is used by Rt_core to forward the received remote memory access requests from the external devices to Rt_client. It is also used to accept responses from Rt_client, then the responses are forwarded by Rt_core to the external devices. The User Request Interface can be used to accept memory access requests from internal components in the FPGA. Then these requests are forwarded by Rt_core to the external devices. It also can deliver the received responses from the external devices to the internal components. But in the current FPGA design, the User Request Interface is not used.

Component Qdr2_core is provided by Cray Inc. It provides an interface for the FPGA to access the four external 36 bit wide QDR II SRAMs. It allows the internal components in the FPGA to simultaneously read and write the external QDR II SRAMs at a clock speeds from 130 MHz up to 199 MHz. Qdr2_core is made up of four fully independent RAM interface blocks, each connected to an external QDR II SRAM. The four RAM interfaces blocks provide full control of the QDR II SRAMs including address, read data, write data, read and write enables. This allows the four QDR II SRAMs to be arranged in many different bank configurations [30].

Component Mux multiplexes memory access buses from Rt_client and Qdr_fdt. After FPGA reset, memory access bus from Rt_client is connected to the qdr2_core and the four QDR II SRAMs can only be accessed by the external devices connected to the RapidArray Interconnect. If update engines in Qdr_fdt is enabled, then memory access bus from Qdr_fdt is connected to the qdr2_core. When the update process completes, the qdr2_core is switched back to Rt_client.

Component Rt_client acts as an agent for the host system to access the local memories and to control component Qdr_fdttd. It processes requests from the external devices and gives response through the Fabric Request Interface, it also monitors the status of Qdr_fdttd and controls the operation of Qdr_fdttd. When it receives requests from the Fabric Request Interface, it checks if the request are read requests or write requests, it also checks the request addresses to tell if the requests should be directed to the internal registers or the external QDR II SRAMs. If the requests are read requests to the internal registers, then data stored in the internal registers are sent as response to the Fabric Request Interface. If the requests are write requests to the internal registers, then data from the Fabric Request Interface are stored into the internal registers and no response is given to the Fabric Request Interface. If the requests are read requests to the external QDR II SRAMs, Rt_client issues memory read requests to component Mux and waits for response from the external QDR II SRAMs. When Rt_client receives the response from Mux, it will forward the response to the Fabric Request Interface. If the requests are write requests to the external QDR II SRAMs, Rt_client issues memory write requests to Mux and completes the requests without waiting for the response.

Component Rt_client contains two internal registers: Update_counter and FPGA_reg. These two registers are used to control the operation of Qdr_fdttd and to monitor the status of Qdr_fdttd. Update_counter is a 32-bit wide register. It specifies the iteration number of the FDTD algorithm calculation Qdr_fdttd should complete. FPGA_reg is a 64-bit wide register used by the host to specify parameters and initiate execution of the Qdr_fdttd engine. The parameters in FPGA_reg are shown in figure 4-8 and described below:

bit 63-48	47-32	31	30	29
Grid_Row	Grid_Column	Access_type	Update_type	HxHy_Source_Location
28	27-2	1	2	
Ez_Source_Location		Update_starter	Update_end	

Figure 4-8 : Parameters in Register FPGA_reg

- Grid_Row : the number of rows of the electromagnetic field matrix
- Grid_Column : the number of columns of the electromagnetic field matrix
- Access_type : if this bit is set to '0', then memory access bus from Rt_client is connected to Qdr2_core enabling the host to access the QDR II SRAMs; if this bit is set to '1', then memory access bus from Qdr_fdttd is connected to Qdr2_core and the update engines has access to the QDR II SRAMs
- Update_type : if this bit is set to '0', then in the first iteration of the FDTD algorithm calculation, update engines magnetic update Hx and magnetic update Hy are enabled; if this bit is set to '1', then in the first iteration of the FDTD algorithm calculation, update engine electric update is enabled
- HxHy_Source_Location : this bit determines which ping-pong buffer to read for Hx and Hy data. If this bit is set to '0', then in the first iteration of the FDTD algorithm calculation, source data value $H_{x,t-1}[i][j]$ is stored in QDR II SRAM 1 at address $i \times \text{Grid_Column} + j$, source data value $H_{y,t-1}[i][j]$ is stored in QDR II SRAM 2 at address $j \times \text{Grid_Row} + i$; If this bit is set to '1', then in the first iteration of the FDTD algorithm calculation, source data value $H_{x,t-1}[i][j]$ is stored in QDR II SRAM 1 at address $\text{Grid_Row} \times \text{Grid_Column} + i \times \text{Grid_Column} + j$, source data value $H_{y,t-1}[i][j]$ is stored in QDR II SRAM 2 at address

$\text{Grid_Row} \times \text{Grid_Column} + j \times \text{Grid_Row} + i$

- **Ez_Source_Location** : this bit determines which ping-pong buffer to read for Ez data. If this bit is set to '0', then in the first iteration of the FDTD algorithm calculation, source data value $E_{z,t-1}[i][j]$ is stored in QDR II SRAM 3 at address $i \times \text{Grid_Column} + j$ and in QDR II SRAM 4 at address $j \times \text{Grid_Row} + i$; If this bit is set to '1', then in the first iteration of the FDTD algorithm calculation, source data value $E_{z,t-1}[i][j]$ is stored in QDR II SRAM 3 at address $\text{Grid_Row} \times \text{Grid_Column} + i \times \text{Grid_Column} + j$ and in QDR II SRAM 4 at address $\text{Grid_Row} \times \text{Grid_Column} + j \times \text{Grid_Row} + i$
- **Update_starter** : when this bit toggles, update engines in Qdr_fdttd start FDTD algorithm calculation
- **Update_end** : when update engines in Qdr_fdttd complete calculation, this bit is set by Qdr_fdttd to '1', otherwise it is '0'

Component Qdr_fdttd can do any iterations of FDTD algorithm calculation under control of Rt_client. The procedure for FDTD algorithm calculation is depicted in figure 4-9.

The Hx, Hy values update procedure in figure 4-9 is as follows:

- 1) Generate read addresses for the external QDR II SRAMs
- 2) Read data from the external QDR II SRAMs. Data from QDR II SRAM 1 are fed to operand "op3" in update engine magnetic update Hx, data from QDR II SRAM 3 are fed to "op1" and "op2" in magnetic update Hx, data from QDR II SRAM 2 are fed to "op3" in magnetic update Hy, data from QDR II SRAM 4 are fed to "op1" and "op2" in magnetic update Hy

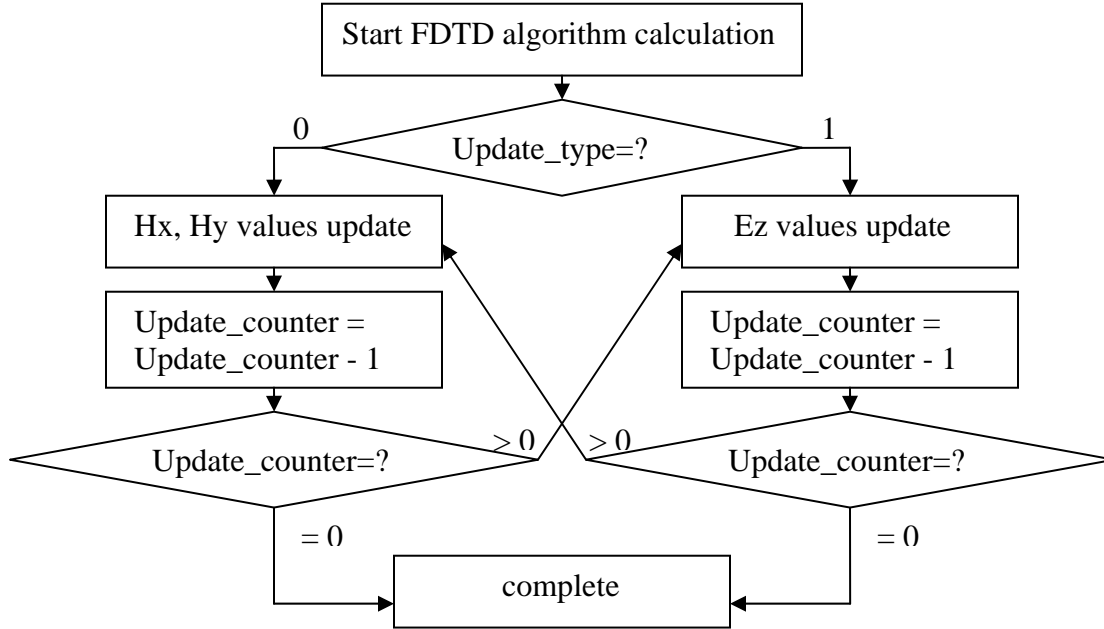


Figure 4-9 : Procedure for FDTD algorithm calculation

- 3) Magnetic update Hx and magnetic update Hy are enabled and run in parallel.

Calculation results will appear after a fixed number of clock cycles. One Hx result and one Hy result will be produced every clock cycle

- 4) Generate write addresses for the external QDR II SRAMs
- 5) Write Hx calculation results from magnetic update Hx to QDR II SRAM 1 and write Hy calculation results from magnetic update Hy to QDR II SRAM 2

The Ez values update procedure in figure 4-9 is as follows:

- 1) Generate read addresses for the external QDR II SRAMs
- 2) Read data from the external QDR II SRAMs. Data from QDR II SRAM 1 are fed to “op3” and “op4” in electric update, data from QDR II SRAM 2 are fed to “op1” and “op2” in electric update, data from QDR II SRAM 3 are fed to “op5” in electric update

- 3) Electric update is enabled, and calculation results will appear after a fixed number of clock cycles. One Ez result will be produced every two clock cycles
- 4) Generate write addresses for the external QDR II SRAMs
- 5) Write Ez calculation results from electric update to QDR II SRAM 3 and 4

The order to generate read addresses for the external QDR II SRAMs in step 1) during the Hx, Hy values update procedure is shown with arrows in figure 4-10.

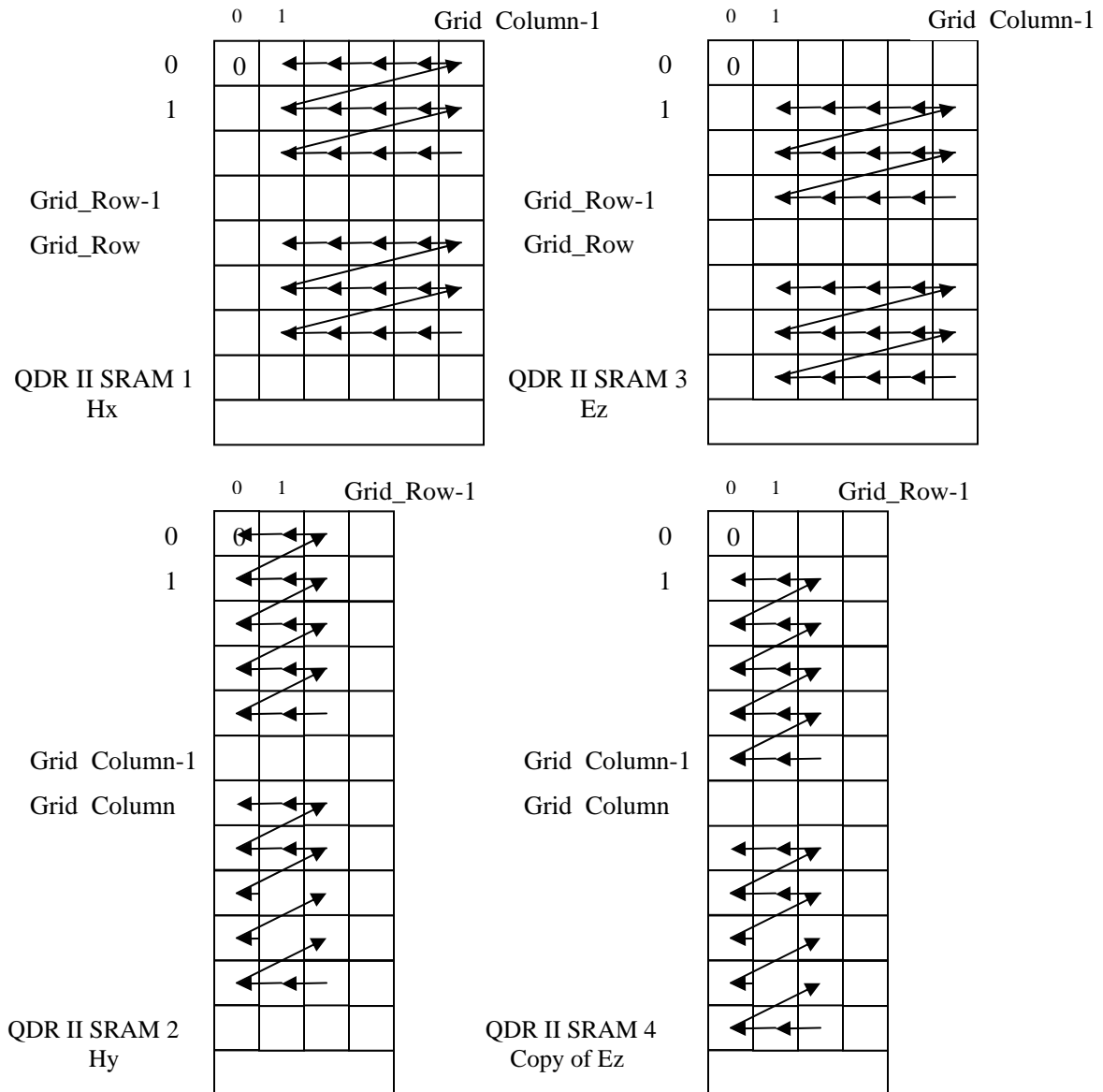


Figure 4-10 Read Addresses Generation Order for Hx, Hy values update

The order to generate write addresses for the external QDR II SRAMs in step 4) during the Hx, Hy values update procedure is shown with arrows in figure 4-11.

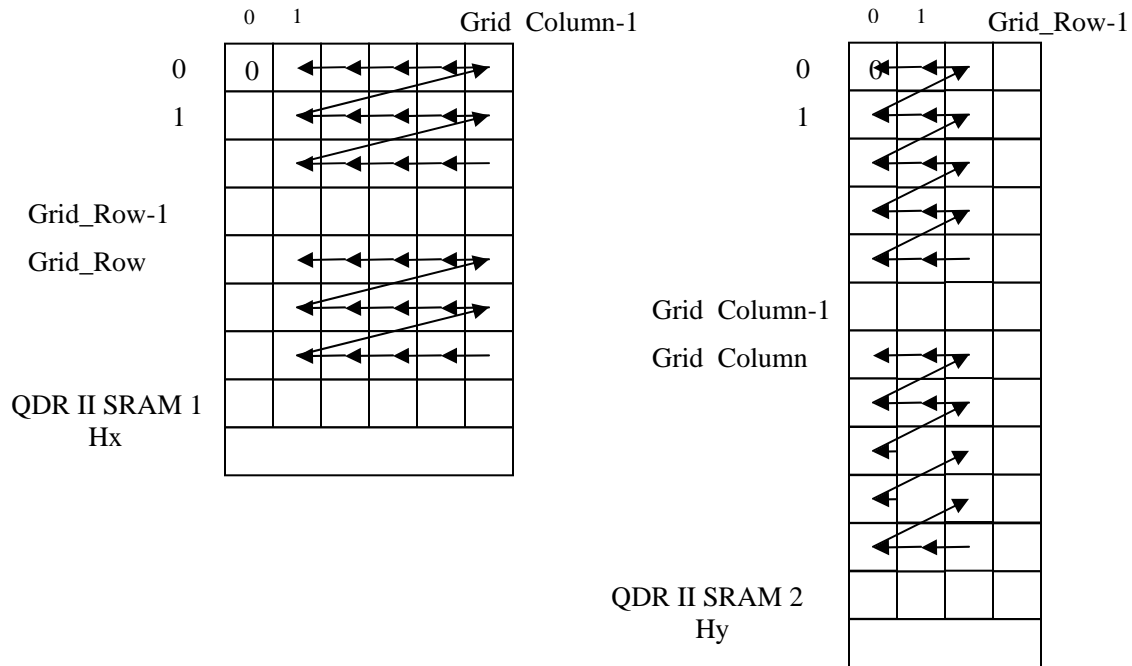


Figure 4-11 Write Addresses Generation Order for Hx, Hy values update

The order to generate read addresses for the external QDR II SRAMs in step 1) during the Ez values update procedure is shown with arrows in figure 4-12.

The order to generate write addresses for the external QDR II SRAMs in step 4) during the Ez values update procedure is shown with arrows in figure 4-13.

Component Qdr_fdttd can update one Hx value and one Hy value every clock cycle. Every clock cycle Qdr_fdttd reads one Hx value from QDR II SRAM 1, one Hy value from QDR II SRAM 2, two Ez values from QDR II SRAM 3, two Ez values from QDR II SRAM 4. These six values or four of them plus two values buffered in the previous clock cycle (totally six values) can be fed to update engines magnetic update Hx and magnetic update Hy in the same clock cycle, thus one Hx value and one Hy value can be updated every clock cycle.

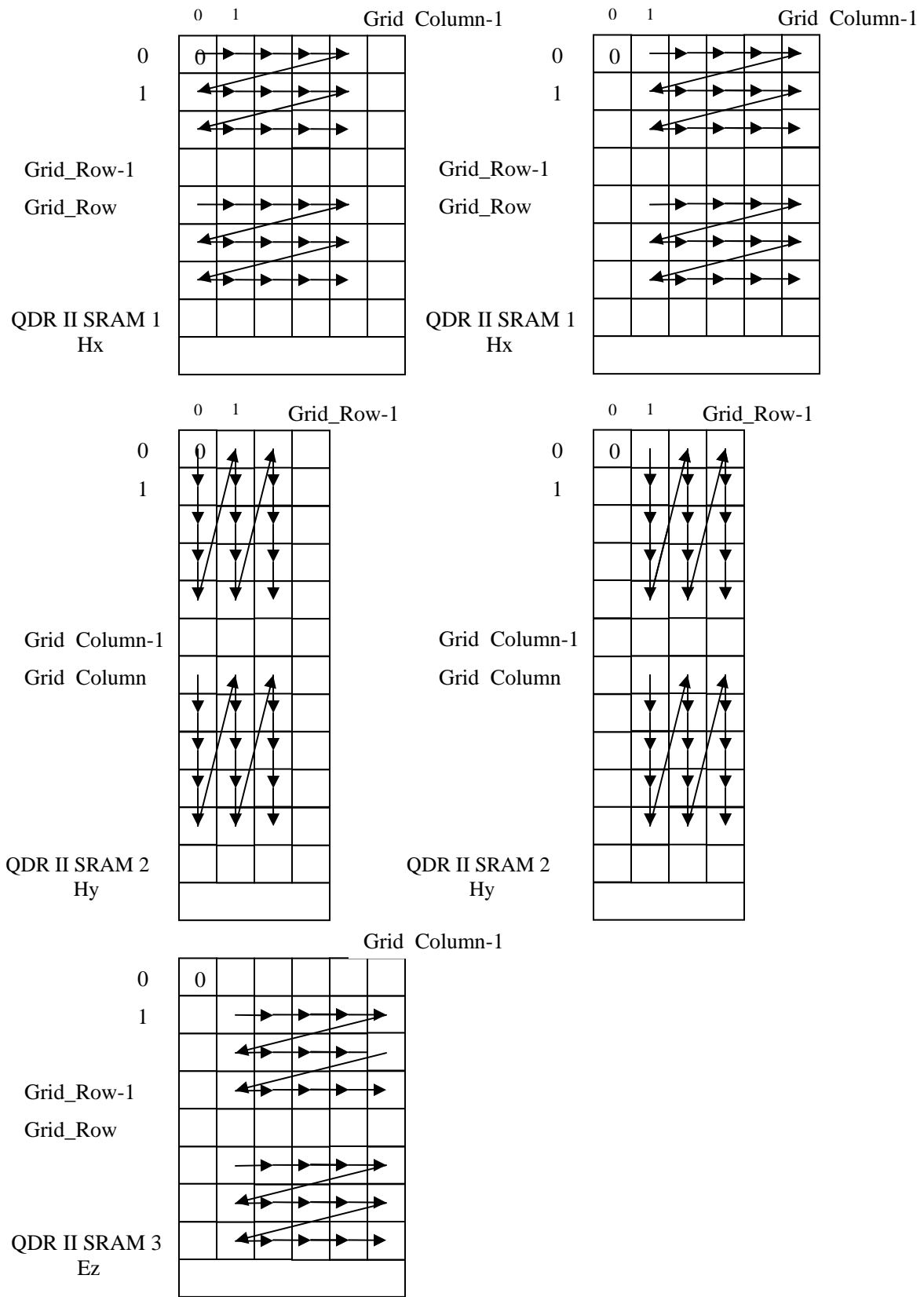


Figure 4-12 Read Addresses Generation Order for E_z values update

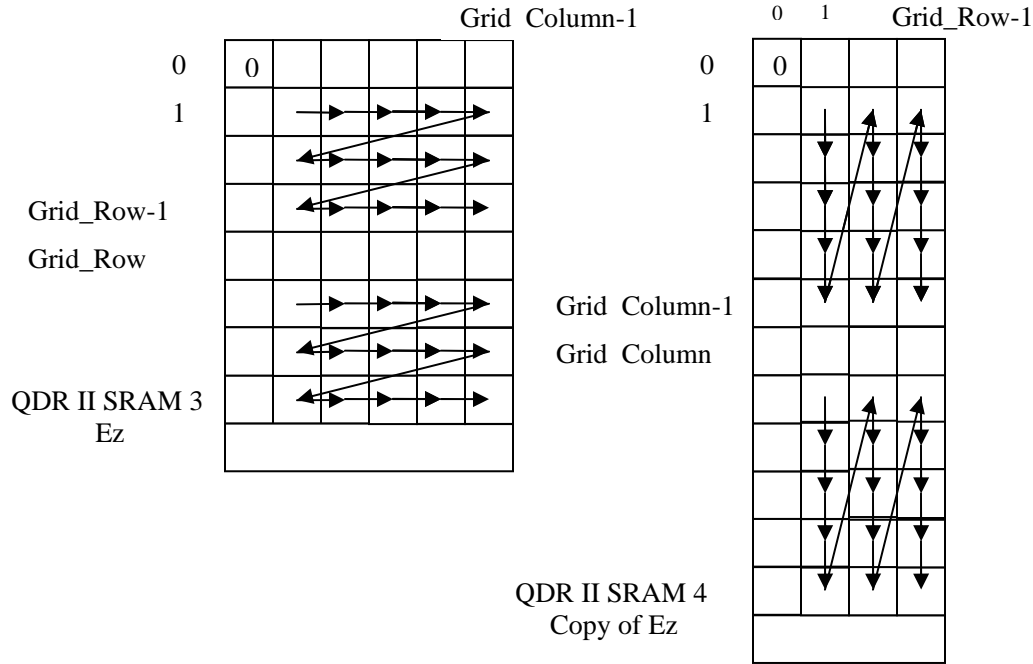


Figure 4-13 Write Addresses Generation Order for E_z values update

Component Qdr_fdttd can update one E_z value every two clock cycles. Every two clock cycles Qdr_fdttd reads two H_x values from QDR II SRAM 1, two H_y values from QDR II SRAM 2, one E_z value from QDR II SRAM 3. These five values are fed to update engine electric update, thus one E_z value can be updated every two clock cycles.

4.3.2 Simulation

The simulation is done in the simulation environment described in chapter 3 (see 3.2.2). The test bench instantiates the AAP FPGA, the fabric model and the QDR II SRAM model. Then it fetches commands such as read request and write request from a stimulus file fabric.in and feeds these commands to the fabric model. The fabric model then translates the requests into low-level bus transactions. For a delay command, the fabric model will insert a time delay between requests. If it is a print command, the fabric

model then will display messages on the simulation console. For a read or write request, the fabric model will send it to the AAP FPGA. The AAP FPGA processes the received request and responds appropriately. For a write request, the AAP FPGA processes the transaction but doesn't give response to the fabric model. For a read request, the AAP FPGA gives response to the fabric model and the fabric model will verify that the response matches the expected data provided in the request.

The operation sequence in the stimulus file fabric.in is the same as the operation sequence in the host application described in 4.3.5.

Structure of the test bench is depicted in figure 4-14.

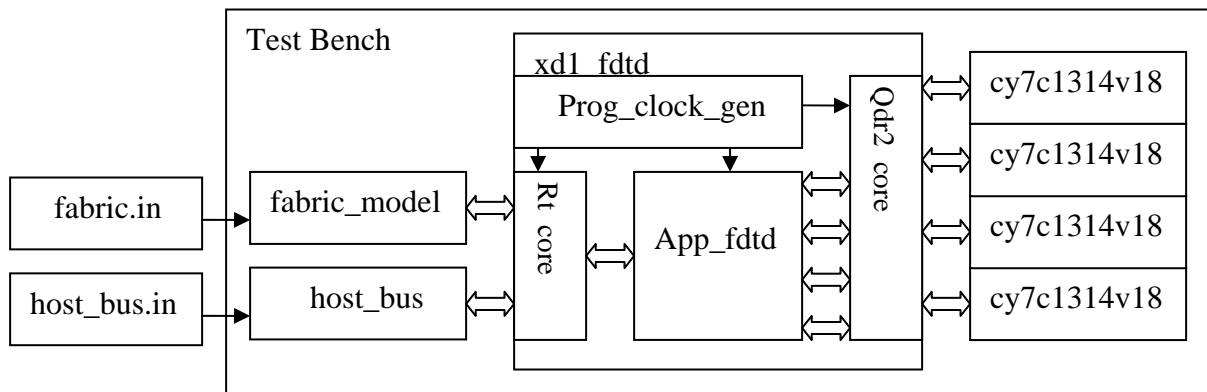


Figure 4-14 Structure of the Test Bench in the Cray XD1

4.3.3 Synthesis

Components Rt_core and Qdr2_core are incorporated into the FPGA design as “black box” components within the VHDL source code. They are provided as Xilinx NGC netlist files instead of VHDL files. The NGC files contain both the gate level design implementation for Rt_core and Qdr2_core plus additional timing and placement constraint information.

The FPGA design is synthesized with Xilinx XST. First all the other components in the FPGA design except Components Rt_core and Qdr2_core are synthesized and netlist files for these components are generated. Then these netlist files are combined with the NGC netlist files of Rt_core and Qdr2_core. The output is an integrated NGD file for the whole FPGA design.

In the process of synthesis, timing and placement constraint information in the user constraint file (UCF) is integrated into the NGD file.

4.3.4 P&R

The generated NGD file in 4.3.3 is placed and routed with P&R tool Xilinx ISE on the target FPGA Xilinx Virtex-II Pro XC2VP50 XCV. The output file is a downloadable FPGA binary file for the FPGA design.

4.3.5 Host Programming

The host application is programmed with 'C' language in the host programming environment described in chapter 3 (see 3.2.3). In order to interact with the AAP FPGA, it calls the FPGA API functions.

The operation sequence in the host application is as follows:

- 1) Open the AAP FPGA
- 2) Download FPGA binary file into AAP FPGA
- 3) Write parameters to register FPGA_reg
- 4) Write parameter Update_counter to register Update_counter
- 5) Write $H_{x_{t-1}}$ values to local memory QDR II SRAM 1

- 6) Write $H_{y_{t-1}}$ values to local memory QDR II SRAM 2
- 7) Write $E_{z_{t-1}}$ values to local memory QDR II SRAM 3
- 8) Write $E_{z_{t-1}}$ values to local memory QDR II SRAM 4
- 9) Toggle parameter Update_starter in register FPGA_reg to start the FDTD
algorithm calculation
- 10) Poll parameter Update_end in register FPGA_reg. If Update_end is set, the FDTD
algorithm calculation ends
- 11) Read H_{x_t} values from local memory QDR II SRAM 1
- 12) Read H_{y_t} values from local memory QDR II SRAM 2
- 13) Read E_{z_t} values from local memory QDR II SRAM 3
- 14) Close the AAP FPGA

Chapter 5

Theoretical Performance Analysis and Future Work

In this chapter, theoretical performance is analyzed and evaluated for the clock rates achieved in the implementations in both systems. Based on the analysis, suggestions for future work are given.

In each system, performance analysis is done for the whole system. It includes execution time for FDTD algorithm calculation and overhead due to configuration and data transfer. The time it takes to update one electromagnetic field value is derived and used as a performance metric.

Definitions for the symbols used in this chapter are:

N : size of the electromagnetic matrix processed by the FPGA. Assume there are

$Grid_Row$ rows and $Grid_Column$ columns in the matrix, then

$$N = Grid_Row \times Grid_Column$$

$2C$: specified number of iterations of FDTD algorithm calculation in the Cray XD1

system, only one iteration can be carried out on a given run on the Prototype

System

T_0 : time needed to transfer $3N$ electromagnetic field values between the host system

and the local memories

D_{pro} : average delay for local memory in the Prototype System

D_{cray} : delay for local memory in the Cray XD1 System

F : latency for internal FIFOs in the Prototype System

M : latency for magnetic update engines

E : latency for electric update engines

T_u : clock period for the update engines

T_{pro} : time needed to update one electromagnetic field value in the Prototype System

T_{cray} : time needed to update one electromagnetic field value in the Cray XD1 System

5.1 Performance Analysis in the Prototype System

In the Prototype system, the complete process to calculate the FDTD algorithm consists of five stages:

- 1) the host system writes electromagnetic field values to MEM0, MEM1, MEM2
- 2) FPGA reads source data from MEM0, MEM1, MEM2
- 3) the update engines execute one iteration of FDTD algorithm calculation
- 4) FPGA writes the result data into MEM3, MEM4
- 5) the host system reads electromagnetic field values back from MEM3, MEM4

Step 2), 3), 4) are overlapped in pipeline fashion. In step 3), if magnetic field values are updated, magnetic update Hx and magnetic update Hy execute serially, and one Hx result and one Hy result can be produced every four clock cycles. If electric field values are updated, one Ez result can be produced every two clock cycles.

To update $2N$ magnetic field values, the five stages should be gone through. The total time to complete the magnetic field values update is $T_0 + (D_{pro} + 2F + M + 4N) \times T_u$. To update N electric field values, the five stages should be gone through again. The total time to complete the electric field values update is $T_0 + (D_{pro} + 2F + E + 2N) \times T_u$. T_{pro} can be derived as:

$$T_{pro} = (2T_0 + (2D_{pro} + 4F + M + E + 6N) \times T_u) / 3N \quad (4a)$$

In equation (4a), T_0 is a function of N and clock rate of the FPGA, assume $T_0 = kNT_u$, then

$$T_{pro} = [(2D_{pro} + 4F + M + E)/3N + (2 + 2k/3)] \times T_u \quad (4b)$$

In equation (4b), D_{pro} is a system constant, F , M , E are design constants. These constants have insignificant influence on the performance.

The significant variables in equation (4b) are N and T_u . If N is very small, T_{pro} will be hundreds of times of T_u , this means very low performance. If N is very large, T_{pro} will approach $(2 + 2k/3) \times T_u$, the performance improves but is still very low.

In the current design, $D_{pro}=1$, $F=2$, $M=22$, $E=30$, the minimum value for k is 3. The following table shows T_{pro} as a function of N :

N	1	5	10	20	50	100	500	1000
T_{pro}	24.6 T_u	8.1 T_u	6.1 T_u	5.0 T_u	4.4 T_u	4.3 T_u	4.04 T_u	4.02 T_u

5.2 Performance Analysis in the Cray XD1 System

In the Cray XD1 system, the complete process to calculate the FDTD algorithm consists of five stages:

- 1) the host system writes electromagnetic field values to QDR II SRAM 1-4
- 2) FPGA reads source data from QDR II SRAM 1-4
- 3) the update engines execute specified number of iterations of FDTD algorithm calculation
- 4) FPGA writes the result data into QDR II SRAM 1-4
- 5) the host system reads electromagnetic field values back from QDR II SRAM 1-4

There are no internal FIFOs between the stages. If magnetic field values are updated, magnetic update Hx and magnetic update Hy execute in parallel, and one Hx result and one Hy result can be produced every clock cycle. If electric field values are updated, one Ez result can be produced every two clock cycles.

The specified number of iterations of FDTD algorithm calculation is $2C$, so $2N \times C$ magnetic field values will be updated and $N \times C$ electric field values will be updated. The total updated values are $3N \times C$. Because two magnetic field values (one Hx value and one Hy value) can be updated every clock cycle and one electric field value can be updated every two clock cycles, the time it takes for the update engines to update the $3N \times C$ values are $3N \times C \times T_u$. And the total time to go through the five stages is

$T_0 + (2D_{cray} + M + E + 3N) \times T_u \times C$, T_{cray} can be derived as:

$$T_{cray} = (T_0 + (2D_{cray} + M + E + 3N) \times T_u \times C) / (3N \times C) \quad (4c)$$

In equation (4c), T_0 is a function of N and clock rate of the FPGA, assume $T_0 = kNT_u$, then

$$T_{cray} = [(2D_{cray} + M + E) / 3N + k / 3C + 1] \times T_u \quad (4d)$$

In equation (4d), D_{cray} is a system constant, M , E are design constants. These constants have insignificant influence on the performance.

The significant variables in equation (4d) are N , C , T_u . If N and C is very small, T_{pro} will be hundreds of times of T_u , this means very low performance. If N and C is very large, T_{pro} will approach T_u , the performance improves greatly.

In the current design, $D_{cray}=1$, $M=22$, $E=30$, the minimum value for k is 3. The following table shows T_{cray} as a function of N and C :

$\begin{matrix} N \\ T_{cray} \\ C \end{matrix}$	1	5	10	20	50	100	500	1000
1	20 T_u	5.6 T_u	3.8 T_u	2.9 T_u	2.36 T_u	2.18 T_u	2.04 T_u	2.02 T_u
5	19.2 T_u	4.8 T_u	3.0 T_u	2.1 T_u	1.56 T_u	1.38 T_u	1.24 T_u	1.22 T_u
10	19.1 T_u	4.7 T_u	2.9 T_u	2.0 T_u	1.46 T_u	1.28 T_u	1.14 T_u	1.12 T_u
50	19.02 T_u	4.62 T_u	2.82 T_u	1.92 T_u	1.38 T_u	1.2 T_u	1.06 T_u	1.04 T_u

The performance goal in the Cray XD1 system is to reduce T_{cray} . According to analysis of equations (4d), the following methods can be adopted for performance improving:

- increase the clock rate of update engines, this will reduce T_u
- increase the size of the electromagnetic matrix processed by the FPGA, this will increase N
- increase the specified number of iterations of FDTD algorithm calculation, this will increase C
- decrease the latency of update engines, this will reduce M and E , but have only small impact on overall performance for reasonable values of N .

5.3 Performance Comparison

The Cray XD1 system greatly outperforms the Prototype system based on analysis of equations (4b) and (4d). Equations (4b) and (4d) are repeated here:

$$T_{pro} = [(2D_{pro} + 4F + M + E)/3N + (2 + 2k/3)] \times T_u \quad (4b)$$

$$T_{cray} = [(2D_{cray} + M + E)/3N + k/3C + 1] \times T_u \quad (4d)$$

If $C=1$, then $T_{pro} \approx 2T_{cray}$. This means for one iteration of FDTD algorithm calculation, the Cray XD1 system outperforms the Prototype system by about 2 times.

When C increases, for a reasonable matrix, $3N \gg (2D_{pro} + 4F + M + E)$, $3N \gg (2D_{cray} + M + E)$, so $T_{pro} \approx (2 + 2k/3) \times T_u$, $T_{cray} \approx T_u$. Normally, the data transfer speed between the host system and the FPGA is very slow, so $k \geq 3$, $T_{pro} \geq 4T_u$. This means for more than one iteration of FDTD algorithm calculation, the Cray XD1 system outperforms the Prototype system by more than 4 times.

For a very special case which can only appear in experiments, N is very small compared with $(2D_{pro} + 4F + M + E)$ and $(2D_{cray} + M + E)$, so both T_{pro} and T_{cray} will be hundreds of times of T_u . This means unacceptable low performance for both systems. To avoid this case, the size of the actual electromagnetic matrix distributed to one FPGA should be large.

5.4 Future Work in the Cray XD1 System

Based on the analysis in 5.2, several suggestions are proposed for the future work in the Cray XD1 system.

First new floating point library may be chosen to replace the current floating point library. Update engines implemented using floating point library is located in the critical

paths of the design. If the clock rate for the update engines improves, the clock rate for the whole FPGA can improve, so does the performance. The current library is provided by Pavle Belanovic [15]. Components in this library are well pipelined but not optimized for speed. So in the future, floating point library well optimized for speed can be tried.

One choice is to use the Nallatech Floating Point Cores. The Nallatech Floating Point Cores employ the FPGA-optimal Nallatech Floating Point format for internal use, with conversion blocks provided for IEEE-754 compatibility. The pre-placed, fully pipelined core architectures offer performance of up to 180MHz. In the current design, the clock rate for the update engines is 75MHz. If the Nallatech floating point cores are used to implement the update engines, the clock rate can be improved to 180MHz.

More parallelism may be explored in the future. The current design occupies 45% of the slices, 11% of the 18×18 bit multiplier blocks, 7% of the 18 Kb Block RAMs. So it is feasible to add one more set of update engines in the FPGA implementation and the performance will double. The memory architecture should change accordingly if one more set of update engines is added in the FPGA.

In the current design, component Qdr_fdttd uses the Fabric Request Interface to send data to the host system. When Qdr_fdttd completes FDTD algorithm calculation, it needs to wait for polling from the host system. If the polling interval is too large, Qdr_fdttd may wait too long. This degrades the whole performance. In the future, the User Request Interface can be used for Qdr_fdttd to send results to the host system as soon as Qdr_fdttd completes FDTD algorithm calculation.

Currently, for use in a parallel simulation boundary data must be transferred between the FPGA and the host system, then sent to other nodes. This is one performance

bottleneck. In the future, the FPGA may exchange data directly with the neighboring FPGAs. Cray has not yet provided cores to support this although the underlying architecture has the capability.

Bibliography:

- [1] C.E. Cox, and W.E. Blanz, "GANGLION- A Fast hardware Implementation of a Connectionist Classifier", IEEE Proc. Custom Integrated Circuits Conference (CICC'91), 6.5.1, 1991
- [2] M. Wazlowski et al. "PRISM-II Compiler and Architecture", Proc. FCCM'93, pp. 9-16, 1993
- [3] D. Buell et al., "Splash 2: FPGAs in a Custom Computing Machine," IEEE Computer Press, 1996 (ISBN0-8186-7413-X)
- [4] T.Tsutsui, and T. Miyazaki, "YARDS:FPGA/MPU Hybrid Architecture for Telecommunication Data Processing", Proc. ACM/SIGDA FPGA'97, pp. 93-99, Feb 1997
- [5] Miyazaki, T. "Reconfigurable systems: a survey", Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific, 10-13 Feb. 1998 Pages: 447-452
- [6] Smith, M.C. ; Drager, S.L. ; Pochet, L. ; Peterson, G.D. ; "High performance reconfigurable computing systems", Circuits and Systems, 2001. MWSCAS 2001. Proceedings of the 44th IEEE 2001 Midwest Symposium on Volume: 1, 14-17 Aug. 2001 Pages: 462-465 vol.1
- [7] Fidanci, O.D. ; Poznanovic, D. ; Gaj, K. ; El-Ghazawi, T. ; Alexandridis, N. "Performance and overhead in a hybrid reconfigurable computer", Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 22-26 April 2003
- [8] El-Araby, E. ; Taher, M. ; Gaj, K. ; El-Ghazawi, T. ; Caliga, D. ; Alexandridis, N.

- “System-level parallelism and throughput optimization in designing reconfigurable computing applications”, Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, April 26-30, 2004 Pages: 136-143
- [9] IEEE Standards Board. “IEEE Standard for Binary Floating-Point Arithmetic”. Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronic Engineers, new York, 1985
- [10] B. Fagin and C. Renard. “Field Programmable Gate Arrays and Floating Point Arithmetic”. IEEE Transactions on VLSI Systems, 2(3), September 1994
- [11] N. Shirazi, A. Walters, and P. Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995, pp. 155-162
- [12] L. Louca, T. A. Cook, and W. H. Johnson. “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs”. In K. L. Pocek and J. Arnold, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pages 107–116, April 1996
- [13] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs”. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1998
- [14] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp. 897-900
- [15] Pavle Belanovic, “Library of Parameterized Hardware Modules for Floating-Point

Arithmetic with An Example Application”, M.S. Thesis, Dept of Electrical and Computer Engineering, Northeastern University, June 2002

- [16] P. Belanović and M. Leeser. “A Library of Parameterized Floating Point Modules and Their Use”. In Proceedings, International Conference on Field Programmable Logic and Applications, Montpellier, France, Aug. 2002
- [17] Jian Liang; Tessier, R.; Mencer, O. “Floating point unit generation and evaluation for FPGAs”. Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on 9-11 April 2003 Pages:185-194
- [18] R. N. Schneider, L. E. Turner, M. M. Okoniewski, “Application of FPGA Technology to Accelerate the Finite-Difference Time-Domain (FDTD) Method”, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays. pp.97-105
- [19] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, D. W. Prather, and M. S. Mirotznik, “Hardware Implementation of a Three-Dimensional Finite-Difference Time-Domain Algorithm”, IEEE Antennas and Wireless Propagation Letters, VOL.2, 2003
- [20] W. Chen, P. Kosmas, M. Leeser, C. Rappaport. “An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm”, International Symposium on Field Programmable Gate Arrays, Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, 2004, pp. 213-222
- [21] Sachin Gandhi, “An FPGA Implementation of FDTD Codes For Reconfigurable High Performance Computing”, Master Thesis, University of Cincinnati, Nov 2004
- [22] J. Stratton, “Electromagnetic Theory”. New York: McGraw-Hill,1941, p. 23

- [23] K. Yee, “Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media”, IEEE Trans. Antennas and Propagation, 16 (1966), pp. 302-307
- [24] “FIREBIRD™ Hardware Reference Manual for FIREBIRD™ /PCI, c/PCI, /PMC Dual WSDP™ , and PMC Dual G-Link”, 12727-0000 Revision 3.0
- [25] “WILDSTAR™ Software Reference Manual for WILDSTAR™ , STARFIRE™ , and FIREBIRD™”, 12723-0000 Revision 2.0
- [26] “Cray XD1™ System Overview”, S-2429-12, Release 1.2
- [27] “Cray XD1™ FPGA Development”, S-6400-12, Release 1.2
- [28] “Cray XD1™ Programming”, S-2433-12, Release 1.2
- [29] “Design of Cray XD1™ RapidArray Transport Core”, S-6411-12, Release 1.2
- [30] “Design of Cray XD1™ QDR II SRAM Core”, S-6412-12, Release 1.2
- [31] “Cray XD1 MINCE FPGA Design”, June 06, 2004, Issue 0.3, PNR-DD-0015