ABSTRACT

A CO-PROCESSOR DESIGN TO SUPPORT MPI PRIMITIVES IN CONFIGURABLE MULTIPROCESSOR DESIGNS

by
Rohan Bafna

The Message-Passing Interface (MPI) is a widely used standard for inter-processor communication in parallel computers. This standard is normally implemented in software, thus resulting in large communication latencies. A hardware implementation can reduce communication latencies significantly, thereby increasing the bandwidth. However, this approach cannot be applied in practice to the very large set of functions in MPI.

Reconfigurable computing has reached levels where entire parallel systems can be built inside one or more FPGAs (Field-Programmable Gate Arrays). In this scheme, specialized components must be built for inter-processor communication and the resulting code is difficult to port to other reconfigurable platforms. In addition, direct performance comparison with conventional parallel computers is not possible since the latter often employ MPI. Introducing MPI primitives in reconfigurable computing creates a framework for efficient code development involving data exchanges, independently of the underlying hardware implementation.

This thesis presents the design and evaluation of a coprocessor that implements a set of MPI primitives. These primitives form a universal and orthogonal set that can be used to implement any other MPI function. A router that can be used to interconnect many such coprocessors in order to build a multi-processor system is also designed and

implemented. The entire design is implemented in the VHDL hardware description language, synthesized using Synplicity Synplify Pro and finally mapped onto the Annapolis Microsystems WILDSTAR-II development board that contains two Xilinx Virtex-II FPGAs.

A CO-PROCESSOR DESIGN TO SUPPORT MPI PRIMITIVES IN CONFIGURABLE MULTIPROCESSOR DESIGNS

by
Rohan Bafna

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

January 2005

APPROVAL PAGE

A CO-PROCESSOR DESIGN TO SUPPORT MPI PRIMITIVES IN CONFIGURABLE MULTIPROCESSOR DESIGNS

Rohan Bafna

Dr. Sotirios Ziavras, Thesis Advisor Professor of Electrical and Computer Engineering, and Computer Science, NJIT	Date

Dr. Alexandros Gerbessiotis, Committee Member Associate Professor of Computer Science, NJIT	Date

Dr. Roberto Rojas-Cessa, Committee Member Assistant Professor of Electrical and Computer Engineering, NJIT	Date

BIOGRAPHICAL SKETCH

Author:	Rohan Bafna
Degree:	Master of Science
Date:	January 2005

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2005
- Bachelor of Engineering in Electrical Engineering,
  Government College of Engineering, Pune, India, 2001

Major: Computer Engineering

To my mother for her unconditional love,
To my father for his guidance,
To my sister and my friends for their encouragement and support,
To MANSHU.

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to Dr. Sotirios Ziavras, who not only served as my research supervisor, providing valuable and countless resources, insight, and intuition, but also constantly gave me support, encouragement, and reassurance. Special thanks are given to Dr. Alexandros Gerbessiotis and Dr. Roberto Rojas-Cessa for actively participating in my committee.

Many of my senior colleagues in the Computer Architecture and Parallel Processing Research Laboratory are deserving of recognition for their support. I would like to thank my senior colleagues Xizhen, Zafrul, Dejiang, Xiaofang for their immense help during practical difficulties. Also, many thanks to Brenda Walker and the entire staff of Electrical and Computer Engineering Department at NJIT. Last, but not the least, I would like to thank Rashmi, Saurabh and Savita – all your understanding and support saw me through difficult times.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(continued)

LIST OF FIGURES (continued)

CHAPTER 1

INTRODUCTION

1.1 Overview of Message Passing Interface (MPI)

The Message-Passing Interface (MPI) is a widely used standard for inter-processor communication in parallel computers. LAM MPI, MPICH and WMPI are examples of a few libraries based on the MPI standard. These libraries are normally implemented in software, thus resulting in large communication latencies for inter-processor communications. A hardware implementation can reduce communication latencies significantly, thereby increasing the bandwidth. However, this approach cannot be applied in practice to the very large set of functions in MPI.

Gerbessiotis and Lee [1] make a strong case for remote memory access (RMA) as an effective way to program a parallel computer. The advantage of using RMA is code cleanliness, reduced programmer confusion and overload and increased code efficiency because two-sided communication is much more tedious than one-sided communication. In addition, the programmer does not need to choose the best way for inter processor communication as there is only one method to do so. Moreover, a communication library implementer (in software or hardware) can more easily provide an efficient implementation of such a method, rather than optimize many apparently equivalent ones. Table 1.1 shows the primitives required for this framework along with their MPI-2 equivalent functions.

Table 1.1 MPI Primitives

LI Function	**Short Description**	**MPI-2 Equivalent**
LIBEGIN (nprocs)	Initiate an SPMD program on nprocs processors	MPI_Init
LIEND ( )	Terminate all nprocs programs	MPI_Finalize
LIABORT()	Abort	MPI_Abort
LINPROCS()	How many processors?	MPI_Comm_Size
LIPID()	Processor id of issuing processor	MPI_Comm_rank
LIPUT(dpid, srcadd, desadd, off, len)	Processor pid sends the contents of its memory starting at address srcaddr of size len bytes to processor's dpid 's memory starting at address desaddr	MPI_PUT(srcaddr, len, MPI_CHAR, dpid, off, len, MPI_CHAR, win);
LIGET(dpid, srcadd, desadd, off, len)	Processor pid gets the contents of dpid 's memory starting at address srcaddr of size len bytes to its own memory starting at address desaddr	Similar to PUT
LIREGISTER (desadd)	Register desaddr as a global variable	MPI_Win_create (……)
LIDEREGISTER (desadd)	Opposite of Register	---
LIBARRIER()	Blocks the calling process until all group members have called the routine	MPI_Barrier (MPI_COMM_WORLD)

The prefix LI indicates "library independent". LIBEGIN(nprocs), LIEND() and LIABORT() are the functions used to start, end and abort a SPMD program. LINPROCS() returns the number of processors in the present SPMD run. Each processor in the system has an identification number and LIPID() returns that number.

LIPUT() and LIGET() are the two RMA functions used for data transfer. *LIPUT(dpid, srcaddr, desaddr, off, len)* is used by a given processor *spid* to send *len* words of data to the processor *dpid*. The data is stored in location *srcaddr* in *spid*, while it is to be written to the location at *off* offset starting from address *desaddr*. Symmetric to the LIPUT () is an LIGET() operation. LIREGISTER () and LIDEREGISTER () are used for registration/ deregistration and have been dealt with in detail in chapter 2. LIBARRIER () is used for synchronization operation and it stalls the issuing processor until all processors in the system have also executed their LIBARRIER () instruction. Hence forth the prefix LI will be dropped and all functions will be referred without this prefix (Eg: LIPUT will be referred simply as PUT).

1.2 Overview of Reconfigurable Computing Systems

Field-Programmable Gate-Arrays (FPGAs) have been used in systems spanning a broad range of applications ever since their introduction in 1985 [14]. Most of the systems use FPGAs as a glue logic providing the advantages of high integration levels without the expense and risk of custom ASIC devices. However, as FPGAs have increased in capacity, their use as in-system configurable computing elements has received considerable attention. The use of FPGAs as reconfigurable computing elements is poised to expand rapidly in the commercial market, where FPGA–based parallel processors will compete with parallel computers and even some supercomputers in computationally intensive applications. Many research projects were done over the past few years in developing these FPGA-based high-performance machines. Reconfigurable FPGA technology holds the potential of reshaping the future of computing by providing the capability to dynamically alter hardware resources to optimally serve immediate computational needs [13].

The FPGA-based reconfigurable systems can be used as specialized co-processors, processor-attached functional units, attached message routers in parallel machines, and specialized systems for parallel processing. This was made possible with the advent of multi-million gate FPGAs. In the past decade, FPGA-based configurable computing machines have acquired significant attention for improving the performance of algorithms in several fields, such as DSP, data communications, genetics, image processing, pattern recognition, etc. FPGA-based co-processors are implemented as attached co-processors dedicated to off-loading computationally intensive tasks from host processors in PCs and workstations. Reconfigurable co-processors are viable platforms for a wide-range of computationally-intensive applications. The FPGA-based configurable computing systems have garnered support from the scientific and academic communities. Many research projects have demonstrated the viability of configurable computing systems that can deliver the performance of supercomputers for specific applications. Most of the FPGA–based parallel machines currently reside in multi-FPGA systems interconnected via a specific network [15]. Some of the configurable computing systems are:

1. The Ganglion Project at the IBM Almaden Research Centre used XC3090 and XC3042 FPGA devices to implement a feed-forward, fully interconnected neural network on a single VME board.

2. DEC's Paris Research Lab has designed and implemented four generations of FPGA-based configurable co-processors called Programmable Active Memories (PAMs).

3. SPLASH-1 includes a 32-stage linear-logic array with a VME-interface to a SUN workstation. Each stage consists of an XC3090 FPGA and a 128Kbyte static memory buffer. SPLASH-1 outperformed Cray-2 by a factor of 325 in specific applications and a custom built NMOS device by a factor of 45. SPLASH-2 uses 17 XC4010 FPGA devices arranged in a linear array and also interconnected via a 16x16 crossbar.

4. PRISM-1 from Brown University coupled XC3090 with the Motorola M68010 microprocessor and PRISM-11 coupled XC4010 FPGA devices as co-processors to an AMD29050 RISC processor.

Advances in VLSI technology not only brought about multi-million gate FPGAs, but also facilitated the integration of numerous functions onto a single FPGA chip. Peripherals formerly attached to the FPGA at the board level now can be embedded into the same chip with the configurable logic. According to Xilinx predictions, the count of FPGA system gates will exceed 50 million and FPGA chips will operate at more than 500 MHz [16].Thus, the availability of multi-million system gates in FPGAs introduced a new design paradigm, System-On-a-Chip (SOC), with which entire systems can be implemented on a single FPGA chip without the need for expensive non-recurring engineering charges or costly software tools.

The FPGAs have provided an alternative method to computing by supporting the fine-tuning of hardware to match software requirements. The fact that the number of system gates in FPGAs has been increasing rapidly in recent years encourages the development of large–scale application-specific custom computing machines on FPGAs for better hardware performance. While these FPGA-based Custom Computing Machines (CCMs) may not challenge the performance of microprocessors for all applications, for specific applications an FPGA-based system can offer extremely high performance.

1.3 Overview of the WILDSTAR-II Hardware Development Board

The Annapolis Micro Systems high performance WILDSTAR-II board combines the high density of Xilinx Virtex-II FPGAs with very high memory and I/O bandwidth capacities. The WILDSTAR-II board supports both VHDL design flow tools and the CoreFire Design Suite.

- The *VHDL* flow consists of four steps: 1) creation of VHDL design using supplied VHDL model; 2) simulation using the ModelTech application; 3) synthesis using Synplicity, and 4) place-and-routing using Xilinx tools.
- The *CoreFire* design flow involves two steps: creating a design using the CoreFire Application Builder, and then place-and-routing using Xilinx FPGA design tools.

A host computer can communicate with the WILDSTAR-II /PCI board via the PCI bus interface, which in turn communicates with the board's PCI controller. The board contains two Xilinx XC2V6000 FPGAs with 128 MB of SDRAM (distributed in 2 banks, giving 1.6 GBytes/sec per board) and 12 MB of DDR II SRAM (distributed in 12 banks, giving approximately 11 GBytes/sec per board) as shown in Figure 1.1. Each FPGA is a complete computing system and is called a processing element (PE). The WILDSTAR-II /PCI controller has access to the PEs using the Local Address Data (LAD) bus. Thus, the host also has direct register access and communication with PEs over the LAD bus. PE0 and PE1 are pin- and bit file-compatible with each other. The PEs are connected by dual 166-pin differential data buses, providing double the data throughput as a single bus.

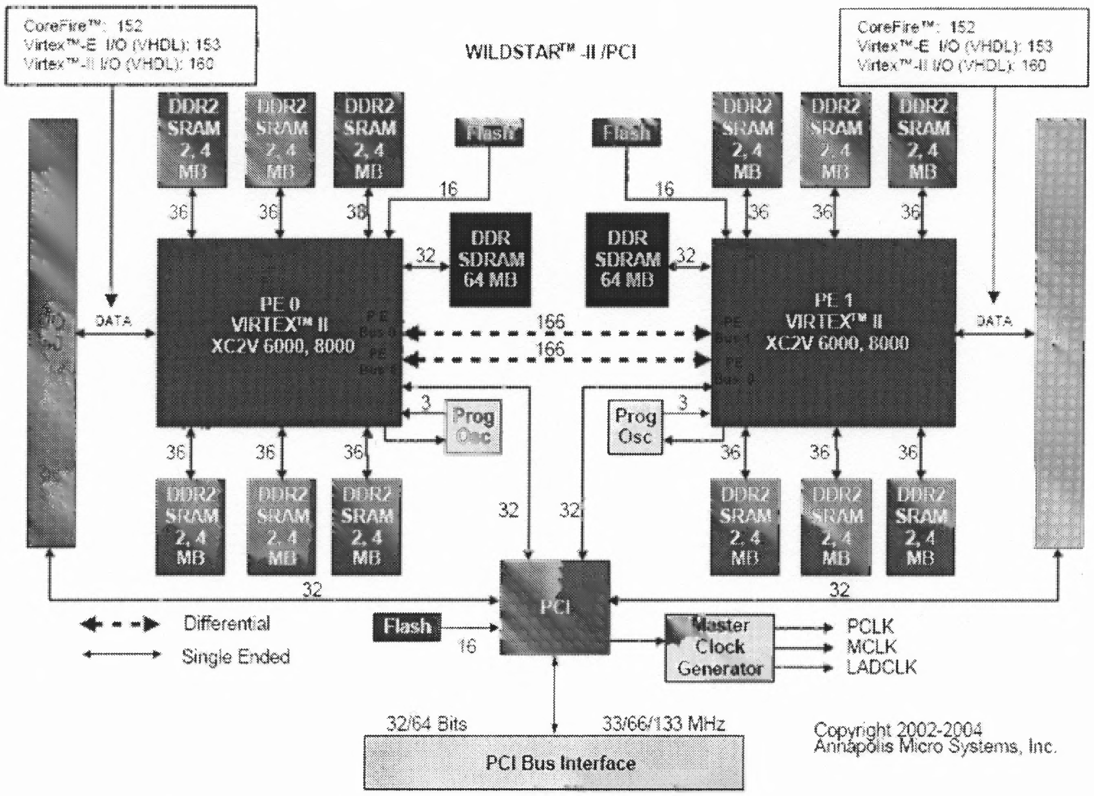
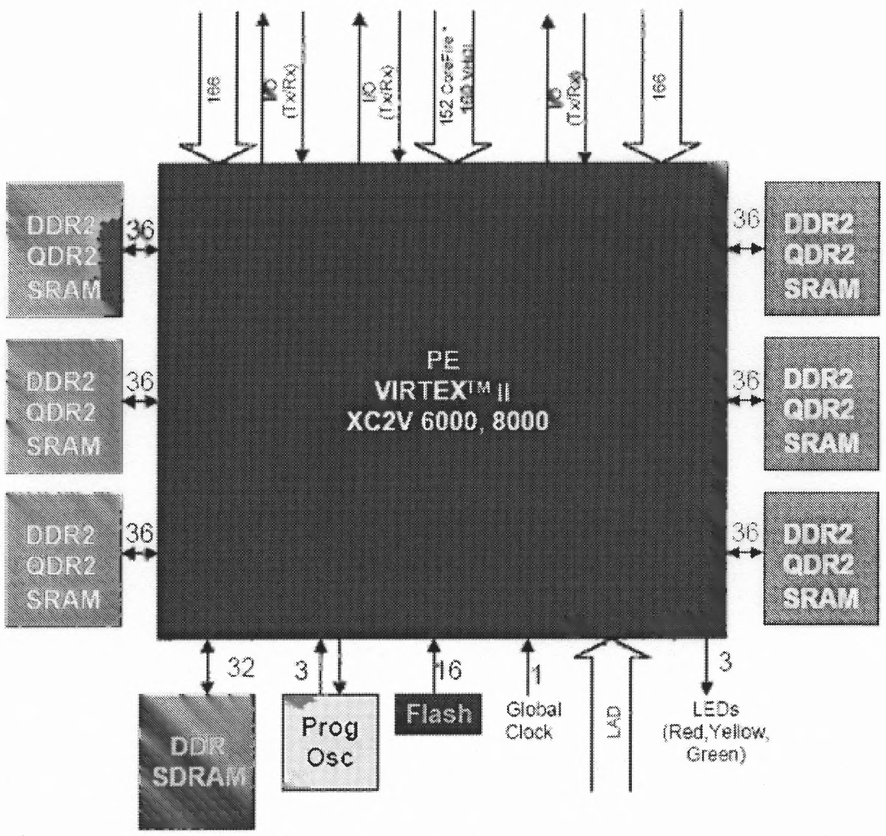


Figure 1.1 Wildstar II/PCI block diagram [17].

1.3.1 WILDSTAR-II PE Modules

Each processing element in the WILDSTAR-II architecture, shown in Figure 1.2, consists of the following:

- One Xilinx Virtex-II XC2V6000 FF1517 – 5C FPGA chip.
- Six independent DDR2 SRAM ports, 960 MBytes/sec bandwidth per port.
- One Bulk DDR DRAM port, 800 MBytes/sec bandwidth.
- Three I/O data buses with dedicated Transmit (Tx) and Receive (Rx) clocks.
- 32-bit LAD bus.
- Flash storage of multiple FPGA images.
- Three global and three local clocks, three user LEDs.



*CoreFire™ uses a small number of bits for overhead requirements.

Figure 1.2 WILDSTAR-II Processing Module [17].

1.3.2 WILDSTAR-II Clocks

The WILDSTAR™-II board has two types of clocks: the global board clocks MCLK, PCLK, and ICLK; and local clocks for each PE, consisting of ACLK, BCLK, and CCLK. MCLK and PCLK are differential, while ICLK is fixed and single-ended. PE local clocks are also single-ended and individually configurable. Both, MCLK and PCLK are differential, asynchronous to each other and configurable through the WILDSTAR™-II host software. ICLK is the Local Address Data Bus (LAD Bus) clock. It is fixed at 132 MHz and single-ended. The PE uses this clock to interface to the PCI Controller for host access via the LAD bus.

1.3.3 Overview of Virtex II FPGAs

The Virtex-II devices are user-programmable gate arrays with various configurable elements [16]. As shown in Figure 1.3, the programmable device is comprised of input/output blocks (IOBs) and internal configurable logic blocks (CLBs). The internal configurable logic includes four major elements organized in a regular array.

- Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.
- Block SelectRAM memory modules provide large 18 Kbit storage elements of true dual-port RAM. Each port is totally synchronous, independent, programmable from 16K x 1 bit to 512 x 36 bits (in various depth and width configurations), offering three "read-during-write" modes.
- Multiplier blocks are 18-bit x 18-bit dedicated multipliers.
- DCM (Digital Clock Manager) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, coarse- and fine-grained clock phase shifting. The DCM also provides 90-, 180-, and 270-degree phase-shifted versions of its output clocks.

A new generation of programmable routing resources called Active Interconnect Technology interconnects all of these elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and designed to support high-speed designs. All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.

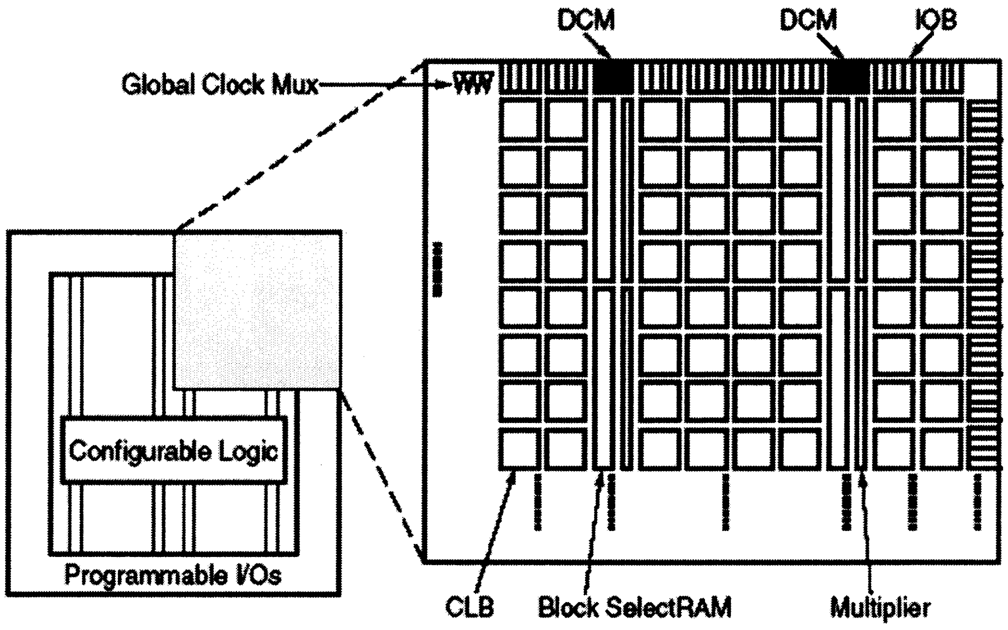


Figure 1.3 Virtex II architecture overview [16].

1.4 Related Work

An efficient MPI-2 port on a the Clint networking prototype architecture which provides two physically separated channels for large packets at high bandwidth and for small packets with low forwarding latency is described in [23]. Other than this, our approach of direct implementation of MPI primitives in hardware is a new approach.

1.5 Motivation

An implementation of inter processor communication in direct hardware can reduce the communication latencies significantly, thereby increasing the bandwidth. However, this approach cannot be applied in practice to the very large set of functions in MPI.

Reconfigurable computing has reached levels where entire parallel systems can be built inside one or more FPGAs (Field-Programmable Gate Arrays). In this scheme, specialized components must be built for inter-processor communication and the resulting code is difficult to port to other reconfigurable platforms. In addition, direct performance comparison with conventional parallel computers is not possible since the latter often employ MPI. Introducing MPI primitives in reconfigurable computing creates a framework for efficient code development involving data exchanges, independently of the underlying hardware implementation.

1.6 Objective

The main objective of this thesis is to design and implement a coprocessor for inter processor communication that implements a set of MPI primitives. This thesis also aims to implement a router which will be used to interconnect many such processor-coprocessor systems to build a multi processor system. The target system is the Annapolis Micro Systems WILDSTAR-II hardware development board with two Xilinx XC2V6000 Virtex-II FPGAs.

CHAPTER 2

REGISTRATION TABLE DESIGN

2.1 Registration / Deregistration

In a multi processor system, a shared global variable may be stored in different memory locations in different processors. But in order to make programming easy and efficient, this abstraction should always be hidden from the user. Registration is a process by which this relationship is established. From the user's perspective, registration means designating a variable as global. Since this local/global scenario arises only when communication is to be done with other processors, it suffices to handle this in the coprocessor. Hence, each coprocessor is entrusted the task of global-local conversion and vice versa. Deregistration is a process opposite to registration.

The global address for this design is 8 bits wide and hence this design can have $2^{=}$ 256 global addresses. The local address is 32 bits wide, allowing each main CPU to index, 2^{32} = 4 Gbytes of main memory. The data structure designed for this functionality is called a Registration Table (RT). The registration operation stores a 32-bit local address in the next vacant position in the table. The index of this 32-bit address into the table is a global address. The deregistration operation removes the 32-bit local address from the table.

Let's look at an example:

At the start the RT is blank.

Assume the operations Register(2000), Register(5000) and Register(10000) in this order.

Now the table contains:

2000	0
5000	1
10000	2
....	..

So the global address of 2000 is 0, of 5000 is 1 and of 10000 is 2.

Now if we execute Deregister (5000), then

2000	0
-	1
10000	2
....	..

Now if we execute Register (2345), then

2000	0
2345	1
10000	2
....	..

2.1.1 Hardware Design of the Registration Table

The following functionality is required in the hardware implementation of RT:

1. Normal RAM lookup:
Give an 8-bit global address and get a 32-bit local address.
(This is required when a packet is received by the coprocessor.)

2. Content Addressable Memory (CAM) look up:
Give a 32-bit local address & get 8-bit global address.
(This is required when the coprocessor executes a PUT or GET and is forming a packet.)

3. Register:
Register a 32-bit address as the next available 8-bit global address.

4. Deregister
De-register the given 32-bit address.

Hence, the hardware required is a CAM/RAM structure with very efficient write/delete operations. The basic building block for the registration table is the CAM32x9 macro shown in Figure 2.1. The unique Virtex-II block RAM approach is used to build the CAM32x9 macro [10]. This methodology is based upon the true dual-port feature of the block SelectRAM+ memories. Ports A and B can be configured independently, anywhere from 16K-word x 1-bit to 512-word x 32-bit. Each port has separate clock inputs and control signals. The internal address mapping of the block SelectRAM+ memory is the primary feature in designing a CAM in a true dual-port block RAM. Each port accesses the same set of 16K memory locations using an addressing scheme dependent on the port width.

2.1.2 CAM32x9

The CAM32x9 macro is shown in Figure 2.1.

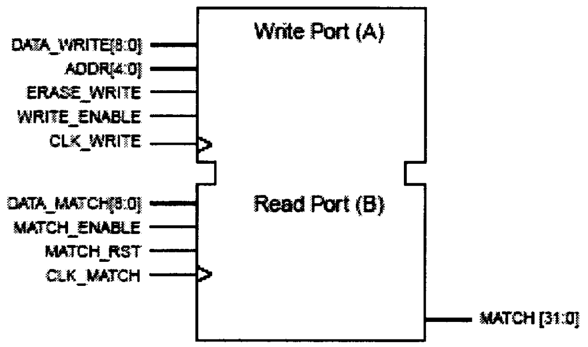


Figure 2.1 CAM32x9 macro [10].

The CAM32x9 macro has the following features:

- 32-word x 9-bit organization
- Independent match (read) and write data input buses
- Decoded address output (or 32-bit "one-hot" decoded address)
- Fully synchronous match port (or read port)
- Fully synchronous write port
- Single clock cycle match (single or multi-matches)
- Single clock cycle write (and single clock cycle erase)

2.1.2.1 CAM32x9 Basics. The CAM32x9 macro stores the 9-bit input word as "one-hot" decoded 512-bit word. A 9-bit word has 29 (= 512) possible values. A classic RAM stores the 9-bit word into a 9-bit location. However, the 9-bit word can also be represented as a 512-bit word, with all zeros and a single "one" at the nth location, where n corresponds to the position given by the decoded 9-bit data. For example, if the data is "000000111" (a decimal seven), the decoded 512-bit word is "0000....000010000000", where the "one" is at the seventh location counting from zero. Thirty-two 512-bit words store thirty-two decoded 9-bit words. As shown in Figure 2.2, an array of 32 x 512 represents 32 addresses from 0 to 31. Column 'i' represents address 'i'. Storing a data value 5 in address 2 means storing "00..10000" in the column 2 as shown by the shaded column in Figure 2.2. Searching for 5 is equivalent to reading out row 5 which is "0000......100", corresponding to a match at address 2 as shown by the shaded row in Figure 2.2.

	ADDR[4:0]																	
	31	30	29	...	...	...	...	...	...	...	...	...	05	04	03	02	01	00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
3	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
4	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
5	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	1	0	0
.	...	...	...	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	...	...	...	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	...	...	...	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	...	...	...	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	...	...	...	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
.	0	0	0	...	...	...	...	...	...	...	...	...	0	0	0	0	0	0
510	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
511	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2.2 "One-hot" decoded data representation for CAM lookup.

2.1.2.2 Implementing the CAM32x9 Macro using Block SelectRAM+.

Read (or match port):

The 16K-bit RAMB16_S1_S36 with a 32-bit wide data port (port B), and a 4-bit wide port for parity, generates 32 values simultaneously. If the 9-bit data (DATA_MATCH) to be searched is connected to the 9-bit address (ADDRB) of port B as shown in Figure 2.3, the 32-bit port B generates the matches concurrently. Using the fact that a particular location corresponds to the decoded 9-bit data, the matching operation is equivalent to searching 32 locations for specific 9-bit data at the same time. Figure 2.3 shows the RAMB16_S1_S36 port B configured as a CAM read port.

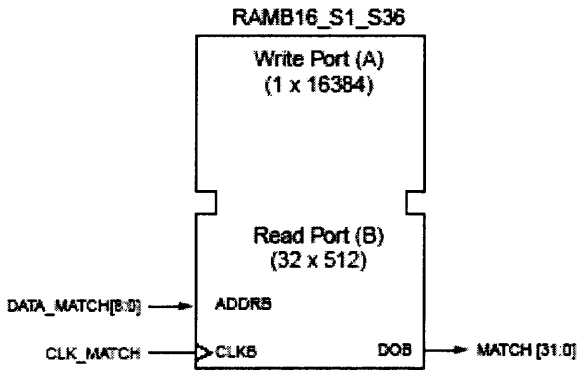
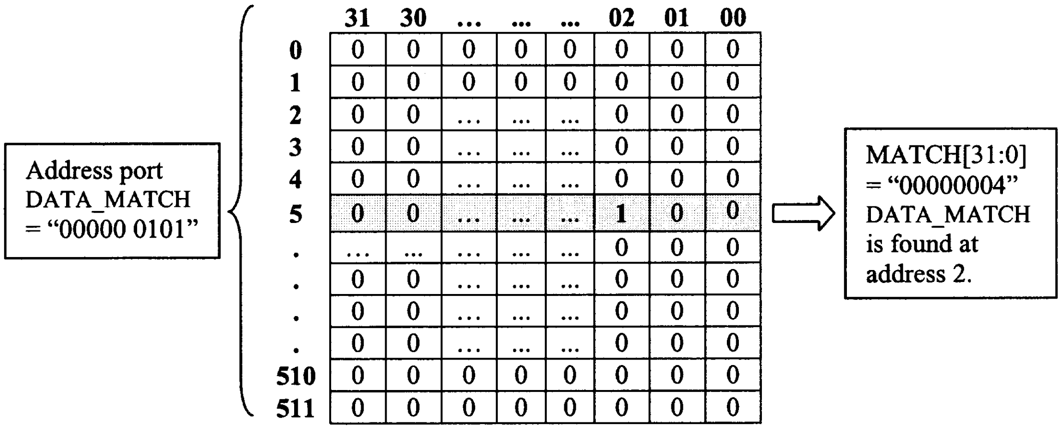


Figure 2.3 Read port of a CAM32x9 in a Block SelectRAM+.

If the word "0000 00101" was previously written at the address "00010" of the CAM32x9, a match operation is equivalent to a read operation of a block RAM with the 9-bit word "0000 0101" placed on the address input of port B (ADDRB[8:0]). The 32-bit port B output is "0000 0000 0000 0000 0000 0000 0000 0100" corresponding to a match found at location 2, as shown in Figure 2.4. Port B output is the decoded CAM address bus. If no match is found, the output is "0000 0000 0000 0000 0000 0000 0000 0000." If one or several matches are found, each corresponding location equals "one". In this last case, the same 9-bit word has been stored at different addresses, and the CAM32x9 output is multi-matches.



READ: DATA_MATCH[8:0] = "00000 0101"
MATCH[31:0] = "0000 0000 0000 0000
0000 0000 0000 0100" (match found)

Figure 2.4 An example of a read into the CAM32x9.

Write Port:

The CAM write port inputs are a 9-bit data bus, an address bus (five bits to address the 32 locations), control signals, and the clock. The 5-bit address bus selects a memory location. Writing new data into this location is equivalent to decoding the 9-bit data into a 512-bit "one-hot" word and storing the 512-bit word. However, if the CAM32x9 macro is initialized to zero, only one bit of the 512-bit word has to toggle. The location of the "one" is determined by the "one-hot" decoded 9-bit value.

Taking into account that the address port of the block SelectRAM+ primitive decodes the address bus, both operations previously described are combined into a simple write in the block RAM. Port A configured as 16384 x 1 has a 1-bit data input and a 14-

bit address input. The data input is asserted to "one", and the 9-bit data plus the 5-bit address are merged in a single 14-bit address input.

With the 9-bit data as MSB and 5-bit address as LSB, the resulting 14-bit address input decodes the 9-bit data and selects one of the 32 memory locations simultaneously. The clock edge stores a "one" at the corresponding location. Figure 2.5 shows the write port of the 32 x 9 CAM in a block SelectRAM+ primitive.

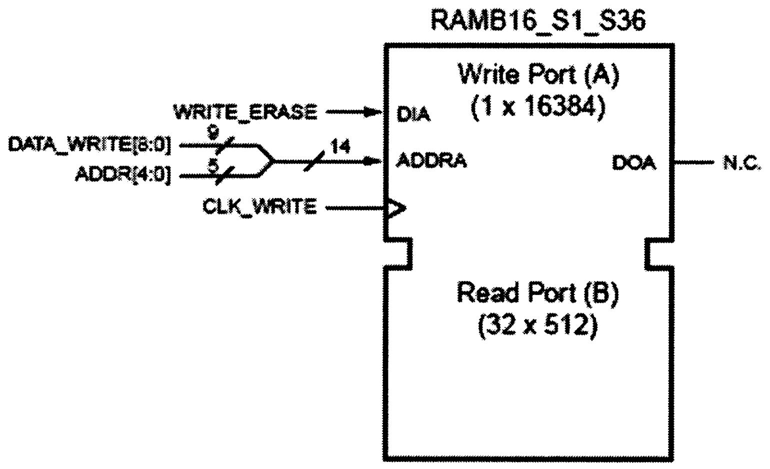


Figure 2.5 Write port of the CAM32x9 in a Block SelectRAM+ [10].

Erase operation:

To erase previously stored data, the selected location must be initialized to "zero" (512-bit word). The basic option is to write "zero" during 512 clock cycles by incrementing the 9-bit MSB of the address input from 0 to 511. The 5-bit LSB of the address input is fixed and is used to select the CAM location.

Single cycle erase operation:

The erase operation is equivalent to the write operation with the exception that a "zero" is stored instead of a "one". The port A address input is again a combination of the 9-bit data to be erased and the 5-bit address bus. To perform the correct selection, the 9-bit data must be stored into a separate RAM block. Figure 2.6 shows the single cycle erase logical implementation of CAM32x9 using nine RAM32x1s blocks (32 x 1 RAM in nine LUTs) to memorize the 32 words.

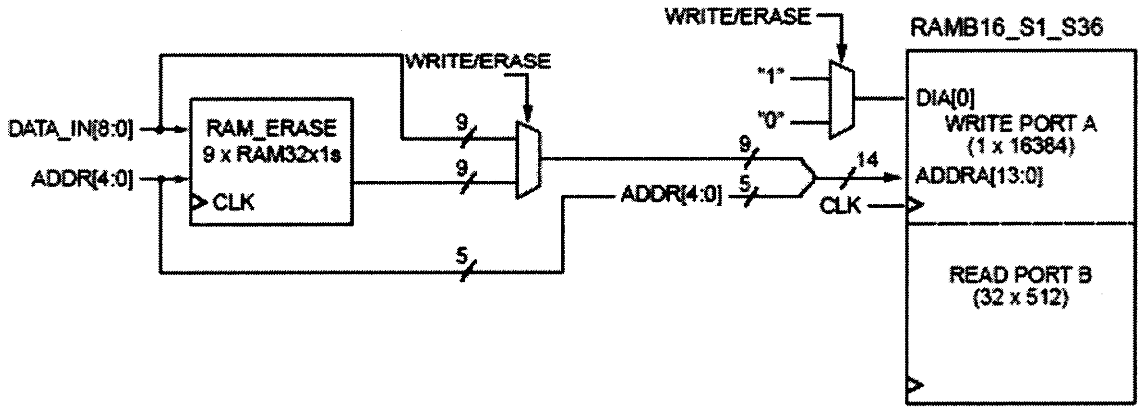


Figure 2.6 CAM32x9 macro: One clock cycle erase and clock cycle write.

During the erase cycle, the old datum is read from RAM_ERASE and is used to erase the previous value in the block RAM. Also simultaneously, the new 9-bit input word is stored in RAM_ERASE. In the next clock cycle, a normal write is performed using this new 9-bit value into the block RAM.

The two multiplexers are controlled by the WRITE/ERASE.

- WRITE/ERASE = 0 connects the output of RAM_ERASE to the block RAM, while WRITE/ERASE = 1 connects the input 9-bit word to the input of the block RAM.
- WRITE/ERASE = 0 connects "0" to the write input of the block RAM, while WRITE/ERASE = 1 connects "1" to the write input of the block RAM.

Figure 2.7 shows the final implementation of the CAM32x9 macro and Figure 2.8 shows the corresponding erase and write waveforms for the CAM32x9. The multiplexer between the DATA_IN to be written and the DATA_WRITE to be erased is implicit in this implementation. (See Figure 2.6). Because the first clock cycle is the erase mode, the old word is read from the ERASE_RAM output DATA_WRITE. It becomes the 9-bit MSB address of ADDRA input. The new data (DATA_IN) is written into the ERASE_RAM and is reflected on the output DATA_WRITE. Writing into the ERASE_RAM is controlled by the WRITE_RAM input. The second clock cycle is the write cycle with the new data automatically used as the nine MSB address, as shown in Figure 2.7. The ADDR[4:0] input is unchanged during the two clock cycles and is used as the five LSB address. The DATA_WRITE output of ERASE_RAM is also the DATA_OUT output of the CAM32x32 to add the normal RAM functionality, making CAM32x9 a CAM/RAM structure.

The WRITE_ENABLE input, which directly drives the WEA[1] and ENA[1] ports of the RAMB16_S1_S36, is used to enable writing into the CAM32x9. The MATCH_ENABLE input, which drives the ENB[1] port of RAMB16_S1_S36, is used to enable the read (or match) operation. The inverted MATCH_RST input, which is connected to the RSTB[1] port of the RAMB16_S1_S36, is used to reset the MATCH output to all 0's. Table 2.1 summarizes all the operations of CAM32x9.

Table 2.1 CAM32x9 Operations - Summary

Operation	**Input Port**	**Output Port**	**Clock Cycles**
RAM_lookup	ADDR[4:0]	DATA_OUT[8:0]	1
CAM_lookup	DATA_MATCH[8:0]	MATCH[31:0]	1
WRITE9	ADDR[4:0], DATA_IN[8:0]	--	2

[1]For functional details of this RAMB16_S1_S36 input port, see Vistex II data sheets [16].

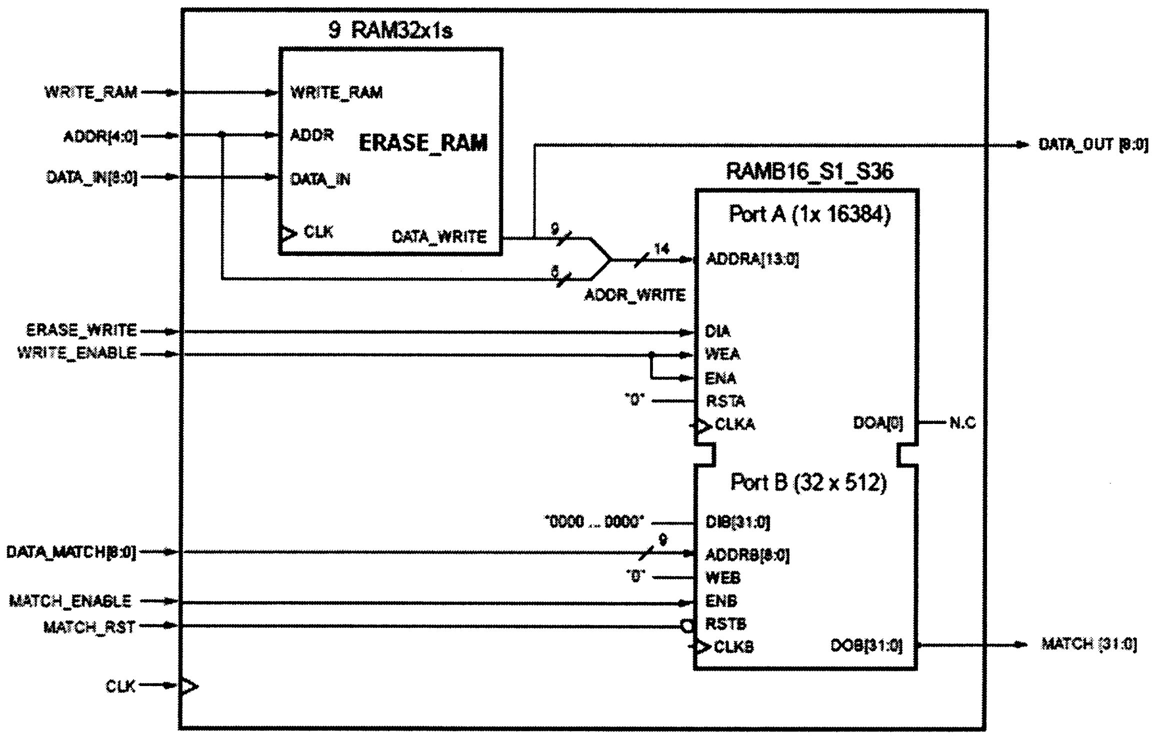


Figure 2.7 CAM32x9

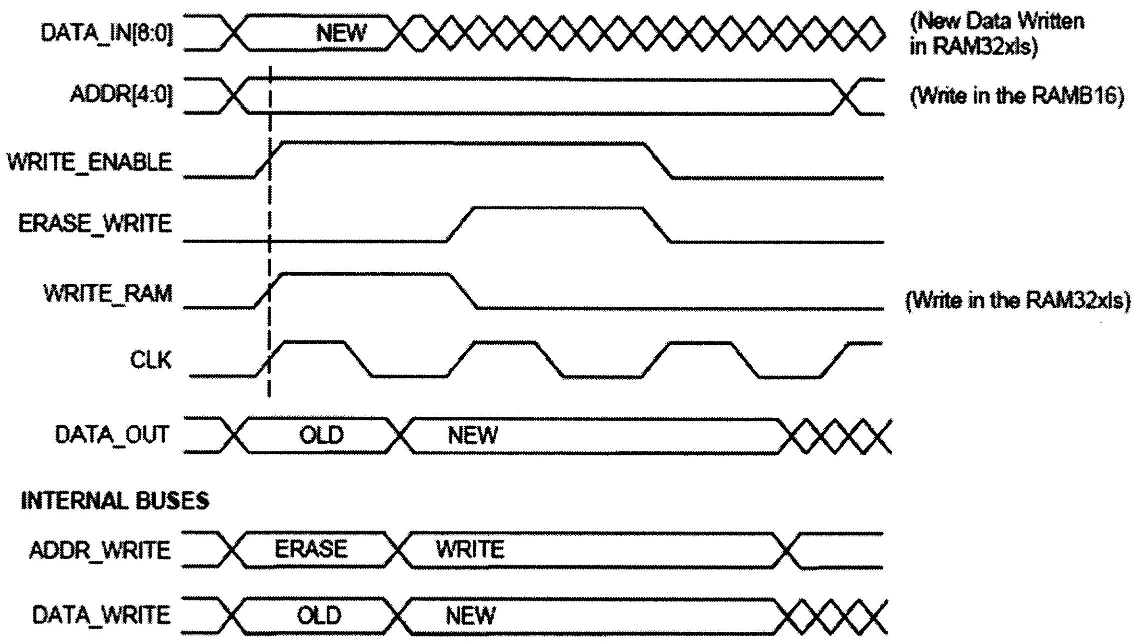


Figure 2.8 Erase and write waveforms (adapted from [10]).

2.1.3 CAM32x32

Four CAM32x9's are cascaded as shown in Figure 2.9 to obtain the 32-bit wide CAM32x32. One input of each CAM32x9 is permanently connected to '0'. Each CAM32x9 stores a part of the 32-bit input word. A match in this case will correspond to a match in all four CAM32x9's and hence the MATCH output is the logical AND of the MATCH outputs of the four CAM32x9s. A 32-to-5 encoder produces the 5-bit MATCH_ADDR output, which corresponds to the address of the input word in the CAM. Table 2.2 summarizes the operations of CAM32x32.

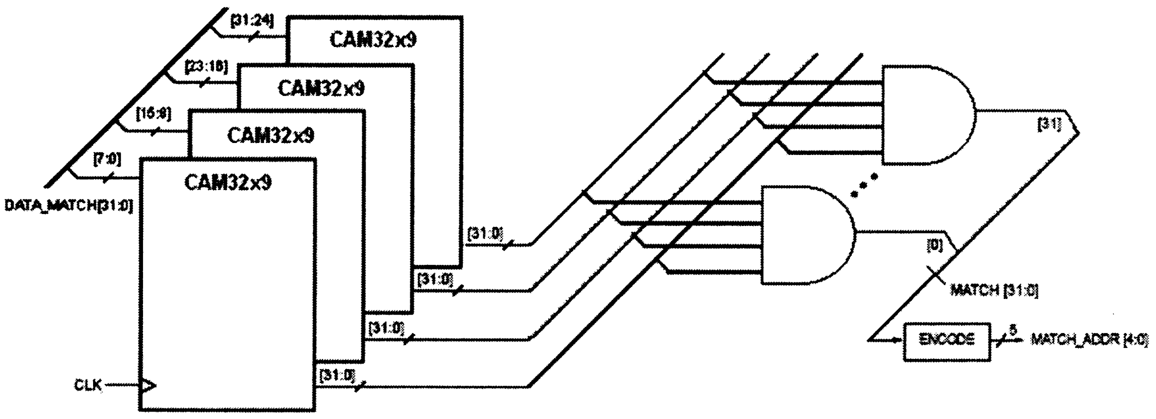


Figure 2.9 CAM32x32: Four cascaded CAM32x9s (adapted from [10]).

Table 2.2 CAM32x32 Operations - Summary

Operation	Input Port	Output Port	Clock Cycles
RAM_lookup	ADDR[4:0]	DATA_OUT[31:0]	1
CAM_lookup	DATA_MATCH[31:0]	MATCH_ADDR[4:0]	1
WRITE32	ADDR[4:0], DATA_IN[31:0]	--	2

2.1.4 Registration Table

A 32 word deep Registration Table (RT), designed using the CAM32x32 block, is shown in Figure 2.10.

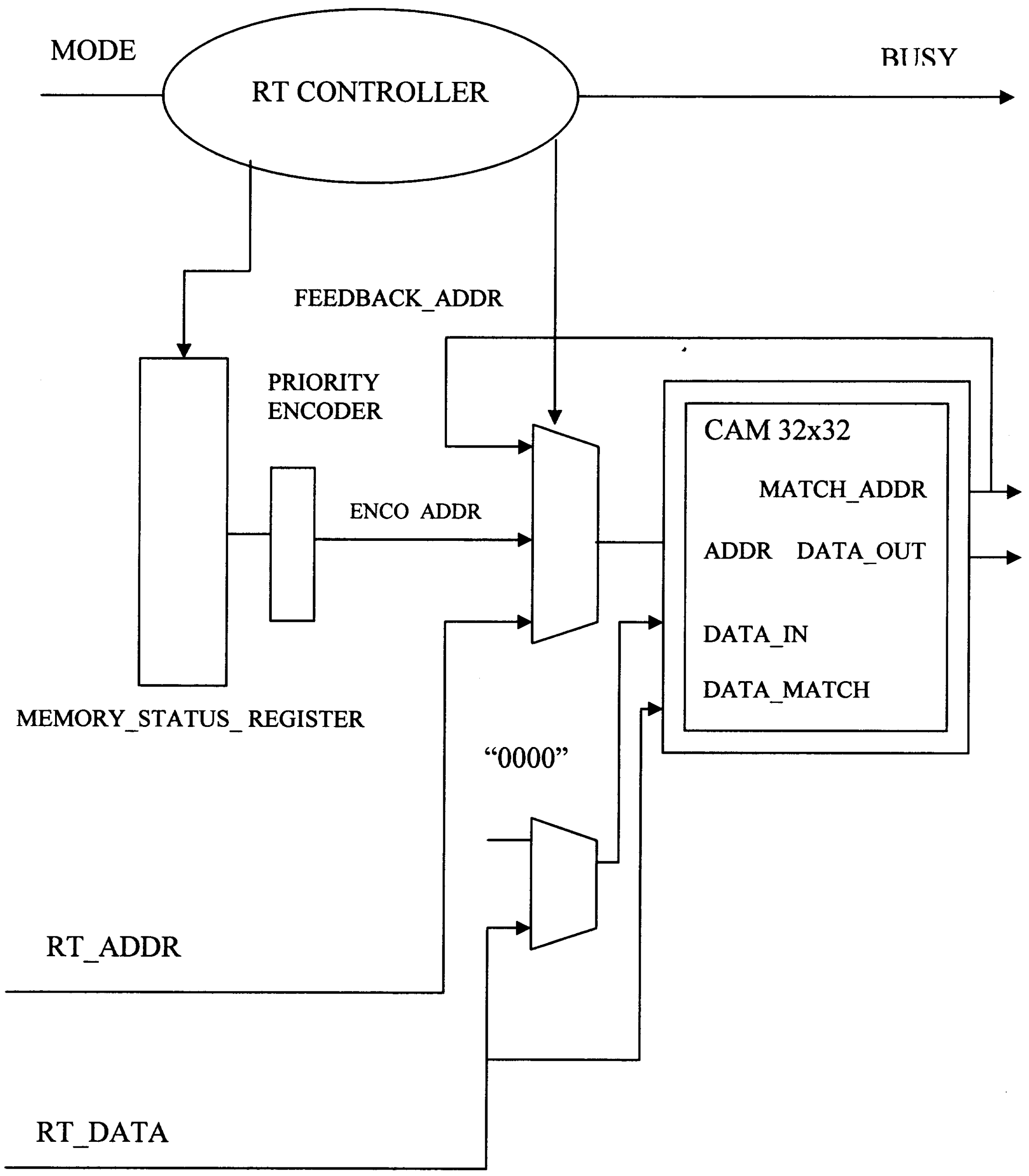


Figure 2.10 Registration table

The 32-bit MEMORY_STATUS_REGISTER keeps record of whether a position is filled or empty in the CAM/RAM. A "1" in position 'i' in the MEMORY_STATUS _REGISTER indicates the corresponding location is filled in the RT. A priority encoder[#] generates a next address for registration opearation (write) by encoding the status register. The output of the priority encoder is the index of next vacant position in the RT. The RT controller block is responsible for generating the control signals for the two multiplexers and also the BUSY output of the CAM. The BUSY signal is asserted for one cycle during the 2-cycle registration operation and for two cycles during the deregistration operation. The four operations required by the RT are handled as follows:

Normal RAM lookup:

The 5-bit RT_ADDR input is passed to the ADDR input of the CAM32x32 block and a "RAM_lookup" operation is performed on CAM32x32, reading out a 32-bit word from DATA_OUT output.

CAM lookup:

The 32-bit RT_DATA_IN input is sent to the DATA_MATCH input of the CAM32x32 block and a "CAM_lookup" operation is performed, giving a 5-bit match value from the MATCH_ADDR output.

Registration:

The 32-bit RT_DATA_IN and the 5-bit ENCO_ADDR are given as inputs to the DATA_IN and ADDR ports of the CAM32x32 block and "WRITE32" operation is performed. Also, a "1' is written into MEMORY_STATUS_REGISTER [ENCO_ADDR].

# Refer the Appendix A for the truth table of the priority encoder

Deregistration:

In the first clock cycle, the 32-bit RT_DATA_IN is passed to the DATA_MATCH input of the CAM32x32 block and a "CAM_lookup" operation is performed. In the next clock cycle, the MATCH_ADDR output of CAM32x32 is fed back to the ADDR input and a "WRITE32" operation is performed using "00000000" as input to the DATA_IN. Also, a "0" is written into MEMORY_STATUS_REGISTER[FEEBBACK_ADDR].

The various modes of operation of the RT are outlined in the Table 2.3.

Table 2.3 Registration Table Operations

Mode	Operation	Input Port	Output Port	Clock Cycles
00	NOP	--	--	1
01	RAM_lookup[1]	RT_ADDR[4:0]	RT_DATA_OUT[31:0]	1
	CAM_lookup[1]	RT_DATA_IN[31:0]	RT_MATCH_ADDR[4:0]	
10	Registration	RT_DATA_IN[31:0]	--	2
11	Deregistration	RT_DATA_IN[31:0]	--	3

[1]The RAM_lookup and CAM_lookup operations can be performed simultaneously in Mode = 01

CHAPTER 3

COPROCESSOR DESIGN

The task at hand was to design a coprocessor for inter-processor communications in a multi processor system that employs MPI. The main objective was to enable the main CPU to off load all the communication tasks to the coprocessor and not waste valuable CPU cycles for communication activities. A similar scheme has been used in IBM Blue Gene/L, where the second processor in each node can be used as communications coprocessor.

3.1 Issues for Coprocessor Design and Instruction Encoding

The coprocessor design has the following objectives:

- Portable.
- Should require no change to the main CPU design.
- Easy to Interface to any main CPU.
- The CPU should be able to transfer the communication tasks to the coprocessor in minimum clock cycles and with minimum overhead.

Hence, the choice was made to attach the coprocessor to the CPU system bus as shown in Figure 3.1. This way the coprocessor will also have access to the system's main memory which will enable it to do RMA transfers for which it is intended. The main CPU to coprocessor communication was made memory mapped as shown in Figure 3.2. Thus, MOV instructions can be used to transfer the commands to the coprocessor. Using MOV has the following advantages:

1. No changes are required in the main CPU design

2. No major changes are required in the compiler to compile programs utilizing the coprocessor. The compiler only has the trivial task of mapping each instruction to one or more MOV instructions.

3. Also operands can be effectively transferred at run time from the register file of the main CPU to the coprocessor.

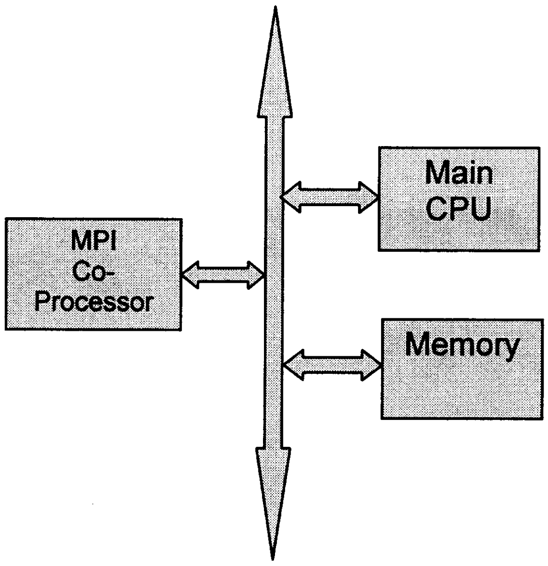
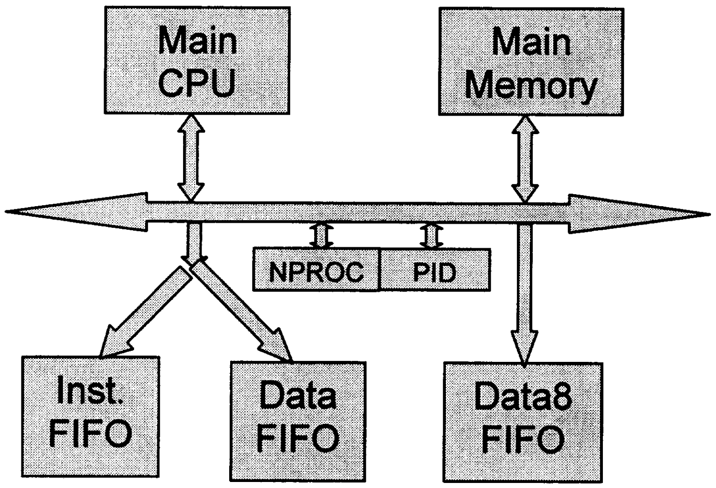


Figure 3.1 Basic architecture.



Address Map

Destination of MOV	Address Bus A1A0
CS = 1	
PID register	00
NPROCS register	01
Inst. & Data FIFO	10
Data8 FIFO	11
CS = 0	
No Effect of MOV	xx

Figure 3.2 Hardware interface between the coprocessor and the main CPU.

Using the Chip Select (CS) and the Address bus (A1, A0) inputs, the system designer can map the coprocessor to any desired address in the system memory map. Once the correct chip select has been generated by decoding the address from the main system address bus, then the coprocessor can use simple MOV commands to transfer data to the three FIFO buffers – Inst. FIFO, Data FIFO and Data8 FIF0. These are First-In-First-Out structures which serve as the instruction and data memory of the coprocessor. Three such buffers have been used to ensure that all the instructions can be smoothly pipelined in the coprocessor and this will become clear when the working of the coprocessor pipeline is discussed. Table 3.1 lists the operands required to be sent to the coprocessor from the main CPU for each instruction.

Table 3.1 Coprocessor Operands

Instruction	Operands needed to be sent to the coprocessor
Put-n	Destination address (32-bit) Source address (32-bit) Destination PID (8-bit) Length of data (8-bit) Offset (8-bit)
Put-1	Destination address (32-bit) Data (32-bit) Destination PID (8-bit) Length of data (8-bit) Offset (8-bit)
Get	Destination Address (32-bit) Source Address (32-bit) Destination PID (8-bit) Length of data (8-bit) Offset (8-bit)
Register / Deregister	Address (32-bit)
Set PID / NPROCS	Value (8-bit)
Begin, End , Abort	None

The Put and Get instructions require the maximum amount of data to be transferred from the CPU to the coprocessor. One way would be to make the data bus 40 bits wide so that one can transfer 32 bits of data plus 8 bits of opcode for the coprocessor. But any change in the main CPU is not desirable. The problem was to be able to transfer 32-bit data and an opcode simultaneously to the one of the FIFO structures in the coprocessor. This was accomplished by using part of the address bus to carry the opcode.

1. Register/ De-register

Data bus

Address
32

Address bus

Opcode	x	x	x
8	8	8	8

2. Set PID/NPROCS

Data bus

All 0's	Value
24	8

Address bus

Opcode	x	x	x
8	8	8	8

3. Put-n/ Put-1

Data bus

source
32

Address bus

Opcode	Len. of data	x	x
8	8	8	8

destination
32

Opcode	dpid	Opcode1	Len of packet	offset
8	8	3	5	8

4. Get

Data bus

destination
32

Address bus

Opcode	Len. of data	x	x
8	8	8	8

source
32

Opcode	dpid	Opcode1	Len of packet	offset
8	8	3	5	8

5. Others

Data bus

All 0's
32

Address bus

Opcode	x	x	x
8	8	8	8

Figure 3.3 Ideal encoding of instructions.

Figure 3.3 shows the ideal encoding of instructions for the coprocessor. The main CPU needs to execute only two MOV instructions to transfer data for PUT and GET. One disadvantage of such encoding is the reduction in the address space of the main CPU since a lot of the addresses become unusable. The major disadvantage of this approach is that all the operands should be determined at compile time, so that the compiler can put the data in the above form for the CPU to use at run time. But this will not be the case always as most of the time the operands for the PUTs and GETs will be decided upon at run time and will be stored in the register file in the CPU. In this scenario, forming such a compressed word from that data in the registers will itself cause the CPU to do too much unnecessary work. Instead it will be much more efficient for the CPU to execute five MOVs in case of PUT or GET to transfer the five operands, one at a time. This is the approach that has been implemented as shown in Figure 3.4 in the actual encoding of instructions.

1. Register/ De-register

Data bus

Address
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

2. Set PID/NPROCS

Data bus

All 0's	Value
24	8

Address bus

Opcode[1]	x	x	x
8	8	8	8

3. Put – n / Put – 1

Data bus

destination address
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

source address

Opcode[1]	x	x	x

32

8	8	8	8

dpid	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Data length	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Offset	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

4. Get

Data bus

Destination address
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

Source address
32

Opcode[1]	x	x	x
8	8	8	8

dpid	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Data length	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Offset	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

5. Others

Data bus

All 0's
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

Figure 3.4 Actual encoding of instructions.

[1]The opcode enables Inst. and Data FIFO addressing, so that the opcode and corresponding 32-bit data are written to the Inst. and Data FIFO respectively.

[2]The opcode1 enables Data8 FIFO addressing, so that the 8-bit data is written to the Data8 Interface register. After every three writes to this register, a write is performed to the Data8 FIFO writing all the three 8-bit operands (dpid, Len and Offset) to the Data8 FIFO.

3.2 Packet Formats for Communication

The main task of the coprocessor is communication, so it sends and receives data for the PUT-1, PUT-n and GET instructions. The PUT-1 and PUT-n packets contain data, while the GET packet only contains information to enable the destination coprocessor to execute a PUT to the sending coprocessor's main memory. The corresponding packet formats are shown in Figure 3.5.

PUT-1 packet format:

dpid	Packet Type	Packet Length	offset	Destination address	Data
8	3	5	8	8	32
Header (32-bit)					Data

PUT-n packet format:

dpid	Packet Type	Packet Length	offset	Destination address	Data1	....	DataN
8	3	5	8	8	32	32	32
Header (32-bit)					Data		

GET packet format:

dpid	Packet Type	Packet Length	offset	Source address	myid	Packet Type1	data length	all 0's	Destination address
8	3	5	8	8	8	3	5	8	8
Header1 (32-bit)					Header2 (32-bit)				

Figure 3.5 PUT-1, PUT-n and GET packet formats.

The data length is implicit in the PUT-1 and PUT-n packets and is equal to Packet Length – 1. Thus the header for both the PUT instructions will be 1 word while that for the GET instruction will be 2 words.

dpid: This is the ID of the destination processor. An 8-bit ID allows addressing of $2^8 = 256$ distinct processors in the system.

Packet type:

Although a 2-bit field is sufficient to encode the Packet type, a 3-bit field allows addition of more packet types.

Table 3.2 Packet Type Encoding

Packet	**Packet type bits**
PUT-1	001
PUT-n	010
GET	011

Packet length:

This is the length of the packet in 32-bit word units. This 5-bit field allows a maximum packet size of 31. Thus each packet can carry a data payload of 30 words (leaving aside one word for header). This field is fixed to "2" for both the PUT-1 and GET packets.

Offset:

An 8-bit offset allows for indexing of arrays of size 256. This offset value is added to the destination address (source address) in the PUT (GET) instructions at the receiving processor.

Source, Destination Address:

This is a 8-bit global address. See the section on registration/deregistration for more details about these 8-bit global addresses.

Handling of GET made easy:

When a coprocessor receives a GET packet, it executes a PUT into the calling processor's memory. The packet format of GET is such that the Header of the PUT packet can be easily derived from it as shown in Figure 3.6.

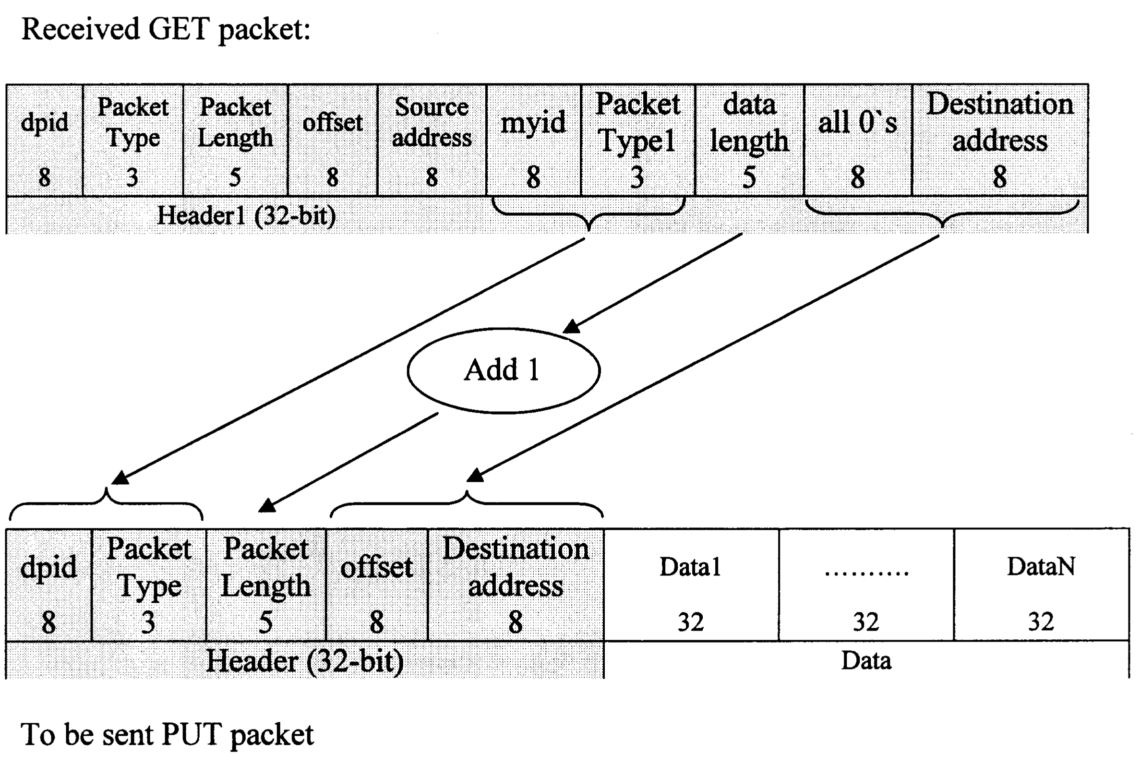


Figure 3.6 Formation of PUT packet from the received GET packet.

3.3 Coprocessor Pipeline Design

3.3.1 Coprocessor Pipeline

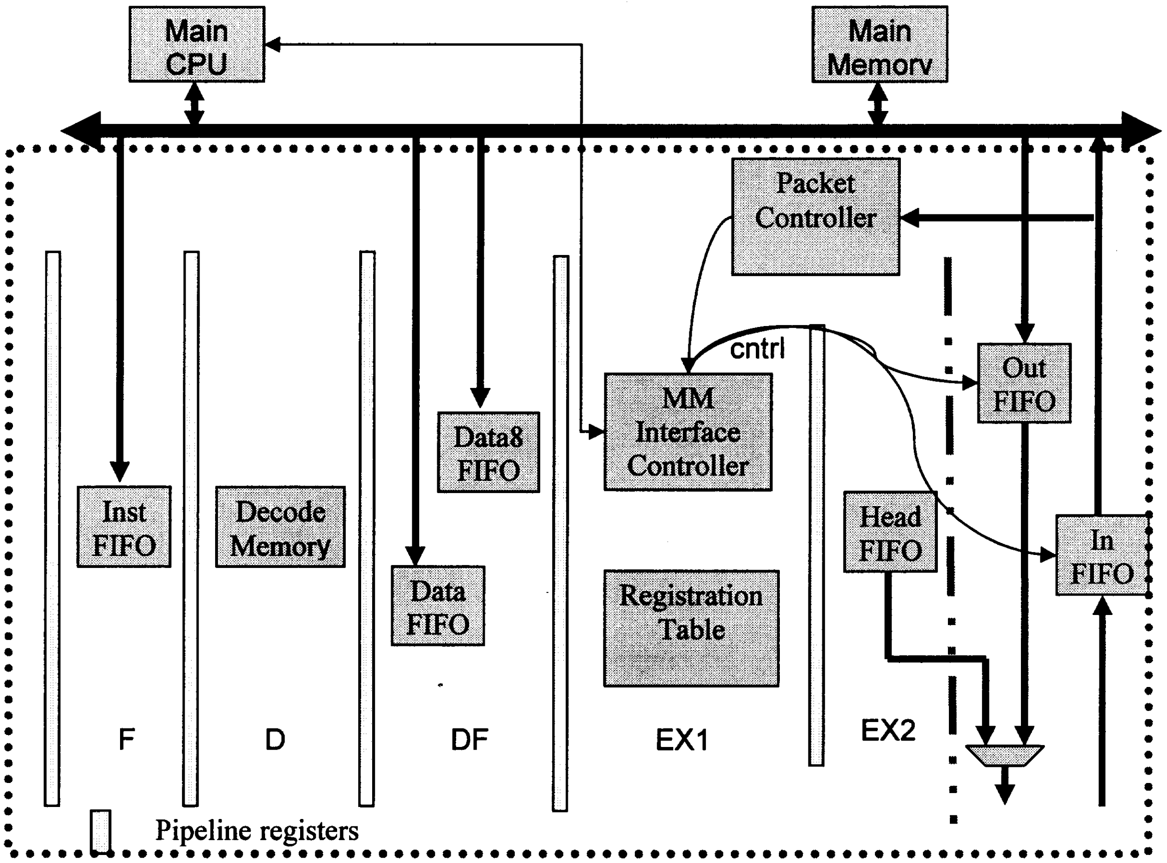


Figure 3.7 Coprocessor pipeline architecture.

The coprocessor is 5-stage pipeline as shown in the Figure 3.7. The five stages are: Fetch (F), Decode (D), Data Fetch (DF), Execute1 (EX1) and Execute2 (EX2).

Fetch (F): The instruction to be executed is read from the Inst. FIFO. Since execution is strictly sequential, no program counter is required.

Decode (D): A ROM decodes the instruction and generates the controls signals required for the further three stages

Data Fetch (DF): Data FIFO and/or Data8 FIFO are read to get the various operands for the 2 execute stages.

EX-1: This stage consists of the Registration Table and the Main Memory (MM) Interface controller.

EX-2: This stage consists of the Head FIFO.

3.3.2 Main Memory Interface Controller (MMIC)

The coprocessor needs to do two types of transfers with the main memory (MM).

- For a PUT instruction or a GET packet, it needs to transfer data from the MM to the coprocessor.
- For a PUT packet, it needs to transfer data from the coprocessor to MM.

During these transfers, the main CPU cannot access MM. Hence, it was desired that the coprocessor use the least MM time possible. Thus, this mechanism of MMIC, Out FIFO and In FIFO was designed. All the data that is written to the MM from coprocessor is present in the In FIFO while all the data that is read from MM is written to Out FIFO. Since both In FIFO and Out FIFO can be read and written to asynchronously, this basically isolates this MM interface from the rest of the coprocessor pipeline.

All the tasks for the MMIC are stored in a FIFO in the form of a triplet

{direction of transfer, length of data, 32-bit address}

direction of transfer = 0	From CPU to Out FIFO
direction of transfer = 0	From In FIFO to CPU

The MMIC gets access to the MM via a request-grant mechanism with the main CPU. Once it gets an access, it executes each task stored as a triplet. It continues executing tasks until there are no tasks left or the Main CPU asks for relinquishing the control of the MM. Even if it is in the middle of a transfer when the Main CPU asks for

MM control, MMIC suspends that transfer and re-starts from where it left off when it gets access to the MM next time. This way two objectives are achieved:

1. The coprocessor does the main memory transfers in burst style and does not try to get access to the MM on per packet or per instruction basis.
2. Also once granted access, it makes full utilization of the MM, if it has tasks pending or else it can just relinquish control back to main CPU.

3.3.3 Packet Controller

The Packet controller handles the packets that are sent to the coprocessor from other coprocessor's in the system. It basically needs to handle 3 types of packets: PUT-1, PUT-n and GET.

Tasks to be done on receiving a Put packet:

1. Read the Header.

dpid	Opcode	Len of packet	offset	dest.
8	3	5	8	8

2. Send the 8 bit global address to Registration Table (RT) to get 32-bit local address. Also calculate the Length of data transfer = Length of packet – 1
3. Add the offset to the 32-bit address got from RT.
4. Load this effective address, Length of data transfer and direction of transfer into the MM Interface controller (MMIC).
5. Wait until MMIC signals completion of transfer. When transfer complete then handle next packet.

Tasks to be done on receiving a Get packet:

1. Read the Header.

dpid	Opcode	Len of packet	offset	source
8	3	5	8	8

2. Send the 8 bit global source address to Registration Table (RT) to get 32-bit local source address.
3. Add the offset to the 32-bit address got from RT.

4. Read the next word of Header

myid	Len of data	all 0's	dest.
8	8	8	8

5. Send the 8 bit global dest. address to Registration Table (RT) to get 32-bit local dest. address.
6. Load the main pipeline with source addr, dest, addr, Len of data and dpid = myid and ask it to execute a Put.
7. Wait until pipeline signals successful start of execution of Put.
8. Handle next packet.

Further improvement in the throughput can be had if the header of the incoming packets can be stored in a separately from the data.

3.3.4 Flowgraphs

3.3.4.1 Flow graph for Put-1

Clock Cycle	F	D	DF	EX1	EX2
1	Fetch Put1 – Part1	-	-	-	-
2	Fetch Put1 – Part2	Decode Put1 – Part1	-	-	-
3	-	Decode Put1 – Part2	1. Read Data FIFO to get destination address. 2. Read Data8 FIFO to get Len, dpid & offset.	-	-
4	-	-	Read Data FIFO to get 32-bit data	Registrations Table Lookup[1]	-
5	-	-	-	Pass 32 bit data to next stage	Write Put1 header to Head FIFO
6	-	-	-	-	Write 32-bit data to Head FIFO*

[1]32-bit destination address is looked up in Registration Table Lookup to get 8-bit global address
* Execution of Put-1 is complete. The packet will be transmitted when all other packets ahead of it in the Head FIFO have been transmitted.

3.3.4.2 Flow graph for Put-n

Clock Cycle	F	D	DF	EX1	EX2
1	Fetch Putn – Part1	-	-	-	-
2	Fetch Putn – Part2	Decode Putn – Part1	-	-	-
3	-	Decode Putn – Part2	1. Read Data FIFO to get source address. 2. Read Data8 FIFO to get Len, dpid & offset.	-	-
4	-	-	Read Data FIFO to get source address	Registrations Table Lookup[1]	-
5	-	-	-	Load MM Interface controller[2]	Write Putn header to Head FIFO*

[1]32-bit destination address is looked up in Registration Table Lookup to get 8-bit global address
[2]MM Interface controller is loaded with the 32-bit source address, Length of data and the direction of data transfer
* Execution of Put-n is complete. When the MM Interface controller fetches the data block from memory, then the packet will be transmitted starting with the header from the Head FIFO and then the data part from the Out FIFO

3.3.4.3 Flow graph for Get

Clock Cycle	F	D	DF	EX1	EX2
1	Fetch Get – Part1	-	-	-	-
2	Fetch Get – Part2	Decode Get – Part1	-	-	-
3	-	Decode Get – Part2	1. Read Data FIFO to get destination address. 2. Read Data8 FIFO to get Len, dpid & offset.	-	-
4	-	-	1. Read Data FIFO to get source address 2. Read PID register to get myid	Registrations Table Lookup[1]	-
5	-	-	-	Registrations Table Lookup[2]	Write Get header – 1 to Head FIFO
6	-	-	-	-	Write Get header – 2 to Head FIFO*

[1]32-bit source address is looked up in Registration Table Lookup to get 8-bit global address
[2]32-bit destination address is looked up in Registration Table Lookup to get 8-bit global address
* Execution of Get is complete. The packet will be transmitted when all other packets ahead of it in the Head FIFO have been transmitted.

3.3.4.4 Flow graph for Register/ Deregister

Clock Cycle	F	D	DF	EX1
1	Fetch	-	-	-
2	-	Decode	-	-
3	-	-	Read Data FIFO to get Address.	-
4	-	-	-	Registrations Table Write/Delete*

* Register/Deregister complete

3.3.5 Barrier Implementation

Whenever the main CPU executes a barrier, the coprocessor must complete the present communication tasks and suspend operation till all the other processors have done the same. In our design execution of barrier in the pipeline enables the barrier mechanism. Implicitly it also means that all the instructions before barrier have been completed through the pipeline. But this does not mean that those instructions have been executed completely because of one of the following two reasons - the instructions may be waiting in the MM interface FIFO waiting for the data to be fetched from the main memory or they may be queued in the OUT FIFO waiting to be transmitted. Even if both these cases are false, still there maybe packets from other coprocessor enqueued in the IN FIFO waiting to be handled. Thus for successful barrier all these FIFOs must be empty and the packet controller must be idle. An assumption is made here that the main CPU does not load any execution into the coprocessor after the barrier instruction until the coprocessor has signal successful completion of barrier. Due to this assumption, the complete implementation of barrier is just the condition that all the FIFOs - INST. FIFO, DATA FIFO, DATA8 FIFO, MM INTERFACE FIFO, HEAD FIFO, OUT FIFO and IN FIFO, are empty and the packet controller is empty. The corresponding barrier implementation is shown in Figure 3.8.

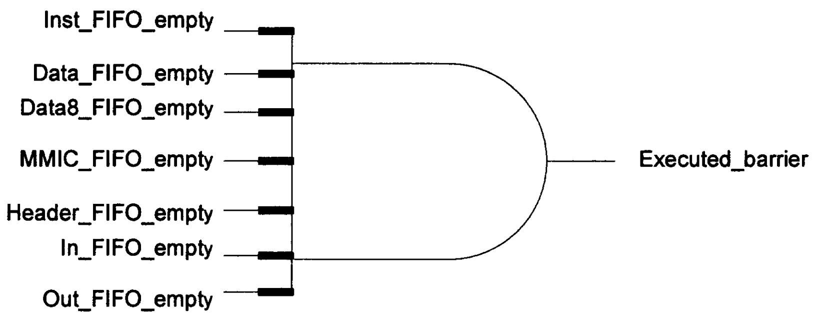


Figure 3.8 Barrier implementation.

Once the coprocessor executes barrier successfully, it asserts the EXECUTED_BARRIER signal and waits for the HAVE_OTHERS_DONE signal to be asserted. This indicates that all the other coprocessors have reached their barrier and a system wide barrier is complete. Each coprocessor now informs this to the main CPU by making the BARRIER_DONE signal true, which is acknowledged by the CPU by the BARRIER_DONE_ACK signal.

System Barrier

On the system level, a barrier can be implemented as a tree as shown in Figure 3.9.

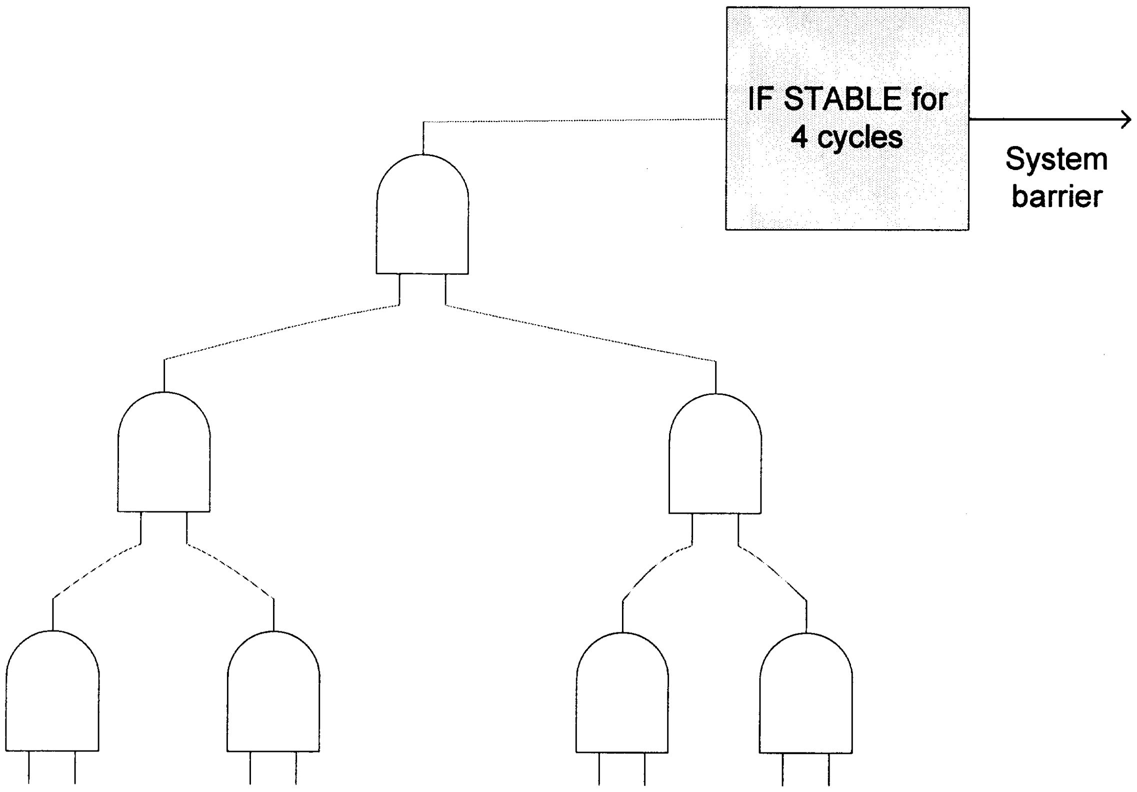


Figure 3.9 System barrier tree structure.

In this barrier not only the coprocessors, but also the routers have to be considered. This is because of the simple reason that packet might still be in the router, while all the coprocessors reach their barrier`s temporarily. So the barrier at the system level must take the router also into consideration.

CHAPTER 4

ROUTER DESIGN

The router (or routing element) is required to interconnect the coprocessors together to build a complete multi processor system. It basically consists of a packet switch using a crossbar. Crossbars are widely used in packet switching applications because of their non-blocking capability, simplicity and their market availability.

There are three different kinds of queuing in high performance packet switches:

- Input Queuing.
- Cross point Queuing.
- Output Queuing.

For comparison of the above three queuing disciplines, see [10, 14].

Head of Line [HOL] blocking is a common problem in Input buffered switches [5]. HOL refers to the situation where the packet at the head of the queue is blocked since its output is busy and this does not allow other packets to be sent to other unused outputs. It has been shown that HOL restricts the throughput to 58.6% [6].

A simple and elegant solution to overcome this problem is to use Virtual Output Queuing (VOQs) where a separate queue is maintained in each input for each output. A comprehensive Design and Evaluation of the Combined Input and Crossbar Queued (CICQ) Switch can be found in [14]. [3] proposed a Combined Input-crosspoint buffered (CIXB-1) switch model, where the crosspoint buffer has one-cell size, with VOQs at the inputs and simple round-robin for input and output arbitration. It showed that the combination of input buffers and single-cell crosspoint buffers and a round-robin arbitration scheme provides 100% throughput under uniform traffic. The implementation here is "Combined Input-Cross point-Output Buffered" architecture as described in [2]. This architecture provides 100% throughput under under uniform and unbalanced traffic. [22] describes a similar network router, which has a bufferless crossbar.

4.1 Input VOQ port

The basic organization of the input VOQ port is shown in Figure 4.1.

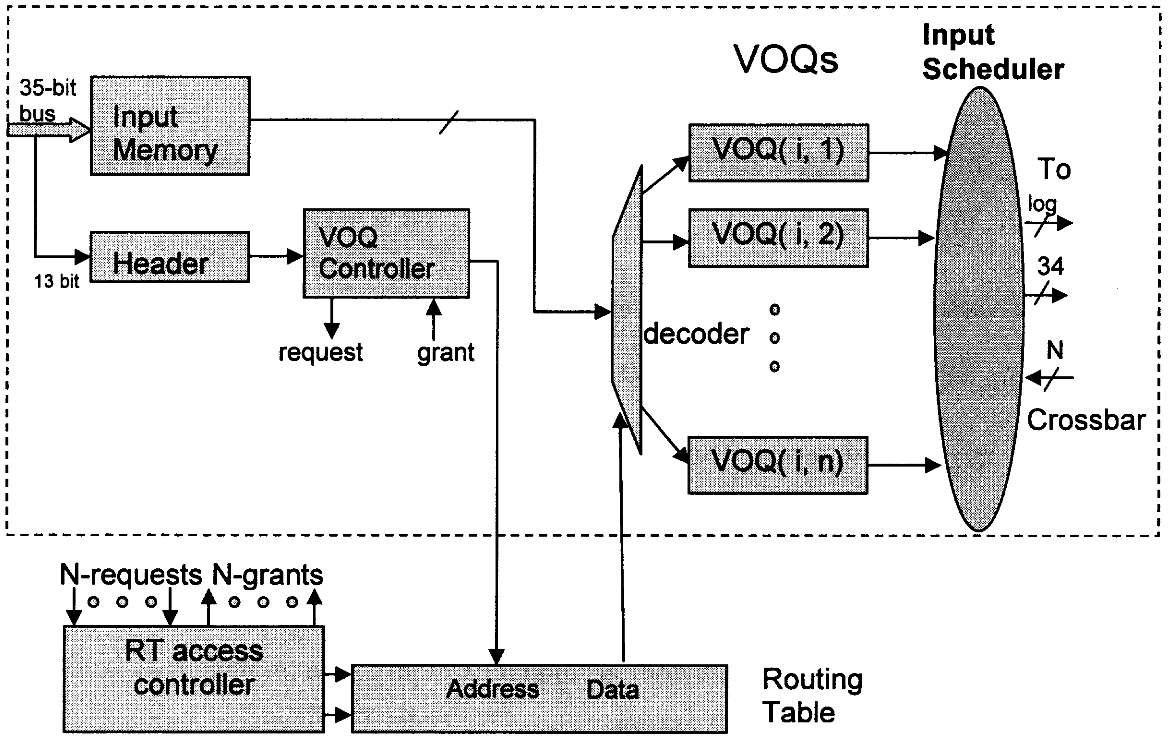


Figure 4.1 Input VOQ port architecture.

Input Memory

This is a FIFO structure which is used to buffer the incoming packets.

Header FIFO

The Header FIFO is used to store the destination address and length field of the header of each packet. This separate storage on header enables the VOQ controller to do the routing lookup independent of the congestion in the Input memory to VOQ path. Using the header of the next packet from this FIFO, the VOQ controller can go ahead with the lookup even if the present packet is still being transmitted to the respective VOQ FIFO. The length field stored for each packet is used to correctly transfer those many number of words form the Input memory to the VOQ FIFO.

VOQ controller

There two functions of the VOQ controller are –

1. To communicate with the Routing Table and get the destination address of the current packet and
2. To control the packet transfer from the INPUT FIFO to the respective VOQ FIFO once the routing lookup has been done.

VOQs:

There are N VOQs in each input port. Each VOQ holds the packets destined for a particular output. VOQ (x, y) holds the packet in Input card 'x' destined for output 'y'. Each VOQ is also a FIFO[1] structure.

Routing:

A simple RAM based routing lookup has been used although all care has been taken in the input card design, so that any fancy routing strategy can be incorporated.

The controller and the RT access controller have an asynchronous request-grant interface. The RT access controller and Routing Table are common for the entire line card in one routing element.

Arbitration:

Round robin arbitration is used to select a non-empty VOQ to send a packet to the cross point. An eligible VOQ is one which is non-empty and for which the corresponding cross point buffer is not full. The status of the cross point buffers is determined by the feedback mechanism from the crossbar. Other arbitrations schemes are described in [7], [8].

Aids for resource Management:

The user can configure the following parameters:
1. Size of input memory.
2. Size of each VOQ buffer.
3. Size of Output memory.

A 35-bit bus called DATA_TRANSFER_BUS used for all the data transfers is shown in figure 4.2 and the components of this 35-bit bus and their functions are described in Table 4.1. This is the transfer medium to and from the coprocessor and also to and from the router. This bus always drives a FIFO at the receiving side. The Data_valid indicates that the data on the bus is valid and is stored on the next rising Clock edge. The Full signal completes the feed back loop indicating to the sender that the receiver has run out of FIFO memory ands hence the sender should stop sending.

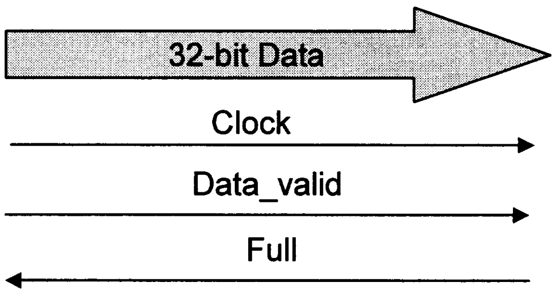


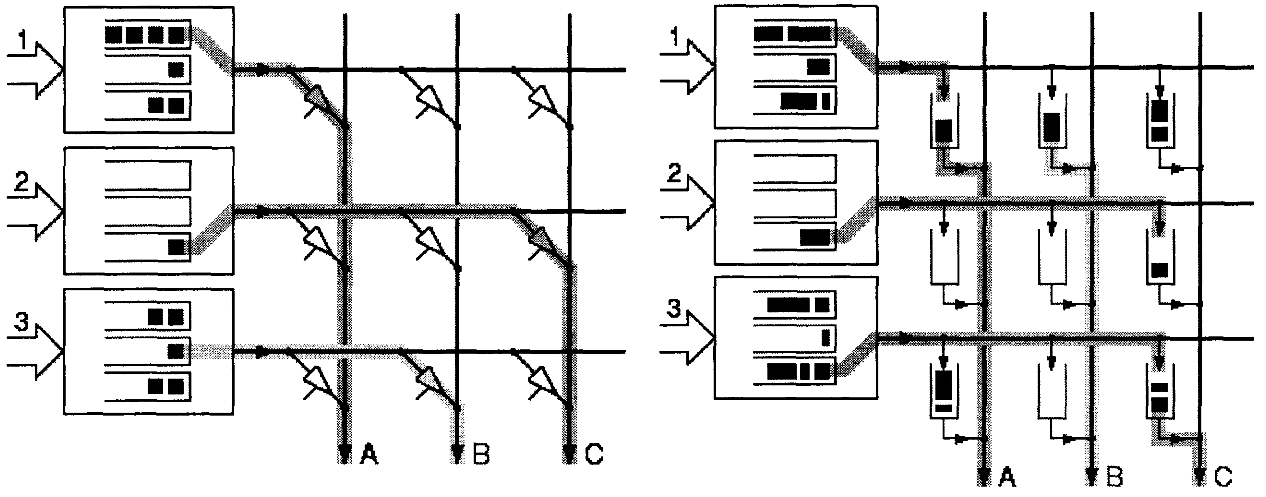
Figure 4.2 DATA_TRANSFER_BUS

Table 4.1 DATA_TRANSFER_BUS Components and Functions

Signal	**Width**	**Direction**	**Purpose**
Data	32	Input	Carries the 32-bit data.
Clock	1	Input	The data is transferred at the positive clock edge.
Data_valid	1	Input	Indicates if the data on the bus is valid.
Full	1	Output	Full=0 indicates that the receiver is ready to receive data and indicates to the sender to start sending. Full=1 means that the receiver is not ready to receive data and indicates to the sender to stop sending.

4.2 Crossbar Design

The crossbar is the most frequently used switching element topology. It offers simplicity and non-blocking operation. However, *when bufferless*, it also requires a centralized scheduler, which must simultaneously satisfy --in each cell time-- all input and all output link constraints [9]. The cost and complexity of this scheduler increases considerably for short cell times and for large switch sizes. Furthermore, bufferless crossbars can only be efficiently used with fixed-size cells arriving from mutually-synchronized line cards; when we need to switch variable-size packets, we must first segment them into fixed-size cells. To compensate for the inefficiencies of scheduling and of packet segmentation, internal (crossbar) speedup is used; commercial crossbars often use a speedup factor of 2 to 3. The net effect is to limit the maximum external line rate to roughly one half to one third the peak achievable crossbar line rate.



In a bufferless crossbar, the scheduling decisions at the input and output ports all depend on each other: each output can only be paired to a single input and conversely for the inputs

(a)

Small buffer memories at the crosspoints allow distributed scheduling decisions; operation with variable-size packets now becomes feasible

(b)

Figure 4.3 3 x 3 Crossbar (a) Unbuffered, (b) buffered [9].

Buffered crossbars go back to a 1982 patent [12]. Buffered crossbars have the following advantages over non-buffered ones -

- Avoids Segmentation and Reassembly (SAR).
- Crossbar can run at input line rate.
- Much simpler and very efficient crossbar scheduling.
- Independent Input and Output clock domains.

The only disadvantage of the buffered crossbars is that they are memory intensive - N x N switch requires N^2 cross point memories. For more information on buffered crossbar design see [2], [4]. A Xilinx application note [13] also describes a buffered crossbar.

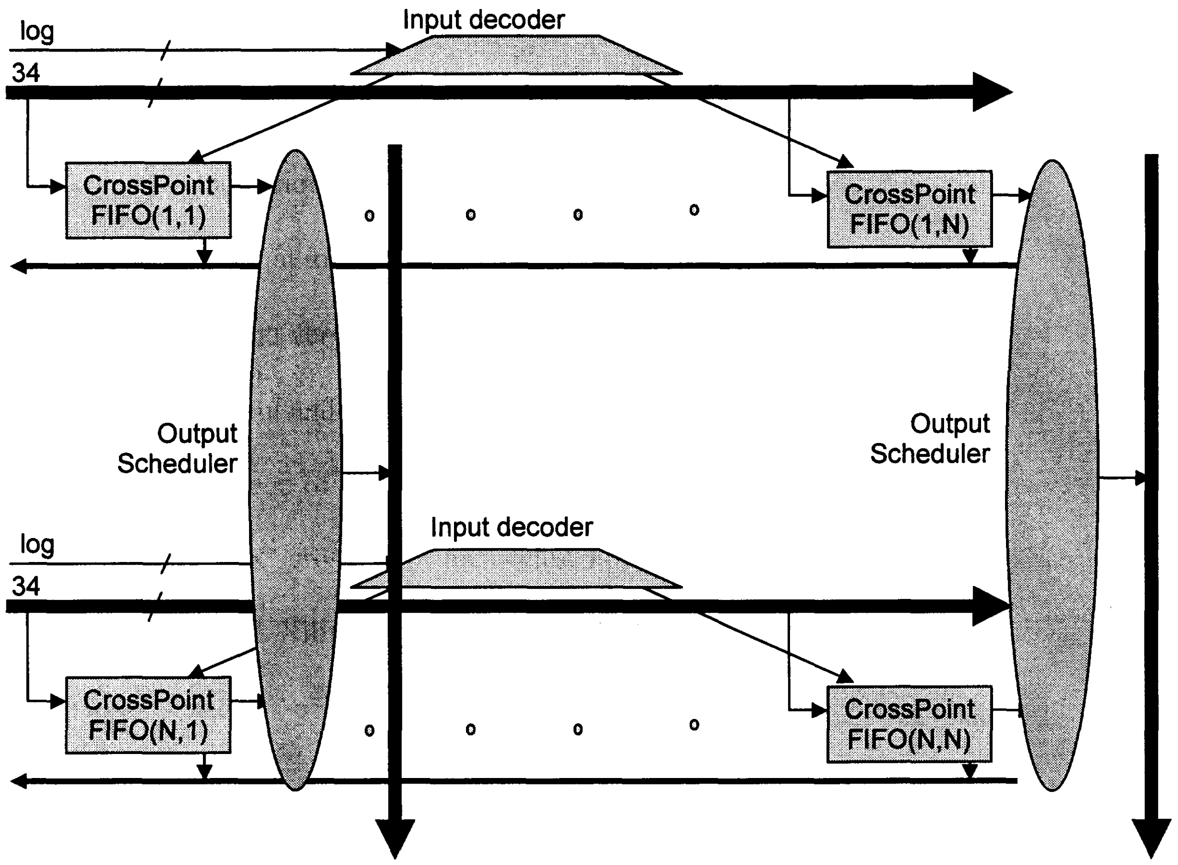


Figure 4.4 Crossbar architecture.

The crossbar consists of N^2 cross points arranged in N rows by N columns as shown above. Each cross point is a fully asynchronous FIFO[1]. Due to this the input and output domains in the cross bar can run independently.

4.2.1 Input Domain

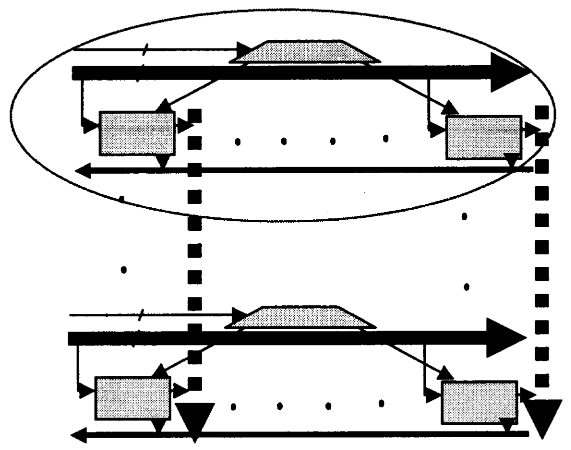


Figure 4.5 Crossbar – input domain.

Packet going from input i to output j is transmitted from Line card i to cross point (i, j). Line Card i is connected to all the cross points in row i. The scheduler in the line card selects the cross point to which the next packet is to be delivered depending on the destination port of the packet andthe availability of space the corresponding cross point. Before transmitting the packet, the line transmits the address of the destination cross point. This address goes to the decoder of that row, where it enables only one cross point and disables all others. The input VOQ port transmits the packet which gets stored in the desired cross point. Also the status of the cross point is sent back to the input VOQ port so that the scheduler in the input VOQ port can do correct scheduling.

4.2.2 Output Domain

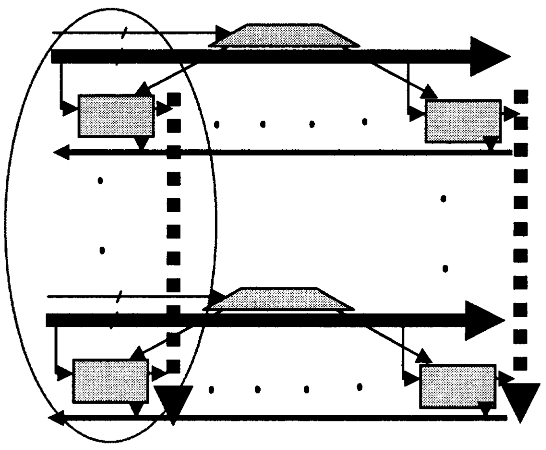


Figure 4.6 Crossbar – output domain.

All the cross points in a column j are connected to output j. Any packet stored in a cross point in column j is destined for output j.

4.2.3 Output Arbitration

An output scheduler selects one cross point from all the cross point belonging to the same column which are non-empty. Round robin arbitration is used to select a non-empty cross point to send a packet to the output.

[1]See the section on FIFO for more details about the FIFO implementation

Figure 4.7 shows complete Router architecture.

Finally, [15] gives a list of literature on this topic of Input queued packet switch design.

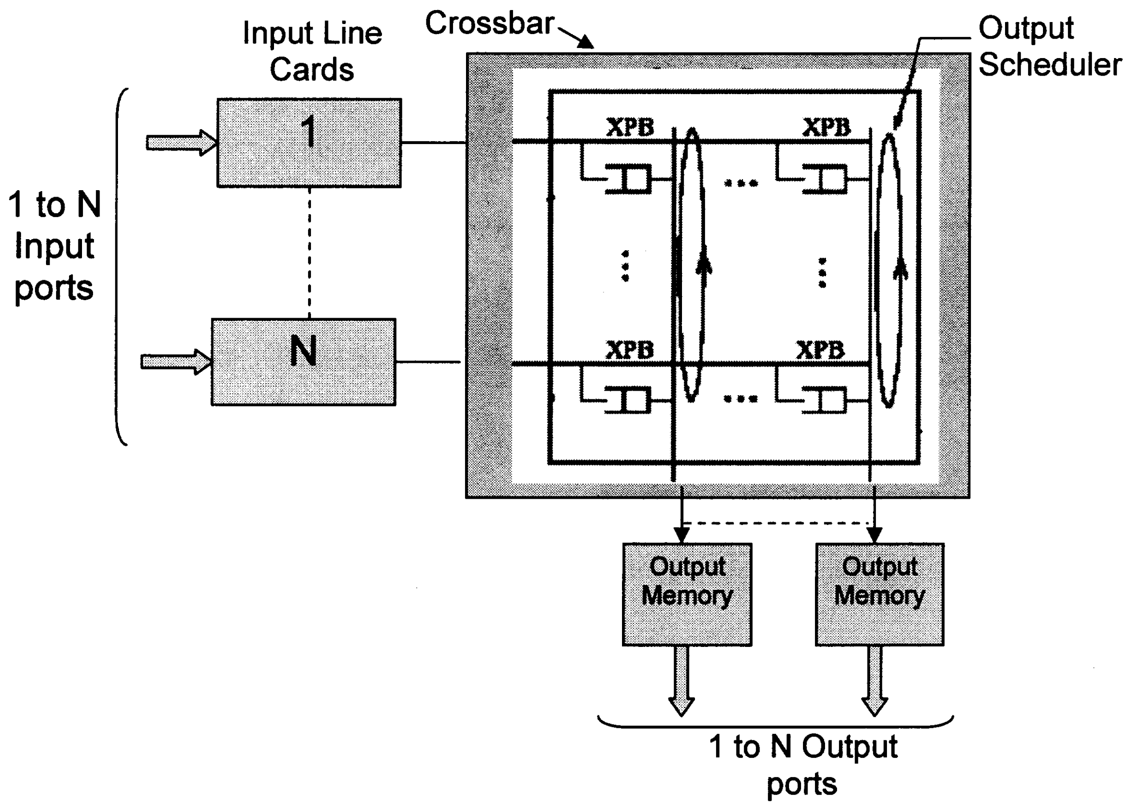


Figure 4.7 Routing element architecture.

4.3 Barrier

In order to implement system wide barriers as outlined in section 3.3.5, a mechanism must be built to indicate the presence of packets in the router. The packets in the router can either be in one of the VOQ ports or the crossbar. If the packet is in the VOQ port it must be in the INPUT MEMORY or one of N VOQ FIFOs. A flag ALL_FIFOs_EMPTY indicates the condition when all these FIFOs empty. A similar flag ALL_FIFOs_EMPTY, which is output by the crossbar indicates that no packet is enqueued in the crossbar. Finally a simple AND of all these signals generate the ROUTER_BARRIER signal which is used for system level implementation of barrier.

4.4 FIFO

FIFO stands for First-In-First-Out structure. It is analogous to the "queue" data type in higher level languages. An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other [11]. All the FIFOs used in this design are asynchronous and the term FIFO refers to asynchronous FIFO throughout.

Also each FIFO is fully user configurable in terms of the following parameters:

1. FIFO Size.
2. Data Width.
3. Choice of Implementation (distributed or Block Select RAM).

Implementation:

The various FIFOs used in this entire design have been implemented (synthesized) in 2 ways depending on the size. FIFOs can be implemented in two styles in Virtex II hardware blocks - using the distributed dual port RAM feature or using true dual port block SelectRAM+ feature. Figure 3.5 shows the comparison of the space required for both the styles for FIFO of width 32 bits. In the case of BRAMs, each of FIFO also requires one block RAM. Figure 3.6 shows the maximum frequency of write (wclk) and read (rclk) clocks for each synthesis. FIFOs > 128 words deep use more than 1000 LUTs which is very high, and also their read clock frequency drops sharply below 150 MHz. Using BRAM for FIFOs < 128 wastes resources since the same BRAM having a capacity of 512 is partially used and the only variation is the small increasein LUTs used to implement the read and write counters. Therefore, FIFOs less than 128 words deep are implemented using the distributed dual port RAM feature and while those equal to or greater than 128 wrods deep are implemented using true dual port block SelectRAM+ feature.

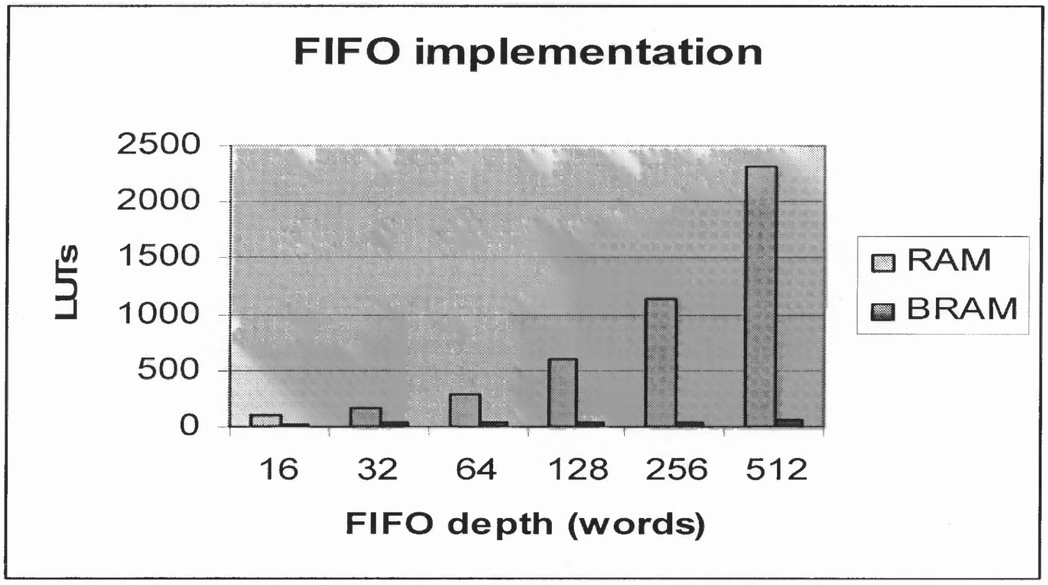


Figure 4.8 FIFO implementation space comparison.

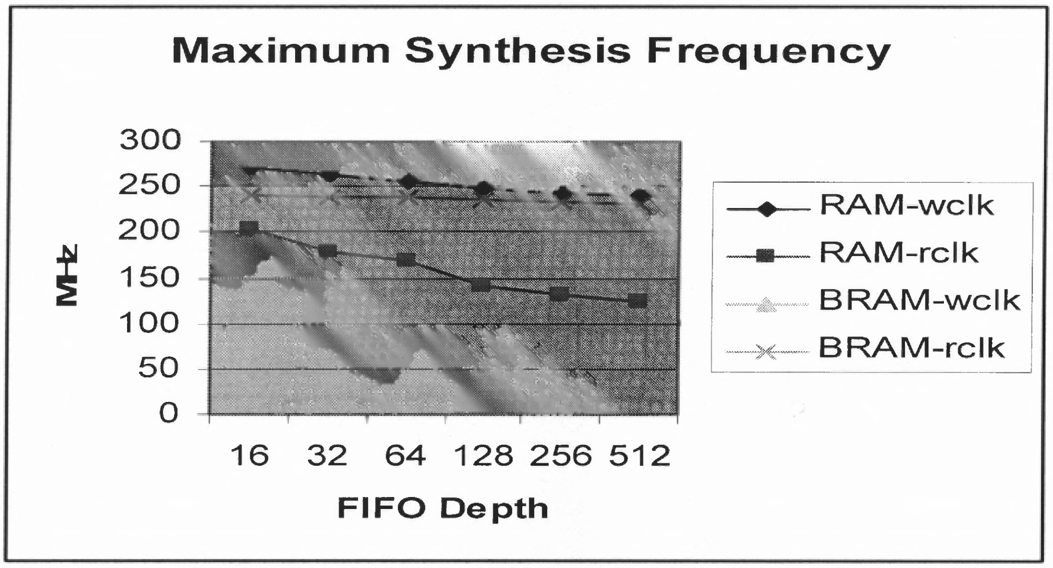


Figure 4.9 Maximum frequency of synthesis for RAM implementation.

CHAPTER 5

IMPLEMENTATION AND RESULTS

5.1 Design Flow and Implementation

The coprocessor is designed using the VHDL hardware description language. Also, the design of the router and the glue logic to interface the LAD bus is done in VHDL. During this design, different tools at various levels of integration are used. We have followed the standard Xilinx design flow in generating the complete system. Figure 5.1 shows the basic steps in the Xilinx standard design flow.

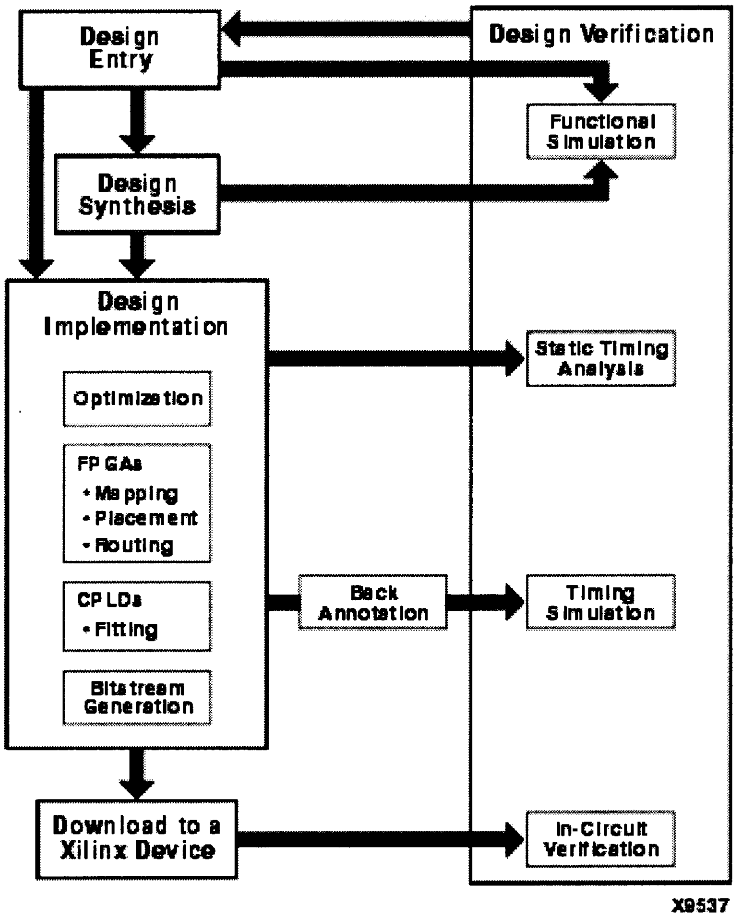


Figure 5.1 FPGA design flow [24].

The following are the steps followed in the FPGA design flow:

1. The design of all modules required by the coprocessor and the router design is done using a synthesizable subset of the VHDL language. The coding and compilation are done using the Mentor Graphics Modelsim simulator.

2. The functional simulation is performed using the Modelsim simulator. Many test benches are developed to test the coprocessor design using simulation. All the instructions for the coprocessor are tested using test benches. The correct and complete routing of packets by the router is tested by test benches under various load conditions.

3. These VHDL files are given as input to the Synplify Pro synthesis tool. During synthesis the behavioral description in the HDL file is translated into a structural netlist and the design is optimized for the Xilinx device XC2V6000. This generates a netlist in the EDIF (Electronic Design Interchange Format) and VHDL formats.

4. The output VHDL file from the synthesis tool is used to verify the functionality by doing post synthesis simulation using the Modelsim simulator.

5. The netlist EDIF file is given to the implementation tools of the Xilinx ISE (6.3i). This step consists of translation, mapping, placing and routing, and bit stream generation. The design implementation begins with the mapping or fitting of the logical design file to a specific device, and is complete when the physical design is completely routed and a bitstream is generated. Timing and static simulations are done to verify the functionality. This tool generates an X86 file which is used to program the FPGA.

6. Then a program in the C language is used. In this program different standard API functions available by Annapolis Microsystems are used for communication between the host system and the board. During execution of this program the host CPU programs the FPGA using available X86 format file, writes the program data to the coprocessor(s) in the system, resets the whole board and after finite given delay, reads the results back. These results are compared with the required for correct functionality of the whole system.

All these steps are followed in a general design methodology to program the FPGA. A small change in VHDL for correct execution leads to again start the design cycle from scratch. This is done till we get the correct results.

5.2 Coprocessor Test Module

The coprocessor test module was designed to interface the coprocessor to the LAD bus on the WILDSTAR-II board and also to facilitate interconnection between many such modules to test a multi-coprocessor configuration. A coprocessor test module connected to the LAD bus is shown in Figure 5.2.

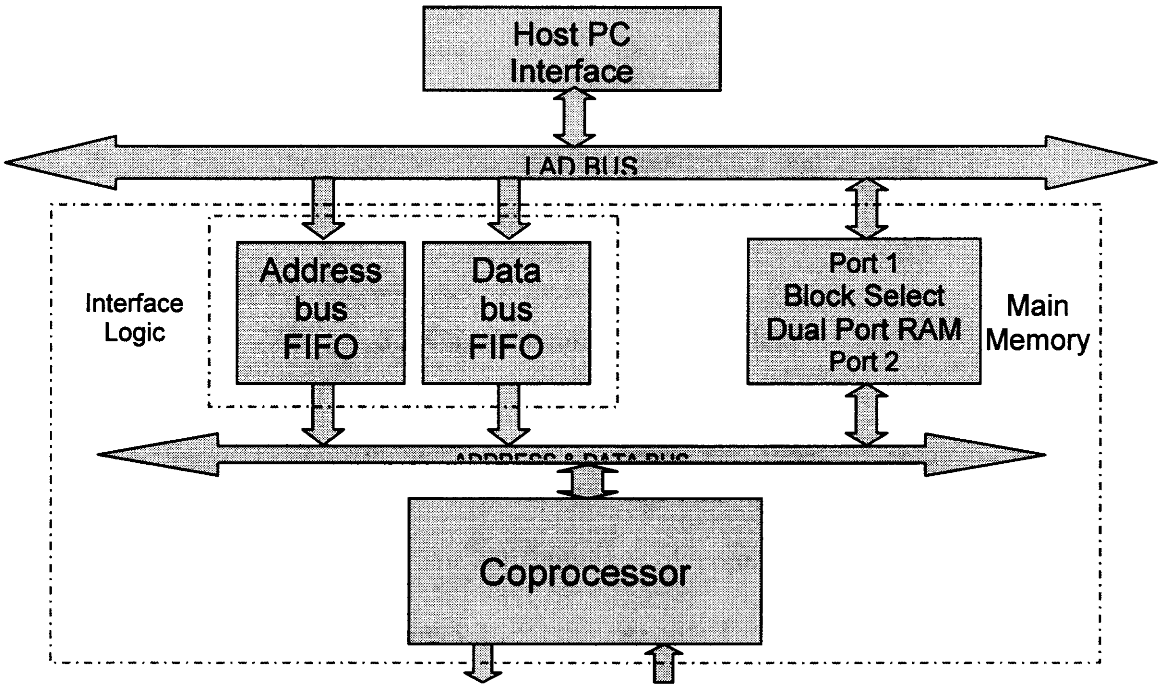


Figure 5.2 Coprocessor test module.

A dual port Block Select+ RAM is used as the main memory with one port connected to the coprocessor via the coprocessor address and data (CAD) bus and the other port connected to the host via the LAD bus. The main memory size is 512 and is mapped to CAD bus addresses 0 to 511. The interface logic consists of two FIFOs – address bus FIFO and data bus FIFO, each 512 words in size. These two FIFOs are used to emulate the execution of a MOV command to transfer opcodes to the coprocessor. The host program writes the 32-bit MOV destination address into the address bus FIFO and the corresponding MOV data into the data bus FIFO. The control logic (not shown in Figure 5.1) reads both these FIFOs simultaneously to the CAD bus and generates the necessary write signal, thus emulating the execution of MOV. The control logic also serves as an arbiter for the CAD bus because the CAD bus is used for two purposes – execution of "virtual" MOVs from the interface FIFOs to the coprocessor and data transfer between main memory and the coprocessor. A global system enable controls the interface logic. This global system enables ensures that all the coprocessors in a system start their program execution simultaneously, emulating an actual processor-coprocessor node as closely as possible. The control register is used to triggers the global system enable. A 10-bit system counter (not shown in Figure 5.1) is used for debugging and timing analysis purposes. The address bus FIFO, the data bus FIFO, the main memory and the system counter are memory mapped into the LAD bus memory map and the Table 5.1 shows the corresponding memory map. Each CTM has a base address which is generic and all the others have their addresses relative to the base address. Using this base address relative addressing scheme, a number of such CTM modules can be connected to the LAD bus to build a multi coprocessor system.

Table 5.1 CTM Memory Map

Module	**Address (hex)**
Control register	0050
System counter	Base_address[1] + 100
Main memory	Base_address[1] + 200
Address bus FIFO	Base_address[1] + 400
Data bus FIFO	Base_address[1] + 600

[1]Base_address is always a multiple of 1000

The steps for user program execution from the host machine are:

1. Write the program$^{\#}$ using simple Perl subroutines and compile it to the output text file.

2. Read the text file in a C program and copy the address words in an address array while the data words into a data array.

3. Write the address array to the address bus FIFO using the WSII C-API call Write_Reg32().

4. Write the data array to the data bus FIFO using the WSII C-API call Write_Reg32().

5. Start execution

6. Wait for execution to complete.

7. Read back the results from the BRAM using WSII C-API call Read_Reg32().

Steps 1- 4 are repeated for each coprocessor in case more than one exist in the system.

5.3 One Coprocessor "Loop-back" System

An experimental one processor "loopback" system is shown in Figure 5.3. The "loop back" here refers to the fact that the output of the coprocessor is connected to input of the same coprocessor. This way any packet sent by the coprocessor is received by the same coprocessor and executed as if it was sent by some other coprocessor. This configuration was primarily used for debugging and to collect timing information about execution of various instructions on the whole. In this configuration, a coprocessor test module is configured to base address = 0000 and the output port is connected to the input port. Also, another block RAM is connected to this port, to enable watching of the packets sent and received by the coprocessor. The other port of BRAM is connected to the LAD bus and is mapped to address 0x800. The system counter was configured to stop counting as soon as the coprocessor executes a barrier.

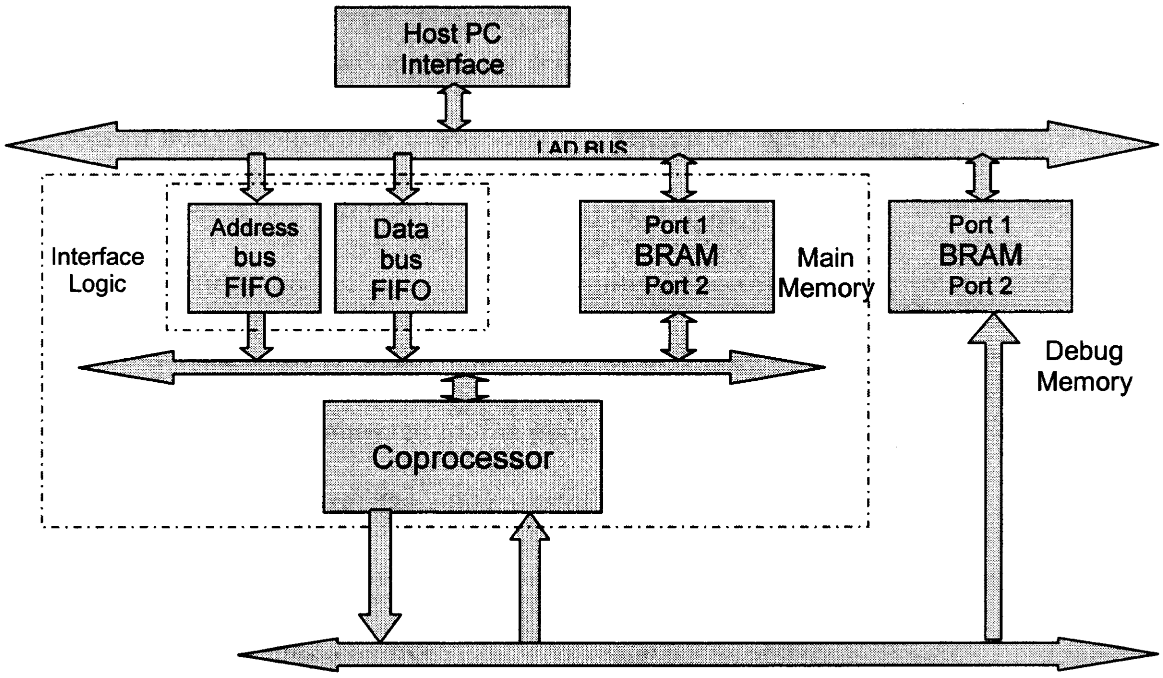


Figure 5.3 One coprocessor "loop-back" configuration.

Using this configuration one can completely debug the complete coprocessor with all its various interfaces like the coprocessor – main memory interface, the input interface and the output interface. The frequency of operation is 50 MHz.

5.4 Two coprocessor system

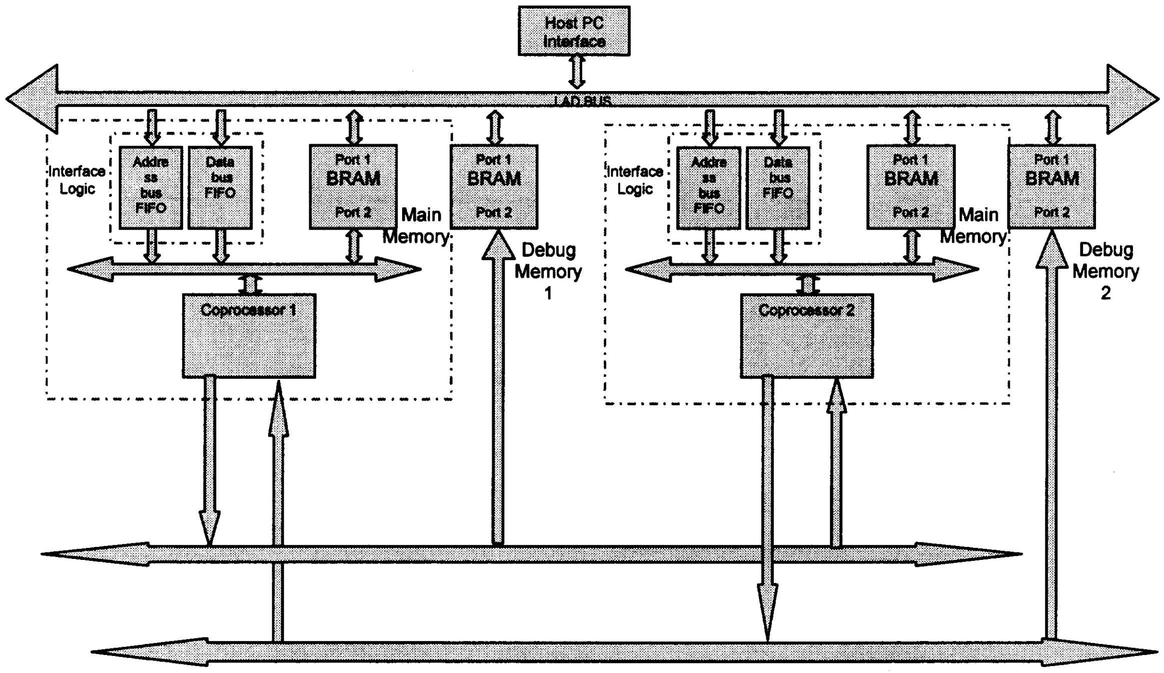


Figure 5.4 Two coprocessor test system (TCTS).

Using the coprocessor test module (CTM) a two coprocessor test system (TCTS) is built as shown in Figure 5.4. The base address of coprocessor 1 and 2 are 0x0000 and 0x1000 respectively. Table 5.2 shows the resulting memory map of this system. The packets being sent and received can be monitored using the Debug Memory 1 and 2. The system frequency is 50 MHz. This system can be converted to a complete 2 processor system by replacing the interface logic by a soft CPU like Microblaze and this will be part of the future work.

Table 5.2 TCTS Memory Map

Module	Address (hex)	
	Coprocessor 1	Coprocessor 2
Control register	0050	
System counter	0100	1100
Main memory	0200	1200
Address bus FIFO	0400	1400
Data bus FIFO	0600	1600

The host C program loads the program into the Address bus FIFO and Data bus FIFO of each CTM one by one. Then it triggers the system enable, starting the execution of the program on both the processors simultaneously. The system counter also starts counting at the same time and it stops when the barrier is reached, thus giving the total time required for that run.

In this configuration many single instructions were executed to get the time required for the completion of individual instruction. In each case, the single instruction was executed on coprocessor-1 followed by a barrier, while only a barrier was executed on coprocessor-2. Table 5.3 shows the time for execution in number of clock cycles to barrier. Note that these are the number of clock cycles from the time the first word of an instruction is written into the coprocessor.

Table 5.3 Single Instruction Execution Time

Instruction	Number of clock cycles to barrier	Time to barrier at 50 MHz
Barrier	8	0.16 μs
Get-8[1]	52	1.04 μs
Registration	10	0.2 μs
Deregistration	11	0.22 μs
Put-8[1]	38	0.76 μs

[1]Get-8 and Put-8 refers to transfer of 8 words of data.

Table 5.4 GET, PUT Execution Time

	PUT-n	GET-n
n	Number of clock cycles to barrier	Number of clock cycles to barrier
n=1	31	44
n=2	32	46
n=4	34	48
n=8	38	52
n=16	46	60
n=30[2]	59	73

[2]The maximum data that can be sent in one packet is 30 words.

The time of execution of GET and PUT requires a little more thought. Since these instructions work between two processors, this is the total time required for the execution on both the processors. Table 5.4 shows that GETs are much more expensive than PUTs.

Time for PUT = Time for PUT packet formation in coprocessor 1 + communication time + time for Put packet handling by processor 2.

Time for GET = Time for GET packet formation by processor 1 + communication time + time for GET packet handling and the execution of PUT by coprocessor 2 + communication time + Time for PUT packet handling by processor 1.

Thus, even if we make the communication time = 0, the overhead for GET is much more than that for PUT. This is inherent in the nature of the 2-way GET operation as compared to a 1-way PUT operation.

Total exchange

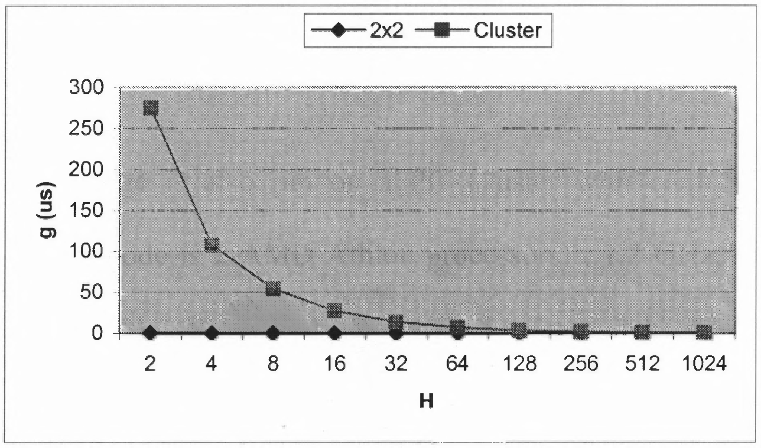
In this test pattern, each coprocessor sends "h" words of data to each other coprocessor including it. The total time required for this pattern to realize is measured. From this an estimate of the effective communication bandwidth is made. The total exchange is realized by a series of PUT-n instructions followed by a barrier at the end.

The total time required is denoted by 't'. The size of the data 'H' is chosen as a multiple of 2 and total exchanges upto 1024 words of data are performed. The parameter 'g' is then calculated as: g = t/(2*H). This gives the time required to transfer one 32 bit word. The reciprocal of g then gives the effective bandwidth of the system for such communication. Table 5.5 shows the result of this experiment.

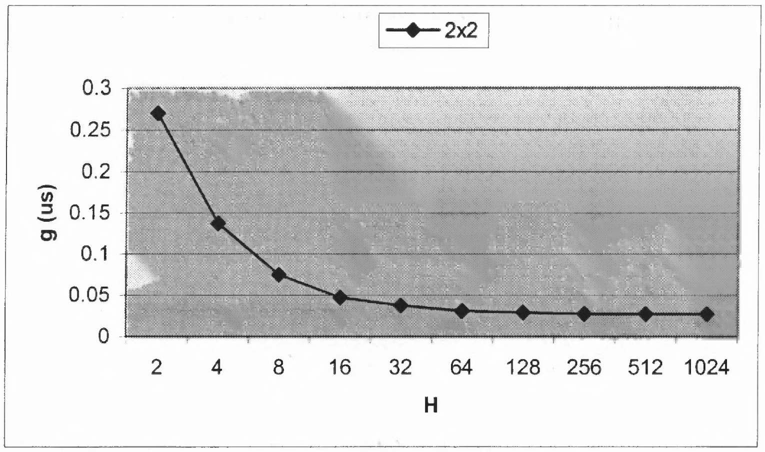
Table 5.5 2x2 Total Exchange Time

H	Time for total exchange		g (µs/32-bits)
	in clock cycles	in µs	
1	53	1.06	0.53
2	54	1.08	0.27
4	55	1.1	0.1375
8	60	1.2	0.075
16	76	1.52	0.0475
32	121	2.42	0.037813
64	198	3.96	0.030938
128	372	7.44	0.029063
256	701	14.02	0.027383
512	1396	27.92	0.027266
1024	2771	55.42	0.027061

The same total exchange is also run on a PC Cluster with configuration – 9 dual processor nodes (each node is 2 AMD Athlon processors) , 1.2 GHz, 64K L1 cache & 256K L2 cache Athlon, 1 GB per node, 1 Gb/s Ethernet switch). The total exchanges were run on 2 nodes, utilizing only one CPU in each node. The code for the total exchange was that provided by Gerbessiotis and Lee [1] on their website. The code is run under LAMMPI. Figure 5.5 shows the comparison of the parameter 'g' for the cluster with that our 2-coprocessor system (2x2).



(a)



(b)

Figure 5.5 (a) Comparison of "g" for the 2x2 system and the cluster (b) "g" for 2x2 system.[1]

[1]Plotted separately to give details of magnitude of "g" in 2x2 case which is not clear in (a).

Other Observations:

For each Put-n instruction, the main CPU executes 5 MOV instructions. In other words, the main CPU has to do 5 MOV operations to transfer 30 words of data. Table 5.6 shows the number of packets and the corresponding number of MOVs required for different data sizes. In order to transfer 1 kB of data, the main CPU executes 175 MOV instructions which are redundant. Hence, this suggests the addition of an instruction like PUT-1000, (and a symmetric GET-1000) which tells the coprocessor to transfer data in multiples of 1 kB. The coprocessor may segment this request into smaller number of packets.

Table 5.6 PUT-n MOV Count

Size of Data	**Number of packets or Put-n instructions**	**Number of MOVs**
8, 16	1	5
32	2	10
64	3	15
128	5	25
256	9	45
512	18	90
1024	35	175

5.5 Four Coprocessor System

Four coprocessor test modules (CTM) are interconnected using a 4x4 router to build a 4-coprocessor system as shown in Figure 5.6. Refer Appendix A for the configuration of the CTM and the router. The same total exchange is run in this case on this system and the comparison of a similar run on the cluster is shown in Figure 5.7.

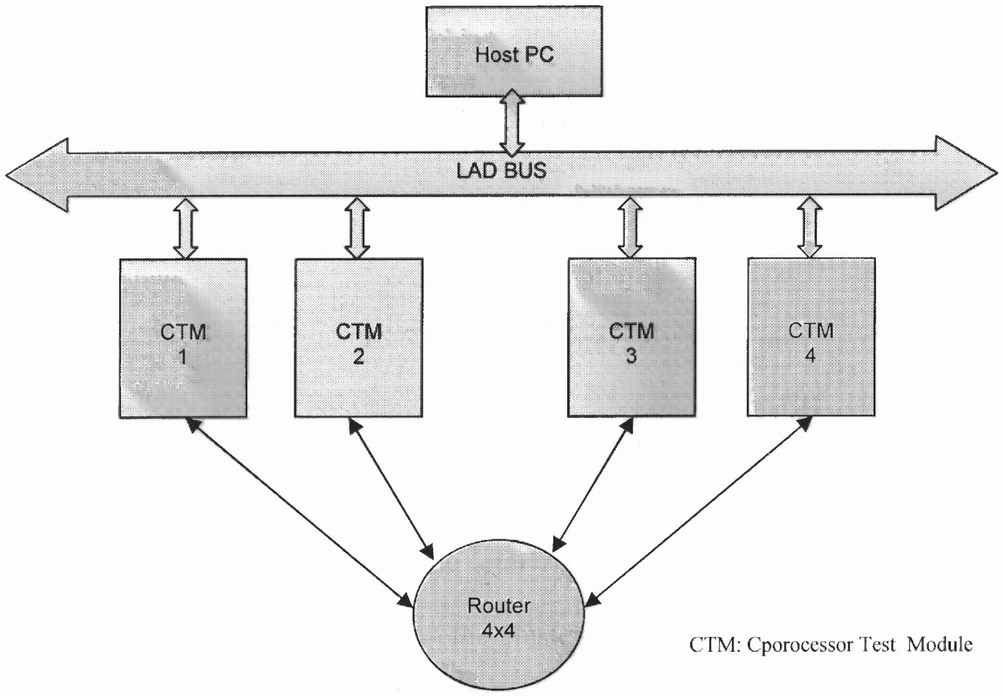
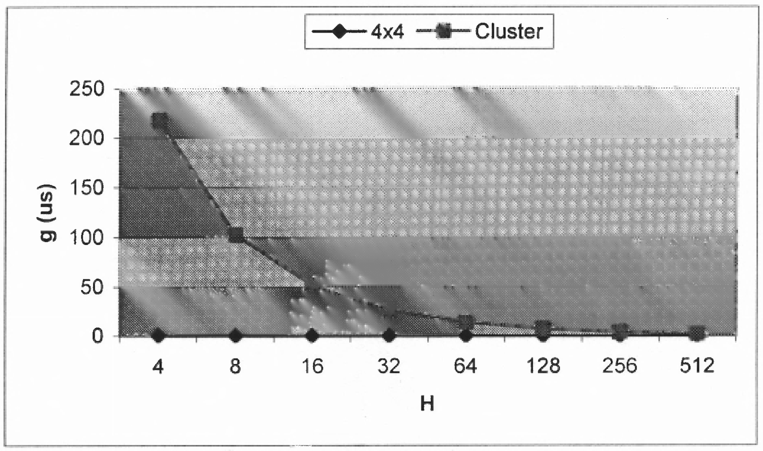
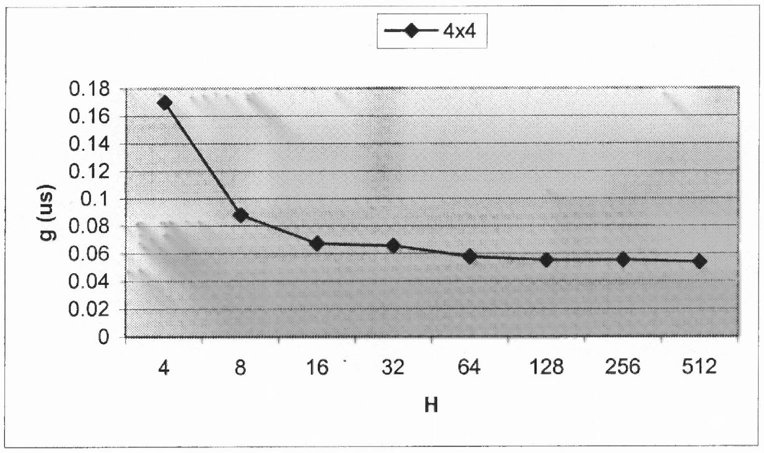


Figure 5.6 Four coprocessor system.



(a)



(b)

Figure 5.7 (a) Comparison of "g" for the 4x4 system and the cluster (b) "g" for 4x4 system.[1]

[1]Plotted separately to give details of magnitude of "g" in 4x4 case which is not clear in (a).

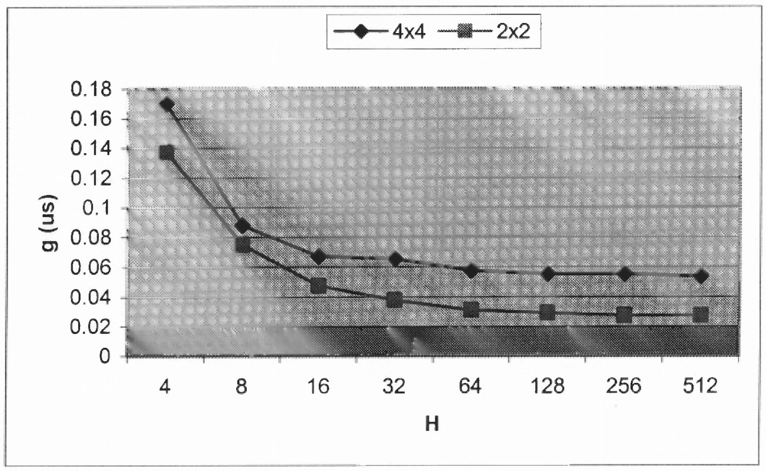


Figure 5.8 Comparison of "g" for the 2x2 and 4x4 system.

Figure 5.8 shows the comparison of the total exchange time for the 2x2 and 4x4 system. The time for 4x4 is greater than that for 2x2 due to the added communication latency due to the router.

5.6 Eight Coprocessor System

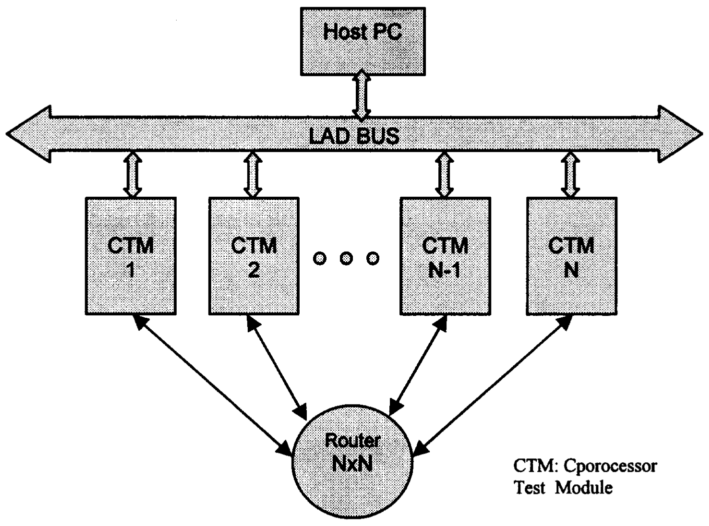


Figure 5.9 N-coprocessor system.

A generic N-coprocessor system is shown in Figure 5.9. Theoretically, such a system can be built for any N using the coprocessor and router. Any such implementation will always be constrained by the logic availability in an FPGA. 4-, 5- and 6-coprocessor systems can easily built into one Virtex 6000 FPGA. In order to build larger systems, a different approach may be used and Figure 5.10 shows an 8-coprocessor system using this 2-level approach.

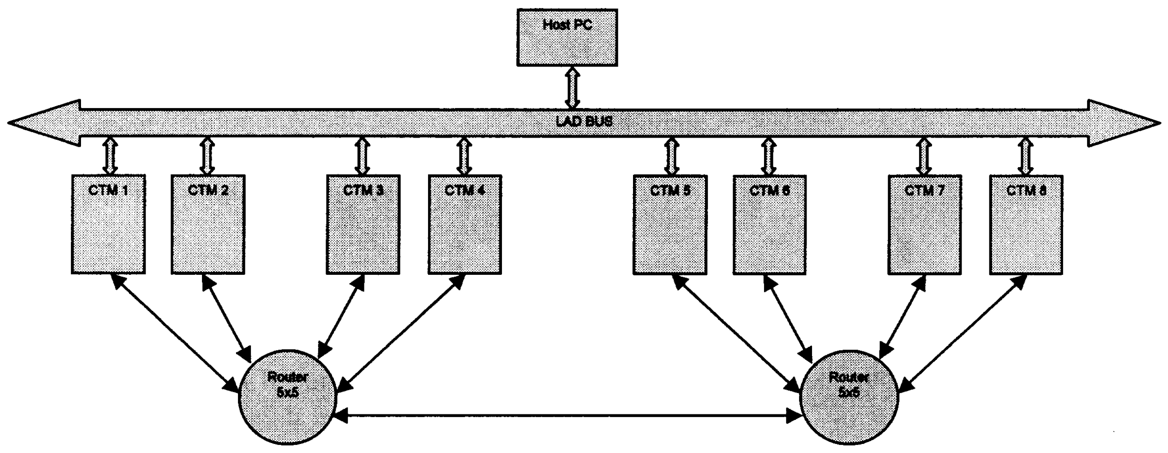


Figure 5.10 Eight processor system built using two-level router approach.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This thesis successfully implements an MPI based coprocessor for inter-processor communications in a multi-processor system. It also implements a router needed to build such a complete system. The communication latencies for MPI based one-sided communications were reduced as compared to that in a typical cluster environment. The next direct step would be to build and test a complete multi-processor system that employs this coprocessor for MPI communications.

6.2 Suggested Future Work

Figures 6.1, 6.2 and 6.3 illustrate how 2-, 4- and 8-processor MIMD machines could be built using the coprocessor and the router. A soft core processor like Microblaze could be used as the main CPU. The on-chip memory of Virtex-II and the on-board memories (SDRAM, SRAM) of the WILDSATR-II could be used as system main memories. Various parallel applications like matrix multiplication, LUT factorization could be implemented.

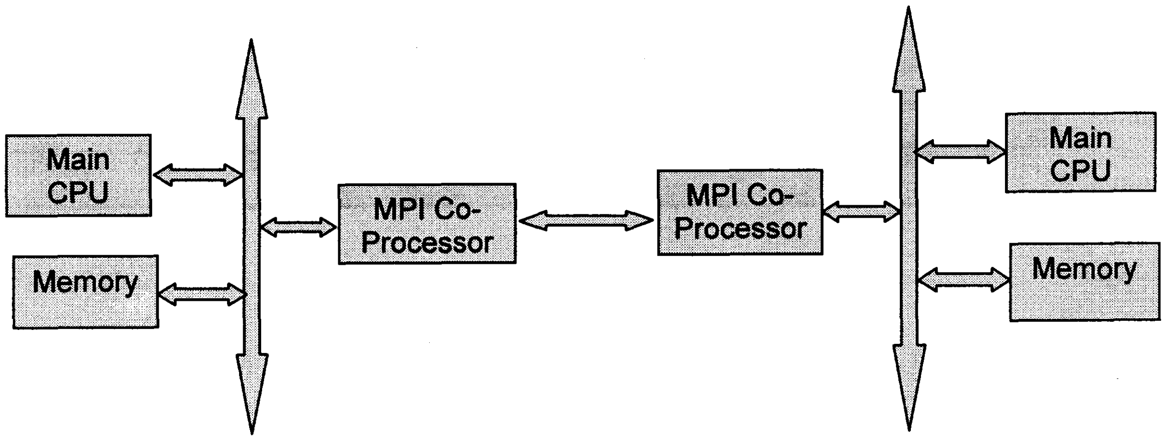


Figure 6.1 Two Processor System.

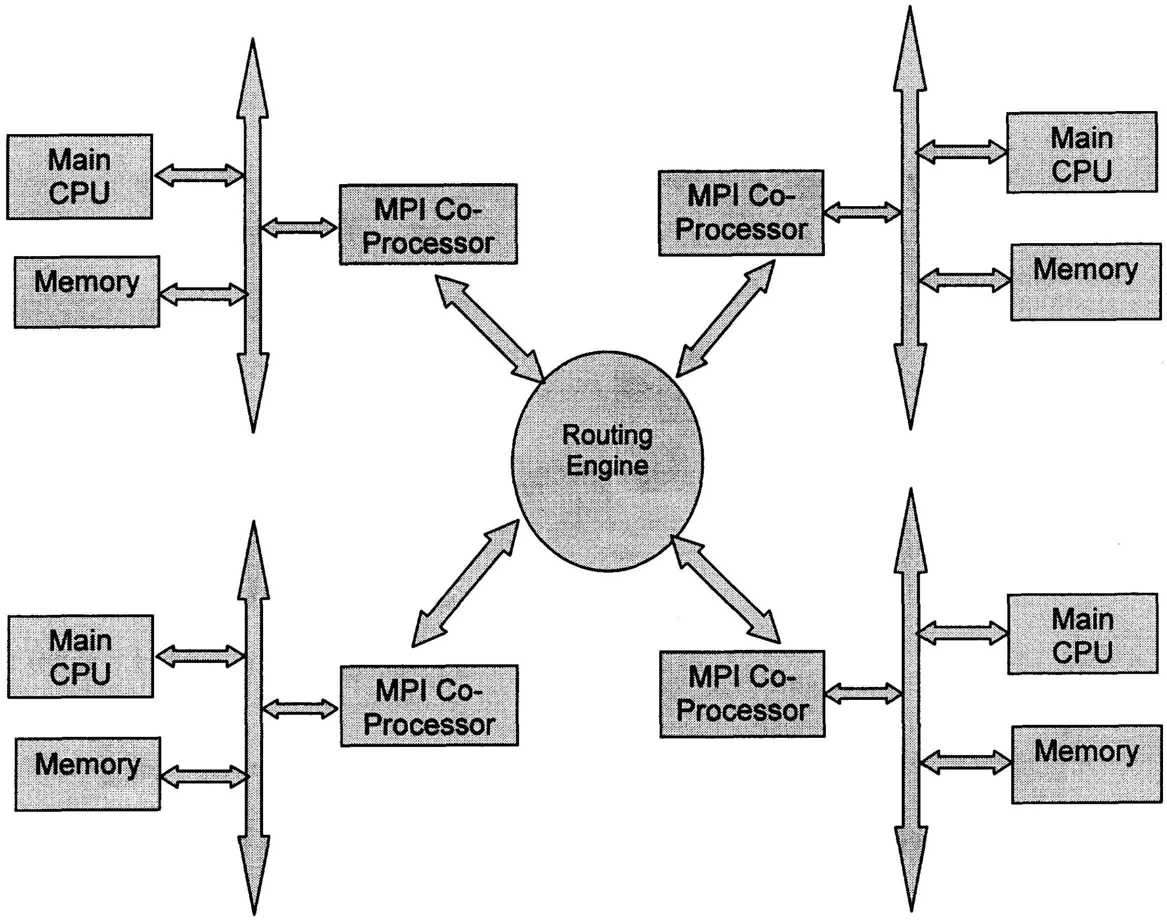


Figure 6.2 Four Processor System.

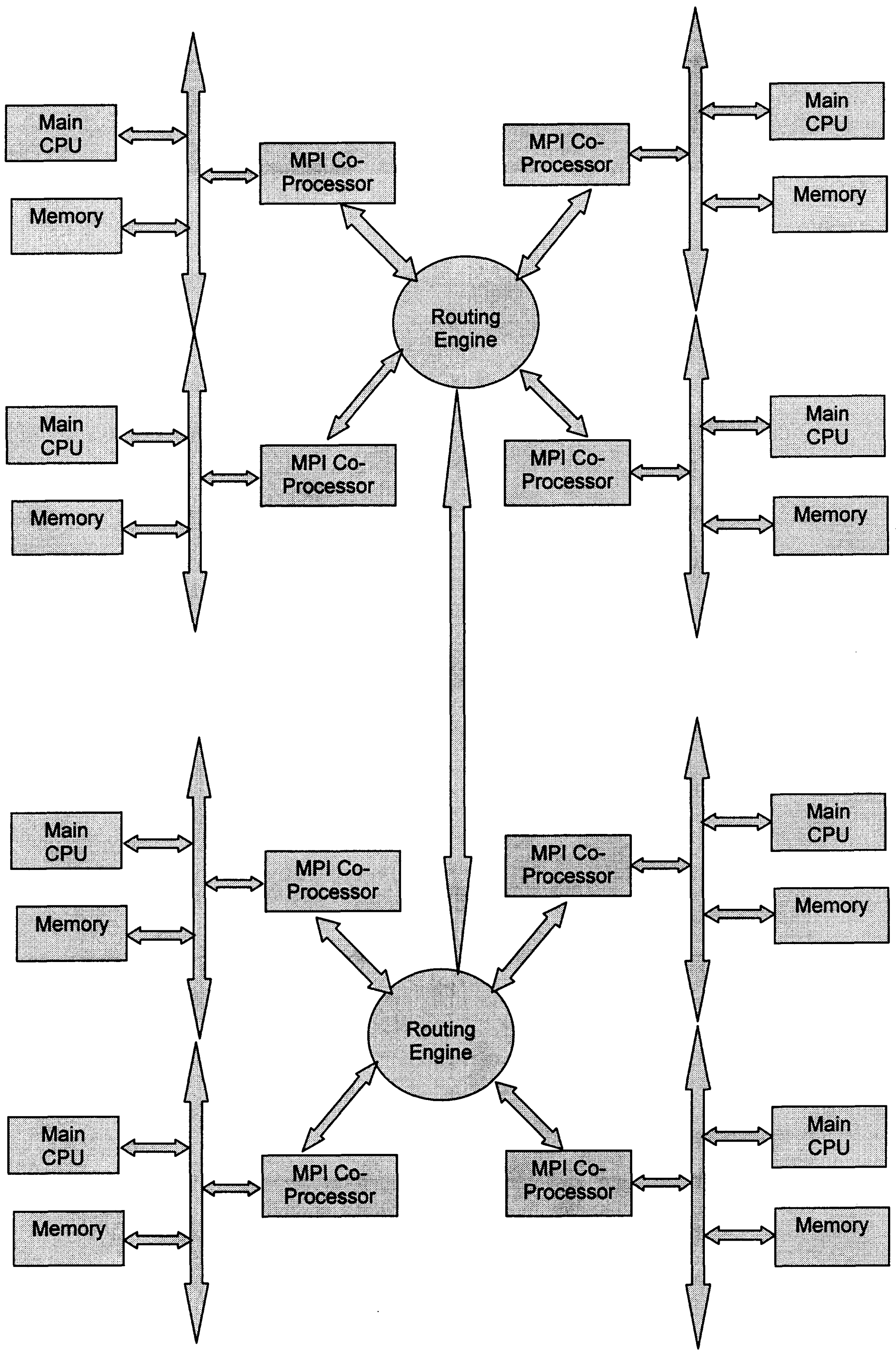


Figure 6.3 Eight processor system.

APPENDIX A

IMPLEMENTATION CONFIGURATION

The system implementer has full flexibility to choose the configuration of the coprocessor and the router depending on the implementation at hand and also on the amount of FPGA resources available. Tables A.1 and A.2 list the configuration of the coprocessor and router used for the implementation results of this thesis.

Table A.1 Coprocessor Configuration

Module	**Size in 32-bit words**	**Implementation Type**
Inst. FIFO	64	Select RAM
Data FIFO	64	Select RAM
Data8 FIFO	64	Select RAM
Header FIFO	16	Select RAM
MM Interface FIFO	16	Select RAM
Input Memory	512	Block RAM
Output Memory	512	Block RAM

Table A.2 Router Configuration

Module	**Number in N x N router**	**Size in 32-bit words**	**Implementation Type**
Input VOQ	N^2	32	Select RAM
Cross point Buffer	N^2	32	Select RAM
Input Memory	N	512	Block RAM
Output Memory	N	32	Select RAM

APPENDIX B

PERL COMPILER

A simple PERL compiler compiles the simple MPI based instructions to generate the MOV instruction necessary to program the coprocessor. Each instruction is executed by a simple PERL subroutine, which writes the address and the data for the MOV in a output file. This output file is then used by the host C program to program the coprocessor. The listing here gives only the subroutines and not the entire PERL program to get the output compile file.

```
sub NOP
{
        print FILE "01000000000000000000000000000111 0 \n";
}


sub register
{
        ($value) = @_;
        print FILE "01000000000000000000000000000101 $value \n";
}


sub deregister
{
        ($value) = @_;
        print FILE "01000000000000000000000000000110 $value \n";
}
```

```
sub Put1
{
        ($dest_add, $data, $dpid, $len , $offset) = @_;
        print FILE "01000000000000000000000000010001 $dest_add \n";
        print FILE "01000000000000000000000000010001 $data \n";
        print FILE "01100000000000000000000000000000 $dpid \n";
        print FILE "01100000000000000000000000000000 $len \n";
        print FILE "01100000000000000000000000000000 $offset \n";
}


sub Putn
{
        ($dest_add, $data, $dpid, $len , $offset) = @_;
        print FILE "01000000000000000000000000010010 $dest_add \n";
        print FILE "01000000000000000000000000010010 $data  \n";
        print FILE "01100000000000000000000000000000 $dpid  \n";
        print FILE "01100000000000000000000000000000 $len \n";
        print FILE "01100000000000000000000000000000 $offset \n";
}


sub Get
{
        ($dest_add, $data, $dpid, $len , $offset) = @_;
        print FILE "01000000000000000000000000010011 $dest_add \n";
        print FILE "01000000000000000000000000010011 $data  \n";
        print FILE "01100000000000000000000000000000 $dpid  \n";
        print FILE "01100000000000000000000000000000 $len    \n";
        print FILE "01100000000000000000000000000000 $offset \n";
}


sub barrier
{
        print FILE "01000000000000000000000000000100 0 \n";
}
```

APPENDIX C

PRIORITY ENCODER

Table C.1 shows the truth table of the priority encoder used by the top level CAM design.

Table C.1 Priority Encoder Truth Table

INPUT (32-bit)	**OUTPUT (5-bit)**
0xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	00000
10xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	00001
110xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	00010
1110xxxxxxxxxxxxxxxxxxxxxxxxxxxx	00011
11110xxxxxxxxxxxxxxxxxxxxxxxxxxx	00100
111110xxxxxxxxxxxxxxxxxxxxxxxxxx	00101
1111110xxxxxxxxxxxxxxxxxxxxxxxxx	00110
11111110xxxxxxxxxxxxxxxxxxxxxxxx	00111
111111110xxxxxxxxxxxxxxxxxxxxxxx	01000
1111111110xxxxxxxxxxxxxxxxxxxxxx	01001
11111111110xxxxxxxxxxxxxxxxxxxxx	01010
111111111110xxxxxxxxxxxxxxxxxxxx	01011
1111111111110xxxxxxxxxxxxxxxxxxx	01100
11111111111110xxxxxxxxxxxxxxxxxx	01101
111111111111110xxxxxxxxxxxxxxxxx	01110
1111111111111110xxxxxxxxxxxxxxxx	01111
11111111111111110xxxxxxxxxxxxxxx	10000
111111111111111110xxxxxxxxxxxxxx	10001
1111111111111111110xxxxxxxxxxxxx	10010
11111111111111111110xxxxxxxxxxxx	10011
111111111111111111110xxxxxxxxxxx	10100
1111111111111111111110xxxxxxxxxx	10101
11111111111111111111110xxxxxxxxx	10110
111111111111111111111110xxxxxxxx	10111
1111111111111111111111110xxxxxxx	11000
11111111111111111111111110xxxxxx	11001
111111111111111111111111110xxxxx	11010
1111111111111111111111111110xxxx	11011
11111111111111111111111111110xxx	11100
111111111111111111111111111110xx	11101
1111111111111111111111111111110x	11110
11111111111111111111111111111110	11111

REFERENCES

[1] A. V. Gerbessiotis and S. Y. Lee, "Remote memory access: A case for portable, efficient and library independent parallel programming," Technical Report CS-03-12, CS Department, New Jersey Institute of Technology.

[2] R. Rojas-Cessa, E. Oki, and H. J. Chao, "CIXOB-k: Combined Input-Crosspoint-Output Buffered Packet Switch," *Proceedings of IEEE GLOBECOM*, pp. 2654-2660, November 2001.

[3] R. Rojas-Cessa, E. Oki, Z. Jing, and H. J. Chao, "CIXB-1: Combined Input-One-Cell-Crosspoint Buffered Switch," *IEEE HPSR2001 Proc.*,May 2001.

[4] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches," *Proc. IEEE International Conference on Communications (ICC 2004)*, Paris, France, 20-24 June 2004, vol. 2, pp. 1090-1096. URL: http://archvlsi.ics.forth.gr/bufxbar/.

[5] M. Karol, M. Hluchyj, "Queuing in High-performance Packet switching," *IEEE J. Select. Area Commun.*, vol. 6, pp. 1587-1597,December 1988.

[6] M. Karol, M. Hluchyj, and S. Morgan: "Input versus Output Queueing on a Space Division Packet Switch," *IEEE Trans. Commun.*, vol. 35, no. 12, Dec. 1987, pp. 1347-56.

[7] N. McKeown, "Scheduling Algorithms for Input-Queued Cell Switches," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California at Berkeley, Berkeley, CA, 1995.

[8] L. Mhamdi, M. Hamdi, "Practical Scheduling Algorithms For High-Performance Packet Switches," *IEEE ICC 2003*, pp. 1659-1663, vol. 3, May 2003.

[9] M. Katevenis, N. Chrysos, G. Passas, and D. Simos, "Buffered Crossbar (CICQ) Switch Architecture," Retrieved November 2, 2004 from the World Wide Web: http://archvlsi.ics.forth.gr/bufxbar/.

[10] J. L. Brelet and L. Gopalakrishnan, "Using Virtex-II Block RAM for High Performance Read/Write CAMs," Xilinx Application Note XAPP260 (v1.1), February 27, 2002.URL: http://xilinx.com/bvdocs/appnotes/xapp260.pdf.

[11] C. E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002, Section TB2, 2nd paper.

[12] R. Bakka and M. Dieudonne, "Switching Circuit for Digital Packet Switching Network," *United States Patent 4,314,367*, February 1982.

[13] V. Singhal and R. Le, "High-Speed Buffered Crossbar Switch Design Using Virtex-EM Devices," Xilinx Application Note XAPP240 (v1.0), March 14, 2000. URL: http://xilinx.com/bvdocs/appnotes/xapp240.pdf.

[14] K. Yoshigoe, "Design and Evaluation of the Combined Input and Crossbar Queued (CICQ) Switch," Ph.D. dissertation, Dept. of Comp. Sci. and Engg., University of South Florida, August 2004.

[15] K. Christensen, "The Gigabit Ethernet Project - Literature review page #1," http://www.csee.usf.edu/~christen/career/lit1.html. Retrieved on November 12, 2004.

[16] "Virtex-II Platform FPGAs: Complete Data Sheet," version 3.3, Xilinx, June 2004.

[17] Wildstar II hardware reference manual, revision 5.0, Annapolis Micro Systems.

[18] B. Radanovich, "An Overview of Advances in Reconfigurable Computing Systems," Proceedings, Conference on System Sciences, 1999.

[19] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," IEEE Proc. Int. Conf. Exhib. Design Automation, Testing Europe, Munich, Germany, 2001, pp. 135-143.

[20] S. Hauck, G. Borriello, C. Ebeling, "Mesh Routing Topologies for Multi-FPGA Systems," International Conference on Computer Design, pp. 170-177, 1994.

[21] X. Wang and S. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," Concurrency and Computation , 2003.

[22] T. Golota and S.G. Ziavras, "A Universal, Dynamically Adaptable and Programmable Network Router for Parallel Computers," VLSI Design , Vol. 12, No. 1, 2001, pp. 25-52.

[23] N. Fugier, M. Herbert, E. Lemoine, B. Tourancheau, "MPI for the Clint Gb/s Interconnect," PVM/MPI, 2003, pp. 395-403.

[24] "ISE 6 Software Manuals", Xilinx.