AN OCTA-CORE PROCESSOR WITH SHARED MEMORY AND MESSAGE-PASSING

Jinal K. Tapar[1], Shrushti K. Tapar[2], Ashish E. Bhande[3]

[1]*PG student, ECE Dept., HVPM's CoET, M.S, India*
[2]*PG student, ECE Dept., HVPM's CoET, M.S, India*
[3]*Asst. Professor, ECE Dept., HVPM's CoET, M.S, India*

Abstract

This being the era of fast, high performance computing, there is the need of having efficient optimizations in the processor architecture and at the same time in memory hierarchy too. Each and every day, the advancement of applications in communication and multimedia systems are compelling to increase number of cores in the main processor viz., dual-core, quad-core, octa-core and so on. But, for enhancing the overall performance of multi processor chip, there are stringent requirements to improve inter-core synchronization. Thus, a MPSoC with 8-cores supporting both message-passing and shared-memory inter-core communication mechanisms is implemented on Virtex 5 LX110T FPGA. Each core is based on MIPS III (Microprocessor without interlocked pipelined stages) ISA, handling only integer type instructions and having six-stage pipeline with data hazard detection unit and forwarding logic. The eight processing cores and one central shared memory core are inter connected using 3x3 2-D mesh topology based Network-on-chip (NoC) with virtual channel router. The router is four stage pipelined supporting DOR X-Y routing algorithm and with round robin arbitration technique. For verification and functionality test of above fully synthesized multi core processor, matrix multiplication operation is mapped onto the above said. Partitioning and scheduling of multiple multiplications and addition for each element of resultant matrix has been done accordingly among eight cores to get maximum throughput. All the codes for processor design are written in Verilog HDL.

Keywords: *MPSoC, message-passing, shared memory, MIPS, ISA, wormhole router, network-on-chip, SIMD, data level parallelism, 2-D Mesh, virtual channel*

----------***----------

1. INTRODUCTION

Multicore systems are dominating the processor market; they enable the increase in computing power of a single chip in proportion to the Moore's law-driven increase in number of transistors. A similar evolution is observed in the system-on-chip (SoC) market through the emergence of multi-processor SoC (MPSoC) designs. Nevertheless, MPSoCs introduce some challenges to the system architects concerning the efficient design of memory hierarchies and system interconnects while maintaining the low power and cost constraints. Finally, a qualitative analysis of the impact of instruction reuse, number of cores, and memory bandwidth on the system throughput in MPSoC systems is presented.

In recent years, the progress of computer science has been focused on the development of parallel computing systems (both hardware and software) and making those systems efficient, cost effective, and easy to program. Parallel computers have been around for many years. However, with the ability to now put multiple processing cores on a single chip the technologies of the past are in many cases obsolete. Technologies which have enabled parallel computing in the past must be re-designed with new constraints in mind. One important part of achieving these objectives has been to investigate various means of implementing mechanisms for communication between on-chip processing elements.

This ultimately has lead to the use of on-chip communication networks [9, 10]. There has been a significant amount of research devoted to on-chip networks, also known as Networks-on-Chip (NoC). Much of this has focused on the development of network topologies, routing algorithms, and router architectures that minimize power consumption and hardware cost while improving latency and throughput. However, no one solution seems to be universally acceptable. It follows that standardized methods to provide communication services on a multicore system-on-chip (SoC) also remain unclear. This can only slow the progress of the adoption of concrete parallel programming practices, which software developers are desperately in need of.

2. LITERATURE REVIEW

Implementation of multicore architecture on FPGAs has been the subject of several research projects. In [9] Del Valle et al present an FPGA-based emulation framework for multiprocessor system-on-chip (MPSoC) architectures. LEON3 [1], a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture, has been used in implementing multiprocessor systems on FPGAs. Andersson et al [2], for example, use the LEON4FT microprocessor to build their Next Generation Multipurpose Microprocessor (NGMP) architecture, which is prototyped on the Xilinx XC5VFX130T FPGA board. However, the

LEON architecture is fairly complex, and it is difficult to instantiate more than two or three on a medium size FPGA. Clack et al [4] investigate the use of FPGAs as a prototyping platform for developing multicore system applications. They use Xilinx MicroBlaze processor for the core, and a bus protocol for the inter-core communication. James-Roxby et al [5] shows similar FPGA design in their proposed architecture for supporting a single program multiple data model of parallel processing. A VHDL-based cycle accurate RTL model for evaluating power and performance of NoC architectures is presented in Banerjee et al [3].

Traditionally, processors in a parallel computer communicate through either shared memory or message passing [13]. In a shared memory system, processors have at least some shared address space with the other processors in the system. When multiple processors are working on the same task they often access the same data. To ensure correctness, accesses to the same data (or variable) must be mutually exclusive. Therefore, when one processor is accessing a shared variable, the others must wait for their turn. This typically works fine when there are only a few processors. However, when there are many, the time required to obtain access can become a serious performance limitation. Perhaps the biggest issue with performance in parallel systems is with data locality [12, 13]. Anytime a program accesses data there is a cost associated with that access. The further the physical location of the data is from the processor, the larger the cost is for accessing it. Caches and memory hierarchies help to combat this problem. However, as systems scale more aggressively the hardware that controls these hierarchies becomes more complex, power hungry, and expensive. Therefore, the programmer must have some control over locality. Through the tool chain, instruction set, or operating system the programmer should be able to specify the location of data structures and the location of the processor that will run the code that uses these data structures.

Each of these challenges is addressed in this work. Limited parallelism is addressed by designing a highly efficient communication system, thus making a program with frequent synchronization points more capable of performing well. Data locality is also dealt with. Lightweight message passing provides a programmer with the ability to copy data to where it needs to be for efficient use. To avoid inefficient use of memory, certain modifications provide the ability to statically allocate memory to only the core that uses it. Likewise, ISA (instruction set architecture) modifications provide the programmer with the ability to specify which core runs which code.

3. SYSTEM ARCHITECTURE

The 8-core processor proposed [14] has 3x3 2D Mesh NoC that links eight processor cores (PCore) based on MIPS III instruction set architecture [16] and a memory cores (MCore). A hybrid inter-core communication scheme is employed supporting both shared-memory and message-passing communications. Shared memory in MCore enables shared-memory communications within the cluster, and the NoC enables message-passing among all PCores. The cluster comprises eight Pcores and one MCore, shared memory in MCore can be accessed by PCores in the same cluster. Data enters the processor through the input First in First out (FIFO), and exits through the output FIFO. An on-chip oscillator generates the system clock, necessary for circuit functioning. Fig. 1 depicts the architecture overview and key features of the proposed processor [15].

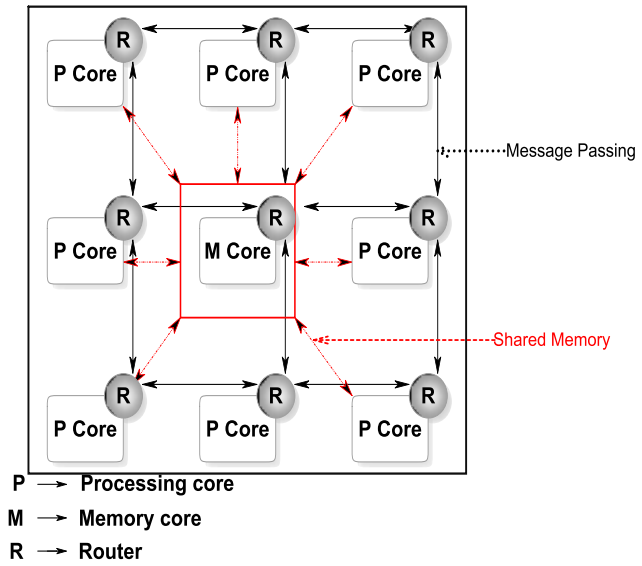


Fig -1: Architecture Overview of project

The PCore includes a typical Reduced Instruction Set Computer (RISC) style processor core with six-stage pipeline, a 256-word instruction memory, a 256-word private data memory, a router and interfaces for inter-core communications. The MCore includes an 1k-word shared memory with four banks. Detailed design and implementation of 32 bit MIPS processor of Pcore is discussed in the following section.

3.1 Processor Overview

The processor core uses a 6-stage in-order pipeline [15] as shown in Fig. 3.3. Initially, a program counter is incremented in stage 1 to provide an address for the instruction fetch in stage 2. It is reasonable to assume that all instruction fetches will complete in one clock cycle. Stage 3 decodes the 32-bit instruction by extracting it's various fields and setting the appropriate control signals based on those fields. The register file is also accessed in this stage. Note that a register can be read in the same clock cycle that it is written to without conflict. Next, the functional units perform the computation for the bulk of the instruction set. The router at each node handles all communication oriented instructions that have been added to the MIPS ISA in this design. Accesses to data memory are handled in stage 5. Operations in this stage may stall the pipeline. Finally, results of memory or functional unit operations are written to their destination registers in stage 6. Much of the basic design follows from the MIPS[16] processor architecture presented in [7]. This includes the logic used for data forwarding, hazard detection, and

pipeline organization. Though it has been heavily modified in this implementation, the design in [7] is a fundamental starting point for any student of this work.

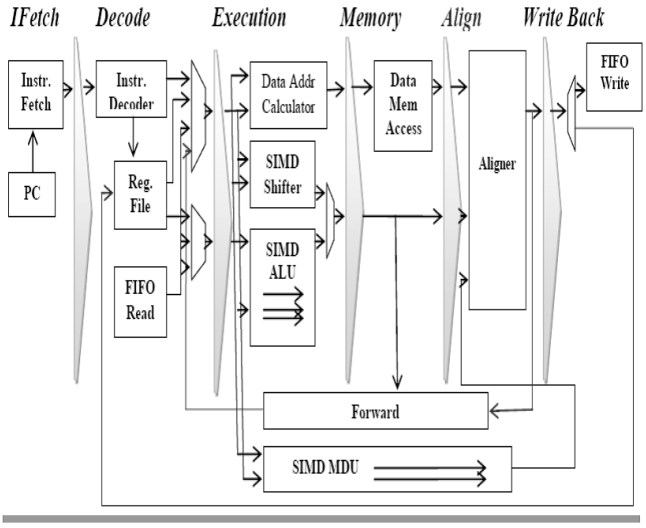


Fig -2: Six-stage pipeline of implemented processor

4. OVERVIEW OF DESIGNED ROUTER

The 2D-NoC system is based upon *Mesh* topology, where *x-addr* and y-*addr* are attributed to each router and define its X and Y coordinates respectively and its position along the network. Many topologies exist for the implementation of NoCs, some are regular (*Torus, tree-based*) and other irregular topologies are customized for some special application. We choose the Mesh topology for this design thanks to its several properties like regularity, concurrent data transmission, and controlled electrical parameters [9]. Figure.3 shows a configuration example of the router designed for implementing 3x3 2D- NoC design.

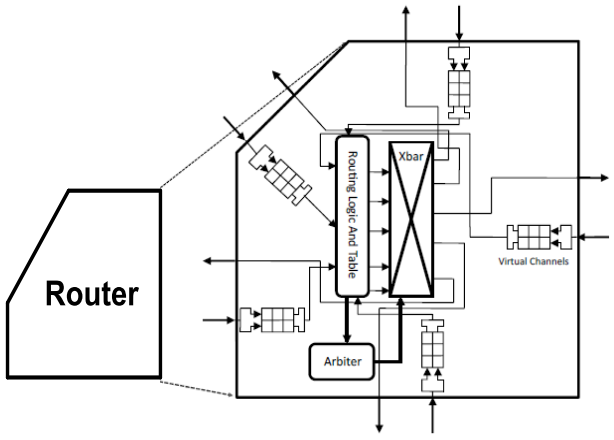


Fig -3: Router Architecture

We can see in this figure that router is having five ports: Out of which four are for four directions' routing and one is local to particular processing element node connected through network interface. Each port is equipped with buffering modules to handle multiple traffic loads.

4.1 Routing Algorithm

The designed system router is based upon the Dimension Order Routing (DOR) XY algorithm [17]. XY routes flits first along the X dimension and then along the Y and to reach its destination. This process is done by comparing the address of the processing node with the destination node's address to determine the *Output-Port*: The routing algorithm implemented in the designed router is summarized in the table -1.

Table -1: Routing Algorithm

X / Y	**xdest > xaddr**	**xdest = xaddr**	**xdest < xaddr**
ydest > yaddr	East	North	West
ydest = yaddr	East	Local	West
ydest < yaddr	East	South	West

- If *xdest* is larger than *xaddr* then *Output-Port* will be EAST. In the opposite case *Output-Port* will be WEST.
- If *ydest* is larger than *yaddr* then *Output-Port* will be NORTH, else *Output-Port* will be SOUTH.
- If *xdest* is equal to *xaddr*, *ydest* is equal to *yaddr* then *Output-Port* will be SELF

4.2 Pipeline

The pipeline for this designed router comprises of the general four stages. Taking a closer look at Fig. 4, we can see that conventional XY-based router pipeline design contains 4 main pipeline stages: Buffer Writing (BW) where the incoming flit is stored in the input buffer, then in Routing Calculation stage (RC) destination address is fetched and decoded to determine the *Output-Port* direction. Information about the selected *Output-Port* is sent to the next stage, Switch Arbitration (SA), to resolve any competition between different requests from different input-ports. Finally the Crossbar traversal stage (CT) handles the transfer of the flit to the next neighboring node. This 4 pipelines router design increases the flit latency and its associated power consumption, since any flit should go through all these stages at each hop while traveling from source to destination.

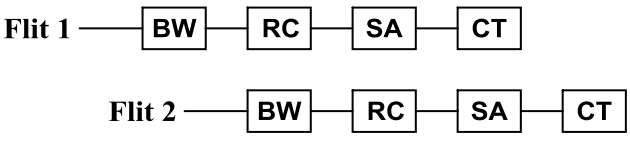


Fig -4: Router pipeline stages

4.3 Switch Arbitration

When two or more packets need to use the same link they must go through an arbitration process to determine which packet should get exclusive access to the link. The arbiter must be fair, but must be capable of granting access very quickly. The arbitration process used in both network types is governed by the finite state machine (FSM) [9, 23] given in Fig. 5. For simplicity, only 3 input channels (north, south, and east) are used and state transitions to and from the initial state have been omitted. This arbitration FSM is instantiated for each output port.

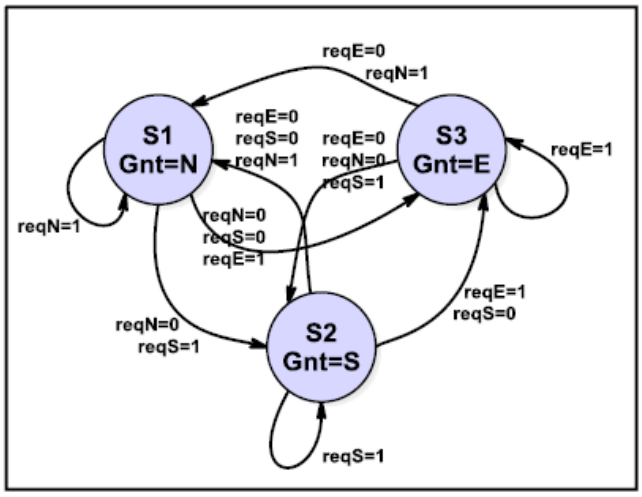


Fig -5: Arbitration FSM

The directions (abbreviated by N, S, and E) in the diagram represent requests and grants to each input channel for use of the output channel associated with the FSM instantiation. The request signals are generated by the routing function and the direction that a packet needs to go. This signal only returns to zero when a tail flit is seen. Notice that each state gives priority to the channel that currently has the grant token. If that request returns to zero (from a tail flit), priority goes to the next channel in a round robin sequence. If there are no pending requests then the FSM returns to the initial state.

5. EVALUATION METHODOLOGY

Our system was designed in Verilog HDL, synthesized and prototyped on commercial CAD tools and FPGA respectively [8]. We evaluate the hardware complexity of this octa-core MPSoC, when implemented Virtex 5 LX110T FPGA, in terms of area utilization, power consumption and clock frequency.

When implementing an application in parallel, the application must be broken down so that separate portions may be executed at the same time. This may be done via task parallelism in which the separate tasks undertaken by the application are executed concurrently, or it may be done via data parallelism in which the data to be operated on is broken into separate sections and each section of data is operated on at the same time. The data parallelism style of programming is "the most common strategy for scientific programs on parallel machines"[13]. To evaluate the performance of this octa-core processor, we selected a well known benchmark: Matrix multiplication, which is a well known application frequently used by many researchers. We chose the Matrix-multiplication since it is widely used in scientific applications. This choice is also made thanks to its potential to achieve its best performance in a parallel architecture [13].

5.1 Matrix-Multiplication

$$\begin{pmatrix} A11 & \cdots & Ak \\ \vdots & \ddots & \vdots \\ Ai1 & \cdots & Aik \end{pmatrix} \times \begin{pmatrix} B11 & \cdots & B1j \\ \vdots & \ddots & \vdots \\ Bk1 & \cdots & Bkj \end{pmatrix} = \begin{pmatrix} R11 & \cdots & R1j \\ \vdots & \ddots & \vdots \\ Ri1 & \cdots & Rij \end{pmatrix}$$

Fig -6: Matrix multiplication example

First we assume that an *ixk* matrix *A* has *i* rows and *k* columns, where A_{ik} is an element of *A* at the *i*-th row and *k*-th column. As it demonstrated in Fig.6, an *ixk* matrix *A* can be multiplied by a *kxj* matrix *B* to obtain an *ixj* matrix *R*. Figure. 6 present how the matrix *R* can be obtained according to following formula.

$$R_{i,j} = \sum_{n=0}^{k-1} A_{i,n} \cdot B_{n,k}$$

When implemented onto octa-core processor, and for sake of convenience or without loss in generality, we can assume that all the matrices are square and having *nxn* size. In this project, each element of the two matrices is assigned to a processor core module which is connected to one router. As a result the number of routers connected to the network is

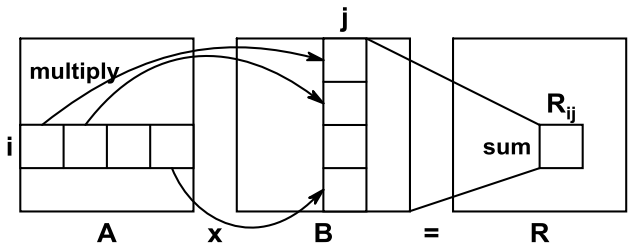


Fig -7: Simple example demonstrating the Matrix multiplication calculation.

the sum of all the elements of three matrices which is equal to $2n^2$. Each element of the matrix *B* receives *n* flits from *n* different elements of the matrix *A* in order to make the multiplication. Then, each element of the matrix *B* sends *n* flits to *n* different elements of the matrix *R* where all the received values are summed then the final resulted value is outputted.

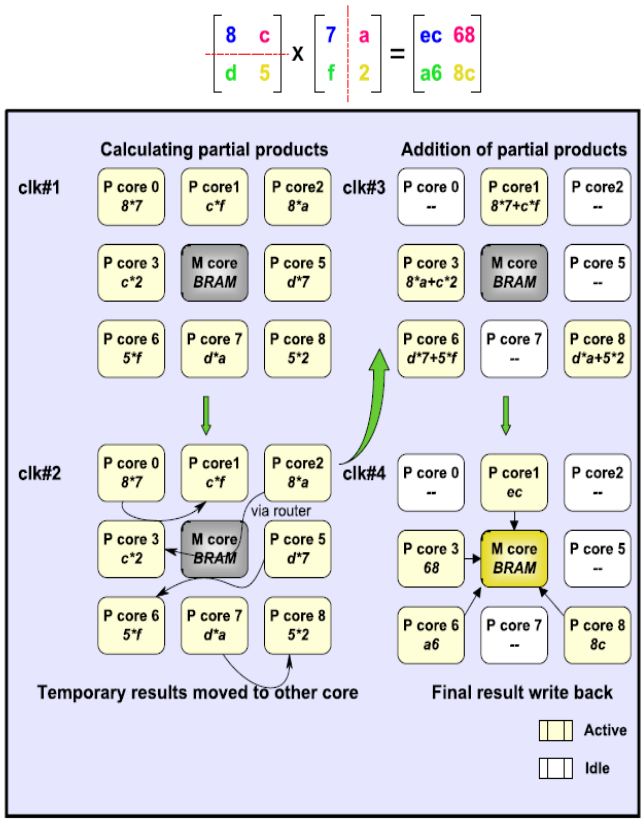


Fig -8: Working summary of Mapped Program

At present, this procedure is purely manual. Figure 8 manifests the overall idea of tasks distribution done among eight cores. The clocking numbers shown are just for explanation and not the exact ones from simulation timings. The figure 8 is just explaining the program flow steps in brief. Firstly, program addresses are loaded into each cores' PC, then data is loaded and finally step by step execution goes on.

5.2 Synthesis Results

After the HDL synthesis phase of the synthesis process, the RTL Viewer is used to view a schematic representation of the pre-optimized design in terms of generic symbols that are independent of the targeted Xilinx device, for example, in terms of adders, multipliers, counters, AND gates, and OR gates.

Figure 9 is the top block automatically generated by RTL Viewer when *real_cores_mesh.ngr* file [18] is opened and run. Further opening means elaborating this top block gives all the basic components utilized and their inter-connections in detail to realize the proposed designed octa-core system can be visualized and analyzed from figure 9.

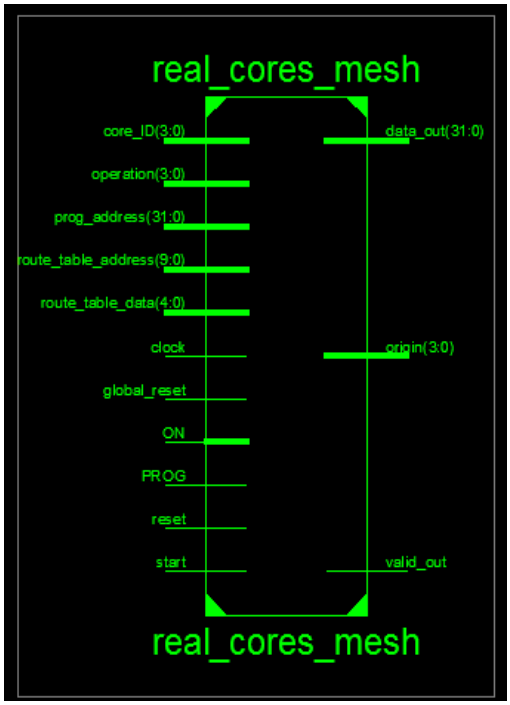
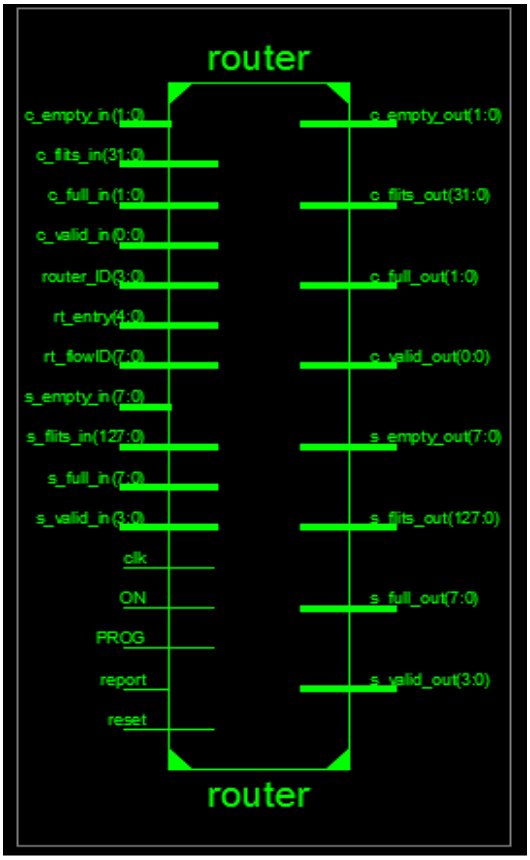
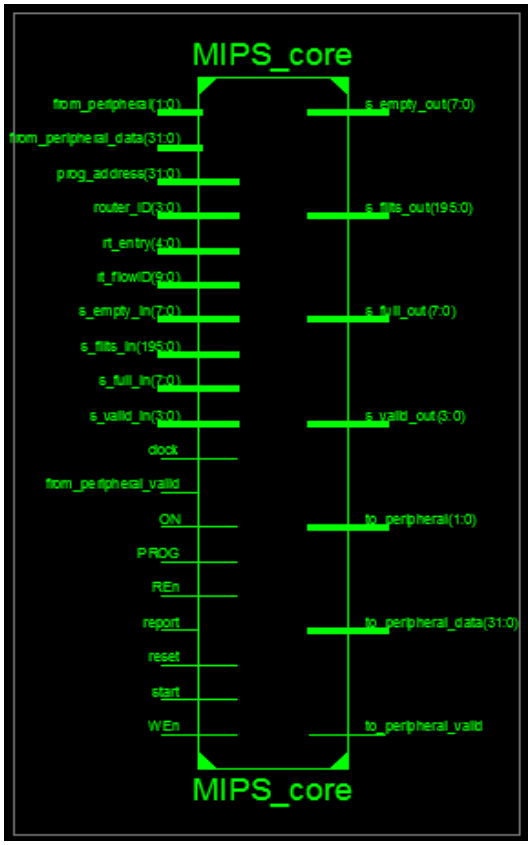


Fig -9: RTL Schematic of Top block *real_cores_mesh*



(a) router



(b) MIPS_core

Fig -10: RTL schematic of other blocks

Figure 10 shows the RTL schematic of other vital sub-blocks of the Octa-core project top module *real_cores_mesh.* Each of the individual MIPS 32-bit processor module's *MIPS_core .ngr* file gives details of RTL logic as shown in figure 10 (b). The router module designed for interconnecting each of cores is synthesized using file *router.ngr.* Its system generated RTL schematic can be seen in figure 10 (a). Technology Viewer can be used to view a schematic representation of the design in terms of logic elements optimized to the target Xilinx device or "technology," for example, in terms of LUTs, carry logic, I/O buffers, and other technology-specific components.

5.3 Device Utilization

The key architectural characteristics of the octa-core were summarized in earlier section 5.2. The actual resources utilization of the selected target device by the designed modules is now presented here. They are being synthesized by Xilinx's ISE design Suite 14.5 EDA tool [18] and create a web file of *Design Summary* in the related project file.

Table 2 summarizes the FPGA resource utilization by the different systems in terms of registers; lookup tables, and block RAMs. The *real_cores_mesh* module uses 99% of block RAM resources at 256 words of local memory per core.

Table -2: Device utilization summary of *real_cores_mesh*

TARGET DEVICE:	Xc5vls110t-1ff1136		
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	51397	69120	74%
Number of Slice LUTs	60916	69120	88%
Number of fully used LUT-FF pairs	25088	87225	28%
Number of bonded IOBs	499	640	78%
Number of Block RAM/FIFO	148	148	100%
Number of BUFG/BUFGCTRL	17	32	54%
Clock Rate (MHz) : 100			
Critical Path (ns) : 14.253			

Table -3: Device utilization summary of *MIPS_core*

TARGET DEVICE:	Xc5vls110t-1ff1136		
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	11278	69120	17%
Number of Slice LUTs	15792	69120	23%
Number of fully used LUT-FF pairs	9861	87225	11%
Number of bonded IOBs	499	640	78%
Number of Block RAM/FIFO	34	148	23%
Number of BUFG/BUFGCTRL	17	32	54%
Clock Rate (MHz) : 100			
Critical Path (ns) : 4.053			

Table -4: Device utilization summary of *router*

TARGET DEVICE:	Xc5vls110t-1ff1136		
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	2640	69120	4%

Number of Slice LUTs	11622	69120	17%
Number of fully used LUT-FF pairs	2199	87225	3%
Number of bonded IOBs	37	640	6%
Number of BUFG/BUFGCTRL	1	32	0%
Clock Rate (MHz) : 33.33			
Critical Path (ns) : 23.697			

The *real_cores_mesh* system is running at 100 MHz, which is the clock frequency of the top module, regardless of the size of the mesh. The *router* module runs slower at 33 MHz speed and introduces some latency. This causes delay in the outcome of desired result. The worst case timing is said to have critical path. Its timing is also quoted in the tables.

5.4 Simulation Results

Here, we have taken an example showing the matrix multiplication of two 2x2 matrices. The program for this was loaded into the *memory.mem* [21] file of instruction memory. The data for two input 2x2 matrices is stored in array in the shared memory from where each P core accesses it as per the described partition of data in the code and runs their instruction following strict scheduling constraints given by programmer. Manually, calculated results and those obtained from the simulation were verified and found to be correct. All the data is in Hexadecimal system.

The simulation waveform generated for the written and loaded test code is shown in figure 11. The simulation for data memory block is summarized here. The console data is printed by test bench for easy verification of results. It shows the results of program run in previous simulation waveform, being written back to memory addresses specified by program code. A snapshot of data in the console window of ISim is shown in figure 12.

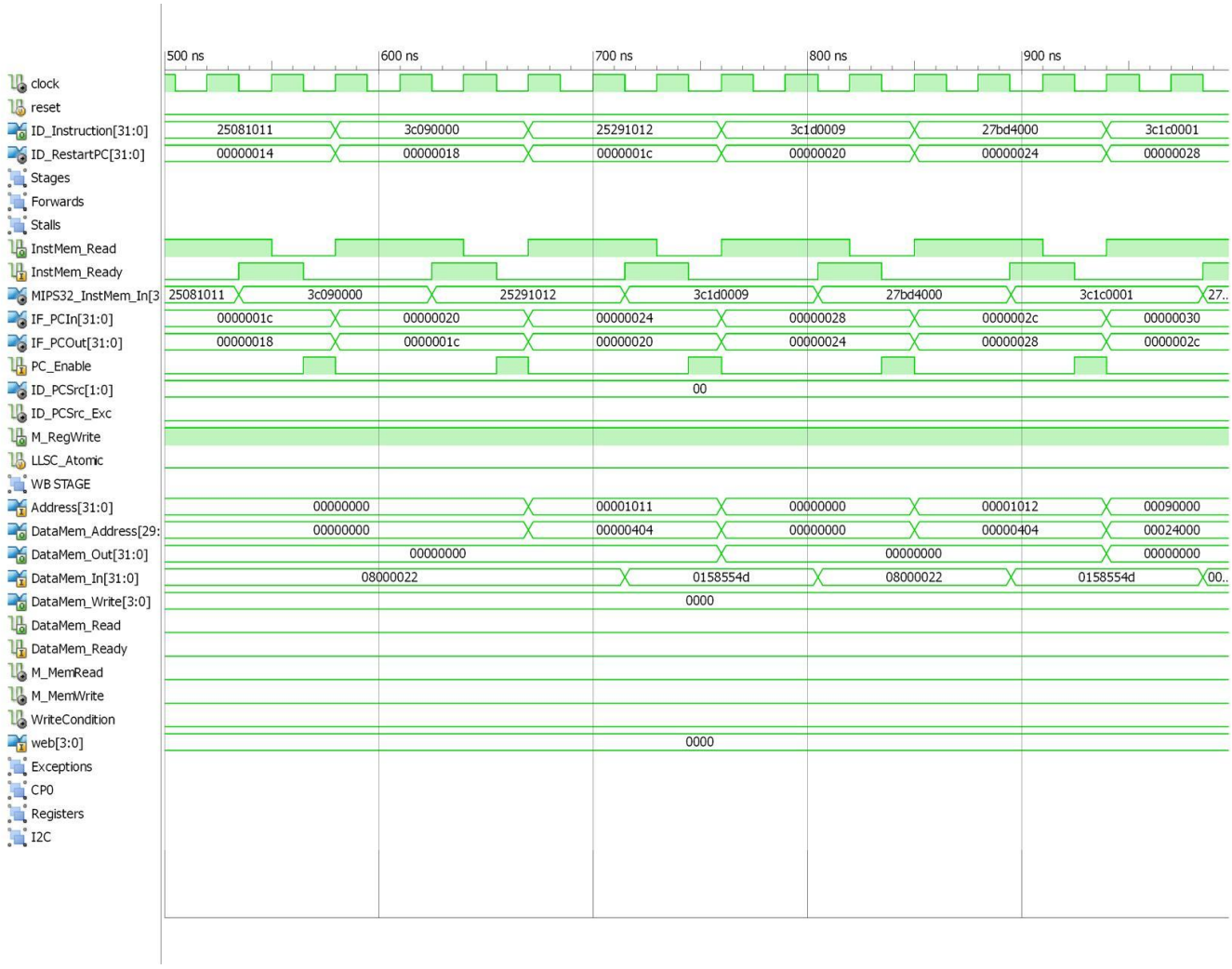


Fig -11: Simulation Waveform for each processor and the memory contents

```
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [32]                           Write Address [00]
Read Data [000000ec]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [32]                           Write Address [00]
Read Data [000000ec]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [33]                           Write Address [00]
Read Data [000000ec]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [33]                           Write Address [00]
Read Data [00000068]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [33]                           Write Address [00]
Read Data [00000068]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [34]                           Write Address [00]
Read Data [00000068]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [34]                           Write Address [00]
Read Data [000000a6]
Write Data [00000000]
```

```
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [34]                           Write Address [00]
Read Data [000000a6]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [35]                           Write Address [00]
Read Data [000000a6]
Write Data [00000000]
----------------------------------------------------------------------------
------------------------------BRAM----------------------------------------
Read [1]                                    Write [0]
Read Address [35]                           Write Address [00]
Read Data [0000008c]
Write Data [00000000]
----------------------------------------------------------------------------
ISim> |
```

Fig -12: Snapshot of console window of ISim showing results of program run

A brief description of simulation is explained here: During first clock cycle after reset and as soon as PROG pin is high, it allows the address of program instructions to be loaded into the PC (program counter) of each core correspondingly at the same time. Due to this a small multiplication program is loaded into instruction memory of each core. As per partitioned data, each core accesses its data elements and starts performing multiplication of each row and corresponding column element. Some of the clock cycles are invested in these computations. Then respective cores request for the partial result data which it needs to add with its result for resultant matrix element. Thus, we get the fast execution of matrix multiplication with less latency by exploiting the parallelism of data manipulation.

5.5 Implementation Results

A large number of cores can be implemented on a modern FPGA. Moreover, having a simple RISC core, MIPS in our case, for the processing element (PE) allows for a good size multicore system. The board being used to verify the multi core processor was a Xilinx University Program (XUP) board that contains Xilinx Virtex 5 XUP VLX110T FPGA, package 1ff1136 [20]. The processor was instantiated into an FPGA module to connect internal signals of the processor to the I/O of the board (push buttons, dip switch, and LEDs) shown in figure 13.

Fig -13: Hardware Implementation of designed Octa- core processor on Virtex 5 FPGA

The FPGA module supports two modes of operation: the onboard 33 MHz [18] clock or a step clock triggered by pressing a push button. The clock mode is set by dip switch 1. An onboard 100 MHz clock is used here. Other available clocking options could be implemented by modifying the user constraint file (UCF) [21]. *iMPact* tool targets the FPGA device by initializing the chain and then programs the selected device by loading the generated *.bit* file through JTAG programming cable.

6. CONCLUSION

We have presented a complete, realistic, fully parameterized, synthesizable, modular, multicore architecture. The system, an octa-core processor, uses a component-based design approach, where the processing element or core, the router and the network-on-chip, and the memory subsystem are independent building blocks, and can be used in other designs. The baseline system has a 6-stage integer-based MIPS core, a virtual-channel wormhole router, with support for both shared memory and message-passing inter core communication techniques. It has been successfully implemented on the Xilinx Virtex-5 LX110T FPGA board. We have introduced a small matrix multiplication program, now presently being written and dumped in the system memory manually. Also the workload distribution amongst multiple cores according to application being run is still a manual process.

6.1 Opportunities

- Good attempt being made to combine the best of both inter-core communication technique viz. Shared memory and Message passing.
- Efficient memory distribution by separating instruction and data memory amongst each core. Thus, memory accesses provide low latency.
- Low power consumption of the system as it is running on slower clock speed viz.100MHz. But, at the same time performance is not sacrificed because of multiple cores.

6.2 Limitations

- Mostly known, parallel programming is quite different and complex than traditional sequential style of programming. So stringent, excellent and careful programming skills are required.
- At present, due to absence of software tool chain program loading into instruction memory and the workload distribution among cores is manual. Thus, it is quite tedious job and requires cautious efforts to avoid any discrepancies.
- During the specific parallel program execution, some of the cores remain idle. Thus, some dynamic task distribution technique need to be revised for efficient exploitation of work capabilities of multiple core system.
- Instruction set implemented for the project has excluded the floating point instruction and some other advanced version of load and store instruction. So, for real time applications instruction set need to be extended

7. FUTURE WORK

As mentioned above, the processing element designed above was simple supporting only basic instructions dealing with integer type only. For future, optimizing this multicore processor by involving all the UART, interrupt handler and real time related peripherals for some specific application is the goal. Designing software toolchain in Linux environment for mapping multi-threaded parallel running programs on multicore platform and exploring appropriate technique for managing dynamic runtime workload among the cores are some of the future goals.

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to my guide Prof. A E Bhande who have constantly encouraged me and showed right directions to accomplish and realize this project concept to reality. Also I am grateful to our HoD Dr. U A. Belorkar and the whole staff of EXTC Dept. of HVPM's CoET for providing the essential resources and environment.

REFERENCES

[1]. A. G. AB. Leon3 processor. Available at: http://www.gaisler.com.

[2]. J. Andersson, J. Gaisler, and R. Weigand. Next generation multipurpose microprocessor. Available at: http://microelectronics.esa.int/ngmp/NGMP-DASIA10-Paper.pdf, 2010.

[3]. N. Banerjee, P. Vellanki, and K. Chatha. A power and performance model for network-on-chip architectures. volume 2, pages 1250 – 1255 Vol.2, feb. 2004.

[4]. C. R. Clack, R. Nathuji, and H.-H. S. Lee. Using an fpga as a prototyping platform for multi-core processor applications. In WARFP-2005: Workshop on Architecture Research using FPGA Platforms, Cambridge, MA, USA, feb. 2005.
[5]. P. James-Roxby, P. Schumacher, and C. Ross. A single program multiple data parallel processing platform for fpgas. In FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pages 302–303, Washington, DC, USA, 2004.
[6]. Xilinx UG347 ML505/ML506/ML507 evaluation platform, User guide
[7]. David A. Patterson and John L. Hennessy. Computer Organization and Design: The Hardware/Software Interface. Elsevier Inc., 3rd edition, 2005
[8]. http://www.xilinx.com
[9]. M. Ali, M. Welzl, and M. Zwicknagl. Networks on chips: Scalable interconnects for future systems on chips. ECCSC, 2008.
[10]. W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. DAC, 2001.
[11]. Xilinx UG348 ML505/ML506/ML507 Getting started tutorial with Evaluation platform, User guide
[12]. Zhiyi Yu, Member, IEEE, Ruijin Xiao, Student Member, IEEE, Kaidi You, Heng Quan, Peng Ou, Zheng Yu, Maofei He, Jiajie Zhang, Yan Ying, Haofan Yang, Jun Han, Xu Cheng, Zhang Zhang, Ming'e Jing, and Xiaoyang Zeng, Member, IEEE "A 16-Core Processor With Shared-Memory and Message-Passing Communications", IEEE transactions on circuits and systems—i: regular papers, vol. 61, no. 4, april 2014
[13]. C. Lin and L. Snyder. Principles of Parallel Programming. AddisonWesley, 2008.
[14]. Jinal Tapar., PG student, Prof. A E. Bhande, Asst. Prof, "A Multi-Core Processor with Efficient Memory", IJAFRC, volume 2, issue 2,February 2015,ISSN 2348 4853
[15]. Jinal Tapar, Shrushti Tapar, Prof. A E. Bhande, Design of 16-bit MIPS Processor for Multi-Core SoC. IETE 46th Mid Term Symposium 11th & 12th April, 2015.
[16]. MIPS ® 32 Architecture Volume II: The MIPS ® 32 The Instruction set
[17]. R. D. Mullins, A. F. West, and S. W. Moore. Low-latency virtual-channel routers for on-chip networks. In Proc. of the 31st Annual Intl. Symp. On Computer Architecture (ISCA), pages 188–197, 2004
[18]. Xilinx UG347 ML505/ML506/ML507 evaluation platform, User guide
[19]. http://www.xilinx.com
[20]. Xilinx UG348 ML505/ML506/ML507 Getting started tutorial with Evaluation platform, User guide
[21]. Data2MEM User Guide UG658 (v 11.2) June 24, 2009 www.xilinx.com
[22]. http://www.wikipedia.com
[23]. Michal D. Cilleti, ―Advanced Digital Design with the Verilog HDL‖, Prentice hall of India Pvt. Limited, New Delhi. ISBN-81-203-2756-X