ISSN No. 0976-5697

Volume 3, No. 2, March-April 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

An Efficient Retargetable Simulator SIM-A for ASIP DSE and Validation with RISC and VLIW based Processors

Dr. Manoj Kumar Jain*
[1]Associate Professor in Computer Science,
Mohanlal Sukhadia University,
Udaipur, India
manoj@cse.iitd.ernet.in

Gajendra Kumar Ranka
Software Engineer,
Fujitsu, Pune
gajendra_ranka@hotmail.com

Abstract: General Purpose Processor (GPP) provides high application flexibility but at a high cost in terms of more silicon area, high power consumption and low performance. In systems where high application flexibility is not required, it is possible to trade off flexibility for lower cost by tailoring the processor to the application to create an Application Specific Instruction set Processor (ASIP) with high performance yet low silicon cost. If we look at the rapid rate at which mobile technology is developing and the constant need for miniaturization, ASIPs seem to be poised in a stronger position compared to ASICs. SIM-A Simulator has been developed that 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. 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 RISC processor and VLIW Processor with traditional memory hierarchies. The technique has been validated against standard toolsets.

Keywords: ASIP; Retargetable Simulator; Simulator; RISC based Simulator; VLIW based Simulator;

I. INTRODUCTION

Modern electronics are controlled by processors that must meet strict constraints in terms of performance, cost, size and power consumption. In a competitive market place, performance and cost are critical in differentiating one product from another. An ASIP is a processor that is designed to efficiently execute the software for a specific 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.

A. 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]

a) Application Analysis

Application is normally written in High level language. Sometimes SUIF can be used as intermediate format. Analysis of the application is essential as it provides the essential requirement from the application that can guide for hardware synthesis as well as instruction set generation.

b) Architecture Design Space Exploration

Output of the Application analysis step along with the range of architecture for Possibility of suitable architecture is explored and the best architecture is selected that satisfy the different characteristics like minimum hardware cost, performance and power.

c) Instruction Set Generation

Till this step we have identified application requirements and the suitable architecture.

d) 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.

e) Hardware Synthesis

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

B. Architecture Design Space Exploration:

System on Chip designs has various goals and objectives. Design space consists of a set of parameters. Architecture under consideration requires a range of good parameter to explore. These parameters may take up the different values. Some of the parameter suggested can be functional unit of different type, Storage units, interconnect resources, number of memory units etc. Further the parameters can also be extended to size of instruction cache and size of data cache. This has been a very crucial step for ASIP design. Design Space exploration helps the SOC designers to make the trade-offs between these goals and arrive at the "optimal" design. Designers explore changes to the architecture or the instruction-set of the processor-memory system. Designers select a suitable architecture that satisfy the performance and power constraint and having minimum hardware cost. Architecture is defined using some suitable architecture description language (ADL).

C. Techniques For Performnace Estimation:

Two major techniques have been used for performance estimation. They are scheduler based and simulator based. In Scheduler based approach, application is scheduled to generate the output like cycle count. Architectural component is already identified at this stage. Target processor architecture can be given in the form of description file. In Simulator based approach, application under consideration runs on a simulator. Depending upon the architecture selected in above steps, application is simulated to compute the performance.

Processor Models are extensively used in system design process. The system design process starts with an application and its implementation. Then the model is tested for its performance and other aspects. In such a scenario an integrated environment is required for the designer where several tools exist like simulator, assembler, compiler etc. Rewriting the tools after each design change is a tedious job. Hence automatic generation of these tools is more desirable according to the design changes.

D. Existing Retargetable Simulator 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

II. 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.

III. 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 interaction is based upon four key pillars:

a. The simulated core can be cycled by calling specific functions. If the processor is modelled in an instruction-accurate fashion, then the generated model can be stepped on an instruction basis. On the other hand, a model derived from a cycle-accurate LISA description can be stepped on both instruction and cycle basis.
b. Core-initiated communication (e.g. reads, writes) is performed through a specific Application Programming Interface (API). It is the task of the external program to provide an implementation of said API.
c. System-initiated communication (e.g. interrupts), if any, can be forwarded to the core when cycling it, and therefore on a fine-grain cycle-by-cycle basis, by proper flipping of extra pins. Of course the LISA core model must be made aware of the meaning of these extra pins to take proper action.
d. An external LISATek Debugger tool can be interfaced to the core via the IPC (Inter-Process Communication) mechanism. The external program must simply invoke the Debugger with proper references; subsequently, the LISATek model and the Debugger interact autonomously.

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. The processor-system description is input using a graphical schematic capture tool, called V-SAT, that outputs an Expression Description which is fed into the toolkit generators to produce DSE tools. The SIMPRESS generated simulator provides feedback information which is back-annotated to the same V-SAT graphical description.

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.

IV. OVERALL APPROACH

Application or a set of application in the form of High Level Language is taken as input and it given as input to retargetable compiler. Architecture description is also given input to retargetable compiler. Retargetable compiler generates the schedule and optimized code. This code is given as input to Simulator SIM-A. None of the existing simulator provides and easy GUI to enter the processor components and simulate the code for target host as shown in Figure 1.

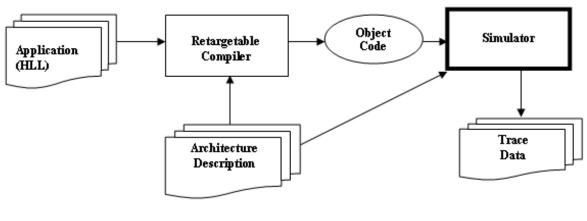


Figure. 1. Simulator based code generation

We are assuming the scheduled and optimize code to be generated from retargetable compiler and this code along with the Processor description or Architecture description is given as input to the Simulator. The Simulator generates the data in the form of cycle count.

V. METHODOLOGY

Our architecture model consists of a number of architecture simulation components that simulate the three main parts of the system - Processing elements, memory and interconnection medium. Figure 2 shows the overall methodology adopted for SIM-A.

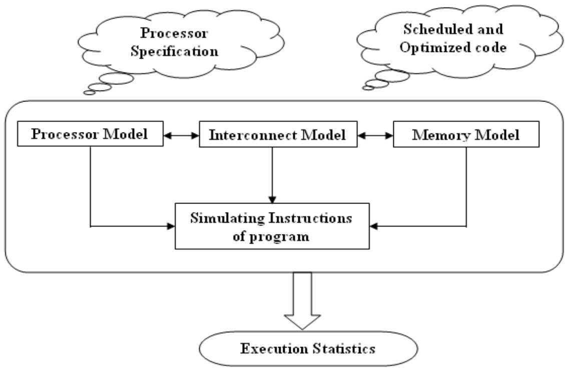


Figure. 2. Overall methodology

A. *Processor Model:*

This method assumes that each instruction completes in a fixed number of cycle. In architecture terms, we can say that our processor has fast private memory where code and local data can be stored. Hence each processing element is equipped with large instruction and local data cache that guarantee a very high hit ratio. Architectural Parameters of the processor component are:

a. Local instruction costs in cycle.
b. Number of available interrupts types.
c. Context switch latency (cycles to save and restore processor state).
d. Interrupt Latency (Cycles needed to save state and branch to interrupt handler).

B. *Memory Model:*

We are considering two-level memory. The levels are main memory and cache(s). We are assuming Shared memory as Main Memory, though we are considering logical partitions with in that memory. Such partitions may be code segment, data segment etc. Cache memory is assumed to be local to each processing element. However any change in this model can be easily incorporated in our model as it can be specified in input description and that is taken care by our simulator. Architectural parameters of the memory components are:

a. Number of memory modules.
b. Memory module size.
c. Cache line size.
d. Cache set size.
e. Number of sets per cache.
f. Cache access latency.
g. Memory module access latency.

C. *Interconnection Model:*

In a real-time system architecture, the notion of a bus component play an important role as it forms the backbone of communication among all the devices of the system. For this purpose, we need a precise specification of buses for applications that will run on top of them.

Our simulator supports the interconnection of bus. All processing elements and all memory modules are connected through a common bus. Uniform shared memory access is assumed, that is, access of any memory module from any processor takes the same amount of time (ignoring delays due to bus contention). The simplest interconnection strategy is to use a single bus which is being shared by every other component for communication. Though this strategy is easy to implement, as the number of processor go up, the bus becomes the bottleneck. All the components connected to this bus should tune their interfaces to use the bus protocol. Apart from this, designers have to implement some arbitration mechanism to resolve the conflicts.

VI. DEVELOPMENT OF RETARGETABLE SIMULATOR

Electronic devices built nowadays are often built with a single IC composed of multitude of hardware blocks that implement the device functionality. In most cases such circuit contains one or more processors that enable to implement a part of the circuit functionality as software that runs on that processor rather than as a specific hardware component. Such IC is commonly referred to as a system-on-a-chip (SoC).

The main CPU features are:

a. 101 instructions with possible addressing modes
b. CPU with independent stack pointer registers
c. Eight 32-bit data, eight 32-bit address and 32-bit status registers
d. 16-bit external memory interface

Main assumptions for the ISS were:

a. Developed in pure Visual basic 6.0 language for high performance.
b. Crystal Report is used as a reporting tool to display the different status.
c. MS Access is used to Store the different schedules and optimized code.
d. Single-instruction accuracy, without taking internal architecture under consideration.
e. Fully static design with the support of loop / wait statements.
f. Usage of native VB types to gain high simulation speed.
g. Communication interfaces separated from functionality.

The main part which contains implementation of main processor's logic (ALU, instruction fetch, decoding and execution routines) together with fields corresponding to the internal resources (all registers). Sub-module features:

a. Fetching and decoding instructions
b. Instruction processing routines
c. Handling interrupts and exceptions
d. Register implementation and registers read/write access
e. Instruction counter

Simulators are critical components of the exploration toolkit for the system designer. Simulators can be used to perform a variety of tasks such as verifying the functionality and / or timing behavior of the system, and generates quantitative measurement, for e.g. Cycle count etc. As per our design Methodology and hypothetical assumption of the Architecture we have taken MIPS Architecture as a base to develop our Retargetable simulator. We have given a Nomenclature to our Simulator as SIM-A {Simulators for Architectures}. We will be using Expression Language for Architecture Description. We have developed the GUI Interface for the same. We have also provided the GUI for easy evaluation and analysis.

A. SIM-A- Basics:

SIM-A is a 32-bit datapath, every instruction is 32 bits wide, and data comes in "words" which are also 32 bits wide. Memory in SIM-A, however, is addressed in bytes. SIM-A is load-store architecture that is, the only instructions that access memory are LW and SW.

B. Memory Organization:

We are considering Two-level memory. The levels are main memory and cache(s). We are assuming main memory as Shared Memory. Cache memory is assumed to be local to each processing element. We are also considering logical partitions with in that memory. Such Partition may be code segment, extra segment etc. Data begins at virtual address 0x10000000 and grows in the direction of increasing virtual addresses (this data is called dynamic data because the machine doesn't know how much of it will be used at runtime). In SIM-A, there is also a concept of stack – that is, data that starts just below virtual address 0x80000000 and grows in the direction of increasing virtual addresses.

C. SIM-A Register Set:

Registers are a small set of fast memory that the datapath has available at its disposal for most immediate operations. All registers are 32-bit wide. SIM-A contains thirty two user registers (that is, registers that the user can access/use in the assembly program) and four special-purpose registers that are hidden from the user.

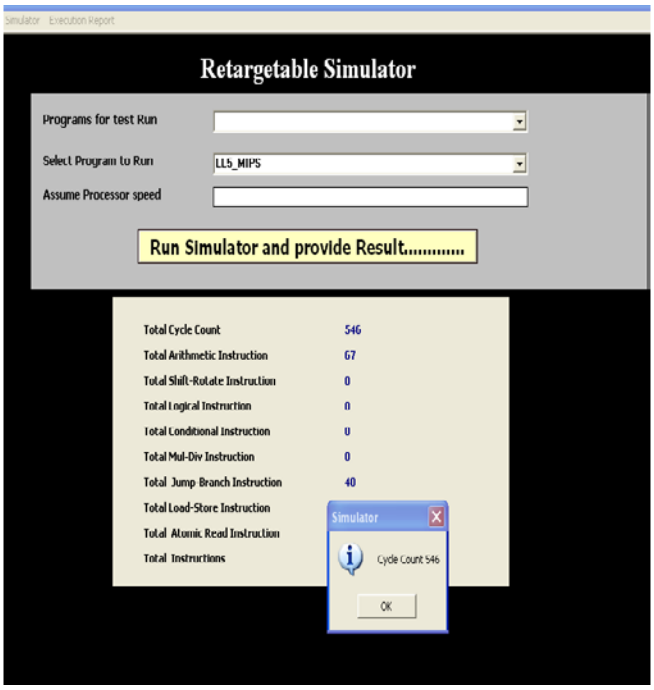
D. SIM-A Instruction Set:

This section describes in detail all the SIM-A instructions. Rs and Rt are source registers – the datapath should fetch their values whenever they are used. Source registers are usually treated as twos-complement signed 32-bit numbers. In some special cases they are treated as unsigned numbers (the note that follows explains such circumstances). Rd is the destination register – the datapath will write the result to that register number.

Immediate values may either be treated as signed or unsigned values, and may either be zero-extended (in which case the padding bits are all zero), or sign extended (in which case the padding bits are all equal to the most significant bit of the immediate value). SOC designs have various design goals. These goals include minimal cost, maximal performance, low power, high reliability, etc. Design Space Exploration allows the SOC designer to make trade-offs between these goals and arrive at an "optimal" design. SOC designer would like to explore changes to the architecture or the instruction-set of the processor-memory system. Common examples of such changes include, but not limited to:

a. *Changing the pipeline structure*. e.g., increasing (or decreasing) the number of stages to increase (or decrease) the clock frequency, adding forwarding paths to reduce pipeline stalls.
b. *Changing the data path structure*. e.g., changing slow units to fast units in order to increase performance, changing connectivity between units and storage elements (like register files) in order to decrease power consumption.
c. *Increasing parallelism*. e.g Adding more functional units that can execute in parallel in order to increase performance.
d. *Changing the instruction-set*. e.g Adding new operations which can be exploited by particular applications.
e. *Changing the memory component*. e.g Changing the size of register file, changing the associativity of the cache, etc.
f. *Changing the memory hierarchy*. e.g Adding a cache between the processor and off-chip memory, changing the on-chip memory hierarchy etc.

E. SIM-A Look and Feel:



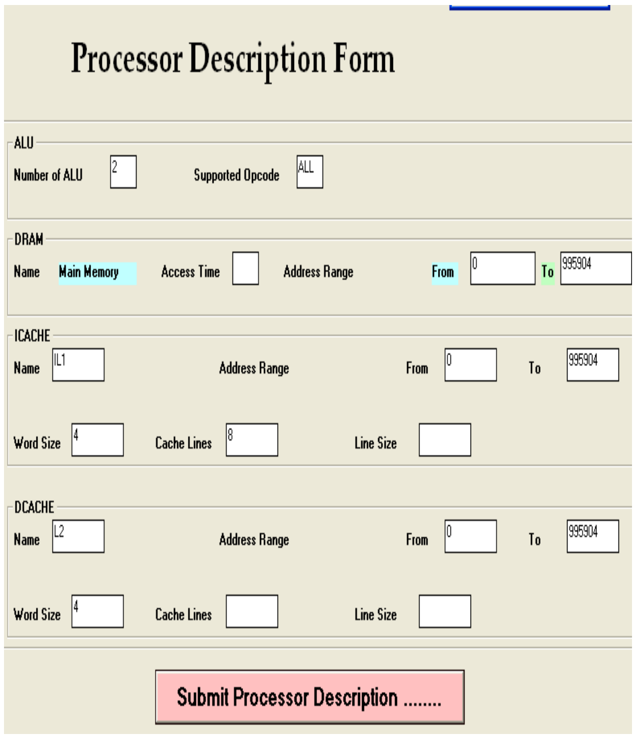
Programs for test Run	
Select Program to Run	LL5_MIPS
Assume Processor speed	

Total Cycle Count	546
Total Arithmetic Instruction	67
Total Shift-Rotate Instruction	0
Total Logical Instruction	0
Total Conditional Instruction	0
Total Mul-Div Instruction	0
Total Jump Branch Instruction	40
Total Load-Store Instruction	
Total Atomic Read Instruction	
Total Instructions	

Cycle Count 546

Figure. 3. GUI for SIM-A simulator

This is the first and main form which helps us to calculate the cycle count of any program as shown in Figure3. First section allows us to select the different programs that we are required to simulate. This is the interface through which user will enter the processor description and will mimic the behaviour of the processor. If you click on the option "Select Program to Run" , it contains all the list of programs. Just select the program that we need to simulate and Click the button "Run Simulator and Provide Result ...." Others buttons are not used right now. Second Section provides the output of the program. It contains information like total Arithmetic instruction, Shift rotate, Logical, Jump Branch etc. It also gives the pop up when the program finishes by providing the cycle count. User will first enter the processor description details as shown in Figure 4. Submit the form to update the processor description file. Then it browses to the GUI form where he can select the programs that he/She needs to simulate.



Processor Description Form

ALU
Number of ALU: 2
Supported Opcode: ALL

DRAM
Name: Main Memory
Access Time:
Address Range From: 0 To: 995904

ICACHE
Name: IL1
Address Range From: 0 To: 995904
Word Size: 4
Cache Lines: 8
Line Size:

DCACHE
Name: L2
Address Range From: 0 To: 995904
Word Size: 4
Cache Lines:
Line Size:

Submit Processor Description ....

Figure. 4. GUI for Processor Description

The above is the brief description of the SIM-A Simulator that has been developed in our Embedded System Lab.

VII. DESIGN SPACE EXPLORATION AND VALIDATION OF SIM-A

Table 1 lists the benchmark programs used to validate the SIM-A simulator. We have validated our developed simulator SIM-A against a number of standard toolsets. This validation also proves easy retagetability and large design space exploration capabilities of SIM-A.

Table 1 List Of Benchmark Programs

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

A. *Validation with SIMPRESS:*

The Framework is based on MIPS 4K like processor architecture. The architecture contains five pipeline stages – fetch, decode, operand read, execute and writeback. There are five parallel issue paths corresponding to two ALU Units, one for floating point unit, a branch unit and a Load/store unit. The memory hierarchy consists of two L1 data caches for instructions and data, a unified L2 cache and a DRAM main memory. There is a 32-bit wide general purpose register file and a 32-bit wide floating point register file, each containing 32 registers. After running the benchmark program on the SIMPRESS as well as SIM-A Simulator, following results are obtained. Figure 5 show the graphical analysis of the SIM-A and SIMPRESS Simulator.

At 1% level of significance, the critical value of '*t*' for (5+5-2) 8 degree of freedom is 3.36 and calculated value is 0.368329. Since the calculated value of '*t*' is 0.368329 which is less than the critical value, which is 3.36, it falls in the acceptance region. Hence it may be concluded that both the results are equally acceptable at 1% level of Significance.

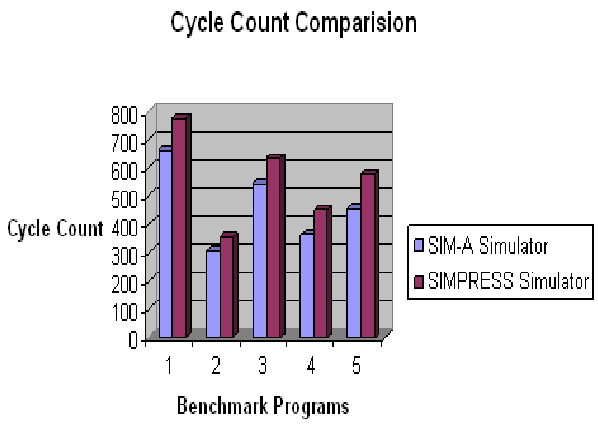


Figure. 5. Comparative analysis of SIM-A and SIMPRESS Simulator of Cycle Count

B. *Validation With Arm Based Keil Software:*

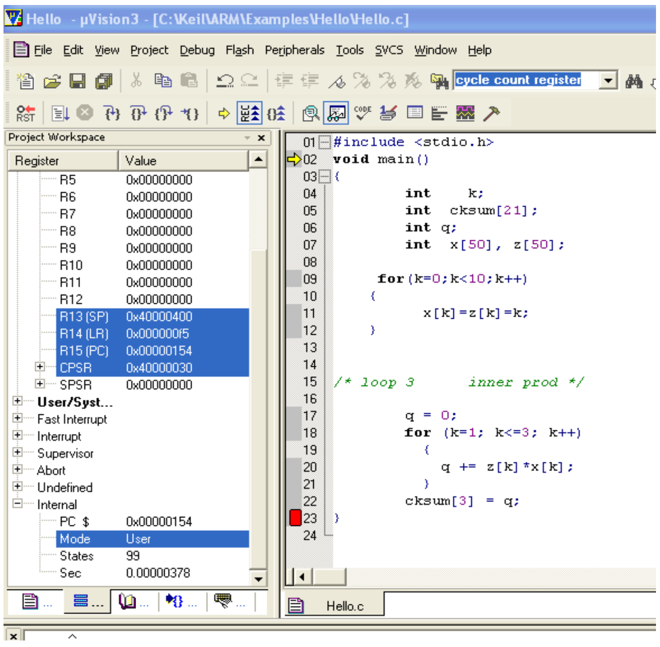
a) *Keil Software and Customizations:*

Keil Software development tools for the ARM microcontroller family support every level of developer from the professional applications engineer to the student just learning embedded software development. µVision3 ensures easy and consistent Project Management. A single project file stores source file names and saves configuration information for Compiler, Assembler, Linker Debugger, Flash Loader, and other utilities. The Project menu provides access to project files and dialogs for project management. When microvision 3 Project started target device needs to be selected from the device database. It displays only those option that are relevant to the selected device.

The Instructions about how to write Keil ARM can be listed as follows.

a. Open Keil uVision Program which is Text Editor of Keil, that ARM uses for writing C Language Source Code Program.
b. Set default value to translate uVision3 Code to use with Keil uVision3 Program and Keil ARM. Click Project Components, Environment, Books… then select default value to use Complier titled Select ARM Development Tools.
c. Open the project and in turn open the .c program and start debugging session. Left hand pane shows the details of the Register details and the states as shown in the Figure 6.

All instructions are 32 bits long. Most instructions execute in a single cycle. Every instruction can be conditionally executed. Data processing instructions act only on registers. Three operand format Combined ALU and shifter for high speed bit manipulation Specific memory access instructions with powerful auto-indexing addressing modes.


Hello - µVision3 - [C:\Keil\ARM\Examples\Hello\Hello.c]
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
cycle count register
Project Workspace
Hello.c

Register	Value
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x40000400
R14 (LR)	0x000000f5
R15 (PC)	0x00000154
CPSR	0x40000030
SPSR	0x00000000
User/Syst...	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x00000154
Mode	User
States	99
Sec	0.00000378

```
#include <stdio.h>
void main()
{
        int     k;
        int   cksum[21];
        int q;
        int   x[50], z[50];

     for(k=0;k<10;k++)
    {
          x[k]=z[k]=k;
    }


/* loop 3       inner prod */

        q = 0;
        for (k=1; k<=3; k++)
          {
             q += z[k]*x[k];
          }
        cksum[3] = q;
}
```

Figure. 6. Editor showing the Register Status and Timing information

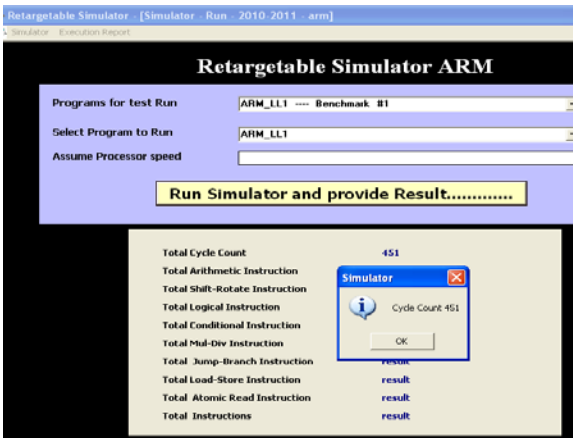
ARM has 37 registers in total, all of which are 32-bits long.

a. 1 dedicated program counter
b. 1 dedicated current program status register
c. 5 dedicated saved program status registers
d. 30 general purpose registers

However these are arranged into several banks, with the accessible bank being governed by the processor mode. Each mode can access as follows.

a. a particular set of r0-r12 registers
b. a particular r13 (the stack pointer) and r14 (link register)
c. r15 (the program counter)
d. cpsr (the current program status register) and privileged modes can also access
e. a particular spsr (saved program status register)

b) *Performance Estimates and Validation:*


Retargetable Simulator - [Simulator - Run - 2010-2011 - arm]
Simulator
Execution Report
Retargetable Simulator ARM
Run Simulator and provide Result............
Simulator
OK

Programs for test Run: ARM_LL1 ---- Benchmark #1
Select Program to Run: ARM_LL1
Assume Processor speed:

```
Total Cycle Count                        451
Total Arithmetic Instruction
Total Shift-Rotate Instruction
Total Logical Instruction
Total Conditional Instruction
Total Mul-Div Instruction
Total  Jump-Branch Instruction           result
Total Load-Store Instruction             result
Total  Atomic Read Instruction           result
Total  Instructions                      result
```

Cycle Count 451

Figure. 7. GUI for ARM-BASED SIM-A

Benchmark programs are selected and run on SIM-A Simulator through a GUI as shown in Figure 7. Once a suitable input program is selected and simulator is run it provides useful outputs including total cycle count. We are able to validate SIM-A Simulator against ARM Based Keil simulator. Figure 8 shows the comparative analysis of SIM-A with ARM based Keil software.

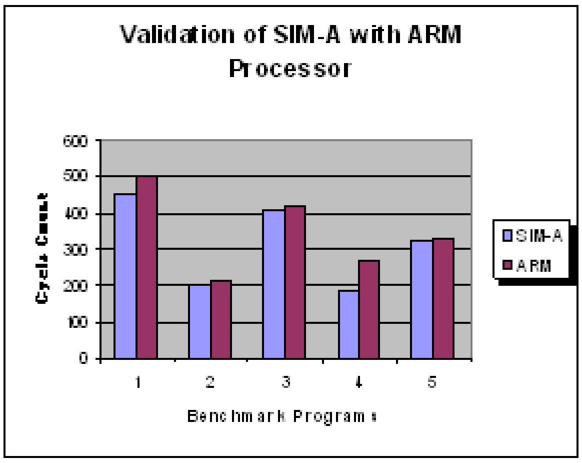


Figure. 8. Comparative analysis of SIM-A and arm based KEIL software for cycle count

C. *Validation With SIMPLESCALAR:*

a) *Simplescalar Basics:*

SimpleScalar is an open source computer architecture simulator developed by Todd Austin.It is a simulator, which in simple words means that it can be used to show that Machine A is better than Machine B without building either Machine A or Machine B. It is written using 'C' programming language. SimpleScalar is a set of tools that model a virtual computer system with CPU, Cache and Memory Hierarchy. Using the SimpleScalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure. The SimpleScalar tools are used widely for research and instruction. The simulators available are sim-fast, sim-safe, sim-profile, sim-cache, sim-bpred, and sim-outorder,

b) *Installation and Customization For Simplescalar:*

The following text describes the procedure of installing the simulator Simplescalar on a modern Linux distribution. The procedure has been tested on Ubuntu Linux 8.04 but should work on most Linux distributions. If you use Ubuntu and 8.10 (Intrepid Ibex) or later. Download the necessary source code files.

i. simpletools-2v0.tgz
ii. simplesim-3v0d-with-cheetah.tar.gz
iii. simpleutils-990811.tar.gz
iv. gcc-2.7.2.3.ss.tar.gz

Setup some environment variables (assuming your home directory is /home/oracle and that you are using a bash-compatible shell1):

```
$ export IDIR=/home/oracle/simplescalar
$ export HOST=i686-pc-linux
$ export TARGET=sslittle-na-sstrix
```

Create the directory "simplescalar" under your home directory, and copy all the four tarballs into it:

```
$ mkdir $IDIR
$ mv simplesim-3v0d-with-cheetah.tgz $IDIR
$ mv simpletools-2v0.tgz $IDIR
```

$ mv simpleutils-990811.tar.gz $IDIR
$ mv gcc-2.7.2.3.ss.tar.gz $IDIR
Update Ubuntu (or whatever distribution you are using) with the following packages:
(i) build-essential, (ii) flex, and (iii) bison
Just un-pack the package file, and remove the old gcc folder
$ cd $IDIR
$ tar xvfz simpletools-2v0.tgz
$ rm -rf gcc-2.6.3

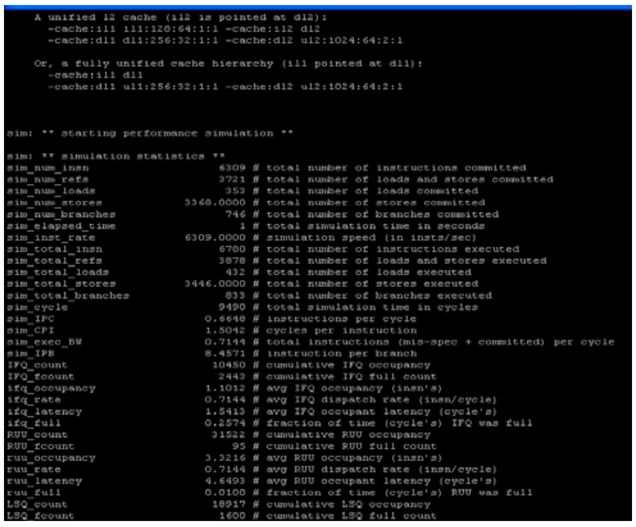
a. Installing Simpleutils

Un-pack the package file as follows:
$ cd $IDIR
$ tar xvfz simpleutils-990811.tar.gz
$ cd simpleutils-990811

b. Building Simulators

Un-pack the simulator package as follows :
$ cd $IDIR
$ tar xvfz simplesim-3v0d-with-cheetah.tar.gz
$ cd simplesim-3.0
$ make config-pisa
$ make
You may test the installation of simplesim by:
$ ./sim-safe tests/bin.little/test-math
Un-pack the source code and configure the installation as follows :
$ cd $IDIR
$ tar xvfz gcc-2.7.2.3.ss.tar.gz
$ cd gcc-2.7.2.3
$ ./configure –-host=$HOST –-target=$TARGET –-with-gnu-as
Next build the compiler:
$ make LANGUAGES=c CFLAGS=-O CC="gcc -m32"
$ make enquire
$ ../simplesim-3.0/sim-safe ./enquire -f > float.h-cross
$ make LANGUAGES=c CFLAGS=-O CC="gcc -m32" install

In a separate directory, create a new file, bench.c, that has the code with empty body main function. Compile it using the following command:
$IDIR/bin/sslittle-na-sstrix-gcc –o bench bench.c
That should generate a file *bench*, which we will run over the simulator:
$IDIR/simplesim-3.0/sim-safe bench
In the output, you should be able to find the following as shown in Figure 9.



```
    A unified l2 cache (il2 is pointed at dl2):
      -cache:il1 il1:128:64:1:l -cache:il2 dl2
      -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

    Or, a fully unified cache hierarchy (il1 pointed at dl1):
      -cache:il1 dl1
      -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l


sim: ** starting performance simulation **

sim: ** simulation statistics **
sim_num_insn                   6309 # total number of instructions committed
sim_num_refs                   3721 # total number of loads and stores committed
sim_num_loads                   353 # total number of loads committed
sim_num_stores            3368.0000 # total number of stores committed
sim_num_branches                746 # total number of branches committed
sim_elapsed_time                  1 # total simulation time in seconds
sim_inst_rate             6309.0000 # simulation speed (in insts/sec)
sim_total_insn                 6780 # total number of instructions executed
sim_total_refs                 3878 # total number of loads and stores executed
sim_total_loads                 432 # total number of loads executed
sim_total_stores          3446.0000 # total number of stores executed
sim_total_branches              833 # total number of branches executed
sim_cycle                      9490 # total simulation time in cycles
sim_IPC                      0.6648 # instructions per cycle
sim_CPI                      1.5042 # cycles per instruction
sim_exec_BW                  0.7144 # total instructions (mis-spec + committed) per cycle
sim_IPB                      8.4571 # instruction per branch
IFQ_count                     10450 # cumulative IFQ occupancy
IFQ_fcount                     2443 # cumulative IFQ full count
ifq_occupancy                1.1012 # avg IFQ occupancy (insn's)
ifq_rate                     0.7144 # avg IFQ dispatch rate (insn/cycle)
ifq_latency                  1.5413 # avg IFQ occupant latency (cycle's)
ifq_full                     0.2574 # fraction of time (cycle's) IFQ was full
RUU_count                     31522 # cumulative RUU occupancy
RUU_fcount                       95 # cumulative RUU full count
ruu_occupancy                3.3216 # avg RUU occupancy (insn's)
ruu_rate                     0.7144 # avg RUU dispatch rate (insn/cycle)
ruu_latency                  4.6493 # avg RUU occupant latency (cycle's)
ruu_full                     0.0100 # fraction of time (cycle's) RUU was full
LSQ_count                     18917 # cumulative LSQ occupancy
LSQ_fcount                     1600 # cumulative LSQ full count
```

Figure. 9. Output of sample programs

c) Performance Estimates and Validation Of Simulator:

The Framework is based on MIPS 4K like processor architecture. The architecture contains five pipeline stages – fetch, decode, operand read, execute and writeback. There are five parallel issue paths corresponding to two ALU Units, one for floating point unit, a branch unit and a Load/store unit. The memory hierarchy consists of two L1 data caches for instructions and data, a unified L2 cache and a DRAM main memory. There is a 32-bit wide general purpose register file and a 32-bit wide floating point register file, each containing 32 registers. After running this benchmark program on the SIM-A as well as SimpleScalar Simulator, following results were obtained. Figure 10 shows the graphical analysis of the SIM-A and SimpleScalar Simulator.

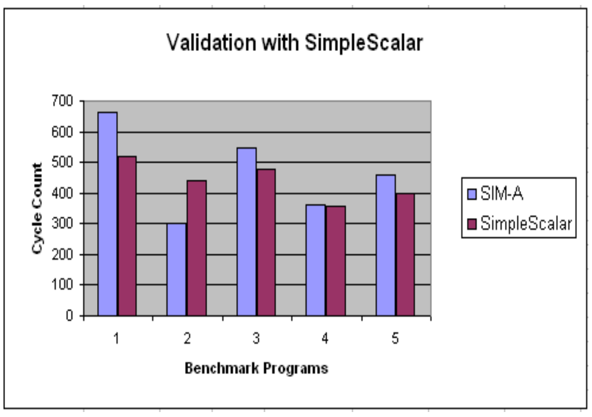


Figure. 10. Comparative analysis of SIM-A against SIMPLESCALAR simulator for cycle count

D. Validation With OVPSim:

a) OVPSim Basics:

It is simulator to develop software on: Fast Simulation, Free open source models, easy to use. Imperas developed some virtual platform and modeling technology to enable simulating embedded systems running real application code. These simulations run at speeds of 100s and 100s of MIPS on typical desktop PCs and are completely Instruction Accurate and model the whole system. OVP has three main components - the OVP APIs that enable a C model to be written, a collection of free open source processor and peripheral models, and OVPSim a fast, easy to download and use simulator that executes these models.OVP put together a simulation model of a platform, compile it to an executable, and connect it to your debugger to provide a very efficient fast embedded software development environment. The focus of OVP is to accelerate the adoption of the new way to develop embedded software - especially for SoC and MPSoC platforms.

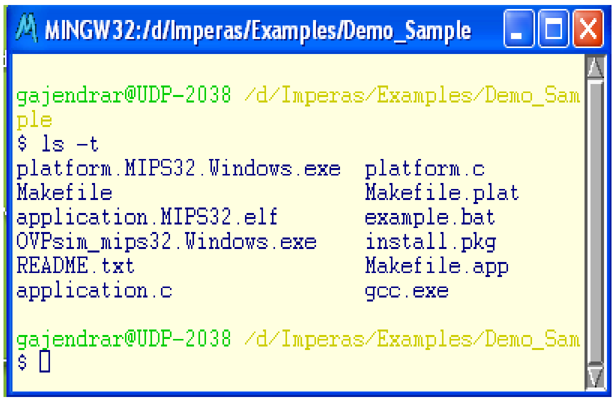
If any software is developed to run in an embedded system we would normally be using an Instruction Set Simulator (ISS). As there are multiple processors or cores in design we need more than just a single ISS. What is needed is a model of your platform that includes models of all the processors or cores and models of the peripherals and behavioral components that the software communicates with. This is a Virtual Platform, or more simply just a simulation model of your design. OVP provides the different libraries of processor and behavioral models, and APIs for building you own processors, peripherals and platforms. This is just what is needed to use existing models or build your own, and OVP is easy to use, open, flexible, and importantly, free for non-commercial use.

b) Installation and Customisation for OVPSim:

The Imperas professional tools and the OVP simulator is installed on Windows Platform. The Imperas tools are superset of the OVP tools. Either delete the OVP installation before installing or install to a different directory. The Windows versions are provided as installers as an executable, either
imperas.<version number>.<dot release>.exe or
OVPsim.<version number>.<dot release>.exe or
OVPsim.Windows.exe (for current version from (www.OVPworld.org).
The following environment variables are automatically set by the installer:
IMPERAS_HOME Points to the root of the installation
IMPERAS_VLNV Points to the compiled library
IMPERAS_ARCH Set to the Host architecture ie Windows
IMPERAS_RUNTIME Specifies which simulator, Imperas (CpuManager) or OVPsim, to load at runtime
PATH modified to include
IMPERAS_HOME/bin/IMPERAS_ARCH

The development of platforms, processor and peripheral models on the Windows operating system has been validated in an environment using MSYS and MINGW. A default build environment is provided with both the Imperas tools and OVPsim installations that will allow models and platforms to be built in this environment.

We need the installation files of either OVPSim or from Imperas tools and atleast one compiler toolchain. In order to cross compile applications under windows the installation of MSYS/MinGW is done. In our Demo we use the openCores openRISC MIPS32 as the target embedded processor. The OVP and Imperas tools are licensed using FLEXlm. Tool execution need a license file. A license file bounds to individual computer through computer's host ID. OVPsim looks for its license file in $IMPERAS_HOME/OVPsim.lic


```
MINGW32:/d/Imperas/Examples/Demo_Sample

gajendrar@UDP-2038 /d/Imperas/Examples/Demo_Sam
ple
$ ls -t
platform.MIPS32.Windows.exe  platform.c
Makefile                     Makefile.plat
application.MIPS32.elf       example.bat
OVPsim_mips32.Windows.exe    install.pkg
README.txt                   Makefile.app
application.c                gcc.exe

gajendrar@UDP-2038 /d/Imperas/Examples/Demo_Sam
$
```

Figure. 11. Essential file needed to run the application

Imperas provides pre-built toolchains for processors by OVP. The Processors include MIPS32, OR1K and ARM. The Makefile is available to provide a default build environment for an application onto a processor. The Makefiles is in the format
<Processor Type>.Makefile.include.

We need to perform some customization to work on MIPS32 processors. We are required to create a folder that contains the entire required file required for execution process.
(a) Copy the different files to a suitable folder as shown in Figure 11. Each file is needed as each file has its own significance.
(b) Change the Makefile to point to MIPS32 processor.
i.e Change the line
CROSS?=MIPS
(c) Create the application.c file and copy the program in application.c. View the application program.
(d) Compile the program, check for any error and remove it.
(e) Run the program.
(f) The program will generate the different statistics as shown in Figure 12. Validation results are presented graphically in Fifure 13. Results show that estimates produced by our simulator and OVPSim simulator are fairly consistent.

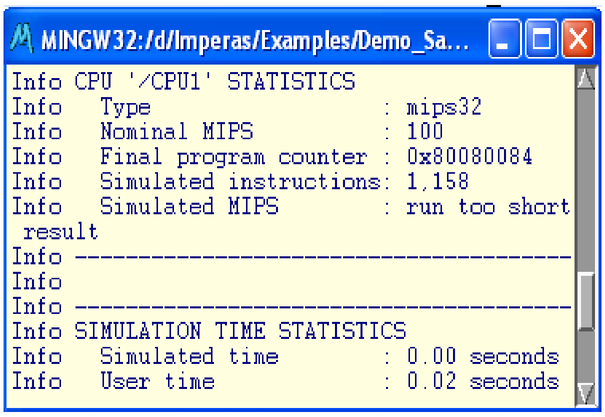

```
MINGW32:/d/Imperas/Examples/Demo_Sa...
Info CPU '/CPU1' STATISTICS
Info   Type                    : mips32
Info   Nominal MIPS            : 100
Info   Final program counter   : 0x80080084
Info   Simulated instructions: 1,158
Info   Simulated MIPS          : run too short
 result
Info ------------------------------------------
Info
Info ------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time          : 0.00 seconds
Info   User time               : 0.02 seconds
```

Figure. 12. Statistics generated by simulator

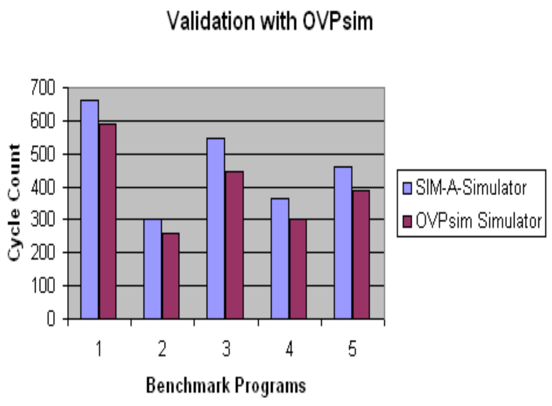


Figure. 13. Comparative analysis of SIM-A and OVP simulator for cycle count

E. Validation with VEX Simulator:

a) VEX Basics:

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:

i. 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.

ii. 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.

iii. 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.

b) Installation and Customisation for VEX:

A sample compilation and Simulation steps can be listed as follows

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

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

ii. 2. Compile (natively) the asm library

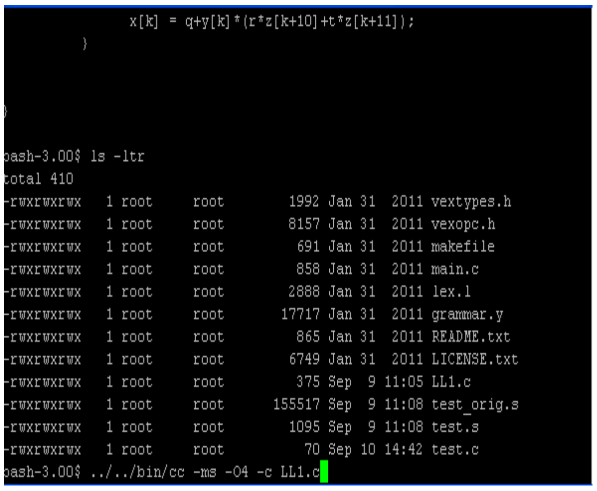
b. gcc c asmlib.c

iii. 3. Link (with the VEX compiler) the 2 parts together

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

iv. 4. 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*. Figure 14 shows a command regarding compilation of the program.



```
            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. 14 Commands to complie and generate .s file Compile individual modules

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

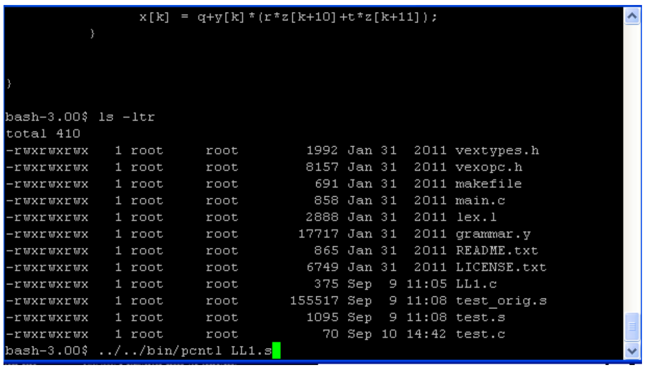
Link (with math library)

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

Run the program

./test

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 15. For example, if we invoke the following command the output produced is shown in Figure 16.



```
            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
```

Fig. 15. Commands to analyze the file

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. 16. Output of sample programs

c) Performance Estimation and Validation:

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.

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

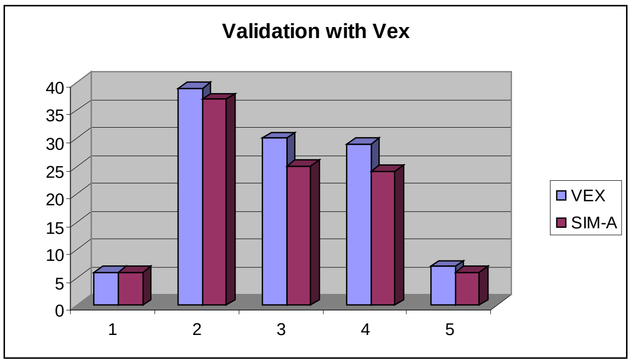


Fig. 17. Comparative analysis of sim-a against vex for cycle count

VIII. CONCLUSION

System-on-a-Chip (SOC) technology enables the system designer to develop highly customized embedded systems that can meet demanding performance, power, cost and size constraints. However there is urgent need for an environment that will allow the system designer to rapidly specify and evaluate design alternatives.

In this paper we presented a SIM-A Simulator entirely developed at our Lab that 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.

IX. 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 Bio Data for the Authors

Dr.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.

Gajendra Kumar Ranka has done his Ph.D. in Computer Science is a research Scholar of Department of Computer Science, MLSU University, Udaipur Rajasthan. Currently, he is Software Enineer at Fujitsu, Pune. His area of research is embedded system design and software toolset development..