PROTOTYPE DEVELOPMENT AND VERIFICATION OF AN IP LOOKUP ENGINE ON FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AKIN ÖZKANER

IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2012

APPROVAL OF THE THESİS:

PROTOTYPE DEVELOPMENT AND VERIFICATION FOR AN IP LOOKUP ENGINE ON FPGAS PERFORMANCE STUDY

submitted by **AKIN ÖZKANER** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____

Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet Erkmen _____

Head of Department, **Electrical and Electronics Engineering**

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı _____

Supervisor, **Electrical and Electronics Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Semih Bilgen _____

Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı _____

Electrical and Electronics Engineering Dept., METU

Prof. Dr. Gözde B. Akar _____

Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Ş. Ece Schmidt _____

Electrical and Electronics Engineering Dept., METU

Dr. Oğuzhan Erdem _____

Electrical and Electronics Engineering Dept., Atatürk Univ.

Date: 08.02.2012

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Akın Özkaner

Signature :

ABSTRACT

PROTOTYPE DEVELOPMENT AND VERIFICATION FOR AN IP LOOKUP ENGINE ON FPGAS PERFORMANCE STUDY

Özkaner, Akın

M. S., Department of Electrical and Electronics Engineering

Supervisor: Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı

February 2012, 117 pages

The increasing use of the internet demands more powerful routers with higher speed, less power consumption and less physical space occupation. IP lookup operation is one of the major concerns in today's routers for providing such attributes. To accomplish IP lookup on routers, hardware or software based solutions can be used. In this thesis, an SRAM based pipelined architecture proposed earlier for ASIC implementation is re-designed and implemented on an FPGA in the form of a BRAM based pipelined 8x8 torus architecture using Xilinx ISE and simulated and verified using Modelsim Simulator. Some necessary modifications and improvements for FPGA implementation are carried out. The results of our experiments, which are performed for a real router lookup table and a real time traffic load with various optimizations, are also presented. Our study and design effort demonstrates the feasibility of the FPGA implementation of the proposed technique, of course with a considerable performance penalty.

Keywords : IP lookup, FPGA, routers.

ÖZ

FPGA ÜZERİNDE IP TARAMA MOTORU PROTOTİP GELİŞTİRMESİ VE DOĞRULAMASI

Özkaner, Akın

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi : Doç. Dr. Cüneyt F. Bazlamaçcı

Şubat 2012, 117 sayfa

Artan internet kullanımı; yüksek hızda çalışan, güç tüketimi düşük olan ve fiziksel olarak az yer kaplayan daha etkili ağ yönlendiricilerinin kullanımını gerektirmektedir. Bu özelliklere sahip ağ yönlendiricileri için en önemli unsurlardan birisi de IP arama işlemidir. Ağ yönlendiriciler üzerinde IP arama işlevini gerçekleştirmek amacıyla yazılım veya donanım temelli çözümler kullanılabilmektedir. Bu tez çalışması, boru hattı davranışlı SRAM tabanlı ASIC model için daha önce önerilmiş bir çalışmanın yeniden tasarlanarak FPGA donanım yapısında gerçeklenmesini içermektedir. Tez kapsamında, boru hattı davranışlı BRAM tabanlı 8x8 torus mimarinin Xilinx ISE ile tasarımı ve Modelsim Simulator ile benzetimi gerçekleştirilmiştir. FPGA tasarımına yönelik bazı iyileştirmeler ve gerekli değişiklikler yapılmıştır. Gerçek yönlendirici arama tabloları ve gerçek zamanlı ağ trafiği ile çeşitli en iyileştirmeler ve denemeler de gerçekleştirilmiş ve sonuçları sunulmuştur. Çalışmamız ve tasarım çabamız daha önce önerilmiş tekniğin, elbette belirgin bir başarım kaybı karşılığında, FPGA gerçeklemesinin de mümkün olduğunu göstermektedir.

Anahtar Kelimeler: IP arama, IP tarama, ağ yönlendiricisi, FPGA üzerinde IP tarama

To My Family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı for his guidance, advice, criticism, encouragement, endless patience and insight throughout the completion of the thesis.

I wish to thank my company ASELSAN A.Ş for giving me the opportunity of continuing my thesis study and supporting me in my efforts to get the Master of Science degree.

I would like to express my appreciation to all my friends and colleagues for their contributions to my thesis with their continuous guidance, advice, encouragement and for expanding my horizons.

Finally, my family. No word can suffice to express how I am grateful to my parents but at least I can mention my sincere gratitude to them for their unwavering support, continual confidence and endless and gratis love. I also wish to thank my wife for her support on my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BRAM	Block RAM
CAM	Content Addressable Memory
CAMP	Circular, Adaptive and Monotonic Pipeline
CIDR	Classless Inter Domain Routing
CCU	Congestion Controller Unit
CR	Contention Resolver
IP	Internet Protocol
ISP	Internet Service Provider
LPM	Longest Prefix Matching
PE	Processing Element
POLP	Parallel Optimized Linear Pipeline
RDL	RAM Data Loader
SAFIL	Systolic Array Architecture for Fast IP Lookup
SU	Selector Unit
TCAM	Ternary Content Addressable Memory

CHAPTER 1

INTRODUCTION

In the developing environment of high performance IP networks, it is expected that local and wide area backbones, enterprise networks, and ISPs will use multigigabit and even terabit networking technologies, where IP routers will be used not only to interconnect backbone segments but also to act as points of attachments to high performance wide area links.

1.1 BACKGROUND

The primary role of IP routers is to forward packets to their final destination address. For this purpose, a router must decide for each incoming packet where to send it next. In other words, the forwarding decision consists of finding the address of the next-hop router and the output port through which the packet should be sent. This information is stored in a lookup table that the router computes based on the information gathered by routing protocols. To consult the lookup table, the router uses the incoming packet's destination address as a key and this process is called *address lookup*. Once the forwarding information is retrieved, the router can transfer the packet from the incoming link to the appropriate outgoing link, in a process called switching.

The rapid growth of the Internet has stressed its routing system. While the link rates have kept pace with the increasing traffic, it has been difficult for the packet processing capacity of routers to keep up with the increased data rates of the link.

The Classful Addressing Scheme

When Internet addressing was initially designed, a simple address allocation scheme was defined, which is known today as the classful addressing scheme.

In IP version 4, IP addresses are 32 bit long and, when broken up into 4 groups of 8 bits, are normally represented as four decimal numbers separated by dots. The IP address scheme initially used a simple two-level hierarchy, with networks at the top level and hosts at the bottom level. This hierarchy is reflected in the fact that an IP address consists of two parts, a network part and a host part. The network part identifies the network to which a host is attached and thus all hosts attached to the same network agree in the network part of their IP addresses.

Since the network part corresponds to the first bits of the IP address, it is called the address prefix. We will write prefixes as bit strings of up to 32 bits in IPv4 followed by a "*". For example, prefix 1000001001010110* represents all 2^{16} addresses that begin with the bit pattern 1000001001010110. Alternatively, prefixes can be indicated using the dotted-decimal notation, so the same prefix can be written as 130.86/16, where the number after the slash indicates the length of the prefix.

With a two-level hierarchy, IP routers forward packets based only on the network part, until packets reach the destination network. As a result, a forwarding table only needs to store a single entry to forward packets to all the hosts attached to the same network. This technique is called address aggregation and allows using prefixes to represent a group of addresses. Each entry in a forwarding table contains a prefix (Figure 1-1). So, finding the forwarding information requires searching for the prefix in the forwarding table that matches the corresponding bits of the destination address.

Destination Address Prefix	Next-hop	Output interface
24.40.32/20	192.41.177.148	2
130.86/16	192.41.177.181	6
208.12.16/20	192.41.177.241	4
208.12.21/24	192.41.177.196	1
167.24.103/24	192.41.177.3	4

Figure 1-1 Example Forwarding Table

1.1.1 Network Topology

Basically, three different sizes of networks were defined in the classful addressing scheme, identified by a class name: class A, B, and C. Size of networks was determined by the number of bits used to represent the network part and the host part. Thus networks of class A, B or C consisted in an 8, 16 or 24-bit network part and a corresponding 24, 16 or 8-bit host part (Figure 1-2).

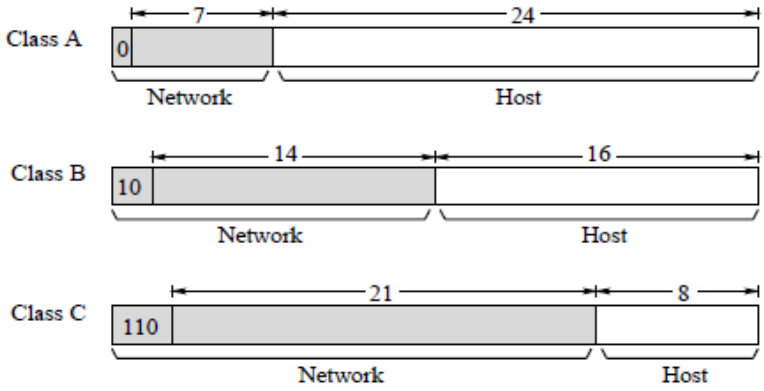

Figure 1-2 Classful Addressing Scheme

With this scheme there were very few class A networks and their addressing space represented 50% of the total IPv4 address space (2^{31} addresses out of a total of 2^{32}). There were 16,384 (2^{14}) class B networks with a maximum of 65,536 hosts per network and 2,097,152 (2^{21}) class C networks with up to 256 hosts. This allocation scheme worked well in the early days of the Internet. However, the continuous growth of the number of hosts and networks has made apparent two problems with this classful addressing architecture. First, with only three different network sizes to

choose, the address space was not used efficiently and the IP address space was getting exhausted very rapidly, even though only a small fraction of the addresses allocated were actually in use.

The CIDR Addressing Scheme

To allow for a more efficient use of the IP address space and to slow down the growth of the backbone forwarding tables, a new scheme called Classless Inter-domain Routing or CIDR was introduced.

In the classful addressing scheme, only 3 different prefix lengths are allowed: 8, 16 and 24 corresponding to the classes A, B and C, respectively. CIDR makes more efficient use of the IP address space by allowing a finer granularity in the prefix lengths. With CIDR, prefixes can be of arbitrary length rather than constraining them to be 8, 16 or 24 bits long.

CIDR allows address aggregation at several levels. Consider the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24. Suppose that in a router all these network addresses are reachable through the same service provider. From the binary representation we can see that the leftmost 20 bits of all the addresses in this range are the same. Thus, we can aggregate these 16 networks into one "super network" represented by the 20-bit prefix, which in decimal notation gives 208.12.16/20 (Figure 1-3). Note that indicating the prefix length is necessary in decimal notation, because the same value may be associated to prefixes of different lengths, for instance 208.12.16/20 (11010000 00001100 0001*) is different from 208.12.16/22 (11010000 00001100 000100*).

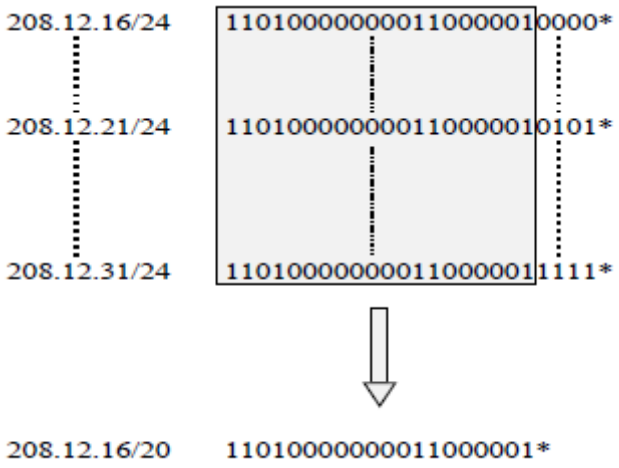

Figure 1-3 Address Aggregation in CIDR Acheme

1.1.2 Router Architecture

The popularity of the Internet has caused the traffic on the Internet to grow drastically every year for the last several years. It has also spurred the emergence of many ISPs. To sustain growth, ISPs need to provide new differentiated services, e.g., tiered service, support for multimedia applications, etc. The routers in the ISPs' networks play a critical role in providing these services. IP traffic in private enterprise networks has also been growing rapidly for some time. These networks face significant bandwidth challenges as new application types, especially desktop applications uniting voice, video, and data traffic need to be delivered on the network infrastructure. This growth in IP traffic is beginning to stress the traditional processor-based design of current-day routers and as a result has created new challenges for router design.

Routers have traditionally been implemented purely in software. Because of the software implementation, the performance of a router was limited by the performance of the processor executing the protocol code. To achieve wire-speed routing, high-performance processors together with large memories were required. This translated into higher cost. Thus, while software-based wire-speed routing was possible at low-speeds, for example, with 10 megabits per second (Mbps) ports, or with a relatively smaller number of 100 Mbps ports, the processing costs and

architectural implications make it difficult to achieve wire-speed routing at higher speeds using software-based processing.

Fortunately, many changes in technology (both networking and silicon) have changed the landscape for implementing high-speed routers. Silicon capability has improved to the point where highly complex systems can be built on a single integrated circuit. The use of 0.35 μm and smaller silicon geometries enables application specific integrated circuit implementations of millions gate-equivalents. Embedded memory and microprocessors are available in addition to high-density logic. This makes it possible to build single-chip, low-cost routing solutions that incorporate both hardware and software as needed for best overall performance.

1.1.3 IP Lookup

Due to the rapid growth of traffic in the Internet, backbone links of several Gigabit/sec are commonly deployed. To handle Gigabit/sec traffic rates, the backbone routers must be able to forward millions of packets per second on each of their ports. Fast IP address lookup in the routers, which uses the packets destination address to determine for each packet the next hop, is therefore crucial to achieve the packet forwarding rates required.

1.2 MOTIVATION

In hardware based IP Lookup solutions for network routers, there are two main categories. These are namely ternary content addressable memory (TCAM) based and random access memory (RAM) based solutions. RAM based solutions include dynamic or static random access memories (DRAM or SRAM) or Block RAM (BRAM) in FPGA or ASIC. Each prefix can be stored in a TCAM with not only using 0's or 1's but also using don't care values. A seach key (i.e. IP address) is compared to all entries cycle and only one matched result, which is the longest matching prefix, appears at the output in one clock cycle. Therefore, TCAM based solutions have been popular for implementing lookup functions in core routers [1]. However, they have high cost and high power consumption as major drawbacks in

addition to their unsuitability in adopting to new addressing and routing protocols [2, 3].

On the other hand, RAM based solutions offer higher memory access speeds, lower power consumption and higher density. To implement RAM based IP lookup architectures, generally tree type data structures for finding LPM are used and the trees are traversed appropriately during a search. However, multiple memory accesses may be needed to search an IP addresses in such structures. For improving the throughput, various pipelined architectures have been proposed [4, 5] the main idea being the storage of a lookup table (represented for example as a binary tree) of a router on seperate and multiple memory elements. When an IP search is in progress in a pipeline, another incoming search key can be admitted into the system. Although the throughput is improved in pipelined solutions, straightforward mapping of the tree on the pipeline stages makes an unbalanced memory distribution inevitable. One of the possible solutions to unbalanced memory problem was proposed earlier using two dimensional, parallel, intersecting, circular and variable length pipelines [6,26]. Our implementation within the scope of this thesis provides minor modifications on the work of [6] to adapt it to be implemented in an FPGA rather than an ASIC.

In [6], nothing was mentioned abour initializing RAM contents for each stage of the pipeline and hence we also proposed loading and updating of RAM contents in each processing element described in Section 4.1.

1.3 CONTRIBUTIONS

- We re-designed and adopted an existing SRAM based pipelined architecture, named SAFIL [6] for FPGA implementation.
    - We added load and update attributes to SAFIL.
    - We utilized FIFOs in CRs in SAFIL to prevent possible head of line blocking.

- We augmented SAFIL by designing a data flow manager (DFM) module in each PE to manage different type of incoming SAFIL frames.

- We implemented and simulated the modified SAFIL structure in the form of a BRAM based pipelined 8x8 torus architecture on an FPGA using Xilinx ISE and Modelsim Simulator.

1.4 OUTLINE

The rest of the thesis is organized as follows. Chapter 2 covers the background and related work for IP lookup approaches. Our array design for trie-based IP lookup and update is discussed in Chapter 3. Chapter 4 introduces the proposed IP lokup architecture and its implementation on FPGA. Finally, Chapter 5 summarizes and concludes our work.

CHAPTER 2

IP LOOKUP APPROACHES

In modern IP routers, Internet Protocol (IP) lookup forms a bottleneck in packet forwarding because the lookup speed cannot catch up with the increase in link bandwidth. To deal with this problem, various software and hardware based solutions have been proposed for over 20 years. In this chapter, a brief overview of prior work on IP lookup solutions will be presented.

2.1 SOFTWARE BASED SOLUTIONS

In IP lookup, the simplest and most popular data structure is *binary trie*. Each node in trie contains two pointers, the left-child pointer and the right-child pointer. Moreover if a trie node contains a valid prefix (corresponding to a routing table entry), then a next hop information (port number) associated with that prefix is also stored in a trie node. Figure 2-1 illustrates a sample prefix table and its corresponding binary trie.

In the rest of this text, the following terms are used:

- *Prefix node* is any trie node that corresponds to a valid prefix (marked as black)
- *Leaf prefix node* is a leaf node which is a valid prefix node (black leaf)

In a trie data structure, a node does not hold any prefix explicitly but the path from the root to another node corresponds to a prefix implicitly.

Prefix	Next Hop
0*	P1
11*	P2
010*	P3
100*	P4
0000*	P5
0001*	P6
0110*	P7
1010*	P8
1011*	P9

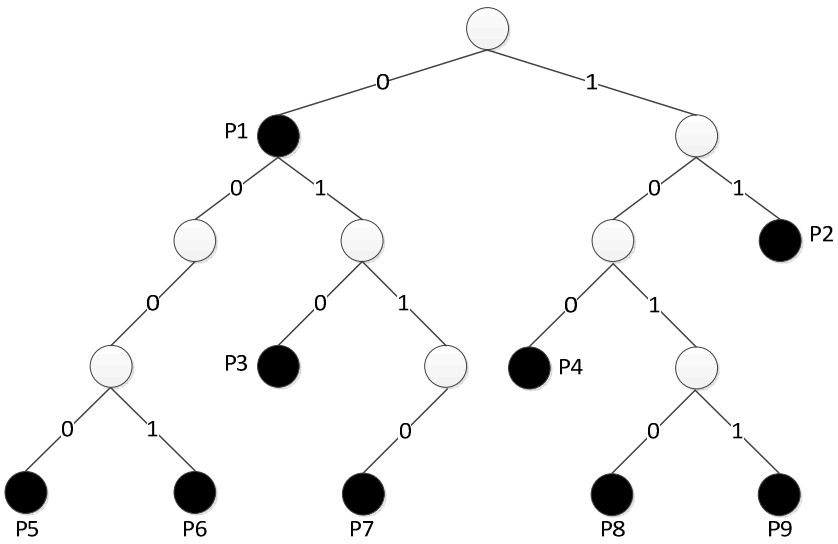

Figure 2-1 A Prefix Table and Corresponding Binary Trie

Any search operation begins with the root node. According to the bits of the IP address, operation continues traversing the trie from parent to child nodes. If following bit of IP address is "0", search continues towards left child node, otherwise right child node. While traversing, port result is updated if a valid node is encountered. Search operation terminates if a null child node or a leaf node is reached. The last matched prefix is selected as the longest matched prefix. For instance, a search key starting with 1010 will match the leaf prefix node whose next hop is P8 according to longest prefix mathing (LPM) rule in Figure 2-1. Update operations such as prefix insertion, deletion and route changes are easy to implement in a binary trie structure. On the other hand, search operation in a binary trie needs 32 memory accesses for IPv4 and 128 memory accesses for IPv6 in the worst case and hence lookup time gets longer.

When all the prefix nodes are pushed to leaves, then a binary trie is called a leaf-pushed binary trie [7]. In a leaf-pushed binary trie, a non-leaf node contains only pointers to its children and the leaf node contains only a next hop information associated with the corresponding prefix. Figure 2-2 shows the leaf-pushed version of the binary trie in Figure 2-1.

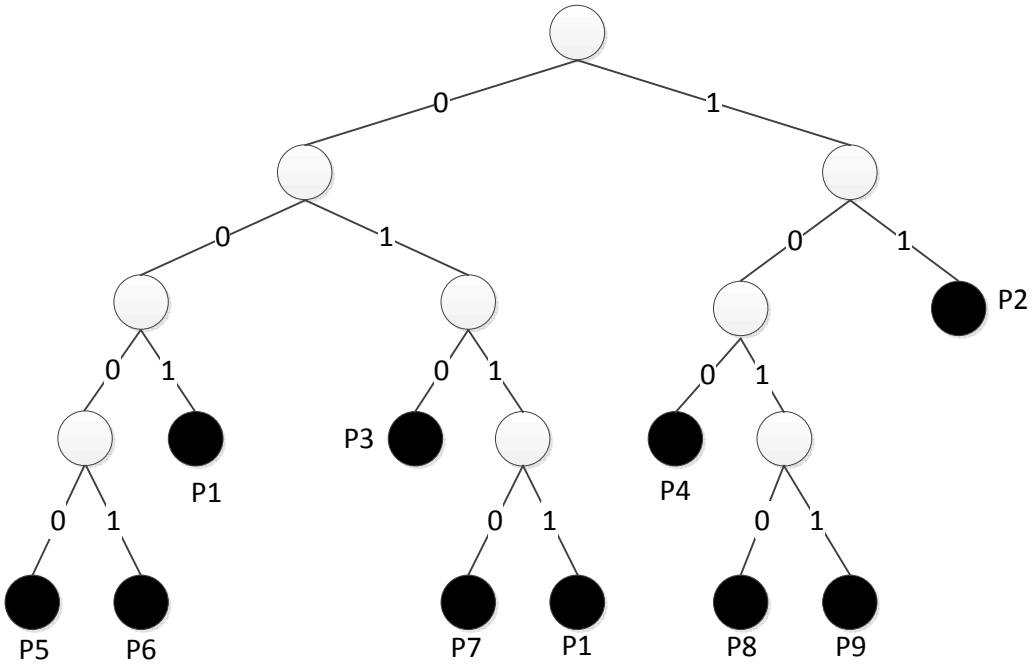

Figure 2-2 The Leaf Pushed Version of the Binary Trie in Figure 2-1

Software based IP lookup solutions are generally based on algorithms that use binary trie. These software approaches are scalable but slow to work in line speeds [8]. To accelarate software based lookup, various solutions were proposed. The search time and the memory performance of a binary trie can be increased with *path compression technique*, which was first proposed by [9]. With this method, any node of a trie can be omitted if it has only one child node to make the path from root node to leaf node shorten. In order to keep the record of removed internal nodes, each node must store a skip value and bit string. The skip value stores the number of bits to be skipped on the path. The bit string stores missing bits from the last skip operation.

2.2 HARDWARE BASED SOLUTIONS

Altgough software based solutions are simple and scalable, they are slow to work in line speeds [8]. To improve performance and especially the throughput, various hardware based solutions were proposed using either SRAM or TCAM based memory technologies. And recently, ASIC or FPGA based solutions have also appeared in the literature [10, 11, 12, 13, 14].

2.2.1 SRAM Based Solutions

Single SRAM based IP lookup solutions are in need of multiple memory accesses during the tree traversal for finding the matched port result. To increase the throughput, various SRAM based pipelined solutions have been proposed. A binary trie can be implemented in an SRAM based pipelined architecture using multiple static random access memory elements. Each stage in binary trie is represented by an SRAM block. Therefore, the number of stages should be equal to the SRAM blocks utilized. The number of memory accesses is then determined by the average depth of the trie that stores a part or all of the routing table. Each search operation can access a seperate memory block only once during a search if each stage of the binary trie is utilized seperately. During a search that checks whether an IP address matches a prefix or not, a new incoming search request must wait for the on-going lookup operation to finish up.

IP lookup in binary tries need multiple memory accesses in order to find LPM node. In a pipelined architectures, the trie is mapped onto the stages of the pipelines. The trie traversal is then performed on these separate and multiple memory elements (SRAMs) through the pipeline. Enough memory stages exist and no stage is accessed more than once during a search in a conventional one dimensional pipeline architecture. Although throughput is improved using a pipeline, an ordinary mapping of the binary trie onto the pipeline stages results in unbalanced memory utilization. Unbalanced trie node distribution over pipeline stages decreases the overall performance of the architecture. Various different solutions have been proposed to address the memory balancing problem [4, 5, 15, 16].

In [4], a ring pipeline architecture, which allows search to start from any pipeline stage, is proposed. This approach is based on dividing the binary trie into subtries and choosing each subtrie starting point to a different pipeline stage to create a balanced pipeline. In this approach, there are two different data path. First one is for finding the starting pipeline stage and the second one is for lookup operation. The matched port result propagates to the final pipeline stage to appear at the output. The throughput of the described Baboescu et al. architecture is 0,5 lookups per clock cycle.

In [5], the previous method is improved with an approach called Circular, Adaptive and Monotonic Pipeline (CAMP). Apart from previous approach, at any pipeline stage, there are two different input and one output stage. In this architecture, maximum 0,8 lookups per clock cycle are possible.

The throughput of pipelined architectures can be improved by using multiple pipelines. Jiang et al. [12] proposed the first parallel multipipeline architecture Parallel Optimized Linear Pipeline (POLP) in which each pipeline can operate concurrently to increase the speed up rate. POLP is improved further in later studies. For example a bidirectional linear pipeline is introduced in [17]. To improve POLP power effciency, a hybrid SRAM/TCAM selector unit is also proposed in [18] and [19], the aim being shortening pipeline lengths by introducing hybrid partitioning schemes.

2.2.2 T-CAM Based Solutions

Binary CAM is the simplest type of CAM which uses data search words consisting entirely of 1s and 0s. Ternary CAM (TCAM) allows a third state of "Don't Care" bits in the stored dataword, thus adding flexibility to the search. For example, a ternary CAM might have a stored word of "10XX0" which will match any of the four search words "10000", "10010", "10100", or "10110". The added search flexibility comes at an additional cost over binary CAM as the internal memory cell must now encode three possible states instead of the two of binary CAM. This additional state is typically implemented by adding a mask bit ("care" or "don't care" bit) to every memory cell. TCAM is more powerful because don’t cares may act as wildcards during a search and hence LPM can be solved naturally in one cycle [20].

As shown in Figure 2-3, in TCAM architectures, prefixes are stored in sorted order based on prefix lengths. When a search key (i.e. an IP address) is admitted into conventional TCAM, incoming bits are distributed to all the entries. The matched entries activate outputs that are fed into a priority encoder. If more than one outputs

are activated, the priority encoder decides which entry is LPM and outputs the longest matching one.

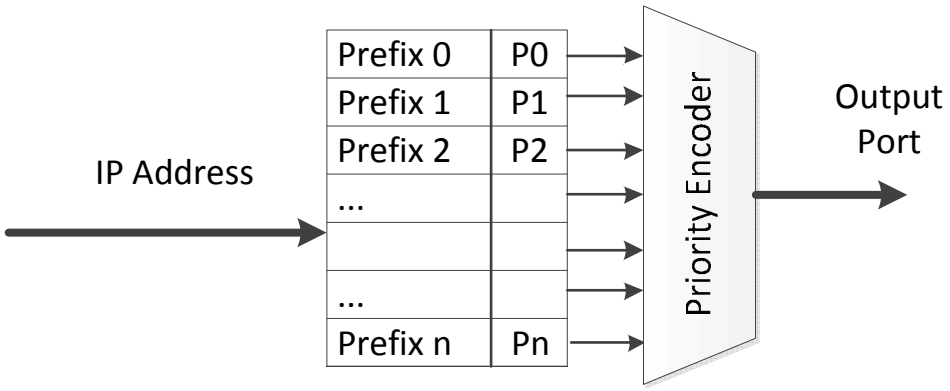

Figure 2-3 TCAM

Although TCAM-based solutions are straightforward and famous, they are expensive, power consuming, and offer little scalability and adaptability to new addressing and routing protocols [2, 3, 21, 22]. While a search is in progress, in which every memory block of the entries are used in active state, TCAM modules consumes high power. Moreover, updating a memory cell may require multiple entry moves, which means that long updating progress may be needed. Additionally, low scalability may arise in the case of changing the order in the priority encoder when updating the contents of the memory cells.

CHAPTER 3

ARRAY DESIGN FOR TRIE BASED IP LOOKUP AND UPDATE

3.1 INTRODUCTION

SRAM based IP lookup solutions are in need of multiple memory accesses to traverse the tree to perform a single search request. Since only one memory access is allowed during a lookup process for an IP address, a new incoming lookup request should wait until previous search is completed. Several researchers have explored various SRAM based pipelined architectures to improve the througput [4,5].

In these architectures, only a single pipeline stage is used for mapping a binary trie. This single pipeline stage is composed of multiple connected sub blocks that represent a node in the trie. Each sub block utilizes an SRAM unit to store node information. Each new search request starts in the boundary sub block and proceed until the LPM node is encountered.

An SRAM based multi pipeline [13, 15, 16, 18, 19] approach improves the throughput considerably by using parallel non-intersecting and costant length pipelines having *m* different sub blocks that contains SRAM units. Each sub block is connected to each other with *n* pipelines.

In this chapter, we review and present a Block RAM based array architecture for fast IP lookup with update functionality, which is a slightly modified version of [6].

3.2 MOTIVATION

In this thesis, we implement a two dimensional multiple pipelined architecture proposed in [6] that has parallel, circular search capabilities on intersecting and variable length pipelines.

In our implementation, we added FIFOs to the system to prevent head of line blocking in Contention Resolvers and a Ram Data Loader module to load or update RAM contents of the system. We also modified the PE architecture slightly to adapt it to Xilinx FPGAs and used BRAMs instead of seperate SRAMs.

3.3 ARRAY ARCHITECTURE for FAST IP LOOKUP WITH UPDATE CAPABILITY

SRAM based array architecture for fast IP lookup (SAFIL) is composed of specially designed processing elements (PEs) that are connected like a 2D torus topology but is operated like a systolic array to benefit from multi-pipeline parallelism [6]. In the following, the systolic array like structure and 2D torus network topology are explained briefly:

Systolic array: A *systolic array* is a pipe network arrangement of processing units called cells. It is a specialized form of parallel computing, where cells (i.e. processors), compute data and store it independently of each other. Each processing element inputs data from one or more neighbors (e.g. North and West) and processes it. The output of the process is given to the neighbors in the opposite direction (e.g. South and East). The task of one cell can be summarized as receive, compute and transmit. The communication with the outside world occurs only at boundary cells. The processing elements share the information with their neighbors after performing the needed operations on the data. Figure 3-1 demonstrates an example of 4x4 systolic array architecture. The systolic arrays have attractive

properties such as synchronization, modularity, regularity, locality, finite connection, parallel pipelining and modular extendibility [6].

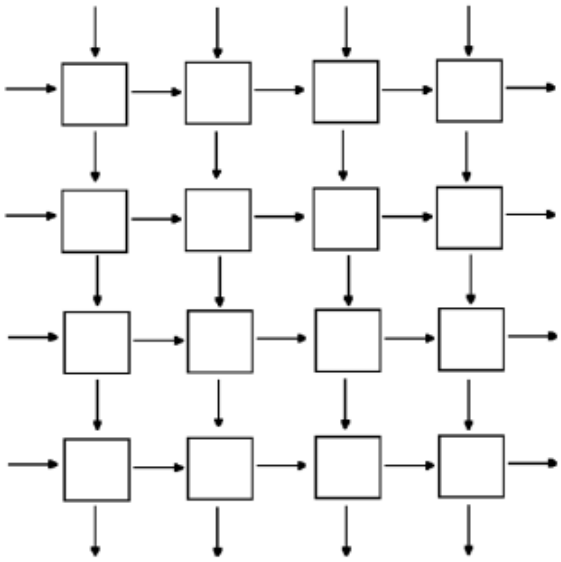

Figure 3-1 A 4x4 Systolic Array

2D-torus: 2D torus is a *k-ary* 2-*cube* network where $k \geq 3$. A *k-ary n*-cube network where *n* is the dimension of the cube and *k* is the radix, is a well-known topology used in communication networks and high performance computing architectures. It consists of $N = k^n$ nodes arranged in *n*-dimensions, with *k*-nodes per dimension. Figure 3-2 illustrates 4-*ary* 2-*cube* network or 4 x 4 torus.

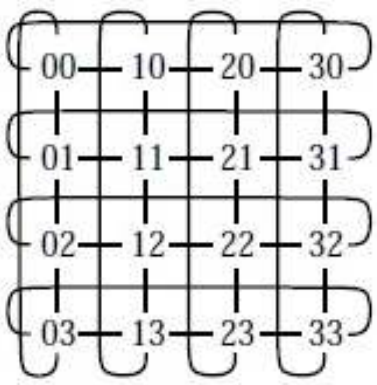

Figure 3-2 4x4 Torus

The topology in SAFIL is like a *k-ary* 2-*cube*; in particular a 2D torus, where $k \geq 3$, in which the wrap-around connections are not between PEs but rather between a PE and a contention resolver (CR) (Figure 3-3). As a result, SAFIL can be regarded as an array of PEs that are connected in a 2D torus topology and is operated like a systolic array to benefit from multi-pipeline parallelism for trie-based IP lookup.

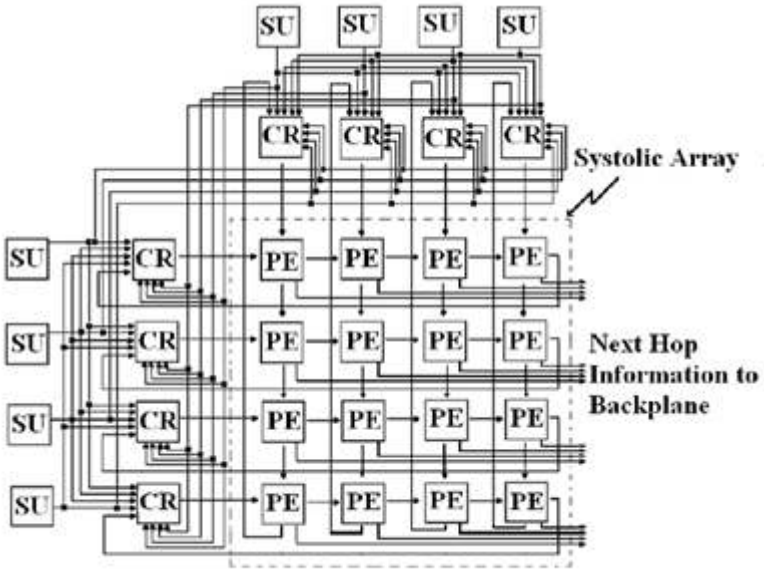

Figure 3-3 4x4 SAFIL Architecture [6]

In a systolic array in general;

i) a PE is similar to a central processing unit except for a program counter,

ii) the operations are synchronous and transport-triggered,

iii) the communication with the outside world occurs only at the array boundary,

iv) there exist structured data parallelism, strict flows along rows/columns and interaction of data streams at the PEs.

SAFIL is not exactly a systolic array since it has the above (*i*)-(*iii*) characteristics but not (*iv*) [6].

3.3.1 IP Lookup Process

An IP Lookup Process starts at an available Selector Unit (SU). The searched IP address arrives at input side of SU to start a new search operation. Then this SU, using initial *r* bits of the IP address, finds the input stage PE and the memory address of the corresponding subtrie root in this PE. Then a SAFIL Frame is constructed as shown in Figure 3-4.

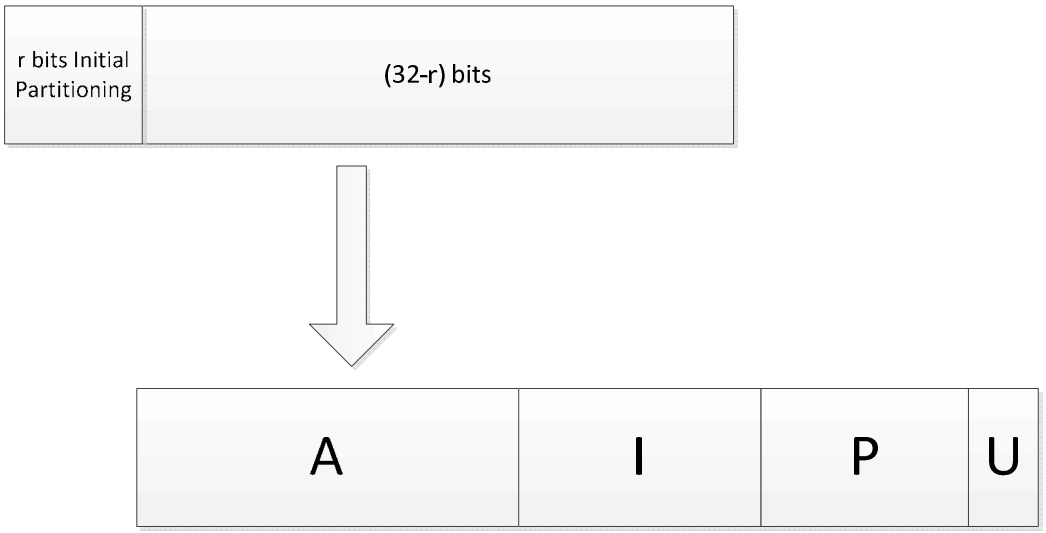

Figure 3-4 SAFIL Frame

A SAFIL Frame consists of the following four fields:

- A-field holds the least significiant (32-*r*) bits of the IP address being searched (most significiant *r* bits are to be used for initial partitioning).
- I-field is a pointer to the Block RAM in PE.
- P-field holds the search result that the IP packet will use to reach to the next router
- U-field holds the type of the frame (IP Lookup or IP Update).

Since more than one search requests may arrive at an input stage PE, in this case contention occurs. The contention resolver (CR) is used to get a SAFIL frame that

contains the IP address to be processed into the system. When contention occurs, only one of the contented packets is selected and the others are put on hold using a suitable strategy. Each SU is connected to every other CR. Number of SUs is a design choice and defines the maximum number of search requests that can be admitted to the system simultaneously. The endpoints of each row and column are connected to their corresponding CRs. Hence a pipeline corresponding to a branch in prefix tree can be mapped onto the array of PEs by wrapping it around the corresponding row or column. If a circulating search exists, other search requests from SUs are accepted by CR into "inside FIFOs". The backplane obtains the search result from any of the PEs.

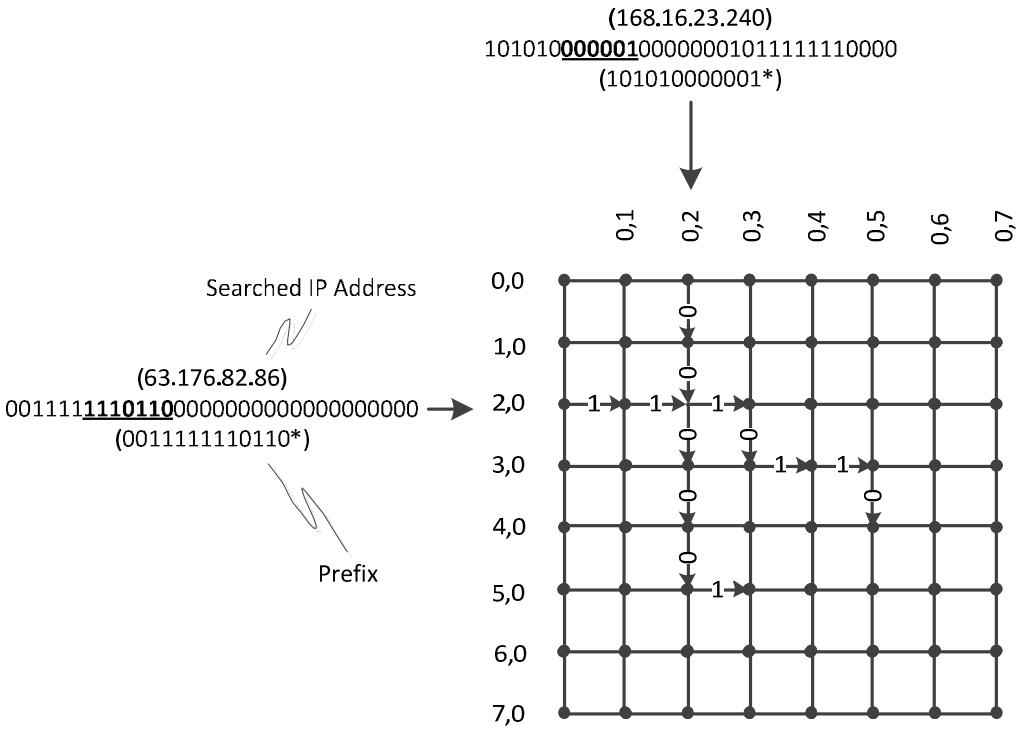

Figure 3-5 Propagation of a SAFIL frame during lookup

Figure 3-5 illustrates the lookup process for two different addresses on an 8x8 SAFIL system assuming an initial stride for partitioning as $r = 8$. These two search keys are assumed to enter into the system at the same time. While search key is traversing through each PE, SAFIL frame is updated and mapped to one of the child node according to MSB of it. If the stored prefix node is valid, the port number field

in the traversing SAFIL frame is updated. At the end of the search operation, port number is output to the backplane.

3.3.2 IP Lookup Table Update and Propagate Process

A binary trie is initially partitioned into several disjoint subtries. These subtries are then mapped on SAFIL starting from the input stage PE (to which CRs ans SUs are connected). The Block RAM contents of each PE is loaded by Ram Data Loader by constructing SAFIL Update Frames. This unit is connected to all CRs located in the northern side of the structure.

A SAFIL Update Frame (shown in Figure 3-6) consists of four fields, namely *m*-bit ram data (D), *p*-bit Block RAM index (I), *n*-bit PE ID code (PE) and 1-bit frame type (U). D-field holds *p*-bit Block RAM data that will be updated, I-field holds Block RAM Address, PE-field holds the identification code of the PE and U-field is the type of the frame (if SAFIL Update Frame U='1' otherwise U='0').

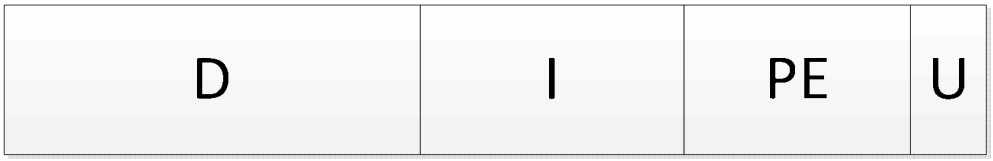

Figure 3-6 SAFIL Update Frame

An update process begins at Ram Data Loader (RDL) by constructing SAFIL update frame. This frame comes to the corresponding column (one of northern side CRs). Since each PE in one column has a unique ID, only one ID of the update packet will match the destination PE ID. The update packet propagates along with the column until the mached ID's are encountered. The propagation of the update packet between the PEs is named as "propagation process". If matching occurs in any PE, this is named as "update process".

3.3.3 Processing Element

A PE consists of two FIFO queue blocks, a Block RAM, a Data Flow Manager and additional combinational logic as shown in Figure 3-7.

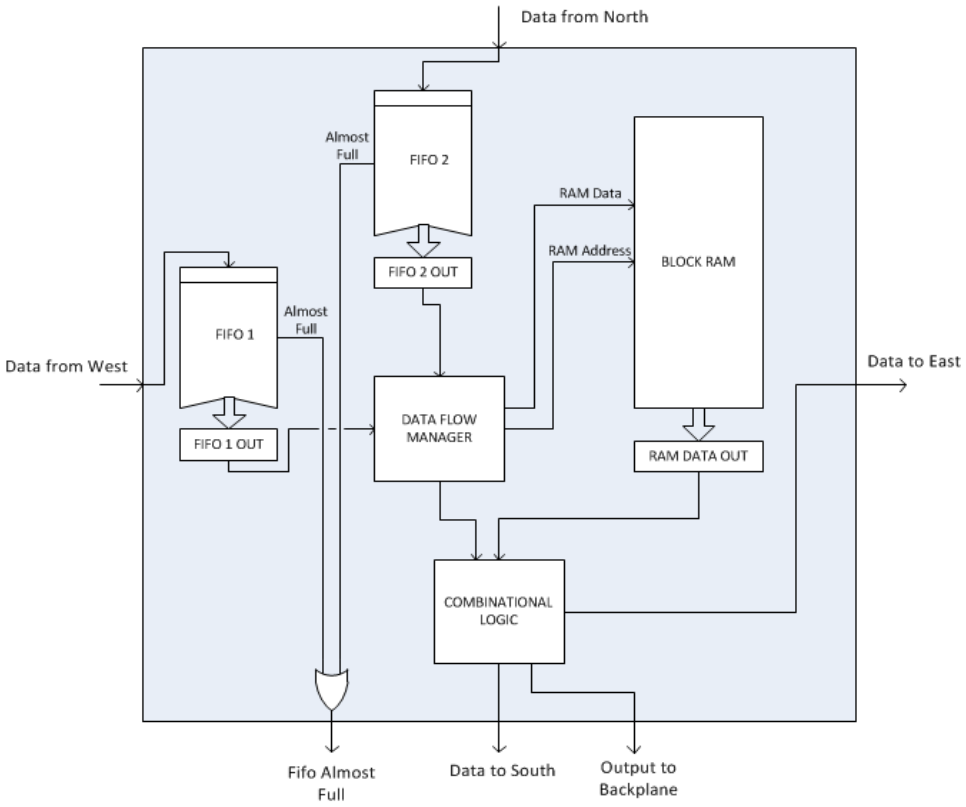

Figure 3-7 Block Diagram of the Processing Element (PE)

Each block in the PE is explained below:

FIFO Block: This block is used for buffering data coming from west or north side of the PE.

Block RAM: This memory element is used to store trie nodes.

Data Flow Manager: This unit is used for managing the data flow coming into PE. It manages two FIFOs using a Round Robin Scheduler by enabling only one of the FIFOs at each cycle. It also decides whether the incoming frame is a SAFIL Frame or a SAFIL Update Frame. Then, according to frame type, it starts one of either lookup, update or propagate processes.

Combinational Logic: This logic is used for deciding whether the IP address searched encounter an LPM or not in the current node of the trie (LPM node: there

is a valid prefix and a match in the current node). With this decision, this logic can modify the SAFIL Frame and put this frame at the output (either south or east) or send the result (i.e., output port information) to the backplane.

A SAFIL Frame consists of four fields, namely *t*-bit address (A), *p*-bit Block RAM index (I), *q*-bit port number (P) and 1-bit frame type (U). A-field holds the least significant *t*-bits of the IP address being searched (most significant (32-*t*) bits are to be used for initial partitioning, I-field is a pointer to the Block RAM in PE, P-field holds the search result that the IP packet will use to reach to the next router and U-field is the type of the frame. (if SAFIL Frame U='0', otherwise U='1')

In addition to the (t+p+q+1)-bits wide data bus connection, a single bit data available (DAV) signal between two neighboring PEs is also used. Each Block RAM unit stores (2p+q+1) bits in each entry, having two p-bit fields of south (SI) and east (EI) Block RAM indices, a q-bit port number (PN) field and a valid (V) bit (indicating whether the current trie node is a prefix or an intermediate node). A PE modifies the P-field in SAFIL frame if the current node is a valid prefix node. A SAFIL frame carries the latest longest matched port number through each traversed PE not to backtrack from the last stage when a search terminates.

Each PE's behavioral structure is shown in Figure 3-8. In each two clock cycle, a PE functions as follows:

i. SAFIL/SAFIL Update Frame arrives from northern or western input ports
ii. Data Flow Manager finds the Frame type:
iii. If the frame is a SAFIL Frame, Combinational Logic decides if the current node is an LPM node or not.
    a. If LPM node, q-bits wide port result is output to backplane by using the information read with a single access from Block RAM.
    b. If not, Combinational Logic modifies the frame by using the information read with a single access from Block RAM and this modified frame is guided to one of eastward or southward output ports.

If the frame is a SAFIL Update Frame, Data Flow Manager modifies the data row of the relevant Block RAM.

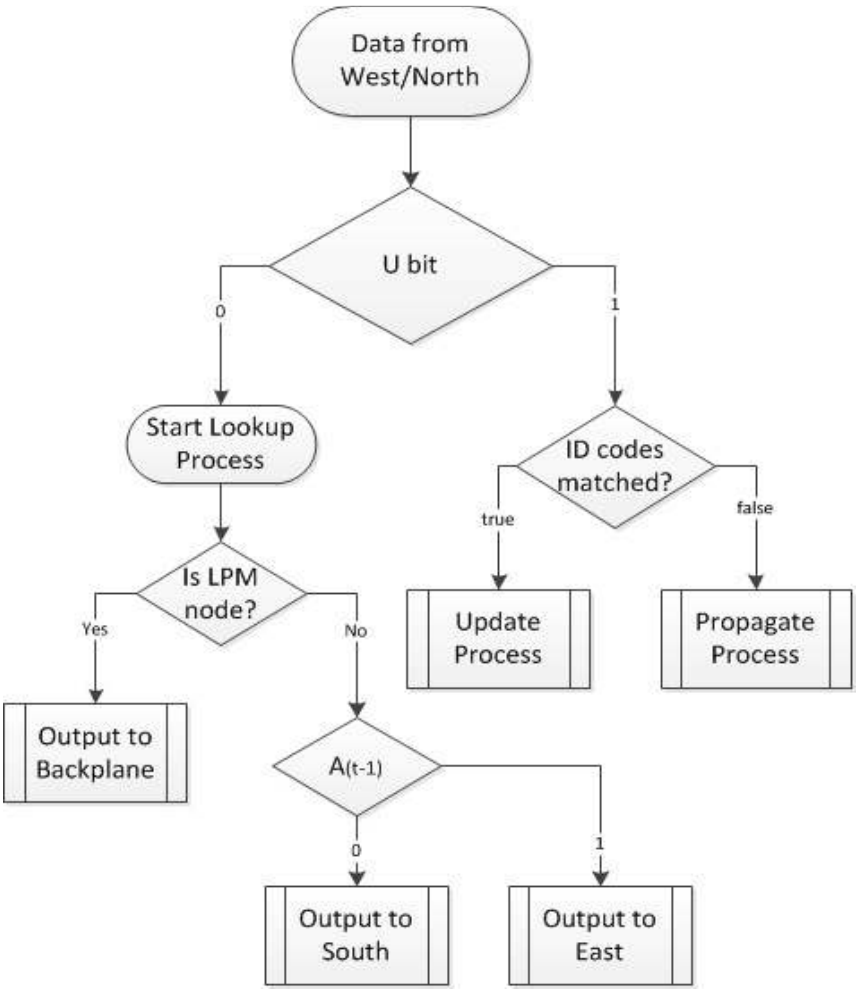

Figure 3-8 Flowchart of Data Processing in PE

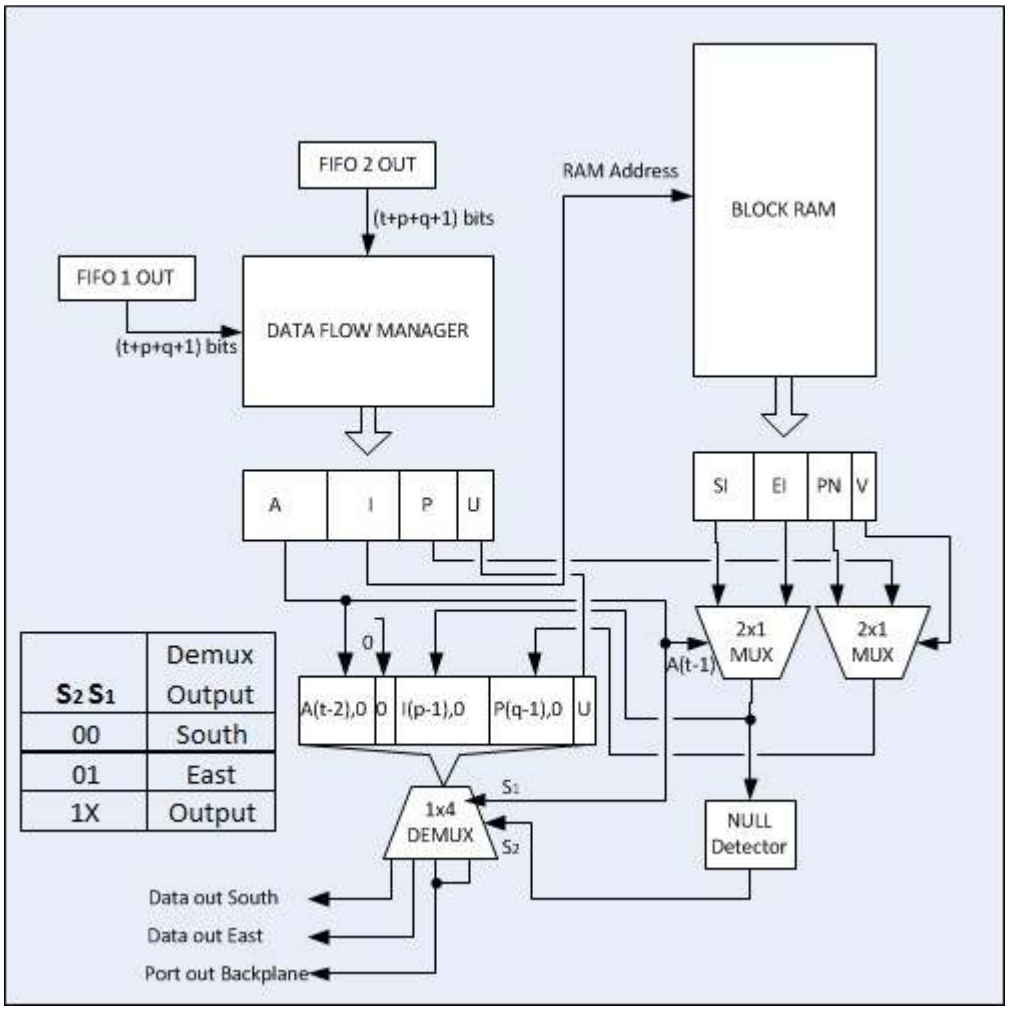

Figure 3-9 Detailed Block Diagram of a PE (Lookup Process)

Figure 3-9 presents the block diagram of a Processing Element in SAFIL for lookup process.

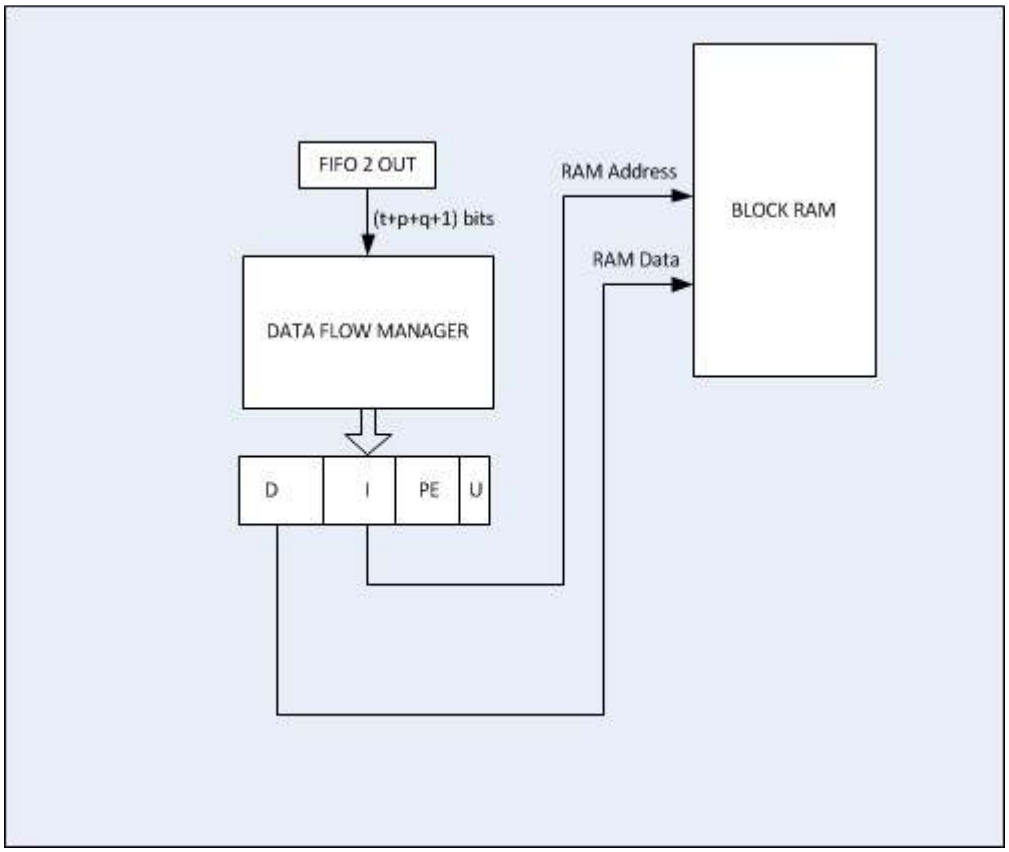

Figure 3-10 Block Diagram of a PE (Update Process)

Figure 3-10 presents the block diagram of a Processing Element in SAFIL for update process.

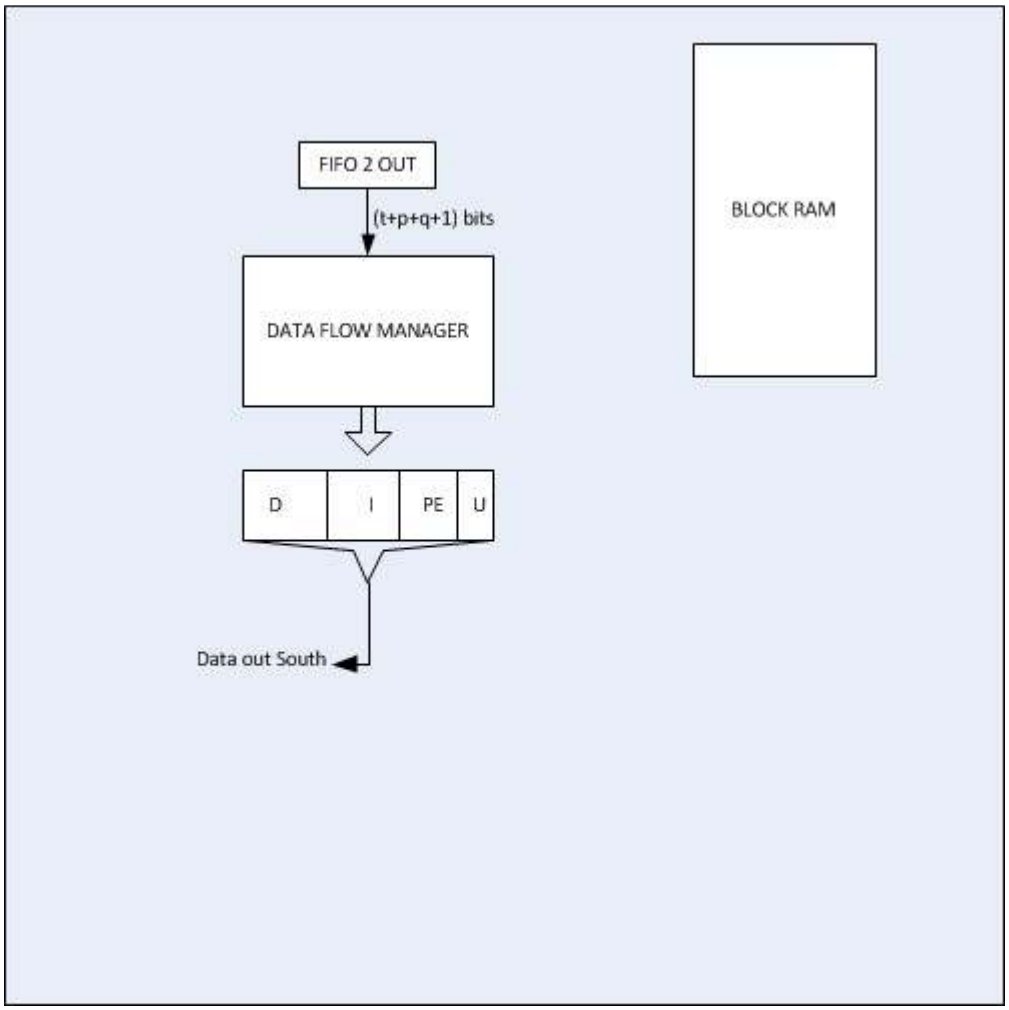

Figure 3-11 Block Diagram of a PE's Propagate Process

Figure 3-11 gives the block diagram of a SAFIL Processing Element's propagate process.

3.3.4 Selector Unit

This unit is used for initial partitioning and constructing SAFIL Frames. Selector Unit is a combinational logic that inputs the destination IP address and processes its initial *r*-bits (r is used for initial partitioning and was discussed in Section 3.3.1). SU functions as follows:

i. It finds the input stage PE by checking the initial r-bits of the IP address.
ii. It constructs the SAFIL Frame by adding the memory address of the root node of the corresponding subtrie.

iii. It puts the constructed SAFIL Frame on its output port.

3.3.5 Contention Resolver

This unit is used for buffering and arranging incoming data from different SU's and wrapped around eastern and southern PE's. Since more than one search request may arrive at an input stage PE, there is a need for a mechanism in order to accept an IP address into the system to be searched. Therefore, only one search operation will be accepted by a Contention Resolver. When contention occurs, only one of the contented packets is selected and routed to the connected PE while the others being stored in FIFO's in each Contention Resolver.

Since more than one packet may arrive at each CR in one cycle, we implemented a FIFO block in each port in a CR to accept incoming data. Since more than one packet want to leave CR at the same time, there is also a need for arranging these packets. A suitable strategy, such as Round Robin for example, can be employed for this operation.

CR functions as follows:

i. SAFIL Frames or SAFIL Update Frames arrives at the input of the CR.
ii. At each input port, receiving data enters the corresponding FIFO.
iii. CR enables one of the FIFOs for taking data to the output port.

3.3.6 Congestion Control Unit

This unit is designed and used to prevent possible FIFO overflow in any PE. Since FIFO's in each PE have limited size, packet loss due to queue overflow is always possible. Using a simple congestion control mechanism, one can control the incoming traffic rate by activating or deactivating the input ports if any of the FIFO's is almost full. A one bit connection from each PE's queue to congestion control unit (CCU) is sufficient for this purpose. One possible congestion control algorithm in activating and deactivating the SU's is additive increase multiplicative

decrease strategy. If the FIFO queue usage exceeds the predefined threshold value, then half of the SUs are deactivated to decrease the load at the input of the whole system. SUs to be deactivated can be chosen arbitrarily because each SU may receive packets from the input queue and has direct connections to each CR. If there is no congestion, number of active SUs are increased by one at each cycle.

CCU functions as follows:

i. If one of the FIFO's in any PE reaches to predefined threshold value, one bit *almost full signal* becomes logic '1'.
ii. CCU deactivates SUs with multiplicative decrease strategy. For example, if *n* SU's are active at some time and one of the FIFO's is almost full, CCU deactivates half of the SU's not to accept a search key (IP Address) anymore having *n*/2 active SUs for the next cycle.
iii. CCU activates SUs with additive increase strategy. For example, if *n* SU's are active at some and none of the FIFO's are almost full, CCU activates one of the inactive SUs making a total of *n*+1 SU's are active.

3.3.7 Ram Data Loader

This unit is used for loading and updating the contents of the Block RAM's inside each PE. Each load or update operation starts at Ram Data Loader (RDL). RDL examines the incoming packet and finds the column, which contains the PE that will be updated. Then, RDL constructs SAFIL Update Frame and sends it to CR of the corresponding column. As was explained in Section 3.3.2, SAFIL Update Frame travels through this column until the ID of the PE and Frame matches. If such a match occurs, the selected PE content will be updated.

Since the update process can be performed while lookup processes are running, the system can be modified without being stopped, which is very beneficial.

CHAPTER 4

FPGA IMPLEMENTATION OF THE ARRAY ARCHITECTURE FOR FAST IP LOOKUP WITH UPDATE CAPABILITY

Our Block RAM based array implementation of SAFIL is composed of 8x8 specially designed processing elements (PEs) that are connected like a 2D torus topology, buffered CRs that connects all SUs to corresponding PE, Ram Data Loader that is used for loading and updating RAM content and finally CCU that is used to regulate the incoming traffic. In this chapter, all of these blocks and sub modules will be explained and detailed including design and simulation studies.

4.1 PROCESSING ELEMENT

A PE consists of two FIFO blocks, a Block RAM, a Data Flow Manager and additional combinational logic. The 49 bit input port of each PE is connected to west and north neighbors. The 49 bit output port of each PE is connected to east and south neighbors. Each PE is also connected to Congestion Control Unit (CCU) by one bit data line.

4.1.1 Design

The PE unit is composed of blocks and sub-units explained below:

FIFO Block: This block is used for buffering data coming from western or northern sides of the PE.

FIFOs are implemented using distributed RAM's on FPGA. To achieve a high speed IP lookup process, FIFO Blocks are designed with "first word fall through" attribute. With this, data written to an empty FIFO appears on the read port at the same clock cycle.

To avoid data overflow in FIFO, *fifo_full* output signal is used for warning CCU to reduce incoming traffic rate. To do this, FIFO Block asserts this almost full signal to high when FIFO memory usage reaches the predefined threshold level.

Since SAFIL Frame and SAFIL Update Frame are 49 bits wide, FIFO width is 49 bits and depth can be choosen as a result of some optimization trials. In our design, FIFO depth was selected as 1024.

The schematic view of FIFO is given in Figure 4-1.

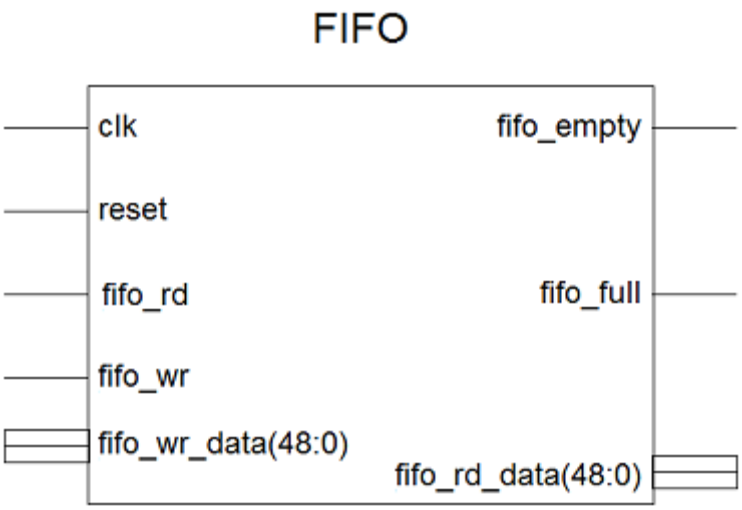

Figure 4-1 FIFO Input and Output Signals

Each in and out signals of FIFO are described in Table 4-1.

Table 4-1 FIFO Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
rst	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the FIFO module into a state that is ready to receive data.
clk	In	1 bit	The clock signal is used in order to synchronize FIFO with other modules
fifo_rd	In	1 bit	The read enable signal
fifo_wr	In	1 bit	The write enable signal
fifo_wr_data	In	49 bits	Data written to FIFO
fifo_empty	Out	1 bit	Indication signal of fifo empty
fifo_full	Out	1 bit	Indication signal of fifo is almost full
fifo_rd_data	Out	49 bits	Output data of FIFO

The VHDL source code for designing FIFO is given in Appendix A.

Block RAM: This memory element is used to store trie nodes.

Each Block RAM has 32 bits width and $2^{13} = 8192$ bits depth. Total memory size for one PE is 8k x 32 = 256 kbits. Since the whole system consists of 64 PEs, total memory size is 64 x 256 kbits = 16,777,216 mbits $\cong$ 2 MByte.

Block RAM's were generated with IP CORE in Xilinx ISE as shown in Figure 4-2.

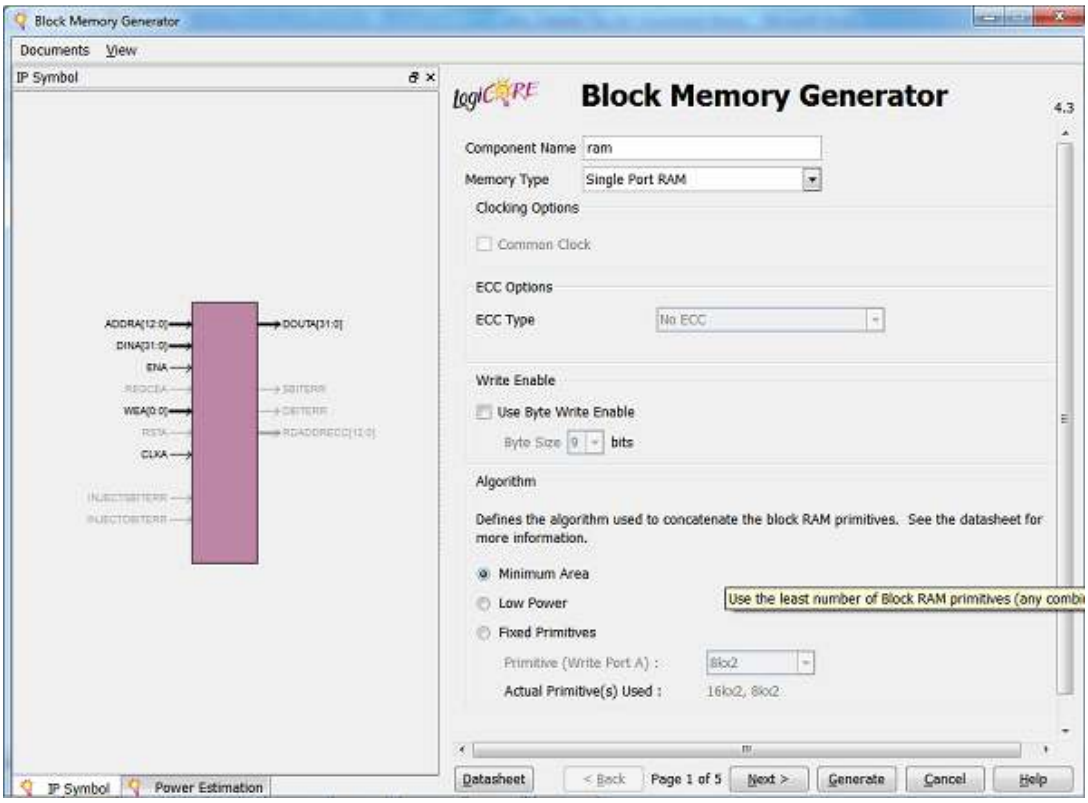

Figure 4-2 IP Core Menu for Block RAM Generation

Data Flow Manager (DFM): This unit is used for managing the data flow coming into PE. It manages two FIFOs with respect to a round robin schedule by enabling only one FIFO at each cycle. It also checks if the incoming frame is a SAFIL Frame or a SAFIL Update Frame and then it starts either lookup, update or propagate processes according to frame type.

The schematic view of Data Flow Manager is given in Figure 4-3.

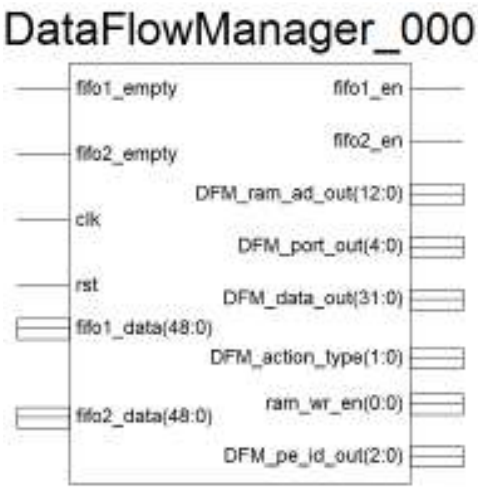

Figure 4-3 Data Flow Manager Schematic View

The VHDL source code for designing DFM is given in Appendix B.

Combinational Logic: This logic is used for deciding whether the IP address searched hits an LPM node or not. With this decision, the logic can modify the SAFIL Frame and directs it to output (either south or east) or send the output port information to backplane.

Combinational Logic inputs the outputs of the DFM.

All of these blocks and sub modules form the PE, whose input and output signals are illustrated in Figure 4-4 Processing Element

Each in and out signals of PE are described in Table 4-2 and the VHDL source code for designing PE is given in Appendix C.

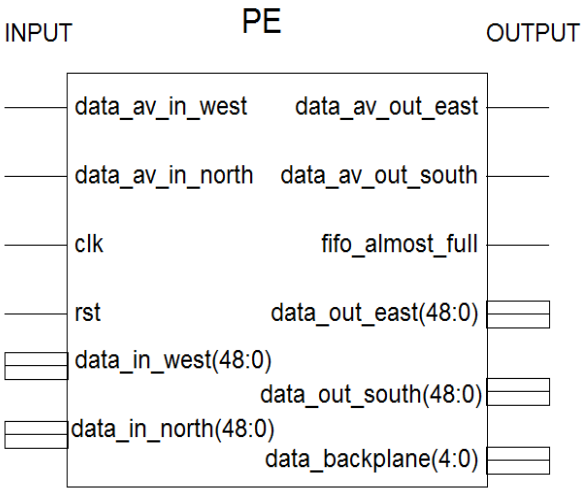

Figure 4-4 Processing Element

Table 4-2 PE Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
rst	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the PE module into a state that is ready to receive data.
clk	In	1 bit	The clock signal is used in order to synchronize different parts of the PE module (cycling at a rate less than the worst-case internal propagation delays)
data_av_in_west	In	1 bit	The available signal of data from west
data_in_west	In	49 bits	The data coming from west side of the PE
data_av_in_north	In	1 bit	The available signal of data from north
data _in_north	In	49 bits	The data coming from north side of the PE
data_av_out_east	Out	1 bit	The available signal of data to east
data _out_east	Out	49 bits	The data going out to east side of the PE
data_av_out_south	Out	1 bit	The available signal of data to south
data_out_south	Out	49 bits	The data going out to south side of the PE
data_backplane	Out	5 bits	The search result (port number) of the IP address looked up
fifo_almost_full	Out	1 bit	The signal indicating one of FIFOs in the PE is almost full

4.1.2 Simulation

SCENARIO 1

Aim: To show and verify southern and eastern port outputs for simultaneously applied predefined input data from northern and western sides of the PE. (Lookup Process)

Test Code:

```
wait for 100 ns;
rst<='1';
wait for clk_period*3;
rst<='0';
wait for clk_period*10;
data_in_west <= '0' & x"EEEEE7800044";
--01110111011101110111001111000 0 0000000000001 00010 0
data_av_in_west <= '1';
wait for clk_period;
data_av_in_west <= '0';
data_in_north <= '1' & x"F83E00780040";
--11111100000111111000000000011111 0000000000001 00000 0
data_av_in_north <= '1';
wait for clk_period;
data_av_in_north <= '0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, data from west and north is sent to the input ports consecutively. Since the input data is of type SAFIL Frame, U-field is '0'. The Block RAM data content in *0000000000001* address is hex 000800EB (binary 0000000000001 0000000000011 10101 1). According to this scenario, the node that refers to tested PE is an intermediate node. Therefore, there shouldn't be any backplane output. Since $A_{t-1} = 0$ for the data from west, the southern available output should be high for one clock cycle after some delay. Since $A_{t-1} = 1$ for the data from north, the eastern available output should be high for one clock cycle after some delay.

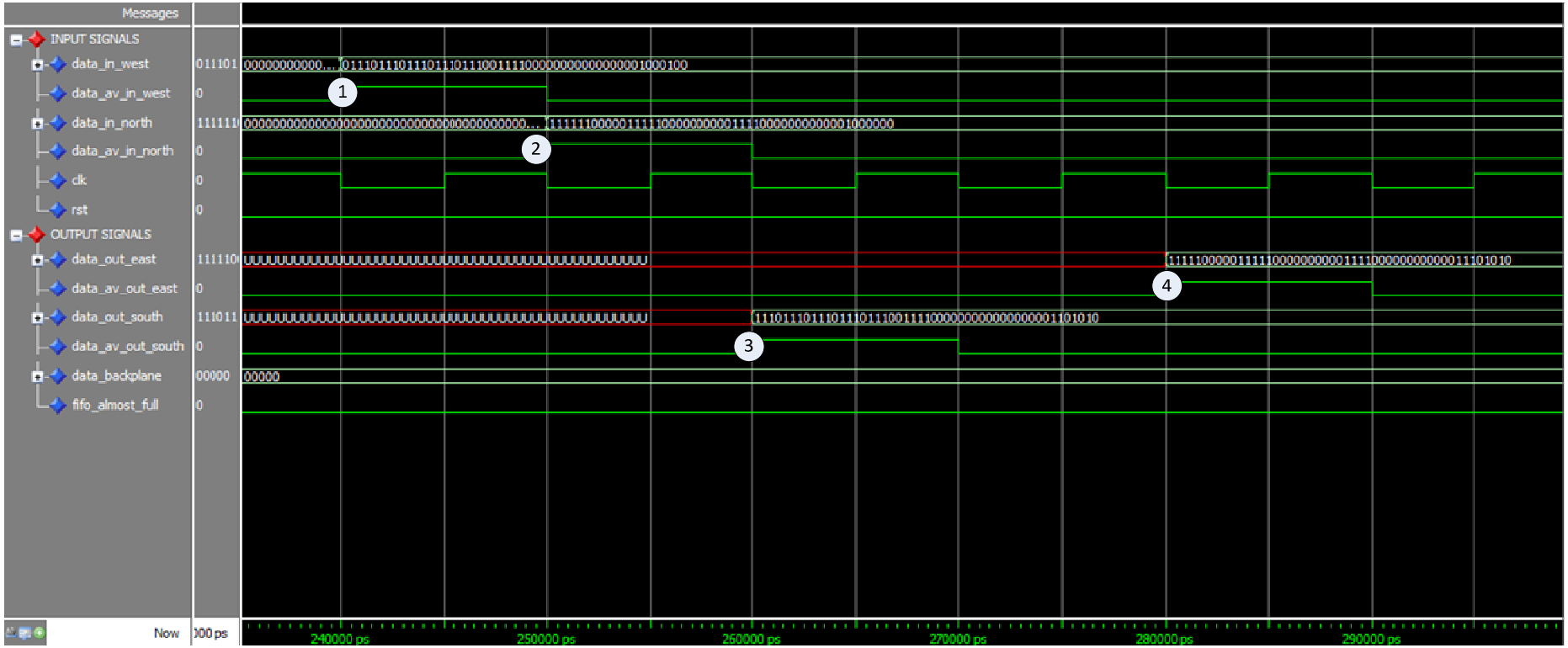

Figure 4-5 Simulation Results for PE in Scenario 1

The result of the simulation is given in **Figure 4-5**. In this scenario, *data_av_in_west* appears at label 1 and *data_av_in_north* appears at label 2. Since the input data is in the form of intermediate node data, at labels 3 and 4, *data_av_out_south* and *data_av_out_east* are activated after 2 clock cycles.

SCENARIO 2

Aim: To show and verify southern and eastern port outputs for simultaneously applied predefined input data from northern and western sides of the PE and Round-Robin Scheduler inside the DFM of the PE. (Lookup Process)

Test Code:

```
wait for 100 ns;
rst<='1';
wait for clk_period*3;
rst<='0';
wait for clk_period*10;
data_in_west <= '0' & x" EEEEE7800104";
-- 01110111011101110111001111000O  0000000000100  00010   0
data_av_in_west <= '1';
data_in_north <= '1' & x" F83E00780100";
 -- 11111100000111111000000000011111  0000000000100  00000  0
data_av_in_north <= '1';
wait for clk_period*3;
data_av_in_north <= '0';
data_av_in_west <= '0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, data from west and north is sent to the input ports concurrently for 3 clock cycles. Since the input data is of type SAFIL Frame, U-field is '0'. The Block RAM data content in *0000000000100* address is hex F4FFFFFF (binary 1111010011111 1111111111111 11111 1). According to this scenario, the node that refers to tested PE is an intermediate node. Therefore, there shouldn't be any backplane output. Since $A_{t-1} = 0$ for the data from west, the southern available output should be high for one clock cycle after some delay. Since $A_{t-1} = 1$ for the data from north, the eastern available output should be high for one clock cycle after some delay. Because there are data on both of the FIFO outputs at the same time, it verifies that Round Robin Scheduler works well.

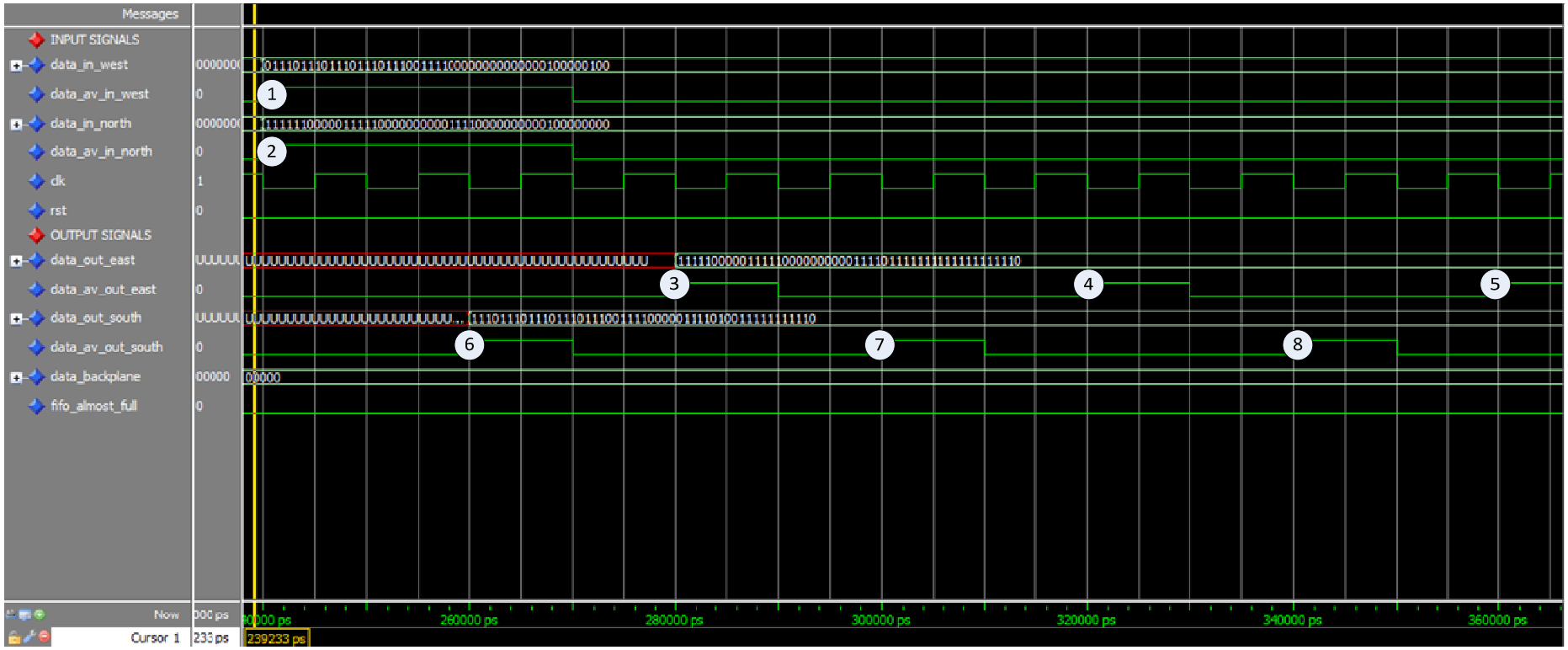

Figure 4-6 Simulation Results for PE in Scenario 2

The result of the simulation is given in **Figure 4-6**. In this scenario, *data_av_in_west* and *data_av_in_north* appear at labels 1 and 2 during 3 clock cycles. Since the input data is in the form of intermediate node data, at labels 3, 4, 5, 6, 7, 8; *data_av_out_south* and *data_av_out_east* are activated by the Round Robin Scheduler.

SCENARIO 3

Aim: To show and verify backplane outputs for simultaneously applied predefined input data from northern and western sides of the PE (Lookup Process)

Test Code:

wait for 100 ns;

rst<='1';

*wait for clk_period*3;*

rst<='0';

*wait for clk_period*10;*

data_in_west <= '0' & x"EEEEE7800004";

--01110111011101110111001111000 0 0000000000000 00010 0

data_av_in_west <= '1';

data_in_north <= '1' & x"F83E007800CE";

--11111100000111111000000000001111 0000000000011 00111 0

data_av_in_north <= '1';

*wait for clk_period*3;*

data_av_in_north <= '0';

data_av_in_west <= '0';

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, data from west and north is sent to the input ports concurrently for 3 clock cycles. Since the input data is of type SAFIL Frame, U-field is '0'. The Block RAM data content in *0000000000000* address is hex 0000000B (binary 0000000000000 0000000000000 00101 1) and in *0000000000011* address is hex 0000003F (binary 0000000000000 0000000000000 11111 1) . Since “SI= all 0's” and “EI=all 0's” in both cases, the node that refers to the tested PE is an LPM node. Therefore, there exists data at the backplane output.

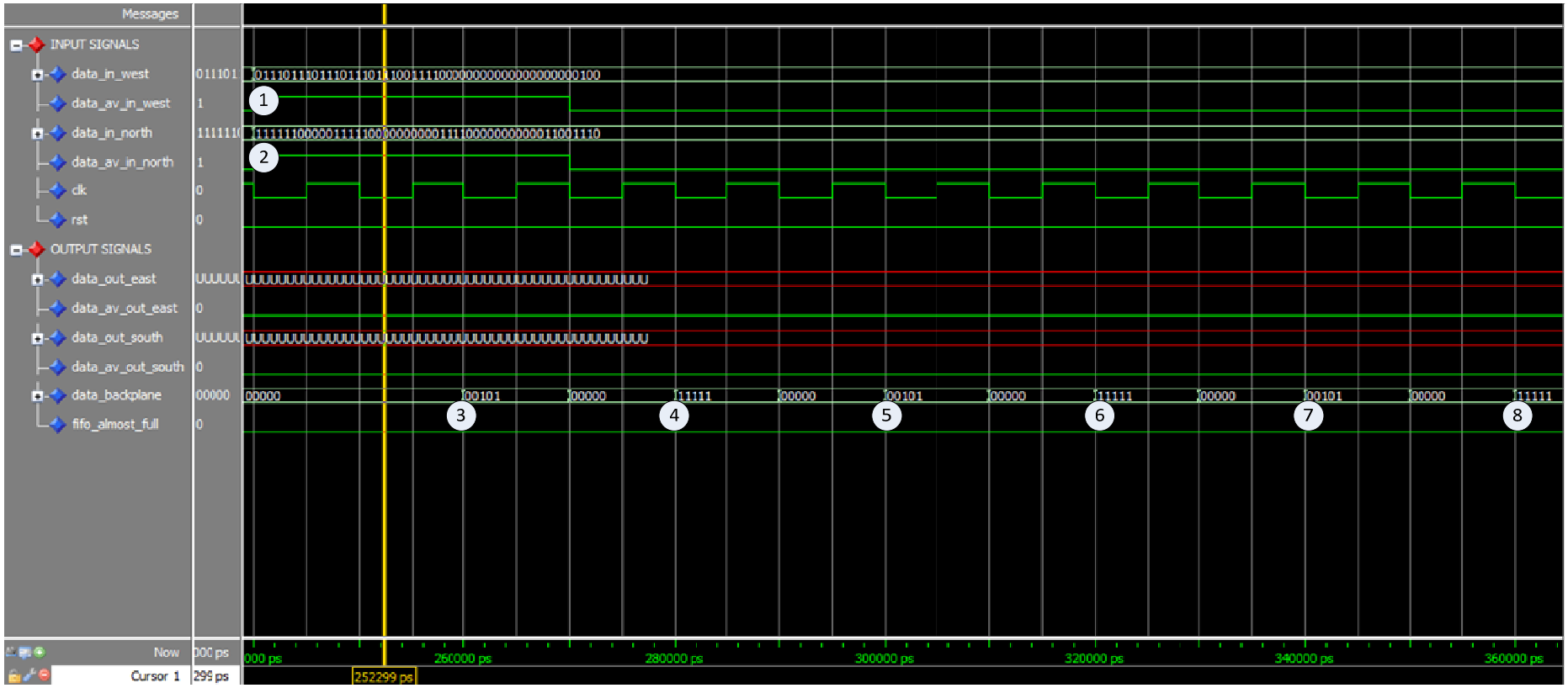

Figure 4-7 Simulation Results for PE in Scenario 3

The result of the simulation is given in **Figure 4-7**. In this scenario, *data_av_in_west* and *data_av_in_north* appear at labels 1 and 2. Since the input data is in form of longest prefix node data, at labels 3, 4, 5, 6, 7 and 8 *data_backplane* output is activated.

SCENARIO 4

Aim: To show and verify southern output for predefined input data applied from northern side of the PE for IP update process (Propagate Process)

Test Code:

```
wait for 100 ns;
rst<='1';
wait for clk_period*3;
rst<='0';
wait for clk_period*10;
data_av_in_west <= '0';
data_in_north <= '1' & x"F83E00780003";
--11111100000111111000000000111100  0000000000000  001  1
data_av_in_north <= '1';
wait for clk_period*3;
data_av_in_north <= '0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, data from north is sent to the input port for 3 clock cycles. Since the input data is of type SAFIL Update Frame, U-field is '1'. Therefore, PE decides which process (Update or Propagate) to run. In this case, propagate process should run because PE ID and ID of the incoming SAFIL Update Frame does not match. Therefore, there exists data at the southern output port.

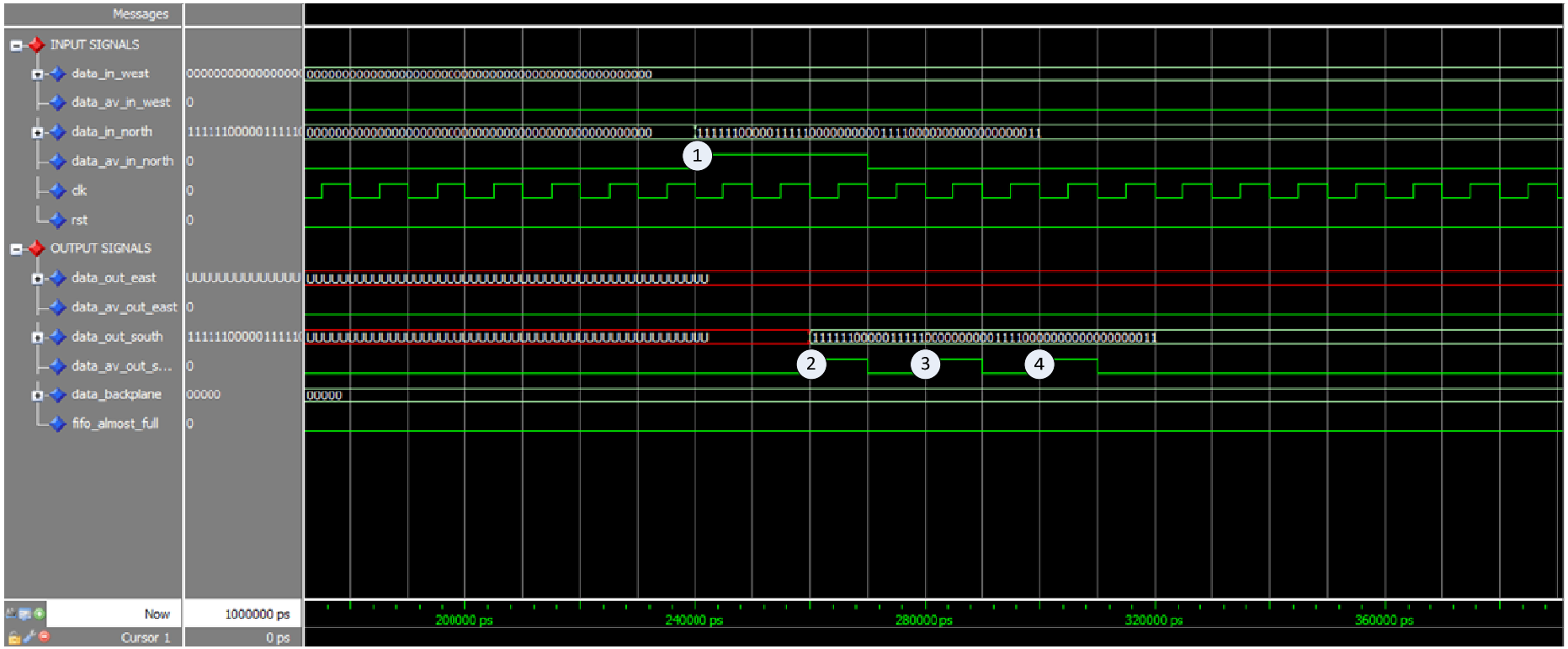

Figure 4-8 Simulation Results for PE in Scenario 4

The result of the simulation is given in **Figure 4-8**. In this scenario, *data_av_in_north* appears at label 1. Since the input data is in form of propagate process data, at labels 2, and, 4 *data_av_out_south* output is activated.

SCENARIO 5

Aim: To verify and show that Block RAM data contents can be loaded/updated via the update process (Update Process).

Test Code:

```
wait for 100 ns;
rst<='1';
wait for clk_period*3;
rst<='0';
wait for clk_period*10;
data_av_in_west <= '0';
data_av_in_north <= '0';
wait for clk_period;
data_in_west <= '1' & x"EEEEE7800040";
--11110111011101110111001111000 0000000000001 00000 0
data_av_in_west <= '1';
wait for clk_period;
data_av_in_west <= '0';
wait for clk_period;
data_in_north <= '1' & x"FFFFFFFE0011";
--11111111111111111111111111111111 0000000000001 000 1
data_av_in_north <= '1';
wait for clk_period;
data_av_in_north <= '0';
wait for clk_period;
data_in_west <= '1' & x"EEEEE7800040";
 --11110111011101110111001111000 0000000000001 00000 0
data_av_in_west <= '1';
wait for clk_period;
data_av_in_west <= '0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, data from west is sent to the input port for 3 clock cycles. First, a SAFIL Frame arrives at the western side of the PE. After 2 clock cycles, a SAFIL Update Frame appears in the Northen side. Since the input data is of type SAFIL Update

Frame, U-field is '1'. Therefore, PE decide which process (Update or Propagate) to run. In this case, update process should run because PE ID and ID of the incoming SAFIL Update Frame match. Finally, the same SAFIL Frame appears again at the western side of the PE to observe that the same SAFIL Frame provides a different SAFIL Frame output after ram update process.

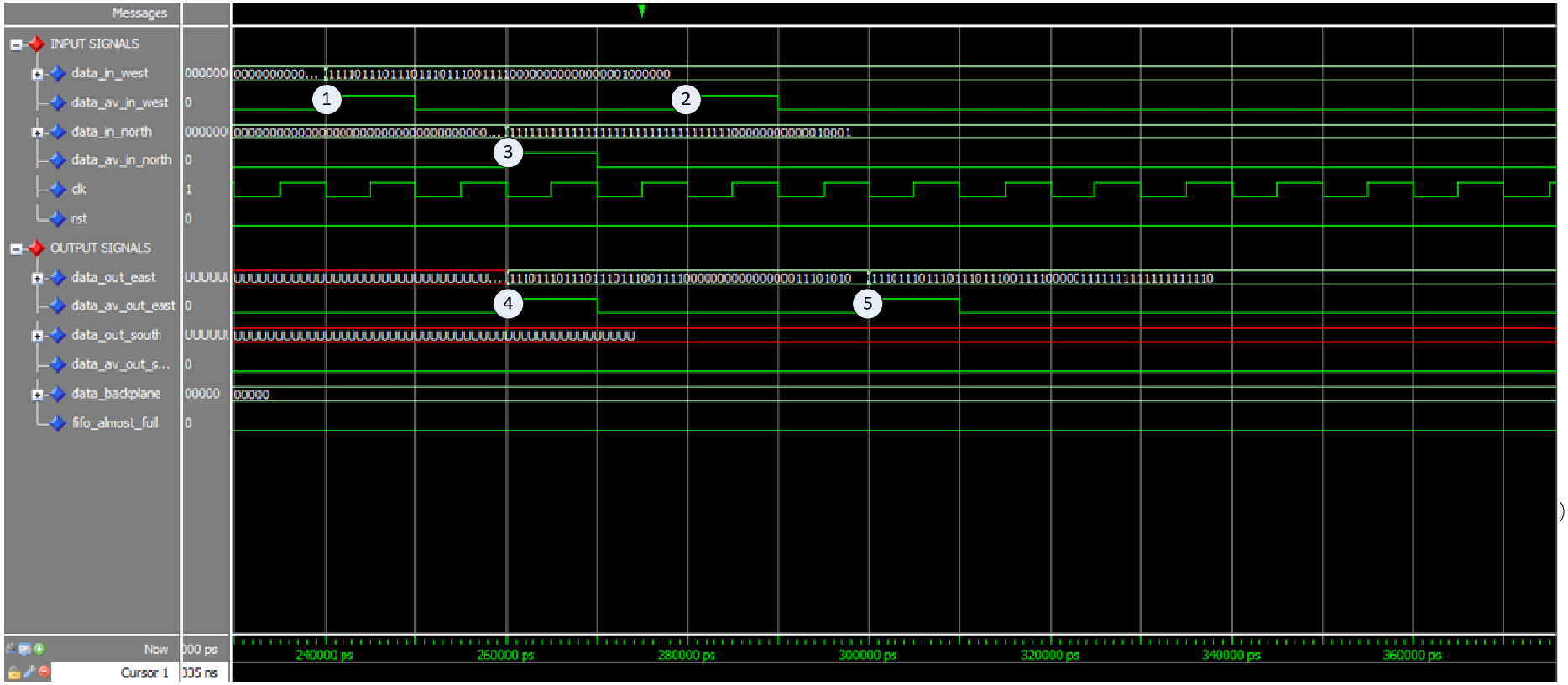

Figure 4-9 Simulation Results for PE in Scenario 5

The result of the simulation is given in **Figure 4-9**. In this scenario, *data_av_in_west* appears for a lookup process at label 1. Then, *data_av_in_north appears* for an update process at label 3. Then, *data_av_in_west* appears for a lookup process again at label 2. Before the update process is completed, the corresponding output for *data_out_east* is activated at label 4. After the update process is completed, the corresponding output for *data_out_east* is activated at label 5.

4.2 SELECTOR UNIT

This unit is designed and used for initial partitioning and for constructing SAFIL Frames.

4.2.1 Design

Just a 4x16 line decoder is sufficient for initial partitioning (Table 4-3). Using leftmost 4-bits of the key IP address, the seacrh is directed to the corresponding CR and PE.

Table 4-3 Initial Partitioning Conversion

Initial 4 Bit	Output Port of SU
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

The 32-bit input port of each SU is connected to incoming traffic port, i.e. the IP address to be searched. The output ports of each SU are connected to all of the CR's input ports.

The schematic view of SU is given in Figure 4-10.

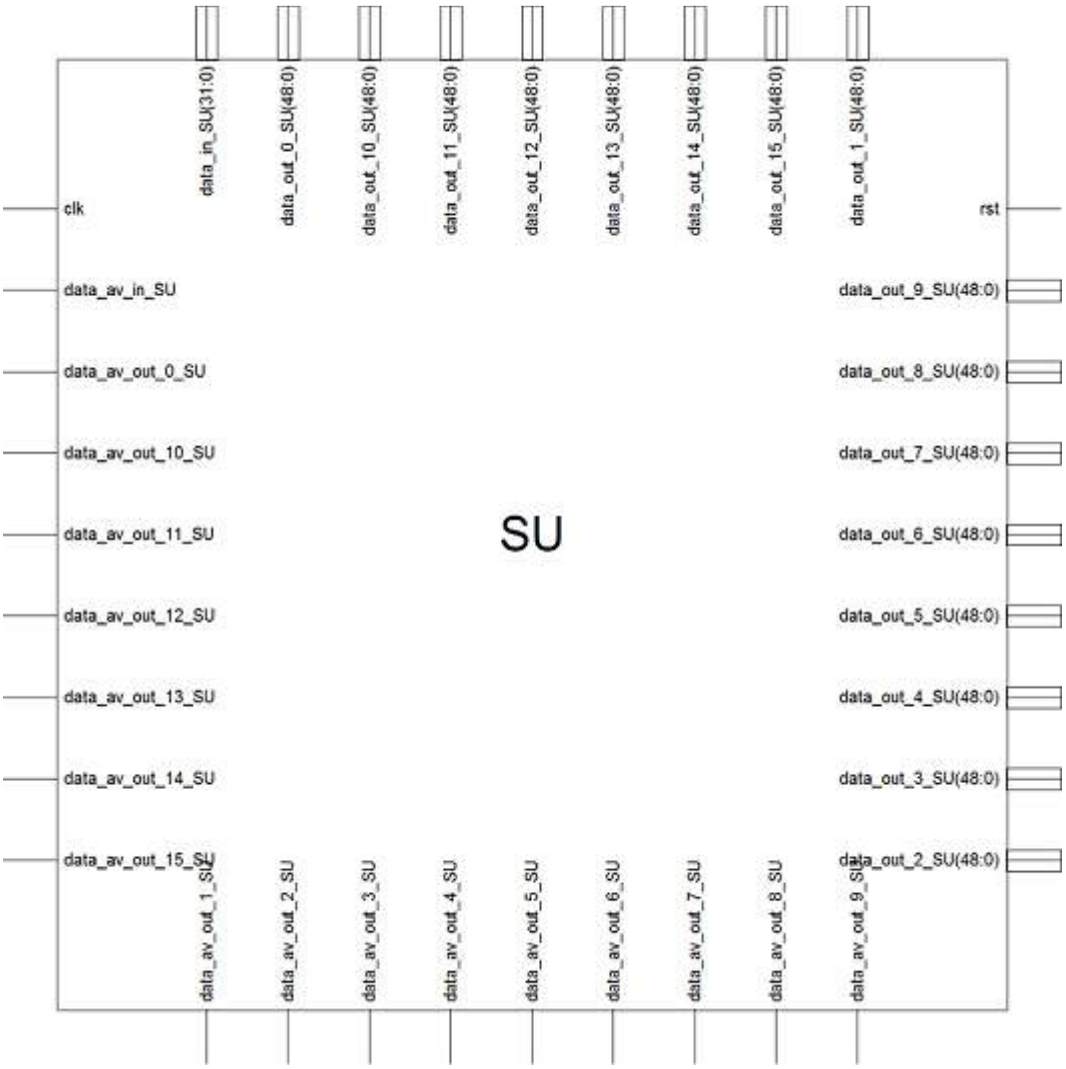

Figure 4-10 Selector Unit Input and Output Signals

In and out signals of SU are described in Table 4-4.

Table 4-4 SU Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
rst	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the SU module into a state that is ready to receive data
clk	In	1 bit	The clock signal is used in order to synchronize SU with other modules
data_av_in_SU	In	1 bit	The available signal of data incoming to SU
data_in_SU	In	32 bits	The data incoming to SU
data_av_out_X_SU	Out	1 bit	The available signal of data outgoing from SU X
data _out_X_SU	Out	49 bits	The data outgoing from SU X

In each falling edge of a cycle, SU checks data available input. If data is available, SU gets the 32-bit wide IP Address into the block. After initial partitioning stage, SU finds the input stage CR and constructs the SAFIL Frame while outputing it at the corresponding output port .

4.2.2 Simulation

SCENARIO 1

Aim: To show and verify that SU constructs SAFIL Frame and outputs it on the corresponding output port.

Test Code:

```
wait for 100 ns;
rst <= '1';
wait for clk_period*3;
rst <= '0';
wait for clk_period*10;
data_av_in_SU <= '1';
data_in_SU <= x"ffffffff" ;
wait for clk_period;
data_in_SU <= x"cfffffff" ;
wait for clk_period;
data_in_SU <= x"0fffffff" ;
wait for clk_period;
data_in_SU <= x"2fffffff" ;
wait for clk_period;
data_av_in_SU <= '0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, data that has different initial partitions at each cycle appears in the input port. According to this scenario, different output ports of the SU be active at each cycle.

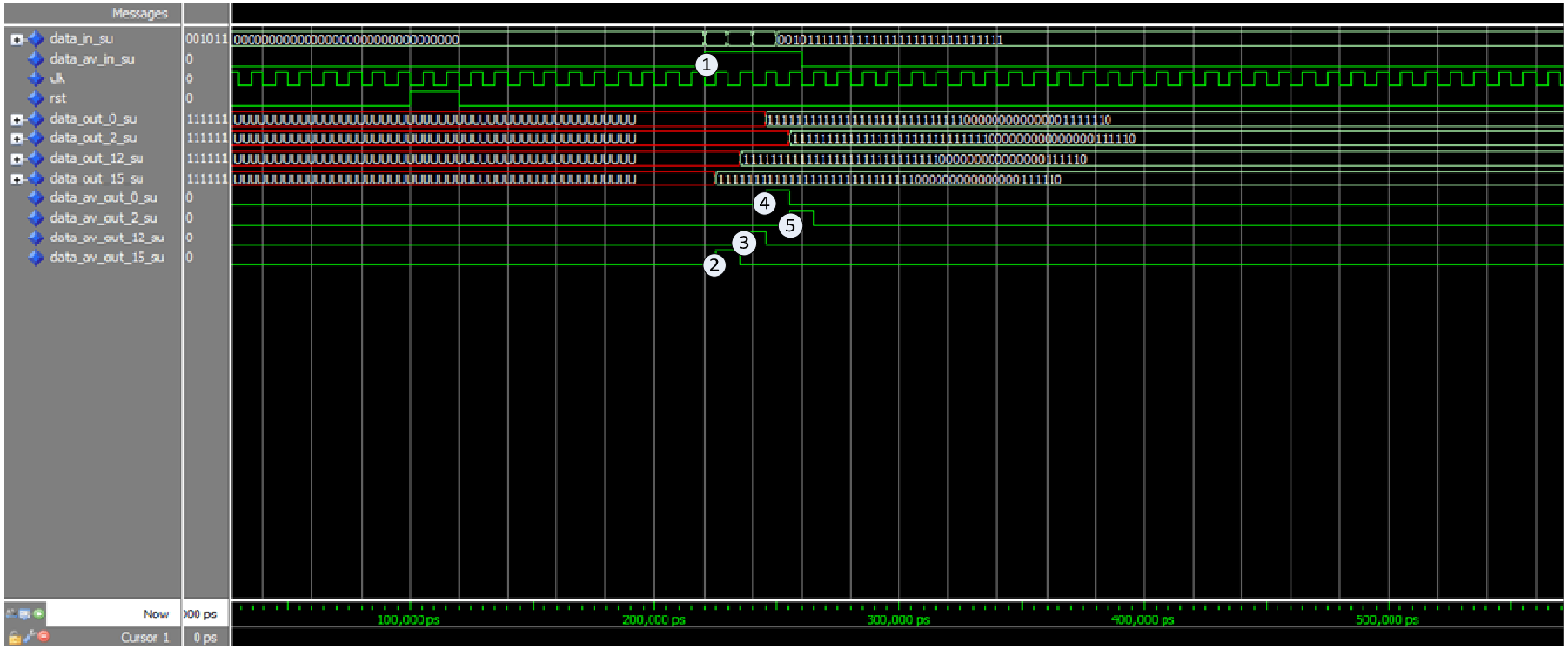

Figure 4-11 Simulation Results for SU in Scenario 1

The result of the simulation is given in **Figure 4-9**. In this scenario, *data_av_in_su* appears at label 1. Then, *data_out_x_su* appears correspondingly at labels 2, 3, 4 and 5.

4.3 CONTENTION RESOLVER

This unit is designed and used for buffering and arranging incoming data from different SU's and eastern and southern PE's.

4.3.1 Design

The 49-bit input ports of each Contention Resolver (CR) is connected to the output ports of SUs and to the western or northern sides of the PE.

The block diagram of CR is given in Figure 4-12. Data from any SU or RDL arrives into the relevant FIFO. Round Robbin Scheduler then selects one of the incoming data and directs it to CR out by taking into account that data coming from PE has higher priority.

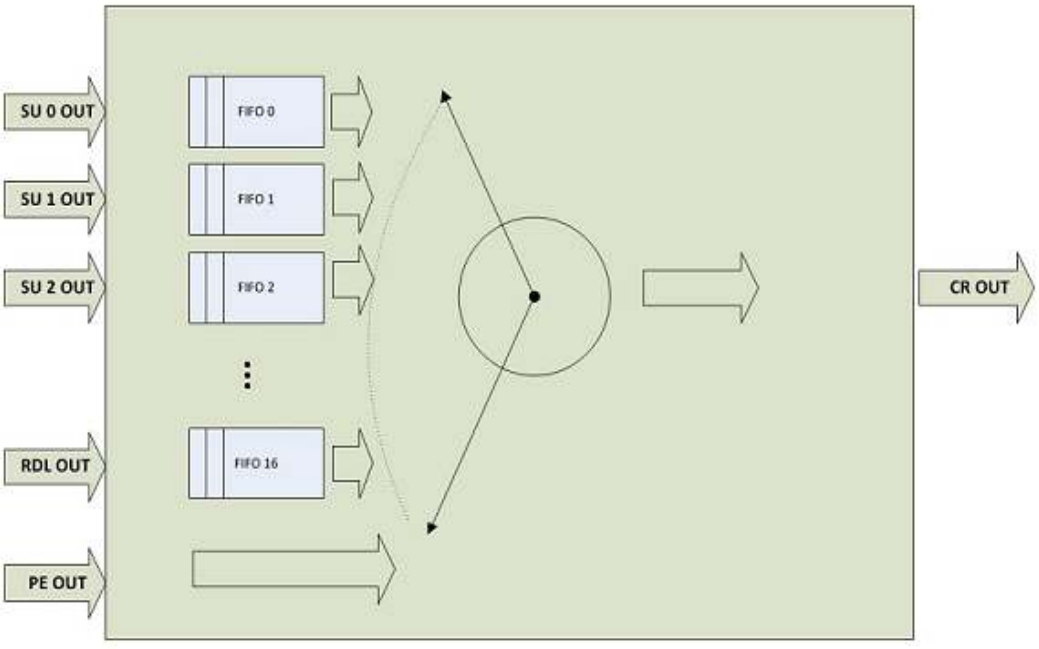

Figure 4-12 Block Diagram of CR

The size of each FIFO depends on the size of FIFO in each PE. In high load, some FIFOs may overflow and to prevent this, incoming traffic should be accepted with a slower rate. As will be explained in CCU design section below, almost full signals in each buffer in PEs goes high when one of the FIFOs reaches the predefined threshold level. Each CR is connected to a neighbour PE and hence if a FIFO in any PE asserts such almost full signel to the CCU, input traffic rate will be reduced. Therefore, the load of neighbouring CR will drop until the FIFO that has reached to its threshold in the PE deasserts almost full signal again

We utilized a ratio of (CR FIFO size/PE FIFO size) which ensures that no CR will reach its full capacity.

Considering the worst case scenario illustrated in Figure 4-13, assume that CR is highly loaded but PE is not. In other words, all incoming traffic is directed to one CR and PE. Since PE is not highly loaded, the other input side of the PE should be idle. Also assume that CR FIFO size is *m*, PE FIFO size is *n* and the number of FIFOs in a CR is S . After t cycles, the examined FIFO in the CR will have load of $t-\frac{t}{S}$. At this cycle, the working FIFO in PE will have a load of $t-\frac{t}{2}$ since PE is designed (due to other reasons) in such a way that it produces one output every two cycle. Then free capactiy in PE should be lower then CR, i.e.,

$$m-(t-\frac{t}{S}) > n-(t-\frac{t}{2})$$

Since in the 8x8 torus architecture, $S = 17$. If t is choosen as *m*, using the above inequality

$$\frac{m}{n} > \frac{34}{19} \cong 1.78$$

When *m* is at least 1.78 x *n*, the free area in a CR FIFO will always be lower than PE FIFO. Because of having a CCU which regulates input traffic by controlling PE FIFO, CR FIFOs will never overflow.

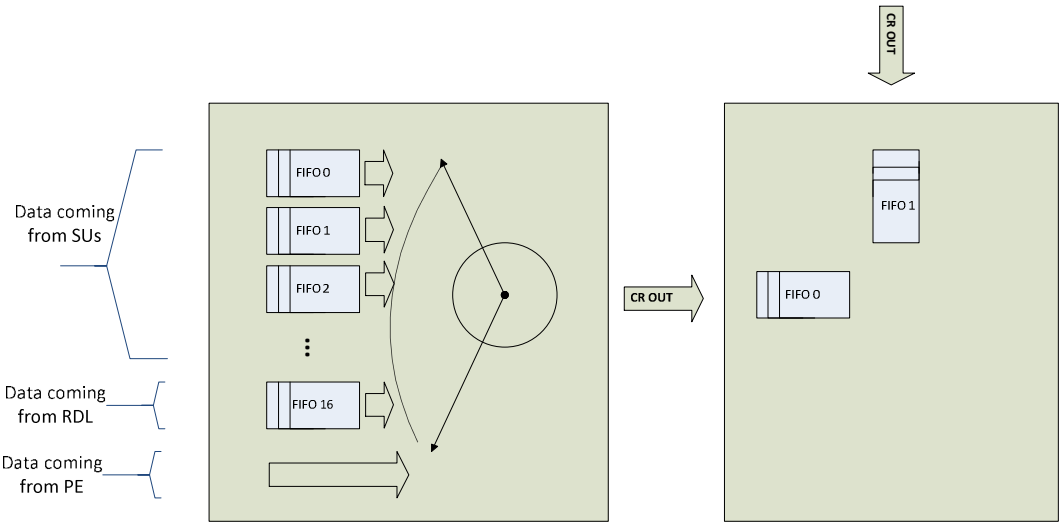

Figure 4-13 Block Diagram of Transition from CR to PE

In and out signals of CR are described in Table 4-5.

Table 4-5 CR Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
rst	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the CR module into a state that is ready to receive data
clk	In	1 bit	The clock signal is used in order to synchronize CR with other modules
data_av_in_X	In	1 bit	The available signal of data incoming to CR
data_in_X	In	49 bits	The data incoming to CR
data_av_out_X_SU	Out	1 bit	The available signal of data outgoing from CR X
data _out_X_SU	Out	49 bits	The data outgoing from CR X

The schematic view of CR is given in Figure 4-14.

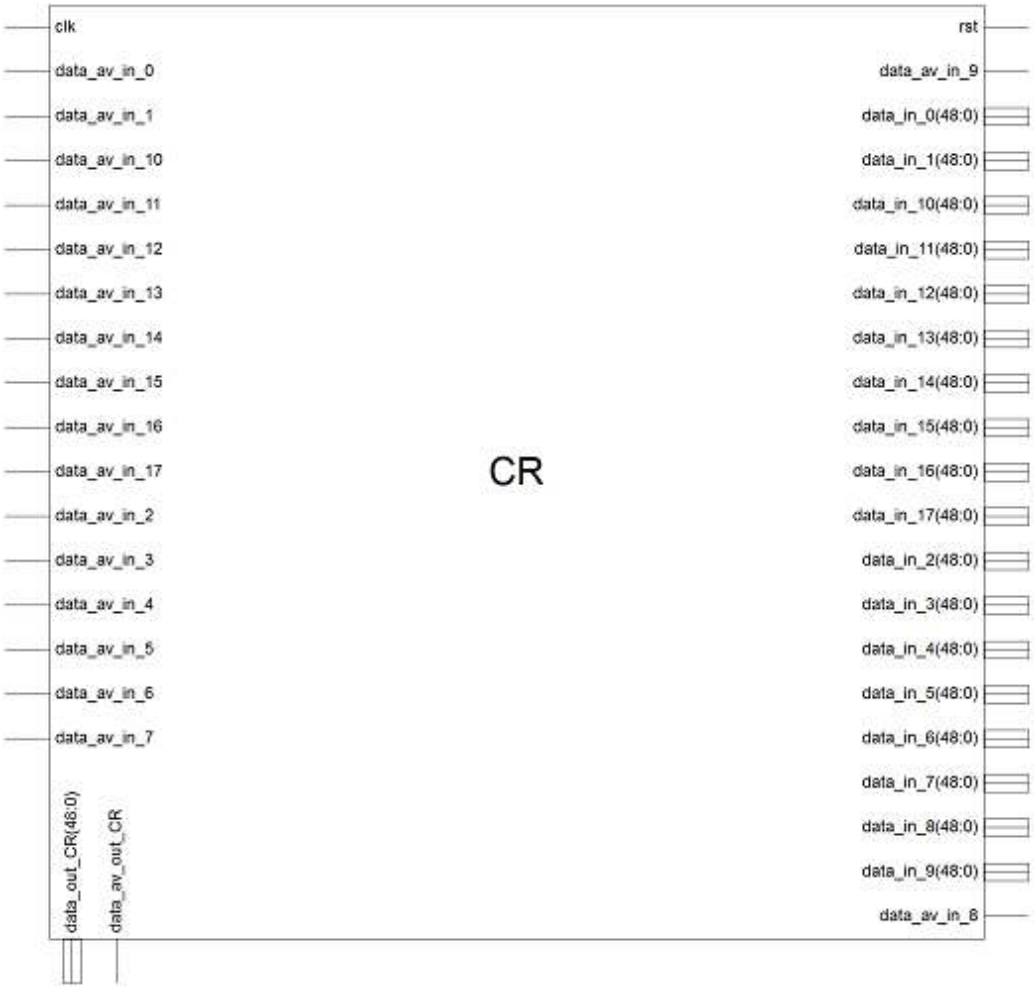

Figure 4-14 Contention Resolver Input and Output Signals

4.3.2 Simulation

SCENARIO 1

Aim: To show and verify that CR outputs incoming data according to round robin scheduling.

Test Code:

```
wait for 100 ns;
rst <= '1' ;
wait for clk_period*3;
rst <= '0' ;
wait for clk_period*10;
data_in_3 <= '1' & x"fffffffffff";
data_av_in_3 <= '1';
wait for clk_period;
data_av_in_3 <= '0';
data_in_7 <=  '1' & x"eeeeeeeeeee";
data_av_in_7 <= '1';
wait for clk_period;
data_av_in_7 <= '0';
data_in_1 <= '1' & x"ccccccccccc";
data_av_in_1 <= '1';
data_av_in_8 <= '1';
data_in_8 <= '1' & x"aaaaaaaaaaa";
wait for clk_period;
data_av_in_1 <= '0';
data_av_in_8 <= '0';
wait for clk_period;
data_av_in_0 <= '1';
data_av_in_1 <= '1';
data_av_in_2 <= '1';
data_av_in_3 <= '1';
data_av_in_4 <= '1';
```

```
data_av_in_5 <= '1';
data_av_in_6 <= '1';
data_av_in_7 <= '1';
data_av_in_8 <= '1';
data_av_in_9 <= '1';
data_av_in_10 <= '1';
data_av_in_11 <= '1';
data_av_in_12 <= '1';
data_av_in_13 <= '1';
data_av_in_14 <= '1';
data_av_in_15 <= '1';
data_av_in_16 <= '1';
data_av_in_17 <= '1';
data_in_0 <= '1' & x"000000000000";
data_in_1 <= '1' & x"111111111111";
data_in_2 <= '1' & x"222222222222";
data_in_3 <= '1' & x"333333333333";
data_in_4 <= '1' & x"444444444444";
data_in_5 <='1' &  x"555555555555";
data_in_6 <= '1' & x"666666666666";
data_in_7 <= '1' & x"777777777777";
data_in_8 <='1' &  x"888888888888";
data_in_9 <= '1' & x"999999999999";
data_in_10 <= '1' & x"101010101010";
data_in_11 <= '1' & x"111111111111";
data_in_12 <= '1' & x"121212121212";
data_in_13 <='1' &  x"131313131313";
data_in_14 <= '1' & x"141414141414";
data_in_15 <= '1' & x"151515151515";
data_in_16 <='1' &  x"161616161616";
data_in_17 <='1' &  x"171717171717";
```

wait for clk_period;

data_av_in_0 <= '0';

data_av_in_1 <= '0';

data_av_in_2 <= '0';

data_av_in_3 <= '0';

data_av_in_4 <= '0';

data_av_in_5 <= '0';

data_av_in_6 <= '0';

data_av_in_7 <= '0';

data_av_in_8 <= '0';

data_av_in_9 <= '0';

data_av_in_10 <= '0';

data_av_in_11 <= '0';

data_av_in_12 <= '0';

data_av_in_13 <= '0';

data_av_in_14 <= '0';

data_av_in_15 <= '0';

data_av_in_16 <= '0';

data_av_in_17 <= '0';

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, all of the CR input ports have SAFIL or SAFIL Update Frame data. In this scenario, data on the output ports should appear with respect to round robin scheduling except for high priority input port 16.

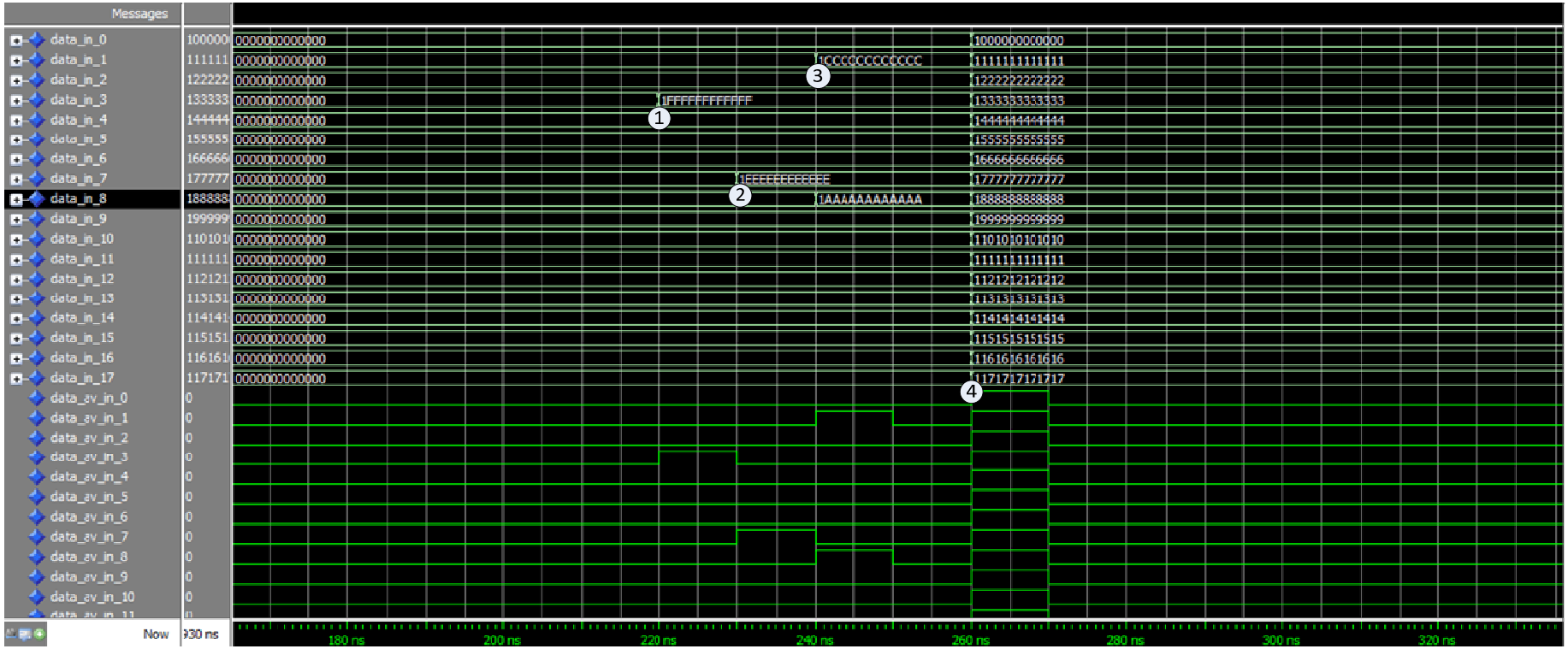

Figure 4-15 Simulation Results for CR in Scenario 1

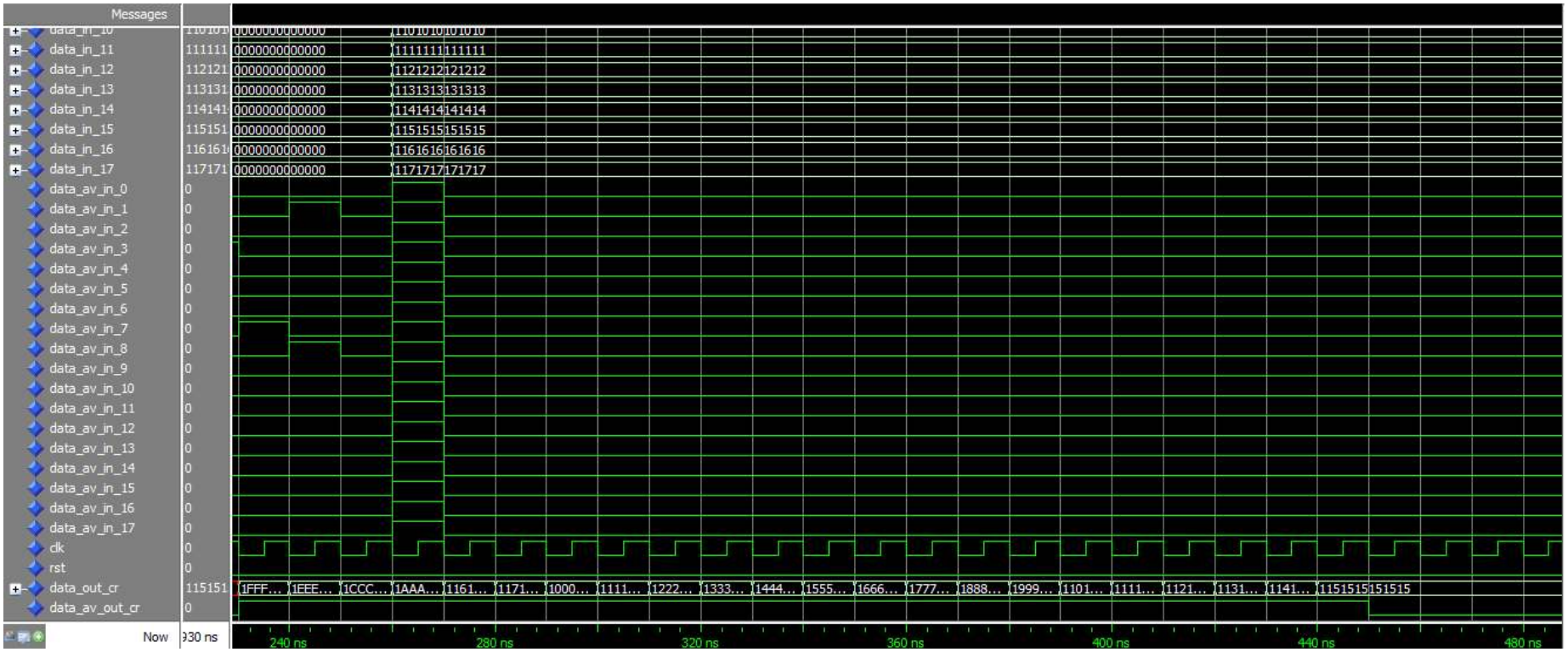

Figure 4-16 Simulation Results for CR in Scenario 1 (continued)

The result of the simulation is given in **Figure 4-15** and **Figure 4-16**. In this scenario, *data_av_in_x* appears at labels 1, 2, 3 and 4. Then, *data_out_cr* appears is generated by the Round Robin Scheduler in order.

4.4 CONGESTION CONTROL UNIT

This unit is designed and used to prevent possible FIFO overflow in any PE.

4.4.1 Design

The input ports of the Congestion Control Unit (CCU) are connected to all of the FIFOs in each PE.

The schematic view of CCU is given in Figure 4-17.

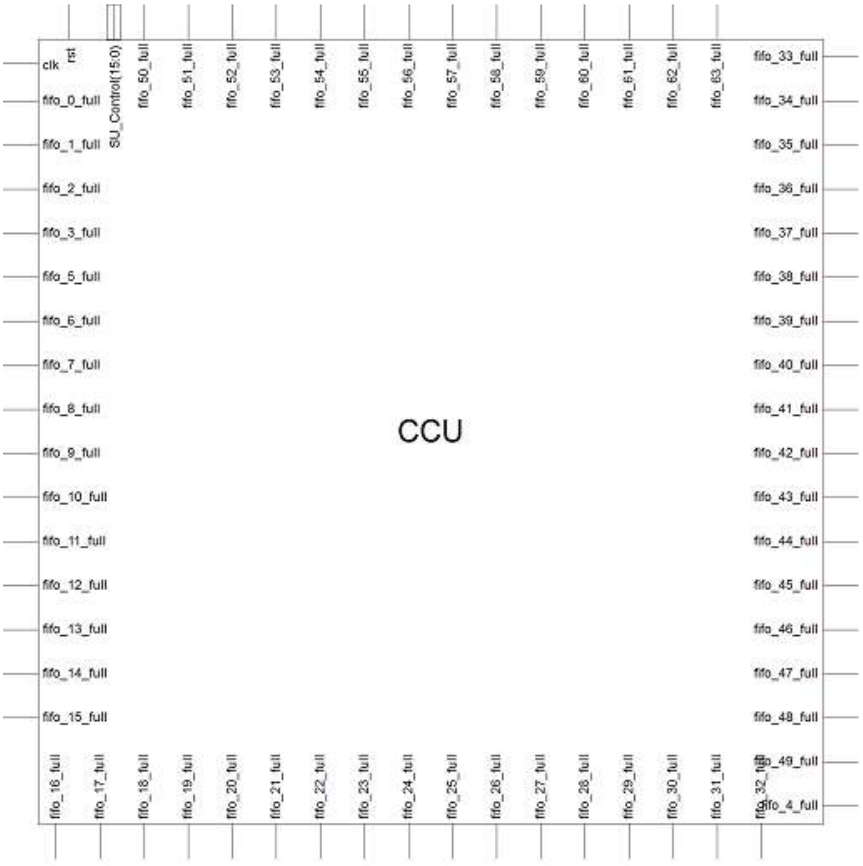

Figure 4-17 Congestion Controller Unit Input and Output Signals

In and out signals of CCU are described in Table 4-6.

Table 4-6 CCU Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
rst	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the CCU module into a state that is ready to receive data
clk	In	1 bit	The clock signal is used in order to synchronize CCU with other modules
fifo_X_full	In	1 bit	The signal indicating that relevant FIFO is almost full
SU_Control	Out	16 bits	The output signal showing that how many incoming traffic port will be enabled

4.4.2 Simulation

SCENARIO 1

Aim: To show and verify that CCU works with additive increase, multiplicative decrease strategy.

Test Code:

```
wait for 100 ns;
rst <= '1' ;
wait for clk_period*3;
rst <= '0' ;
wait for clk_period*10;
fifo_0_full<='1';
wait for clk_period;
fifo_14_full<='1';
wait for clk_period;
fifo_35_full<='1';
wait for clk_period;
fifo_0_full<='0';
fifo_14_full<='0';
fifo_35_full<='0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, some of the FIFOs assert full messages. In this scenario, when fifo full signals appear at the input port, SU control output port changes according to additive increase multiplicative decrease strategy. When there are no full messages, output should be hex "FFFF" while the output changes to hex "FFOO" if one of the FIFOs sends full.

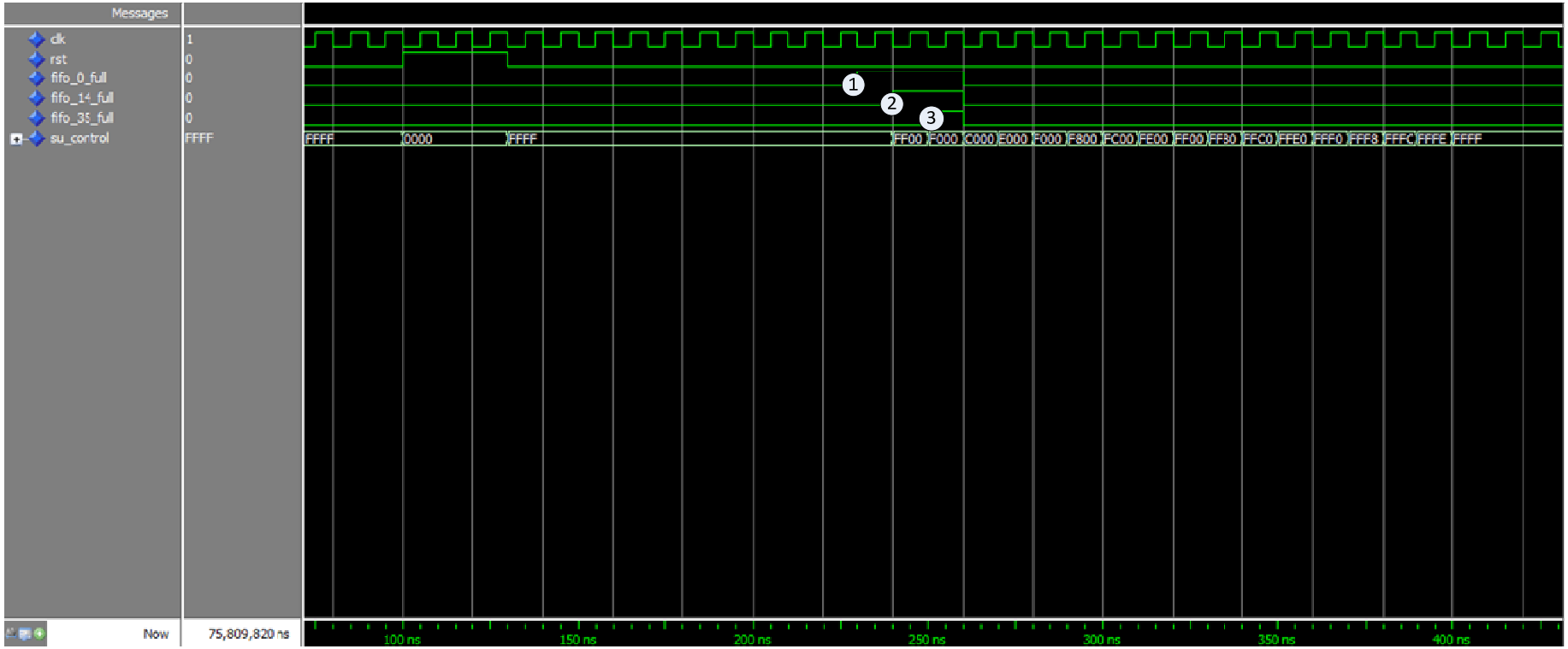

Figure 4-18 Simulation Results for CCU in Scenario 1

The result of the simulation is given in **Figure 4-18**. In this scenario, *fifo_x_full* signals appear at labels 1, 2 and 3. Then *su_control* output changes according to FIFO full alert signals.

4.5 RAM DATA LOADER

This unit is designed and used for loading and updating the contents of the Block RAM's located inside each PE. To achieve this, SAFIL Update Frame is constructed by this unit.

4.5.1 Design

Each Block RAM has a width of 32 bits and a depth of $2^{13} = 8192$ bits, making the total memory size for one PE 8k x 32 = 256 kb. Since the whole system consists of 64 PEs, the total memory required is 64 x 256 kb = 16,777,216 mb $\cong$ 2 MB.

RAM Data Loader (RDL) unit is connected to all Contention Resolver's located on the northern side. The input port has 52 bits width. The first 32 bits holds BRAM contents, the next 13 bits holds the address of the relevant BRAM and the following 6 bits holds PE ID that will be modified while the last bit indicates the Update operation. If Block RAM Data Load or Update Operation is started, this bit should be '1'; otherwise '0'.

The schematic view of RDL is given in Figure 4-19.

In and out signals of RDL are described in Table 4-7.

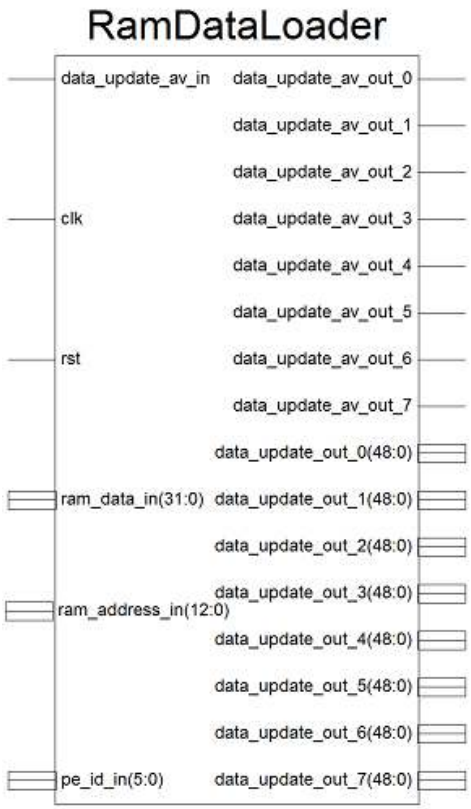

Figure 4-19 RDL Unit Input and Output Signals

Table 4-7 RDL Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
rst	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the RDL module into a state that is ready to receive data
clk	In	1 bit	The clock signal is used in order to synchronize RDL with other modules
data_update_av_in	In	1 bit	The available signal of data incoming to RDL
ram_data_in	In	32 bits	RAM content to be stored
ram_address_in	In	12 bits	RAM address that will be updated
pe_id_in	In	6 bits	The processing element ID whose RAM content will be updated
data_update_av_out	Out	1 bit	The available signal of data incoming to RDL
data_update _out	Out	49 bits	SAFIL Update Frame constructed

4.5.2 Simulation

SCENARIO 1

Aim: To show and verify that RDL constructs SAFIL Update Frame and presents this data on its output.

Test Code:

wait for 20 ns;

rst<='1';

wait for 100 ns;

rst <='0';

ram_data_in <= x"ffffffff";

ram_address_in <= "1010101010101";

pe_id_in <= "111010";

data_update_av_in <= '1';

wait for clk_period;

data_update_av_in <= '0';

Pre-Statement: Initially, reset input is set to "high" for 100 ns. After reset state, RAM update data signals appear at input ports. In this scenario, SAFIL Update Frame is constructed and appears at the output port.

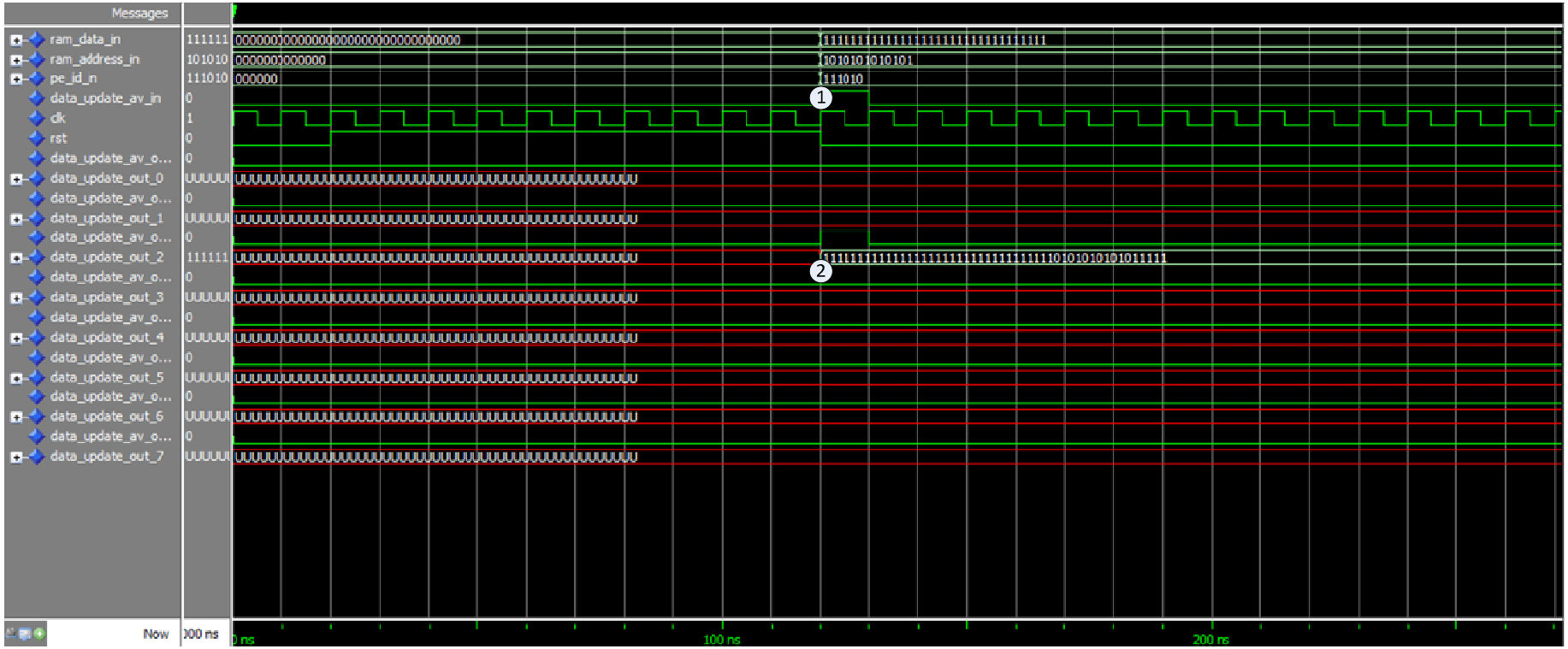

Figure 4-20 Simulation Results for RDL in Scenario 1

The result of the simulation is given in **Figure 4-20**. In this scenario, *data_update_av_in* signal appears at label 1. Then *data_update_out_2* output is activated at label 2.

4.6 SYSTEM INTEGRATION

In this present subsection, the blocks designed in the preceding subsections are integrated into a whole system to fulfill the following requirements:

- IP addresses that are to be searched should be admitted to the system via SUs at any time.
- Search keys should be mapped to correct CR and PE using initial partitioning.
- The incoming SAFIL or SAFIL update frames should to be buffered in CRs first.
- At any time SAFIL frames and SAFIL update frames, if exist, should be admitted to the system simultaneously.
- The latency encountered at each PE should be as small as possible.
- The system should be capable of initializing the RAM contents or updating them using RDL.
- The search results (i.e. port number) should be observed at the backplane.
- The traffic load should be regulated via CCU.

4.6.1 Design

Overall system is composed of the following modules:

- 8 x 8 = 64 PEs
- 8 + 8 = 16 SUs
- 8 + 8 = 16 CRs
- 1 CCU
- 1 RDL

As will be explained at the end of this section, the available resources on the currently selected FPGA target board allows an 8 x 8 SAFIL system (illustrated in) as the largest that can be implemented for the time being.

SU is connected to CRs. Incoming traffic arrives at the input ports of the SU, in which SAFIL frames are constructed and mapped to corresponding CR. These search keys are then directed to the boundary.

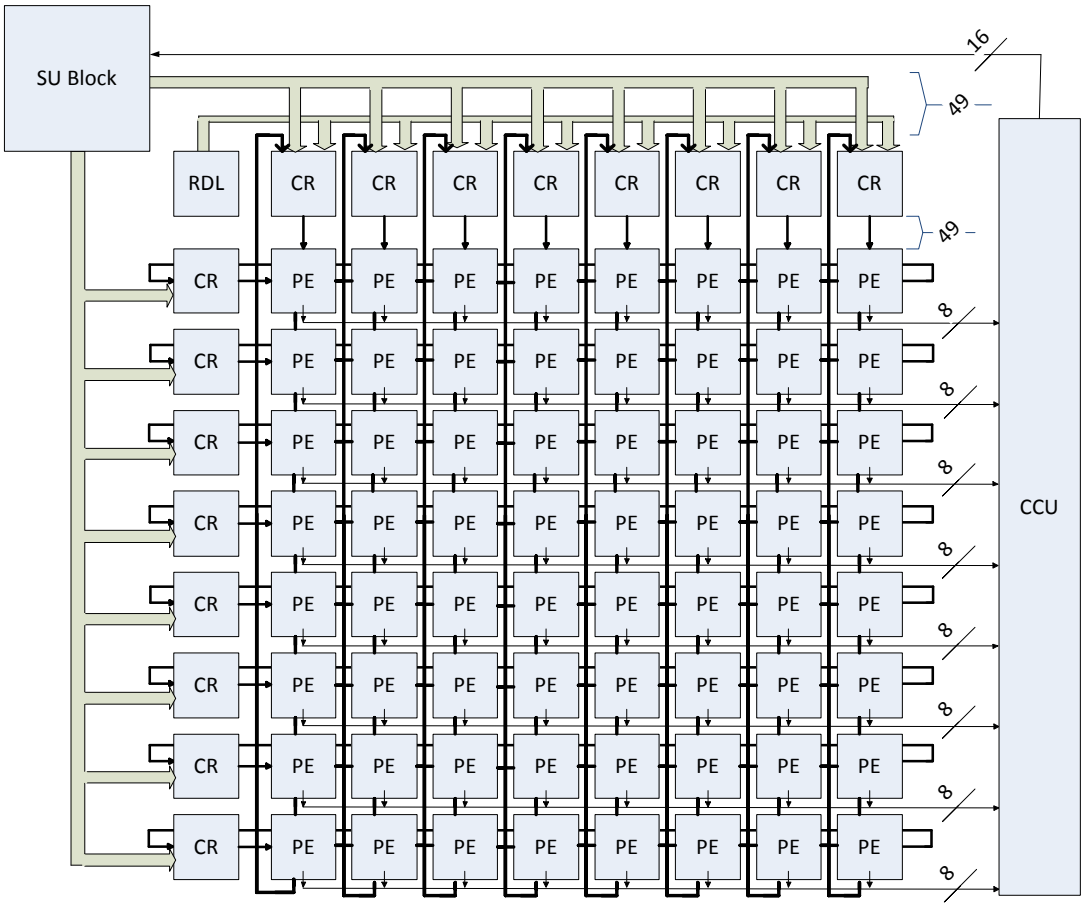

Figure 4-21 8x8 SAFIL System

Data arrive to each PE either from west or north side. In each PE, using the corresponding bit in the search key the incoming data is modified and directed to either eastern or southern neighbours. Search operation terminates when a null pointer is reached.

RAM contents of the system can be initially loaded and updated via our RDL unit. This module is connected to CRs located at the northern side. The update packets propagate through the corresponding column until the PE to be updated is reached.

All PEs are connected to CCU with a control line of one bit. When one of the PEs becomes almost full, this bit goes high and CCU regulates the incoming traffic using "multiplicative decrease additive increase" strategy.

The active signal duration between each module is one clock cycle. As shown in Figure 4-22, the searched IP address is admitted into the SU at a rising edge of the clock. With the falling edge of the clock, SAFIL frame is constructed and mapped to the corresponding CR. At each falling edge of the clock, PE accepts an incoming frame. In each PE, the latency between data arrival and departure processes is 2 clock cycles.

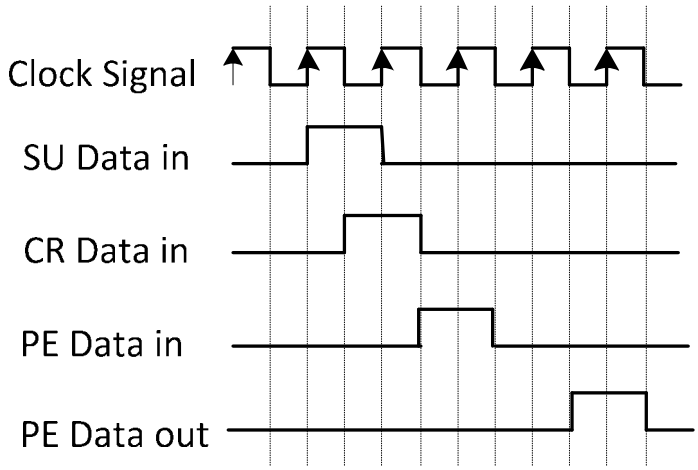

Figure 4-22 Timing Diagram for the Whole System

The schematic view of the system is given in **Figure 4-23**.

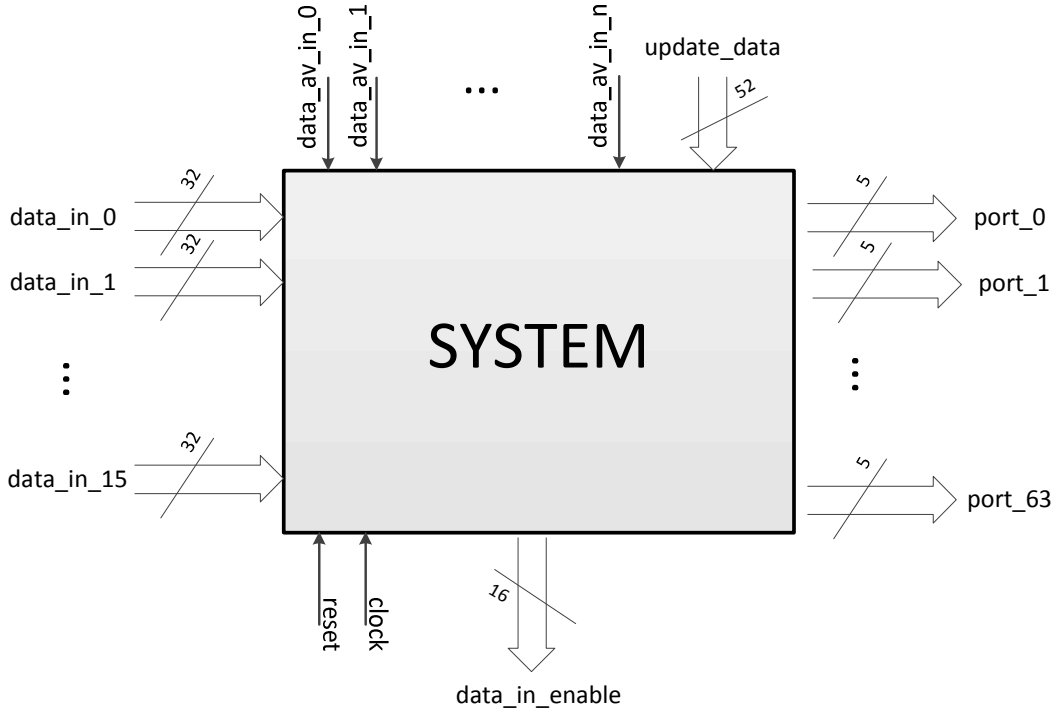

Figure 4-23 System Input and Output Signals

In and out signals of the system are described in Table 4-8.

Table 4-8 The System Signal Descriptions

Signal Name	Signal Type	Signal Length	Signal Description
reset	In	1 bit	Reset signal is used in order to set all configuration registers to zero (low) and to bring the system into a state that is ready to receive and send data
clock	In	1 bit	The clock signal is used in order to synchronize system with modules inside it.
data_ in_X	In	32 bit	The IP address that will be searched.
data_ av_in_X	In	1 bit	The available signal of data incoming to the system
port_X	Out	5 bits	The port number result of the searched IP address
data_ in_enable	Out	16 bits	The signal that indicates maximum allowable search is

The system is designed and implemented in Xilinx ISE Design Suite 12.4 and 13.3 NT 64 release version. VHDL is used to implement all the modules and the whole

system. In selecting a suitable Xilinx device, constraints such as the number of I/O banks, the total capacity of Block RAMs and distributed RAMs, etc. were taken into consideration. Table 4-9 gives the design requirements for the whole system. As was explained in section 4.1 and 4.3, FIFOs used in CRs and PEs are implemeted using distributed RAM and RAMs in PEs are implemented using Block RAMs.

Table 4-9 Design Requirements

Type	The Source of Requirement	Requirement Size	
I/O Pin	*data_in_X*	32 bit x 16	512
I/O Pin	*data_av_in_X*	1 bit x 16	16
I/O Pin	*port_X*	5 bit x 64	320
I/O Pin	*data_in_enable*	16 bit	16
Total Capacity of Block RAM Blocks	Block RAMs in PEs	256 Kbit x 64	16384 Kbit
Total Capacity of Distributed RAM Blocks	FIFOs inside PEs and CRs	(16 Kbit x 49) + (64 Kbit x 49)	3920 Kbit

Table 4-10 gives device attributes for some Xilinx family members.

Table 4-10 Attributes of Some Xilinx Family Members

Family	Device	Max I/O Size	Block RAM Blocks	Max Distributed RAM
Virtex-5	XC5VLX330	1200	10368 Kbit	3420 Kbit
Virtex-5	XC5VLX330T	960	11664 Kbit	3420 Kbit
Virtex-5	XC5VSX240T	960	18576 Kbit	4200 Kbit
Virtex-6	XC6VLX550T	1200	22752 Kbit	6200 Kbit
Virtex-6	XC6VLX760	1200	25920 Kbit	8280 Kbit
Artix-7	XC7A350T	600	18540 Kbit	4638 Kbit
Kintex-7	XC7K480T	400	34380 Kbit	6788 Kbit
Virtex-7	XC7V2000T	1200	46512 Kbit	21550 Kbit

Comparing Table 4-9 and Table 4-10, we observe that there exist devices in Virtex-5, Virtex-6 or Virtex-7 families, which fulfills our requirements. Since the listed devices are the latest and the most advanced in their own categories, Block RAM capacity and number of I/O pins is the limiting constraints and hence a 8 x 8 SAFIL

implementation is the largest possible for the time being. For example, if one requires a 16 x 16 SAFIL System similar to the one proposed and simulated using Visual C++ in [6], the whole system cannot fit in any Virtex family device. Table 4-11 demonstrates that at most a 10 x 10 SAFIL system can fit in XC6VLX760 or XC7V2000T Xilinx device. A larger architecture can not be implemented in the current state of the art Xilinx devices using a single FPGA only. However, implementation of a larger architecture such as 16 x 16 is always possible using more than one Virtex devices.

Table 4-11 Feasibiliy of SAFIL Implementations on Xilinx Family Members

SAFIL System	Virtex Device	Max I/O Size	Block RAM Blocks	Max Distributed RAM
8 x 8	XC6VLX760	✓	✓	✓
8 x 8	XC7V2000T	✓	✓	✓
9 x 9	XC6VLX760	✓	✓	✓
9 x 9	XC7V2000T	✓	✓	✓
10 x 10	XC6VLX760	✓	✓	✓
10 x 10	XC7V2000T	✓	✓	✓
11 x 11	XC6VLX760	✗	✗	✓
11 x 11	XC7V2000T	✗	✓	✓
12 x 12	XC7V2000T	✗	✓	✓
13 x 13	XC7V2000T	✗	✓	✓
14 x 14	XC7V2000T	✗	✗	✓

4.6.2 Simulation

We performed overall system simulations using real life backbone IP packet traces from [23] and by constructing the corresponding routing tables using real life prefix length distributions [25]. In our simulations, the routing table was composed of 150K prefixes. Our incoming traffic traces are composed of 1200K IP packets.

First, the routing table was adapted to our system to be loaded via the RDL unit. To achieve this, a "Data Adapter" program is written in Microsoft Visual C#. Figure 4-24 shows the form view of the "Data Adapter".

Figure 4-24 Data Adapter

When "RAM DATA" button on the main form is pressed, the RAM contents of the whole system is adapted to SAFIL Update Frame type. This adopted adat can then be loaded via RDL input port into our system. To insert data into RDL, this program also creates "file_data.in" shown in Figure 4-25. To use this file as an input in our simulation, "textio" library has been utilized in Xilinx ISE platform.

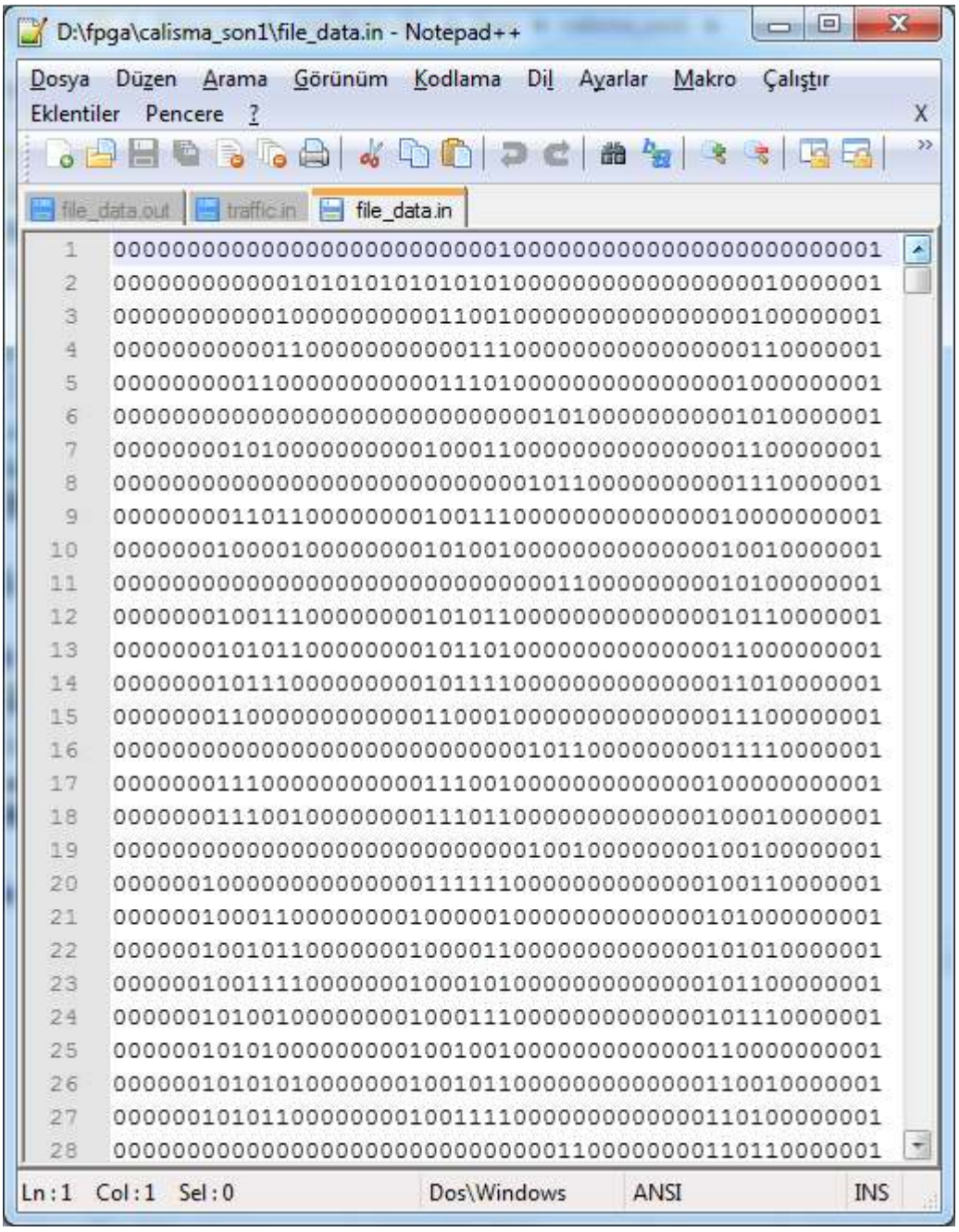

Figure 4-25 The Binary View of "file_data.in"

When "TRAFFIC DATA" button on the main form is pressed, the 32-bit IP addresses are converted into binary form. To insert this data into any SU, our program also creates "traffic.in" shown in **Figure 4-26**.

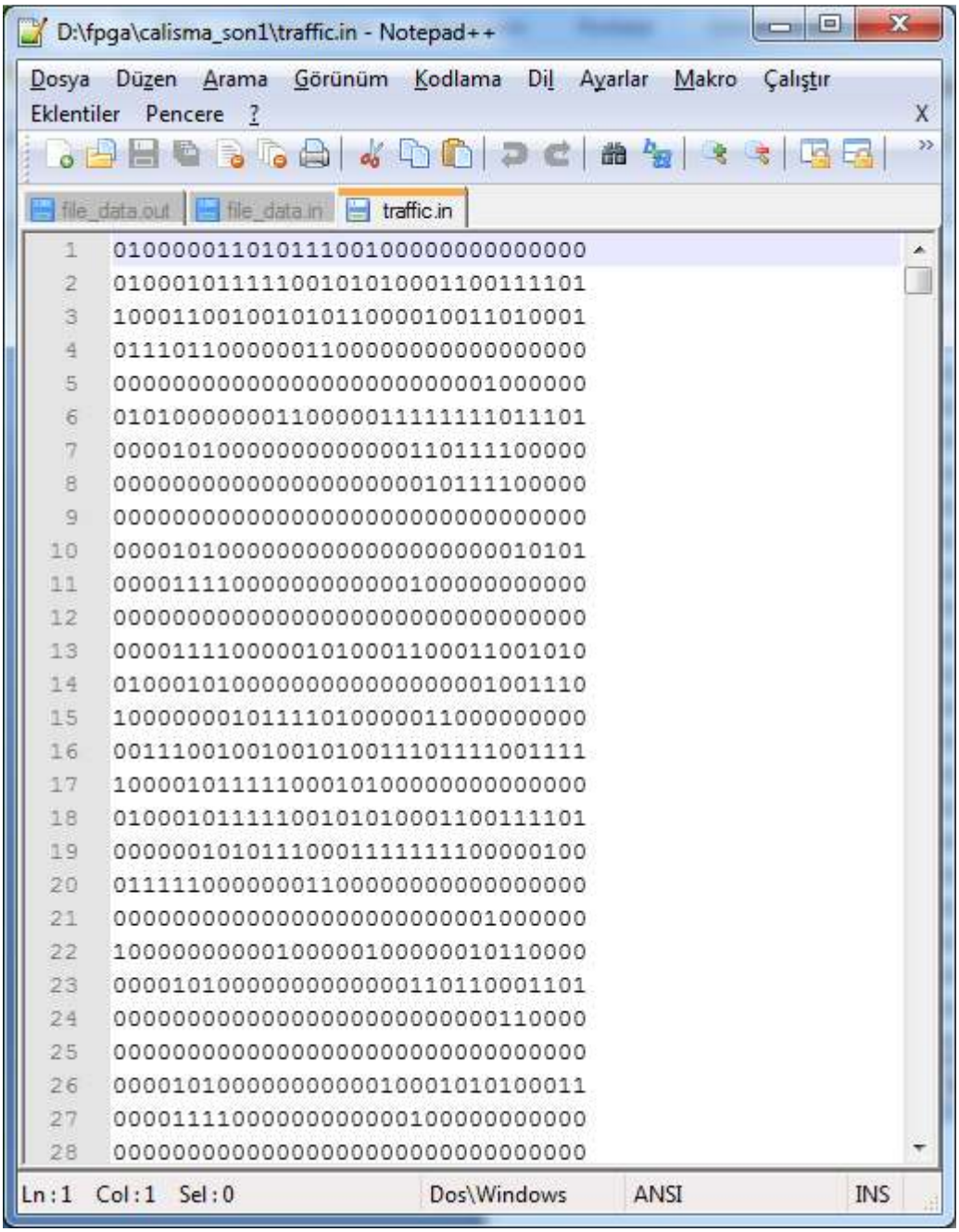

Figure 4-26 The Binary View of "traffic.in"

These files created are used as inputs in our simulations.

SCENARIO 1

Aim: To show and verify that RDL loads RAM contents into the whole system and one IP lookup operation produces the port result correctly in accordance with the trie mapped onto the system.

Test Code:

```
stim_proc: process
file input : TEXT open READ_MODE is "file_data.in";
variable input_line : LINE;
variable temp : std_logic_vector(51 downto 0) := x"0000000000000";
variable i_temp : integer ;
file input2 : TEXT open READ_MODE is "traffic_onepacket.in";
variable input_line2 : LINE;
variable temp2 : std_logic_vector(31 downto 0) := x"00000000";
variable i_temp2 : integer ;
begin
wait for clk_period*3;
rst <= '1';
load_done <= '0';
wait for clk_period*2;
rst <= '0';
wait for clk_period*2;
loop
exit when endfile(input);
readline(input, input_line);
read(input_line,temp);
update_data <= temp;
writeline(output, input_line);
wait for clk_period;
update_data<=x"0000000000000";
wait for clk_period;
end loop;
load_done <= '1';
```

```
update_data <= x"0000000000000";
wait for clk_period;
loop
exit when endfile(input2);
wait until falling_edge(clk);
if (su_enable_out(0)='1') then
 readline(input2, input_line2);
read(input_line2,temp2);
data_in_0 <= temp2;
data_av_in_0 <= '1';
else
data_av_in_0 <= '0';
end if;
end loop;
wait for clk_period;
data_av_in_0 <= '0';
```

Pre-Statement: Initially, reset input is set to "high" for 3 clock cycles. After reset state, RAM contents are loaded into the system. "*update_data*" is an input signal of the RDL unit. After load operation is completed, only one search key (32-bit IP address) is admitted into the Selector Unit 1. Figure 4-27 shows the simulation wave window for update data coming from "*file_data.in*" into RDL. After load operation finishes *"load_done"* signal goes high as shown in Figure 4-28. Then one search key is addmitted to the system for finding its corresponding port result. The IP address (1.37.59.207) to be searched starts with "*0000*" and is therefore directed to SU # 0. The SAFIL frame travels along the structure using the following 0's or 1's until null pointer is reached. The last PE produces the matched port output.

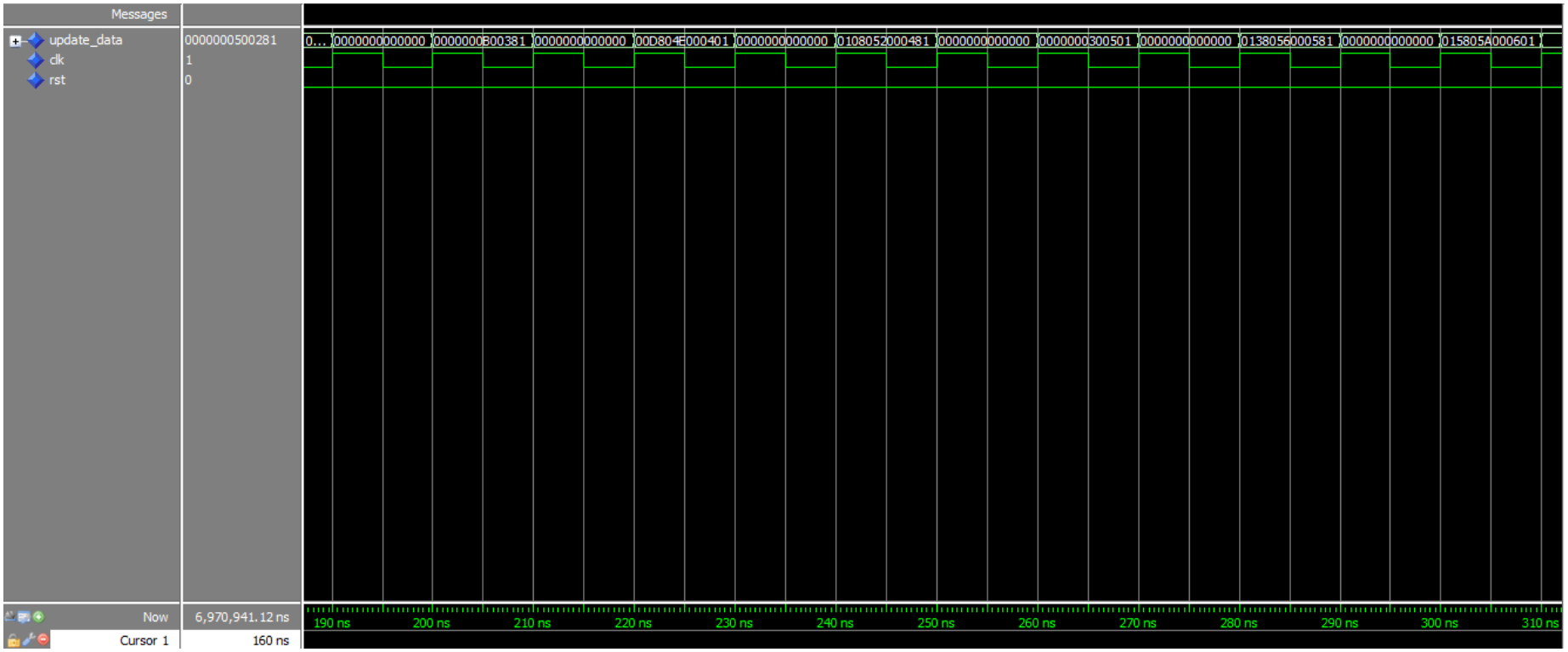

Figure 4-27 Simulation Results for the System in Scenario 1

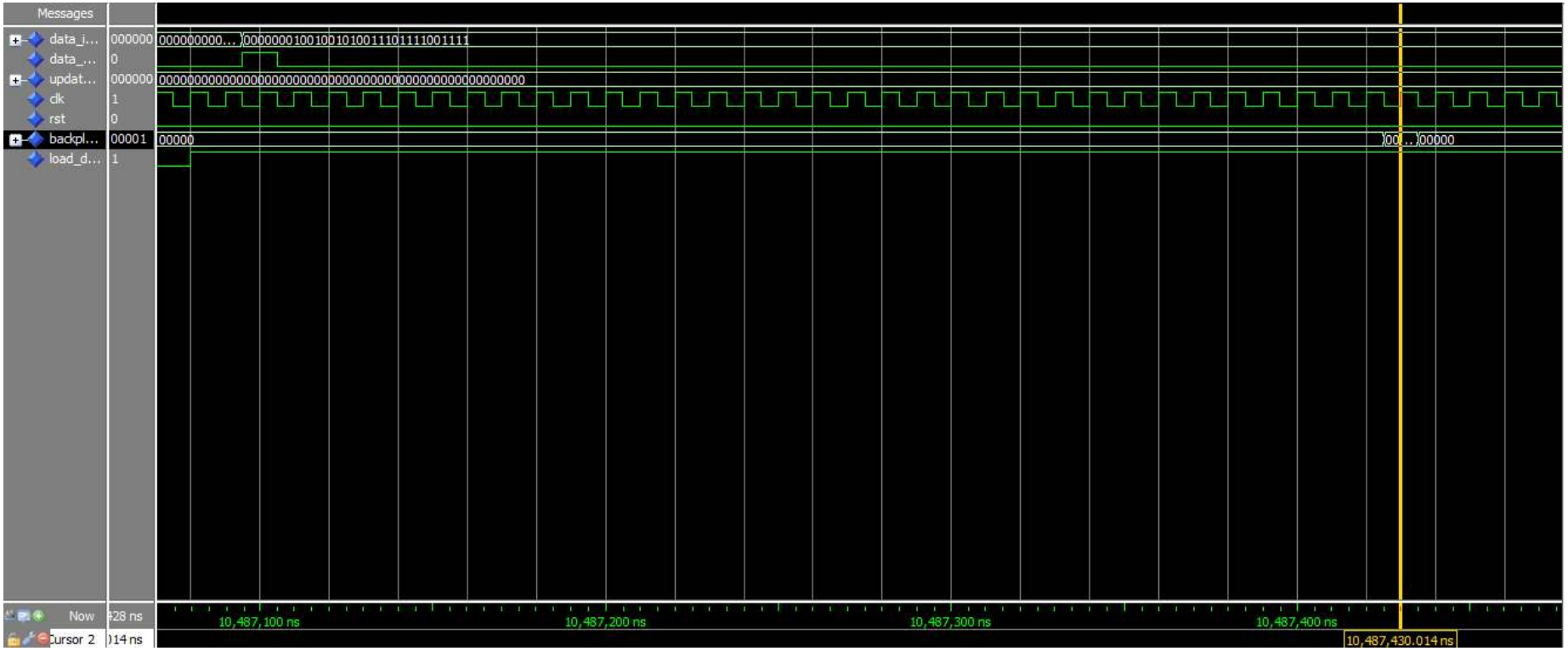

Figure 4-28 Simulation Results for the System in Scenario 1 (continued)

SCENARIO 2

Aim: To show and verify that IP lookup operation produces corresponding the port results correctly when 1200K IP packet traces are admitted into the system.

Pre-Statement: In this scenario, incoming traffic is generated by using real life backbone IP packet traces. The traces are composed of 1200K IP packets. Trace-1, which was obtained using [25], is admitted to the system first. Then, trace-2, which was derived from trace-1 is admitted to the system to make the input traffic almost even with respect to prefix distribution. Each packet's arrival time to the system is saved in a text file. Each port result and time of assertion of this result are also saved in another text file. Using this data that contains input arrival time and output assertion time for each packet, total average latency can be calculated.

Figure 4-29 gives the prefix distribution in trace-1. The majority of the incoming traffic is in the form of prefix 0.

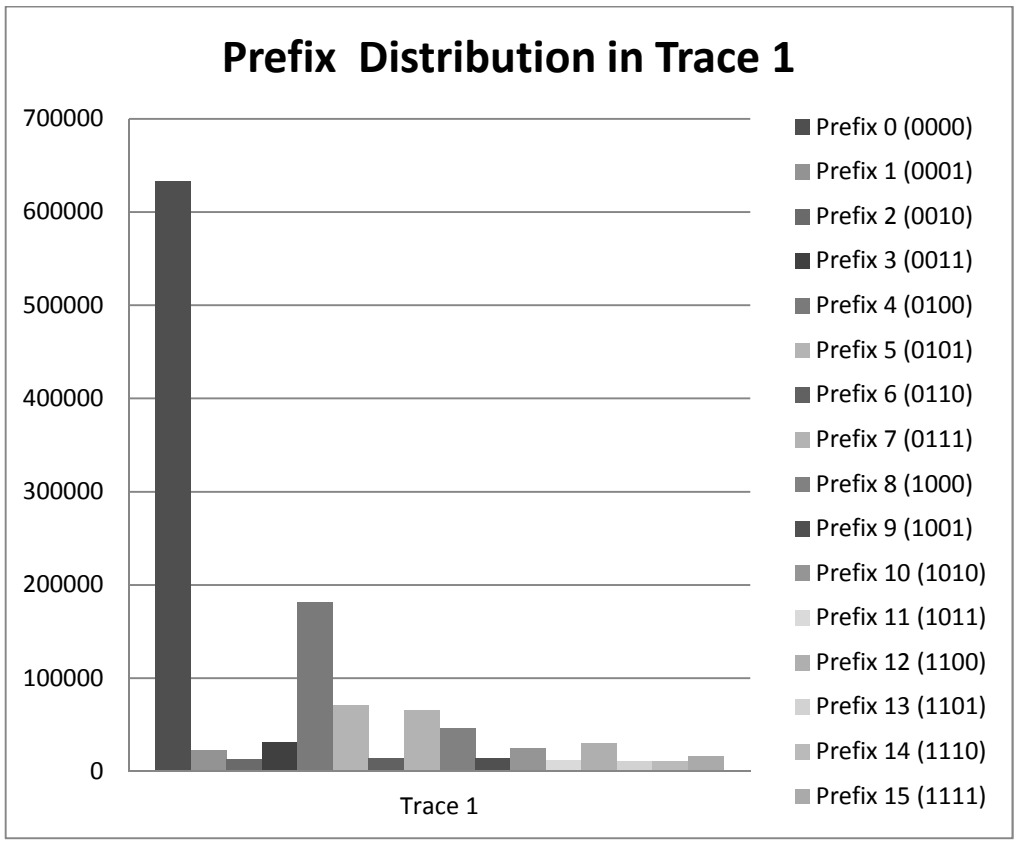

Figure 4-29 Trace 1 Distribution

After trace-1 was admitted into the system, the port results and corresponding timing data were written into the files. Figure 4-30 gives the corresponding port results for trace 1. Figure 4-32 gives part of the simulation waveform for the trace-1.

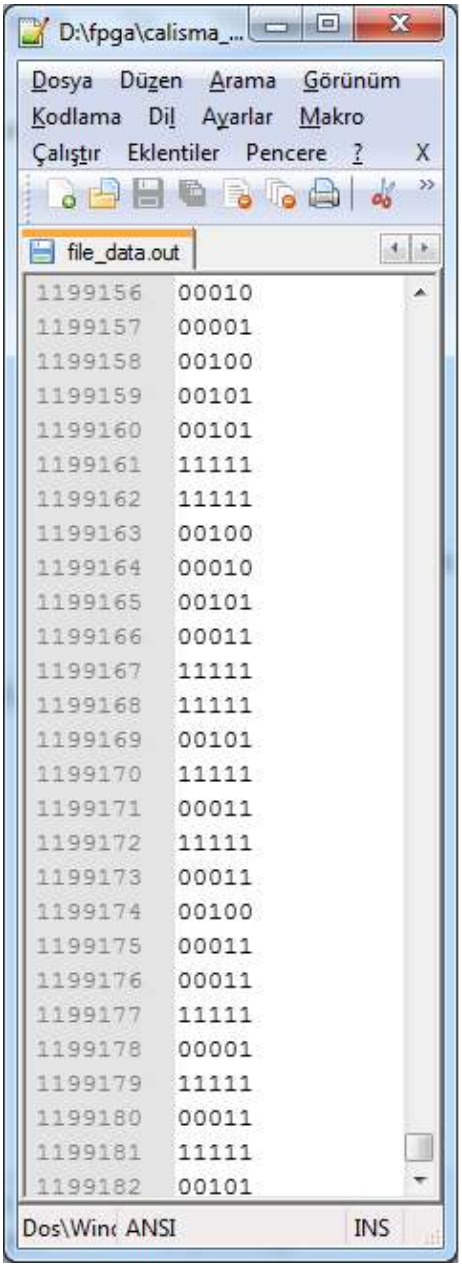

Figure 4-30 Port Results of Trace 1

Figure 4-31 gives the prefix distribution in trace-2. The incoming traffic is almost equally distributed with respect to the initial four bits.

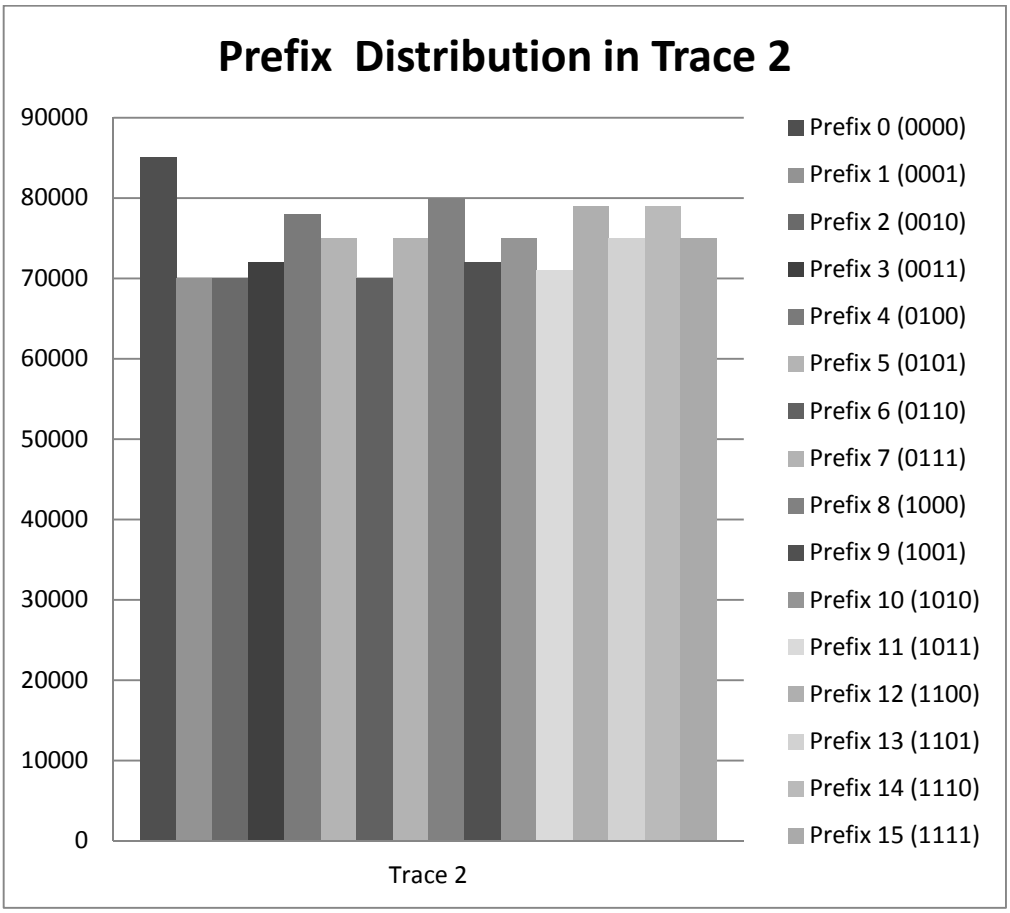

Figure 4-31 Trace 2 Distribution

After trace-2 was admitted into the system, the port results and corresponding timing data were written into the files. Figure 4-33 gives part of this simulation waveform for the trace-2.

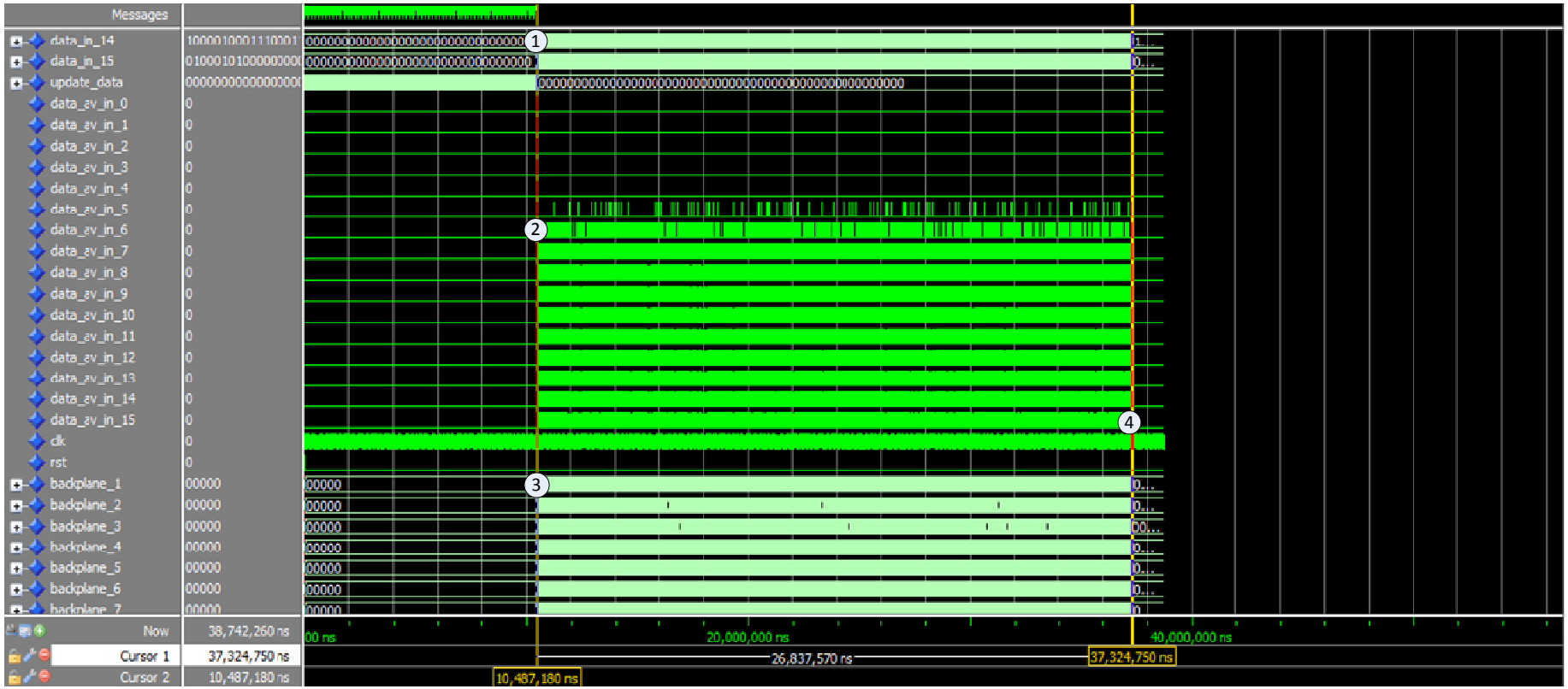

Figure 4-32 Trace 1 Simulation Results

In this scenario, incoming traffic starts with labels 1 and 2, ends with label 4. The backplane results are started to be observed at label 3.

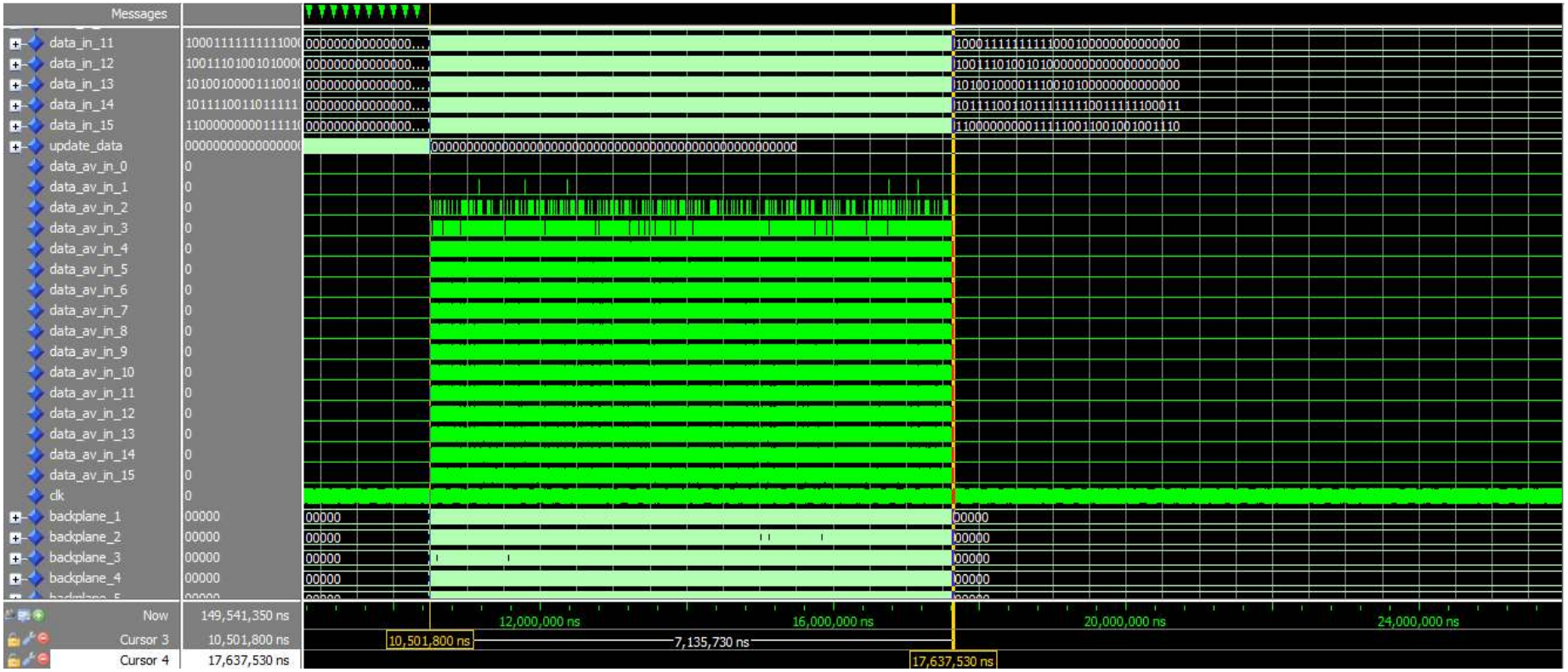

Figure 4-33 Trace 2 Simulation Results

4.7 PERFORMANCE EVALUATION

Our simulation is based on the experiment which admits the 1200K incoming IP packet stream into the system.

4.7.1 Speedup and Throughput

The first experiment was carried out using trace-1 shown in Figure 4-29. The total 1200K incoming IP addresses are searched through the 8 x 8 SAFIL System and the mathced port results are observed in the backplane during 26,863,800 ns (clock cycle was choosen as 10 ns). Therefore,

$$Speed\ up = \frac{1{,}200{,}000\,packets}{26{,}837{,}570\,ns} = \frac{1200000}{2683757} \approx 0{,}447\,packets/cycle$$

With minimum size (40-byte) IP packets, $\frac{0{,}447 x 40 byte}{10 ns} = 1.788\,\text{GBps} = 14.3\ \text{Gbps}$ throughtput has been achieved.

The second experiment was carried out using trace-2 shown in Figure 4-31. For this case, total time passed between the start of incoming IP address admission and final outgoing port result observation is found to be 7,135,730 ns. Hence

$$Speed\ up = \frac{1{,}200{,}000\,packets}{7{,}135{,}730\,ns} = \frac{1200000}{713573} \approx 1{,}681\,packets/cycle$$

With minimum size (40-byte) IP packets, $\frac{1{,}681 x 40 byte}{10 ns} = 6.724\,\text{Gbps} = 53.79\ \text{Gbps}$ throughtput has been achieved.

4.7.2 Latency

The average latency is also calculated. For this, all incoming packet arrival times and the time at which its corresponding port result appears at the backplane were saved and averaged over all packets.

For the first experiment using trace-1, average latency is calculated to be 97,57 clock cycles.

For the other experiment using trace-2, average latency is calculated to be 60,67 clock cycles.

These performance metrics are far below the values acheied in the ASIC architecture prposed in [6] since we have a much higher latency in each Processing Element (PE), a high latency I/O in the FPGA board and a limit in memory capacity and also on the synthesizable maximum clock speed. To achieve terabit performance IP lookup our FPGA architecture, there should be FPGA devices with higher operating frequencies and with more I/O pin capabilities. Use of techniques such as zero or one skip clustering and use of cache [6] should also be implemented to improve the throughput.

4.8 OPTIMIZATION

After the above initial simulations, some optimization steps have been carried out on the system in order to decrease the average latency and increase the speedup further. For this, minor modifications on PE, CR and SU blocks have been carried out as are explained below.

The first optimization is based on the threshold level used in PE FIFOs. This value plays an important role since it directly affects CCU in regulating the incoming traffic. When this value is low, the system operates relatively slowly. On the other hand selecting this value high may cause packet drops. Although CCU slows down the incoming traffic, there may still be packets waiting in CR and PE FIFOs, which will travel to other PEs that may have already reached the threshold level. Selection of this level affects latency and speedup directly. We performed some trials to choose the right FIFO threshold level (in percentage of the FIFO size) as shown in Table 4-12. The highlighted rows in Table 4-12 give the optimum threshold level.

Table 4-12 Effect of Threshold Level on Performance

Trace	Threshold Level	Speed up	Packet Drop Rate
Trace-1	% 75	0,490	% 4.5
Trace-1	% 60	0,455	% 0.6
Trace-1	% 50	0,447	% 0.07

Trace-1	% 40	0,441	% 0.06
Trace-1	% 34	0,438	% 0
Trace-2	% 75	2,178	% 2.8
Trace-2	% 60	2,166	% 0.7
Trace-2	% 58	2,154	% 0
Trace-2	% 50	1,681	% 0
Trace-2	% 40	1,672	% 0

The second optimization is based on CRs. While designing CRs in section 4.3.1, we already used FIFO modules to buffer the incoming data. When simulating the whole system, the buffer capacity plays an important role as was explained above in the first optimization. At this point, we removed all FIFOs inside the CRs and turned CR into a state machine. To implement CR without FIFOs, we also modified the SU to communicate in both directions.

The state diagram of our modified CR is given in Figure 4-34. In addition to the "data_av_in" signal, "data_need" signal has been added. When data is available in SU and CR needs data from SU, available data arrives to the input port of the CR in one clock cycle. Figure 4-35 gives the block diagram of the connection from one SU to one PE.

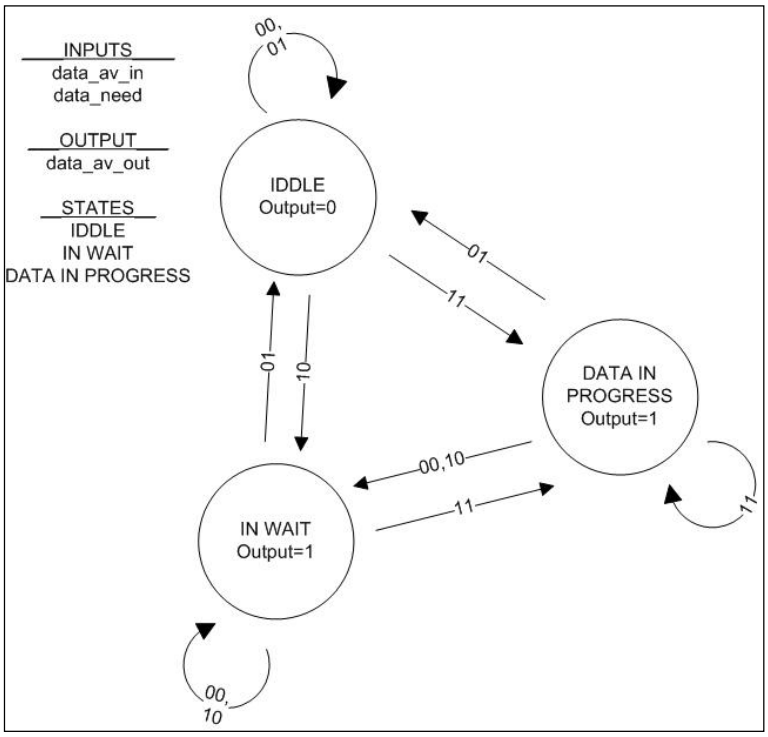

Figure 4-34 State Diagram for Our Modified CR

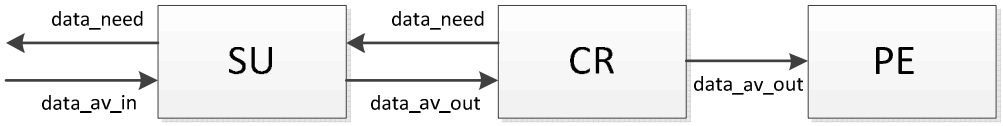

Figure 4-35 Block Diagram of the Modified System

When these modified modules were used in the system, two different traces were admitted into the system. Table 4-13 gives the effects of removing FIFOs from CRs on performance. When both Tables 4-12 and 4-13 are compared, speed up is decreased by approximately 10% in the case of 50% threshold level. However, latency is improved by approximately 40% when both results in Table 4-13 and Section 4.7.2 are compared in the case of 50% threshold level.

Table 4-13 Effects of Removing FIFOs from CRs on Performance

Trace	Threshold Level	Speed up	Latency	Packet Drop Rate
Trace-1	%50	0,411	45,12	% 0
Trace-2	%50	1,505	36,36	% 0

CHAPTER 5

CONCLUSION

In this thesis study, prototype development and verification for an IP lookup engine on FPGAs is carried out. The system is evaluated experimentally and the results are exhibited including detailed discussions.

We first focused on the feasibility of the FPGA implementation of the SRAM based pipelined architecture proposed earlier originally for ASIC. We made some minor modifications and improvements on the architecture for FPGA adaptation. We then carried out other optimizations to improve the speed up and latency. The proposed prototype achieves a sustained throughput of 57 Gbps in case of uniform traffic load without any packet drop. In our system design, we utilized an existing FPGA board to realize an 8x8 torus architecture . In this architecture maximum 2 MB entries of the router lookup table could have be stored in BRAMs. A larger lookup engine can not be implemented in the current state of the art XILINX devices using a single FPGA only. However, implementation of a larger engine such as 16 x 16 is always possible using more than one virtex devices.

As a future work, further study can be carried out for improving the performance with trying to reduce the latency in each PE with the latest Virtex devices. Using more than one Virtex device or latest devices coming in the future to support larger router lookup tables can be experimented. Moreover, some techniques to improve the performance for example cache using and zero skipping can be utilized in our architecture.

REFERENCES

[1] D. Pao, Z. Lu and Y.H. Poon, ”Bit-Shuffled Trie: IP Lookup with Multi-Level Index Tables” *In Proc. of IEEE International Conference on Communications (ICC'11)*, pp. 1-5, June 2011.

[2] K. Pagiamtzis and A. Sheikholeslami, ”Content-addressable memory (CAM) circuits and architectures: A tutorial and survey” *IEEE Journal of Solid-State Circuits*, vol. 41, no.3, pp. 712-727, March 2006.

[3] A.J. McAuley and P. Francis, ”*Fast routing table lookup using CAMs” In Proc. of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM’93)*, vol. 3, pp. 1382-1391, April 1993.

[4] F. Baboescu, D.M. Tullsen, G. Rosu and S. Singh, ”A tree based router search engine architecture with single port memories” *In Proc. of the 32nd Annual International Symposium on Computer Architecture (ISCA’05)*, pp. 123-133, June 2005.

[5] S. Kumar, M. Becchi, P. Crowley and J. Turner, ”Camp: fast and e_cient IP lookup architecture” *In Proc. of the 2nd Symposium on Architectures for Networking and Communications Systems (ANCS’06)*, pp. 51-60, December 2006.

[6] O. Erdem and C.F. Bazlamaçcı, ”Array design for trie-based IP lookup” *IEEE Communications Letters*, vol.14, no.8, pp. 773-775, August 2010.

[7] V. Srinivasan and G. Varghese, ”Fast address lookups using controlled prefix expansion” *ACM Transaction on Computer Systems*, vol. 17, no. 1, pp. 1-40, February 1999.

[8] H. Mohammadi, N. Yazdani, B. Robatmili and M. Nourani, ” HASIL: Hardware Assisted Software-based IP Lookup for Large Routing Tables” *In Proc. of the 11th IEEE International Conference (ICON 2003),*pp. 99-104, September 2003.

[9] D. R. Morrison, ”Patricia: practical algorithm to retrieve information coded in alphanumeric” *Journal ACM*, vol. 15, no.4, pp. 514-534, October 1968.

[10] H. Le, W. Jiang and V. K. Prasanna, ” A SRAM Based Architecture for Trie-based IP Lookup Using FPGA” *In Proc. of the 16th Annual International Symposium on Field- Programmable Custom Computing Machines (FCCM '08)*, pp. 33-42, April 2008.

[11] Y.H. E. Yang, O. Erdem and V. K. Prasanna, ”High performance IP lookup on FPGA with combined length infix pipelined search” *In Proc. of the 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*, pp. 77- 80, May 2011.

[12] O. Erdem, H. Le and V. K. Prasanna, ”Clustered hierarchical search structure for largescale packet classification on FPGA” *In Proc. of the 21st International Conference on Field Programmable Logic and Applications (FPL'11)*, pp. 201-206, September 2011.

[13] H. Le and V. K. Prasanna, ”Scalable high throughput and power effcient IP-lookup on FPGA” *In Proc. of 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'09)*, pp. 167-174, April 2009.

[14] H. Le, W. Jiang and V.K. Prasanna, ”Scalable high-throughput SRAM-based architecture for IP-lookup using FPGA*” In Proc. of International Conference on Field Programmable Logic and Applications (FPL'08)*, pp.137-142, September 2008.

[15] W. Jiang, Q. Wang and V.K. Prasanna, ”Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup” *In Proc. of the 27th Annual Joint 125 Conference of the IEEE Computer and Communications Societies (INFOCOM'08)*, pp. 2458-2466, April 2008.

[16] W. Jiang and V. K. Prasanna, ”A memory-balanced linear pipeline architecture for triebased IP lookup” *In Proc. of the 15th IEEE Hot Interconnects Symposium (HOTI'07)*, pp. 83-90, August 2007.

[17] W. Jiang and V.K. Prasanna, ”Multi-terabit IP lookup using parallel bidirectinal pipelines*” In Proc. of the 5th conference on Computing Frontiers (CF'08)*, pp. 241-250, May 2008.

[18] W. Jiang and V.K. Prasanna, ”Multi-way pipelining for power effcient IP lookup” *In Proc. of IEEE Global Communications Conference (GLOBECOM'08)*, pp. 1-5, December 2008.

[19] W. Jiang and V.K. Prasanna, ”Towards green routers: Depth bounded multi pipeline architecture for power effcient IP lookup” *In Proc. of the 27th IEEE*

International Performance Computing and Communications Conference (IPCCC'08), pp. 185-192, December 2008.

[20] M.J. Akhbarizadeh, M. Nourani and C.D. Cantrell, "Prefix segregation scheme for a TCAM based IP forwarding engine" *IEEE Micro*, vol. 25, pp. 48-63, no. 4, August 2005.

[21] M. J. Akhbarizadeh, M. Nourani, R. Panigrahy and S. Sharma. "A TCAM-based parallel architecture for high-speed packet forwarding" *IEEE Transaction on Computers*, vol. 56, no.1, pp. 58-72, January 2007.

[22] D. Lin, Y. Zhang, C. Hu, B. Liu, X. Zhang and D. Pao, "Route table partitioning and load balancing for parallel searching with TCAMs" *In Proc. of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pp. 1-10, March 2007.

[23] "AMPATH-I Traces" *http://pma.nlanr.net*, last visited date, 10/05/2010.

[24] "CACTI tool" *http://quid.hpl.hp.com:9081/cacti*, last visited date, 16/08/2011.

[25] "BGP Routing Table Analysis Reports" *http://bgp.potaroo.net*, last visited date, 16/08/2011.

[26] "High performance IP lookup engine with compact clustered trie search", *Computer Journal*, (2012), (in print), doi: 10.1093/comjnl/bxs008.

APPENDIX A

SOURCE CODE for FIFO IMPLEMENTATION

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:20:35 09/12/2010
-- Design Name:
-- Module Name:    fifo - arch
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity fifoCR is
        generic(
                Data_Bits: natural:=49;         -- number of data bits
                Address_Bits: natural:=12       -- number of address bits
                );
  Port    (
        clk                     :       in  std_logic;
       reset                    :       in  std_logic;
        fifo_rd                 :       in  std_logic;
        fifo_wr                 :       in  std_logic;
        fifo_wr_data            :       in  std_logic_vector (Data_Bits-1 downto 0);
        fifo_empty              :       out  std_logic;
        fifo_full               :       out  std_logic;
        fifo_rd_data            :       out  std_logic_vector (Data_Bits-1 downto 0)
        );
end fifoCR;
```

```
architecture arch of fifoCR is
        type register_type is array ((2**Address_Bits)-1 downto 0) of
        std_logic_vector(Data_Bits-1 downto 0);
        signal t_array_reg              : register_type;
        signal write_ptr_now  : std_logic_vector(Address_Bits-1 downto 0);
        signal write_ptr_next  : std_logic_vector(Address_Bits-1 downto 0);
        signal write_ptr_inc    : std_logic_vector(Address_Bits-1 downto 0);
        signal read_ptr_now   : std_logic_vector(Address_Bits-1 downto 0);
        signal read_ptr_next   : std_logic_vector(Address_Bits-1 downto 0);
        signal read_ptr_inc     : std_logic_vector(Address_Bits-1 downto 0);
        signal read_data_now : std_logic_vector(Data_Bits-1 downto 0);
        signal read_data_next : std_logic_vector(Data_Bits-1 downto 0);
        signal write_data       : std_logic_vector(Data_Bits-1 downto 0);
        signal fifo_operator    : std_logic_vector(1 downto 0);
        signal fifo_full_now   : std_logic;
        signal fifo_full_next   : std_logic;
        signal fifo_empty_now         : std_logic;
        signal fifo_empty_next         : std_logic;
        signal write_enable             : std_logic;
begin
        -- Clocking Process
        process(clk,reset)
        begin
                if (reset='1') then
                        t_array_reg <= (others =>(others=>'0'));
                        write_ptr_now <= (others => '0');
```

```
                read_ptr_now <= (others => '0');
                fifo_full_now <= '0';
                fifo_empty_now <= '1';
            elsif (clk'event and clk='1') then
                read_ptr_now <= read_ptr_next;
                read_data_now <= read_data_next;
                fifo_empty_now <= fifo_empty_next;
                fifo_full_now <= fifo_full_next;
                write_ptr_now <= write_ptr_next;
                if (write_enable='1') then
                t_array_reg(to_integer(unsigned(write_ptr_now)))        <=
write_data;
                end if;
            end if;
        end process;
        fifo_rd_data <= read_data_now;
        fifo_full <= fifo_full_now;
        fifo_empty <= fifo_empty_now;
        -- Inputs
        write_data <= fifo_wr_data;
        write_enable <= fifo_wr and (not fifo_full_now);
        -- This computes the next pointer values
        write_ptr_inc <= std_logic_vector(unsigned(write_ptr_now)+1);
        read_ptr_inc <= std_logic_vector(unsigned(read_ptr_now)+1);
        fifo_operator <= fifo_wr & fifo_rd;
```

```
        process(fifo_empty_next, read_ptr_next, t_array_reg,
                        fifo_empty_now,fifo_wr_data)
begin
read_data_next  <= t_array_reg(to_integer(unsigned(read_ptr_next)));
        if (fifo_empty_next = '1') then
                -- if fifo is empty output '0'
                read_data_next  <= (others => '0');
        else
                if (fifo_empty_now = '1') then
                -- If fifo empty next = 0 and fifo empty now = 1 then
                -- feed data straight through to output on this clock cycle
                        -- (First Word Fall Through)
                        read_data_next <= fifo_wr_data;
                end if;
        end if;
end process;
process (write_ptr_now, write_ptr_inc, read_ptr_now, read_ptr_inc,
        fifo_operator, fifo_empty_now, fifo_full_now)
begin
        write_ptr_next <= write_ptr_now;
        read_ptr_next <= read_ptr_now;
        fifo_full_next <= fifo_full_now;
        fifo_empty_next <= fifo_empty_now;
        case fifo_operator is
                when "00" => -- This means no operation
                when "01" => --read
```

```
                    if (fifo_empty_now /= '1') then --not empty
                        read_ptr_next <= read_ptr_inc;
                        fifo_full_next <= '0';
                        if (read_ptr_inc=write_ptr_now) then
                            fifo_empty_next <= '1';
                        end if;
                    end if;
                when "10" => -- write
                    if (fifo_full_now /= '1') then -- not full
                        write_ptr_next <= write_ptr_inc;
                        fifo_empty_next <= '0';
                        if (write_ptr_inc=read_ptr_now) then
                            fifo_full_next <= '1';
                        end if;
                    end if;
                when others => --read/write
                    write_ptr_next <= write_ptr_inc;
                    read_ptr_next <= read_ptr_inc;
                end case;
            end process;
end arch;
```

APPENDIX B

SOURCE CODE for DATA FLOW MANAGER

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    12:55:43 10/23/2011
-- Design Name:
-- Module Name:    DataFlowManager - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VComponents.all;
entity DataFlowManager_111 is
 generic (safil : Integer := 49);
    Port ( fifo1_data : in  STD_LOGIC_VECTOR (safil-1 downto 0);
           fifo1_empty : in  STD_LOGIC;
           fifo2_data : in  STD_LOGIC_VECTOR (safil-1 downto 0);
           fifo2_empty : in  STD_LOGIC;
           fifo1_en : out  STD_LOGIC;
           fifo2_en : out  STD_LOGIC;
           DFM_ram_ad_out : out  STD_LOGIC_VECTOR (12 downto 0);
            DFM_port_out : out  STD_LOGIC_VECTOR (4 downto 0);
            DFM_data_out : out  STD_LOGIC_VECTOR (31 downto 0);
             DFM_action_type : out STD_LOGIC_VECTOR (1 downto 0);
            ram_wr_en : out std_logic_vector(0 downto 0);
            DFM_pe_id_out : out std_logic_vector(2 downto 0);
           clk : in  STD_LOGIC;
           rst : in  STD_LOGIC);
end DataFlowManager_111;
architecture Behavioral of DataFlowManager_111 is
signal counter_data : std_logic_vector (1 downto 0) ;
begin
CONTROLLER: process (clk)
begin
            if rst ='0' then
```

```
  if clk'event and clk='0' then
if counter_data = "00" then
        if fifo1_empty='0' then
        fifo1_en <= '1';
        fifo2_en <= '0';
        counter_data <= "01";
        if fifo1_data(0) = '1' and fifo1_data(3 downto 1) = "111" then
        DFM_action_type <= "11";
        ram_wr_en <="1";
        DFM_data_out <= fifo1_data (48 downto 17);
        DFM_ram_ad_out <= fifo1_data (16 downto 4);
        elsif fifo1_data(0) = '1' and fifo1_data(3 downto 1) /= "111" then
        DFM_action_type <= "10";
        DFM_data_out <= fifo1_data (48 downto 17);
        DFM_ram_ad_out <= fifo1_data (16 downto 4);
        DFM_pe_id_out <= fifo1_data (3 downto 1);
        ram_wr_en <="0";
        else
        DFM_action_type <= "01";
        DFM_data_out  <= fifo1_data (48 downto 19) & "00";
        DFM_ram_ad_out <= fifo1_data (18 downto 6);
        DFM_port_out <= fifo1_data (5 downto 1);
        ram_wr_en <="0";
        end if;
        elsif fifo2_empty='0' then
        fifo1_en <= '0';
```

```
fifo2_en <= '1';
counter_data <= "11";
if fifo2_data(0) = '1' and fifo2_data(3 downto 1) = "111" then
DFM_action_type <= "11";
ram_wr_en <="1";
DFM_data_out <= fifo2_data (48 downto 17);
DFM_ram_ad_out <= fifo2_data (16 downto 4);
elsif fifo2_data(0) = '1' and fifo2_data(3 downto 1) /= "111" then
DFM_action_type <= "10";
DFM_data_out <= fifo2_data (48 downto 17);
DFM_ram_ad_out <= fifo2_data (16 downto 4);
DFM_pe_id_out <= fifo2_data (3 downto 1);
ram_wr_en <="0";
else
DFM_action_type <= "01";
DFM_data_out  <= fifo2_data (48 downto 19) & "00";
DFM_ram_ad_out <= fifo2_data (18 downto 6);
DFM_port_out <= fifo2_data (5 downto 1);
ram_wr_en <="0";
end if;
else
fifo1_en <= '0';
        fifo2_en <= '0';
        DFM_action_type <= "00";
        counter_data <= "10";
        ram_wr_en <="0";
```

```
end if;
elsif  counter_data = "01" then
DFM_action_type <= "00";
fifo1_en <= '0';
fifo2_en <= '0';
counter_data <= "10";
ram_wr_en <="0";
elsif  counter_data = "10" then
if fifo2_empty='0' then
fifo1_en <= '0';
fifo2_en <= '1';
counter_data <= "11";
if fifo2_data(0) = '1' and fifo2_data(3 downto 1) = "111" then
DFM_action_type <= "11";
ram_wr_en <="1";
DFM_data_out <= fifo2_data (48 downto 17);
DFM_ram_ad_out <= fifo2_data (16 downto 4);
elsif fifo2_data(0) = '1' and fifo2_data(3 downto 1) /= "111" then
DFM_action_type <= "10";
DFM_data_out <= fifo2_data (48 downto 17);
DFM_ram_ad_out <= fifo2_data (16 downto 4);
DFM_pe_id_out <= fifo2_data (3 downto 1);
ram_wr_en <="0";
else
DFM_action_type <= "01";
DFM_data_out  <= fifo2_data (48 downto 19) & "00";
```

```
DFM_ram_ad_out <= fifo2_data (18 downto 6);
DFM_port_out <= fifo2_data (5 downto 1);
ram_wr_en <="0";
end if;
elsif fifo1_empty='0' then
fifo1_en <= '1';
fifo2_en <= '0';
counter_data <= "01";
if fifo1_data(0) = '1' and fifo1_data(3 downto 1) = "111" then
DFM_action_type <= "11";
ram_wr_en <="1";
DFM_data_out <= fifo1_data (48 downto 17);
DFM_ram_ad_out <= fifo1_data (16 downto 4);
elsif fifo1_data(0) = '1' and fifo1_data(3 downto 1) /= "111" then
DFM_action_type <= "10";
DFM_data_out <= fifo1_data (48 downto 17);
DFM_ram_ad_out <= fifo1_data (16 downto 4);
DFM_pe_id_out <= fifo1_data (3 downto 1);
ram_wr_en <="0";
else
DFM_action_type <= "01";
DFM_data_out  <= fifo1_data (48 downto 19) & "00";
DFM_ram_ad_out <= fifo1_data (18 downto 6);
DFM_port_out <= fifo1_data (5 downto 1);
ram_wr_en <="0";
end if;
```

```
                    else
                    fifo1_en <= '0';
                    fifo2_en <= '0';
                    DFM_action_type <= "00";
                    counter_data <= "00";
                    ram_wr_en <="0";
                    end if;
                    elsif  counter_data = "11" then
                    DFM_action_type <= "00";
                    fifo1_en <= '0';
                    fifo2_en <= '0';
                    counter_data <= "00";
                    ram_wr_en <="0";
                    end if;
        end if;
else
        counter_data <= "00";
        ram_wr_en <="0";
        DFM_action_type <="00";
        fifo1_en <= '0';
        fifo2_en <= '0';
        DFM_data_out <= x"00000000";
        DFM_ram_ad_out <= "0000000000000";
        DFM_port_out <= "00000";
        DFM_pe_id_out <= "000";
end if;
```

```
end process;

end Behavioral;
```

APPENDIX C

SOURCE CODE for PE IMPLEMENTATION

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    20:40:20 10/18/2011
-- Design Name:
-- Module Name:    pe - module
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;
entity pe_111 is
generic (safil : Integer := 49);
    Port ( data_in_west : in  STD_LOGIC_VECTOR (safil-1 downto 0);
           data_av_in_west : in  STD_LOGIC;
           data_in_north : in  STD_LOGIC_VECTOR (safil-1 downto 0);
           data_av_in_north : in  STD_LOGIC;
           data_out_east : out  STD_LOGIC_VECTOR (safil-1 downto 0);
           data_av_out_east : out  STD_LOGIC;
           data_out_south : out  STD_LOGIC_VECTOR (safil-1 downto 0);
           data_av_out_south : out  STD_LOGIC;
            data_backplane : out  STD_LOGIC_VECTOR (4 downto 0);
            fifo_almost_full : out STD_LOGIC;
           clk : in  STD_LOGIC;
           rst : in  STD_LOGIC
            );
end pe_111;
 architecture module of pe_111 is
signal rd_en_fifo1, rd_en_fifo2, empty_fifo1, empty_fifo2, il_data_out_pr,
prpg_data_out_pr : std_logic;
signal ram_wr : std_logic_vector (0 downto 0);
signal dataout_fifo1, dataout_fifo2, dataout_buffer : std_logic_vector (safil-1
downto 0);
```

```
signal DFM_data, ram_out_data : std_logic_vector (31 downto 0);
signal ram_address : std_logic_vector (12 downto 0);
signal dfm_port : std_logic_vector (4 downto 0);
signal dfm_action : std_logic_vector (1 downto 0);
signal pe_id : std_logic_vector (2 downto 0);
signal fifo1_full, fifo2_full : std_logic;
COMPONENT fifo
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                i_fifo_rd : IN std_logic;
                i_fifo_wr : IN std_logic;
                i_fifo_wr_data : IN std_logic_vector(48 downto 0);
                o_fifo_empty : OUT std_logic;
                o_fifo_full : OUT std_logic;
                o_fifo_rd_data : OUT std_logic_vector(48 downto 0)
                );
        END COMPONENT;
        COMPONENT DataFlowManager_111
        PORT(
                fifo1_data : IN std_logic_vector(48 downto 0);
                fifo1_empty : IN std_logic;
                fifo2_data : IN std_logic_vector(48 downto 0);
                fifo2_empty : IN std_logic;
                clk : IN std_logic;
                rst : IN std_logic;
```

```
        fifo1_en : OUT std_logic;
        fifo2_en : OUT std_logic;
        DFM_ram_ad_out : OUT std_logic_vector(12 downto 0);
        DFM_port_out : OUT std_logic_vector(4 downto 0);
        DFM_data_out : OUT std_logic_vector(31 downto 0);
        DFM_action_type : OUT std_logic_vector(1 downto 0);
        ram_wr_en : OUT std_logic_vector(0 to 0);
        DFM_pe_id_out : OUT std_logic_vector(2 downto 0)
        );
END COMPONENT;
component ram
port (
clka: in std_logic;
ena: in std_logic;
wea: in std_logic_vector(0 downto 0);
addra: in std_logic_vector(12 downto 0);
dina: in std_logic_vector(31 downto 0);
douta: out std_logic_vector(31 downto 0));
end component;
begin


fifo1: fifo PORT MAP(
        clk => clk,
        reset => rst,
        i_fifo_rd => rd_en_fifo1,
```

```
        i_fifo_wr =>data_av_in_west,
        i_fifo_wr_data => data_in_west,
        o_fifo_empty => empty_fifo1,
        o_fifo_full => fifo1_full,
        o_fifo_rd_data => dataout_fifo1
);
fifo2: fifo PORT MAP(
        clk => clk,
        reset => rst,
        i_fifo_rd => rd_en_fifo2,
        i_fifo_wr => data_av_in_north,
        i_fifo_wr_data => data_in_north,
        o_fifo_empty => empty_fifo2,
        o_fifo_full => fifo2_full,
        o_fifo_rd_data => dataout_fifo2
);
DFM_111: DataFlowManager_111 PORT MAP(
        fifo1_data => dataout_fifo1,
        fifo1_empty => empty_fifo1,
        fifo2_data => dataout_fifo2,
        fifo2_empty => empty_fifo2,
        fifo1_en => rd_en_fifo1,
        fifo2_en => rd_en_fifo2,
        DFM_ram_ad_out => ram_address,
        DFM_port_out => dfm_port,
        DFM_data_out => DFM_data,
```

```
                DFM_action_type => dfm_action,
                ram_wr_en => ram_wr,
                DFM_pe_id_out => pe_id,
                clk => clk,
                rst => rst
        );
        bram : ram
                port map (
                        clka => clk,
                        ena => dfm_action(0),
                        wea => ram_wr,
                        addra => ram_address,
                        dina =>  DFM_data,
                        douta => ram_out_data
        );
data_out: process (clk)
begin
         if rising_edge(clk) then
                fifo_almost_full <= fifo1_full or fifo2_full;
                if dfm_action= "01"    then
                il_data_out_pr <= '1';
                prpg_data_out_pr<='0';
                elsif dfm_action= "10"          then
                prpg_data_out_pr<='1';
                il_data_out_pr<='0';
                else
```

```
                il_data_out_pr<='0';
                prpg_data_out_pr<='0';
                end if;
        end if;
end process;
combinational: process (clk)
variable index_mux : std_logic_vector(12 downto 0);
variable index_mux_null_det : std_logic;
variable port_mux : std_logic_vector(4 downto 0);
begin
if falling_edge (clk) then
        if il_data_out_pr='1' then
        c1: case DFM_data(31) is
        when '0' => index_mux := ram_out_data (31 downto 19);
        when '1' => index_mux := ram_out_data (18 downto 6);
        when others => index_mux:= "0000000000000";
        end case;
        c2: case ram_out_data(0) is
        when '0' => port_mux := dfm_port ;
        when '1' => port_mux := ram_out_data (5 downto 1);
        when others => port_mux:= "00000";
        end case;
        if index_mux = "0000000000000" then
        index_mux_null_det := '1';
        else
        index_mux_null_det := '0';
```

```
        end if;
        if  index_mux_null_det = '1' then
        data_backplane <= port_mux;
        data_av_out_south <= '0';
        data_av_out_east <= '0';
        else
        if DFM_data(31) = '0' then
        data_av_out_south <= '1';
        data_av_out_east <= '0';
        data_out_south <= DFM_data(30 downto 2) & '0' & index_mux(12 downto
0) & port_mux(4 downto 0) & '0';
        elsif DFM_data(31) = '1' then
        data_av_out_south <= '0';
        data_av_out_east <= '1';
        data_out_east <= DFM_data(30 downto 2) & '0' & index_mux(12 downto 0)
& port_mux(4 downto 0) & '0';
        end if;
        end if;

        elsif prpg_data_out_pr='1'      then
        data_av_out_south <= '1';
        data_av_out_east <= '0';
        data_out_south <= DFM_data & ram_address & pe_id & '1';
        else
        data_av_out_south <= '0';
        data_av_out_east <= '0';
```

```
        data_backplane <= "00000";
        end if;
end if;
end process;
end module;
```