Volume 2, No. 9, September 2011
Journal of Global Research in Computer Science
ISSN-2229-371X
RESEARCH PAPER
Available Online at www.jgrcs.info

VLIW BASED VEX TOOL AND VALIDATION OF SIM-A WITH VEX

Dr. Manoj Kumar Jain[1] and Gajendra Kumar Ranka*[2]
*Research Scholar, Associate Professor
Department of Computer Science MLSU University, Udaipur
manoj@cse.iitd.ernet.in[1]
*rankagajendra@rediffmail.com[2]

Abstract:- There is a growing demand for application-specific embedded processors in system-on-a-chip designs. Current tools and design methodologies often require designers to manually specialize the processor based on an application. An application-specific instruction-set processor (ASIP) is a component used in system-on-a-chip design. The instruction set of an ASIP is tailored to benefit a specific application. This specialization of the core provides a trade-off between the flexibility of a general purpose CPU and the performance of an ASIC. The major contribution of this paper lies in verifying or substantiating SIM-A with VLIW based tool - Vex. Simulator SIM-A measures cycle count for application executed on processor. This paper discusses working with vex and its configuration required to execute the benchmark application on Vex.

Keywords: ASIP, Application Specific Instruction Processors, Retargetable Simulator, Embedded Systems, Processors, ASIP Simulators, Design Space Exploration, Vex Simulator, SIM-A

INTRODUCTION

In Consumer electronics and telecommunications, high product volumes are increasingly going along with short lifetimes. Driven by the advances in semiconductor technology combined with the need for new applications like digital television and wireless broadband communications, the amount of system functionality realized on a single chip is growing enormously. Higher integration and, thus, increasing miniaturization have led to a shift from using distributed hardware components toward heterogeneous system-on-chip (SOC) designs. Due to the complexity introduced by such SOC designs and time-to-market constraints, the designer's productivity has be- come the vital factor for successful products. For this reason a growing amount of system functions and signal processing algorithms is implemented in software rather than in hardware by employing embedded processor cores.

In the current technical environment, embedded processors and the necessary development tools are designed manually, with very little automation. This is because the design and implementation of an embedded processor, such as a digital-signal-processor (DSP) device embedded in a cellular phone, is a highly complex process. Any embedded system may compose of the following phases:

a. Architecture exploration;
b. Architecture implementation;
c. Application software design;
d. System integration and verification.

During the architecture exploration phase, software development tools [i.e., high-level language (HLL) compiler, assembler, linker, and cycle-accurate simulator] are required to profile and benchmark the target application on different architectural alternatives. This process is usually an iterative one that is repeated until a best fit between selected architecture and target application is obtained. Every change to the architecture specification requires a complete new set of software development tools. As these changes on the tools are carried out mainly manually, this results in a long, tedious, and extremely error-prone process. Furthermore, the lack of automation makes it very difficult to match the profiling tools to an abstract specification of the target architecture. In the architecture implementation phase, the specified processor has to be converted into a synthesizable hardware description language (HDL) model. With this additional manual transformation, it is quite obvious that considerable consistency problems arise between the architecture specification, the software development tools, and the hardware implementation.

During the software application design phase, software designers need a set of production-quality software development tools. Since the demands of the software application designer and the hardware processor designer place different requirements on software development tools, new tools are required. For example, the processor designer needs a cycle/phase-accurate simulator for hardware–software partitioning and profiling, which is very accurate, but inevitably slow, whereas the application designer demands more simulation speed than accuracy. At this point, the complete software development tool suite is usually reimplemented by hand—consistency problems are self-evident. In the system integration and verification phase, co simulation interfaces must be developed to integrate the software simulator for the chosen architecture into a system simulation environment. These interfaces vary with the architecture that is currently under test. Again, manual modification of the interfaces is required with each change of the architecture.

An ASIP is a processor that is designed to efficiently execute the software for a specific application. Regardless of whether a newly designed ASIP or a pre-existing processor core is used, the selected processor should be well suited for the given application. Although incorporating a complete system on a single IC may improve performance, cost, and

power consumption requirements, such a high level of integration constraints the size of the system components.

Steps in ASIP Synthesis:

Various methodologies have been reported to meet these requirements. All these have been studied and five steps are suggested for synthesis of ASIPs [1]

Application Analysis: Application is normally written in High level language. Proper analysis of this application under consideration is done and the output of the information is stored in some suitable intermediate format.

Architectural Design Space Exploration: Output of the Application analysis step along with the range of architecture for Design Space Exploration is used to select a suitable architecture.

Instruction Set Generation: Based on this input instruction sets are generated in terms of required micro operation. This instruction set is used during the further steps for code synthesis and hardware synthesis.

Code Synthesis: Till this step, architecture template, instruction set, and application are identified. This step generates the code. Generated code can be retargetable code generator or compiler generator.

Hardware Synthesis: In this step the hardware is generated using the ASIP architectural template and instruction set architecture using standard tools

Architecture Design Space Exploration:

System on Chip designs has various goals and objectives. Design space consists of a set of parameters. The main focus of designers lies on minimal cost and maximal performance, low power, high reliability etc. Architecture under consideration requires a range of good parameter to explore. These parameters may take up the different values.

Existing Retargetable Simulators Approaches:

Retargetable functional simulator (Fsimg) [2] focus on tools that deal with the machine language of processors, like assemblers, disassembler, instruction set simulator etc.Retargetable Function Simulator (Fsimg) was designed using Sim-nML language which is primarily an extension of the nML [3] language for processor modeling. Fsimg takes the specification of the processor in the intermediate representation [4] and an executable for the processor in ELF [5] format and generates a functional simulator (Fsim) which in turn gives the functional behaviour of the processor model for the given program.

RELATED WORK

Over the past several decades a considerable amount of research has been performed in the area of computer architecture simulation. These simulators can be broadly divided into several categories: full-system simulators, Instruction Set Architecture (ISA), and retargetable Simulators. Each category serves an entirely different purpose, but all have been used for the advancement of computer architecture research.

The purpose of full-system simulators is to model an entire computer system including the processor, memory system and any I/O. These simulators are capable of running real software completely unmodified just like a virtual machine. There are many simulation suites that take this approach, including PTLSim [6], M5 [7], Bochs [8], ASIM [9], GxEmul [10] and Simics [11]. Simics has several extensions that constitute their own full-system simulators such as VASA [12] and GEMS [13].ISA simulators are less descriptive than full system simulators. Their objective is to model processor alone.ISA simulators performs the various functionalities.

It simulate and debug machine instructions of a processor type that differs from the simulation host, it also emphasis on investigating how the various instructions (or a series of instruction) affect the simulated processor. Hence modeling of the full computer system is unnecessary and would impose additional delay and complexity. Example of this type of simulator includes SimpleScalar [14], WWT-II [15], and RSIM [16]. Over the past decade, a few interesting ADLs have been introduced together with their supporting software tools. These ADL include MIMOLA, UDL/I, nML, ISDL, CSDL, Maril, HMDES, TDL, LISA, RADL, EXPRESSION and PRMDL.

EXISTING RETARGETABLE SIMULATORS

Anahita Processor Description Language (APDL), APDL [17] is one of the most recent contributions in the area of retargetable simulator. The language was introduced in 2007 by N. Honarmand et al. from the Shahid Beheshti University, IRAN. The Primary difference between APDL and other ADLs is the addition of Timed Register Transfer Level (T-RTL), which enables the simulation designer to define the latencies and hardware requirement of the processor operations. This separation of configuration data enables APDL to better integrate with external software for analysis as the T-RTL data is organized separately from the remainder of the processor description. Moreover, APDL can describe both instruction and structure descriptions of a target processor.

The Pascal-like syntax of APDL is clearly more intuitive than many other ADLs such as LISA and EXPRESSION. While the language is easier to read and understand, the researchers have not yet implemented a compiler to produce simulations. Furthermore, despite APDL's relative ease, users are still faced with the task of learning the details of the syntax.

ISDL [18] was introduced in 1997 by G.Hadjiyiannis, S.Hanono, and S. Devadas from Massachusetts Institute of Technology. The purpose of ISDL was to provide a language for describing instruction sets along with a limited amount of details of a processor structure for the automatic construction of compilers, assembler, and simulators. ISDL enables users to define their target processors in several ways. First, users can define operations, their format, and the associated assembly language instruction. Second users can define the storage resources available to the processor, including the register file and memory. Third users can define constraints in the processor such as instructions requesting the same data path, or restrictions regarding assembly syntax.

ReXSim [19] was introduced in 2003 by a computer architecture research team at Irvine. ReXSim is an extension of EXPRESSION language which sought to improve

simulation speed by integrating a novel method of decoding instructions of the simulated program before execution of the simulation. As a result, the instruction decoding process was removed from the execution loop of the simulator, and thus improved the simulation speed significantly. Using this method, the team was able to produce retargetable simulations that showed performance in excess of major simulators like SimpleScalar, which is widely considered to be a simulation performance benchmark.

Reduced Colored Petri Net (RCPN) [20] was introduced in 2005 by M.Reshadi and N. Dutta from University of California, Irvine. RCPN takes a vastly different approach to retargetable simulation, in which pipelines are modeled using a simplified version of Colored Petri Nets (CPN). Petri Nets are graph based mathematical method of describing a process. The nodes of the graph represent particular discrete events, states, or functions, and the graph edges represent the transitions of data between nodes. The transitions can be enabled or disabled based on conditions specified at the nodes.

The purpose of RCPN is to provide retargetable simulations for modeling of pipelined processors. RCPN reduces the functionality of a regular CPN by limiting the capabilities of the nodes in the graph for the purpose of increasing simulation speed and usability. Additionally, RCPN takes the advantage of some of the natural properties of CPNs to prevent structural and control hazards.

Retargetable functional simulator (Fsimg) [21] focus on tools that deal with the machine language of processors, like assemblers, disassembler, instruction set simulator etc. The objective was to have a single processor model for all the tools. Hence Retargetable Function Simulator (Fsimg) was designed using Sim-nML language which is primarily an extension of the nML language for processor modeling. Fsimg takes the specification of the processor in the *intermediate representation* and an executable for the processor in ELF.

Format and generates a *functional simulator (Fsim)* which in turn gives the functional behaviour of the processor model for the given program. Around 237 instructions have been specified with the resource usage model and pipeline. *Macro Preprocessor (nMP)* for processing Sim-nML macros is implemented.

It has some limitation. Fsimg is imposing a strong restriction on specification writing. Current bit-operator library supports only integer data types. The trace produced by Fsim is not compressed. It makes it difficult to handle and process trace files. It is very slow.

The LISATek [22] processor design flow is based on LISA 2.0 processor models. Given a LISA model, the LISATek tool is able to generate instruction-set simulators for the processor under design. Typically, the debugger in form of a dynamic library directly uses the generated simulator. However, a compiled static simulator library is also generated, and specifications exist to integrate it into the system environment. The system environment would be the MPARM. All the core models generated by the LISATek suite, regardless of the nature of the ASIP at hand, have the same interface.The implementation of these function calls depends completely on the communication method used in the system. The implemented API will translate the requests into SystemC signals which can be understood by the MPARM [23] platform. The Assessment of the performance of alternative hardware communication is not addressed. Retargetability is poor.

All of these simulators use techniques to speed up the execution of application programs. This is achieved by minimizing the amount of details about the processor, needed for program execution on the simulator. Even though some of these previous approaches target ADL-based automatic toolkit generation and DSE, not much work has been done in bringing together these elements in an early DSE environment. Furthermore, previous approaches are restricted to certain classes of processor families and assume a fixed memory/cache organization. For a wide variety of such processor and memory IP library, the designer needs to be able to specify and analyze the interaction between the processor instruction set and architecture, and the application and explore the different points in design space.

This problem is addressed in SIMPRESS simulators. The EXPRESSION ADL captures both the instruction set and architecture information for a design draw from an IP library. The library contains a variety of parameterizable processor cores and customizable memory / cache organizations. Simpress produces a structural simulator capable of providing detailed structural feedback in terms of utilization, bottle-necks in the processor architecture.Though SIMPRESS Simulators addresses many issues, it has certain limitation. The application having function calls are not supported. Compilation steps exist in three passes: PcProGUI, Expression console, acesMIPS console. Basically it is very complex to understand the process of compilation and simulator. The Application needs .proc and .def file. The .c program generates these files. There is no clear cut method as how .c is converted to .proc and .def, especially in case of windows environment. This is strong limitation as we can not simulate our own program written in .c. this has to be first converting to .procs and .defs and for that we need to depend on their servers to provide for the same, which is not functional right now.

In order to overcome all these complexities, we suggest a simple and elegant solution. Just there is a need to provide the standard application program in the form of scheduled and optimized code along with the processor description to our Simulator and you will get the cycle count as an output of the simulation.

WHAT IS VEX

VEX ("VLIW Example") is a compilation-simulation system that targets a wide class of VLIW processor architectures, and enables compiling, simulating, analysing and evaluating C programs for them VEX includes three basic components:

A. The VEX Instruction Set Architecture:

VEX defines a 32-bit clustered VLIW ISA that is scalable and customizable. Scalability includes the ability to change the number of clusters, execution units, registers and latencies; customizability enables users to define *special-purpose* instructions in a structured way.

B. The VEX C Compiler:

The VEX C compiler is a derivation of the Lx/ST200 C compiler, itself a descendant of the *Multiflow* C compiler. It exposes some of the parameters to allow architecture exploration by changing the number of clusters, execution units, issue width and operation latencies, without having to recompile the compiler.

C. The VEX Simulation System:

The VEX simulator is an architecture-level (functional) simulator that uses *compiled simulator* technology.
The VEX simulator used a so-called *compiled simulation* technique. The compiled simulator (CS) translates the VEX binary to the binary of the host computer, by first converting VEX to C, and then invoking the host C compiler to produce a host executable.

In addition to the standard semantics of the instructions, CS also emits instrumentation code to count cycles (and other interesting statistics), as well as code to dump the results to a log file at the end of the simulation. Timing instrumentation is turned on with the "-mas_t" flag passed to the compiler driver (or "-mas_ti" and "-mas_td" for finer grain control - see the section on compiler flags).

CS operates on each of the individual VEX assembler (.s) files corresponding to the compilation units of a program and translates them back to C by implementing the VEX operation semantics, the calling convention (ABI), and introducing the appropriate instrumentation code. The CS-generated C files are then compiled with the host platform C compiler (e.g., gcc for Linux) and linked with the support libraries that deal with the instrumentation. During linking, the CS ld wrapper ensures that the right libraries are linked in the right order, and performs the necessary "magic" (such as wrapping system functions so that they don't cause problems) for the binary to execute correctly.By default, VEX links in a simple cache simulation library, which models an L1 instruction and data cache. The cache simulator is really a trace simulator, which is embedded in the same binary for performance reasons, but only communicates with the VEX execution engines through simple events that identify memory locations, access types and simulation time.

INSTALLATION AND CUSTOMISATION FOR VEX

A sample compilation and Simulation steps can be listed as follows

A. Compile the VEX with the _asm() calls:

a. <vex>/bin/cc c average.

B. Compile (natively) the asm library:

b. gcc c asmlib.c

C. Link (with the VEX compiler) the 2 parts together:

c. <vex>/bin/cc o average average.o asmlib.o

D. Run the average binary:

The first example is a simple "compile-and-run" sequence of a program composed of two compilation units *file1.o* and *file2.o*

```
                x[k] = q+y[k]*(r*z[k+10]+t*z[k+11]);
        }


}

bash-3.00$ ls -ltr
total 410
-rwxrwxrwx   1 root     root        1992 Jan 31  2011 vextypes.h
-rwxrwxrwx   1 root     root        8157 Jan 31  2011 vexopc.h
-rwxrwxrwx   1 root     root         691 Jan 31  2011 makefile
-rwxrwxrwx   1 root     root         858 Jan 31  2011 main.c
-rwxrwxrwx   1 root     root        2888 Jan 31  2011 lex.l
-rwxrwxrwx   1 root     root       17717 Jan 31  2011 grammar.y
-rwxrwxrwx   1 root     root         865 Jan 31  2011 README.txt
-rwxrwxrwx   1 root     root        6749 Jan 31  2011 LICENSE.txt
-rwxrwxrwx   1 root     root         375 Sep  9 11:05 LL1.c
-rwxrwxrwx   1 root     root      155517 Sep  9 11:08 test_orig.s
-rwxrwxrwx   1 root     root        1095 Sep  9 11:08 test.s
-rwxrwxrwx   1 root     root          70 Sep 10 14:42 test.c
bash-3.00$ ../../bin/cc -ms -O4 -c LL1.c
```

Figure 1: Command to compile and generate .s file

Figure 1 shows a command regarding compilation of the program.

Compile individual modules

/home/vex/bin/cc -ms -O4 c file1.c
/home/vex/bin/cc -ms -O4 c file2.c

Link (with math library)

/home/bin/cc o test file1.o file2.o -lm

Run the program

./test

```
                x[k] = q+y[k]*(r*z[k+10]+t*z[k+11]);
        }


}

bash-3.00$ ls -ltr
total 410
-rwxrwxrwx   1 root     root        1992 Jan 31  2011 vextypes.h
-rwxrwxrwx   1 root     root        8157 Jan 31  2011 vexopc.h
-rwxrwxrwx   1 root     root         691 Jan 31  2011 makefile
-rwxrwxrwx   1 root     root         858 Jan 31  2011 main.c
-rwxrwxrwx   1 root     root        2888 Jan 31  2011 lex.l
-rwxrwxrwx   1 root     root       17717 Jan 31  2011 grammar.y
-rwxrwxrwx   1 root     root         865 Jan 31  2011 README.txt
-rwxrwxrwx   1 root     root        6749 Jan 31  2011 LICENSE.txt
-rwxrwxrwx   1 root     root         375 Sep  9 11:05 LL1.c
-rwxrwxrwx   1 root     root      155517 Sep  9 11:08 test_orig.s
-rwxrwxrwx   1 root     root        1095 Sep  9 11:08 test.s
-rwxrwxrwx   1 root     root          70 Sep 10 14:42 test.c
bash-3.00$ ../../bin/pcntl LL1.s
```

Figure 2: Command to analyze the file

The assembler files are useful to check the static behavior of the compiler, and can be analyzed with the *pcntl* utility which collects static compile information from a VEX assembler file.
This is shown in Figure 2.
For example, if we invoke

Analyze *file1.s*

/home/vex/bin/pcntl file1.s

```
Procedure: main::
 Trace     IPC  Cycles    Oper    Copy     Nop
-----------------------------------------------
     3    2.83         6      17       0       0
     1    1.58        24      38       0       0
     4    1.56        25      39       0       4
Operations     = 94
Instructions   = 55
Reg. moves     = 0
Nops           = 4
Avg ILP = 1.70909
```

Figure 3: Output of Sample Programs

The sample output is shown in Figure 3.

PERFORMANCE ESTIMATES AND VALIDATION OF SIMULATOR

The Framework is based on VLIW based processor architecture. A VEX architecture issues multiple operations in an instruction in a single cycle, and these operations are executed as a single atomic action (this is sometimes called VLIW mode). Instructions are executed strictly in program order, but within an instruction, all operands are read before any results are written. For example, it is possible to swap the value of a pair of registers in a single instruction. Instructions cannot contain sequential constraints among their operations. An exception caused by an instruction may not affect the execution of any instruction that was issued earlier and must prevent the instruction generating the exception from modifying the programmer visible state.

Table 1: Benchmark Programs along with Description

No	Name	Description
1	SIM-A-BENCH#1(SIM1)	Excerpt from a hydrodynamic code
2	SIM-A-BENCH#2(SIM2)	Standard Inner product function of Linear Algebra
3	SIM-A-BENCH#3(SIM3)	Excerpt from a Tridiagonal Elimination routine
4	SIM-A-BENCH#4(SIM4)	First Sum
5	SIM-A-BENCH#5(SIM5)	First Difference

The execution behavior is that of an in-order machine: each instruction executes to completion before the start of the next one. In other words, all syllables of an instruction start together and commit their results together. Committing results includes modifying register state, updating memory, and generating exceptions. Table 1 lists all the benchmarks programs that have been used to validate the simulators.

After running this benchmark program on the SIM-A as well as VLIW based Vex Simulator, following results were obtained.

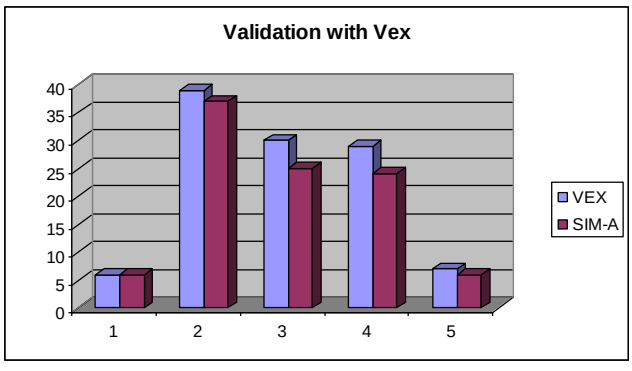

Figure 4: Comparative analysis of SIM-A and Vex Simulator of Cycle Count

Figure 4 show the graphical analysis of the SIM-A and SimpleScalar Simulator.

CONCLUSION AND FUTURE DIRECTION

In this paper we have verified SIM-A Simulator with VLIW based Vex Simulator. This paper discuss the working and configurationally issues involve in Vex Simulator. The different customization needed to run the application program has been discussed in detail.

SIM-A Simulator developed at our MLSU embedded Lab generates the performance estimates for the application under consideration. Processor description is captured in the form of GUI, which allows the user to specify the architecture in visual form. The cycle accurate, structural simulator generated using SIM-A allows the user to collect statistics called cycle count. It definitely helps the designer to analyze the design and modify the critical portions.

The SIM-A environment has been designed to allow modeling of diverse range of processors. This has been demonstrated to an extent through the modeling of VLIW processor.

REFERENCES

[1] Manoj Kumar Jain, M. Balakrishnan, Anshul Kumar. "ASIP Design Methodologies: Survey and Issues "In proceedings of the IEEE/ACM International Conference on VLSI Design. (VLSI 2001)", pages 76-81, January 2001.

[2] Y Subhash Chandra. Retargetable functional simulator – M.Tech Thesis, Department of Computer Science, IIT Kanpur, June 1999.

[3] FREERICK, M. The nML Machine Description Formalism, July 1993.

[4] JAIN, N.C. Disassemble using High level Processor Models. Master's thesis, Department of Computer Science and Engg, IIT Kanpur, Jan 1999.

[5] UNIX System V Rel 4, Programmers Guide : ANSI C and Programming Support Tools. PHI, New Delhi 1992. Executable and Linkable format (ELF), Tools Interface Standards (TIS), Portable Formats Specification, Version 1.1.

[6] M. Yourst, "Ptlsim." http://www.ptlsim.org/. Jan. 2010.

[7] "M5." http://www.m5sim.org. Jan2010.

[8] "bochs: The open source IA-32 emulation project." http://bochs.sourceforge.net/. Jan. 2010.

[9] J. Emer, P.Ahuja, and E.Borch, "Asim: A performance model framework" pp.68-76, 2002.

[10] "Gxemul" http://gxemul.sourceforge.net/ Jan 2010.

[11] P.M et al. , "Simics : A Full system simulation platform, " Computer, Vol.35, pp. 50-58, 2002.

[12] D. Wallin, H.Zeffer, M.Karlsson, and E.Hagersten, "Vasa: A Simulator infrastructure with adjustable fidelity," Parallel and Distributed Computing, 2005.

[13] M.M. et al., "Multifacets general execution-driven multiprocessor simulator (gems) toolset," SIGARCH Computer Architecture News, pp. 92-99, 2005.

[14] "SimpleScalar LLC." http://www.simplescalar.com/, August 2010

[15] S.M. et al., "Wisconsin wind tunnel ii: A fast and portable parallel architecture simulator," Workshop on performance Analysis and Its Impact on Design, June 1997.

[16] V. Pai, P. Ranganathan, and S.Adve, "Rsim : An execution-driven simulator for ilp-based shared memory multiprocessor

and uniprocessors," Third Workshop on Computer Architecture Education, Feb 1997.

[17] N. Honarmand, H.Sohofi, M. Abbaspour, and Z.Navabi, " Processor description in APDL for design space exploration of embedded processors," Proc. EWDTS, 2007.

[18] G.H. et al . ,"ISDL : An Instruction set description language for retargetability," In proc Design Automation Conference , pp.299-302,,1997.

[19] Mehrdad Reshadi, Prabhat Mishra, Nikhil Bansal, Nikhil Dutt. "Rexsim : A Retargetable framework for instruction-set architecture simulation" CECS Technical Report #03-05 ,Feb,2003

[20] M. Reshadi and N.Dutt, "Generic pipedlined processor modelling and high performance cycle-accurate simulator generation," Vol.2, pp. 786-791, 2005.

[21] Y Subhash Chandra. Retargetable functional simulator – M.Tech Thesis June 1999.

[22] Fedrico Angiolini,;Jianjiang Ceng; Rainer Leuper ;Cesare Ferri;Luca Benini; "An Integrated Open Framework for Heterogeneous MPSoc Design Space Exploration",page3 , Date06,2006 EDAA.

[23] M.Loghi; F.Angioni; D.Bertozzi; L.Benini. "Analyzing on-chip communication in a MPSoC environment" In proceeding of the 2004, Design, Automation and test in Europe Conference (DATE'04), IEEE, 2004.

SHORT BIODATA OF THE AUTHOR'S

Gajendra Kumar Ranka He is a research Scholar of Department of Computer Science, MLSU University, Udaipur Rajasthan. His area of research is embedded system.

M.K. Jain received the M.Sc. degree from M.L. Sukhadia University, Udaipur, India, in 1989. He received M.Tech. degree in Computer Applications and PhD in Computer Science & Engineering from IIT Delhi, India in 1993 and 2004 respectively. He is Associate Professor in Computer Science at M.L. Sukhadia University Udaipur. His current research interests include application specific instruction set processor design, wireless sensor networks, semantic web and embedded systems.

