Implementation Issues of Building a Multicomputer on a Chip

A Thesis

Presented to

the Faculty of the Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in Electrical Engineering

by

Chaitanya Adapa

August 2001

Implementation Issues of Building a Multicompute on a Chip

Chaitanya Adapa

Approved:

Chairman of the Committee
Martin Herbordt, Associate Professor,
Electrical and Computer Engineering

Committee Members:

Pauline Markenscoff, Associate Professor,
Electrical and Computer Engineering

Jaspal Subhlok, Associate Professor,
Computer Science

E. J. Charlson, Associate Dean,
Cullen College of Engineering

Fritz Claydon, Professor and Chair
Electrical and Computer Engineering

Acknowledgements

Special thanks and gratitude to my advisor Dr. Martin Herbordt for his guidance, friendship without which this thesis would not be possible. Special thanks to Dr. Pauline Markenscoff and Dr. Subhlok for serving on my committee. This research was supported in part by the National Science Foundation through CAREER award \#9702483 and by the Texas Advanced Technology Program under grant 003652-0424.

Implementation Issues of Building a MultiComputer on a Chip

An Abstract of

of a

Thesis

Presented to

the Faculty of the Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in Electrical Engineering

by

Chaitanya Adapa

August 2001

ABSTRACT

Availability of highly dense chips has made systems-on-a-chip a reality. In this context, systems with multiple processors are being built on a single chip to achieve higher performance. With the availability of freely distributed Intellectual Property (IP) processor cores and related tools, it is possible for university projects such as ours to engage in this type of system development. We have built fine-grained, highly parallel, virtual multicomputers-on-a-chip using IP cores to establish this proof-of-concept. We have established an evaluation methodology whereby the various architectural tradeoffs in this class of designs can be examined. The functionality of individual components and several systems as a whole are verified using commercial simulation tools. Sample area and timing results that were generated using commercial synthesis tools have indicated possible physical implementations of multicomputers-on-a-chip.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

Recent advances in process technology have resulted in increased gate densities and better wiring technology, enabling systems-on-a-chip. Consequently, it is now possible to integrate multiple processors and a communication network onto a single substrate to create multicomputer systems. Such a system avoids costly off-chip inter-processor communication resulting in high-speed transfer of data among processors and potentially very high performance for appropriate applications. The availability of processor cores and tools for their integration allows university projects such as ours to explore the design space of high-performance single chip systems, particularly for domains such as computer vision. Recently, resources have become available that provide a wide choice of 8-bit and 32-bit processor cores. We use two such cores RISC8 and LEON to do experimentation and understand the issues involved and to gain experience with building a virtual multicomputer system.

1.2. Objective

System-on-a-chip technology has the advantages of lower power, greater reliability, and lower cost. With removal of chip boundaries, the off-chip communication among processor chips is converted to on-chip communication among processing elements. Removal of chip boundaries, however, also changes some of the design considerations for a multicomputer. Among these are processor granularity, type of communication network, bus width to access memory off-chip, and the amount of memory on-chip. In this thesis, we address the implementation issues of building a multicomputer system-on-a-chip using IP cores with the goal of enabling exploration of this design space.

1.3. Context

Process technology has improved over the years and it is now possible to have up to 100 million transistors or more on a single chip. Thus, more and more system functionality can be integrated onto a single chip, reducing the number of chips in the system. However, there is a choice of what to put in a single chip. Two extremes are a powerful processor with L2 cache on the one hand and a large number of very simple processors and their interconnections on the other. Processors such as the wide issue, dynamically scheduled, super scalar Alpha 21464 are a good example of the first type. However, studies [11] show that the implementation complexity of the dynamic issue mechanisms and the size of register file scale quadratically with increasing issue width and ultimately impact the cycle time of the machine. Therefore, an alternative is suggested [11] where several less-complex processors on the same chip would use the real estate efficiently and scale well. Presently systems like RAPTOR [7] have demonstrated the integration of multiple processors on a single chip. Further improvement in the transistor count is inevitable, which only means higher integration or the integration of whole multiple processor systems onto a single chip. We feel that a logical extension of the multiple processors on a chip approach is to consider a variety of candidate processors, including those with very simple CPUs.

New design methodologies have been devised to handle this complexity and have reduced product time-to-market considerably. Use of IP cores is one such improvement:

integrating these "pre-designed" components reduces development time as the system is built around the core. Semiconductor companies such as LSI logic [2] license IP cores along with related tool support. Lately, open source IP cores have also become available and rapid development of designs at very low cost is now possible. Web resources such as the OPENCORES [10] organization are attempting to standardize a methodology in developing designs using these cores. We have used the resources provided by such open source providers [3, 5].

1.4. Design Criteria and System Specification

In the previous section, we briefly described some approaches to building systems-on-a-chip containing multiple processors. We believe that a potentially fruitful design path for high performance processors is to investigate the use of very simple CPUs. The reasoning is that current microprocessors have almost universally been designed with the chip boundary as a primary constraint: maximum functionality is placed on the chip and the entire available chip area is used. This approach has especially good possibilities for computer vision and graphics applications.

The design criteria for building systems of this type are:

- Maximum processing capability
- Control autonomy of the processors
- Flexible communication
- Ease of design

System specifications that satisfy the above criteria

- RISC8 and LEON processor cores are relatively simple and publicly available.
- The communication networks developed in the CAAD lab provide the necessary communication capability.
- Tools such as the MPLAB, LEONCCS are useful in designing, debugging and programming.

1.5 Design Methodology and Flow

The following method for developing the system has been used

- Obtain processor as an IP core
- Develop the network interface using Verilog
- Integrate the communication network
- Connect the components of the system into a top module
- Use the top module for simulation and synthesis

The above design methodology is implemented using the design flow shown in Figure1.1, which illustrates the various tools used. The design component is entered in Verilog or VHDL, or as an IP core. The simulation is done using SILOS, which does Verilog simulation only, and Aldec Active-HDL, which does mixed mode simulation. After simulation and verification of the design, the synthesis tool (Synopsys) is used to synthesize the design to extract area and timing parameters.

Use of freely distributed cores has the advantage of low cost in procurement of core and related tools. However, the working of these designs is not guaranteed.

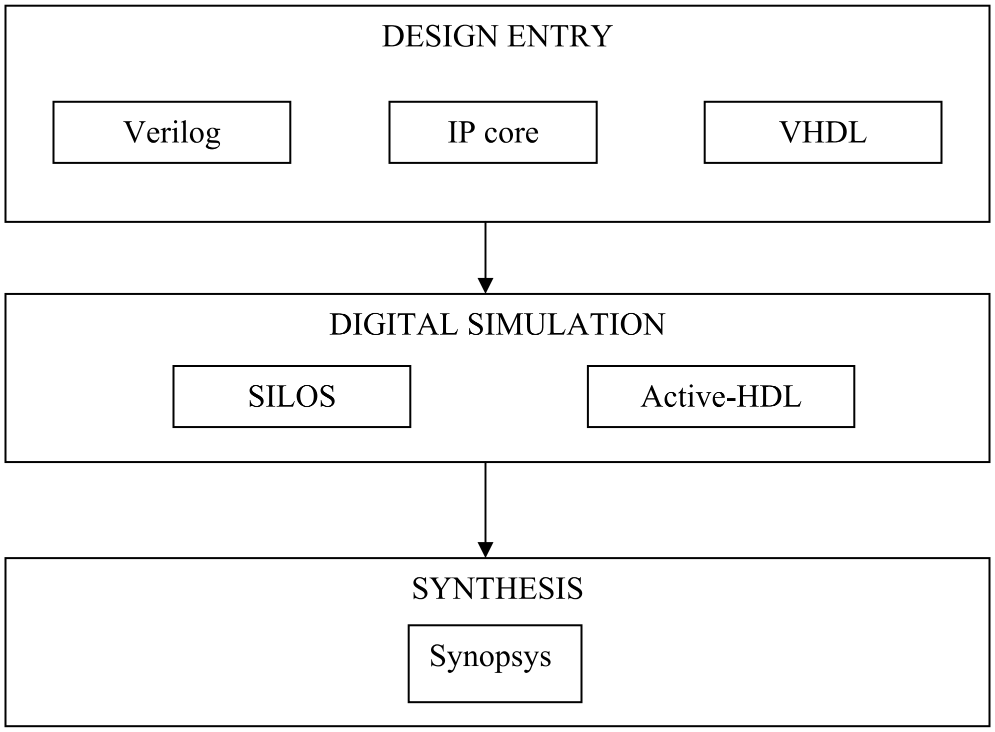


Figure 1.1 Design Flow

1.6 Results Overview and Significance

The goal of this research is to build a working virtual multicomputer system using IP cores. The processors read programs from the program memory and execute the code successfully. These programs are compiled or cross-compiled into the processor instruction set and stored into the processors program memory. Results are obtained by simulation and synthesis. Simulations verify the working of the system and synthesis results give an idea of the physical implementation of the system. Also, issues like amount of memory on-chip and off-chip, number of processors and area occupied by communication network are shown to be the concerns when building systems on a chip. It was possible to synthesize individual components but an attempt to synthesize the whole system failed due to the limitation of the Synthesis tool.

Experience has been gained in building a Virtual Multicomputer system: from the results it is realized that better tools for synthesis will enable better estimation of the physical implementation. With the next generation of process technologies, it will be possible to design a larger system with a more significant processor.

1.7 Thesis Outline

The next chapter builds the background of the traditional multicomputer and integrated circuit design. Chapter 3 gives the analysis on the design space by providing the assumptions and decisions made on design parameters. Chapter 4 describes issues encountered when implementing various components and integrating into a system. Experiments done on the implementations are also explained in Chapter 4 and results obtained from them are detailed in Chapter 5. Chapter 6 gives the conclusions and some ideas for future work.

2. BACKGROUND

This chapter explains the changes in multicomputer design with respect to network interface and interconnection networks. The earlier network interfaces and changes that have been brought about in recent architectures are discussed. In addition, different interconnection networks are mentioned and the reason for selecting an array of routers as the communication network is given. Also included is a brief description of the design methodology used in building the system using IP cores.

2.1 Multicomputer Systems

Multicomputer architectures employ complete computers as building blocks – including processor, memory and I/O system. Early multicomputers were essentially the same as NUMA shared memory architectures, differing in integration of communication at the I/O level rather than the memory level. However, the trend has been to integrate the communication module more deeply into the memory system as well and to transfer data directly from user address space. Some designs provide DMA transfers across the network, from memory on one machine to memory on the other machine, so the network interface or the communication module is integrated fairly deep into the memory system. The nodes, consisting of the processor and network interface, interact with each other by connecting themselves to interconnection network such as a crossbar, multistage interconnection network or a bus interconnect. Proper design of a network interface and the communication network can result in a high performance system. The next section describes the developments in network interface design.

2.1.1 Network Interface Design

Early multicomputers provided hardware primitives that were very close to the simple send/receive user-level communication. A node of the multicomputer system was connected to a fixed set of neighbors in a regular pattern by point-point links that behaved as simple FIFOs. Hypercube was an example of such an organization, where the synchronous message passing was implemented. This direct FIFO design was soon replaced by direct memory access (DMA), allowing non-blocking sends. The physical topology of these designs dominated the programming model. However, to make the machines more general purpose, support for communication between arbitrary processors was provided rather than just the physical neighbors. One example of such an approach is the store and forward where the transfer time is proportional to the number of hops it takes through the network.

The emphasis on network topology was significantly reduced with the introduction of more general-purpose networks, which pipelined the message transfer through each of the routers forming the interconnection machine. Processor clock frequencies are approaching the gigahertz range and the network switch latencies dropped to tens of nano seconds. This explosive growth also exposes processor accesses to the network interface as the critical bottleneck for fine-grain communication. The incremental delay introduced by each hop is small enough that the transfer time is dominated by the time to simply move the data between processor and the network. So emphasis is given to reduce the latency caused by overhead due to Operating System involvement. We have mentioned earlier in this section that network interface might prove to be a bottleneck when we have fast processor and the network, however there are some issues like scaling which make us think about the communication network we are going to use in the system on chip. The next section explains the reason for selection of a communication network for our system.

2.2 Communication Network Design

The communication networks in multicomputers, multiprocessor and Network Of Workstations (NOW) are asynchronous networks with distributed control. These networks can be classified based on network topology as shared, direct, indirect or hybrid mediums.

Shared Medium: all the devices share the transmission medium. Scaling the devices will not scale the bandwidth so there might be a decrease in performance with scaling.

Direct Medium: The devices are connected to the nearest neighbors with point-to-point links. Communication with non-neighbors will require passing through many devices resulting in large latency.

Indirect Medium: The devices use switches to connect to each other. The switches have point-to-point links among them.

Hybrid Medium: from the name it is obvious that a hybrid network is a combination of the above networks.

As mentioned earlier Shared medium networks are not scalable as their bandwidth gets spilt between the devices. So the best alternative is to use switch or router based networks as the bandwidth scales with the number of nodes. But the disadvantage would be that the cost increases due to increase in the number of switches/routers. An alternative, wherein the cost of increasing the routers is small is required. With advances in process technology chips with greater density are possible, which can incorporate whole of the multicomputer system with the nodes and switches/routers. So the cost reduces to manufacturing one single chip and we meet the budget even with a better scaling network. Thus the use of an array of routers with point-to-point links as an interconnection network will provide scaling and due to better process technology, be cost effective.

The next section introduces Integrated circuit design flow and also the design methodology, which illustrates IP core usage in the design. This should help in building a background for understanding our design methodology.

2.3. Integrated Circuit Design

In the current revolution of IC and microelectronics design, large number of logic elements can be put on a single chip die. Number of transistors on a single die can be as many as 33 million, which enables designers to put large systems on a single chip. This progress, in turns, lead to the introduction of new design methodologies and techniques. This technology is called System-on-Chip "SoC" that uses different design blocks made by several designers. These designs should be reusable ones so as to reduce the time needed to develop the same functionality again by other group.

This system-on-chip (SOC) solution has the advantages of lower power, greater reliability and low cost, making it possible to have a cost-effective solution for high performance parallel architectures.

2.3.1. ASIC Design Flow

The steps involved in the conventional ASIC design [6] are listed below (numbered to correspond to the labels in Figure 2.1 with brief description of the function of each step.

1. Design Entry: Enter the design into an ASIC design system, either using a hardware description language (HDL), schematic entry, block diagram or state diagram.
2. Logic Synthesis: Use an HDL (VHDL or Verilog) and a logic synthesis tool to produce a net list, which is a description of the logic cells and their connections.
3. System Partitioning: Divide a large system into ASIC-sized pieces.
4. Prelayout Simulation: Check to see if the design functions correctly
5. Floorplanning: Arrange the blocks of the net list on the chip.
6. Placement: Decide the location of the cells in the block.
7. Routing: Make the connections between cells and blocks.
8. Extraction: Determine the resistance and capacitance of the interconnect.
9. Postlayout Simulation: Check to see the design still works with the added loads of the interconnect.

The sequence of steps in design flow, to design an Application Specific Integrated Circuit (ASIC) is shown in Figure 2.1.

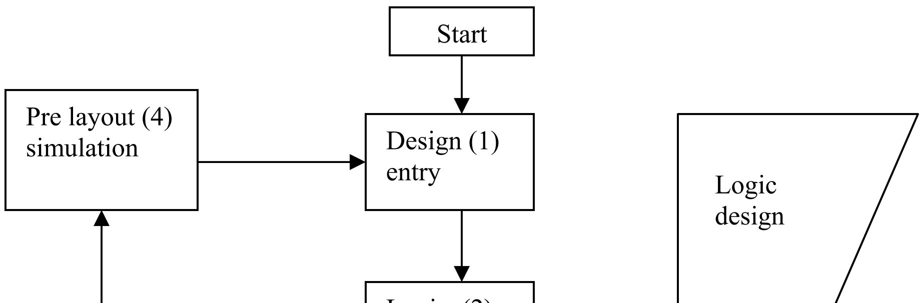


Figure 2.1 ASIC Design flow

We are concerned with the first two stages in the design flow, design entry and logic synthesis in our thesis. The next section describes the conventional design methodology of using IP cores, which is what we use in our system design.

2.3.2. Design Methodology

The functional blocks used in chip design are available as Intellectual Property (IP) cores or macros. Use of hard, soft, and configurable cores for system-on-a-chip (SOC) designs is becoming common with new and improved design processes and design flow. One instance of this design methodology is LSI Logic [1] CoreWare Design Methodology. LSI logic, the semiconductor company, through its CoreWare methodology provides the IP core and related support with tools and libraries which enable successful integration of the IP core and additional logic surrounding the core onto a single silicon substrate. Use of proven cores aids in design reuse, which reduces the design time.

A core or macro is the Intellectual Property of the core developers, made available as soft, firm and hard macros. The soft macros are the RTL net lists, which can be synthesized and configured. To protect the macros they may be encrypted. Firm macros are Synthesized Gate level net lists. Simulation using gate level net list takes a prohibitively long time so behavior level code is also given. Hard macros are already laid out and characterized for a specific process technology. Here also behavior level code is given for simulation. The advantage of a hard macro is that it reduces the number of iterations, but the disadvantage is that it is tied up to a process technology. The hard macro is like a black box whose interface is known, but the implementation is not obvious to the user of the core.

In the next chapter we look at the design parameters and the assumptions made in order to make it possible to build the multicomputer system.

3. DESIGN SPACE

In this chapter we outline the parameters and assumptions made in the design of multicomputer system on a chip.

3.1. Assumptions and Parameters

With removal of chip boundaries the design parameters or concerns have shifted to the processor granularity (area, frequency and performance), network latency, on-chip memory and off-chip memory. The following sections give the details of the analysis we have done to explore the design space.

3.1.1. Processor Evaluation

The first task in building the multicomputer system was to acquire the processor as an IP core. We obtained the technology libraries for logical synthesis from LSI logic [2] and so evaluated the processor cores available from LSI logic. The following Table3.1 summarizes the features of each processor with respect to area, frequency and performance.

TABLE 3.1. Processor Evaluation

Core	**Area**	**Freque-ncy**	**Perfor-mance.**	**Power**	**ISA**	**Process**	**Cache**
Tiny RISC 4102	1.1 mm^2	85MHz	N/A	0.5mW/ MHz	MIPS I/II	2.5v @ 0.25um	No cache

Tiny RISC 4103	1.9 mm^2	120MHz	N/A	0.5mW/MHz	MIPS I/II	1.8v @ 0.18um	32KB I-cache and D-cache
Mini RISC EZ4021	12 mm^2	250MHz	275 DMIPS @250MH	2.6mW/MHz	MIPS III	1.8v @ 0.18um	16KB 2-way I-cache and D-cache
SR1-GX	21 mm^2 to 29 mm^2	333-400 MHz	2 MIPS/MHz	1.4W @ 400MHz	MIPS IV	1.8v @ 0.18um	Upto 64KB
ARM946E-S	2.6mm^2 @0.18um (ARM966E-S)	200MHz	1.1 MIPS/MHz	N/A	ARM9E	1.8v @0.18um	4kB to 1MB

LSI Logic has developed a robust library containing some of the most advanced cores in the industry. All cores have been proven in silicon with a fixed layout to ensure accurate timing and predictability. Microprocessor cores provided by LSI Logic include ARM7TDMI, ARM9E, MiniRISC, TinyRISC. The search was for a processor, which is small but powerful and a good number of these, along with network interface and interconnection network should fit in a 1 cm^2 die. Among the above five short-listed, TinyRISC is the smallest, but its frequency of operation is small and it does not include on-chip cache. The next one in line is the MiniRISC, which has on-chip cache and higher performance, and with 12mm^2 area, it appears to be a good candidate. ARM7TDMI with 2.6mm^2 is even better.

To avoid the expense involved in licensing these processor cores, at least initially, we decided to experiment with toy processors available for free. Two of them have been downloaded, one is the RISC8 and the other is LEON processor.

3.1.2. RISC8

A free IP core compatible with the MicroChip 16c57 8-bit processor was obtained from the web site of Thomas A Coonan [5]. It is coded in Verilog.

3.1.2.1. Features

- Harvard Architecture with separate program and data memory
- Binary code compatible with the Microchip 16C57
- Up to 72 8-bit data words and 2048 12-bit program words (configurable)
- It has an accumulator-based instruction set (33 instructions).
- Pipelines its Fetch and Execute
- The Register File uses a banking scheme and an Indirect Addressing mode.
- The Program memory (PRAM) is a separate memory from the Register File and is outside the core.

- The ALU is very simple and includes the minimal set of 8-bit operations (ADD, SUB, OR, AND, XOR, ROTATE, etc.)
- The Instruction Decoder is a purely combinatorial look-up Table that supplies key control signals

3.1.2.2. Block Diagram

The RISC8 had two input ports and one output port, which were unidirectional. The Figure 3.1 below shows the components of the original processor core, where output data is sent to Port B and PortC input data is read from Port A.

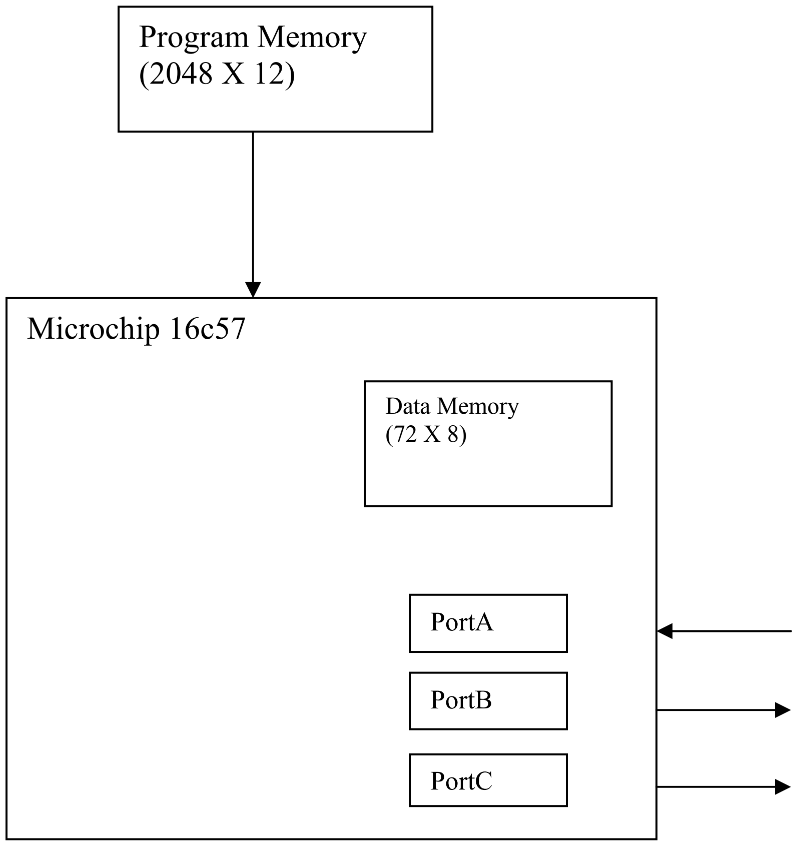


Figure 3.1 The Block Diagram of RISC8

3.1.3. LEON

The second free IP core is based on the 32-bit architecture of SPARC version 8. It is obtained from the site maintained by Jiri Gaisler [3], which coded in VHDL. The VHDL model is fully synthesizable and contains synthesis scripts.

3.1.3.1. Features

- The Integer Unit has the following features:
    - 5-stage instruction pipeline
    - Separate instruction and data cache interface
    - Support for 2 - 32 register windows
- Multiprocessor synchronization instructions
    - Atomic read-then-set-memory operation
    - Atomic exchange-register-with-memory operation.
- Architecture defines coprocessor instruction set in addition to the floating-point instruction set.

3.1.3.2. Block Diagram

The Figure 3.2 below shows the block diagram of the LEON processor with integer unit, cache, optional FPU and coprocessor, AHB/APB bus controllers and memory controller.

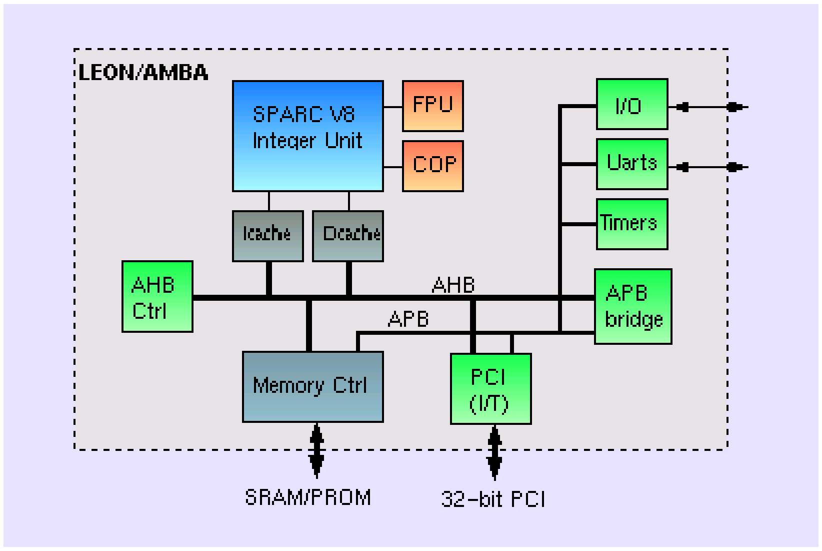


Figure 3.2 Block Diagram of LEON Processor Core

3.1.3.3. Previous Implementations

Targeting a 0.25 um CMOS process (std-cell), more than 125 MHz can be reached with a gate count of less than 30 K gates. The processor also fits in an Altera Xilinx XV300 The Following synthesis results were achieved

TABLE 3.2. Previous implementations of LEON

Technology	**Area**	**Timing**
UMC 0.25 CMOS std-cell	35K gates + RAM	130 MHz (pre-layout)
Atmel 0.25 CMOS std-cell	33K gates + RAM	140 MHz (pre-layout, log file)
Atmel 0.35 CMOS std-cell	2 mm2 + RAM	65 MHz (pre-layout, log file)

Xilinx XCV300E-8	4,800 LUT + block RAM	45 MHz (post-layout)
Xilinx XCV800-6	4,800 LUT + block RAM	35 MHz (post-layout, log1, log2)
Altera 20K200C-7	5,700 LCELLs + EAB RAM	49 MHz (post-layout) log file
Altera 20K200E-1X	6,700 LCELLs + ESB RAM	37 MHz (post-layout) log file

The area in the Table reflects the complete processor with on-chip peripherals and memory controller. The area of the processor core only (IU + cache controllers) is about half of that. The timing for the ASIC technologies has been obtained using worst-case process corner and industrial temperature range.

Based on the above information it is clear that LEON has larger cache on-chip, is pipelined and is better suited for systems with multiple processors. But complexity of LEON is high due to the amount of control involved in instruction processing compared to RISC8, so initial experiments have been conducted with RISC8 and then LEON has been used later on.

3.1.4. Network Interface

The existing designs of network interface can be broadly grouped into four categories [1].

1. *OS Based DMA Interface:* The message handling is relegated to the DMA. A message is written into a memory location and an operating system send directive is executed. At the hardware level both machines send and receive messages by initiating a DMA transfer between memory and the network channel. Examples of such a parallel system are NCUBE and the iPSC/2. There is a large overhead due to the involvement of OS, which is justified, as OS involvement is required for protection.
2. *User Level Memory Mapped:* The important feature of these interfaces is that the latency of accessing the network interface is similar to accessing memory. The system calls are avoided and since interfaces are user level, extra copies between system and user space are eliminated. Examples of this approach include the MDP Machine, the CM-5 , the memory communication of iWarp , and the message-passing interface of the MIT Alewife .
3. *User level register mapped:* The interface resides in the processor register file thus allowing rapid access. These models do not support message-passing functions. Two examples which support this communication model are the grid network of CM-2 and the systolic array of iWARP.
4. *Hardwired interfaces:* The communication function is hardwired into the processor thus eliminating any software overhead. This approach does not provide the programmer the flexibility to allocate resources and has no control over the details of how the communication occurs. Examples of this approach are shared memory interface of MIT Alewife.

From the above four cases, we understand that the network interface we build should satisfy the following

- Be user programmable, should not invoke OS
- Sending and Receiving should be under the control of user program
- The processor-network interface should be located closer to the processor register file for rapid transfer
- Frequent message operations like dispatching should be assisted by hardware mechanisms

3.2. The Multicomputer System

Increase in chip density can be exploited to integrate a complete system on a chip providing a cost effective solution. However, there lies the issue of access to program and data, which reside in memory. Since all of it cannot reside on-chip, off-chip memory access is inevitable. How much of memory resides on chip will affect the number of processors or alternatively their granularity or both. The number of processors or nodes will determine the off-chip memory access bandwidth due to pin limitation. Tradeoff has to be made between number of processors, off-chip memory access bandwidth and memory on-chip.

The interconnection network could be a bus, cross bar, multistage switches or an array of routers. As mentioned in the previous chapter it is better to use an array of routers due to better scaling provided by them and the use of systems on chip technology and tools we can make it cost effective.

The next chapter gives the specification of the architecture implementations of components that will be used for the experimentation.

4. IMPLEMENTATION ISSUES

This chapter gives the details of the implementations done with RISC8 and procedure for working with LEON. The necessary changes and proper interface is defined for the processor cores in order for them to be part of a Multicomputer System.

4.1. RISC8

Since RISC8 is the simpler of the two cores, it has been utilized to gain experience in using IP cores. RISC8 is available as a soft core described in Verilog HDL, allowing modification in its architecture. From the description of RISC8 in previous chapter it is known that it has two output ports and one input port. To have point-to-point communication among neighboring processors and reduce latency two implementations were done.

4.1.1. Implementation-1

Figure 4.1 below shows the processor after a Communication Module and four bidirectional ports were added . The data is put out by storing value in PortB and data is taken in by reading from PortA. PortC is used to conFigure the ports as inputs or outputs. All the three registers are part of processor register file. At a time one value can be read from a port and one value can be broadcast to four ports.

The ports are conFigured as

PortC [7:4]	PortC [3:0]
0001: N = PortB	0001: PortA = N
0010: E = PortB	0010: PortA = E
0100: W = PortB	0100: PortA = W
1000: S = PortB	1000: PortA = S

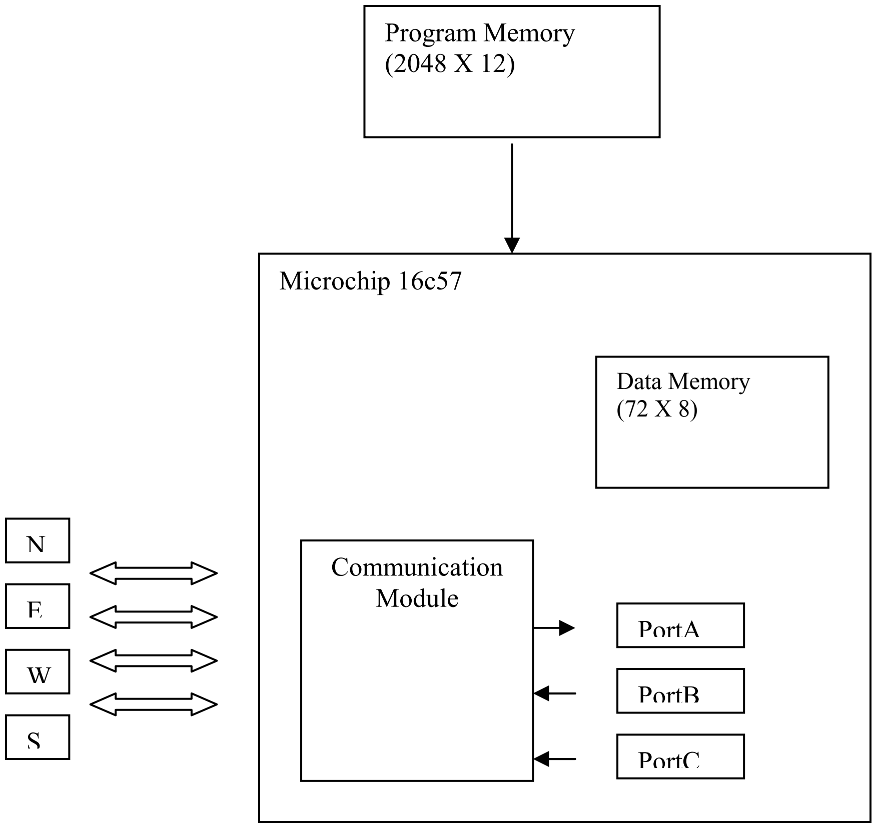


FIG 4.1. Block diagram of Implementation-1 of RISC8

4.1.2 Implementation-2

Changes have been made to enable block transfer at the hardware level. Three existing registers TrisA, TrisB, TrisC were included in the communication module. However, data can be transfered from accumlator to these registers but not vice-versa. Since the Instruction Set Architecture is accumlator based it would be useful to have data transfer capability from these three registers to the accumlator. So PortC, which is the configuration register for the ports, can also be used for determining the transfer of data between one of the Tris registers and the accumlator. Figure 4.2 shows the communication module added to the RISC8 core.

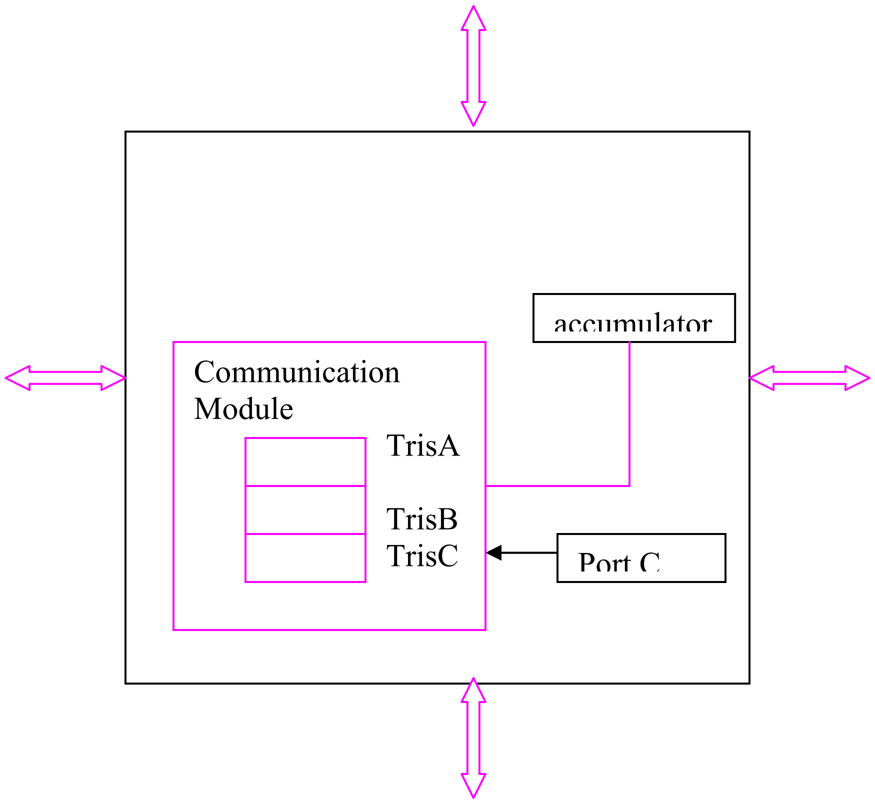


FIG 4.2. Block diagram of Implementation-2

The bidirectional Ports are conFigured as

PortC [7:6]: 01

PortC [5]: Read/Write

PortC [4:3]: Decide among N, E, W or S

PortC [2:1]: Number of bytes to be transferred 1, 2 or 3

PortC [0]: done bit indicating completion of operation

Data transfer between accumulator and the 'Tris' registers is determined as follows

PortC [7:6]: 10

PortC [5:3]: 101 Acc <- TrisA

PortC [5:3]: 110 Acc <- TrisB

PortC [5:3]: 111 Acc <- TrisC

4.2. LEON

The LEON processor is SPARC V8 compatible and has support from GNU cross compiler, which compiles the C and C++ programs into the instruction set of SPARC V8. The cross compiler is known as LEONCCS and is available in the web site of JiriGaisler [3]. Also provided are utilities like “mkprom”, which is a binary utility, used to build a memory BFM in VHDL for use in simulation. The application program is first cross-compiled into the processor’s instruction set and then the "mkprom" utility is used to form a memory BFM. The memory BFM has the following responsibilities

- The register files of IU and FPU (if present) are initialized.

- The LEON control, wait state and memory configuration registers are set according to the specified options.
- The ram is initialized and the application is decompressed and installed.
- The text part of the application is optionally write-protected, except the lower 4K where the trap table is assumed to reside.
- Finally, the application is started, setting the stack pointer to the top of external memory.

After booting the processor the application is loaded into the external SRAM and then the processor starts fetching the application code.

4.3. Network Interface Design

To connect and access the network interface the processors should have the required signal interface and instructions in the instruction set. Analysis that has been done for each of the cores in this regard is given in this section.

4.3.1. RISC8:

The RISC8 was analysed for its use in a Multicomputer configuration and the following facts have been understood. In the Multicomputer configuration it was required that the processor has to interface to the Router, which has a buffer. To transfer data between Processor and the Router there needs to be some control signals coming out of the processor. Upon exploring we found that the processor has a set of ports known as the expansion ports, which can be used for interfacing with external devices. The description of the ports is given in Table 4.1 below

Table 4.1. Expansion Interface Signals

Signal	**Description**
Expdin [7:0]	Input back to the RISC8 core. This is 8-bit data from the expansion module(s) to the core. Should be valid when 'expread' is asserted.
Expdout [7:0]	Output from the RISC8 core. This is 8-bit data to the expansion module(s) from the core. Is valid when 'expwrite' is asserted. The expansion module is responsible for decoding 'expaddr' in order to know which expansion address is being written to.
Expaddr [6:0]	This is the final data space address for reads or writes. It includes any Indirect addressing.
Expread	Asserted (HIGH) when the RISC8 core is reading from an expansion address.
Expwrite	Asserted (HIGH) when the RISC8 core is writing to an expansion address.

The port access is similar to register access. The highest four slots in register address space (0x FC, 0x FD, 0x FE and 0x FF) are allotted to these ports. Since we have signals ‘expread’ and ‘expwrite’, it is easy to interface through these signals. However, it is seen that some control signals are to be sent to the processor to indicate the status of external device and so PortA is used for this purpose. Experimental test were done to verify the send and receive of data. The network interface has two buffers each of four 10-bit locations. One acts as an input buffer and the other is the output buffer. The network interface interaction with router can be reviewed from the earlier chapters. The network interface consists the following components

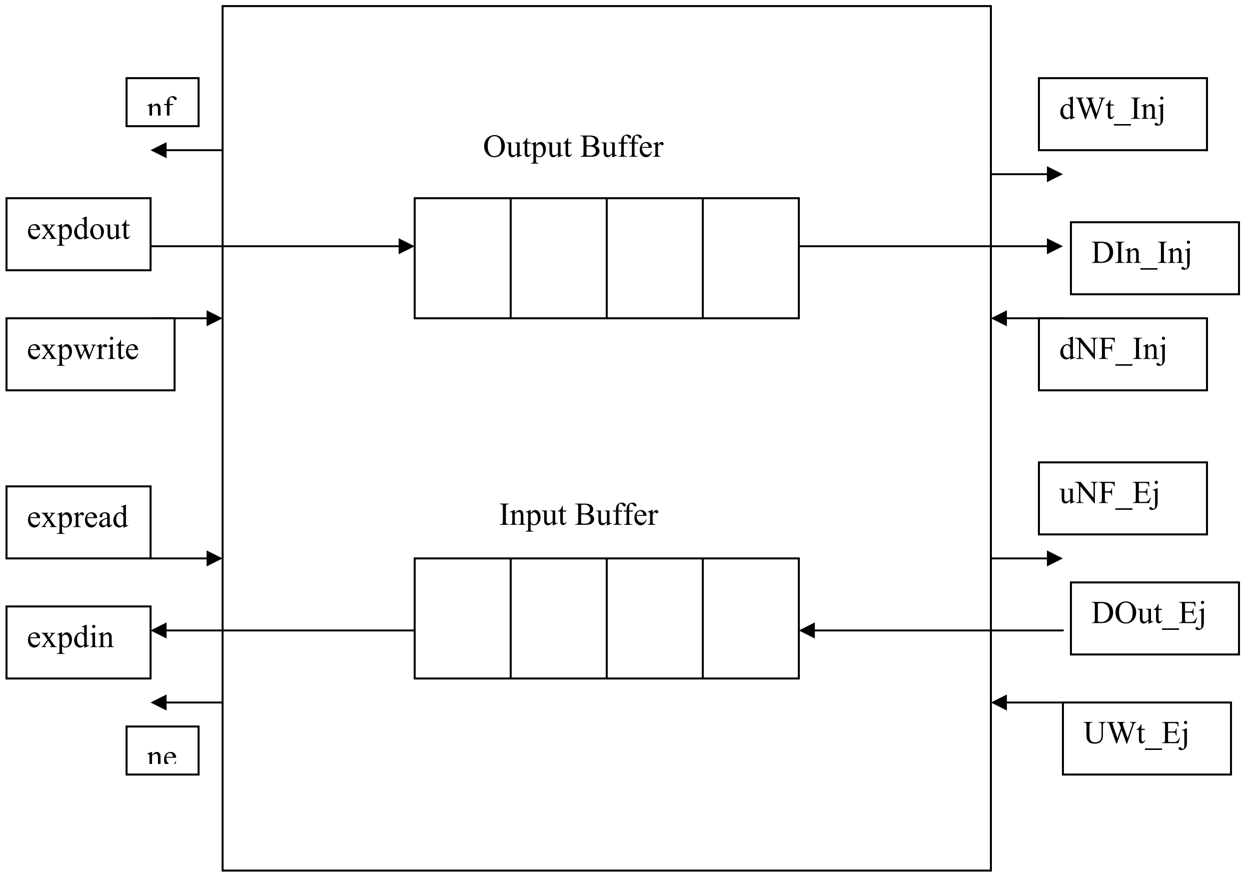


Figure 4.3. Network Interface for RISC8 in a Multicomputer Configuration

The network interface has the responsibility to convert 8 bit data into 10 bit flit data to send it to the router and convert 10-bit flit data into 8 bit data and send it to the processor. When sending data from network interface to processor bits 2 and 3 of the PortA will be used to indicate if the data is a header, a tail or just data.

4.3.2 LEON

For LEON to be part of a Multicomputer system there have to be proper signals to interface to the network through a network interface. LEON provides the flexibility to connect the network interface at different levels differentiated by the closeness to the processor.

Network interface as a Coprocessor:

Connecting network interface as a coprocessor would be the best option as it allows rapid transfer of data, but it has been realized that the VHDL model does not allow the coprocessor to interact with the outside world. So in order to connect a network interface as a coprocessor the coprocessor signals have to be taken out of the core. The following coprocessor signals have to be taken out for use

rst	: Reset
clk	: main clock
holdn	: pipeline hold
cpi	: data and control into coprocessor
cpo	: data and control out of coprocessor

‘cpi’ is a bunch of signals containing data bus into the coprocessor and control signals for pipeline control, exception indication from the pipeline and exception acknowledge from integer unit. ‘cpo’ is a bunch of signals containing the data bus going out from the coprocessor, exception indication and condition codes from coprocessor. There are two instructions, CPOP1 and CPOP2 in the instruction set of SPARC V8 [3], which will define the operations done in the coprocessor or the network interface in this case. There can be 32 registers in the coprocessor, which can be addressed for loading and storing form memory using coprocessor load and store instructions.

Network interface as an I/O device:

An alternative is to connect the network interface as an Input/Output device. The I/O is memory mapped and so the access is same as memory and capable of addressing up to 512MB of memory mapped I/O. The I/O access is programmed through the Memory Configuration Register1 (MCR1) in the memory controller. The description of various fields in MCR1 is provided in the appendix. To ensure proper connection and operation of the I/O device to LEON processor core, memory configuration register 1 has to be appropriately set.

- [19]: I/O enable. If set, the access to the memory bus I/O area are enabled.
- [23:20]: I/O wait states. Defines the number of wait states during I/O accesses (“0000”=0,“0001”=1, “0010”=2,..., “1111”=15).
- [28:27]: I/O bus width. Defines the data with of the I/O area (“00”=8, “01”=16, “10”=32).

As it can be seen bits 28-27, 23-20 and 19 of the memory configuration register are to be set accordingly. The address range for of I/O is Ox20000000 to Ox 3FFFFFFF.For the purpose this thesis we have set the wait states to be zero([23:20] = “0000”) and the I/O bus width to be 32bits ([28:27] = “10”). SPARC assumes that input/output registers are accessed via load/store alternate instructions, normal load/store instructions, coprocessor instructions, or read/write ancillary state register instructions.

4.4. Communication Network

The communication network is formed using routers described in Verilog HDL.

4.4.1. Functional Description

The basic features are

- Buffering at both input and output is modeled inside the router.
- Each virtual channel may be constructed with several lanes for performance reasons rather than for deadlock prevention.
- All virtual channel in the same direction share the same physical channel, unless specified otherwise.
- Width of the physical channel is assumed to be equal to that of its flits.

4.4.2. Interface signals

The network interface interacts with the router using two unidirectional buses. One bus is used by node to inject into channel through router, the other ejects the data from the channel into the node through the router. The signals can be seen on the right side of the network interface in the Figure 4.3.

Signals to inject data into Router from Node

DIn_Inj : Data Injected from Node to Router

dWt_Inj: Write signal from Node to Router

dNF_Inj: Not full signal from Router to Node

Signals to eject data out of Router into Node

DOut_Ej: Data ejected out from Node to Router

uWt_Ej: Write signal from Node to Router

uNF_Ej: Not full signal from Router to Node

Only the NF signal is asserted the Node or Router will write to the other by asserting Wt signal. The data is sent in the form of a Flit of size 10 bits. The first flit is the header containing address and the rest of the flits are considered as data. The header is identified by a “01“ in MSB position and the tail data flit has “10” as the MSB. The Router directly monitors injector bus without latching it to detect a header. So care should be taken by the Network interface to put all zeros on the injector bus so that a previous value in the buffer does not mislead the Router.

4.5. The Multicomputer system

The nodes of the multicomputer system will contain the processor core and a network interface, which connect the processor to the Router. The neighboring Routers are connected with point-to-point links. The processor will have some on-chip memory and cache. In order to access the external memory the UART provided with the LEON processor could be used and the program or application could be downloaded as srecords. But this method delays the process, as the access is slow even though it saves the pins. With better technologies more memory can be located on chip and with higher pin counts the external access can have higher bandwidth.

The multicomputer system is as shown in the Figure 4.4 below

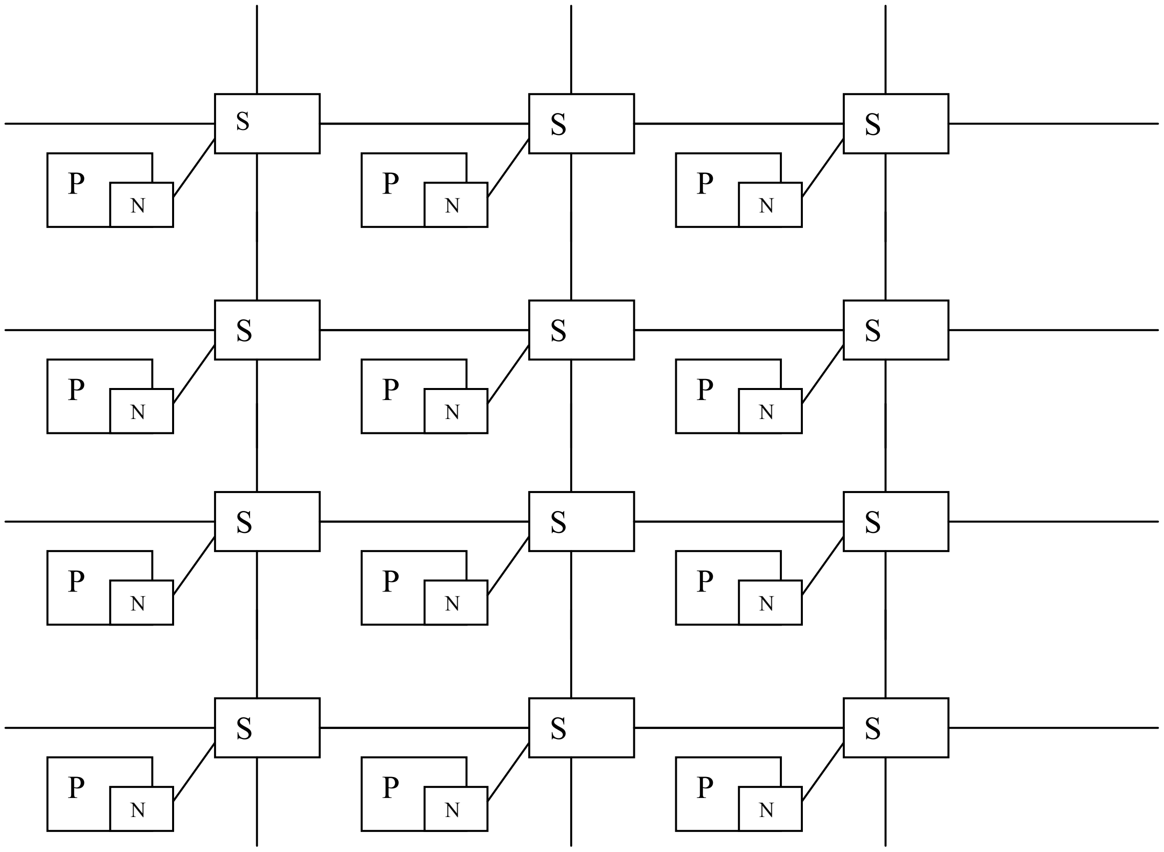


FIG 4.4. The multi-computer System

P: Processor NI: Network Interface S: Switch

In the next chapter we describe the experiments done to verify the use of IP core and results which to help us in understanding the issues with building the system.

5. EXPERIMENTS AND RESULTS

5.1. Simulation Results

Simulation have been utilized to verify the proper functionality of the cores and also estimate the performance in terms of number of cycles taken for the data transfer. During the course of this thesis two simulation tools were used. One is Silos, which does Verilog simulation only and the other is Active-HDL, which is capable of mixed simulation of Verilog and VHDL. This sections ends with a description of the procedure of using LEON VHDL model for simulation.

5.1.1. RISC8

One of the main purposes of this thesis is to build a working system that executes program code compiled from high-level languages like C.

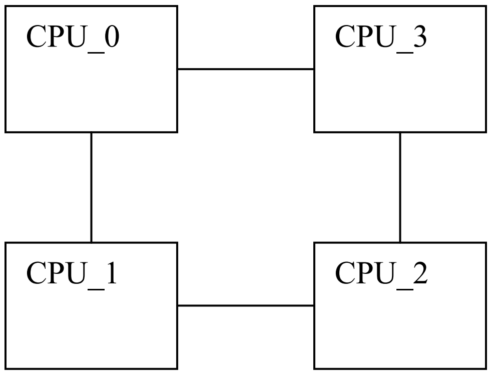


Figure 5.1. Multiprocessor Configuration using RISC8

In the process of building the multicomputer system initial experimentation was done using RISC8. Two Multiprocessor systems and one Multicomputer system using the RISC8 processor core have been built. The two implementations of RISC8 in

Multiprocessor configuration differed in the links between the processors. One had unidirectional links and the other had bi-directional links between processors. The general structure is illustrated in the Figure 5.1 above. The purpose of building the multiprocessor configurations was to test the core for running simple to complex problems. Also the communication capability has been tested.

5.1.1.1. Apparatus

In order to perform the experiment by executing an algorithm we have to generate instructions, which are subset of the RISC8 Instruction Set. Since RISC8 is binary compatible with Microchip's 16c57 processor Instruction Set we obtained MPLAB, which is a Windows-based Integrated Development Environment (IDE) for the Microchip Technology Incorporated [8] PICmicro micro controller families. MPLAB allows us to write, debug, and optimize PICmicro applications for firmware product designs. MPLAB includes a text editor, simulator, and project manager.

The output from MPLAB is a hex file, which is to be converted to a memory BFM in Verilog for simulation purposes. For this purpose a C- program, hex_pram.c (given in the Appendix), has been developed as part of this thesis. In order to perform the simulations Silos simulation tool has been used which was available through the services provided in CAAD lab.

5.1.1.2. Experiment-1

The experiment has been performed on the multiprocessor configuration shown in the Figure 5.2, with unidirectional links between the processors.

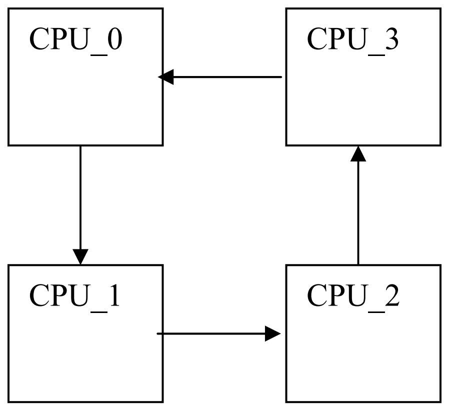


Figure 5.2. Multiprocessor configuration for Experiment-1

Problem

The experiment was intended to verify the processing capability and communication capability of RISC8. The summary of the operation is given under operations and actual data transfer among the registers of the RISC8 is shown in Code (referring to the instruction code).

Note: PortA is the input port, PortB is the output port and W is the accumulator through which all the instructions pass data to ports.

Results: The rest of this section illustrates the code being executed by the four processors with the help of Figures and assembly code.

CPU_0

Operation: Perform 1 + 2 and send it out to CPU_1

code:

```
mem[0]=12'b1100_0000_0001; //load 8'h01 into w
mem[1]=12'b0000_0010_0110;//w --> portb
mem[2]=12'b1100_0000_0010; //load 8'h02 into w
mem[3]=12'b0001_1110_0110; //Add w + portb --> portb
mem[4]=12'b0000_0000_0000;//NOP
mem[5]=12'b0000_0000_0000;//NOP
mem[6]=12'b0000_0000_0000;//NOP
mem[7]=12'b0000_0000_0000;//NOP
mem[8]=12'b0000_0000_0000;//NOP
mem[9]=12'b0000_0000_0000;//NOP
mem[10]=12'b0000_0000_0000;//NOP
mem[11]=12'b0000_0000_0000;//NOP
mem[12]=12'b0000_0000_0000;//NOP
mem[13]=12'b0000_0000_0000;//NOP
```

Simulation result:

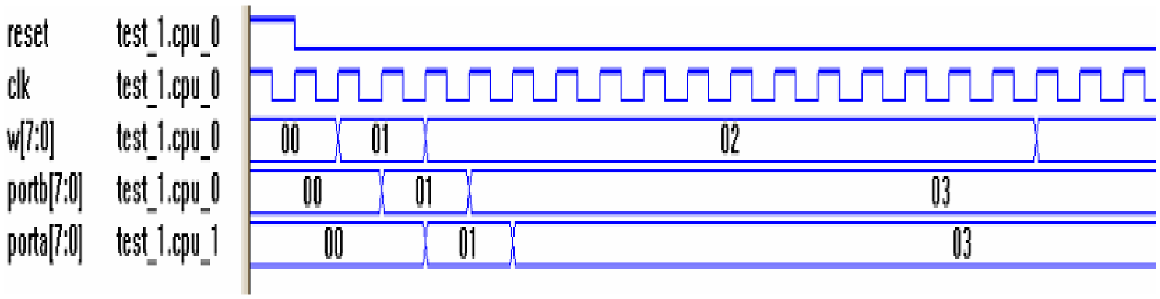


Figure 5.3 Simulation result-1, Experiment-1

CPU_1

Operation: Calculate 1 + 3 and later when CPU_1 sends its data (1 + 2) add it the previous result (1 + 3) and send it out to CPU_2

Code :

```
mem[0]=12'b1100_0000_0001; //load 8'h01 into w
mem[1]=12'b0000_0010_0110;//w --> portb
mem[2]=12'b1100_0000_0011; //load 8'h03 into w
mem[3]=12'b0001_1110_0110; //Add w + portb --> portb
mem[4]=12'b0000_0000_0000;//NOP
mem[5]=12'b0010_0000_0101;//w <-- porta
mem[6]=12'b0001_1110_0110; //Add w + portb --> portb
mem[7]=12'b0000_0000_0000;//NOP
mem[8]=12'b0000_0000_0000;//NOP
mem[9]=12'b0000_0000_0000;//NOP
mem[10]=12'b0000_0000_0000;//NOP
mem[11]=12'b0000_0000_0000;//NOP
mem[12]=12'b0000_0000_0000;//NOP
mem[13]=12'b0000_0000_0000;//NOP
```

Simulation result:

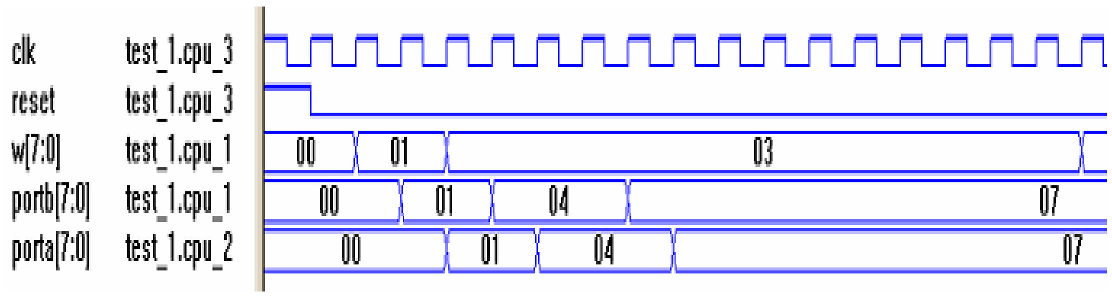


Figure 5.4 Simulation result-2, Experiment-1

CPU_2:

Operation:

Calculates 1 + 1 and sends the result to CPU_3, later when data arrives form CPU_1, it is sent to the CPU_3.

Code:

```
mem[0]=12'b1100_0000_0001; //load 8'h01 into w
mem[1]=12'b0000_0010_0110;//w --> portb
mem[2]=12'b1100_0000_0001; //load 8'h01 into w
mem[3]=12'b0001_1110_0110; //Add w + portb --> portb
mem[4]=12'b0000_0000_0000;//NOP
mem[5]=12'b0000_0000_0000;//NOP
mem[6]=12'b0000_0000_0000;//NOP
mem[7]=12'b0000_0000_0000;//NOP
mem[8]=12'b0010_0000_0101;//w <-- porta
mem[9]=12'b0000_0010_0110;//w --> portb
mem[10]=12'b0000_0000_0000;//NOP
mem[11]=12'b0000_0000_0000;//NOP
mem[12]=12'b0000_0000_0000;//NOP
mem[13]=12'b0000_0000_0000;//NOP
```

Simulation result:

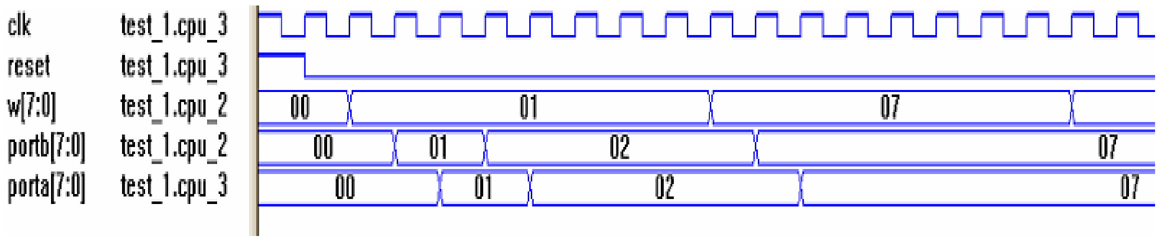


Figure 5.5. Simulation result-3, Experiment-1

CPU_3

Operation:

Calculates 2 + 3 and stores it in portb, later when data (1 + 1) arrives form CPU_2 it is added to the above result (2 +3) in portb and stored back into portb (07). When the second and last data arrives from CPU_2 (07) it added to the previous result in portb (07) and stored back into portb (0E).

Code:

```
mem[0]=12'b1100_0000_0010; //load 8'h02 into w
mem[1]=12'b0000_0010_0110;//w --> portb
mem[2]=12'b1100_0000_0011; //load 8'h03 into w
mem[3]=12'b0001_1110_0110; //Add w + portb --> portb
mem[4]=12'b0000_0000_0000;//NOP
mem[5]=12'b0010_0000_0101;//w <-- porta
mem[6]=12'b0001_1110_0110; //Add w + portb --> portb
mem[7]=12'b0000_0000_0000;//NOP
mem[8]=12'b0000_0000_0000;//NOP
mem[9]=12'b0000_0000_0000;//NOP
mem[10]=12'b0000_0000_0000;//NOP
mem[11]=12'b0010_0000_0101;//w <-- porta
mem[12]=12'b0001_1110_0110; //Add w + portb --> portb
mem[13]=12'b0000_0000_0000;//NOP
mem[14]=12'b0000_0000_0000;//NOP
mem[15]=12'b0000_0000_0000;//NOP
mem[16]=12'b0000_0000_0000;//NOP
mem[17]=12'b0000_0000_0000;//NOP
```

Simulation result:

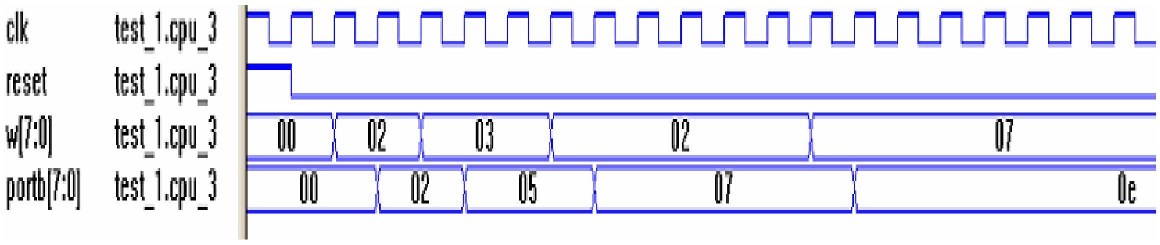


Figure 5.6. Simulation result-4, Experiment-1

Thus from the above Figures the computation and communication capability of the processor is established.

5.1.1.3 Experiment-2

The unidirectional link was changed to a bi-directional bus and an I/O module was added to handle PE-to-PE communication. The configuration is shown in Figure 5.7 below. This experiment is conducted using implementation-1 of RISC8.

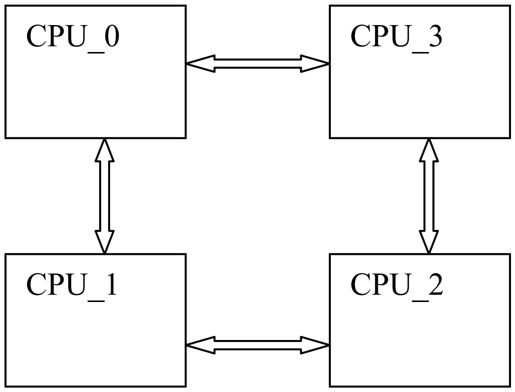


Figure 5.7. Multiprocessor configuration for Experiment-2

Moving Average Window Problem: The problem involved computing the average of a 2-Dimensional image data. The data is an 8 X 8 matrix and the window size for finding the average was 3 X 3. All the elements in that window were added and stored into the position occupied by the top left corner element. For the elements on the edges the window cannot have nine elements so the rest of the elements are assumed to be zeros. The data is distributed equally among four processor thus each processor contains a 4X 4 matrix. The Figure 5.8 below illustrates the explanation above by showing a sample data and the window for the first element and the edge element of the result matrix.

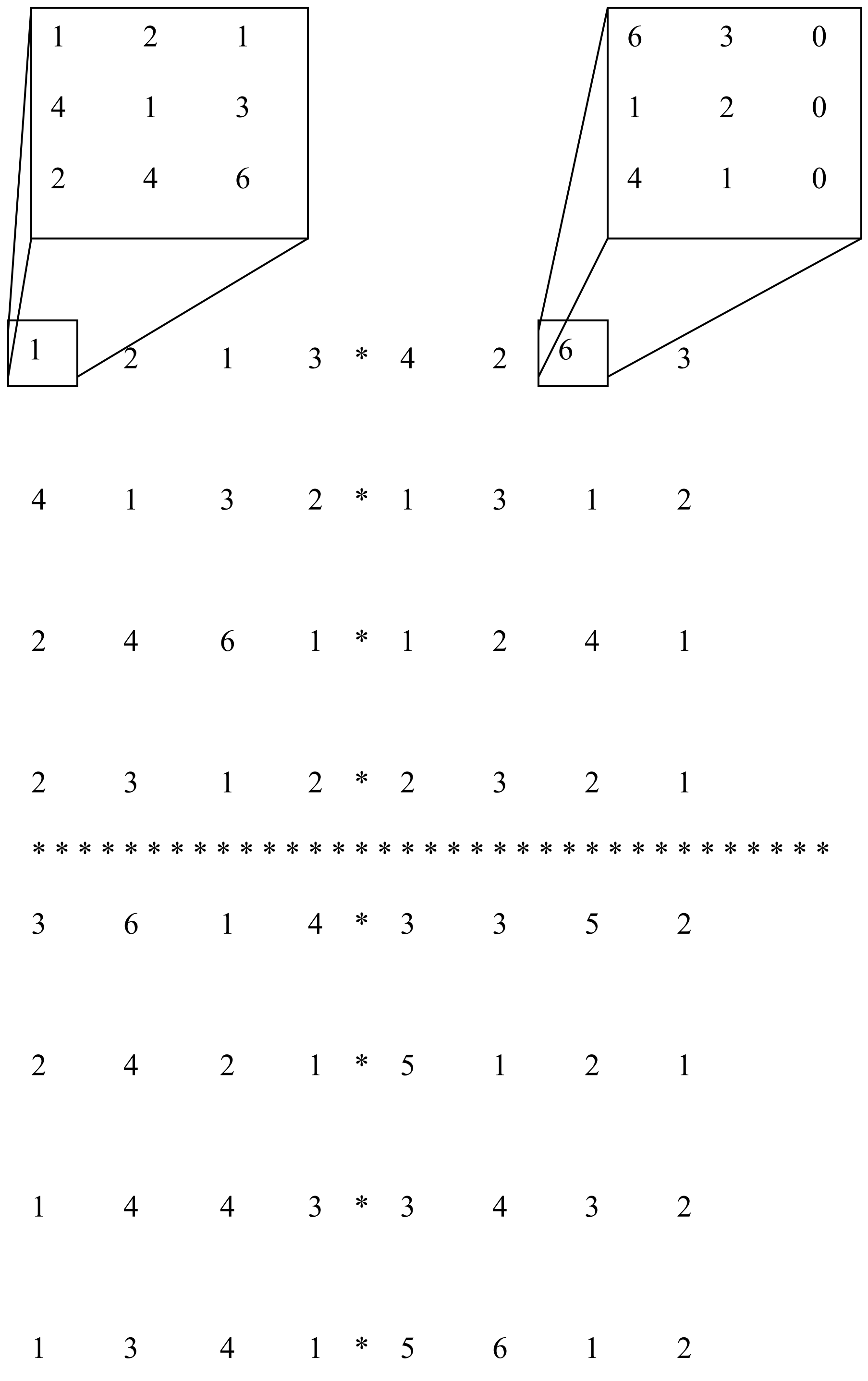


Figure 5.8. Illustration of Moving Average Window Problem

The windows in Figure 5.8 above show the elements under consideration for each window. The small squares to which they are mapped show the position in the final result matrix, which has the same size as the original data. The partitions show the data distribution among four processors. Each processor can calculate 16 averages of which four are complete sums and 12 are partial sums. These partial sums require data from other processors so each processor requests communication. Out of the 12 partials sums only five of these will be completed within each processor thus each processor will have nine complete sums by the end of the algorithm. Thus, we have 36 complete sums for an 8 X 8 matrix and window size of 3 X 3. Modifications were done to the RISC8 processor to have two implementations of RISC8, which differed in the way they implement communication.

Result

The moving avaerage problem algorithm was executed and each processor generated four complete sums and 12 partial sums. Communication events among these four processor then enabled the completion of the rest of the five sums in each processor.

Observations:

Receiving data:

It takes two clock cycles to get the value on the port into the register PortA, which is part of the register file.

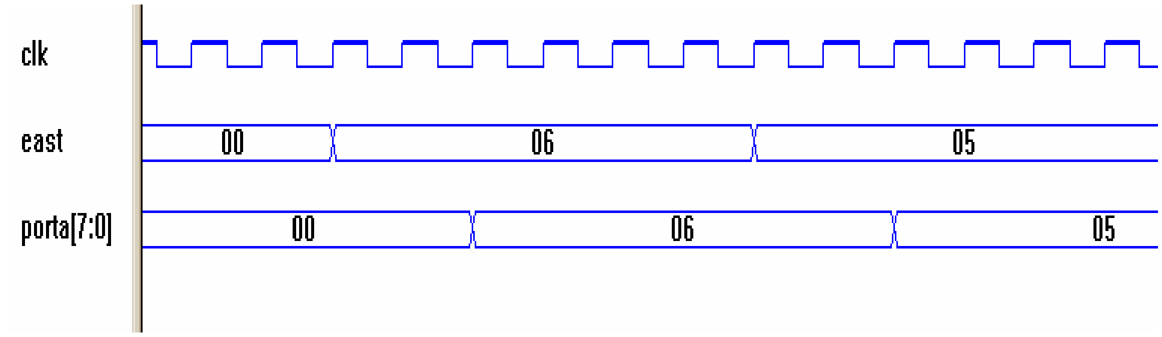


Figure 5.9. Simulation result-1, Experiment-2

The Figure 5.9 above illustrates that the value on east port takes two clock cycles to come onto the register porta.

Sending data:

In order to send one 8-bit data, Implementation-1 takes three clock cycle

- In first cycle PortC is used to conFigure ports
- In the second cycle data is stored in accumlator
- In third cycle data is sent to output port register from accumlator
- In fourth cycle data will appear on port

This above mentioned sequence is verified in the Figure 5.10 below

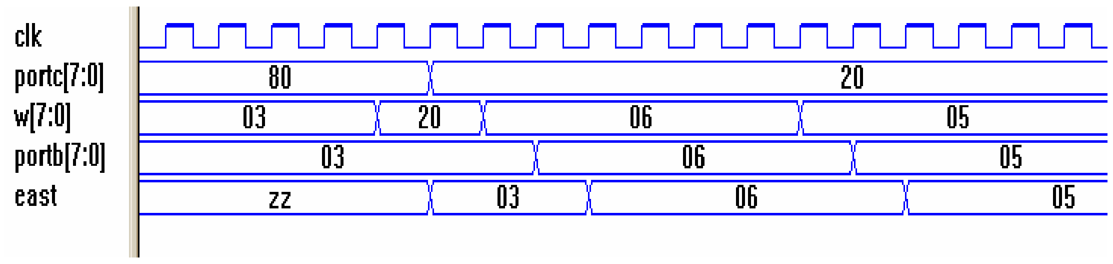


Figure 5.10. Simulation result-2, Experiment-2

5.1.1.4. Multicomputer System using RISC8

The Multicomputer system built is shown in the Figure 4.4. It has a node with a Processor and a Network Interface to connect the processor to the network of Routers. The expansion ports of the RISC8 are used for the purpose of interfacing to the network. The expansion ports have been explained in section 4.2.1. The functionality of the system has been tested in three phases. First the Processor- Network Interface side is tested for sending and receiving packets. Next the Network Interface and the Network are tested for sending packets to random destinations. In the last phase the whole System is integrated and tested by sending packets from one source to random destinations.

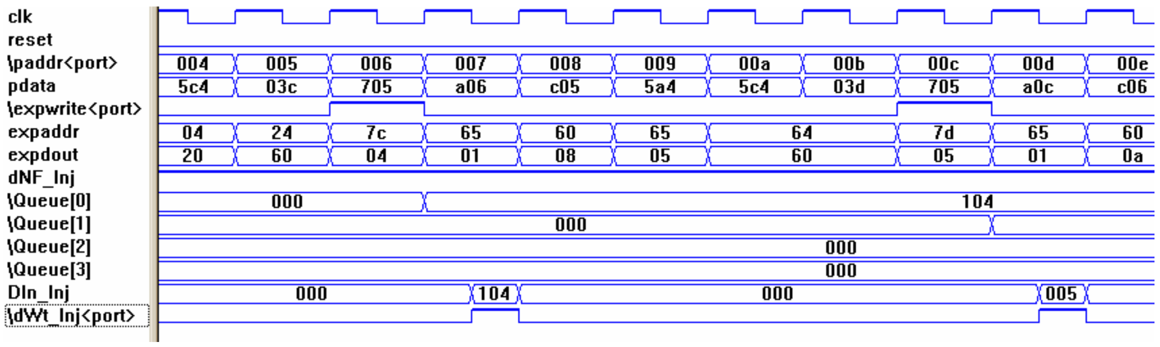
Phase1: Test between the Processor and the Network Interface has been conducted by testing the send and read separately

Send : The source executes the program shown below in the code. All the other Processors in the Network consume the packet as soon as it appears in the Network Interface.

C Code:

```
// Directives to the compiler
#pragma char porta  @ PORTA  // porta points to PORTA
#pragma char portb  @ PORTB // portb points to PORTB
#pragma char portc  @ PORTC // portc points to PORTC
#pragma char header @ 0x7C   // header indicates that the data in expdout[7:0] is header
#pragma char data   @ 0x7D  // data indicates valid data in expdout[7:0]
#pragma char tail   @ 0x7E   // tail indicates that expdout contains tail
#pragma char read_d @ 0x7F // read_d indicates read from input buffer
#pragma bit nf      @ PORTA.0 // bit 0 of PORTA is Not Full signal from input buffer
#pragma bit ne      @ PORTA.1 // the bit 1 is the Not Empty signal
void main(void){
// code to send a packet
  while(nf == 0){
    ;
   }
  header = 4; //the value 0x 04 is sent as header, 0x104 by network interface
  while(nf == 0){
    ;
    }
  data = 5; // value 0x 05 is sent as 0x 005
  while(nf == 0){
     ; }
  tail = 6; // value 0x 06 is sent as tail, 0x 206 by network interface
}// ends the main program
```

Result : The above C-program was compiled using MPLAB to a hex format, which is converted into a memory BFM in Verilog using the "hex_pram.c" file developed by us. The Figure 5.11 below shows the data being written into the buffer of network interface. When the address 0x 7C appears in the 'expaddr' and 'expwrite' is asserted, the data in the expdout is considered as the header and a '01' is added to the data. This data is then stored in the buffer indicated by Queue in the timing diagram.



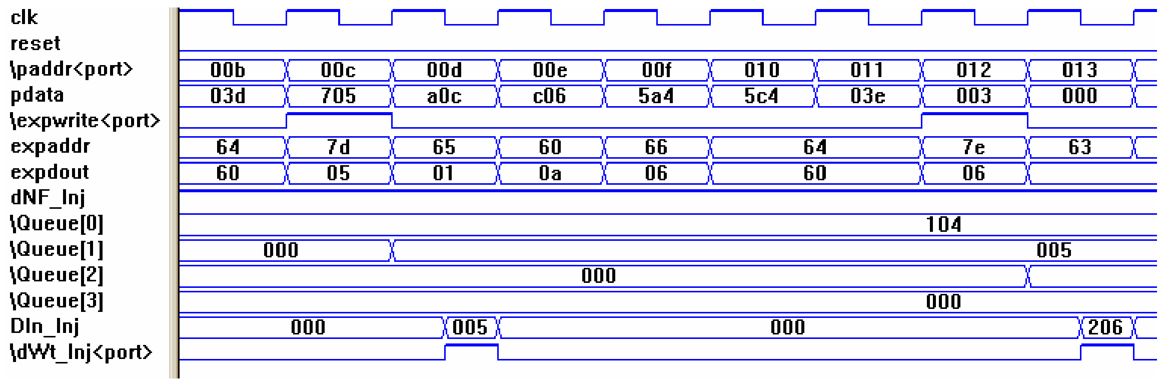


Figure 5.11. Timing diagram illustrating the send sequence

Read: To read data from the network interface the processor uses 'ne' signal to check for the input buffer to have new data. It then asserts the 'expread' signal, which will output data onto 'expdin' bus.

Code : The following code will read a data form the

```
#pragma char porta  @ PORTA
#pragma char portb  @ PORTB
#pragma char portc  @ PORTC
#pragma char header @ 0x7C
#pragma char data   @ 0x7D
#pragma char tail   @ 0x7E
#pragma char read_d @ 0x7F
#pragma bit nf      @ PORTA.0
#pragma bit ne      @ PORTA.1
void main(void){
// code to read a packet
```

```
  //wait for the input buffer to have a value, indicated by ne =1
   while(ne!=1){
     ;
    }
    portb = read_d; // read data
   while(ne !=1){
     ;
    }
   portb = read_d;


   while(ne !=1){
    ;
    }
   portb = read_d;
} //end of main program
```

Results:

The Figure 5.12 shows the timing diagram of the sequence in reading a data value from network. The Router passes the data on 'DOut_Ej' bus, which is stored in the input buffer of network interface. The ‘ne’ signal is set indicating availability of data in the buffer.

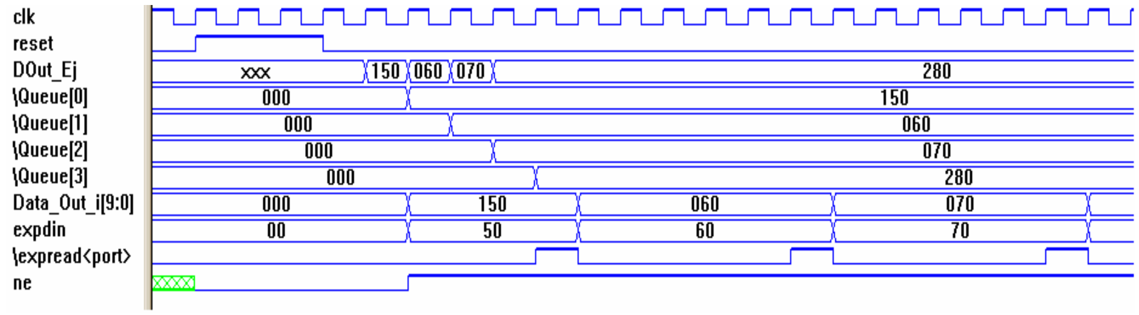


Figure 5.12. Timing diagram illustrating the read sequence

The processor senses the 'ne' signal and reads the data from the input buffer by asserting the 'expread' signal. As the Figure 5.13 above shows three reads occur.

From the above results we have established that the RISC processor is suitable of interfacing to the network through the network interface.

Phase2:

The network is formed from an array of routers, which are fully connected. In order to have the flexibility in changing the array size and also to avoid rewriting the connections in the top module for simulations, a C-program has been written that will generate the array and also define the connection among the routers and the network interfaces. This C-file is given in the Appendix. Simulations were done to determine the working of the interface signals, which helped in the network interface development.

Simulaiton1:

The first simulation was done to check if the connectivity among the routers is proper. Checking if the path selection resulted in the smallest route did this. Four packets were sent from node 0 to node 49. The path taken was node 0 to node 59, node 59 to node 49. All the packets traveled through the negative Y direction.

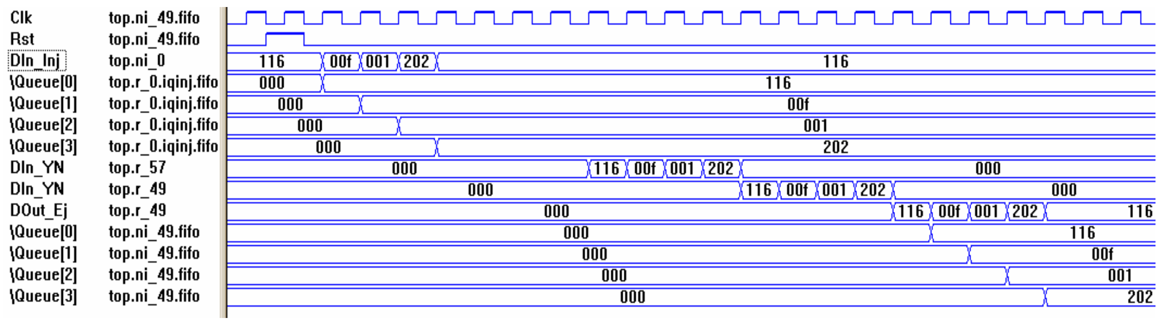


Figure 5.13. Depicts the Path Taken By Four Packets From node 0 to node 49

Simulation 2:

The next simulation was done to check if there were any packets lost. We performed a simple experiment of sending 8 data flits each to all the nodes from one single node. The successful transmission ensures reliability to some extent. To distinguish among packets sent to each node, we encoded the position of the node with respect to the array and sent it in each packet. It is clear from Figure 5.14 that packet 0x 100, 0x 002, 0x003 are destined to node 00.

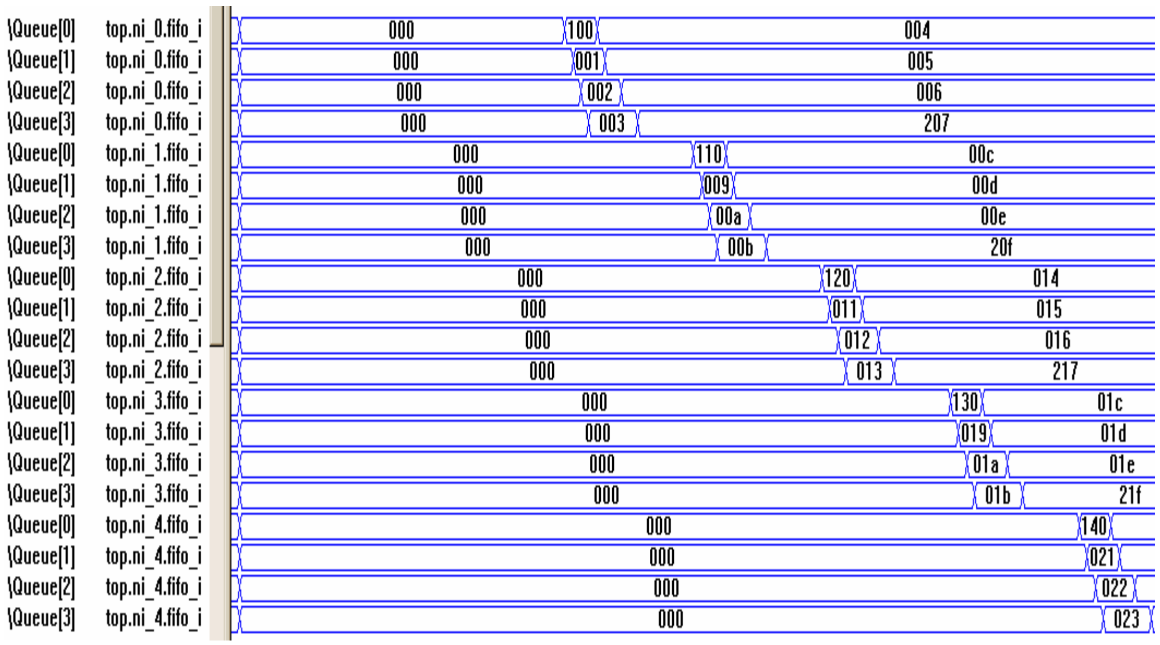

\Queue[0] top.ni_0.fifo_i 000 100 004
\Queue[1] top.ni_0.fifo_i 000 001 005
\Queue[2] top.ni_0.fifo_i 000 002 006
\Queue[3] top.ni_0.fifo_i 000 003 207
\Queue[0] top.ni_1.fifo_i 000 110 00c
\Queue[1] top.ni_1.fifo_i 000 009 00d
\Queue[2] top.ni_1.fifo_i 000 00a 00e
\Queue[3] top.ni_1.fifo_i 000 00b 20f
\Queue[0] top.ni_2.fifo_i 000 120 014
\Queue[1] top.ni_2.fifo_i 000 011 015
\Queue[2] top.ni_2.fifo_i 000 012 016
\Queue[3] top.ni_2.fifo_i 000 013 217
\Queue[0] top.ni_3.fifo_i 000 130 01c
\Queue[1] top.ni_3.fifo_i 000 019 01d
\Queue[2] top.ni_3.fifo_i 000 01a 01e
\Queue[3] top.ni_3.fifo_i 000 01b 21f
\Queue[0] top.ni_4.fifo_i 000 140
\Queue[1] top.ni_4.fifo_i 000 021
\Queue[2] top.ni_4.fifo_i 000 022
\Queue[3] top.ni_4.fifo_i 000 023

Figure 5.14 Final Destination of Packets

Thus the Network side has been verified . The last phase is to verify the whole system

Phase 3: The last phase in the testing has been to connect all the components in a top module and test it by sending packets to random locations.

Code: The source Processor sends one header and eight data packets including the tail by executing the following code

```
/ /code to generate file to be sent to ni
#pragma rambank 1
int8 i, x, y, temp;
```

```
//temp3, image_index, mask_index, y_index, m1, m2, n1, n2, mask[9];
#pragma char porta  @ PORTA
#pragma char portb  @ PORTB
#pragma char portc  @ PORTC
#pragma char header @ 0x7C
#pragma char data   @ 0x7D
#pragma char tail   @ 0x7E
#pragma char read_d @ 0x7F
#pragma bit nf      @ PORTA.0
#pragma bit ne      @ PORTA.1
void main(void){
// code to send packet to consecutive locations
   temp = 0;
   for(x=0;x<2;x++)
   {
    for(y=0;y<2;y++)
    {
      while(nf!=1)
      {
        ;
      }
      header = x * 16 + y ;       //send the header
      while(nf!=1)
```

```
        {
         ;
        }
//data1
        data = temp  ;
         while(nf!=1)
        {
         ;
        }
//data2
        data = temp + 1;


        while(nf!=1)
        {
         ;
        }
//data3
        data = temp + 2;


        while(nf!=1)
        {
         ;
        }
```

```
//data4
     data = temp + 3;


     while(nf!=1)
     {
       ;
     }
//data5
     data = temp + 4;


     while(nf!=1)
     {
       ;
     }
//data6
     data = temp +5;


     while(nf!=1)
     {
       ;
     }
//data7
     data = temp  + 6;
```

```
       while(nf!=1)
       {
         ;
       }
//data8
      tail = temp + 7;


     }//for ends
    }//for ends


//consume all packets that are sent
    while(1)
    {
      while(ne!=1)
      {
        ;
      }
      portb = read_d;
    }


}
```

All the other processors in the network execute the following code to consume all the packets that are sent to them.

Code:

```
// code to consume all the packets sent
#pragma char porta  @ PORTA
#pragma char portb  @ PORTB
#pragma char portc  @ PORTC
#pragma char header @ 0x7C
#pragma char data   @ 0x7D
#pragma char tail   @ 0x7E
#pragma char read_d @ 0x7F
#pragma bit nf      @ PORTA.0
#pragma bit ne      @ PORTA.1

void main(void){
// code to continuously read a packet
  while(1){
    while(ne==0){
    ;
    }
   portb = read_d;// the flits read are displayed on PortB
  }
```

}// end of main program

The Figure 5.15 below shows the Source processor 24 sending the data 0x00 with an address of 0x7C implying header. The Network Interface recognizes this as the header and concatenates a ‘01’ to the 8-bit value to make it a 10-bit flit. The subsequent data are sent to 0x7D implying data flits and so a ‘00’ is appended to the data. For the last data ‘10’ is appended when the address is 0x7E as shown in Figure 5.16.

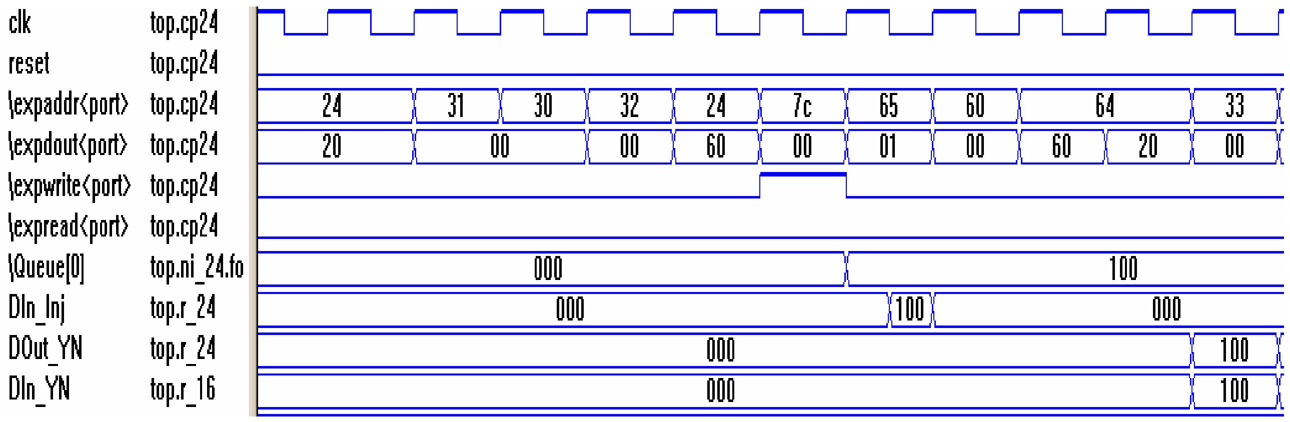


Figure 5.15. System testing, header flit formation

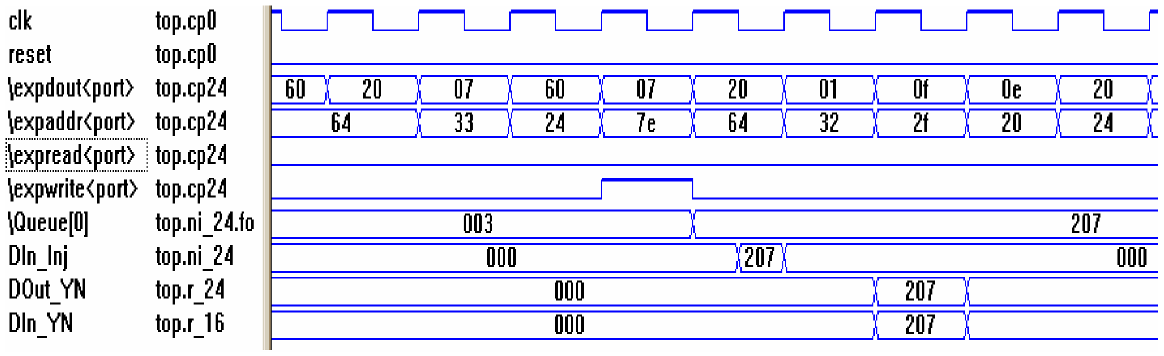


Figure 5.16. System testing, tail flit formation

The Figure below shows all the flits reaching the destination. All the packets take the same route. The values taken are displayed on PortB.

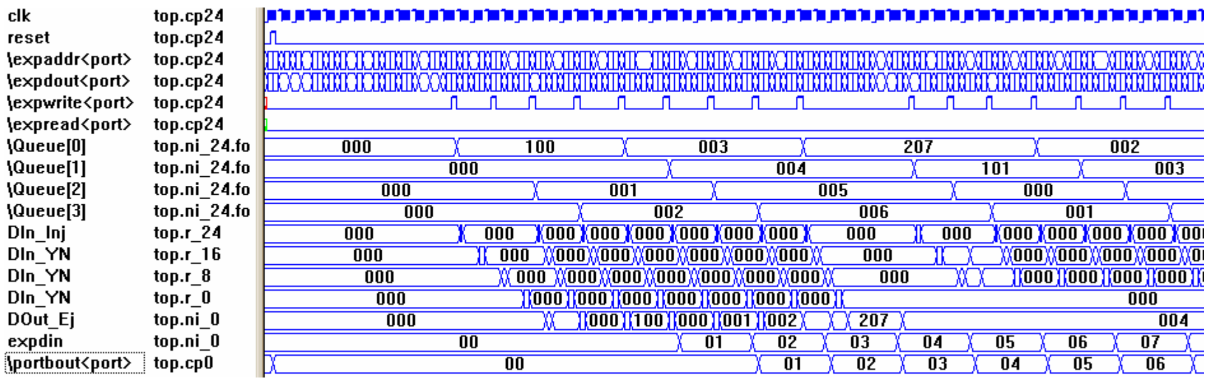

clk top.cp24
reset top.cp24
\expaddr<port> top.cp24
\expdout<port> top.cp24
\expwrite<port> top.cp24
\expread<port> top.cp24
\Queue[0] top.ni_24.fo 000 100 003 207 002
\Queue[1] top.ni_24.fo 000 004 101 003
\Queue[2] top.ni_24.fo 000 001 005 000
\Queue[3] top.ni_24.fo 000 002 006 001
DIn_Inj top.r_24
DIn_YN top.r_16
DIn_YN top.r_8
DIn_YN top.r_0
DOut_Ej top.ni_0 000 100 000 001 002 207 004
expdin top.ni_0 00 01 02 03 04 05 06 07
\portbout<port> top.cp0 00 01 02 03 04 05 06

Figure 5.17. System testing, packets reaching destination

Thus in three phases the proper functionality of the Multicomputer System is verified.

Summary

Experiments have been done and the following results have been observed

- Gained experience with the core, experiment 1 and 2
- Verified computational and communication capability of core, experiment1
- Verified the functionality of modified core, experiment 2
- To verify the individual components of the System, experiment on multicomputer Configuration
- To verify the complete Multicomputer system, experiment on multicomputer Configuration

5.1.2. LEON

To perform simulations the LEON VHDL model has to be compiled to the simulation tools format, which in this case is Active-HDL. Some changes had to be made in some of the modules of the model in order to ensure proper compilation. VHDL model have to be compiled in the following order

- amba.vhd
- target.vhd
- device.vhd
- config.vhd
- sparcv8.vhd (the library IEEE.std_logic_arith.all had to be included in this module)
- iface.vhd
- macro.vhd
- debug.vhd
- ambacomp.vhd
- multlib.vhd
- tech_generic.vhd
- tech_atc35.vhd
- tech_atc25.vhd
- bprom.vhd
- tech_virtex.vhd
- tech_fs90.vhd
- tech_map.vhd

- fpulib.vhd
- fp1eu.vhd
- mul.vhd
- div.vhd
- clkgen.vhd
- rstgen.vhd
- iu.vhd
- regfile.vhd
- icache.vhd
- dcache.vhd
- cachemem.vhd
- acache.vhd
- cache.vhd
- proc.vhd
- apbmst.vhd
- ahbarb.vhd
- lconf.vhd
- wprot.vhd
- ahbtest.vhd
- ahbstat.vhd
- timers.vhd (had to include IEEE.std_logic_arith.all and also changed std_logic_vector(TPREC(..)) to conv_std_logic_vector(TPREC(..)))
- uart.vhd (the above change should be done in this module also)

- irqctrl.vhd
- irqctrl2.vhd
- ioport.vhd
- mctrl.vhd
- pci_is.vhd
- pci_arb.vhd
- pci_esa.vhd
- mcore.vhd
- leon_pci.vhd
- leon.vhd

Mixed simulation has been done and so the VHDL module 'leon' is instantiated in the Verilog top module just as any other Verilog module is done with the module name and an instance name after it. The example of 'leon' module is given below

```
leon L1(resetn, clk, errorn, address, data, ramsn, ramoen, rwen,
     romsn, iosn, oen, read, writen, brdyn, bexcn, pio, wdogn,
     test);
```

5.2. Synthesis Results

Synthesis of the design is done in the pre-layout stage as illustrated by Figure 2.1. Synthesis is used to get an estimate of the timing and area of the design before the layout is done in order to check if the design complies with the constraints set. A synthesis tool uses wire load models (wlm) to calculate the capacitance, resistance, and wire area within a synthesis block. These values are only estimates since synthesis tools cannot predict the actual placement of cells and routing of nets.

5.2.1. Technology Libraries

A technology library is compiled using a library compiler into the format of the tool; in this case with Synopsys it is ".db". The library contains the information such as attributes and environment in which the process is performed. The environment is the process operating conditions, which determine the approximate delay values and area of the design. The interconnect model is defined by a "tree_type" attribute, which can take three values as best_case_tree, worst_case_tree or balanced_tree. In the worst_case_tree the load pin is most distant from the driver, in best_case_tree type the driver is next to the load pin and in the balanaced_tree type the load pins are separate and at equal distance from the driver. The synthesis results of area are measured in cell units, which is 11.54 um^2 for G-11p process technology.

For the purpose of Logic synthesis G11-p technology libraries have been used. G11-p cell based ASICs are LSI Logic's highest performance 2.5v and 1.8v products [4]. It can have up to 8.1 million usable gates with effective gate length of 0.18um (Leff), 8Mbits RAM on chip and up to 6 metal layer achieving system on chip. The next level of process technology is the G12-p where the gate count reaches 33 million enabling multiple systems on chip with a gate length of 0.13um and 16Mbits of RAM. Table 5.1 below gives the technology constants of G11-p technology.

Table 5.1. Technology constants of G11-p

Quantity	**Value**	**Units**
Standard Load	0.0111995	pF
Grid Size	0.9	mm
Cell Width (Standard Cell)	12.6	mm
Cell Unit*	11.34	mm^2

* A cell unit is the area used by a cell one grid tall and a standard cell width wide.

5.2.2 RISC8

The hierarchy of the RISC8 CPU is given in the appendix. The following Table summarizes the results of synthesizing the unmodified RISC8 core

Table 5.2. Synthesis Results of RISC8 (unmodified) in G11-p process technology

Design / module name	Area (um^2)	%	Timing (ns)
cpu	741734.176	100.0	7.61
alu	16430.787	2.2	2.27
idec	12935.753	1.7	1.56
regs	565576.841	76.24	2.84
dram	563834.663	76	2.06
exp	18859.577	2.5	1.02

Observation:

- 75% of the cpu area is occupied by the memory
- All the other components except memory occupy 6.4%
- The logic around these components inside the cpu is 17.36 % (100 - 76.24 - 6.4)
- From the timing results we can say that the maximum frequency of operation could be approximately 130MHz (1 / 7.61ns)

Implementation-1

A communication module (io module) is added to the RISC8 core in Implementation-1. Table 5.3 below shows the area of this single module.

Table 5.3. Synthesis Results of I/O Module in Implementation-1 in G11-p process technology

Design / module name	Area (um^2)	Timing (ns)

Io	7803.222	0.18

Implementation-2

In implementation-2 the communication module is given three additional registers. The Table 5.4 below shows the area of this single module.

Table 5.4. Synthesis Results of I/O Module in Implementation-2 in G11-p process technology

Design / module name	**Area (um^2)**	**Timing (ns)**
Io	31166.187	0.42

Observation:

- Implementation-1 is 25% of implementation-2 in other words implementation-2 is 4 times larger than implementation-1.

The difference between implementation-1 and implementation-2 is the addition of three registers and also a state machine intended to do one, two or three 8-bit data transfers.

5.2.3. LEON

The description for the modules names of LEON VHDL model can be found in the appendix. The following Table gives the area of the various components that make up the 'leon' VHDL model after synthesis with the library "lcbg11p_nom.db" from LSI Logic.

Table 5.5. Synthesis Results of LEON in G11-p process technology

Design	**Area (um^2)**	**%**	**Timing (ns)**
Leon	1780211.87	100.00	0.80
Clkgen	308.871	0.00	0.12
Ahbarb	8876.787	0.49	0.81
Ahbmst	90967.053	5.10	7.00
Bprom	90804.687	5.10	4.14
Proc	1223437.261	68.00	2.00
Icache	82209.923	4.61	2.58
Dcache	280007.551	15.72	4.01
Iu	853623.897	47.95	6.96
Irqctrl	64018.228	3.59	0.70
Ioport	73441.906	4.12	0.89
Timers	137051.192	7.69	1.51
Uart	66652.617	3.74	0.88
ahbstat	31115.836	1.75	0.58
ahbtest	66520.476	3.73	7.00
rstgen	1366.086	0.00	0.20
mctrl	251249.466	14.11	2.33
Router	2460926.998	138.00	1.95
Ni	95303.833	5.35	1.23

Observation:

- The majority of the real estate is taken by processor (Proc), which contains the integer unit (Iu), data cache controller (Dcache) and the memory controller (Mctrl).
- Dcache and Icache refer to the cache controllers and not memories themselves. So the need for control logic is more in data cache than instruction cache.
- The worst critical path of 7.00 ns is obtained for Ahbtest and Ahbmst modules
- The critical path of 'leon' module is quite less at 0.80

Estimation of Multicomputer System area

An attempt to synthesize the whole system failed due to limitation of the tool. To get an estimate the following procedure has been used. The Multicomputer system is formed by replication of the Processor, Network Interface and the Router. By calculating the area of this building block an estimate can be obtained. In order to get an estimate of the interconnect area the following assumption was made. It is a fact that the interconnect area cannot reduce beyond a point with decrease in feature size as the resistance of the interconnect increases with decrease in area ($R = \rho * L/A$). Thus with decrease in feature size the ratio of interconnect area to cell area increases. Since the area of the building block includes the area of interconnect within each module the external and overhead interconnect area has to be estimated. Based on the above assumption the interconnect area can be estimated to be around 25% of the chip.

With a 1cm^2 chip we will have 0.25cm^2 for external and overhead interconnect and so 0.75cm^2 for the cells. With 4.27 mm^2 area for the building block in the case of

LEON, the size of the System can be 16 (17.56 rounded). With 3.23 mm^2 for RISC8 the System size is 24 (23.12 rounded).

5.2.4. Discussion

The general observation from the above synthesis results is about the percentage of area memory and Router occupy in the System. For RISC8 memory is almost 75% of the processor area. In such a scenario an increase in memory on chip would reduce the area for processors. So the number of processors needed on the chip limits the amount of memory. The Router also occupies a large area, which is comparable to the processors. So the System size would have to be a trade-off between the Communication Network size, memory on chip and the number of processors.

The off- chip memory access bandwidth depends upon the pins available for the chip used. Since RISC8 is 8-bit RISC architecture, there can be a large number of processors. But LEON cannot have the advantage, as LEON is 32-bit architecture. The off-chip access bandwidth is configurable in LEON but then reducing the access bandwidth would mean more number of cycles to access each instruction or data. LEON can also access or download program instructions as srecords from a UART but the access is obviously slow.

6. CONCLUSIONS

This thesis describes the work we have done in the context of building a system. The major contribution is

- Building of a single chip multicomputer
- Finding a processor core for experimentation
- Investigating the various network interfaces and building one based on the research
- Developing two configurations of RISC8, one with direct connection and the other with a network
- Investigating the LEON processor to find a way of connecting it to the network in order to form a multicomputer system
- Showing the environment, like the tools and software, that is need to build the system
- Obtaining simulation and synthesis results for the components that form the multicomputer system.
- Attempting to synthesize the whole system with the nodes and the network

From the above work we gained significant experience in building a system and building an environment for this purpose.

Future Scope

From the results we obtained it can be seen that with the present technology it has been possible to integrate a multicomputer system along with the communication network. However, we have also discussed the tradeoff in the amount of memory off-chip and the number of processors. Future process technology will enable integration of large systems with 100s of processors and large on-chip memory onto a single chip. From our failure to synthesize the whole system we realize that synthesis tools that can handle large designs are needed to compile large designs and produce area and timing results.

REFERENCES

1. Dana S.Henry and Christopher F. Joerg "A Tightly-Coupled Processor Network Interface", *Proceedings of the 5th ASPLOS,* October 1992, pp. 111-122.
2. LSI Logic Corporation Homepage available at, http://www.lsilogic.com.
3. JiriGaisler's Research Homepage available at, http://www.gaisler.com.
4. "Overview of G11™ Cell-Based Technologies", LSI Logic ASKK Documentation System, Copyright © 1983- 1999 LSI Logic Corporation.
5. Thomas Connan's Home Page available at, http://www.mindspring.com/~tcoonan/.
6. Michael John Sebastian Smith, "Application Specific Integrated Circuits ", *VLSI system series, Addison Wesley,* 1997*,* pages 16-17.
7. San-Won Lee, Yun-Seob Song, Soo-Won Kim, Hyeong-Cheol Oh, Woo-Jong Hahn "RAPTOR: A Single Chip Multiprocessor", *The First IEEE Asia Pacific Conference on ASICs*, 1999.Ap-ASIC '99 on pages 217-220.
8. MPLAB, Integrated Development Environment from Microchip available at, http://www.microchip.com.
9. Aldec Incorporation, Design Verification Company, available at, http://www.aldec.com.
10. OPENCORES organisation home page available at, http://www.opencores.org/.
11. Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson and Kunyung Chang," The Case of a Single Chip Multiprocessor", *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII),* pages 2 to 11.

APPENDIX

A.1. VHDL model:

TABLE A.1 LEON model hierarchy

Entity/Package	**File name**	**Function**
LEON	leon.vhd	LEON top level entity
LEON_PCI	leon_pci.vhd	LEON/PCI top level entity
LEON/MCORE	mcore.vhd	Main core
LEON/MCORE/CLKGEN	clkgen.vhd	Clock generator
LEON/MCORE/RSTGEN	rstgen.vhd	Reset generator
LEON/MCORE/AHBARB	ahbarb.vhd	AMBA/AHB controller
LEON/MCORE/APBMST	apbmst.vhd	AMBA/APB controller
LEON/MCORE/MCTRL	mctrl.vhd	Memory controller
LEON/MCORE/MCTRL/BPROM	bprom.vhd	Internal boot prom
LEON/MCORE/PROC	proc.vhd	Processor core
LEON/MCORE/PROC/CACHE	cache.vhd	Cache module
LEON/MCORE/PROC/CACHE/CACHEMEM	cachemem.vhd	Cache ram

LEON/MCORE/PROC/CACHE/DCACHE	dcache.vhd	Data cache controller
LEON/MCORE/PROC/CACHE/ICACHE	icache.vhd	Instruction cache controller
LEON/MCORE/PROC/CACHE/ACACHE	acache.vhd	AHB/cache interface module
LEON/MCORE/PROC/IU	iu.vhd	Processor integer unit
LEON/MCORE/PROC/MUL	mul.vhd	Multiplier state machined
LEON/MCORE/PROC/DIV	div.vhd	radix-2 divider
LEON/MCORE/PROC/FP1EU	fp1eu.vhd	parallel FPU interface
LEON/MCORE/PROC/REGFILE	regfile.vhd	Processor register file
LEON/MCORE/IRQCTRL	irqctrl.vhd	Interrupt controller
LEON/MCORE/IOPORT	ioport.vhd	Parallel I/O port
LEON/MCORE/TIMERS	timers.vhd	Timers and watchdog
LEON/MCORE/UART	uart.vhd	UARTs
LEON/MCORE/LCONF	lconf.vhd	LEON configuration

		register
LEON/MCORE/AHBSTAT	ahbstat.vhd	AHB status register

TABLE A.2 LEON packages

Package	**File name**	**Function**
TARGET	target.vhd	Pre-defined configurations for various targets
DEVICE	device.vhd	Current configuration
CONFIG	config.vhd	Generation of various constants for processor and caches
SPARCV8	sparcv8.vhd	SPARCV8 opcode definitions
IFACE	iface.vhd	Type declarations for module interface signals
MACRO	macro.vhd	Various utility functions
AMBA	amba.vhd	Type definitions for the AMBA buses
AMBACOMP	ambacomp.vhd	AMBA component declarations
MULTLIB	multlib.vhd	Multiplier modules

FPULIB	fpu.vhd	FPU interface package
DEBUG	debug.vhd	Debug package with SPARC disassembler
TECH_GENERIC	tech_generic.vhd	Generic regfile and pad models
TECH_ATC25	tech_atc25.vhd	Atmel ATC25 specific regfile, ram and pad generators
TECH_ATC35	tech_atc35.vhd	Atmel ATC35 specific regfile, ram and pad generators
TECH_MAP	tech_map.vhd	Maps mega-cells according to selected target

This hierarchy can be better visualized using the Figure 3.2. The files of our concern are target.vhd and device.vhd. "target.vhd" contains the various configurations for each of the components from which we can select the LEON configuration

A.2 Configuring LEON:

The following features of LEON are configurable

- Cache 2K – 64Kbytes
- Optional Coprocessor
- Optional Floating point Unit
- Ram and ROM sizes (upto 512MB ROM and 1GB RAM)
- 8 bit or 16 bit operation of bus

The memory configuration is used to configure the prom and I/O access. The following is the description of each field.

- [3:0]: Prom read wait states. Defines the number of wait states during prom read cycles (“0000”=0,“0001”=1,... “1111”=15).
- [7:4]: Prom write wait states. Defines the number of wait states during prom write cycles (“0000”=0, “0001”=1,... “1111”=15).
- [9:8]: Prom with. Defines the data with of the prom area (“00”=8, “01”=16, “10”=32).
- [10]: Reserved
- [11]: Prom write enable. If set, enables write cycles to the prom area.
- [17:12]: Reserved
- [18]: External address latch enable. If set, the address is sent out unlatched and must be latched by external address latches.
- [19]: I/O enable. If set, the access to the memory bus I/O area are enabled.
- [23:20]: I/O wait states. Defines the number of wait states during I/O accesses (“0000”=0,“0001”=1, “0010”=2,..., “1111”=15).
- [25]: Bus error (BEXCN) enable.
- [26]:Bus ready (BRDYN) enable.

- [28:27]: I/O bus width. Defines the data with of the I/O area (“00”=8, “01”=16, “10”=32).

A.3 Hex to Memory BFM Converter:

The C-program shown below is used to convert the output hex file obtained from MPLAB to a memory BFM compatible with the RISC8 core.

hex_pram.c:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  FILE *fptr1, *fptr2;
  char hex_file[20], line[44], pram_file[20];
  int x=0,index=0;

  printf("Give the file name to read:");
  scanf("%s",hex_file);

  // transfer the data to pram only if
  // file is opened without error
  if((fptr1=fopen(hex_file,"r"))!=NULL){
        printf("Give the output file name:");

        scanf("%s",pram_file);

        fptr2=fopen(pram_file,"w+");


      // initial statements in teh file

        fprintf(fptr2,"//

// Synchornous Data RAM, 12x2048

//

// Replace with your actual memory model..

//

module pram_00 (

   clk,

   address,

   we,

   din,

   dout

);


input           clk;

input [10:0]    address;

input           we;

input [11:0]    din;

output [11:0]   dout;
```

```
// synopsys translate_off

parameter word_depth = 2048;


reg [10:0]        address_latched;


// Instantiate the memory array itself.

reg [11:0]        mem[0:word_depth-1];

initial

  begin ");//end of first part




      // read from the file till

      // the EOFis encountered

      while( fscanf(fptr1,"%s",line)!=EOF){

          printf(" \n\t %s",line);

          x=9;


          // write the memory values to pram file

          while(line[x+2]!='\0'){


fprintf(fptr2,"mem[%d]=12'h%c%c%c;\n",index,line[x+3],line[x],line[x+1]);

              x=x+4;
```

```
            index++;
          }//ends while(line[....
        }//ends while(fscanf...

      //  the last instruction after sleep
      //  should be a nop
        fprintf(fptr2,"mem[%d]=12'h000;\n end",index-1);



       // write the second part
        fprintf(fptr2,"// Latch address\n
always @(posedge clk)\n
   address_latched <= address;\n
   \n
// READ\n
//assign dout = mem[address_latched];\n
assign dout = mem[address];\n
\n
// WRITE\n
always @(posedge clk)\n
   if (we) mem[address] <= din;\n
\n
// synopsys translate_on\n
\n
endmodule\n");
     fclose(fptr2);//closes pram_file
  }
  else{
     printf("unable to open the file:  %s",hex_file);
  }//ends if-else
  printf("\n");
  fclose(fptr1);
}
```

A.4. Router Array Generator

In order to have the flexibility in the size of the router array and to avoid rewriting the connection between the router, network interface and a processor core, we developed the following two files, ro_array.c and roa_pic.c. The executables of these files take the array size and name of the output file as the input.

ro_array.c: This file generates the array of routers along with network interfaces.

```
#include <stdio.h>



main()
{
 FILE * fptr;
 char ver_file[20];
 int size, i,j,pos,l,r,t,b;
 int modulus( int x, int y);
 printf("\n Give the size of the array\n (Ex: For 16 x 16 size is 16):");
 fflush(stdin);
 scanf("%d",&size);
 if(size<2 || (size%2 !=0))
 {
   printf("\n\n The size has to be an even number greater than or equal to 2\n");
   exit(1);
 }
 printf("\n Give output file name:");
 fflush(stdin);
```

```
  scanf("%s",ver_file);


  //create a file
   fptr = fopen(ver_file, "w+");
   if(fptr == NULL)
   {
     printf(" \n error in opening a file");
   }


  fprintf(fptr,"\n module top;
reg           Clk, Rst;
reg[3:0] HalfWidth;    //used to sel. the short route inside dim ring
reg[3:0] WidthM1;   //the right (top) end node number");


   for(i=0;i<size;i++)
   {
    for(j=0;j< size;j++)
    {
      pos = i  + j * size;


   // the defining of ports
      fprintf(fptr," \n
reg[7:0] NodeAddr%d;     //assume a 8 bits addr, 2+8 bits flit/phit",pos);
```

```
     fprintf(fptr," \nwire[9:0] D%dIn_Inj;",pos);
     fprintf(fptr," \nwire  d%dWt_Inj;",pos);
     fprintf(fptr," \nwire  d%dNF_Inj;",pos);
     fprintf(fptr," \nwire[9:0] D%dOut_YP, D%dOut_YN, D%dOut_XP, D%dOut_XN,
D%dOut_Ej;",pos,pos,pos,pos,pos);
     fprintf(fptr," \nwire[2:0] u%dWt_YP,  u%dWt_YN,  u%dWt_XP,
u%dWt_XN;",pos,pos,pos,pos);
     fprintf(fptr," \nwire u%dWt_Ej;", pos);
     fprintf(fptr," \nwire[2:0] u%dNF_YP,  u%dNF_YN,  u%dNF_XP,
u%dNF_XN;",pos,pos,pos,pos);
     fprintf(fptr," \nwire  u%dNF_Ej;", pos);
     fprintf(fptr," \nwire[1:0] u%dE_YP,   u%dE_YN,   u%dE_XP,
u%dE_XN;",pos,pos,pos,pos);
    }
   }



  for(i=0;i<size;i++)
  {
    for(j=0;j< size;j++)
    {
     pos = i + j * size;

     l = modulus((i-1), size)  + j * size;

     r = modulus((i+1), size)  + j * size;

     t = modulus((j+1), size) *  size + i;

     b = modulus((j-1), size) *  size + i;



  // instantiation of modules

     fprintf(fptr, "\n\n ni ni_%d(Clk,Rst, D%dIn_Inj,d%dWt_Inj,d%dNF_Inj,
D%dOut_Ej,u%dWt_Ej,u%dNF_Ej);",pos,pos,pos,pos,pos,pos,pos);

     fprintf(fptr,"\n Router r_%d(Clk,Rst,NodeAddr%d, HalfWidth, WidthM1,",pos,pos);

     fprintf(fptr," \n     D%dOut_YP, D%dOut_YN, D%dOut_XP, D%dOut_XN,
D%dIn_Inj,",b,t,l,r,pos);

     fprintf(fptr," \n     u%dWt_YP,  u%dWt_YN,  u%dWt_XP,  u%dWt_XN,
d%dWt_Inj,",b,t,l,r,pos);

     fprintf(fptr," \n     u%dNF_YP,  u%dNF_YN,  u%dNF_XP,  u%dNF_XN,
d%dNF_Inj,",b,t,l,r,pos);

     fprintf(fptr," \n     u%dE_YP,   u%dE_YN,   u%dE_XP,   u%dE_XN, ",b,t,l,r);

     fprintf(fptr," \n     D%dOut_YP, D%dOut_YN, D%dOut_XP, D%dOut_XN,
D%dOut_Ej,",pos,pos,pos,pos,pos);

     fprintf(fptr," \n     u%dWt_YP,  u%dWt_YN,  u%dWt_XP,  u%dWt_XN,
u%dWt_Ej,", pos,pos,pos,pos,pos);
```

```
     fprintf(fptr," \n     u%dNF_YP,  u%dNF_YN,  u%dNF_XP,  u%dNF_XN,
u%dNF_Ej,", pos,pos,pos,pos,pos);
     fprintf(fptr," \n     u%dE_YP,   u%dE_YN,   u%dE_XP,   u%dE_XN);",
pos,pos,pos,pos);
    }//end of loop for j
  }// end of loop for i
  fprintf(fptr,"\n\ninitial
 begin
  HalfWidth  <= 4'd%d;
  WidthM1    <= 4'd%d;
  Clk <= 1'b0;
  Rst <=1'b0;",size/2,size);


  fprintf(fptr,"\n end


always
 begin
  #5 Clk = ~Clk;
 end


initial
 begin
  #10 Rst <=1'b1;
  #10 Rst <=1'b0;


  #500
  $finish;
 end ");


  fprintf(fptr," \n\n endmodule\n");
  fclose(fptr);
} // end of main prog


int modulus(int i, int j)
{
  int x;
  if((i%j)<0)
  {
    x =  j  + (i % j);
  }
  else
  {
   x = i%j;
  }
  return x;
```

```
}
```

roa_pic.c: This file generates the top module for a multicomputer system with the array of routers connected to the network interface along with a processor. The input is the array size and output file name.

```
#include <stdio.h>

main ()
{
 FILE * fptr;
 char ver_file[20];
 int size, i,j,pos,l,r,t,b;
 int modulus( int x, int y);
 printf("\n Give the size of the array\n (Ex: For 16 x 16 size is 16):");
 fflush(stdin);
 scanf("%d",&size);
 if(size<2 || (size%2 !=0))
 {
   printf("\n\n The size has to be an even number greater than or equal to 2\n");
```

```
    exit(1);
  }
  printf("\n Give output file name:");
  fflush(stdin);
  scanf("%s",ver_file);

  //create a file
   fptr = fopen(ver_file, "w+");
   if(fptr == NULL)
   {
     printf(" \n error in opening a file");
   }

  fprintf(fptr,"\n module top;
reg          Clk, Rst;
reg[3:0] HalfWidth;    //used to sel. the short route inside dim ring
reg[3:0] WidthM1;   //the right (top) end node number");

   for(i=0;i<size;i++)
   {
    for(j=0;j< size;j++)
    {
      pos = i  + j * size;
   // the defining of ports
      fprintf(fptr," \n
reg[7:0] NodeAddr%d;     //assume a 8 bits addr, 2+8 bits flit/phit",pos);


      fprintf(fptr," \nwire[9:0] D%dIn_Inj;",pos);
      fprintf(fptr," \nwire  d%dWt_Inj;",pos);
      fprintf(fptr," \nwire  d%dNF_Inj;",pos);
      fprintf(fptr," \nwire[9:0] D%dOut_YP, D%dOut_YN, D%dOut_XP, D%dOut_XN,
D%dOut_Ej;",pos,pos,pos,pos,pos);
      fprintf(fptr," \nwire[2:0] u%dWt_YP,  u%dWt_YN,  u%dWt_XP,
u%dWt_XN;",pos,pos,pos,pos);
      fprintf(fptr," \nwire u%dWt_Ej;", pos);
      fprintf(fptr," \nwire[2:0] u%dNF_YP,  u%dNF_YN,  u%dNF_XP,
u%dNF_XN;",pos,pos,pos,pos);
      fprintf(fptr," \nwire  u%dNF_Ej;", pos);
      fprintf(fptr," \nwire[1:0] u%dE_YP,   u%dE_YN,   u%dE_XP,
u%dE_XN;",pos,pos,pos,pos);
      fprintf(fptr," \nwire
nf%d,ne%d,dt0%d,dt1%d,expread%d,expwrite%d;",pos,pos,pos,pos,pos,pos);
      fprintf(fptr," \nwire[7:0]expdin%d,expdout%d;",pos,pos);
      fprintf(fptr," \nwire[6:0]expaddr%d;",pos);
      fprintf(fptr," \nwire[10:0]paddr%d,debugpc%d;",pos,pos);
      fprintf(fptr," \nwire[11:0]pdata%d,debuginst%d;",pos,pos);
      fprintf(fptr," \nreg [11:0]din%d;",pos);
      fprintf(fptr," \nreg we%d;",pos);
      fprintf(fptr,"
\nwire[7:0]portain%d,portbout%d,portcout%d,debugw%d,debugstatus%d;",pos,pos,pos,p
os,pos,pos);
     }
   }

  for(i=0;i<size;i++)
  {
    for(j=0;j< size;j++)
    {
      pos = i + j * size;
      l = modulus((i-1), size)  + j * size;
      r = modulus((i+1), size)  + j * size;
      t = modulus((j+1), size) *  size + i;
      b = modulus((j-1), size) *  size + i;


  // instantiation of modules
```

```
    fprintf(fptr,"\n\n cpu
cp%d(Clk,Rst,paddr%d,pdata%d,portain%d,portbout%d,portcout%d,",pos,pos,pos,pos,p
os,pos);

fprintf(fptr,"expdin%d,expdout%d,expaddr%d,expread%d,expwrite%d,debugw%d,debug
pc%d,",pos,pos,pos,pos,pos,pos,pos);
    fprintf(fptr,"debuginst%d,debugstatus%d);",pos,pos);
    fprintf(fptr,"\n pram_00
p%d(Clk,paddr%d,we%d,din%d,pdata%d);",pos,pos,pos,pos,pos);
    fprintf(fptr, "\n ni
ni_%d(Clk,Rst,portain%d[0],portain%d[1],portain%d[2],portain%d[3],expdin%d,expdou
t%d,expaddr%d,expread%d,expwrite%d,",pos,pos,pos,pos,pos,pos,pos,pos,pos,pos);
    fprintf(fptr," D%dIn_Inj,d%dWt_Inj,d%dNF_Inj,
D%dOut_Ej,u%dWt_Ej,u%dNF_Ej);",pos,pos,pos,pos,pos,pos);
    fprintf(fptr,"\n Router r_%d(Clk,Rst,NodeAddr%d, HalfWidth, WidthM1,",pos,pos);
    fprintf(fptr," \n     D%dOut_YP, D%dOut_YN, D%dOut_XP, D%dOut_XN,
D%dIn_Inj,",b,t,l,r,pos);
    fprintf(fptr," \n     u%dWt_YP,  u%dWt_YN,  u%dWt_XP,  u%dWt_XN,
d%dWt_Inj,",b,t,l,r,pos);
    fprintf(fptr," \n     u%dNF_YP,  u%dNF_YN,  u%dNF_XP,  u%dNF_XN,
d%dNF_Inj,",b,t,l,r,pos);
    fprintf(fptr," \n     u%dE_YP,   u%dE_YN,   u%dE_XP,   u%dE_XN, ",b,t,l,r);
```

```
     fprintf(fptr," \n     D%dOut_YP, D%dOut_YN, D%dOut_XP, D%dOut_XN,
D%dOut_Ej,",pos,pos,pos,pos,pos);
     fprintf(fptr," \n     u%dWt_YP,  u%dWt_YN,  u%dWt_XP,  u%dWt_XN,
u%dWt_Ej,", pos,pos,pos,pos,pos);
     fprintf(fptr," \n     u%dNF_YP,  u%dNF_YN,  u%dNF_XP,  u%dNF_XN,
u%dNF_Ej,", pos,pos,pos,pos,pos);
     fprintf(fptr," \n     u%dE_YP,   u%dE_YN,   u%dE_XP,   u%dE_XN);",
pos,pos,pos,pos);
    }//end of loop for j
  }// end of loop for i
  fprintf(fptr,"\n\ninitial
 begin
  HalfWidth  <= 4'd%d;
  WidthM1    <= 4'd%d;
  Clk <= 1'b0;
  Rst <=1'b0;",size/2,size);
  fprintf(fptr,"\n end
always
 begin
  #5 Clk = ~Clk;
 end


initial
```

```
 begin
  #10 Rst <=1'b1;


  #10 Rst <=1'b0;


  #500
  $finish;
 end ");


  fprintf(fptr," \n\n endmodule\n");
  fclose(fptr);
} // end of main prog


int modulus(int i, int j)
{
  int x;
  if((i%j)<0)
  {
    x =  j  + (i % j);
  }
  else
  {
   x = i%j;
  }
  return x;
}
```