Architecture and Simulation of Very Long Instruction Word Reconfigurable Instruction Set Processor (VLIW-RISP)

Submitted by

Uzma Saeed
275-FAS/MS/CS/F05

Supervised by

M. Aqeel Iqbal
and
Asim Munir

**Department of Computer Science,
Faculty of Applied Sciences,
International Islamic University,
Islamabad**

IN
THE
NAME
OF ALLAH
THE MOST BENIFICIENT
AND THE MOST MERCIFUL

Department of Computer Science
International Islamic University
Islamabad

Date of External 25th August, 2009

FINAL APPROVAL

It is certified that we have read the project report submitted by Ms. Uzma Saeed reg. no 275-FAS/MS/CS/F05 and that this project is of sufficient standard to warrant its acceptance by International Islamic University, Islamabad for the Masters of Science in Computer Sciences.

COMMITTEE

External Examiner

Dr. Abdul Sattar
Director General (Ret.)
Pakistan Computer Bureau

[illegible]

Internal Examiner

Mr. Muhammad Nadeem
Assistant Professor,
International Islamic University,
Islamabad

[illegible]

External Supervisor

Mr. M. Aqeel Iqbal
Assistant Professor,
Fauji Foundation University,
Rawalpindi

M. Aqeel Iqbal

Internal Supervisor

Mr. Asim Munir
Assistant Professor,
International Islamic University,
Islamabad

[illegible]

ACKNOWLEDGEMENT

First of all I would like to sincerely thank my supervisors Mr. M. Aqeel Iqbal and Mr. Asim Munir for giving me this wonderful opportunity to work on this project under their kind supervision and guidance throughout the project. I do acknowledge their true coordination and support.

Then at last but not the least, I would like to thank my dear parents for their moral and all kind of support during this project. I never forget their help during the tough times of the project.

Thank you all again!

Uzma Saeed
275-FAS/MS/CS/F05

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amount to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Uzma Saeed
275-FAS/MS/CS/F05

Table of Contents

Table of Figures

List of Tables

Chapter No. 1
Introduction

Chapter No.1

INTRODUCTION

The revolution occurred in the field of embedded systems due to the microelectronics market is ever-increasing. In such a context the definition of efficient and cost-effective design approaches is mandatory. Hardware and software co-design solutions generally take into account architectures composed of one or more standard microprocessors and of suitable application specific integrated circuits executing the most time-critical segments of the application. Recently, the innovative concept of "mass customization" has been introduced. This concept considers the possibility of specialization of a micro-processor instruction set so as to optimize its performance for a given application or for a group of applications. Such an approach combines the time efficiency of application-specific functional units with the flexibility of *programmable logic circuits*. This innovation of programmable technology opens doors towards a new field of research known as *Flexible Instruction Set* Micro-processors.

In a broader sense the different levels of coupling or integration can be envisioned in architecture. A more traditional design approach is that of considering the reconfigurable part as a coprocessor, which is effectively working like a hardware accelerator that stalls the core-processor when under execution. Normally the co-processor approach requires coarse-grain functions to be executed in the part and therefore the speedup given by the co-processor program execution, when compared to processor application execution, must be considerably high [4]. A more realistic or innovative approach and an interesting challenge in academic terms is one that sees the part of the processor *as a Functional Unit* while operating in parallel with the other data-paths of the processor and where an extension of the instruction set is executed. The programmable processor so envisioned issues a set of native instructions to the native Functional Units, while these new customized instructions are issued to the Reconfigurable Functional Units (RFUs) [5]. A fine-grained function can be chosen for implementation in the part when compared to the co-processor approach since the integration is much higher and the RFU can be reached without any additional delay being embedded in the micro-processor.

Reconfigurable Architectures can be divided in two main categories: *fine grained* and *coarse grained* architectures [2]. The fine-grained architectures are based on programmable devices such as FPGAs (Field Programmable Gate Arrays) which include units (CLBs – Configurable Logic Blocks) that perform single functions on a bit basis. On the other hand, the coarse-grained architectures include word length units or small microprocessor distributed on an array of processing units. All architectures also include I/O, memory and Inter connect units. The advantages of coarse-grained architectures over fine-grained ones are mainly the reduction of configuration time and reconfiguration memory [1]. Fine-grained architectures also use significantly more area overhead to routing functions between CLBs and expend significantly more energy.

Reconfigurable computing devices can be configured after their fabrication to solve any computational algorithm or task. Such kind of reconfigurable devices are best exemplified now-a-days by FPGA [3]. In such devices the algorithms or tasks are implemented by spatially composing the built-in or primitive operations and operators with the possibility of temporally *varying or* changing the hardware of the operators. The re-configurable processor on FPGA can perform different operations on each bit of data or program and hence the re-configurable devices can be optimized to the data width of streaming data flows. The main theme of this kind of research work is to mix the advantages of non Von-Neumann architectures with the advantages of re-configurable processing devices or fabrics.

Field-Programmable Gate Array (FPGA) is a kind of silicon chip containing a set or an array of configurable logic blocks known as CLBs [3]. Unlike an Application Specific Integrated Circuit (ASIC) which can perform a single dedicated or specific function for the lifetime of the chip; a FPGA can be re-programmed many times to perform a variety of different functions in a matter of micro-seconds. Before it is programmed an FPGA knows nothing about how to communicate with the external connected devices surrounding it. Hence this is in fact both a blessing and a curse as it allows a great deal of flexibility in using the FPGA while greatly increasing the complexity of programming it. This type of generic ability to re-program FPGAs has led them to be widely used by hardware engineers and designers for prototyping digital electronic circuits. The performance advantage achieved from the FPGAs derives from the fact that the programmable hardware is likely to be customized to a particular algorithm. The field programmable gate arrays are configured to comprise only the operations that are appearing in the concerned algorithms [6]. The specialized instruction set micro-processor in fact contain ALUs of specific or specialized data bandwidths like 8-bits, 16-bits and 32-bits and always has pre-coded or determined control flow patterns.

The re-programmability and versatility of FPGAs definitely comes at a price. Only a few years ago, the algorithms or tasks that could be implemented in a single FPGA chip were very small. For example in 1995 the largest FPGAs could be programmed for circuits of about maximum of 10,000 to 15,000 logic gates at most. Since only a fast 32-bit adder requires a few hundreds logic gates, the capabilities of such devices were somewhat bounded. More recently the FPGAs have reached a size where it is possible to implement reasonable sub-pieces of an application in a single FPGA part [7]. This has led to an emerging new concept for computing. If a processor was to include one or more FPGA-like devices, it could in theory support a specialized application-specific circuit for each program.

The unlimited re-configurability of an FPGA permits a continuous sequence of custom circuits to be employed where each one is optimized for the task of the moment. Because FPGAs demonstrate a better performance scale than superscalar techniques, such designs have the potential to make better use of continuing advances in device electronics in the long term. The idea of reconfigurable computing has been a subject of research for a more than a decade, but most projects have investigated the potential of connecting one or more commercial FPGAs to an existing micro-processor via a standard external bus such as the PCI bus [8]. If reconfigurable computing is really to become the computing paradigm of the future high speed platforms, then the main parts must be brought closer together. Only a few studies have considered the integrating of a micro-processor core and FPGA into a single device with the both tailored to co-operate very closely with each other and so there remains an important question about how such a device might be built and re-programmed and how it would fit within an existing general purpose-computing framework. Such a question must be addressed before the bigger issue of whether reconfigurable computing is really a good design model for computations can be answered.

Reconfigurable computing using reconfigurable devices like FPGAs have become an alternative to fill the gap between ASICs and general-purpose computing systems. Although the basic concept of reconfigurable computing was proposed in the 1960s, the reconfigurable computing systems have only recently become very vital and quite feasible. This is mainly due to the availability of high-density VLSI devices that use the programmable switches and routing networks to implement the extremely flexible hardware architectures. Most of the reconfigurable systems consist of a general-purpose processor core tightly or loosely coupled with reconfigurable logic [9]. These systems can implement specific functionality of applications or a set of applications on reconfigurable hardware rather than on the general-purpose processor and hence providing significantly better performance.

In a ***statically programmed system the*** individual data operations of an application will remain idle when they are no more required. For an example, the data dependencies within an application program may cause an operation to be idle and waiting for data inputs from other operations. Hence placing all operations onto the FPGA all at once is a poor choice hence resulting of wasting of a large no of precious hardware resources. Run-time re-configuration can be used to remove or recover such kind of idle operations by making them share limited hardware resources. Also the run-time reconfiguration provides a design method for large class of applications that are too big for the available hardware resources on the FPGA [10].

Many recently advanced systems, such as Garp [4], PipeRench [11] and Chimaera [10], are using run-time reconfiguration. In such kind of systems the hardware configuration can change frequently at run-time to reuse hardware resources for several different parts of a computation. Such systems have the potential to make more effective use of chip hardware resources than even standard well designed ASICs, where fixed hardware may be used only in a portion of the application algorithm or computation. Also the run-time reconfigurable systems have been shown to accelerate a variety of applications. An example of such kind of systems is the run-time reconfiguration within *automatic target recognition* (ATR) application developed at the UCLA to accelerate a template matching. The algorithm in this system was based on a correlation process between incoming image data from the radar and a set of target templates. Without taking considerations of the reconfiguration time, this system improves performance by a factor of 25 to 30 over a general-purpose computing system.

However, the drastic speed advantages of run-time reconfiguration do not come without a cost. By requiring a set of multiple reconfigurations to complete a computation, the time it takes to reconfigure the FPGA becomes a really significantly a key concern. The serial-shift configuration method transfers programming bits into the FPGA device in a serial way. This type of very slow method or programming approach is still used by a large class of existing FPGAs [7], [12]. Recent devices have moved the technology to cutting-edge domain and hence resulting in FPGAs with more than one million logic gates. The configuration size for such devices is more than one megabyte [13]. It could take from few hundred milliseconds to few seconds to transfer such a large configuration data stream using the serial-shift approach.

In most of such systems the devices must sit idle while they are being reconfigured and hence wasting cycles that could otherwise be performing useful work. For example, the ATR system uses 98% of its execution time on performing reconfiguration process, meaning that it uses merely 2% of total time in doing computation. DISC and DISC-II systems have been found to spend up to 90% [6] of their execution time on performing reconfiguration process. It is obvious that a significant improvement in system performance can be achieved by eliminating or reducing this configuration overhead associated with reconfiguration delays.

1.1 RC Architecture

Reconfigurable Instruction Set Processor (RISP) design offers many advantages over ASIC Processor design. It offers the flexibility of in circuit hardware re-programmability. By using RISP design we can get the speed advantage of nearly an ASIC Processor design and flexibility of software. RISP design is not an easy task [11]. In RISP designing the most important issue is the reconfiguration methodology. Many different techniques have been introduced to provide an efficient reconfiguration process including partial reconfiguration, run time reconfiguration, static reconfiguration and the most recently introduced configuration cloning [1], [2]. Most of these methods of reconfiguration suffer from a problem of excessive reconfiguration (reconfiguration overheads).

Till now the reconfigurable computing is suffering from this problem and no one has provided satisfactory solution. Reconfiguration time should be minimized in order to obtain a level of satisfactory performance [3], [4], [11].This can be easily achieved by using the already existing resources within the programmed device [15]. But the solution to this problem is in software tools not hardware [14].

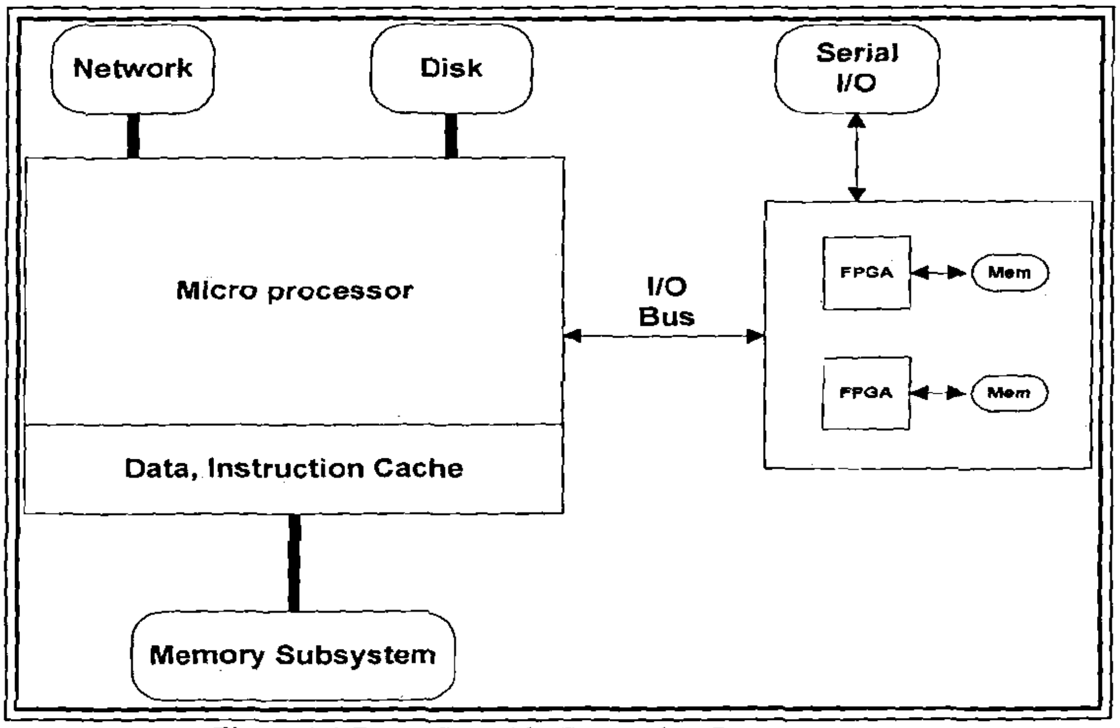


Figure 1.1 Reconfigurable Computing System

VLIW Architectures

Very Long Instruction Word architecture refers to a CPU architecture that has been designed to take the advantage of instruction level parallelism (ILP) found in the program execution. A micro-processor that executes every instruction one after the other may use the micro-processor resources inefficiently at any time instant, potentially leading to drastically poor performance. The performance of such system can be improved by executing different sub-steps of sequential instructions simultaneously using the concept of pipelining or even executing the multiple instructions entirely simultaneously as is done in superscalar micro-processor architectures [15], [16].

Increase of computational performance is better achieved, for this case, if a micro-processor architecture supporting instruction level parallelism is chosen as the architectural platform or paradigm. Instruction level parallelism processing has become the new emerging architectural challenge since the eighties up to now, by enabling issue and execution of multiple instructions of an application within the same clock cycle. This paradigm would allow our envisioned reconfigurable architecture to fully exploit the native functional unit in parallel with the customized, reconfigurable cores.

Two main classes of ILP machines naming superscalar and VLIW have been flourishing. The former performs dynamic scheduling algorithms of instructions, and therefore granting binary compatibility with previous code versions, the latter relies on static or compile time scheduling, by delivering all instruction dependence analysis to the related compiler.

1.2 Motivation

Future interactive multimedia applications will be based on standards like MPEG-4. Using an object-based approach to describe and composite an audiovisual scene, MPEG-4 combines many different coding tools not only for natural audio and video but also for synthetic objects and graphics. Objects are coded and transmitted separately and composed at the decoder side, letting the receiver interact and influence the way the scene is presented on the receiving display and speakers. Due to this user interaction, the number and the type of decoders that needs to be implemented on the system is not known at the design time, but rather at the run-time [17].

This fact forces the designers of the platforms for these applications to use new approaches. Traditionally, multimedia applications have been implemented on custom VLIW processors that provide enough parallelism to accelerate these computation intensive applications [18], while at the same time retaining low power consumption. In order to increase even further the computational power of these devices, they have been enhanced with costume hard ware for acceleration of the most common multimedia operations. An example of this is the Trimedia Processor [18], which contains the specialized units for DCT (Discrete Cosine Transform) and motion estimation.

Unfortunately, due to the a variety of the algorithms that can be used in new interactive applications and the fact that the actual number and the type of the objects is not known till run time, it is no longer economically viable to make specialized functional units for each algorithm. The picture is further complicated if we also take into account that a platform designed for these applications may have to decode an object encoded with an algorithm for which it was not conceived. Therefore, in order to maintain the power efficiency and the real time constrains, we need a platform that can be specialized at run-time to the algorithm at hand. A platform based on reconfigurable instruction set processors (RISPs) provides this type of run-time specialization [5].

Chapter No. 2
Literature Review

Chapter No.2

Literature Review

Introduction

Reconfigurable computing architectures have been evolved from the most prominently Field Programmable Gate Arrays (FPGAs). Recently, there is a variety of FPGAs now available commercially. A large class of computing systems has been developed by integrating multiple FPGAs chips and dedicated memory modules. A small no of systems have been coupled with a general purpose processor or some kind of application specific integrated circuit core such as a DSP processor to the field programmable gate arrays. In order to minimize the communication overhead and memory access bottlenecks being faced by the system for configuration bits the new arriving computing systems are integrating a reconfigurable logic onto the single same chip as that of the processor core chip [19].

2.1 Classification

There has been much different reconfigurable computing architecture that has been developed over the last few years by researchers. Reconfigurable computing architectures can be broadly classified based on several different parameters. In the following section, some of the most distinguishing architectural parameters which can be used to classify reconfigurable computing architectures have been discussed.

2.1.1 Granularity

The granularity of the computational or reconfigurable logic is the size of the smallest functional unit that is addressed by the software mapping tools. In general the FPGAs have smaller granularity such as 2-input or 4-input functional units [20]. Several reconfigurable computing architectures such as Chameleon [19] implement coarse grain computational or arithmetic units of larger size such as 32-bits. Lower granularity in fact provides more flexibility in adapting the hardware to the computational algorithm. But, lower granularity has a great performance penalty due to the larger delays introduced when constructing computation modules of larger size using smaller functional units. A class of reconfigurable computing architectures implements features that are specifically targeted towards the reduction of these computational overheads.

2.1.2 Host Coupling

A large amount of logic is utilized as a processing fabric attached to a host micro-processor. The host micro-processor performs the control and supervision functions to configure the logic, schedule data input and output streams, external interfacing, among other the things. The type of logic coupling to such a host system dictates the computational as well as configuration overheads in utilizing logic to speed-up computations. *The degree of coupling in fact affects the reconfiguration and the data access cost.* The degree of coupling can be roughly partitioned into three basic classes:

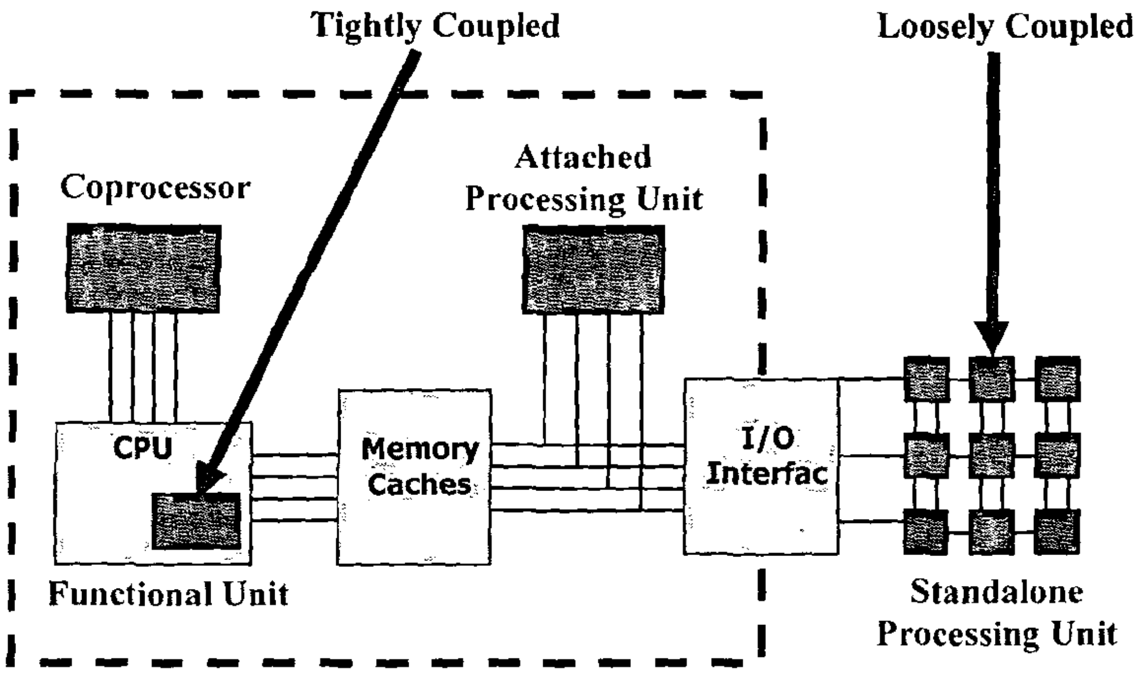


Figure 2.1 Host Coupling Approaches

System-level Coupling (Loosely coupled): This type of coupling includes the computing architectures which have logic communicating to the host through an I/O interface similar to other peripheral devices. A large class of initial FPGA based logic boards were architected with this degree of host coupling. Such architectures include SPLASH.

Chip-level Coupling (Coprocessor or Attached Processing Unit): These systems reduce the overheads in communicating to the relevant host by using direct communication between the host and the reconfigurable logic. An example of such a computing architecture is the PRISM [25]. A large class of the existing computing architectures with reconfigurable logic has been architected using this technique.

Tight On-chip coupling (Tightly coupled): The availability of large class of the transistors has resulted in the intensive integration of reconfigurable logic on the same chip as a host micro-processor, and hence significantly reducing the communication overheads between different components of the architecture. Such kind of architectures includes the well known Garp, Chameleon etc.

2.1.3 Reconfiguration Methodology

Conventionally, a reconfigurable device is configured by downloading a sequence of bits known as a bit-stream onto the device [21] during its operation. The speed and methodology being used during the download of bit stream depends on the interface supported by the device [22]. There are two main types of interfaces namely, Bit-serial and bit-parallel interfaces. The time required for the configuration of the device is directly proportional to the size of the bit-streams as well as on the type of the interfaces being used for this purpose [22]. Fine-grain and Coarse-grain devices have difference in the configuration overheads or time because course grain devices typically need smaller amount of configuration bit-streams. The flexibility of reconfiguration is achieved at the cost of reconfiguration cost. Reconfigurable logic has to stop computations for initiating a new configuration process. This reconfiguration time or overhead can be significant, especially for fine-grain multi-million gate FPGAs.

Some architectures support partial and dynamic reconfiguration processes. Partial reconfiguration allows the reconfiguration of the functionality of a piece or portion of the device while the remaining portion retains its functionality [23]. On the other hand the dynamic reconfiguration allows the reconfiguration of a piece or portion of the device while other portions of the device are performing computations [23].

A large class of other computing architectures addresses this problem by utilizing multiple contexts of reconfiguration or a reconfiguration cache [24]. Both are similar in basic principle. Some configurations of the device can be stored in on-chip memory. At run-time, it is less expensive to switch to one of the configurations being available in these memory chips or areas compared with loading a new configuration from external memory devices [24]. The organization of the cache varies among the computing architectures. Some computing architectures implement the architecture as an external memory device, whereas some other architectures have distributed context memories. For example, Chameleon RCP has a cache holding one configuration on-chip, which allows single cycle reconfiguration completion [27], [28].

2.1.4 Memory Organization

The computations being performed on the reconfigurable logic needs to access data from memory. Intermediate results from computations also need to be stored back before the reconfigurable logic can be reconfigured to perform the next computation [29]. *The organization of the memory system affects the data access cost and is a really significant fraction of the actual execution time or overhead.* Recently the most of the computing architectures include a large memory on the reconfigurable logic device [31], [33]. This memory can be implemented as large memory blocks just like those being available in Virtex FPGA as a Block RAMs or as distributed memory blocks just like those being available in Chameleon LSMs [34], [35].

2.2 Reconfigurable Computing Architectures; A Survey

In the latest computing domain the parallel processing techniques being based on Field Programmable Gate Arrays first time appeared in the domain of computing in the year of start of 1985-1990. In a broader sense the reconfigurable computational architectures can be classified into four main categories:

1. The Input / Output Bus Accelerators Systems
2. The Massively Parallelly Processing FPGAs Architectures
3. The Reconfigurable Computing Super-computers
4. The Reconfigurable Processing Logic Co-processors

2.4 Related Work

2.4.1 Coupling of Reconfigurable Architecture and a Multithreaded Processor Core with Integrated Real-Time Scheduling [47]

This research paper defines a real-time interface between the simultaneous multi-threaded CarCore micro-processor and a MOLEN-based reconfigurable computing unit. The CarCore is in fact IP core that enables simultaneous execution of one hard-real-time thread and further multiple non-real-time threads. The type of the hardware coupling described in this research paper extends CarCore by a reconfigurable computing hardware such that the both can execute different threads simultaneously, while the real-time behaviour of the hard-real-time thread is not disturbed. The main challenge under consideration is the design of a common memory interface for both, the CarCore and the reconfigurable computing hardware such that the memory operations fulfil hard-real-time constraints. Experimental results with an MJPEG benchmark have been obtained which show an overall application speedup of 2.75 which approaches the theoretically attainable maximum speedup of 2.78.

2.4.2 Introduction to Reconfigurable Computing Architectures [48]

In fact this mentioned research paper describes an overview of the research of the currently developed hardware and software based systems for reconfigurable computing architectures. This research paper also presents the alternating techniques that dedicatedly are targeting the concept of run-time reconfiguration process. They conclude this discussion by considering FPGAs in general and also by an exploration of the various hardware architectures used in reconfigurable computing systems. Further they worked at the layer of software based applications that were required for the assembling or de-compilation or compilation of the algorithms to reconfigurable computing systems and the trade-offs between manual map and rout and automatic techniques. Further more they have discussed FPGAs hardware in details and have also presented the detailed study about the hardware level coupling of the reconfigurable computing devices.

2.4.3 Reconfigurable Instruction Set Processors from a Hardware/Software Perspective [49]

In this research paper the authors have presented the design alternatives for reconfigurable instruction set processors (RISPs) from a hardware/software point of view. Reconfigurable instructions set processors are in fact programmable processors that contain a reconfigurable logic in one or more of their functional units. Hardware design of such processors can be split in two main tasks. First task is the design of the reconfigurable logic and the design of the interfacing mechanisms to the rest of the micro-processor. Among the most important design parameters include the granularity of the reconfigurable logic, the design or structure of the configuration memory, the instruction encoding formats and the type of instructions being supported. On the software side the code generation tools require new techniques to be coping up with the reconfigurability of the processor. Aside from the traditional computing techniques, the code generation requires the creation and evaluation of new reconfigurable instructions and the selection of instructions to minimize reconfiguration time.

2.4.4 Re-configurable VLIW processor for streaming data [50]

This research paper describes the instruction set level design of a reconfigurable VLIW processor for streaming data applications with alternating data bandwidths. It discusses the design of reconfigurable data stream processor, the design of VLIW processor for the reconfigurable approach, data control and address path design of the configurable VLIW and generating the FPGA code - VLIW re-configurable procedure.

Reconfigurable RISC processor for variable data bandwidths:

The reconfigurable processor core is in fact a two-address machine with RISC instruction set architecture and orthogonal general purpose register file.

- Address bus width is of size of 16 bits.
- Data busses width is of sizes of 8-bits, 16-bits, 32-bits and 64 bits for different functional units (ALU, GPR)

Re-configurable systolic array - the data width sorter:

The reconfigurable systolic array - *the data width sorter* is based on the hardware design research work. The research in Generic Algorithms is centered on the development of a novel design which uses systolic arrays. The generic concept is in fact extended by exploiting the pipeline architecture and principle to design a device that is independent of the lengths of the chromosomes being used in a particular problem [36]. The systolic arrays themselves are easily scalable to implement different population sizes. Prototype systolic array cells have been designed and targeted to the Xilinx XC4000 FPGA [37].

Re-configurable VLIW-CPU instruction set and format

The first task designing the instruction set is to discuss the instruction to join the instruction set for the data stream approach in order to ensure ISA and EXO compatibility of the processor. Each VLIW instruction has 8 major fields:

- The *Systolic sorter* fields controls the systolic operation ALU and the global LOAD/STORE operations via crossbar. The information on the streaming data type sorted on every data output of the systolic sorter is coded as output in the *FPGA Condition Code Registers* of the systolic sorter

- The *R-CPUa, R-CPUb, R-CPUc and R-CPUd* fields control the four R-CPU's function. The R-CPU is a two-address machine.

- The *FPU_memory* and *FPU_control* fields control the 32 bit RISC *FixedProcesor Unit (FPU)* in performing LOAD/STORE and/or control oprerations.

- The *FPGA-code* contains the FPGA-SRAM images of the RPU and systolic units.

Data control and address path design of the configurable VLIW

The VLIW core implements the host function for the systolic sorter and the four reconfigurable R-CPU calculators. The VLIW core executes all ALU, control and LOAD/STORE instructions in the program. There are no streaming data instructions. The main task of the VLIW core is to synchronize the Out-of-Order the operations of the R-CPU and the systolic sorter to execute the FPGA based code to reconfigure the R-CPUs and to invoice the LOAD/STORE operations for the systolic sorter. The *crossbar switch* between the *R-CPU data registers*, the *main memory* and the execution units is in fact a central part of the VLIW architecture. The *R-CPU data* register set is read-only through this device which virtually provides it with four ports. The crossbar switch extends the R-CPU data register set's read ports, making four "vertical" buses for all R-CPU and each bus is connected to one of the input ports of the dual-port memory with "horizontal" buses. It also performs some data width formatting (byte, word, etc). Accessing a R-CPU data register takes two cycles.One cycle for the register set and another for the crossbar.

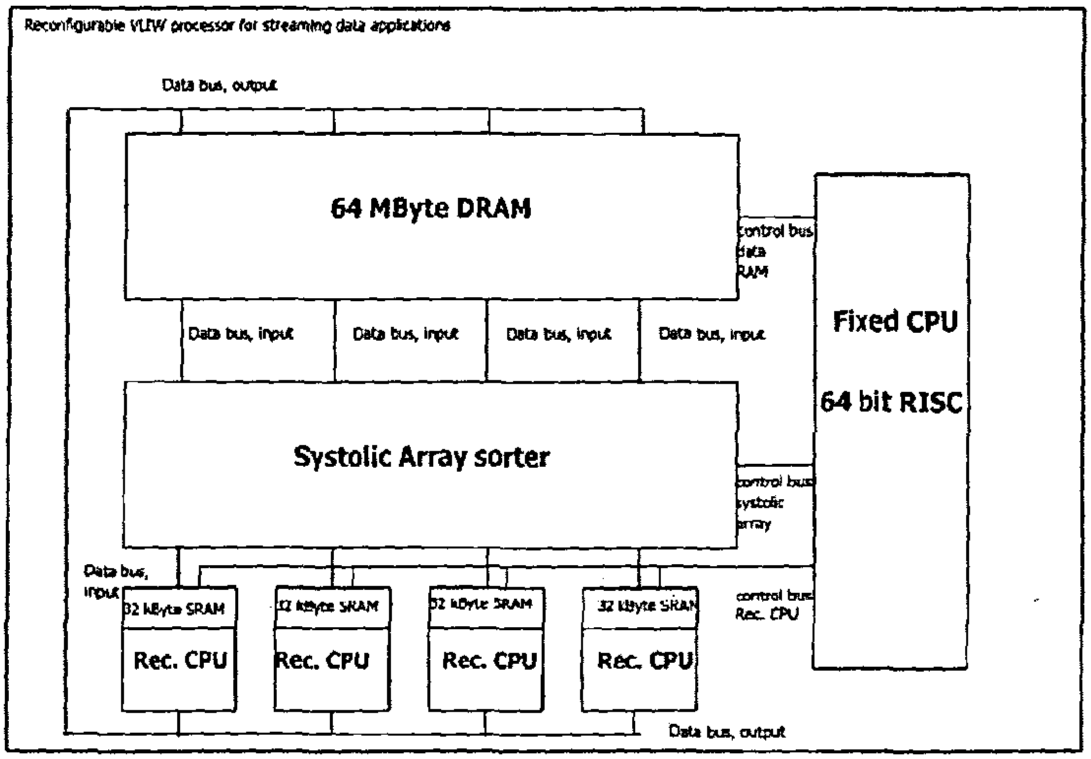


Figure 2.2 Generating the FPGA code - VLIW re-configurable procedure

The main task of the systolic sorter is to generate a *condition code* for the different data widths as the result of sorting the streaming data. The compiler drives reconfigurations of the FPGA prior to execution of the application code, or possibly at the beginning of every section of code that requires reconfiguration.

2.4.5 Reconfigurable Computing Systems Design: Issues at System-Level Architectures

In this research paper the authors discuss the issues involved in the design space of reconfigurable computing systems. They have identified nine key steps in RCS design as application analysis, system partitioning into hardware (HW) and software (SW), architectural design space analysis, mapping of the design library onto the architecture, partitioning of fixed HW and RLU of HW part, reconfiguration process, HW and SW synthesis, compilation and scheduling tasks and Integration of all the components. They briefly described the different models, architectures, compilation and scheduling of tasks, reconfiguration methods, optimal mapping of the design library on the RLU and the state-of-the art of RCSs. Finally they explain how they are going to solve some of the issues and methods in their system design. The nine steps of system design described in seven steps are as follows.

1. **First step**

The first step is the modeling different architectural choices for a given application which will be optimized in terms of performance versus either given constraints or default design constraints after the application analysis.

2. **Second step**

The second step is the proposing the optimized reconfigurable the architecture for a given application by exploring the different 'design space of the architecture' for reconfigurable architectures.

3. **Third step**

The third step is the translating application onto DFG/CDFG or Hybrid architecture depending upon application requirement.

4. **Forth step**

The fourth step is the partitioning the application using hardware (HW)-software (SW) partitioning methods and algorithms; here we may use best existing HW-SW partitioning methods and algorithms for our application with two level one level for basic partitioning that is HW-SW tasks and other level is reconfigurable logic block (RLB)-fixed kind of HW (F-HW) partitioning.

5. **Fifth step**

The fifth step is the design and implementation of the optimized algorithms for mapping of the design library on to the proposed reconfigurable architecture.

6. **Sixth step**

The sixth step is the design and implementation of optimized algorithms for scheduling the reconfigurable tasks (RTs), which will be implemented in RLB before mapping the design on to the reconfigurable architecture (RA).

7. **Seventh step**

The seventh step is the implementation of prototype of the complete system; this involves integration of the entire modules using designed algorithms for scheduling of RTs and mapping of these RTs on to the proposed RA for given application.

2.4.6 Intelligent Reconfigurable Instruction Set Processor (I-RISP) Design [38]

This Thesis presents the design alternatives for Reconfigurable Instruction Set Processor (RISP) from hardware point of view. Reconfigurable Instruction Set Processors are programmable processors that contain reconfigurable logic in one or more of their processing units. In RISP the most important aspect is the re-configurability. Reconfiguration time will have to be minimized in order to obtain satisfactory performance. The solution to this problem is in software tools or to design such a hardware which minimizes the configuration overheads. In order to avoid excessive reconfiguration, the I-RISP (Intelligent Reconfigurable Instruction Set Processor) design has introduced an ICAU (Intelligent Configuration Analyzer Unit) using hardware approach. ICAU intelligently analyzes the expected configurations and reuses the existing resources (configuration). The ability to reuse the existing resources significantly increases the performance of I-RISP.

The proposed processor is based on VLIW architecture, having an Intelligent Reconfiguration Analyzer Unit. The purpose of Intelligent Reconfiguration Analyzer Unit is to minimize re-configuration overheads faced by RISPs. In existing systems the intelligence was created by using software techniques. This software based intelligent reconfiguration techniques are considerably slow speed due to the conventional instruction cycle concept. In order to eliminate this software based overhead the intelligence has been incorporated in hardware rather then in software.
The detailed architecture of the proposed design consists of the following modules:

1. Scheduler
2. Fetch Unit (FU)
3. VLIW (Very Long Instruction Word)
4. Intelligent Computational Unit
5. Intelligent Configuration Analyzer Unit (ICAU)
6. Configuration Unit (CU)
7. Reconfigurable Processing Units (RPUs)

Chapter No. 3
Dedicated Machines and Reconfigurable Computing

Chapter No.3

Dedicated Machines and Reconfigurable Computing

3.1 The Philosophy of a dedicated machine

One of the main issues in the evolution of computing architectures is their *specialization*. Many reasons can lead hardware designers towards pursuing a specialized computing architecture. The requirement of obtaining the high performances in a typical or particular application domain is one reason for it. Also, the timing issues in real-time embedded applications pushed the designers towards architecturing application-specific solutions which could more aggressively meet these requirements. Similarly other very important parameters include the cost and the power consumptions. These are the key design factors in the domain of embedded systems.

All these reasons and constraints introduce the requirements for a new design paradigm which takes these features into account and leads to the definition of new specialized cores or architectures. Hence one option is the general purpose computers, where the main issue is that of achieving more generally high performance in a very large spectrum of applications. The second option is the application-specific computers, where embedded applications or algorithms guide architectural design in more compact form. Dedicated or application specific architecture, performances are generally not measured on conventional benchmarks but they are measured only on the application the machine is designed for.

A large no of the different approaches have been taken by architectural researchers to guide the design of application-specific solutions or algorithms. One of the early approaches proposed for the design of this kind of machines was the ASIP (Applications Specific Instruction set Processor) philosophy. It was an emerging design paradigm in the field of application specific computing. ASIPs are programmable processors where the Instruction Set is adopted to a particular application.

3.1.1 The main theme behind Instruction Set Specialization

The elements of the Instruction Set of a micro-processor, i.e. the op-codes are the *bricks* into which a high level code is broken down for execution on a micro-processor. The operation specified by each op-code is commonly executed on a most dedicated and highly optimized functional unit and therefore we can see the functions specified by each of the op-codes as the *hardware execution bricks* of the software execution flow.

If these bricks increase in granularity or size by performing more complex operations than those which are typically available in the instruction set of RISC micro-processors, then the instructions in the code will more generally correspond to more complex functional units, possibly characterized by more latency and performing 'larger' computations when compared to RISC functional units. Since we know that the hardware execution speed is much faster than the software execution speed hence an application broken down into bigger bricks will in generally be faster to execute than one broken down into smaller ones. However, no doughty that the simple operations are common to many software applications, the more complex operations become the less likely it is that they are common to a large number of applications. Therefore, while a more complex functional unit will speed up execution of the algorithm when extensively used by a typical application, it is also true that the area dedicated to such a type of the unit would result wasted, when running those applications programs that do not exhibit such complex operations in their program code. Moreover, the presence of complex instructions mostly complicates some program code generation tasks such as code selection and register allocation among others.

The ISA specialization has been studied deeply in the past few decades [39]. In a library of re-current sub-graph patterns is generated. Pattern matching and graphs covering is then performed. The problem of optimizing area of functional units through their specialization is also deeply considered and it is observed that much of the ALU functionality is not used when only one or a few embedded system applications are considered for execution. The embedded system application code is analyzed so as to discover that which functionalities of the ALU are actually required. Functional Units having lower cost and area can then be designed and specialized to the application analyzed.

3.2 FPGAs

A Field Programmable Gate Array is consisting of an array of combinational logic clouds or blocks overlaid with an interconnection network of horizontal and vertical wires. Both the combinational logic blocks and the interconnection network are configurable or programmable [40]. Their configurability is achieved by using either anti-fuse elements based technology or SRAM based memory bits to control the configurations or programming of the transistors. The Anti-fuse based technology uses a strong electric or electronic charges or currents in order to create a programming connection between the two adjacent required terminals. Hence in this way in fact this is a typically less reprogrammable system. Static RAM based program configuration can be re-programmed unlimited number of times on the fly by simply downloading a set of different configuration bit streams into the Static RAM based memory cells.

Typically a configurable logic block shortly known as CLB architecture in fact is consisting of a look-up table shortly known as LUT, a Delay flip-flop shortly known as D-FF, some other form of additional combinational logic circuitry and a also consisting of a set of Static RAM based memory cells to control the process of configuration of the laid down configurable logic blocks (See Figure 3.1) [41]. The laid down digital logic circuitry or blocks of the FPGA device also perform the Input/Output operations in order to load and store the data streams being required for the processing. On the other hand the horizontal and vertical interconnecting networks can also be reconfigured by programming or changing the connections between the laid down configurable logic blocks and the set of wires and by configuring the integrated programmable switch boxes shortly known as PSB, which connect different horizontal and vertical wires. These programmable switch boxes (PSB) for the interconnecting networks are also controlled or programmed by the Static RAM based memory cells. In this way the logical functions being computed inside the available configurable logic block (CLBs), the horizontal and vertical interconnecting networks and the Input/Output blocks can be configured and reconfigured by using the external data streams. Field Programmable Gate Arrays typically allow the unlimited number of reconfigurations for the laid down device. These versatile kind of programmable devices so far have been used to build even large scale micro-processor cores and co-processor cores whose internal architectures and as well as interconnections can be reconfigured in order to match the requirements of a hand on running application. In order to a very brief reconfigurable computing architectural survey of Field Programmable Gate Arrays and some other important improvements of the concerned technology consider the references of [3], [7], [13].

Current and future generations of reconfigurable computing systems or devices have ameliorated the reconfiguration and configuration costs by providing a typically high speed and most optimized partial and dynamic reconfigurability process [42]. In the process of partial reconfiguration of the under laid device [23], it is quite possible to up date or change the configuration bit streams of any one part of the working under laid device while the at the same time the configurations of the remaining part is still retained in its original form. On the other hand in the process of dynamic reconfiguration of the under laid device, the under laid devices allow this partial reconfiguration process even during the interval when even other configurable logic blocks are performing their regular operations or computations [43].

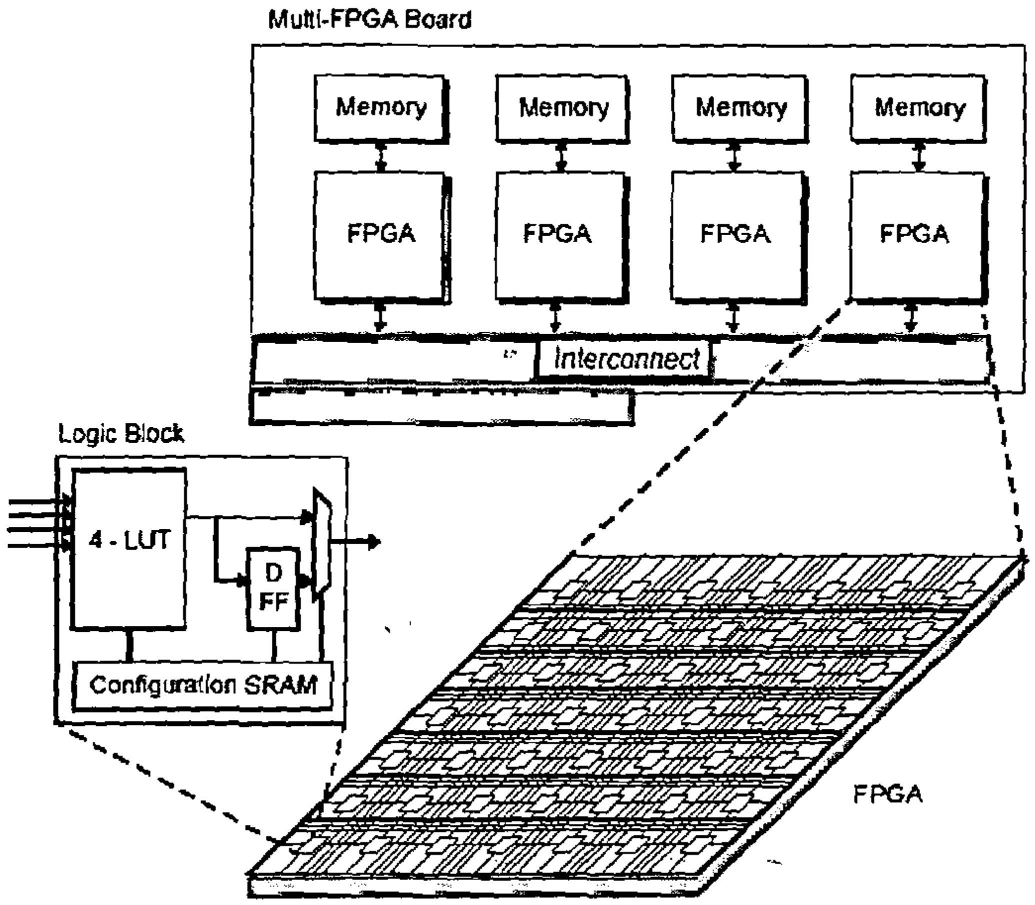


Figure 2 Typical FPGA Board, Device and Logic block architecture

Typically, the requirements of the applications are increasing at a rate which is much faster than the increasing rate in the size or density of the computational logic resources mostly available on the most of Field Programmable Gate Arrays. Field Programmable Gate Arrays architectures have limitations on their Input/Output capability of processing due to the limitation on the available total number of Input/Output (IO) pins which are physically available on the under laid device. In order to map and rout such kind of large scale user applications onto the available configurable logic blocks, the different computing systems have been designed which have several Field Programmable Gate Arrays some times on a single board or even some time on a single chip.

These board level reconfigurable computing architectures are mostly designed to function under the supervision an external configuration controller or kind of configuration supervisor or some times even they may use one of the on-board Field Programmable Gate Arrays as a main controller. There are a large number of such systems available. The examples of such systems may include the DECPeRLe board, and SPLASH-2 [30], the TERAMAC and the WILD series of devices being provided from Annapolis Micro-systems. Also some other sort of software tools exist which have the ability to automatically partition the whole design between the physically available multiple Field Programmable Gate Arrays on a single board by using the kind of higher-level of programming abstractions,[27].

3.2.1 The Basic FPGA Architecture

The basic architecture of Field Programmable Gate Arrays consists of broadly three kinds of components. These mentioned components include configurable logic blocks, programmable routing resources and a set of input/output logic blocks or IO blocks [44].

Generally, Field Programmable Gate Arrays consist of an array of programmable logic blocks known as CLBs that can be interconnected to other CLBs and also as well as to the programmable Input/Output blocks of the system through some kind of reprogrammable routing resources or architecture. Figure 3.2 provides a very simplified block diagram of the internal architecture of a generic Field Programmable Gate Array.

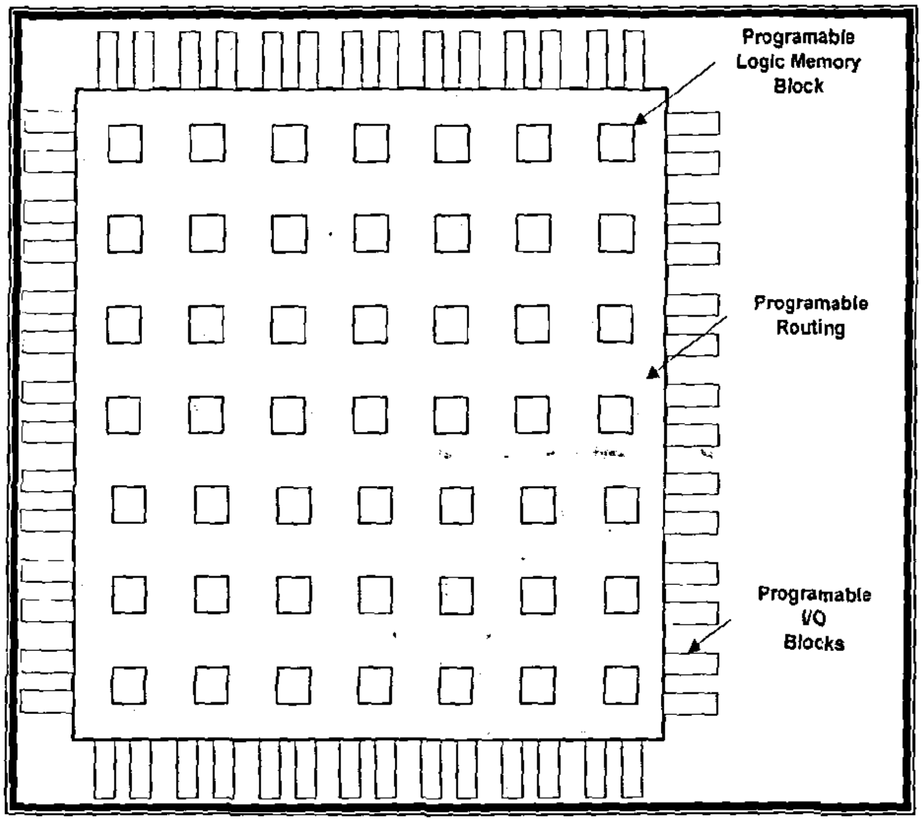


Figure 3 A Generic FPGA Architecture

i. The Programmable Logic

Field Programmable Gate Arrays designers have designed a large number of a variety of programmable logic architectures for Field Programmable Gate Arrays after their great invention in the mid-1980-1990. Since from few decades the much of the programmable logic structures being used in Field Programmable Gate Arrays can be optimally generalized as shown in Figure 3.3. The fundamental programmable logic element being integrated inside the FPGA generally consists of some kind of programmable combinational logic or CLB, a delay type flip-flop D-FF or kind of latching mechanism, and a kind of fast carry control logic to reduce the area density and typical delay costs for implementing such kind of carry logic.

Unlike other generic configurable logic component or element, the currently available commercial Field Programmable Gate Arrays devices provide a large number of programming flexibility within the available logic element. For example, a delay flip-flop D-FF in many commercially available Field Programmable Gate Arrays can be made to operate as a simple latch circuit and can be programmed to have many combinations of asynchronous as well as synchronous sets / resets and can be negatively- edge triggered or positively-edge triggered.

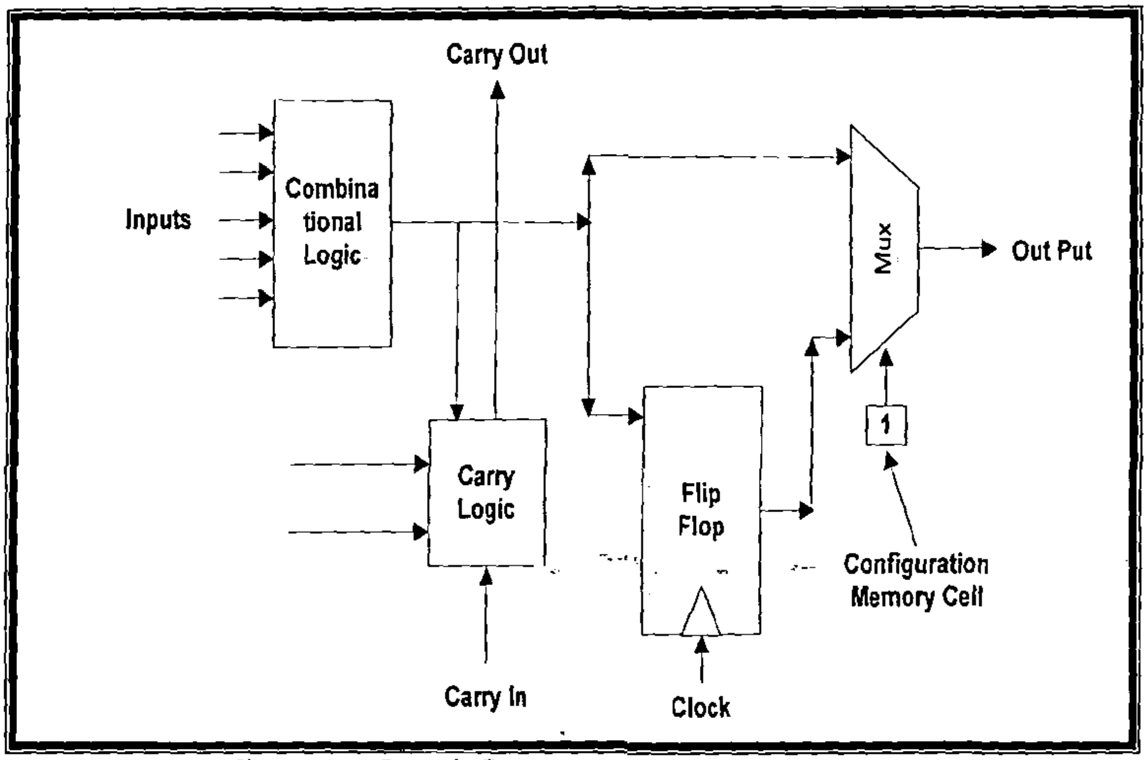


Figure 4 A Generic Programmable Logic Block

Although the most of the reprogrammable Field Programmable Gate Arrays use Look-Up Tables for their combinational logic, several other architectures like [12], [13], [14]) have been found to be used combinations of multiplexers and digital logic design gates to implement this programmable logic architecture or structures [45].

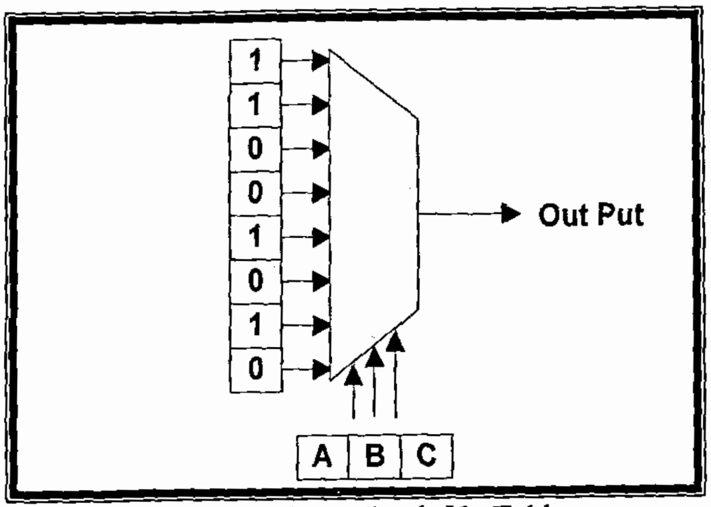


Figure 5 Three-Input Look-Up Table

Device Name	Year	LUT Width	Cluster Name	Cluster Size
Xilinx XC2000	1985	4	CLB	1
Xilinx XC3000	1987	4	CLB	2
Xilinx XC4000	1990	3 & 4	CLB	1 & 2
Altera Flex 8000	1992	4	LAB	8
Altera Flex 10K	1995	4	LAB	8
Xilinx Vertex	1998	4	CLB	4
Altera Apex 20K	1998	4	LAB	10
Xilinx Vertex II	2000	4	CLB	8
Altera Apex II	2001	4	LAB	10
Altera Stratix	2002	4	LAB	10
Xilinx Vertex 4	2004	4	CLB	8
Altera Stratix II	2004	3 & 4	LAB	24 & 16

Table 1 Logic Component Clustering Sizes of LUT Based FPGAs

ii. Programmable Routing Resources

The Field Programmable Gate Arrays designers have used a large class of different routing resources or structures within their Field Programmable Gate Arrays. Different kinds of forms or designs of routing resources exist through out the designs of each Field Programmable Gate Array. Commonly some amount of routing resources is also included within the design of the each logic clustering element so that the laid down logic elements can be combined to form bigger and more complex functions.

In order to implement the nature of the programmable routing of the resources, there are three basic switch types that have been used. These types include the digital multiplexers, the pass transistor circuits and a commonly used tri-state buffer gates. Figure 3.5 describes each of these mentioned switches with a Static RAM memory cells controlling their outputs. Commonly, the passing transistor circuits and the digital multiplexers have been used within the area of a logic cluster in order to connect the logic elements or components together while all of the above three have been used for the more global routing structure.

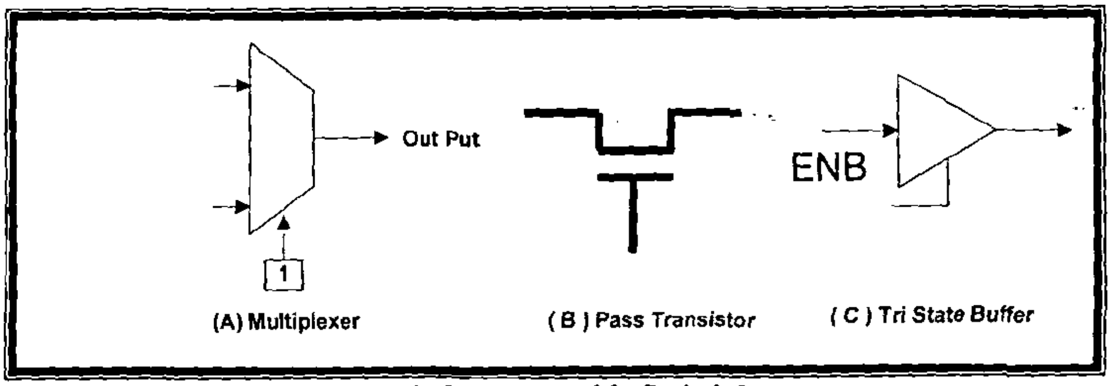


Figure 6 Basic Programmable Switch Types

iii. Programmable Input/Output Architectures

Unlike programmable logic resources and routing resources, the basic input/output resources or IO architecture, as is shown in Figure 3.6, is very similar among the different Field Programmable Gate Array families being evolved so far. The Input/Output logic blocks have been found with the tri-state buffer gates for the outputs and input buffer gates for the inputs of the system. The tri-state logic enable / disable signal, the output logic control signal and the input logic control signals can be individually latched or registered by using flip-flops within the Input/Output blocks or can be programmed as the un-registered being depending on the fact that how the I/O block is being programmed.

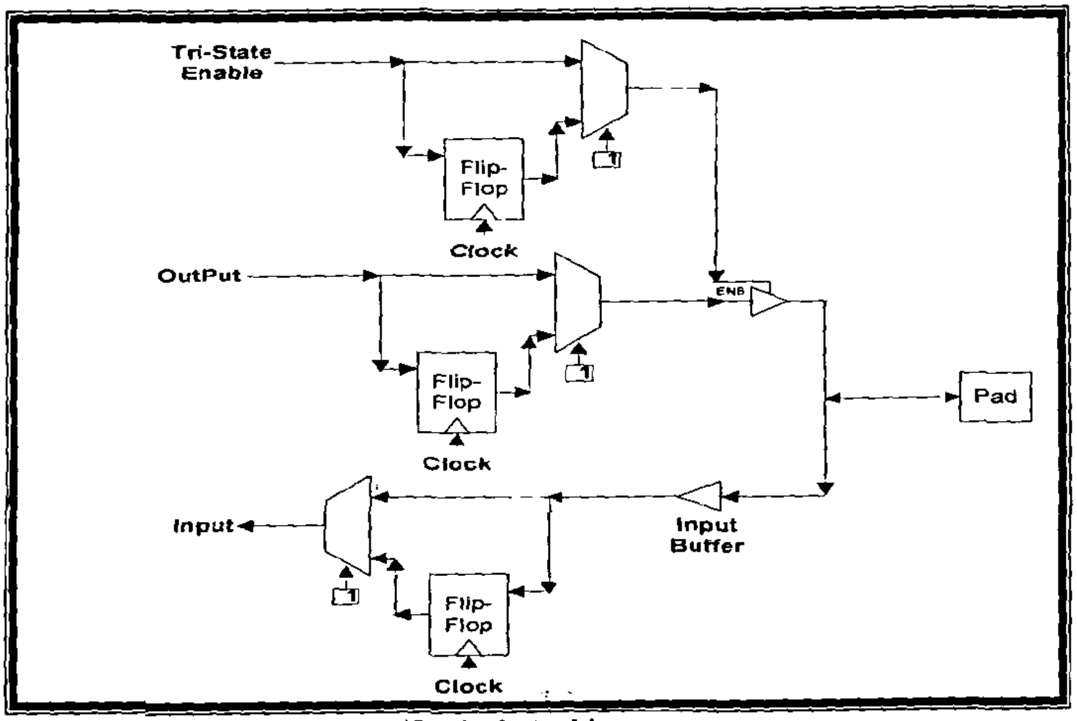


Figure 7 I/O Block Architecture

3.2.2 FPGA technology

Field Programmable Gate Arrays can be traditionally divided into two main categories:

1. Anti-fuse Based FPGAs

2. SRAM Based FPGAs

Recently another FPGA technology has been introduced by Alcatel, which is based on Flash/CMOS based circuits [72].

The first type or category of FPGAs uses anti-fuses as a mean to program the device. Anti-fuses implemented in a CMOS technology are being initially open circuits and after that once they are programmed they take on a low resistance. The main characteristic of this kind of FPGA is the fact that it can be programmed *only once*.

The second category of FPGAs uses SRAM cells as a mean for programming the device. A 1-bit SRAM can control FPGA switches in two different ways including either by controlling the gate node of a pass transistor or by controlling the select line of a multiplexer drives the inputs of logic blocks.

The main advantage of SRAM-based FPGAs lays in their nature of *re-programmability*. The logic value of the SRAM cells can be overwritten or updated for a number of times and hence allowing the FPGA to be reconfigured even *on-the-fly*. Another characteristic of such kind of FPGAs is the fact of being volatile, i.e. the configuration must be loaded onto the device every time the system is booted up or powered-up. The interest of this thesis for SRAM-based FPGA is indeed due to re-programmability nature of it on-the-fly: as it will be seen, this allows change of the IS of the proposed architecture while the application is running.

Further FPGA technology features of interest for this thesis include the latest techniques of partial reconfiguration or re-programmability and context switching. The former is the capability of re-programming only a specific part of the whole device, while the rest part of the device is remains operational. The latter feature called context-switching of the configuration is a new technology which enables an FPGA to hold contexts of the multiple configurations at the same time. Configurations are stored in a series of memory blocks or memory banks so that it is possible to rapidly switch between them within the delay of nanoseconds [36].

FPGA Mapping Tools

Implementation of a circuit onto an FPGA platform requires sophisticated CAD tools. Hardware description language or a schematic description is used to enter the design. In the process of transforming such descriptions into the FPGA configuration, there are three main phases.

1. *Mapping phase*
2. *Placement phase*
3. *Routing phase*

In the mapping phase the circuit is logically partitioned into modules or parts so that each one is assigned to one configurable logic block of the FPGA. The second phase, called placement, is the process of assigning the computation of every configurable logic cell generated in the mapping phase to a *physical* logic cell being available on the device. The third phase, called routing, targets at defining all the connections among cells through programming the available horizontal and vertical switches. The complexity of the CAD tools is very high and the three phases can take much more time from few seconds to few hours for large circuits.

3.2.4 Commercial FPGAs

Xilinx:

Xilinx FPGA devices consist of a two dimensional array of configurable logic cells connected by horizontal and vertical layers of wires. The most widely used FPGA include the generations of Xilinx devices XC4000 [45] which claimed the density or capacity from 2K to 15K equivalent gates.

In the XC4000 FPGA device Configurable Logic Block (CLB) is based on lookup tables. A lookup table is in fact an array of 1-bit memory cells, where inputs are the address of memory lines, and the one bit output is the data line: thus, a lookup table with K inputs will have 2^k-1 bit possible memory cells. A lookup table can implement any k-inputs logic function or computation. A CLB mostly has more than one lookup tables, and one or more than one flip-flops. The XC4000 CLB consists of 3 lookup tables, two of them with 4-inputs and one with 3-inputs and two flip-flops. One of the circuits that a CLB can implement is a full adder, so that CLB can be configured or programmed to implement any fast arithmetic circuit as carry-save or carry-look-ahead adders. CLBs can also be used as read/write SRAM cells. A CLB is programmed by setting the memory cells to the values given by the truth table of the digital logic circuit to be implemented.

The second distinguishing feature of FPGAs is their routing connections or interconnects. The XC4000 features horizontal and vertical channels. Each of these channels consists in short, long, and medium distances wires. Short distance wires are used for interconnecting two close CLBs, while the long distance wires can connect far CLBs. In fact long distance wires tend to have much less switches than shorter wires. The final delay of the circuit depends heavily on how the CAD tool has assigned wire segments to physical wires after the process of routing.

Altera FLEX:

The Altera Flex 8000 series of FPGAs has a logic density or capacity of 4K to more than 15K equivalent logic gates. The device consists of a 3-level hierarchy. The lowest level is a set of lookup tables. The basic logic block, called configurable logic element, consists of a 4-input lookup table, a flip-flop and some additional circuitry for fast carry propagation purposes. At a highest level, is the Logic Array Block? This consists of eight logic elements, connected together through a local interconnect. Any two logic elements of the block can be connected to each other by programmable wires. The local interconnect and every logic element are, connected to the Fast Track global interconnects, similar to the XC4000 long distance wires.

The Flex 10000 family of FPGA features a different characteristic to that of 8000 series of family. Every row of the device contains an embedded array block which can be configured either as an SRAM memory cell block or as a lookup table. The latter use serves to implement any complex digital logic circuit through a multi-output lookup table. To more exploit this feature Flex tools contain various macro-functions to be implemented in embedded logic blocks.

Atmel:

The AT40K and AT6000 FPGAs family or series present the particular feature of *partial configurability*. This means that specific parts of the an FPGA device can be reconfigured while the rest continues to operate without disruption. This is particularly useful in reconfigurable systems where instructions are taken in and out of th programmable functional unit. The AT6000 logic cell contains a D-type register and about twelve logic gates. Interconnect is peculiar in that it provides diagonal connection as well, in addition to the usual vertical and horizontal programmable interconnections. Every cell is therefore *octagonal and hence* it can be connected with eight neighbors. The Atmel devices provide an internal SRAM memory that can be used for caching configurations. A more advanced context-switching FPGA device is currently under development at NEC. It is claimed that the area required to store multiple contexts does not grow linearly with the number of context [12]. In the future, the usage of even DRAM cells instead of SRAM cells to save FPGA contexts could increase even more FPGA potentialities.

3.2.5 FPGA performances

The FPGA performance in terms of execution speed is affected by two main features. The time needed to download a configuration describing a certain circuit and the time needed to execute the function implemented by loaded circuit. Another important issue is the time the software tool takes to generate configurations.

This is not as easier to give these performance figures in a straight forward way. The time overhead for reconfiguration varies considerably depending on the size of the device. Obviously a device implementing small logic functions will take less time to reconfigure because a smaller number of reconfiguration bits streams are involved. A good measure of reconfiguration time overhead could therefore be given *per configurable logic block used* or even *per gate equivalent* which is even better because it is not affected by the block granularity of the FPGA device. Another important issue is the frequency at which the device operates, which is directly proportional to reconfiguration time overhead. In the Atmel 6000 the reconfiguration for the full device takes from 1 to 8 mili seconds and this corresponds to almost 0.2 micro seconds for per Cell. Altera 10K claims 115 MHz performances with a density of up to 250 K logic gates. Xilinx has recently launched a new line of FPGAs called VIRTEX series which targets operation frequency of more than 300 MHz.

Table 3.1 gives performance of the Atmel6000 in terms of execution delay of some arithmetic circuits.

Circuit	**Cell Count**	**Max Speed**
16bit Ripple Carry Adder	64	8.9MHZ
16bit Fast Ripple Carry Adder	96	11.4MHZ
16bit Carry Select Adder	222	15.7MHZ

Table 2 Cell Count and maximum Operating speed (one operation per cycle) for some arithmetic circuits in the Atmel 6000 device

3.3 Reconfigurable CPUs

TH5942

Since FPGAs present both advantages of re-programmability as well as the high performance of custom circuit; it is appealing to combine a micro-processor core with reconfigurable resources in order to achieve a speed or performance improvement over either a separate micro-processor or a separate reconfigurable FPGA device [3]. While it is possible to combine conventional micro-processors with available reconfigurable FPGA devices at the circuit board level; the integration changes the I/O costs for both devices. An architectural solution which is definitely appealing is therefore the integration of an FPGA on-chip. A number of different solutions are as under:

i. *Totally reconfigurable processor.*
ii. *The co-processor approach.*
iii. *A partially reconfigurable CPU.*

3.3.1 Totally Reconfigurable Processors

This is a kind of the design approach where a CPU together with a reconfigurable accelerator is designed or implemented by means of a reconfigurable logic. An important project of this kind is the Dynamic Instruction Set Computer (DISC), designed at Brigham Young University [6]. DISC consisted of two CLAy31 FPGAs developed by National Semiconductors and memory on a circuit board connected to a personal micro computer. DISC made extensive use of the latest technique of partial reconfiguration. The first FPGA was consisting of a permanent control unit while the second was divided into rows to simplify array management and allow for custom-instruction spatial caching. Custom instructions were intended to be swapped into and out of the FPGA similar to that of the pages of virtual memory. Before initiating execution of a custom instruction the device operating program asks the FPGA for the presence of the custom-instruction configuration. If the custom instruction is currently really on the FPGA, then the execution is initiated otherwise program execution pauses while the custom instruction is configured on the FPGA.

3.3.2 The Co processor Approach

Such an approach in fact proposes a *loose* coupling between micro-processor core and the reconfigurable logic. The latter acts as a coprocessor where it is a slave computational unit located either on the same die as the processor or off the chip. The granularity of the implemented function in the reconfigurable section is much higher than that of the mixed CPU approach. This is due to the fact that the communication with a coprocessor is much slower than that of the communication with a parallel data-path of the CPU. Therefore the block of computation delivered to it must be large in order to give high performance gain or improvement. A research project that fits in this category of reconfigurable computing is the Garp by the BRASS group at Berkeley [4]. The proposed architecture in fact consists of a MIPS processor placed in the same die with a reconfigurable coprocessor. The coprocessor is activated by the processor when a reconfigurable function is called. At this point the processor turns off and the coprocessor carries out the computation of the reconfigurable instruction set having also access to both the processor core and cache memory. The coprocessor also includes a memory for caching configurations so that to allow a fast context switching.

3.3.3 A Partially Reconfigurable CPU

This kind of approach proposes a very close or tight coupling between the main processor core and the CPU. Contrarily to the coprocessor approach here the CPU and the reconfigurable logic both compute simultaneously. The reconfigurable module is indeed added to the data-paths of the core CPU and hence introducing a special kind of functional units called Reprogrammable Functional Units (RFUs). This kind of system organization allows definition of an extension of the architecture Instruction Set by the implementation of new instructions on the RFUs. Also since the FPGA can be run-time reprogrammed; one element of the instruction set can be mapped onto an RFU for the whole length of the application as well as such element can vary during the application execution time through reconfiguration of the FPGA section. A number of projects have used the mixed CPU architectures in the past. The PRISC project [35] developed at Harvard presented to extend the Instruction Set of a RISC processor through implementation of particular functions onto one or more Programmable Functional Units. A framework is proposed where the choice of the functions to be implemented in the PFUs is very transparent to the programmer. The most general computational model for a PFU is said to be a multi-cycle sequential state machine. Performance gains were measured on the SPECint92 benchmark suite and a speedup factor of 10% to 90% was reported. A second proposal of tightly coupled micro-processor core and a

Reconfigurable Logic is that of the OneChip project. *OneChip proposes an* architecture which is very close to that of PRISC. The major difference from the PRISC approach is that in this system any kind of function is allowed to be implemented in the PFU.

Chimaera [10], is a reconfigurable system that was developed at Northwestern University. In this system the FPGA and the processor core are placed in the same chip. It focuses on the optimization of the reconfiguration overhead and elimination of the *communication bottleneck between the reconfigurable logic and the micro-processor* core. By enhancing the speed of reconfigurations and communications, even fine grained reconfiguration can become practical. This project mainly focuses on the definition of the Chimaera architecture. A caching logic is also present in order to hold multiple configurations and minimize overhead. Reconfiguration is done on a per-raw basis and RFU functions occupy one or more of the rows.

Chapter No. 4
The Proposed Architecture

Chapter No.4
The Proposed Architecture

This chapter focuses on the design of a Very Long Instruction Word Reconfigurable Instruction set Processor (VLIW-RISP).Hardware design aspects and concerning issues have been discussed along with each component under consideration. In proposed system an alternative design for Reconfigurable Instruction Set Processor (RISP) has been proposed with the capability of the most optimal configuration overhead for Very Long Instruction Word (VLIW) based architectures. The processor has been integrated with the high speed partially reconfigurable Field Programmable Gate Array (FPGA) cores as its Reconfigurable Functional Units (RFUs) in place of ALUs and it treats instructions as removable modules which can be paged in and paged out through the partial reconfigurations according to the requirements of the application being under execution.

4.1 Overall Design Goal

The overall goal of the thesis was to develop VLIW based Reconfigurable Instruction Set Processor with a reconfigurable ALU that can implement any computational algorithm on fly and reconfigure it later on for some other algorithm. The processor was required to be the partially reconfigurable during the execution of the application. As such, the design has been divided into two main modules:

Module No.1 (Compiler Design for VLIW-RISP)

This module deals with the design of the compiler for the VLIW-RISP. The compiler is supposed to be able to allocate the Op-Codes to the instructions according to the available configuration streams in the configuration memory. In this thesis the compiler designing is not the main goal and hence a hypothetical compiler according to the requirements of the proposed RISP is considered for the instructions being used for execution purpose on the processor. Compiler is supposed to generate the instructions with the instruction format as is used in the designed processor.

Module No.2 (VLIW-RISP Design using Verilog-HDL)

This module deals with the design of the RISC based partially reconfigurable VLIW-RISP processor. In this phase the proposed VLIW-RISP and its programming interface is developed using the Verilog-HDL. Inside the proposed design only the computational units (ALUs) are supposed to be reconfigurable and the remaining all components of the design are truly non-reconfigurable.

4.2 Tool Issues

For designing the processor a hardware description language was required, Verilog HDL was chosen due to its popularity and simplicity.

4.2.1. Importance of HDLs

HDLs have many advantages compared to traditional schematic-based design. Designs can be described at very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.

By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gat-level netlist or physical layout is minimized. Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

4.2.2. Why not to use a general purpose language

General-purpose programming languages do not provide support for structure and instantiation of objects or modules. Also they do not support bit-level behavior description. Execution in general-purpose languages is sequential, therefore are unable to support the concurrent nature of hardware modules. Also, they do not provide the required timing support.

4.2.3. Verilog-HDL

Verilog HDL has evolved as a standard Hardware Description Language. Verilog HDL offers many useful features for hardware design. Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL. Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchal design. Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers. All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors. Verilog is really a language for modeling event driven systems. The design flow using Verilog-HDL or VHDL is shown in figure 4.1.

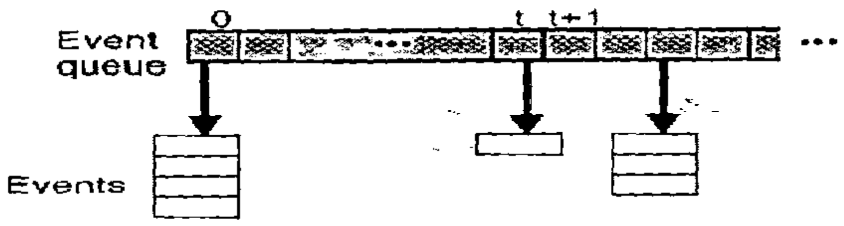


Figure 8 Event driven Systems

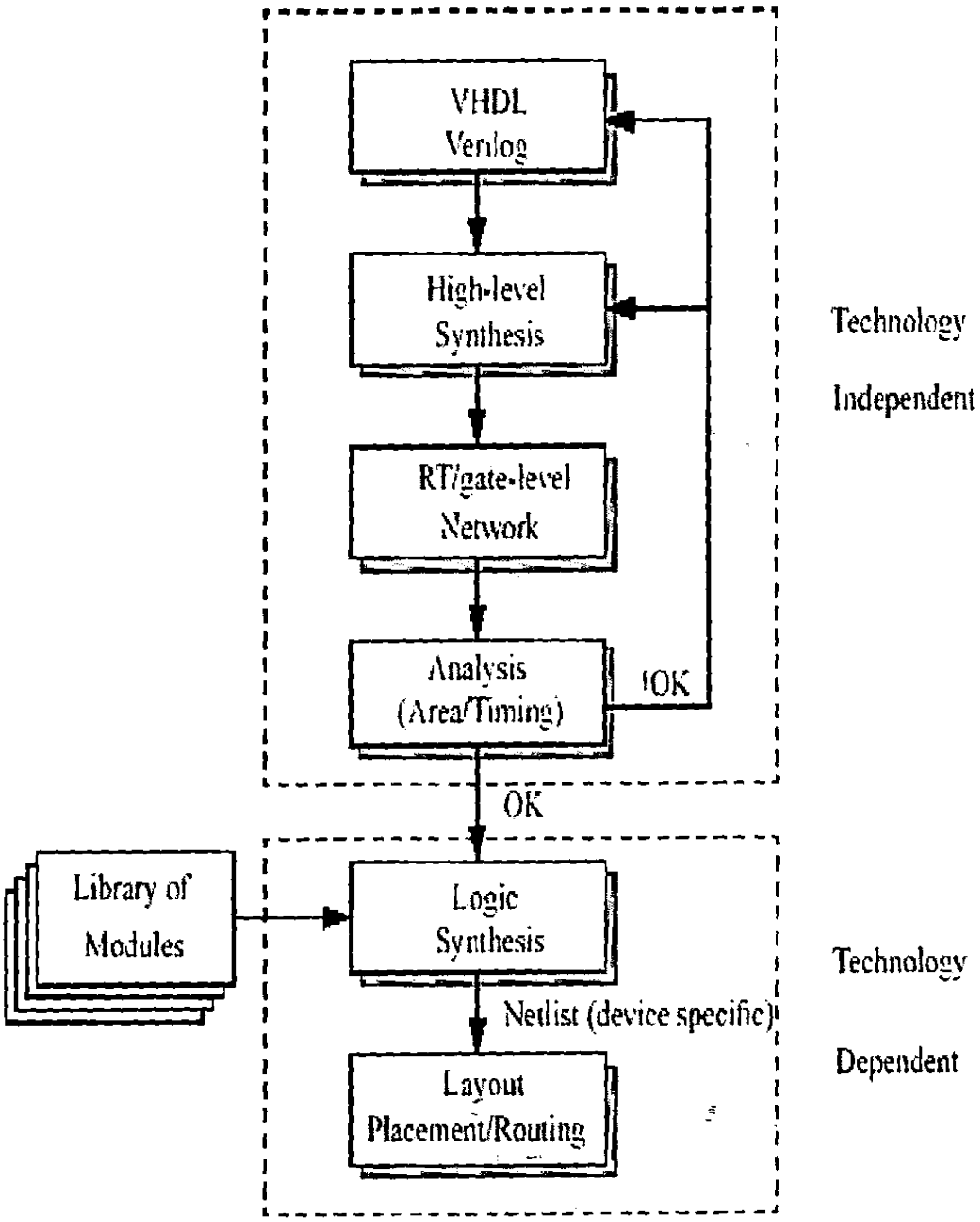


Figure 9 Design Flow Using Verilog-HDL

4.3 Instruction Format of proposed VLIW-RISP

The proposed Very Long Instruction Word Reconfigurable Instruction Set Processor VLIW-RISP is basically a reconfigurable RISC architecture having each instruction of size 32-bits. Instruction format is given below.

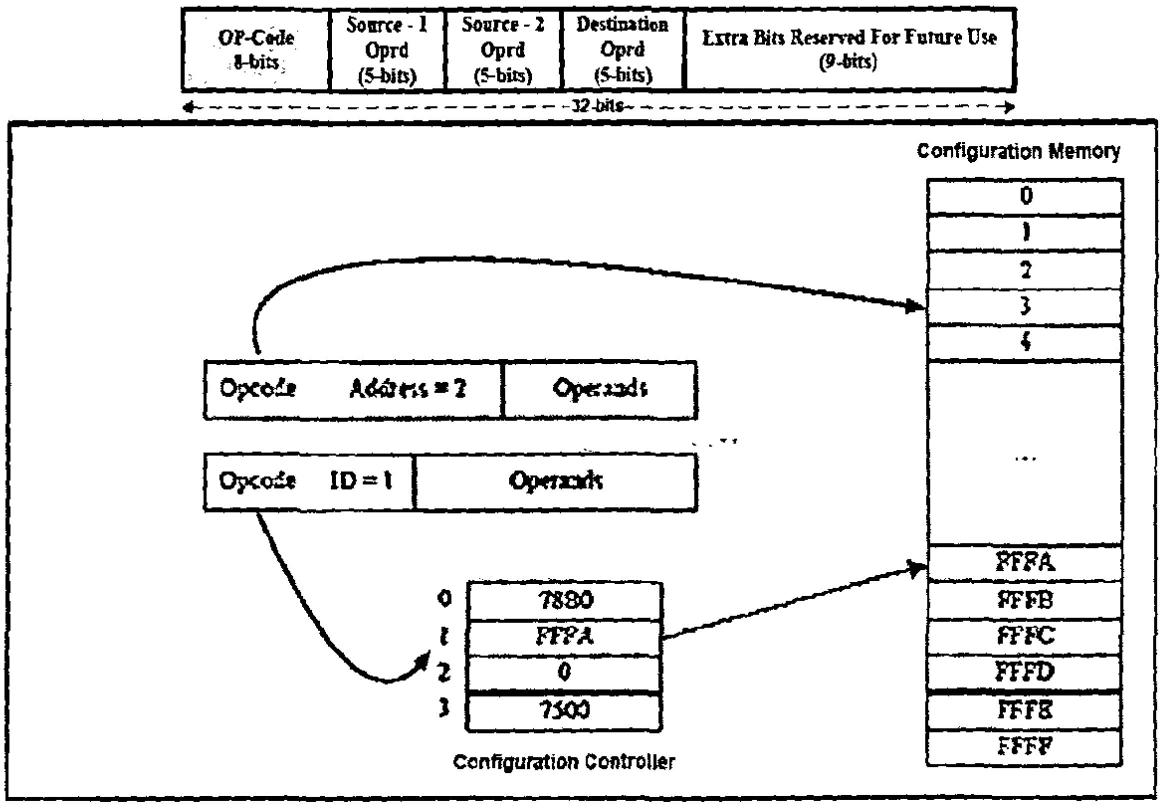


Figure 10 Op-Code Interpretation

This is the instruction format for the instructions of the application to be executed on the proposed VLIW-RISP. These instructions are of the size 32-bits. The 8-bits on the most significant side of the instructions represent to the operation code shortly known as the OP-CODE and hence leading to a maximum of the 256 possible operations or instructions in the instruction set being active (Configured) at any time. Each op-code of an instruction is in fact a pointer to some configuration block in the multi-port configuration memory as shown in figure 4.3 and hence is responsible for loading the required configuration stream of the relevant hardware module. Here each op-code is a relocatable pointer which can be reconfigured for some other hardware module by loading a new kind of bit stream over there in the configuration memory. Hence due to this relocatable nature of the op-codes, the instruction set of the proposed processor is bigger than the actual one supported by the design according to 8-bits of the op-codes.

Theoretically the reconfigurable instruction set processor defines to an unlimited sized instruction set due to relocatable nature of op-codes.

Because the proposed VLIW-RISP is basically a RISC architecture using only the Register-Register architecture with a register file having 32-Registers each register of the size 32-bits, hence in order to access each register for source or for the destination requires an address of size 5-bits. Also the instruction format is a three-address instruction format containing two addresses for sources and one address for the destination. Hence there are three addresses Source1, Source2 and Destination operand address, each of the size 5-bits hence consuming a total of 15-bits of the instruction. There are a total of the 9-bits being declared as the Un-Used Bits. These bits will be used in the future to further enhance the VLIW-RISP design and the instruction set.

4.4 VLIW-RISP Design Simulation using Verilog-HDL

In the simulation of the design the program will be written in the editor and will be compiled. The compiler after doing its all jobs will generate a binary file which will contain the instructions of the program in the binary form as are required by the designed VLIW-RISP. Hence now this file contains one instruction of size 32-bits in one row and so on. Then this file is given to the *"Stimulus of the VLIW-RISP"*. This stimulus loads this file into the instruction cache of the processor and then processor executes it as it is designed for. Before loading the application program written by the user, the required data operands are loaded into the data cache of the VLIW-RISP as are required by the proposed design. Then these data operands are loaded into the register file of the processor, containing a total of 24-General Purpose and 8-Flag Registers, where each one is a 32-bits register.

4.5 PROPOSED RISP DESIGN:

Very Long Instruction Word Reconfigurable Instruction Set processor VLIW-RISP design is divided into different modules that were interfaced together to make the whole processor. The different modules being designed for the VLIW-RISP using Verilog-HDL are followings:

1. Input/Output Interface **(IOI)**
2. Cache Memories
3. Pre-fetch Unit **(PFU)**
4. Instruction Scheduler Unit **(ISU)**
5. Instruction Pack Logic **(IPL)**
6. Computational Pipeline-1 **(CP-1)**

- VLIW Fetch Unit (**VFU**)
- VLIW Dispatch Unit (**VDU**)

7. Computational Pipeline-2 (**CP-2**)
   - VLIW Execution Unit (**VEU**)
   - VLIW Configuration Unit (**VCU**)
8. *Micro-programmed Control Unit* (***MCU***)

In this chapter the detailed architecture of the proposed RISP has been discussed with the detailed computational and control functionality explanations. The detailed architecture has been overviewed in a top-down hierarchy. The detailed architecture of the proposed processor is shown in the Fig.4.4 and different modules are discussed below along with their functionality.

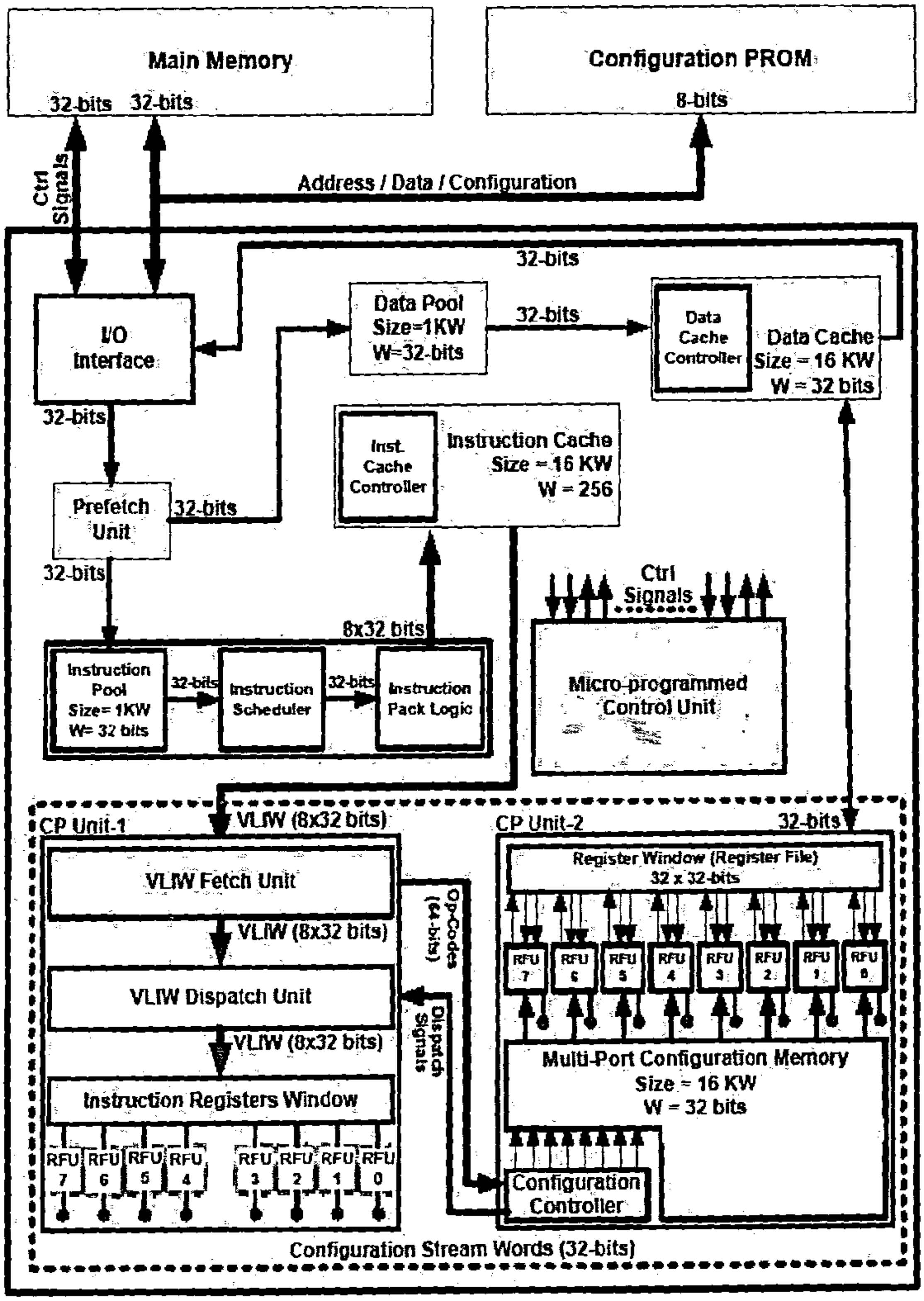


Figure 11 Proposed VLIW-RISP Design

4.5.1 Input / Output Interface (IO Interface):

The IO interface of RISP is used to communicate with the external devices being interfaced with it. The first job of the I/O Interface is to load the configuration streams from external Configuration EPROM or main memory of system during the booting processes of the processor and it takes only a few clock cycles. These configuration streams contain the different hardware modules like Adders, Subtractors, Multipliers and Shifters etc. The second job of the IO interface is to load the instructions and their relevant data operands to be executed on the processor. The third job of the IO interface is to store the results of the computations performed on the processor in main memory of the system. The fourth job of the IO interface is to send and receive the control signals generated and acknowledged by the control unit of the RISP to the external devices. I/O Interface interacts with external environment by using the following signals.

1. Data Bus Signals
2. Address Bus Signals
3. Control Bus Signals

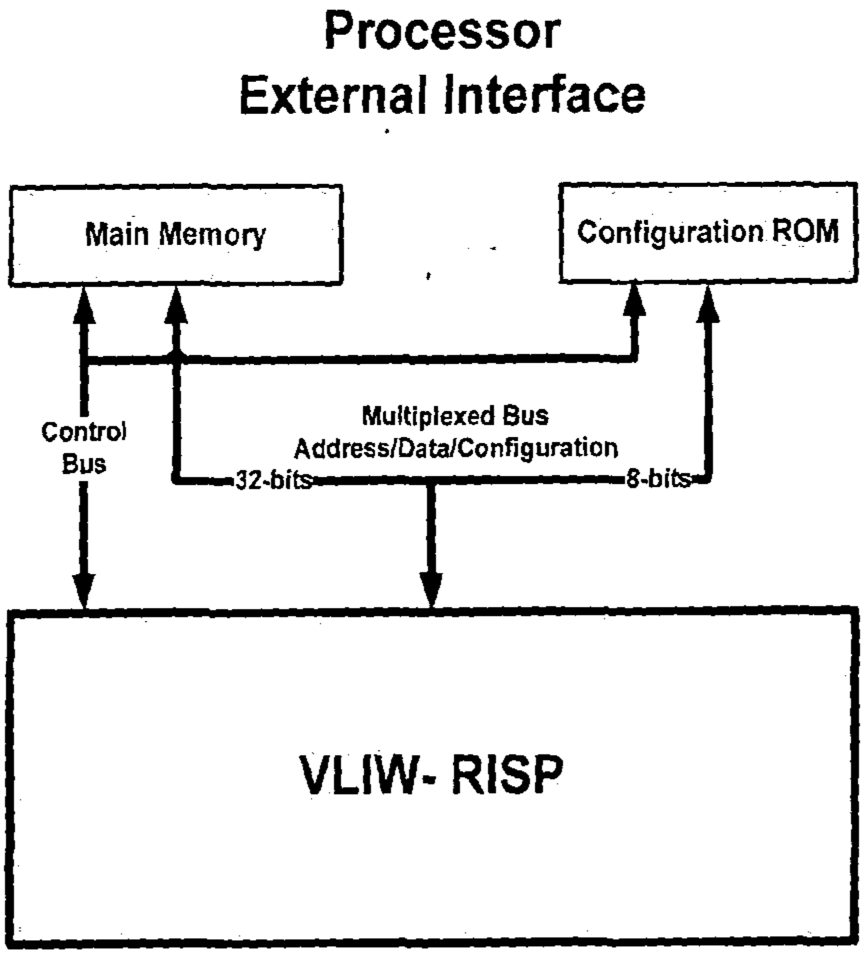


Figure 12 External Interface

The major functions and requirements for an **I/O** module fall into the following categories [28].

a. Control and timing

b. Processor Communication

c. Device Communication

d. Data Buffering

e. Error Detection

f. Processor Configuration

Cache Memories

The cache memory holds (stores) the data used by a program and also the instruction of the program. The cache is organized as set associative cache, with each location (line) containing 32-bits of data in case of Data Cache and 32 x 8-bits in case of Instruction Cache. The cache operates as a write through cache. Note that the cache changes only if a miss occurs. This means that data written to a memory location not already cached are not written to the cache. In many cases, much of the active portion of the program is found completely inside the cache memory. This causes the execution to occur at the rate of one clock cycle for many of the instructions that are commonly used in a program [29]. The architecture of the cache is supposed to be the standard cache architecture being used by the standard micro processors.

1- Write-Back Cache: When the system writes to a memory location that is currently held in cache, it only writes the new information to the appropriate cache line. When the cache line is eventually needed for some other memory address, the changed data is "written back" to system memory. This type of cache provides better performance than a write-through cache, because it saves on (time-consuming) write cycles to memory.

2- Write-Through Cache: When the system writes to a memory location that is currently held in cache, it writes the new information both to the appropriate cache line and the memory location itself at the same time. This type of caching provides worse performance than write-back, but is simpler to implement and has the advantage of internal consistency, because the cache is never out of synchronous with the memory the way it is with a write-back cache.

a. Instruction Cache

The user interface, along with the compiler, generates a program file containing the application program written by the user in the specified editor. Each row of the file contains a 32-bits instruction. This program file is loaded into the to internal instruction cache of the VLIW-RISP.

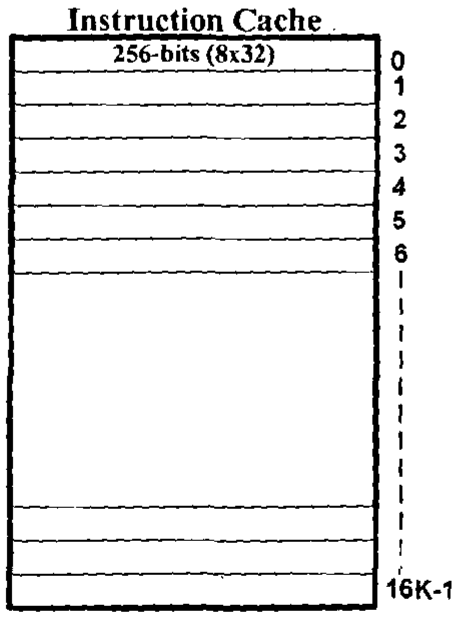


Figure 13 Instruction Cache

The internal instruction cache of the RISP is of the size 16KW. Where W=Memory Word. The size of the memory word is same as that of the size of the VLIW and is 32 x 8 = 256 bits.

b. Data Cache

The data operands given by the user for the registers of the VLIW-RISP through the interface are written to a data file initially. The each row of this data file is of the size 32-bits and contains a single data operand of some instruction. This data file is loaded into the internal data memory of the VLIW-RISP. The internal data memory of the VLIW-RISP is of the size 16KW, Where W=Memory Word. The size of the data memory word is 32-bits because the processor being designed is a 32-bits machine.

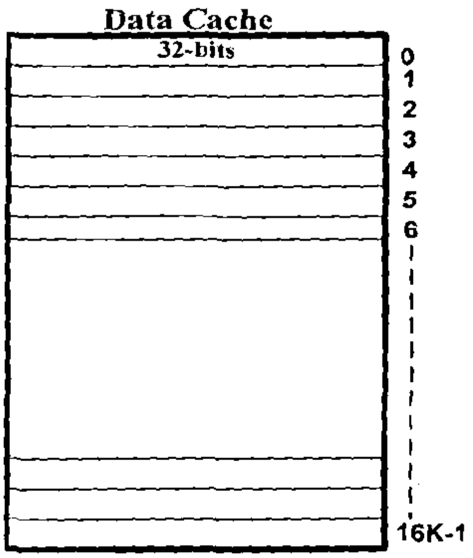


Figure 14 Data Cache

This internal data cache of the VLIW-RISP is used for many different jobs and these are as followings.

Initially the data cache is loaded with the configuration streams that are then being transferred into the multi-port configuration memory. Then the data cache is loaded with the data operands of the application program. These data operands are the source operands of the different instructions, written by the user in the program editor being developed. These data operands are then loaded into the register file of the processor so that the Register Window becomes able to fetch them during the execution of the relevant instructions. The last job of the internal data cache is to store the results being generated by the execution of the program instructions. These results are initially stored into the registers of the VLIW-RISP and later on these results are shifted into the internal data memory of the VLIW-RISP. From this data cache the results are stored into the external data memory (Data File) of the system from where the user interface receives and displays them on the system.

4.5.3 Pre-fetch Unit (PFU):

The basic job of the PFU is to fetch or pre-fetch the configuration stream or instruction stream and the data stream of the application program being under execution. Fetched configuration stream is loaded into the multi-port configuration memory and instructions are loaded in the *Instruction Pool* and then transferred into the *Instruction Cache*. Similarly the data stream is loaded into the *Data Pool* and then transferred into the *Data Cache*.

4.5.4 Instruction Scheduler Unit (ISU)

The ISU is the micro-programmed implementation of the *Tomasoulo's Algorithm* being used in VLIW and Super-scalar processors for the scheduling of the instructions. The instruction scheduler reads instructions from the instruction pool and then it analyzes them for dependencies (if any) and resolves these dependencies. Dependencies being analyzed include Data Dependency, Control Dependency, Resource Conflicts and Data Hazards etc. Then it after analysis ISU transfers these instructions to IPL.

4.5.5 Instruction Pack Logic (IPL)

The main job of the IPL is to pack the eight instructions in the form of a VLIW. The 32-bits instructions transferred from the ISU are given to the IPL. The IPL arranges these instructions in a buffer in a FIFO order on their arrival from the ISU. After the arrival of each instruction, the IPL increments its instruction counter and checks either there are eight instruction arrived from the ISU or not. If a total of eight instructions have been arrived from the ISU then the IPL transfers them into a VLIW buffer of size 8 x 32-bits. Then it enables this buffer to transfers this VLIW to instruction cache of the RISP if signal Load_VLIW =1. The same process is repeated constantly throughout the application execution. Consider the Fig. 4.8 of IPL.

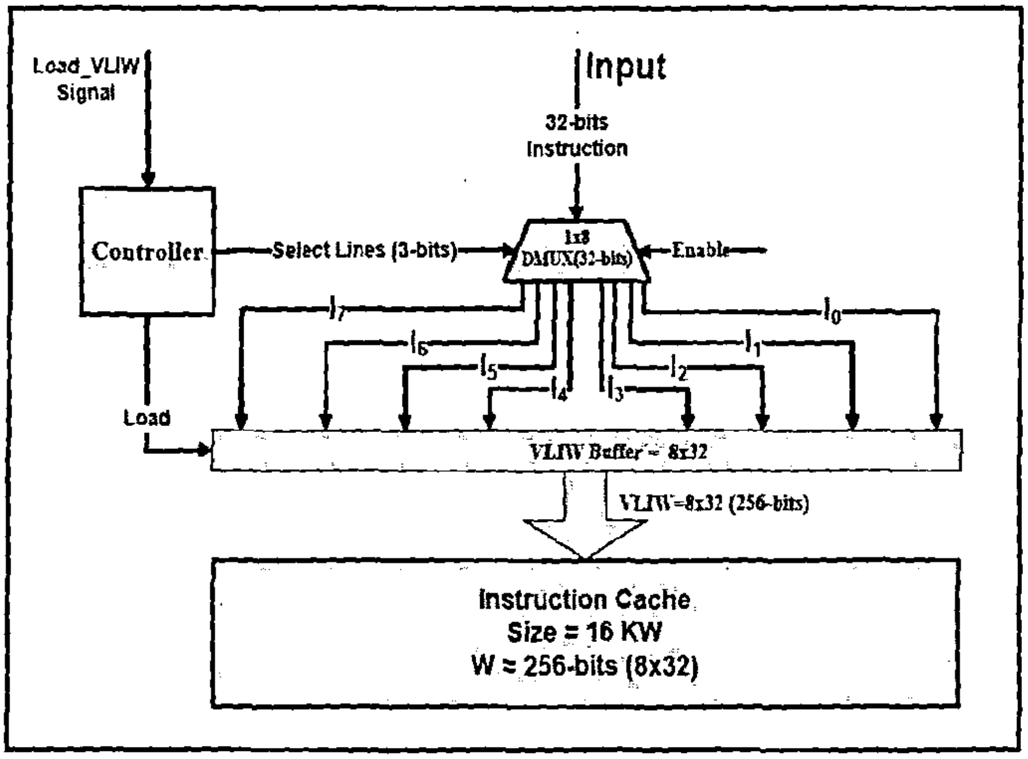


Figure 4.8 Instruction Pack Logic

4.5.6 Computational Pipeline-1 (CP-1)

CP-1 is consisting of a VLIW Fetch Unit (VFU) and a VLIW Dispatch Unit (VDU).

i. **VLIW Fetch Unit (VFU):**

VFU is a *State Machine* based unit and works like a *Programmable Counter*. VFU fetches VLIW from the instruction cache and the *Op-Codes* of all instructions of the VLIW are transferred to the *Configuration Unit* and the VLIW itself is transferred to VDU.

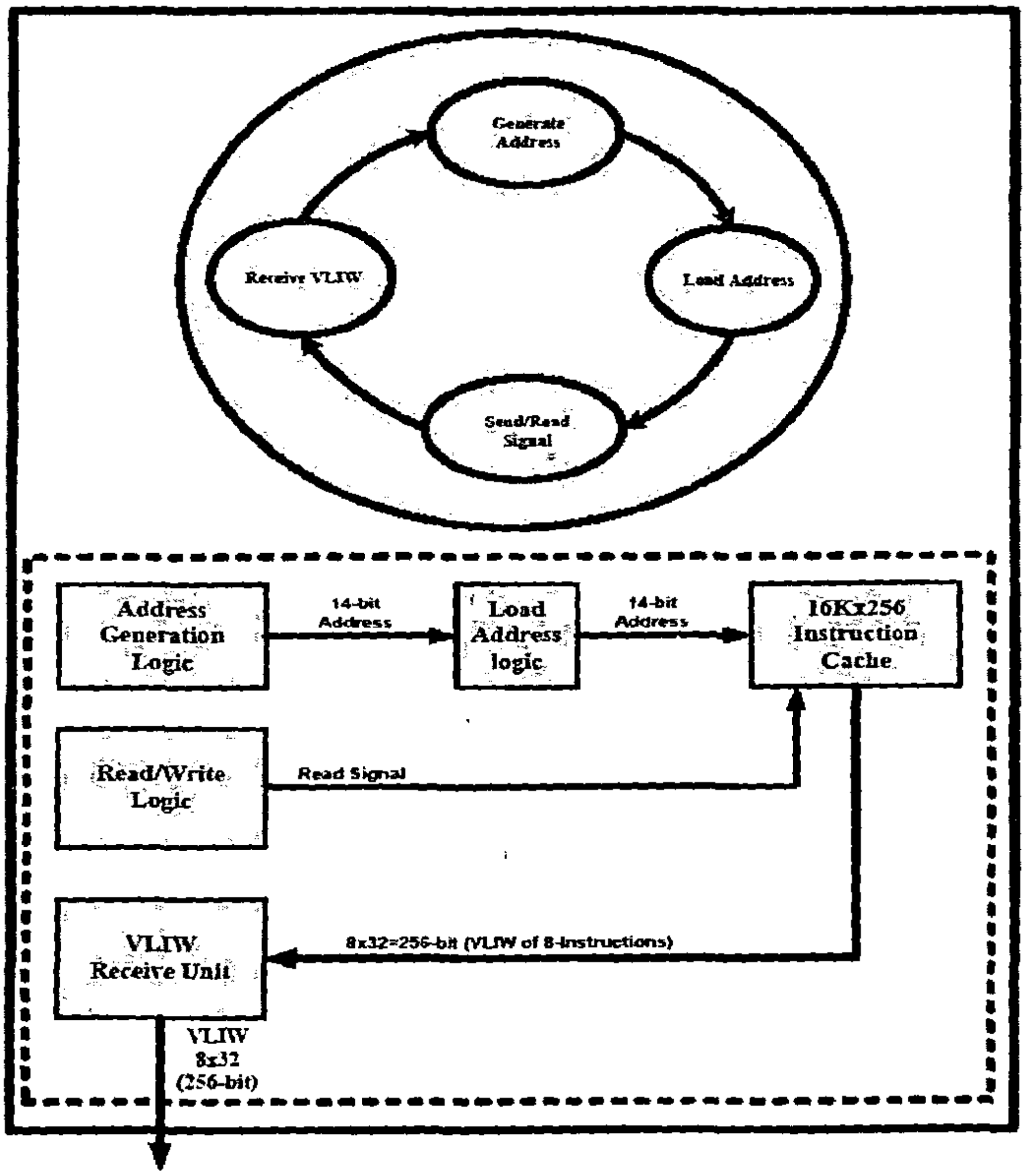


Figure 15 VLIW Fetch Unit

ii. **VLIW Dispatch Unit (VDU):**

VDU is consisting of an array of eight De-MUXs whose select lines are controlled by the configuration controller. According to the select lines activated by the configuration controller all of the instructions of VLIW are dispatched or issued by VDU to their relevant RFUs. Consider the Fig. 4.10 of VDU.

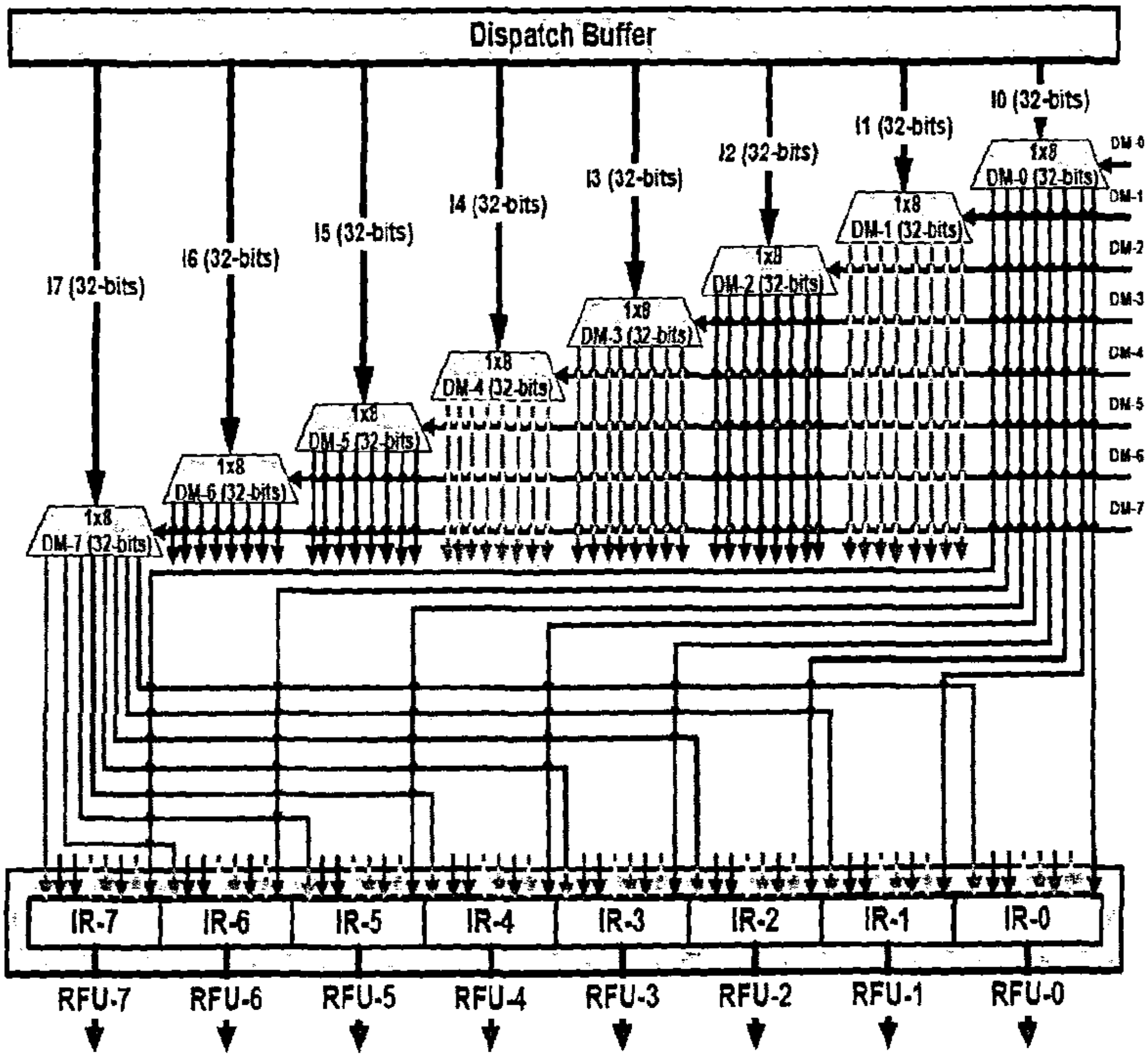


Figure 16 VLIW Dispatch Unit

4.5.7 Computational Pipeline-2 (CP-2)

The CP-2 is composed of a *VLIW Execution Unit (VEU)* which contains an array of eight RFUs and a Register Window of 32 registers (32-bits) and a *Configuration Unit* which contains a *Configuration Controller* and a *Multi-port Configuration Memory.*

i. VLIW Execution Unit (VEU):

VEU is the core component of the processor because it contains an array of RFUs being used for program execution. Consider the Fig. 4.11 of VEU. The VEU contains the following major modules.

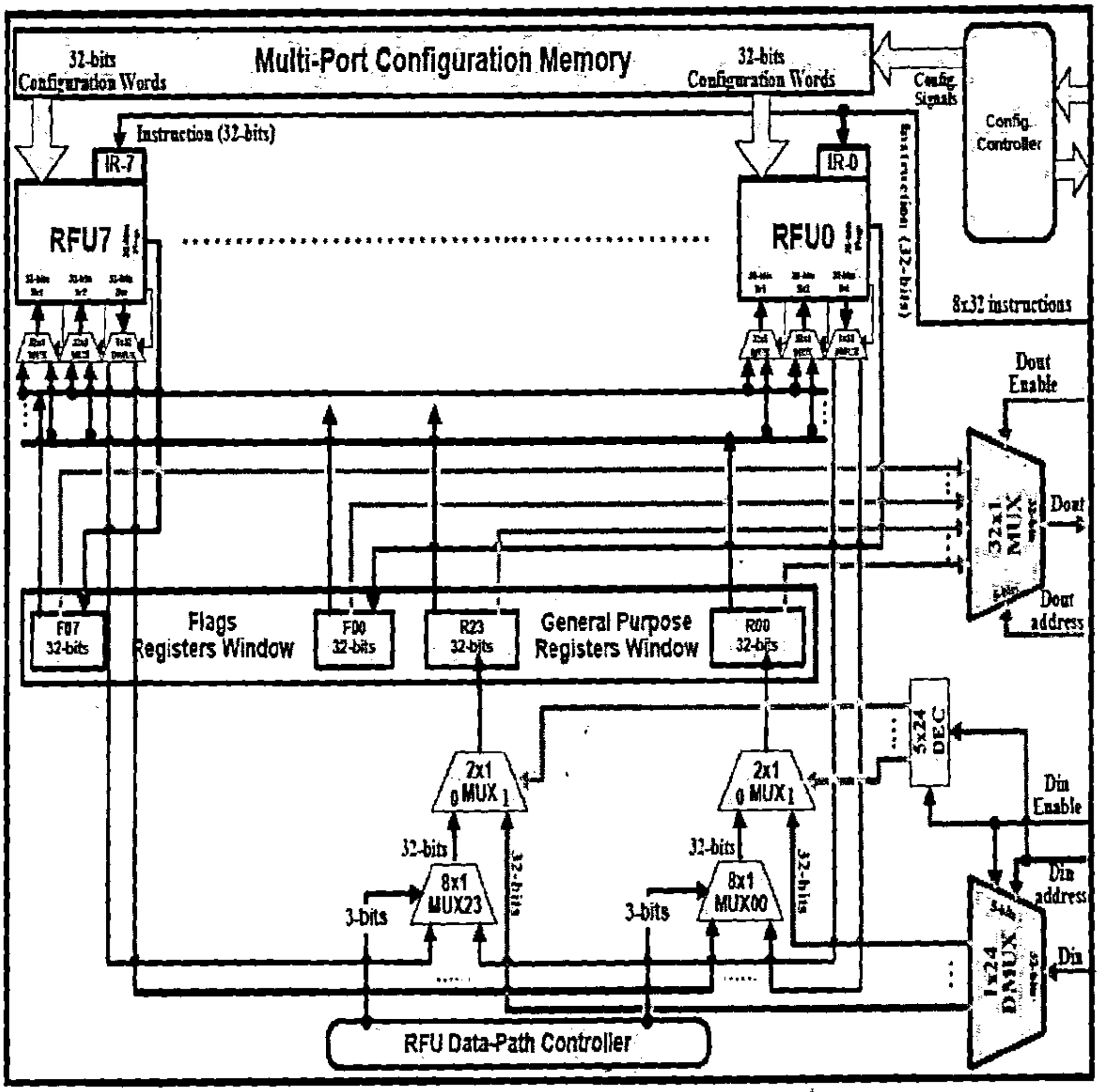


Figure 17 VLIW Execution Unit

a) External IO Logic (EIOL)

b) RFUs Data-in/Data-out Logic (RDIOL)

c) General-Purpose and Flag Registers (GFRs)

d) Registers Input/Output Logic (RIOL)

e) Reconfigurable Functional Units (RFUs)

f) Flags Generation Logic (FGL)

a) External IO Logic (EIOL)

The EIOL of the VEU is used to load instructions in the instruction register, source operands in general-purpose registers and the configuration stream in RFUs. The second job of the EIOL is to store the configuration stream being loaded in the RFUs for the analysis purpose and results being generated after the execution of VLIW.

The source operands Sr-1and Sr-2 are loaded into the internal general-purpose registers (GPRs) by the External De-MUX of size 1 x 24. The address given for the Data-in is connected to the select lines of De-MUX as well as to Decoder (5 x 24) input. De-MUX selects one of the general-purpose registers for data loading and the decoder enables its output channel connecting to the registers through the MUX of the size 2 x1. This MUX receives 32-bits data operand from External De-MUX at input "1" and receives 32-bits results from RFUs at the input "0". If the Ext_IO_En=0 then it selects the result coming from the RFUs and loads it in the register. If the Ext_IO_En=1 then it selects the data coming from the External De-MUX and loads it in the registers. Since there are eight RFUs that can load their results in the same register, hence in order to solve this problem an 8 x 1 MUX (32-bits) is interfaced with each register input. Each MUX is controlled by the *RFU Data-path Controller* which analyzes the Destination Addresses of all the RFUs and selects only that RFU whose output is valid output. In order to store the results and the flags being available in the GPRs and flag registers (FRs) into the data cache of the RISP, the 32 x 1 External MUX (32-bits) is used which can read the contents of the selected register and sends it to the data cache of the RISP.

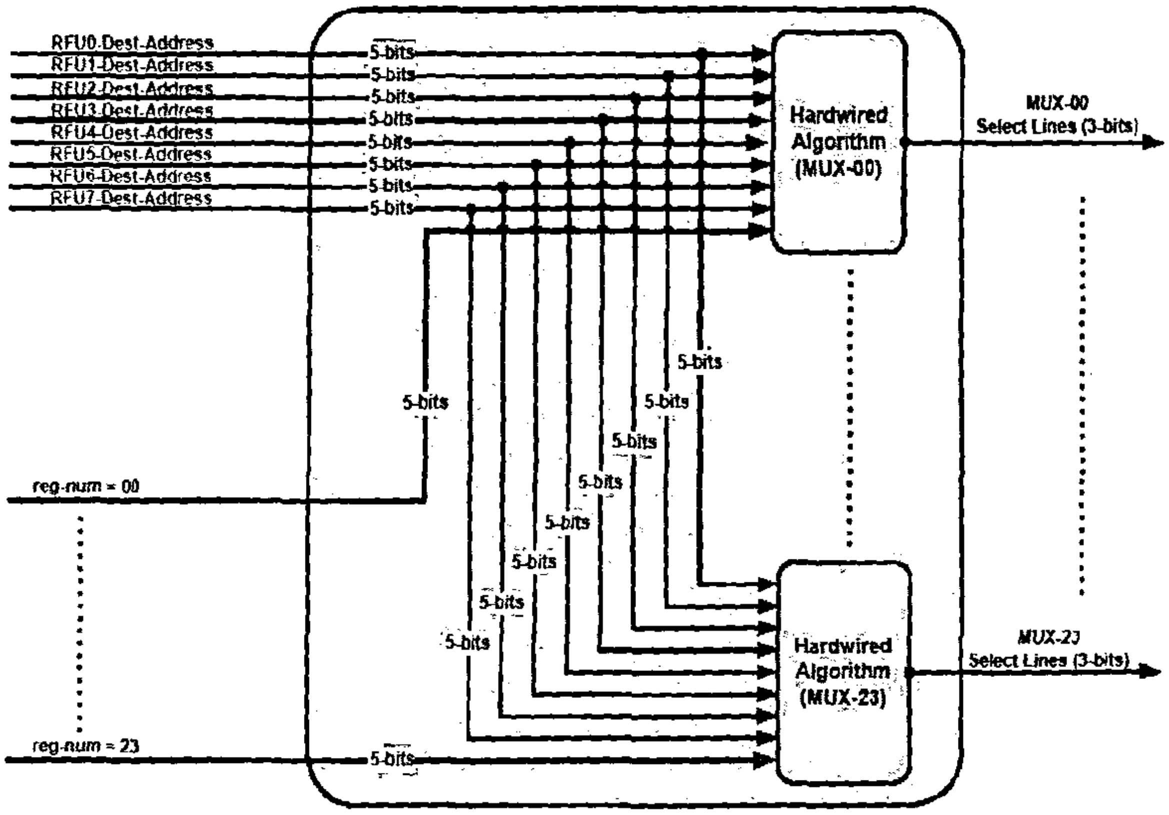


Figure 18 RFU Data Path Controller

b) RFUs Data-in / Data-out Logic (**RDIOL*****)***

In order to load/store the data across the RFUs there are two 32 x 1 MUXs (32-bits) and one 1 x 24 De-MUX (32-bits) for each RFU. Using the two MUXs the RFU is able to read the source data operands (Sr-1 and Sr-2) from any one of the 32 registers and using the one De-MUX it stores its results back to any one of the GPRs. Flags generated during the execution of the VLIW are loaded into the relevant FRs.

c) General-Purpose and Flag Registers (GFRs)

There is an array of eight FRs (32-bits) and twenty four GPRs (32-bits). GPRs can be read and written by the programmer but the FRs can only be read by the programmer and can not be written. RFUs can read/write any one of these thirty two registers. More than one RFU can read the contents of the same register at the same time but only one RFU can write in a register at the same time because the read operation is shareable but the write operation is not shareable.

d) Registers Input/Output Logic (RIOL)

FRs are loaded with the flags, being generated by the RFUs and can be read by the programmer through the External MUX. In case of the GPRs, the programmer can read the registers through the External MUX but in order to write contents into registers there is a 2 x 1 MUX (32-bits) which selects the data for the register either from some RFU output or from data cache. The 8 x 1 MUX interfaced at the input of the 2 x 1 MUX selects the valid RFU for the results to be stored in the register. In order to select the valid RFU for results, there is a RFU Data path Controller shown in Fig.6 is attached with all MUXs. This controller reads the select lines of all the De-MUXs of RFUs and after analysis it selects that RFU whose output is a valid output.

e) Reconfigurable Functional Units (RFUs)

There are a total of eight reconfigurable functional units RFUs. Each RFU has some standard IO interfaces for configuration and data flow in and out the unit as is shown in the fig. 4.13

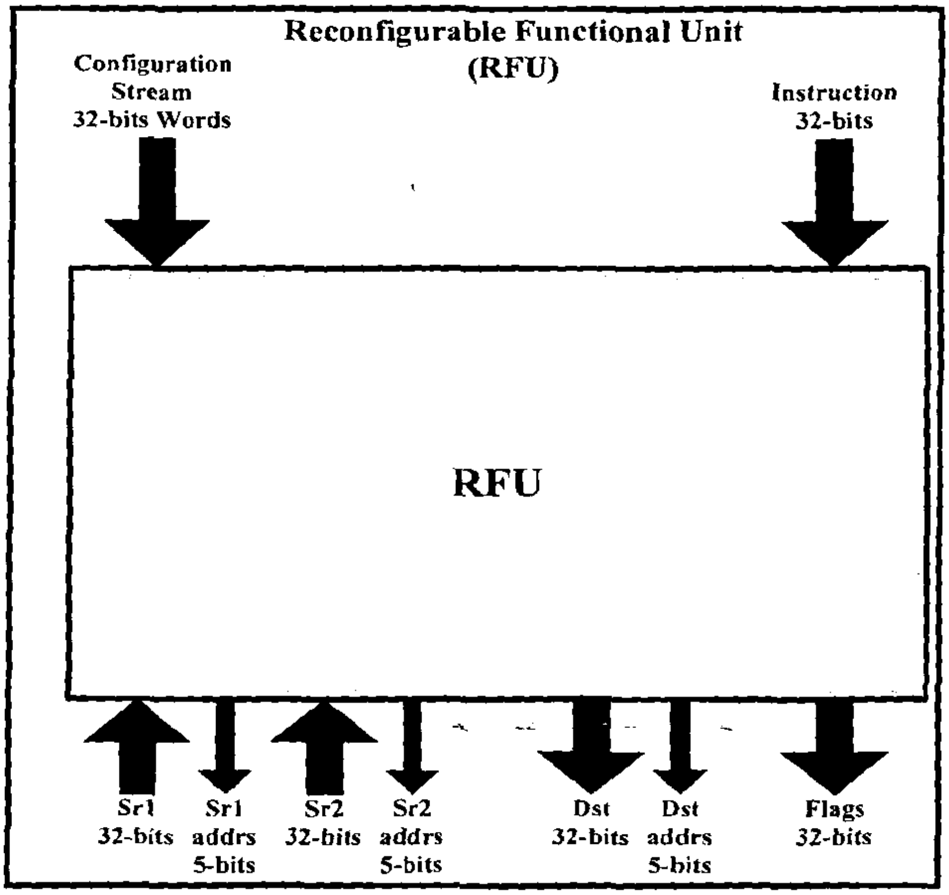


Figure 19 Reconfigurable Functional Unit Interfaces

Hence from the fig it is obvious that it receives 32-bits sized two source operands from the register file of the VLIW-RISP and after computation generates one 32-bits result and 32-bits flags. When it reads its source operands and sends the destination operands from/to registers it also sends, 5-bits sized each, source and the destination addresses. Each RFU has a data bus dedicated for instruction loading and its size is 32-bits. Also there is a 32-bits configuration bus that is used by it, to read/write the configuration data into or out of the device.

RFU Data-path Controller Algorithm

The Algorithm Initially Reads the Register Address (Rmn) and Destination Operand Addresses of all RFUs

```
 if (RFU0-Dest-Address == Rmn Address)
    Then Sel_out = 0;

else if (RFU1-Dest-Address == Rmn Address)
    Then Sel_out = 1;

else if (RFU2-Dest-Address == Rmn Address)
    Then Sel_out = 2;

else if (RFU3-Dest-Address == Rmn Address)
    Then Sel_out = 3;

else if (RFU4-Dest-Address == Rmn Address)
    Then Sel_out = 4;

else if (RFU5-Dest-Address == Rmn Address)
    Then Sel_out = 5;

else if (RFU6-Dest-Address == Rmn Address)
    Then Sel_out = 6;

else if (RFU7-Dest-Address == Rmn Address)
    Then Sel_out = 7;

else Sel_out = Nil;
```

If we take the more detailed view of the RFU, we get the picture shown in the fig.4.14. It contains an Instruction Register IR and an FPGA Logic. The FPGA Logic is further subdivided into the two areas. One is the Non-Reconfigurable Area, which generates the flags of the RFU, and the second area is the Reconfigurable area, which is the most important portion of the VLIW-RISP. This is the only region inside the VLIW-RISP that can be reconfigured. This reconfigurable are is used to map many hardwares on the device and reconfiguration of the device during its execution.

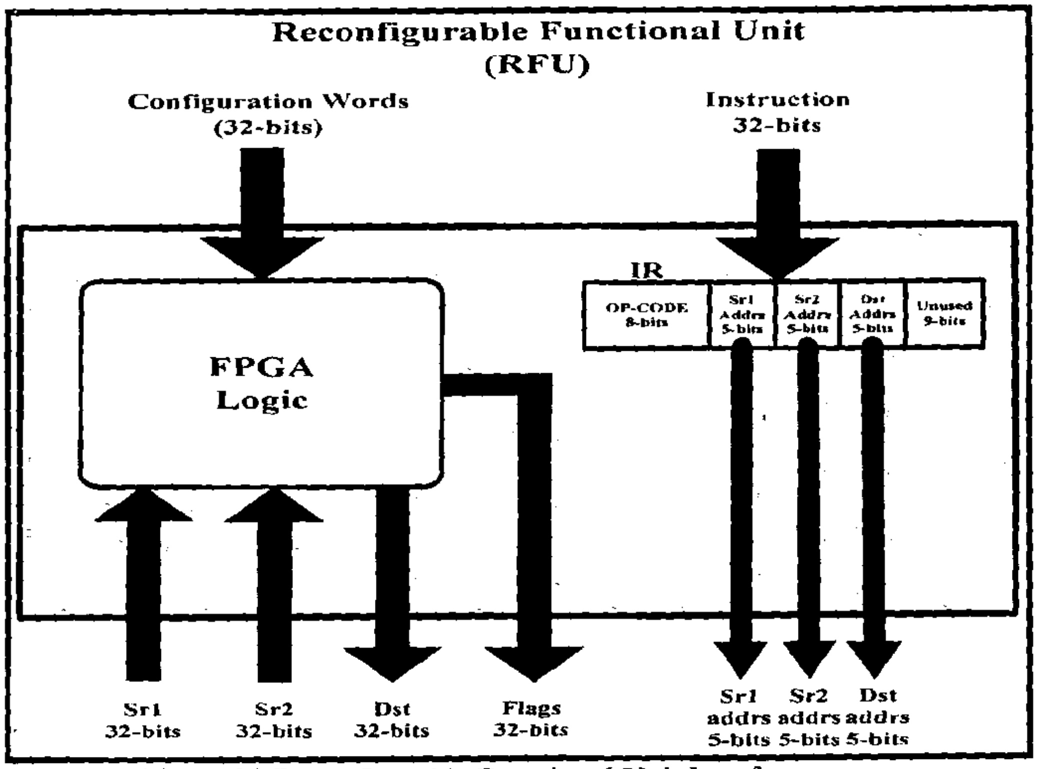


Figure 20 Reconfigurable Functional Unit Interfaces

The reconfigurable functional Unit is the reconfigurable area of the proposed VLIW-RISP. In fact this area is the area of the Field Programmable Gate Array FPGA being used for the design and the testing of the VLIW-RISP. The FPGA being required should have the property of the partial reconfiguration at any time of the device working. There are many different venders of the FPGA devices like Xilinx, Altera, Atmel and Triscend etc. But only a few of them are providing the FPGAs that can be configured at run time, partially. They include the well known Xilinx Corporation and the Atmel. The Virtex-Series of the FPGAs provided by the Xilinx Corporation are all partially reconfigurable at run time of the device. Also the 6200 Series of the FPGAs provided by the Atmel are also partially reconfigurable. But if we compare both of them, then we will found that the Virtex-Series FPGAs provided by the Xilinx Corporation are much better solution than the 6200 Series FPGAs of the Atmel. This comparison is based on the following factors

- Device Capacity (No of the logic gates)
- Device Speed (Configuration and Working Speeds)
- Device Flexibility (Methods of usage of internal components)
- Device Compatibility (Ineffaceability with Processors)
- Device Maturity (Device architecture Maturity)

- Device Availability (Device Market)

Hence due to all these factors, the device being chosen for the proposed VLIW-RISP design is the Virtex-E of the Xilinx Corporation. Hence in this manner the reconfigurable are being mentioned above is the area of the Virtex-E FPGA. The internal structure and the working of the structure of the Virtex-E FPGA is explained here in detail.

f) Flags Generation Logic (FGL)

The outputs generated by the RFUs are also read by the FGL and the flags are calculated for each RFU. Flag register is a 32-bits register but recently only Carry Flag, Sign Flag, Zero Flag, Overflow Flag and Equal Flag have been computed in the system and the remaining twenty-seven bits are available for the future extension.

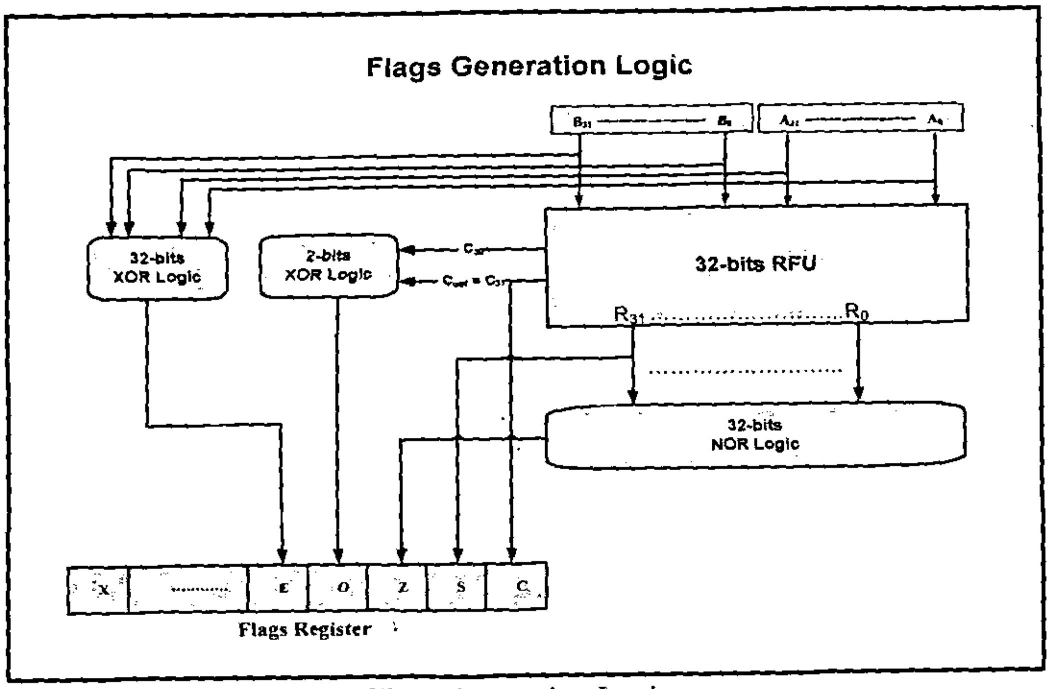


Figure 21 Flags Generation Logic

i. VLIW Configuration Unit (VCU):

VCU is composed of a *Configuration Controller* as shown in Fig.4.16 and a *Multi-port Configuration Memory* as shown. Configuration controller receives the op-codes of the eight instructions of the VLIW from the VFU and on the basis of these op-codes it decides to load one of the configuration blocks available in the memory for each RFU (if required). Also it checks if the op-code is a No Operation (NOP) or is same as that of any one of the existing op-codes. If so then the configuration controller does not load this new configuration into the RFUs but the hardware that is already loaded in the RFUs is reused and hence the configuration time that was required for the reconfiguration of RFUs is saved. Hence only those RFUs are reconfigured that are quite new ones. Hence the processor always takes the minimum possible time to reconfigure the RFUs during the execution of the application program and always has the most optimal configuration overhead.

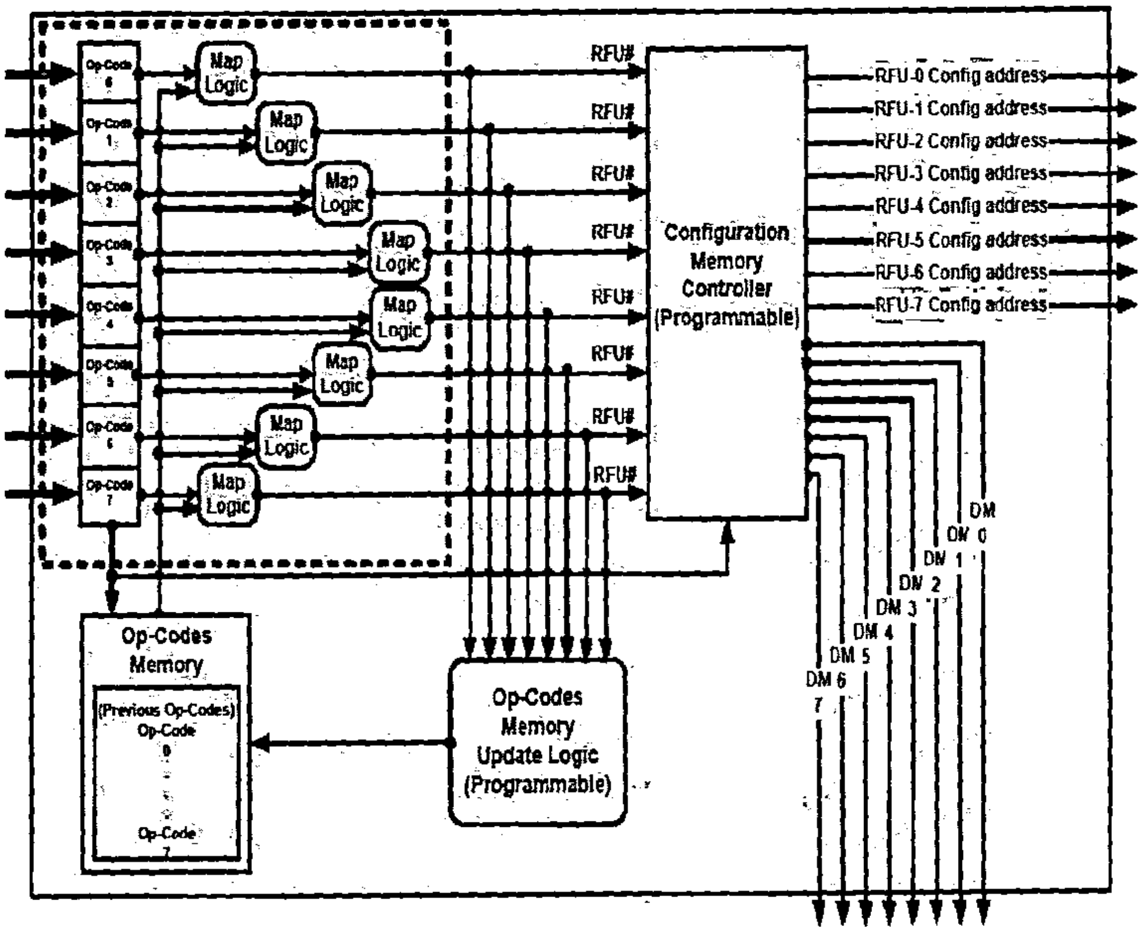


Figure 4.16 RFU Configuration Controller

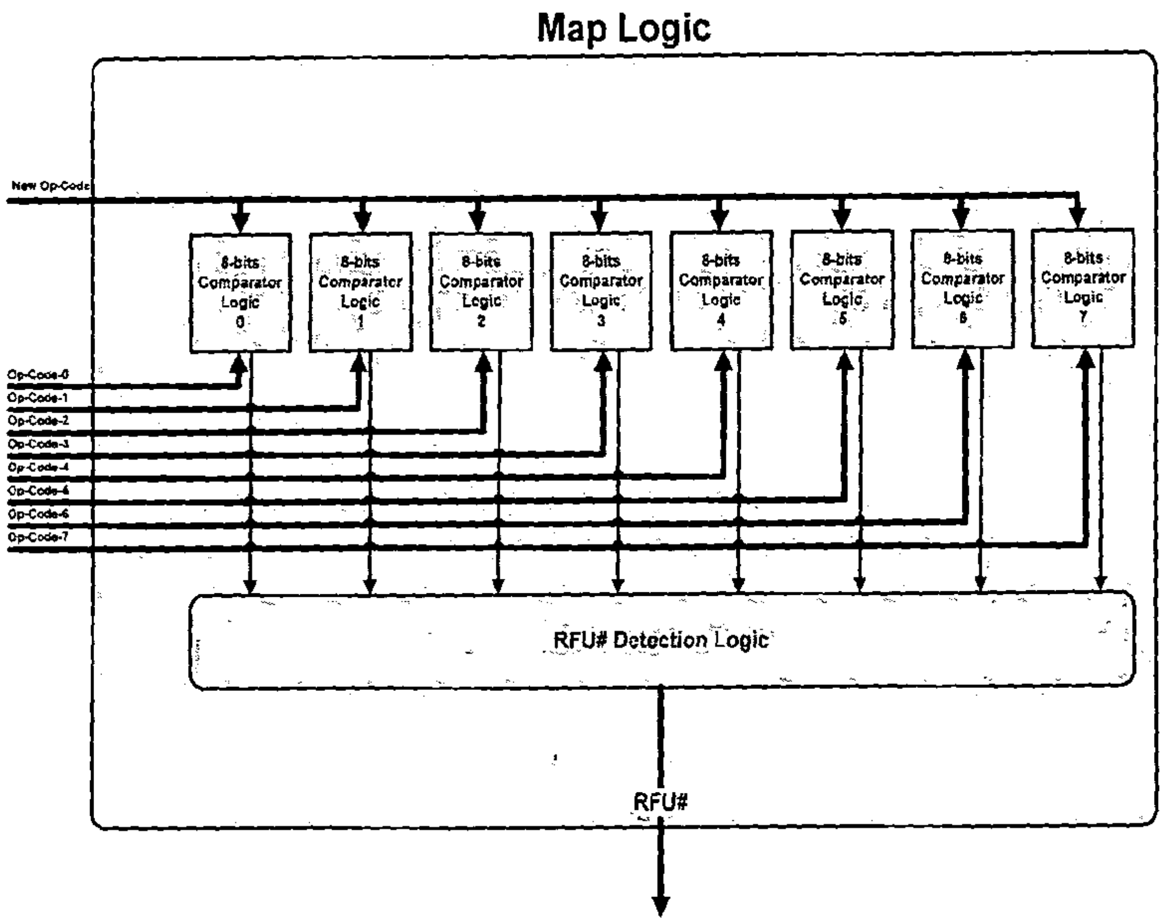

Map Logic
New Op-Code
8-bits Comparator Logic 0
8-bits Comparator Logic 1
8-bits Comparator Logic 2
8-bits Comparator Logic 3
8-bits Comparator Logic 4
8-bits Comparator Logic 5
8-bits Comparator Logic 6
8-bits Comparator Logic 7
Op-Code-0
Op-Code-1
Op-Code-2
Op-Code-3
Op-Code-4
Op-Code-5
Op-Code-6
Op-Code-7
RFU# Detection Logic
RFU#

Figure 226 Micro-programmed Control Unit

4.5.8 Micro-programmed Control Unit (MCU)

The control unit is the central controlling module of the RISP. All activities are generated and managed inside the control unit. There are two different approaches available for the design of the control unit. One is the Hardwired Control Unit Design and the other is the Micro-Programmed Control Unit. The control unit of the RISP is based on the Micro-programmed technology. It is a micro-coded state machines design. State machine of the VLIW-RISP control unit is shown later

It controls all the activities inside and outside of the processor from the hard ware configuration to the program execution. MCU is being designed using Micro-*coded State Machine* architecture. Consider the Fig.4.18 of MCU.

At each state of the control unit state machine, a set of the micro codes is generated and sent to the VLIW-RISP modules. These control signals actually control the processing of the processor. There is a handshaking mechanism developed between the control unit and the other modules of the VLIW-RISP. Due to this handshaking mechanism the different modules of the processor are synchronized with each other.

The control unit of a microprocessor directs the operation of the other units by providing timing and control signals. It is the function of the microcomputer to execute programs which are stored in memory in the form of instructions and data. The control unit contains the necessary logic to interpret instructions and to generate the necessary signals for the execution of those instructions. The descriptive words "fetch" and "execute" are used to describe the actions of the control unit. It fetches an instruction by sending address and a read command to the memory unit. The instruction at that memory address is transferred to the control unit for decoding. It then generates the necessary signals to execute the instruction.

For the control unit to perform its function, it must have input that allow it to determine the state of the system and output that allow it to control the behavior of the system. These are the external specification of the control unit. Internally, the control unit must have the logic required to perform its sequencing and execution functions [28]. Figure 3.4 is a general model of the control unit, showing all its inputs and outputs.

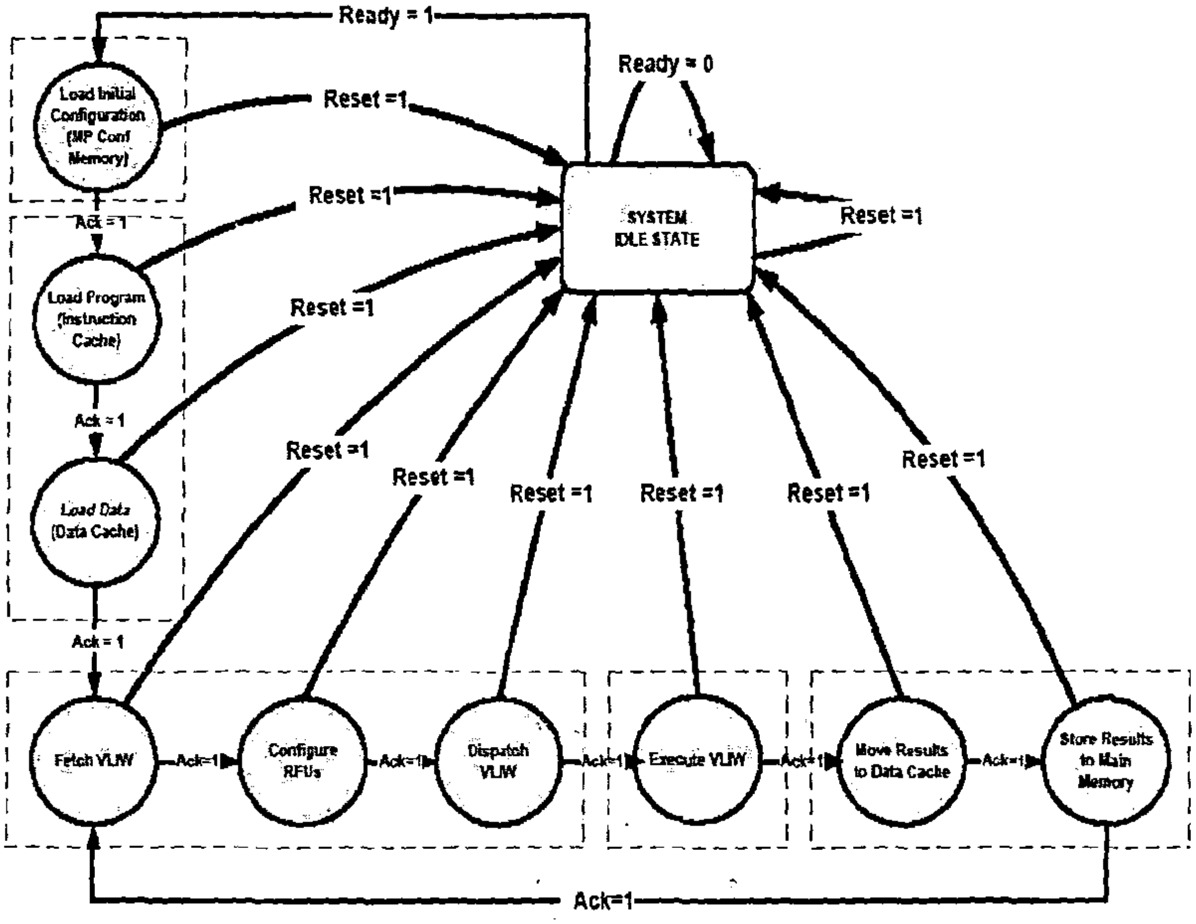

Ready = 1
Ready = 0
Reset =1
SYSTEM IDLE STATE
Load Initial Configuration (MP Conf Memory)
Ack = 1
Load Program (Instruction Cache)
Load Data (Data Cache)
Fetch VLIW
Configure RFUs
Dispatch VLIW
Execute VLIW
Move Results to Data Cache
Store Results to Main Memory
Ack=1

Figure 23 Micro-programmed Control Unit

The Inputs:

Clock: This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro operations) to be performed for each clock pulse. This is sometime referred to as the processor cycle time, or the clock cycle.

Instruction register: The op code of the instruction is used to determine which micro-operation to perform during the execute cycle.

Flags: These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example for the increment and skip-if-zero (ISZ) instruction, the control unit will increment the PC if the zero flag is set.

Control Signals from control Bus: the control bus portion of the system bus provides signals to the control unit, such as interrupt signals and acknowledgements.

The Outputs:

Control signals within the processor
These are of two types: Those that cause data to be moved from one register to another, and those that activate specific RFUs functions.

Control Signals to control bus
These are also of two types: Control signals to memory, and control signals to the I/O modules.

Chapter No. 5
Statistics and Performance Analysis

Chapter No.5

Statistics and Performance Analysis

5.1 DSP (TMS320C6X) Statistics [46]

In order to compare the performance of the proposed VLIW-RISP with a DSP we have chosen the DSP processor TMS320C6X provided by the Texas Instruments. It is a fixed-point VLIW architecture containing a total of eight functional units. They include two Multipliers and six ALUs. The pipeline of the TMS320C6X can fetch a VLIW of eight instructions. It is known as ***Fetch-Packet.*** A fetch packet is converted into an ***Execute-Packet*** by looking at the resources available. An execute packet consists of those instructions that can be executed in the pipeline in parallel without any resource conflicts. The program fetch, the program dispatch and instruction decode units can deliver up to eight 32-bits instructions (One VLIW) to the functional units every CPU clock cycle. Hence it can execute a maximum of eight instructions in a single CPU clock cycle, if these instructions have no internal resource conflicts. In case of internal resource conflicts, these fetch-packets are converted into two to eight execute packets and then each execute-packet takes one CPU cycle to execute it.

The execution of fixed-point instructions of the TMS320C6X can be defined in terms of ***Delay Slots***. The number of delay slots is equivalent to the number of cycles required after the source operands are read for the result to be available for reading. For a single-cycle type instruction (such as ADD, SUB) source operands read in cycle i produce a result that can be read in cycle i + 1 (Hence Delay slot is zero). For a multiply instruction (MPY), source operands read in cycle i produce a result that can be read in cycle i + 2 (Hence Delay slot is one). Delay slots are equivalent to an execution or result latency. All of the instructions that are common to the 'C62x and 'C67x have a functional unit latency of 1. This means that a new instruction can be started on the functional unit each cycle. The following statistics and execution formula are calculated from the technical notes of DSP processor (TMS320C6X)

$$T_T = FP\,(T_{PFT} + T_{OFT}) + ((F_n + D_n) + \ldots + (F_0 + D_0)) \text{ Cycles}$$

Where

Sr#	Parameter	Values
1	No of fetch packets (FP)	1, 2, 3, ...n in each program
2	Packet Fetch Time (T_{PFT})	1-Cycle for each fetched packet
3	Operands Fetch Time (T_{OFT})	1-Cycle for each fetched packet
4	Execute packets ($E_n = F_n + D_n$)	1-4 for each fetched packet
5	Delay Slots (D_n)	0-Cycles for ADD/SUB ,1-Cycle for MUL
6	Functional Unit Latency (F_n)	1-Cycle for each execute packet
7	Total Execution Time	T_T

Table 3 Statistics and Execution Formula of DSP Processor

5.2 VLIW-RISP Statistics

The VLIW-RISP is fetching the instructions externally one by one using a Pre-fetch Unit. This pre-fetching of instructions and its packing into VLIW and loading into the Instruction cache has been overlapped with the program execution. Hence time consumed is considered to be zero.

The VLIW-Fetch Unit takes one cycle to fetch one VLIW. Since the proposed architecture is a Register-Register Architecture hence operands fetch time for each fetched VLIW is always one cycle. The Configuration Unit takes maximum of one cycle to update the configuration of RFUs. The VLIW-Dispatch Unit takes one cycle to dispatch (Issue) one VLIW. The execution time taken by Execution Unit depends upon the type of the instructions to be executed. The followings are the statistics and execution formula of the proposed VLIW-RISP.

$$T_T = FP\,(T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_n + D_n) \ldots + (F_0 + D_0)) \quad \text{Cycles}$$

Where

Sr#	Parameter	Values
1	No of fetch packets (FP)	1, 2, 3, ...n in each program
2	Packet Fetch Time (T_{PFT})	1-Cycle for each fetched packet
3	Operands Fetch Time (T_{OFT})	1-Cycle for each fetched packet
4	Execute packets ($E_n = F_n + D_n$)	1 for each fetched packet
5	Delay Slots (D_n)	0-Cycles for ADD/SUB ,1-Cycle for MUL
6	Functional Unit Latency (Fn)	1-Cycle for each execute packet
7	Configuration Time T_C	0-Cycle (Min), 1-Cycle (Max)
8	Dispatching Time T_D	1-Cycle
9	Total Execution Time	T_T

Table 4 Statistics and Execution Formula of VLIW-RISP

Now according to the above statistics the following assembly language programs have been executed and the no of execution cycles have been calculated.

==

Program No.1:

This application program is consisting of simple arithmetic operations including Addition, Subtraction and Multiplication. In simulation and calculations, the all of these operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of only eight instructions which make only one VLIW for the proposed processor.

```
ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
```

1. **TMS320C6X Performance (Maximum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + (F_0 + D_0)$ Cycles
$= 1\ (1 + 1) + (2) = 4$ **Cycles**

2. **VLIW-RISP Performance (Minimum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_0 + D_0))$ Cycles
$= 1\ (1 + 1) + (1) + (1) + ((2)) = 6$ **Cycles**

==

Program No.2:

This application program is consisting of simple arithmetic operations including Addition and Subtraction. In simulation and calculations, the all of these operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of only eight instructions which make only one VLIW for the proposed processor.

```
ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
SUB  R00, R01, R08;
SUB  R00, R01, R09;
```

1. **TMS320C6X Performance (Maximum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + ((F_I + D_I) + (F_0 + D_0))$ Cycles
$= 1\ (1 + 1) + ((1) + (1)) = 4$ **Cycles**

2. VLIW-RISP Performance (Minimum)

$T_T = FP\,(T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_0 + D_0))$ Cycles

$= 1\,(1 + 1) + (1) + (1) + ((1)) = 5$ **Cycles**

==

Program No.3

This application program is consisting of simple arithmetic operation of Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of only eight instructions which make only one VLIW for the proposed processor.

```
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
```

1. TMS320C6X Performance (Maximum)

$T_T = FP\,(T_{PFT} + T_{OFT}) + ((F_3 + D_3) + \ldots\ldots\ldots + (F_0 + D_0))$ Cycles

$= 1\,(1 + 1) + ((2) + (2) + (2) + (2)) = 10$ **Cycles**

2. VLIW-RISP Performance (Minimum)

$T_T = FP\,(T_{PFT} + T_{OFT}) + (T_C) + (T_D) + (F_0 + D_0))$ Cycles

$= 1\,(1 + 1) + (1) + (1) + ((2)) = 6$ **Cycles**

==

Program No.4:

This application program is consisting of simple arithmetic operations including only Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of sixteen instructions which make two VLIWs for the proposed processor. Since all instructions are representing to the same operation, hence only first VLIW will be reconfigured and the same configuration will be used by the second VLIW. Hence the performance will be much higher as compared to a conventional DSP processor.

```
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
```

1. **TMS320C6X Performance (*Maximum*)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + (F_7 + D_7) + \ldots\ldots\ldots + (F_0 + D_0)$ Cycles
$= 2\ (1 + 1) + ((2) + (2) + (2) + (2) + (2) + (2) + (2) + (2)) =$ **20 Cycles**

2. **VLIW-RISP Performance (Minimum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_1 + D_1) + (F_0 + D_0))$ Cycles
$= 2\ (1 + 1) + (1) + (2) + ((2) + (2)) =$ **11Cycles**

==

Program No.5:

This application program is consisting of simple arithmetic operations including Addition, Subtraction and Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of sixteen instructions which make three VLIWs for the proposed processor. Since all instructions are representing to the different operation, hence both times the VLIWs will be reconfigured and hence some what higher configuration time will be used by the second VLIW. Hence the performance will be effected as compared to a conventional DSP processor.

```
ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
SUB  R00, R01, R08;
SUB  R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
```

1. TMS320C6X Performance (Maximum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + ((F_5 + D_5) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 2\ (1 + 1) + ((2) + (2) + (2) + (2) + (1) + (1)) =$ **14 Cycles**

2. VLIW-RISP Performance (Minimum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_1 + D_1) + (F_0 + D_0))$ Cycles
$= 2\ (1 + 1) + (2) + (2) + ((2) + (1)) =$ **11Cycles**

==

Program No.6:

This application program is consisting of simple arithmetic operations including only Addition and Subtraction. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of sixteen instructions which make two VLIWs for the proposed processor. Since both VLIWs are representing to the same operations, hence only first VLIW will be reconfigured and the same configuration will be used by the second VLIW. Hence the performance will be much higher as compared to a conventional DSP processor.

```
ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
SUB  R00, R01, R08;
SUB  R00, R01, R09;

ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
SUB  R00, R01, R08;
SUB  R00, R01, R09;
```

1. **TMS320C6X Performance (Maximum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + ((F_3 + D_3) + \ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 2\ (1 + 1) + ((1) + (1) + (1) + (1)) = 8$ **Cycles**

2. VLIW-RISP Performance (Minimum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_1 + D_1) + \ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 2\ (1 + 1) + (1) + (2) + ((1) + (1)) = 9$ **Cycles**

==

Program No.7:

This application program is consisting of simple arithmetic operations including Addition, Subtraction and Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of twenty four instructions which make three VLIWs for the proposed processor. Since all instructions are representing to the different operation, hence first VLIW will be reconfigured and similarly second and third VLIW will also be reconfigured. Hence the configuration time of the program will be higher.

```
ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
SUB  R00, R01, R08;
SUB  R00, R01, R09;
```

```
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
SUB  R00, R01, R06;
SUB  R00, R01, R07;
SUB  R00, R01, R08;
SUB  R00, R01, R09;
```

1. **TMS320C6X Performance (Maximum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + ((F_7 + D_7) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 3\ (1 + 1) + ((1) + (1) + (2) + (2) + (2) + (2) + (1) + (1)) \approx$ **18 Cycles**

2. **VLIW-RISP Performance (Minimum)**

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_2 + D_2) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 3\ (1 + 1) + (3) + (3) + ((1) + (2) + (1)) =$ **16 Cycles**

==

Program No.8:

This application program is consisting of simple arithmetic operations including Addition, Subtraction and Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of twenty four instructions which make three VLIWs for the proposed processor. Since all instructions are representing to the different operation, hence first VLIW will be reconfigured and similarly second and third VLIW will also be reconfigured. Hence the configuration time of the program will be much higher.

```
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

ADD R00, R01, R02;
ADD R00, R01, R03;
ADD R00, R01, R04;
ADD R00, R01, R05;
ADD R00, R01, R06;
ADD R00, R01, R07;
ADD R00, R01, R08;
ADD R00, R01, R09;

SUB R00, R01, R02;
SUB R00, R01, R03;
SUB R00, R01, R04;
SUB R00, R01, R05;
SUB R00, R01, R06;
SUB R00, R01, R07;
SUB R00, R01, R08;
SUB R00, R01, R09;
```

1. TMS320C6X Performance (Maximum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + ((F_7 + D_7) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 3\ (1 + 1) + ((1) + (1) + (1) + (1) + (2) + (2) + (2) + (2)) = 18$ **Cycles**

2. VLIW-RISP Performance (Minimum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_1 + D_1) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles
$= 3\ (1 + 1) + (3) + (3) + ((1) + (1) + (2)) = 16$ **Cycles**

==

Program No.9:

This application program is consisting of simple arithmetic operations including only Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of forty instructions which make five VLIWs for the proposed processor. Since all instructions are representing to the same operation, hence only first VLIW will be reconfigured and the same configuration will be used by the remaining four VLIWs. Hence the performance will be much higher as compared to a conventional DSP processor.

```
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
```

1. **TMS320C6X Performance (Maximum)**

$T_T = FP\,(T_{PFT} + T_{OFT}) + ((F_{19} + D_{19}) + \dots\dots\dots\dots + (F_0 + D_0))$ Cycles

$= 5\,(1 + 1) + ((2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) +$
$+ (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2))$

= 50 Cycles

2. VLIW-RISP Performance (Minimum)

$T_T = FP\,(T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_I + D_I) + \dots\dots\dots\dots + (F_0 + D_0))$ Cycles

$= 5\,(1 + 1) + (1) + (5) + ((2) + (2) + (2) + (2) + (2))$

= 26 Cycles

==

Program No.10:

This application program is consisting of simple arithmetic operations including only Multiplication. In simulation and calculations, the operations have been taken with the same delay slots and functional unit latencies as that of those provided by TMS320C6X DSP processor. The assumptions have been taken for the sake of easy and authentic performance comparison. This program has been supposed to be consisting of eighty instructions which make ten VLIWs for the proposed processor. Since all instructions are representing to the same operation, hence only first VLIW will be reconfigured and the same configuration will be used by the remaining nine VLIWs. Hence the performance will be much higher as compared to a conventional DSP processor. This program execution on proposed RISP shows that the reconfigurable processor exhibits a much higher performance gain than any conventional DSP processor.

```
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;

MUL R00, R01, R02;
MUL R00, R01, R03;
MUL R00, R01, R04;
MUL R00, R01, R05;
MUL R00, R01, R06;
MUL R00, R01, R07;
MUL R00, R01, R08;
MUL R00, R01, R09;
```

1. TMS320C6X Performance (Maximum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + ((F_{19} + D_{19}) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles

= 10 (1 + 1) + (
(2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) +
(2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) +
(2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) +
(2))
= 100 Cycles

2. VLIW-RISP Performance (Minimum)

$T_T = FP\ (T_{PFT} + T_{OFT}) + (T_C) + (T_D) + ((F_9 + D_9) + \ldots\ldots\ldots\ldots + (F_0 + D_0))$ Cycles

= 10 (1 + 1) + (1) + (10) + (

(2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2) + (2))

= 51 Cycles

==

Calculated Statistics

Program No.	DSP Execution Time (No of Cycles)	VLIW-RISP Execution Time (No of Cycles)
1	4	6
2	4	5
3	10	6
4	20	11
5	14	11
6	8	9
7	18	16
8	18	16
9	50	26
10	100	51

Table 5 Calculated Statistics of Both Processors

5.3 DSP vs VLIW-RISP Performance Analysis

The graph being obtained by comparing the speed of the conventional DSP processor named TMS320C6X with the proposed RISP is shown in Figure 5.1.

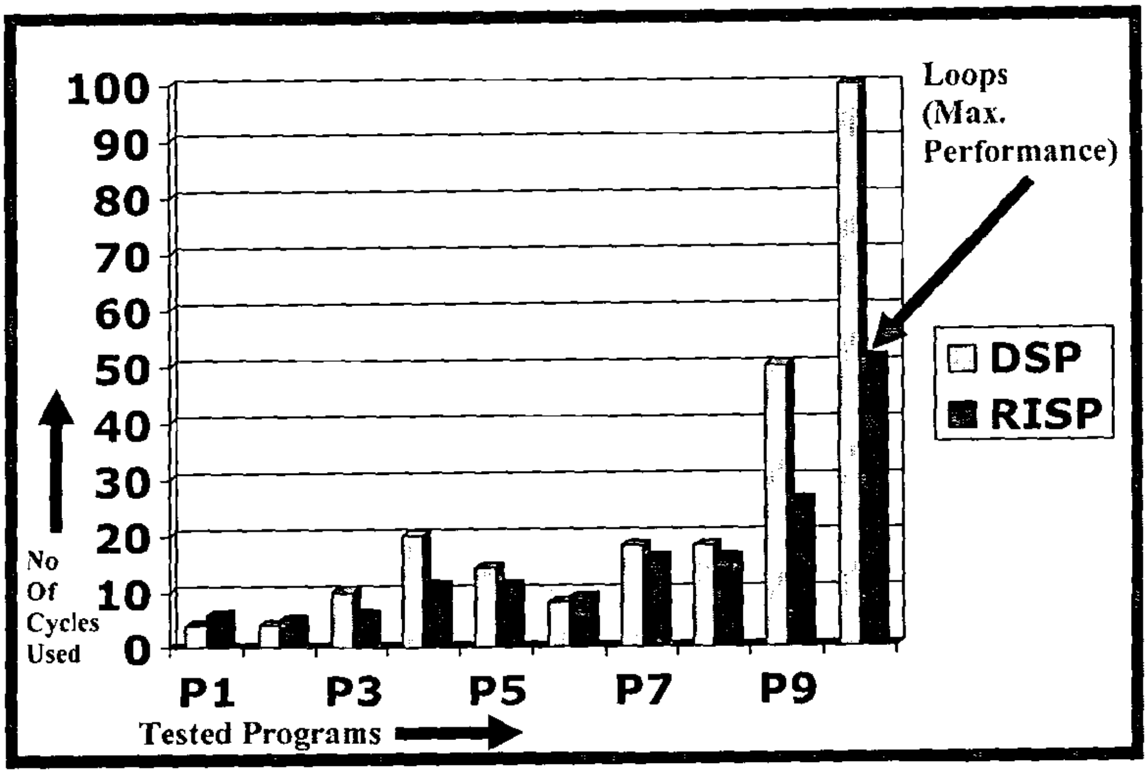


Figure 24 DSP vs VLIW-RISP Performance Analysis

Chapter No. 6
Conclusion and Future Work

Chapter No.6

Conclusion and Future Work

6.1 Conclusion

Now the reconfigurable computing based systems are becoming an important part of research work by different researchers in the fields of computer architectures. In this domain of computing; placing the computationally very intense portions of any under execution application program onto the reconfigurable computing hardware, that application is being accelerated to a much high performance. It happens due to the fact that reconfigurable computing architectures combine the advantages of both the software based and Application Specific Integrated Circuits based implementations. Like software based applications, the mapped circuits are quite flexible and hence can be changed during the execution time of the system. Similar reconfigurable computing systems provide us a method to map circuits into hardware in the same manner as that of the ASICs. Therefore the reconfigurable computing systems or devices have a great potential to achieve much greater performance gain as compared to that of the software based solutions due to bypassing the conventional fetch-decode-execute instruction cycle of the general or traditional microprocessors.

Reconfigurable Instruction Set Processors (RISPs) have been evolved through many design alternatives but the main theme of the design was always the tightly coupled nature of integrated reconfigurable logic inside the processor core. In the resent era the main focus of the research is to overcome the drastic execution delays being introduced by the configuration overheads of RFUs. Researchers have introduced different techniques to tackle this overhead including Run-time Reconfiguration, Partial Reconfiguration, Configuration Compression, Pipelined Configuration, Multi-threaded Configurations, Configuration Cloning, Configuration Re-usability and Configuration Overhead Optimization using the intelligent configuration controllers.

In this research thesis a Reconfigurable Instruction Set Processor (RISP) design has been proposed with the capability of the most optimized configuration overheads. Due to the VLIW nature of the proposed processor; at one hand the multi-threaded reconfiguration of the RFUs has been exploited along with the partial run-time reconfiguration as well as on the other hand the configuration intelligent re-usability has been overlapped. In order to achieve the multi-threaded reconfiguration and the intelligent re-usability of the existing configurations, a multi-port configuration memory and a hard wired algorithmic configuration controller has been designed so that to optimize the configuration overheads by configuring the minimum number of RFUs. The processor always takes the maximum advantage of the existing configurations and hence providing the minimum possible configuration overheads.

6.2 Future Work

It is in general true that no project is ever finished and done with 100% satisfactory performance in accordance with the requirements leading to its origin. It is just halted on different stages in the development process and is realized as product. As a designer it is an obligation to look forward some generations and make sure that the design will be able to continue to improve. Some suggestions are given below.

6.2.1 Hardware Improvement

The possible refinement in the proposed VLIW-RISP design is to improve the design of Configuration Unit so that to adopt the most complicated and advanced techniques of partial reconfiguration. While keeping the external interface same. The Configuration Unit is using multi-port memory to store the status of RPUs. Multi-port memory is an expensive solution. We must find an alternate solution to reduce the cost.

6.2.2 Configuration Protocol

The processor is reconfigured through an external interface, hence slow. We can work on the reconfiguration protocol as part of the processor's module. The external processor would then be able to reconfigure the FPGA simply by writing the configuration into a special memory area handled by the static module. Static module will in turn reconfigure the RPUs.

6.2.3 Configuration Techniques

Keeping all the existing resources of the proposed processor unchanged there are many new techniques, which can be used to minimize the configuration size and configuration overheads. The configuration minimization is a process relevant to configuration compression techniques. This area is quite new and open for researches to give compression techniques. The configuration overhead can be reduced by providing the emerging technique of partial reconfiguration. Along with partial reconfiguration techniques a new process of configuration cloning has been introduced in which the existing configuration streams can be replicated within the chip and hence introducing another very optimistic process for research work.

Bibliography

Bibliography

[1] M. Aqeel Iqbal and Uzma Saeed Awan, 'Reconfigurable Instruction Set Processor Design Using Software Based Configuration', *Proceedings of IEEE computer society, IEEE* International Conference on Advanced Computer Theory and Engineering 2008 (ICACTE-2008), *December 20-22, 2008, Phuket Island, Thailand.*

[2] M. Aqeel Iqbal, Shoab A. Khan and Uzma Saeed Awan, 'RISP Design with Most Optimal Configuration Overhead for VLIW Based Architectures', *Proceedings of IEEE computer society, 2nd IEEE International Conference on Electrical Engineering 2008 (ICEE-2008), March 25-26, 2008, UET Lahore, Pakistan.*

[3] M. Aqeel Iqbal and Uzma Saeed Awan, "An Efficient Configuration Unit Design for VLIW Based Reconfigurable Processors", Proceedings of IEEE Computer Society, 12th IEEE International Multi-topic Conference 2008 (IEEE INMIC-2008), December 23-24, 2008, Bahria University, Karachi, Pakistan.

[4] M. Aqeel Iqbal and Uzma Saeed Awan, "Run-Time Reconfigurable Instruction Set Processor Design: RT-RISP", Proceedings of IEEE Computer Society, 2nd IEEE International Conference on Computer, Control and Communication 2009 (IEEE ICCCC-2009), February 17-18, 2009, Pakistan Navy Engineering College, Karachi, Pakistan.

[5] M. Aqeel Iqbal and Uzma Saeed Awan, "Reconfigurable Processor Architecture For High Speed Applications", Proceedings of IEEE Computer Society, IEEE International Advance Computing Conference 2009 (IEEE IACC-2009), March 6-7, 2009, Patiala, India.

[6] M. J. Wirthlin, Brad L. Hutchings. A dynamic instruction set computer. In Peter Athanas and Kenneth L. Pocek, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp 99--107, April 1995.

[7] Lucent Technology Inc. FPGA Data Book, 1998.

[8] Xilinx Inc. The Programmable Logic Data Book, 1994.

[9] A. DeHon. Reconfigurable Architectures for General Purpose Computing. PhD thesis, MIT AI Lab, September 1996.

[10] S. Hauck, et al, *The Chimaera Reconfigurable Functional Unit,* Proc. 5th IEEE Symp. FCCM, 1997, pp. 87-96.

[11] S.C. Goldstein, et al, *PipeRench: a Coprocessor for Streaming Multimedia Acceleration,* Proc. Int'l Symp. Computer Architecture, 1999, pp. 28-39.

[12] Altera Inc.. Altera Mega Core Functions, *http://www.altera.com/html/tools/megacore.html*, San Jose, CA, 1999.

[13] Xilinx, Inc. Virtex II Configuration Architecture Advanced Users' Guide. March, 2000.

[14] G. Estrin et al. Parallel Processing in a Restructurable Computer System. *IEEE Trans. Electronic Computers*, pp. 747-755, 1963

[15] P. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18,March 1993.

[16] J. D. Hadley and B. L. Hutchings, "Designing a partially reconfigured system," in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607 (J. Schewel, ed.), (Bellingham, WA), pp. 210–220, SPIE – The International Society for Optical Engineering, 1995.

[17] Johannes Kneip, Bernd Schmale, Henning Moller, "Applying and Implementing the MPEG-4 Multimedia Standards", IEEE Micro November/December 1999 (Vol. 19, No. 6)

[18] Trimedia Technologies Inc, "Trimedia 32 CPU Hand Book", http://www.trimedia.com.

[19] C. Systems. http://www.chameleonsystems.com/.

[20] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.

[21] Xilinx, Inc. Virtex Configuration Architecture Advanced Users' Guide. June, 1999.

[22] S. Brown and J. Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design* & Test of Computers, Summer 1996.

[23] Edson L. Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Washington University Department of Computer Science Technical Report WUCS-01-13. July 2001. (Available at http://wyww.arl.wustl.edu/arl/projects/fpx/parbit

[24] S. Scalera and J. V´azquez. "The design and implementation of a context switching Field Programmable Gate Array". Published in the IEEE Symposium organized on the Field-Programmable Computing Machines, held in April 1998.

[25] Athanas and Silverman. "The Processor Reconfiguration Through Instruction-Set Metamorphosis". Published in *IEEE Computer spciety,* 26(3):11–18, organized in March 1993.

[26] Xilinx, Inc.. XC6200 Field Programmable Gate Arrays Product Description. April 1997.

[27] Rose, E. Gamal and Sangiovanni. "The Architecture of the Field Programmable Gate Arrays (FPGAs). Published in the *Proceedings of the IEEE*, in July 1993.

[28] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field Programmable Gate Arrays. *Proceedings of the IEEE*, July 1993.

[29] A. DeHon. Reconfigurable Architectures for General Purpose Computing. PhD thesis, MIT AI Lab, September 1996.

[30] By Hoang, "Searching the genetic databases on splash 2" in IEEE Workshop on Field Programmable Gate Arrays (FPGAs) for Custom Computing Machines (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 185–191, IEEE Computer Society Press, 1993.

[31] S. J. Hauck. "The Roles of Field Programmable Gate Arrays in Programmable Systems". Published in the *Proceedings of IEEE*, 86, in April 1998.

[32] P. S. Sidhu, A. Mei, and V. K. Prasanna, "String matching on multicontext FPGAs using self-reconfiguration", in ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pages 217-226, Monterey, CA, February 1999.

[33] Philip James-Roxby and Steven A. Guccione. Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configurations. In Proceedings of IEEE Workshop on Field Programmable Custom Computing Machines, pages 153-161, April 2000.

[34] By Dehon. "DPGA-Coupled Microprocessors; The Commodity ICs for the Early 21st Century". Published in IEEE Symposium on Field Programmable Gate Arrays for Custom Computing Machines, held in April 1994.

[35] R. Razdan. *PRISC: ProgrammableReduced Instruction Set Computers*. PhD thesis, Harvard University, May 1994. ftp.eecs.harvard.edu:users/smith/theses/razdan-thesis.tar.gz.

[36] C. J. Rupp and M. L. Landguth and Garverick and Gomersall and Gokhale. "The NAPA Adaptive Processing Architecture". Published in *IEEE Symposium* on Field Programmable Gate Arrays for Custom Computing Machines, held in April 1998.

[37] Xilinx Inc.(www.xilinx.com). *Xilinx Platform FPGAs*.

[38] Aziz-Ur-Rehman, Dr. Aqeel A. Syed and M. Aqeel Iqbal, 'Intelligent Reconfigurable Instruction Set Processor (IRISP) Design', *Proceedings of IEEE*

computer society, 11th IEEE International Multi-topic Conference 2007 (*INMIC-2007), Dec 28-30, 2007, COMSATS Lahore, Pakistan.*

[39] P. Bertin, H. Touati, and E. Lagnese, "PAM programming environments: Practice and experience," in IEEE Workshop on FPGAs for Custom Computing Machines (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 133–138, IEEE Computer Society Press, 1994.

[40] Atmel Inc., ATMEL AT6000 data sheet, 1996.

[41] Xilinx Inc.(www.xilinx.com). *Virtex Series FPGAs.*

[42] J. M. Ditmar, "A Dynamically Reconfigurable FPGA-based Content Addressable Memory for IP Characterization," Master's thesis, KTH- Royal Institute of Technology, Stockholm, Sweden, 2000.

[43] V. C. Corporation. Reconfigurable Computing Products, http://www.vcc.com/.

[44] Xilinx Inc., "Virtex 2.5 V Field Programmable Gate Arrays", Advance Product Data Sheet, 1998.

[45] X. Inc., "Virtex-E 1.8 v field programmable gate arrays." Xilinx DS022, 2001.

[46] TMS320C62x / C67x CPU and Instruction Set Reference Guide Literature Number: SPRU189C March 1998.

[47] Sascha Uhrig, Stefan Maier, Georgi Kuzmanov, Theo Ungerer, "Coupling of a Reconfigurable Architecture and a Multithreaded Processor Core with Integrated Real-Time Scheduling", 2006 IEEE.

[48] Katherine Compton, Scott Hauck, "An Introduction to Reconfigurable Computing", *IEEE Computer*, April, 2000.

[49] Francisco Barat, Rudy Lauwereins, Geert Deconinck, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 28, NO. 9, SEPTEMBER 2002.

[50] K. Solomon Raju, M. V. Kartikeyan, R C Joshi and Chandra Shekhar, "Reconfigurable Computing Systems Design: Issues at System-Level Architectures".

Appendix

An Efficient Configuration Unit Design For VLIW Based Reconfigurable Processors

M.Aqeel Iqbal
Faculty of Engineering and IT
Foundation University, Institute of Engineering and Management Sciences, Rawalpindi, Pakistan
maqeeliqbal@hotmail.com

Uzma Saeed Awan
Department of Computer Sciences
International Islamic University, Islamabad
Police Line, Islamabad, Pakistan
uawan_80@hotmail.com

Abstract — **The reconfigurable processors are the leading platforms being under consideration as a role model for reconfigurable computing systems. An application can be greatly accelerated by placing its computationally intensive portions of algorithms onto the reconfigurable platform. The gains are realized because the reconfigurable computing combines the benefits of both; the software and the ASIC solutions. However, the advantages of reconfigurable computing do not come without a cost. By requiring multiple reconfigurations to complete a computation, the time required to reconfigure the hardware significantly degrades the performance of such systems. The emerging reconfigurable architectures are focusing the efficient solutions for the configuration unit designs. Configuration unit is responsible for managing all activities relevant to the system configuration and hence it plays a vital role in reconfigurable processors. In this research paper an efficient configuration unit design has been presented for a VLIW based reconfigurable processor. The presented configuration unit is expected to be one of the most efficient design alternatives being available for reconfigurable processors. The presented configuration unit design is capable of loading the minimum configuration streams with the most optimal configuration overheads and hence it leads to a dramatic enhancement in the performance of reconfigurable processor.**

Key Words — Configurable Logic Blocks, Field Programmable Gate Arrays, Multi-port Configuration Memory, Reconfigurable Processors, Reconfigurable Logic.

I. Reconfigurable Architectures

The architecture of a computing system often can affect its performance for a given application. Issues such as dedicated and non-dedicated resources, memory sizes and organizations, communication interfaces and instruction sets all affect the performance capability of computing systems. Reconfigurable processor is a combination of reconfigurable logic (like FPGAs) with a general-purpose microprocessor core (like standard CPU). The architectural goal is to achieve the higher performance than the typically available software-only solutions with more flexibility than the application specific integrated circuits (ASICs) as shown in the Fig.2. In reconfigurable processors, the microprocessor performs those operations that cannot be done efficiently in the reconfigurable logic such as loops, branches and possible memory accesses while computational cores are mapped to reconfigurable logic [4]. Performance of reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) now rivals that of the custom ASICs but with design flexibility not available in custom hardware. The role of FPGAs and reconfigurable processors include many scientific and signal processing applications.

The design of a reconfigurable processor can be divided in two main tasks. The first one is the interfacing between the microprocessor core and the reconfigurable logic. This includes all the issues related to how data is transferred to and from the reconfigurable logic, as well as synchronization between the two elements. The second task is the design of the reconfigurable logic itself. Granularity, Reconfigurability and Interconnections are the issues included in this task. The reconfigurable logic will provide hardware specialization to the application being under execution. It will provide similar benefits to those offered by Application Specific Instruction Set Processors (ASIPs). ASIPs have specialized hardware that accelerates the execution of the applications it was designed for. A reconfigurable processor would have this same benefit but without having to commit the hardware into silicon. Reconfigurable processors can be adapted after design, in the same way as that of programmable processors can adapt to application changes.

Different coupling approaches for the reconfigurable core being used in the reconfigurable systems include; as a Functional Unit Coupling, as a Co-processing Unit Coupling, as an Attached Processing Unit Coupling and as a Standalone Processing Unit Coupling as shown in Fig.1. Many of recent computationally intensive applications can benefit from the speed offered by application specific hardware co-processors (ASIC or ASIP), but for applications with multiple specialized needs, it is not feasible to have a different co-processor for every specialized function. Such diverse applications stand to benefit from the flexibility of reconfigurable computing architectures since one reconfigurable computing unit can provide functionality of several ASIC or ASIP Co-processors. Many research groups have demonstrated the successful launch of reconfigurable computing architectures [3]. Another area in which the reconfigurable devices are becoming more popular is the Systems on Chip (SoC) technology. Known as Systems on a Programmable Chip (SoPC), the Xilinx [5], the Altera [6] and other venders have developed programmable devices which give the flexibility to application user to include

978-1-4244-2824-3/08/$25.00 ©2008 IEEE

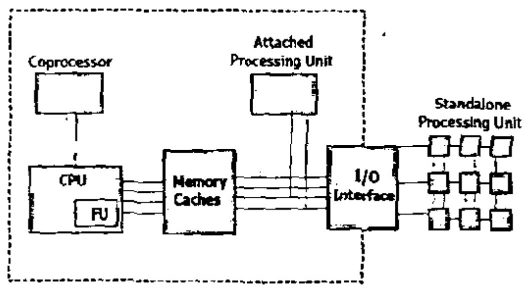


Fig. 1 Coupling Approaches for Reconfigurable Logic

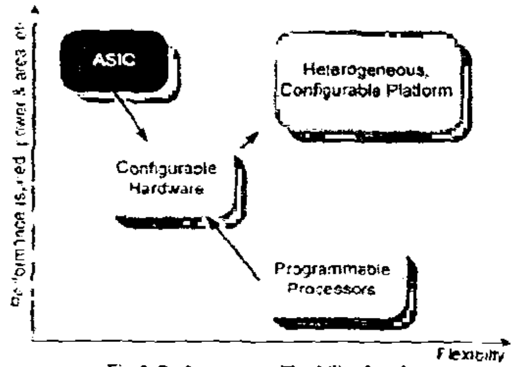

ASIC
Heterogeneous, Configurable Platform
Configurable Hardware
Programmable Processors
Performance is, red, power & area of
Flexibility

Fig. 2 Performance vs Flexibility Graph

the user reconfigurable area in addition to the sophisticated intellectual property cores, embedded processors, memory and other complex logic all on same chip.

FPGAs or FPGA-like devices are most common hardwares being used for reconfigurable computing. FPGA contains an array of the computational elements whose functionality is determined through multiple SRAM based configuration bit streams. These elements, also known as configurable logic blocks (CLBs), are connected using a set of routing resources that is also programmable. In this way, the custom circuits can be mapped to the FPGA by computing the logic functions of the circuit within the CLBs and then using the configurable routing to connect the blocks to form the necessary circuit. Although the logic capacity of FPGAs is lower than that of ASICs because of the area overhead for providing undedicated logic and routing, FPGAs provide significantly higher flexibility than the ASICs, while still offering a considerable speedup over general purpose systems as shown in Fig.2. In addition, the run-time reconfigurability provided by advanced FPGAs like Xilinx Virtex series has greatly improved the hardware utilization [5].

II. Related Research Work

A large number of reconfigurable architectures have been proposed in the last few decades. The previously proposed reconfigurable architectures generally fit into one of two major categories depending on the grain of computations they map onto reconfigurable logic. Fine-grained Reconfigurable Architectures, such as CHIMERAE [7] integrate the small blocks of the reconfigurable logic into superscalar processor architectures, treating reconfigurable logic as programmable ALUs that can be configured to implement the application specific instructions. These systems can achieve the better performance than conventional superscalar processors on a wide range of applications by mapping commonly-executed sequences of instructions onto their reconfigurable units, but the maximum speedup they can achieve is limited by the small amount of logic in their reconfigurable units. Coarse-grained Reconfigurable Architectures, such as PipeRench [8] provide larger blocks of reconfigurable logic that are less tightly-coupled with programmable portions of the processor. These architectures can achieve extremely good performance on applications that contain long-running nested loops that can be mapped onto the processor's reconfigurable logics but perform less well on applications that require frequent communication between programmable and reconfigurable portions of the processor.

III. Proposed Architecture

The performance of the reconfigurable processor is mainly dependant on the time overhead required by it to configure its reconfigurable function units (RFUs). Normally it has been observed that this configuration overhead negatively hits to the computational speed of any reconfigurable processor [1]. Hence researchers are now focusing the issue of configuration overhead minimization for reconfigurable processors [1]. In this regard many software and hardware based solutions have been proposed which include the Configuration Compression, the Configuration Caching, the Configuration Cloning, Partial Configuration, the Run-time Configuration [1], Multi-threaded Configuration, the Bit Parallel Configuration, the Intelligent Configuration [2] and the Optimal Configuration [1] etc.

In order to minimize the configuration overheads for the reconfigurable processors; an efficient hybrid design has been proposed for configuration unit of a typical VLIW based reconfigurable processor. The proposed design includes both the hardwired and the programmable logic modules. The reconfigurable processor being targeted in this research paper is a VLIW processor having a very long instruction word of eight instructions where each instruction is 32-bits instruction. Configuration unit plays a vital role in the performance enhancement of the reconfigurable processor. Hence in such a type of processor there is an extra hardware unit being known as configuration unit along with standard micro-programmed control unit whose job is to manage the configuration activities of the reconfigurable processor. The location and the interconnections of the configuration unit inside a typical VLIW based reconfigurable processor have been shown in Fig.3. Few aspects of design are described below.

A. Instruction Formats Encoding

In case of a standard programmable micro-processor the instruction format is mainly composed of an Op-code and an

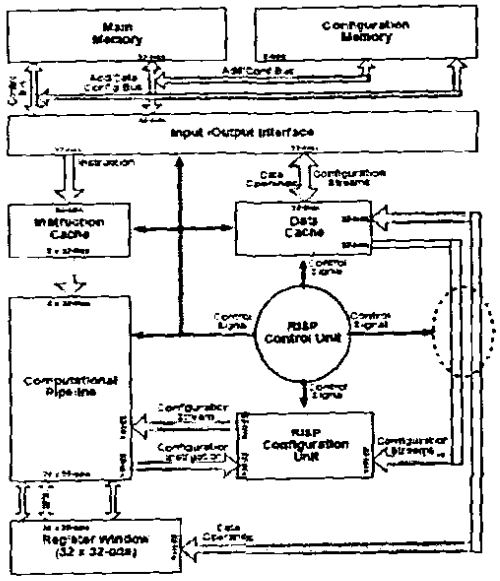


Fig 3 Typical VLIW Based RISP Architecture

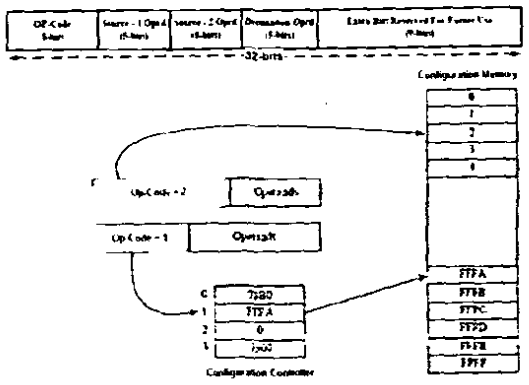


Fig. 4 Typical RISP Instruction Format

Operand addresses field. The Op-code of the instruction format defines the nature of operation to be performed by the instruction while the operand addresses field defines the source and destination addresses for data operands for this operation. In a similar way the instruction format of reconfigurable processor is consisting of an Op-code and its operand addresses field but the Op-code of the instruction is being mapped or converted into an address of either the configuration table entry which contains the effective addresses of concerned configuration streams in configuration memory or it is itself an effective address of the configuration streams in the multi-port configuration memory. Configuration memory being used is a multi-port random access memory which contains a set a configuration streams that are required by the RFUs of reconfigurable processor to configure the relevant hardware modules like Adders, Subtractors, Multipliers and Shifters etc. Consider the Fig.4 for instruction format of a typical reconfigurable processor.

B. Computational Pipeline Design

The proposed reconfigurable processor is a high speed VLIW based design using an intensive pipelined architecture. The computation pipeline contains a Fetch Unit (FU), Schedule Unit (SU), Dispatch Unit (DU), Execution Unit (EU) and Register Window (RW) as shown in Fig.5.

The FU of the pipeline is responsible to fetch a packet (Long Word) of eight instructions where each instruction is a 32-bits instruction. The FU is a State Machine (Mealy Machine or Moore Machine) based module. It fetches a long word from the instruction cache of processor and loads it into the SU of the pipeline.

The SU of the pipeline is responsible for scheduling the received long word from the FU. SU loads the long word into the DU of the pipeline and Op-codes of all instructions of the long word are also transferred towards the configuration unit of the processor which updates the RFUs configurations and accordingly sends the dispatch signals to DU so that the instruction can be dispatched to their relevant RFUs.

The DU of the pipeline is responsible for dispatching the instructions of long word into their relevant RFUs inside the EU for execution.

The DU contains a layer of eight De-multiplexers whose control signals are received from the configuration unit of processor. Each De-multiplexer is a 1 x 8 DMUX of 32-bits size. It transfers one instruction of the long word to one of the eight RFUs which has been reconfigured for it by the configuration unit.

The EU of the pipeline is responsible for the execution of the instructions of the long word. The EU contains a layer of eight RFUs. Each RFU has been integrated with an FPGA core like provided by the Xilinx Corporation and a layer of common data buses. The FPGA core is configured by the configuration unit of the processor according to the execution requirements of running application program. The proposed design is a Register-Register Architecture in which the source operands required by each instruction are fetched from the register window of the processor and similarly the results generated after the execution of each instruction are stored back temporarily to same register window of processor.

The register window of pipeline is responsible for providing the source data operands for the execution of eight instructions of the long word and temporarily storing their results. The register window contains a layer of thirty two registers where each register is a 32-bits register.

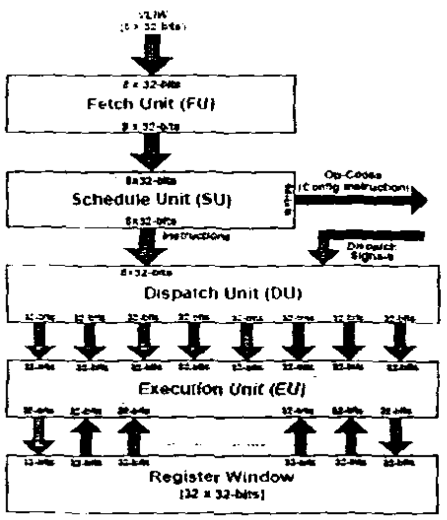


Fig. 5 Typical RISP Computational Pipeline

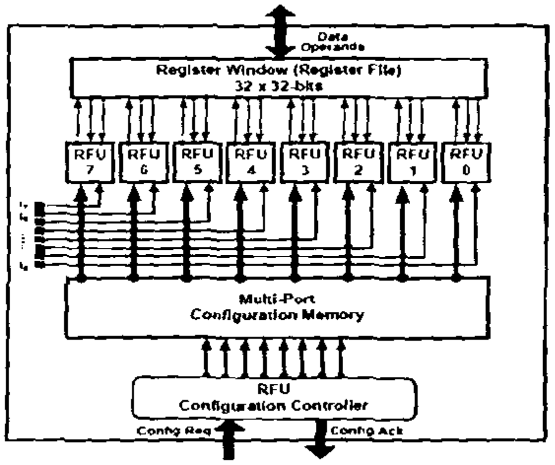


Fig. 6 Configuration Unit Interfaces

C. *Configuration Unit Design*

The main job of the configuration unit is to update the loaded configurations of the RFUs according to the changing requirements of the running application. The RFUs have been integrated with the high speed partially reconfigurable FPGA cores like those provided by the Xilinx Virtex series of FPGAs. RFUs make the actual execution layer of the EU being available inside the computational pipeline. The reconfigurable systems mostly require a lot of time to perform this configuration update process. Hence this configuration overhead greatly degrades the performance of such systems.

In order to optimize the configuration overhead of a typical reconfigurable processor based on VLIW architecture; a unique idea of *RFU Partial Configuration* has been introduced. In this technique the configuration unit constantly keeps on monitoring the currently loaded configurations and newly demanded configurations. It maps the newly demanded configurations on the currently available configurations and maximizes the reusability of the already available configurations and loads only those configurations which are no more currently available in any RFU of the reconfigurable processor.

The configuration unit contains a layer of *Op-code Map Logic (OML)* which actually compares each Op-code of the incoming long word with all Op-codes of the currently configured long word in the RFUs. This mapping process is performed concurrently with a high speed ASIC circuit which contains a parallel network of comparators as shown in Fig.7 and in Fig.8. All those op-codes who have been compared with any of the existing op-codes are then allocated their respective RFU no, where they will be executed. If an op-code has not been compared with any one of the existing op-codes then it is not allocated any RFU no. This information of RFU allocation or not allocation is sent to two programmable logic controllers. One is known as the *Configuration Memory Controller (CMC)* and is responsible to generate the RFU Configuration addresses for only those RFUs who really need configuration updation and at the same time it calculates and sends the sufficient control signals to DU of computational pipeline. On the basis of these signals the DU dispatches the instructions into their relevant RFUs. Second is known as the *Op-codes Memory Update Logic (OMUL)* and is responsible to update the op-codes memory contents according to the newly arrived op-codes of the long word. The time taken by the Map Logic to compare the all incoming op-codes with the all existing op-codes is always constant and is equal to 1-Cycle. But the time taken by the CMC and OMUL are variable and are dependent on the no of the newly arrived op-codes that are not matched with the existing op-codes and it may vary from 0-Cycles to 8-Cycles. If all op-codes are matched then its latency is 0-Cycles and if none of them is matched with the existing op-codes then its latency will be 8-Cycles and so on. For those applications where the same operation is repeated again and again like the operation of convolution in conventional DSPs; they will always be given 0-Cycle latency and hence it dramatically enhances the computation speed of the system by minimizing the configuration overhead to 0-Cycles. Such kind of drastic performance revolutions that have been observed are shown in the performance graph of RISP in Fig.9 which have been obtain by using the proposed configuration unit in a typical VLIW based reconfigurable processor and benchmarking its performance with a DSP (TMS320C6X). The configuration unit has a RFU Configuration Controller and a Multi-port Configuration Memory as shown in Fig.6. RFU configuration controller is responsible for providing optimal configuration overhead. The multi-port configuration memory contains a set of most frequently used configurations that can be dynamically changed during the execution of the application by loading them externally through Configuration EPROM.

IV. Performance Analysis Equation

Following is the mathematical equation being formulated for the calculations of the total no of cycles (T_{Total}), consumed for the updation of the RFU configurations for each of the VLIW packet. Consider the equation parameters in Table.1

$$T_{Total} = T_{OML} + \sum T_{OMU} + \sum T_{CMC} + T_{CNF}$$

V. Research Areas In Active Domain

There are many dimensions of reconfigurable computing being under active research work. The following topics outline the different aspects of reconfigurable computing that research has been addressing in the past several years:

A. Reconfigurable Architectures

Device and system architectures are being developed which propose the various ways of organizing and interfacing the configurable logic. Some reconfigurable architectures are based on coarse grain functional units that are configured on the fly to execute an operation from a given set of operations. Commercial architectures are exploring integration of reconfigurable logic and microprocessors on the same chip.

B. Reconfigurable Applications

Specialized configurable architectures, which are utilized for speeding up specific applications, are replacing some ASICs. Some applications also exploit optimization based on a specific input instance of the computation.

C. Algorithmic Synthesis

Dynamically reconfigurable architectures give rise to new classes of problems in mapping computations onto the architectures. New algorithmic techniques are needed to schedule the computations. Existing algorithmic mapping techniques focus primarily on loops in general purpose programs. Loop structures provide repetitive computations, scope for pipelining and parallelization are candidates for mapping to reconfigurable hardware.

TABLE.I ANALYSIS EQUATION PARAMETERS

Parameter Name	Parameter Description	Possible Values
T_{OML}	Time required to map new op-codes with all of the existing op-codes	1 Cycle
T_{OMU}	Time required to update the op-codes memory for new op-codes	0, 1, 2 ...8 Cycles
T_{CMC}	Time required to generate the RFU config. addresses and dispatch unit signals	0, 1, 2 ...8 Cycles
T_{CNF}	Time required to configure all RFUs for each of new requirements	0, 1 Cycle
T_{Total}	Total time required to update the configuration of all RFUs of Execution Unit	N Cycles

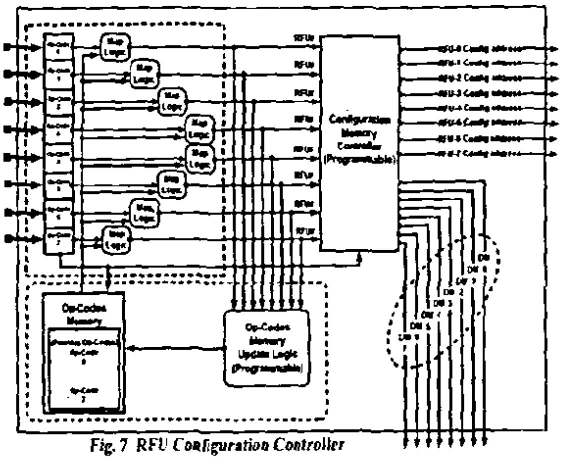

Configuration Memory Controller (Programmable)
Op-Codes Memory
Op-Codes Memory Update Logic (Programmable)

Fig. 7 RFU Configuration Controller

D. Software Tools

Current software tools still rely on CAD based mapping techniques. But there are several tools being developed to address run-time reconfiguration, compilation from high-level languages such as C, simulation of dynamically reconfigurable logic in software and complete operating system for dynamically reconfigurable platforms. There is a significant lack of research in development of models of reconfigurable architectures that can be utilized for developing a formal framework for mapping applications. The Reconfigurable Mesh model was the earliest theoretical model that addressed dynamic reconfiguration in computation and communication structure. However, Reconfigurable Mesh model is more theoretical and hardware implementations have only been able to approximate the delay and speed assumptions in the model. There have been several research efforts that focused on developing architectures and the associated software tools for mapping onto their specific architecture. Some of these projects have addressed generic mapping techniques that can be extended to a class of the reconfigurable architectures. Such projects include Garp [9], PipeRench [8] and SPLASH [10].

Customizing the configurable hardware to suit the computations has been acknowledged as the most significant advantage of such architectures. Some researchers have adapted the hardware to perform computations with exactly the required precision for the computations. Such static approaches do not exploit the ability of configurable hardware to be adapted to the exact required precision as the computations progress. The maximum possible precision of variables, which is determined in the static approach, can still involve execution with superfluous precision and unnecessary overheads. Several efforts have also focused on developing parameterized libraries and components, precision being one of the parameters. Most FPGA device vendors provide such highly optimized parameterized libraries for their architectures. Efforts have also been made to generate such modules using the high-level descriptions.

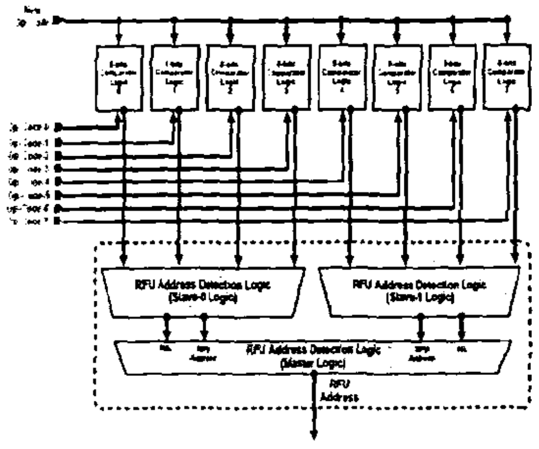

RFU Address Detection Logic (Slave-0 Logic)
RFU Address Detection Logic (Slave-1 Logic)
RFU Address Detection Logic (Master Logic)
RFU Address

Fig. 8 Map Logic

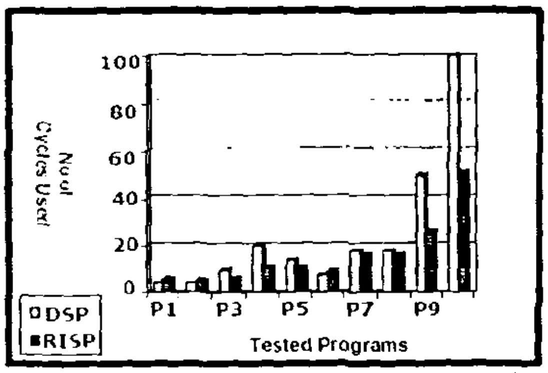


Fig. 9 Performance of RISP Using Proposed Configuration Unit

E. Simulation Tools

Several simulation tools have been developed for the reprogrammable FPGAs. Most of the tools are device based simulators and are not system level simulators. The most significant effort in this area has been the Dynamic Circuit Switching (DCS) based simulation tools. These tools study the dynamically reconfigurable behavior of FPGAs and are integrated into the CAD framework. Though the simulation tools can analyze the dynamic circuit behavior of FPGAs, the tools are still low level.

VI. Conclusion

In domain of reconfigurable computing the reconfigurable processors are becoming an important part of research due to their ability to exhibit the high performance of ASICs and flexibility of programmable processors. The performance of such a processor is greatly dependent on the configuration overhead required by it to provide the flexibility of hardware design. In order to provide the most optimal configuration overhead for these processors, an efficient configuration unit design has been proposed which always tries to optimize the configuration overhead by loading the minimum possible configuration bit streams. The proposed configuration unit always analyzes the configuration requirements of the application being under execution and loads only those configurations which are not currently available in the RFUs and those which are available are reused as many times as needed. The performance band of computing can be greatly enhanced by using the reconfigurable processors which will be integrated with such kind of efficient configuration units.

References

[1] M.Aqeel Iqbal, Shoab A. Khan and Uzma Saeed Awan, 'RISP Design with Most Optimal Configuration Overhead for VLIW Based Architectures', Proceedings of IEEE computer society, 2nd IEEE ICEE-2008 Conference, March 25-26, 2008, UET Lahore.

[2] Aziz-Ur-Rehman, Dr. Aqeel A. Syed and M. Aqeel Iqbal, 'Intelligent Reconfigurable Instruction Set Processor (IRISP) Design', Proceedings of IEEE computer society, 11th IEEE INMIC-2007 Conference, Dec 28-30, 2007,COMSATS Lahore.

[3] Leong, P. H. W., Leong, M. P., Cheung, O. Y. H., Tung, T., Kwok, C. M., Wong, M. Y., and Lee, K. H., "Pilchard - A Reconfigurable Computing Platform With Memory Slot Interface," Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, California USA, IEEE.

[4] Compton, K. and Hauck, S., "Configurable Computing: A Survey of Systems and Software," Northwestern University, Dept. of ECE Technical Report, 1999, Northwestern University.

[5] Xilinx, Virtex Series FPGAs, http://www.xilinx.com, 2001.

[6] Altera: Systems on a Programmable Chip, 2001.

[7] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," Proceedings of the 27th International Symposium on Computer Architecture, pp. 225-235, 2000.

[8] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R.Taylor, and R. Laufer. "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", in Proc. Intl. Symp. on Computer Architecture, May 1999.

[9] J.R. Hauser, and J. Wawrzynek, Garp: A MIPS Processor with a Reconfigurable Coprocessor, Proc. IEEE Symp. FCCM, April 1997, pp.12-21.

[10] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. Splash 2: FPGAs in a Custom Computing Machine. IEEE Computer Society Press, 1996.

:009 IEEE International Advance Computing Conference (IACC 2009)
'atiala, India, 6–7 March 2009

Reconfigurable Processor Architecture For High Speed Applications

[1]M.Aqeel Iqbal
Faculty of Engineering and IT
Foundation University, Institute of Engineering and
Management Sciences, Rawalpindi, Pakistan
maqeeliqbal@hotmail.com

[2]Uzma Saeed Awan
Department of Computer Sciences
International Islamic University, Islamabad
Police Line, Islamabad, Pakistan
uawan_80@hotmail.com

Abstract — ***Revolutions in the domain of computing have molded the structures and characteristics of computing systems. Conventional computing techniques involved the use of application specific integrated circuits to achieve a high performance at the cost of extremely inflexible hardware design meanwhile the flexibility of hardware design was achieved at the cost of slow speed processing by using programmable processors. The emergence of reconfigurable computing has filled the gap between the flexibility and performance of system. Reconfigurable computing combines the high speed of application specific integrated circuits with the flexibility of the programmable processors. The reconfigurable processors have further boosted up the dramatic nature of reconfigurable computing systems. These processors configure the most optimal and efficient hardware resources according to the demands of running application. The configured hardware resources can be modified or reconfigured later on according to the new demands of the running application. In this research paper reconfigurable processor architecture has been presented for high speed applications. The proposed reconfigurable processor is based on very long instruction word architecture. The proposed processor is using an efficient multi-threaded configuration controller and a multi-ported configuration memory to configure the multiple reconfigurable function units concurrently with minimum possible configuration overhead.***

Keywords — Configurations, Configuration Overheads, Reconfigurable Computing, Reconfigurable Functional Units.

I. INTRODUCTION

Reconfigurable processor is based on a reconfigurable functional unit (RFU) being integrated inside the processors as shown in Fig. 1. RFU is composed of many computational elements whose functionality can be determined through the programmable configuration bit streams. Reconfigurable computing is introduced to fill the gap between hardware and software based systems. The goal is to achieve the performance better than that of software based solutions while maintaining the greater flexibility than that of the hardware based solutions as shown in Fig. 2. Reconfigurable computing is an alternative of the superscalar and VLIW paradigms. The main distinction between a reconfigurable processor and a standard processor is in the instruction stream. In its purest form, a reconfigurable processor has no cycle-by-cycle instruction stream. Rather, the processor is configured by loading a complete specification of the function of each part of it at once. Once configured, the intention is for the processor to run in that configuration for a decent interval before being reconfigured. Each configuration responses an ASIC-like circuit, like that specialized for the particular task at hand. Changing configurations might take from a few clock cycles to a few thousand clock cycles. In accordance with the simpler programming mechanism, the dynamic forwarding crossbar is replaced by a less flexible configurable network for making static connections among the functional units and short queues of retiming registers associated with each functional unit take the place of the traditional processor's shared, multi-ported register file.

The most fascinating and familiar 90-10 rule asserts that 90% of execution time is some times consumed by about 10% of a program's code and that 10% is generally comprising of inner loops. Reconfigurable processors excel in those cases where the computation represented by a configuration is repeated many times and so the time required to load a configuration can be amortized over a long execution time and/or overlapped with other executions. When all of an application's important loop bodies can be configured to fit within the reconfigurable processor (one at a time), there would seem to be no need for the overhead of a fully dynamic instruction fetch and issue mechanism, allowing the processor to be leaner and more efficient. By reducing the hardware to just the essentials needed to support computation, the reconfigurable processor scales better to larger sizes than the more complex superscalar and VLIW based systems. Although a native expansion of the configurable network would cause it to grow quadratically with the number of functional units, it only needs to grow enough to support the connectivity required by the real applications. Furthermore, unlike a superscalar or VLIW processor, the reconfigurable processor can easily exploit not only simple instruction level parallelism but also inter-iteration and thread parallelism, making reconfigurable computing well poised to work with a large number of functional units.

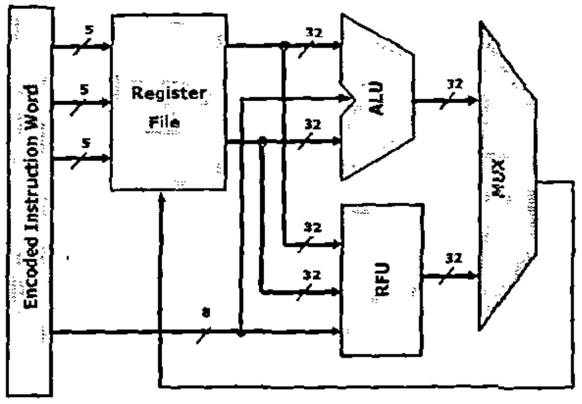


Fig. 1 Reconfigurable Processor Data-path

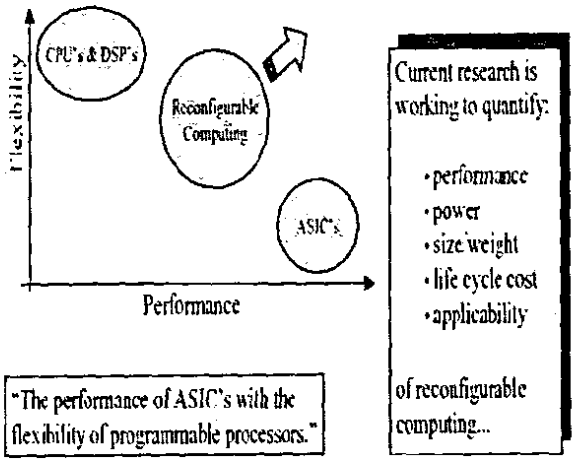

Flexibility
CPU's & DSP's
Reconfigurable Computing
ASIC's
Performance
Current research is working to quantify:
•performance
•power
•size weight
•life cycle cost
•applicability
of reconfigurable computing...
"The performance of ASIC's with the flexibility of programmable processors."

Fig. 2 Performance vs Flexibility

II. Research Work In Active Domain

A large number of reconfigurable computing systems have been proposed with different design objectives, parameters, methodologies and implementations but they do share the same design framework. Reconfigurable functional units (RFUs) with the configurable interconnects are the foundation of a reconfigurable computing platform. Various configuration combinations can define numerous possible functionalities. Design implementations of a reconfigurable processing unit can be a simple microprocessor or even a gate level operator such as lookup tables being available inside the CLBs of most of currently available SRAM based field programmable gate arrays (FPGAs). Interconnects or routing networks in different reconfigurable systems have different structures as well, such as mesh, linear and crossbar structures.

For reconfigurable systems a compiler based software tool is required to map an application onto the reconfigurable core. This application is expressed in the form of configuration bits used to define the operation of each CLB and inter-connect. This compilation tool can be as simple as an assisting tool that helps a programmer to perform manual circuit mapping and can be as complex as a fully automated system that can deal with all configuration works by itself. The configurable nature of a reconfigurable system allows the hardware to be programmed with new sets of configurations to support new operations. Depending on the nature of the architecture, some systems can only be reprogrammed in non-executing state and are commonly known as simple reconfigurable systems while some may support dynamic reconfiguration at the run time by allowing an operation to be altered during execution and are commonly known as run-time reconfigurable systems. The reconfiguration process latency will also vary from system to system and from hardware to hardware to be reconfigured. Recently a large number of reconfigurable computing systems are available in the market; as well many still undergo research. Followings are the some of well known examples:

A- MIT Raw:
The MIT Raw [1] is based on a mesh structure of interconnected simple RISC processors cores. Its basic design goal is to benefit the parallel execution of applications from multiple microprocessors at a coarse-grained environment. The static communication network in the architecture makes good use of pre-defined communication pattern at compile time and reduces network latency by well ahead preparation. This architecture can provide great flexibility and processing power beyond that of a single processor. Raw can perform well with random programs but its performance is much better with parallel applications. However, high power consumption will result from the execution of multiple processors, which is a big drawback of the architecture.

B- CMU PipeRench:
The CMU PipeRench [3] is hardware based computing solution being specialized for pipeline based applications. Run-time reconfiguration of hardware modules is used to execute a large sized application using small amount of hardware resources. The efficient architecture and simple implementation of the design dissipates less than one watt of average power while achieving good performance. This architecture is a perfect candidate for pipeline based applications because of its highly specialized design, small area consumption and low power implementation.

C- Xputer:
The Xputer [6] is a computing design suggested to use data driven control instead of instruction sequence control as in conventional computers. Its basic aim is to avoid data latency and data dependency problems by executing in the order of data accessing sequence. The applications with regular data patterns such as multimedia, streaming and encryption applications can fit well with Xputer design.

D- NEC DRP:
The NEC DRP [7] is a coarse-grained reconfigurable system. The system composes of many small processing elements for computations where repository of contexts is stored on-chip. By choosing a different context, the chip will implement a different data-path to represent a new operation or algorithm. This feature enables the dynamic run-time reconfiguration in a single clock cycle. Applications such as networks, image processing and signal processing work well with the parallel processing environment and fast run-time reconfiguration for any dynamic events.

E- NASA Evolvable Hardware:
The NASA Evolvable Hardware [8] is reconfigurable hardware with the configuration process working under the control of a genetic algorithm. In evolutionary synthesis of analog and digital circuits, a hardware circuit evolves to realize a design specification dynamically at run time without the need of any pre-defined information. The ultimate goal of this research is to develop an architecture that can adapt to any possible environment without any human control. Hence the theme of the design is to provide an evolvable intelligent machine that can be used to perform work independently in environment such as space exploration. Negatively hurting

parameters of design are the resource demanding and time consuming evolution process.

F- IPFlex DAPDNA:

The IPFlex DAPDNA [9] is basically a dual-core processor including a RISC core coupled with a two-dimensional processing matrix. The two-dimensional processing matrix is a reconfigurable core. The reconfiguration of the processing matrix is controlled by the RISC core to support different operations to achieve parallel processing efficiently. The system has shown a dramatic performance gain for multi-threaded applications.

G- MathStar FPOA:

The MathStar FPOA or Field Programmable Object Array [10] system is an enhanced FPGA based solution. Instead of using CLBs or lookup tables as elementary cell in the device, FPOA uses its own building blocks as foundations. Having pre-defined block types allow the blocks to achieve higher performance gain, less area consumption and a better communication with other working blocks. PipeRench is a hardware based pipelined architecture with great flexibility, while Raw is software based general purpose processor approach with enhanced parallelism. The two systems are very representative to the two extremes of design. NEC DRP, IPFlex DAPDNA, and MathStar FPOA are commercial products and are FPGA based solutions with higher granularity and advanced features.

H- Chimerae:

The Chimerae [4] is a fine-grain architecture which integrates the small blocks of reconfigurable logic into superscalar processor architectures, treating the reconfigurable logic as programmable ALUs that can be configured to implement application-specific instructions. These systems can achieve the better performance than the conventional superscalar processors on a wide range of applications by mapping the commonly executed sequences of instructions onto their reconfigurable units, but the maximum speedup they can achieve is limited by the small amount of logic in their reconfigurable units.

I- Remarc:

The Remarc [2] is a coarse-grain architecture which provides larger blocks of reconfigurable logic that are less tightly-coupled with the programmable portions of the processor. These architectures can achieve extremely good performance on applications that contain long-running active nested loops that can be mapped onto the processor's reconfigurable arrays but perform less well on applications that require frequent communication between programmable and reconfigurable portions of the processor. Systems such as Pilchard that integrates FPGAs into conventional workstations over the processor's memory bus display similar behavior, although the relatively low bandwidth of a microprocessor's memory bus makes them even more sensitive to the amount of the communication that an application requires between the processor and the FPGA.

III. Proposed Processor Architecture

In this section the detailed architecture of the proposed reconfigurable processor has been discussed. The detailed architecture of the proposed processor is shown in Fig. 3 and the different modules are discussed below along with their functionality.

A- Input/Output Interface (IOI):

The IO interface of processor is used to communicate with the external devices being interfaced with it. The first job of the I/O Interface is to load the configuration streams from external Configuration EPROM or main memory of system during the booting processes of the processor and it takes only a few clock cycles. These configuration streams contain the different hardware modules like Adders, Subtractors, Multipliers and Shifters etc. The second job of the IO interface is to load the instructions and their relevant data operands to be executed on the processor. The third job of the IO interface is to store the results of the computations performed on the processor in main memory of the system. The fourth job of the IO interface is to send and receive the control signals generated and acknowledged by the control unit of the processor to the external devices.

B- Prefetch Unit (PFU):

The basic job of the PFU is to fetch or pre-fetch the instruction stream and the data stream of the application program being under execution. Fetched instructions are loaded in the *Instruction Pool* and then transferred into the *Instruction Cache*. Similarly the data stream is loaded into the *Data Cache*. Consider the Fig. 4 for instruction format encoding of the proposed processor.

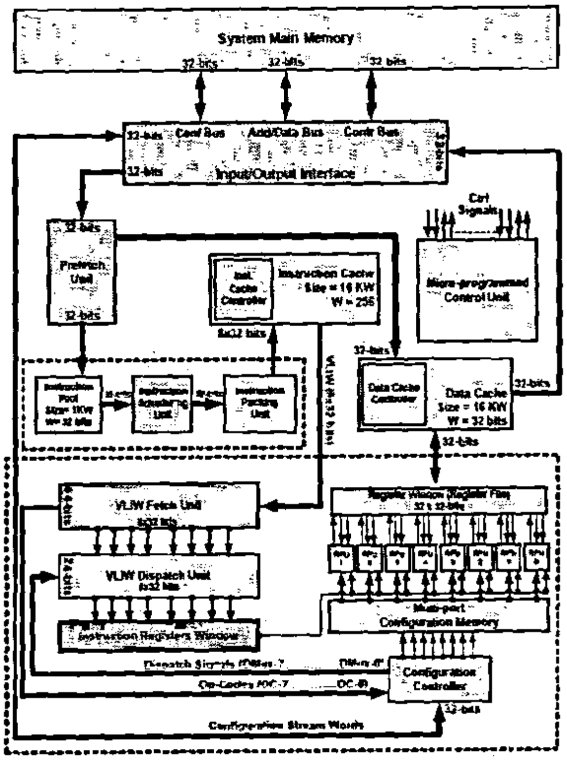


Fig. 3. Proposed Processor Architecture

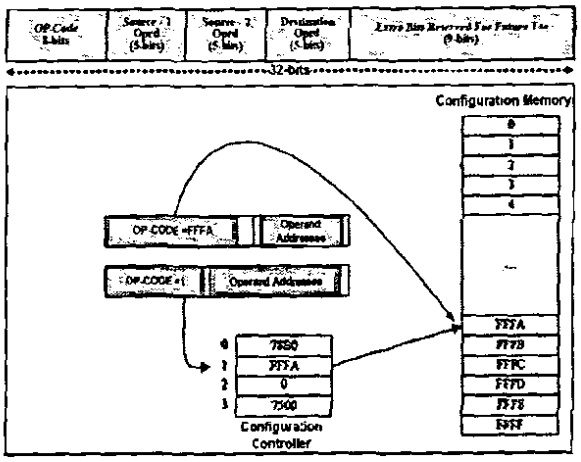


Fig. 4 Processor Instruction Format Encoding

C- Instruction Scheduling Unit (ISU):
The ISU is the micro-programmed implementation of the *Tomasoulo's Algorithm* being used in VLIW and Super-scalar processors for the scheduling of the instructions. The instruction scheduler reads instructions from the instruction pool and then it analyzes them for dependencies (if any) and resolves these dependencies. Dependencies being analyzed include Data Dependency, Control Dependency, Resource Conflicts and Data Hazards etc. Then it after analysis ISU transfers these instructions to instruction packing unit (IPU).

D- Instruction Packing Unit (IPU):
The main job of the IPU is to pack the eight instructions in the form of a VLIW. The 32-bits instructions transferred from the ISU are given to the IPU. The IPU arranges these instructions in a buffer in a FIFO order on their arrival from the ISU. After the arrival of each instruction, the IPU increments its instruction counter and checks either there are eight instruction arrived from the ISU or not. If a total of eight instructions have been arrived from the ISU then the IPU transfers them into a VLIW buffer of size 8 x 32-bits. Then it enables this buffer to transfers this VLIW to instruction cache of the processor if signal Load_VLIW =1. The same process is repeated constantly throughout the application execution.

E- VLIW Fetch Unit (VFU):
VFU is a state machine based unit and works like a programmable counter. VFU fetches VLIW from the instruction cache and the *Op-Codes* of all instructions of the VLIW are transferred to the *Configuration Management Unit* and the VLIW itself is transferred to VDU.

F- VLIW Dispatch Unit (VDU):
VDU is consisting of an array of eight De-MUXs whose select lines are controlled by the configuration controller. According to the select lines activated by the configuration controller all of the instructions of VLIW are dispatched by VDU to their relevant RFUs.

G- VLIW Execution Unit (VEU):
VEU is the core component of the processor because it contains an array of RFUs being used for program execution. Consider the Fig. 5 of VEU. The VEU contains the following major modules.

a) External IO Logic (EIOL):
The EIOL of the VEU is used to load instructions in the instruction register, source operands in general-purpose registers and the configuration stream in RFUs. The second job of the EIOL is to store the configuration stream being loaded in the RFUs for the analysis purpose and results being generated after the execution of VLIW. The source operands Sr-1and Sr-2 are loaded into the internal general-purpose registers (GPRs) by the External De-MUX of size 1 x 24. The address given for the Data-in is connected to the select lines of De-MUX as well as to Decoder (5 x 24) input. De-MUX selects one of the general-purpose registers for data loading and the decoder enables its output channel connecting to the registers through the MUX of the size 2 x1. This MUX receives 32-bits data operand from External De-MUX at input "1" and receives 32-bits results from RFUs at the input "0". If the Ext_IO_En=0 then it selects the result coming from the RFUs and loads it in the register. If the Ext_IO_En=1 then it selects the data coming from the External De-MUX and loads it in the registers. Since there are eight RFUs that can load their results in the same register, hence in order to solve this problem an 8 x 1 MUX (32-bits) is interfaced with each register input. Each MUX is controlled by the *RFU Data-path Controller* which analyzes the Destination Addresses of all the RFUs and selects only that RFU whose output is valid output. In order to store the results and the flags being available in the GPRs and flag registers (FRs) into the data cache of the RISP, the 32 x 1 External MUX (32-bits) is used which can read the contents of the selected register.

b) RFUs Data-in / Data-out Logic (RDIOL):
In order to load/store the data across the RFUs there are two 32 x 1 MUXs (32-bits) and one 1 x 24 De-MUX (32-bits) for each RFU. Using the two MUXs the RFU is able to read the source data operands (Sr-1 and Sr-2) from any one of the 32 registers and using the one De-MUX it stores its results back to any one of the GPRs. Flags generated during the execution of the VLIW are loaded into the relevant FRs.

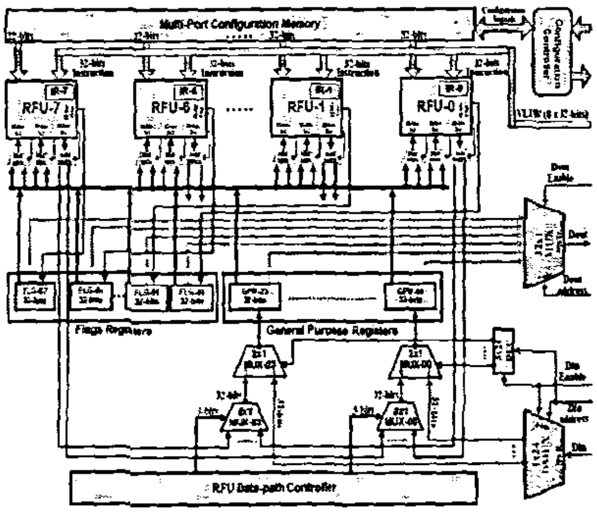


Fig. 5 Processor Execution Unit

c) General-Purpose and Flag Registers (GFRs):
There is an array of eight FRs (32-bits) and twenty four GPRs (32-bits). GPRs can be read and written by the programmer but the FRs can only be read by the programmer and can not be written. RFUs can read/write any one of these thirty two registers. More than one RFU can read the contents of the same register at the same time but only one RFU can write in a register at the same time.

d) Registers Input/Output Logic (RIOL):
FRs are loaded with the flags, being generated by the RFUs and can be read by the programmer through the External MUX. In case of the GPRs, the programmer can read the registers through the External MUX but in order to write contents into registers there is a 2 x 1 MUX (32-bits) which selects the data for the register either from some RFU output or from data cache. The 8 x 1 MUX interfaced at the input of the 2 x 1 MUX selects the valid RFU for the results to be stored in the register. In order to select the valid RFU for results, there is a RFU Data path Controller as shown in Fig. 5 is attached with all MUXs. This controller reads the select lines of all the De-MUXs of RFUs and after analysis it selects that RFU whose output is a valid output.

e) Reconfigurable Functional Units (RFUs):
RFUs are the computational units of processor and can be reconfigured at any time according to the application demand. They have been tightly coupled in the form of an integrated FPGA core.

H- Configuration Management Unit (CMU):
CMU is composed of a *Configuration Controller* and a *Multi-port Configuration Memory* as shown in Fig.5. Configuration controller as shown in Fig. 6 receives the op-codes of the eight instructions of the VLIW from the VFU and on the basis of these op-codes it decides to load one of the configuration blocks available in the memory for each RFU (if required). Also it checks if the op-code is a No Operation (NOP) or is same as that of any one of the existing op-codes. If so then the configuration controller does not load this new configuration into the RFUs but the hardware that is already loaded in the RFUs is reused and hence the configuration time that was required for the reconfiguration of RFUs is saved. Hence only those RFUs are reconfigured that are quite new ones. Hence the processor always takes the minimum possible time to reconfigure the RFUs during the execution of the application program and always has the most optimal configuration overhead. A micro-programmed control unit has been used to work like a control unit of processor.

IV. Performance Analysis Model

Following is the mathematical model being formulated for the calculations of the total no of cycles (T_{Total}) consumed by proposed reconfigurable processor for the execution of an application. Consider the Table.1 for the model parameters.

$$T_{Total} = \{(N_{IN} + N_{NO})/8\} (T_{VFT} + T_{OFT}) + E_{VLIW} + (N_{CNF} \times T_{CNF})$$

Where $E_{VLIW} = \sum (E_{VLIW\text{-}0}, E_{VLIW\text{-}1} \ldots\ldots\ldots\ldots E_{VLIW\text{-}N})$

Table. 1 Mathematical Model Parameters

Parameters Description	Possible Values
Total Instructions, N_{IN}	1 -K per program
Total NOPs Used, N_{NO}	0 - 7 per VLIW
VLIW Fetch Time, T_{VFT}	1 cycle per VLIW
Operand Fetch Time, T_{OFT}	0 - 1 cycle per VLIW
VLIWs Exe Time, E_{VLIW}	1 - L cycles per program
Total Config, N_{CNF}	0 - $(N_{IN} + N_{NO})/8$ per program
Configuration Time, T_{CNF}	1 - M cycles per VLIW

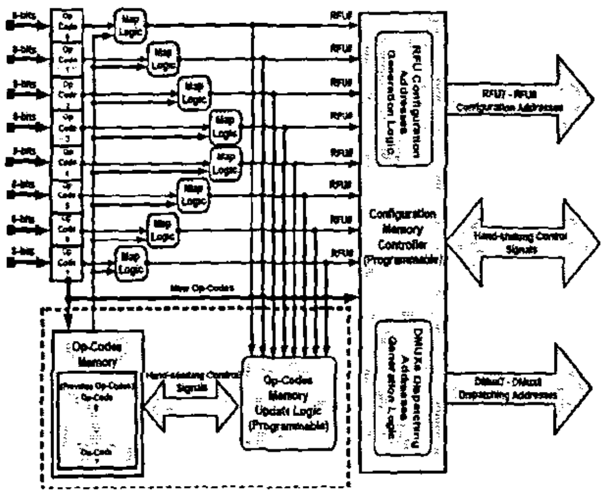
Performance statistics have been measured in terms of the no of clock cycles consumed by a typical DSP [5] and proposed reconfigurable processor for the execution of different application programs. It has been observed that the segments of code of an application containing loops of repeated operations will be drastically boasted up when executed on the proposed reconfigurable processor as shown in Fig. 7.

V. Comparison With Existing Architectures

In this section the proposed reconfigurable processor architecture is compared with some of the well known reconfigurable architectures.

A- Configuration Granularity:
The proposed processor is fine grain architecture. There exist many systems using this approach like CHIMERAE [4]. Using fine grain approach the system can be reconfigured at instruction level and even at operator level. But there exist many other systems which use the coarse grain architecture and can be reconfigured at ALU level. Among them are REMARC [2], PipeRench [10] and RAW [9].

B- RFU Coupling Approach:
The proposed processor is a tightly coupled architecture like CHIMAERA [4]. Others may use a coprocessor approach or attached processor approach. Tightly coupled designs have the small configuration overheads but are suffered by the dependant execution of RFU with standard CPU core.



8-bits
Op Code 0
Op Code 1
Op Code 2
Op Code 3
Op Code 4
Op Code 5
Op Code 6
Op Code 7
Map Logic
RFU#
RFU Configuration Addresses Generation Logic
RFU1 - RFU8 Configuration Addresses
Configuration Memory Controller (Programmable)
Hand-shaking Control Signals
New Op-Codes
Op-Codes Memory
(Previous Op-Codes)
Op-Code 0
Op-Code 7
Op-Codes Memory Update Logic (Programmable)
DMUXs Dispatching Addresses Generation Logic
DMUX1 - DMUX8 Dispatching Addresses

Fig. 6 Configuration Controller

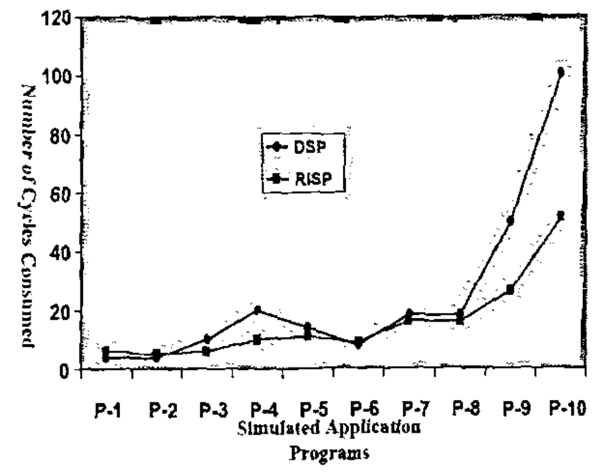


Fig. 7 Proposed Processor vs DSP

C- Operands Address Decoding:
The proposed processor is based on a *fixed operand coding* scheme like PipeRench [3]. But some designs are based on the *hardwired operand coding* scheme like CHIMAERA [4].

D- Instruction Format Encoding:
The instruction of proposed processor is decoded such that the op-code of each instruction is being translated into the address of the concerned configuration block in the configuration memory. Other alternative is to use the op-code as an identifier to a configuration table which contains the address of the concerned configuration block in the configuration memory like in CHIMAERA [4].

E- Application Multi-threading:
The proposed processor can execute more than one instruction (eight) at the same time. Most reconfigurable processors are only able to execute one instruction at the same time. They are based on both CISC and RISC designs. In order to maintain the high performance of system, the multi-threading has been supported by the on chip hardware support.

F- Configuration Memory Design:
The proposed processor has introduced a new concept of configuration memory being implemented as a multi-port RAM memory unlike the existing architecture which are so for being designed using the simple single-port RAM or Cache.

VI. CONCLUSION

Reconfigurable computing is becoming an important part of research in the domain of the high performance computing. Reconfigurable processors are intensively used platforms for achieving such a kind of high performance in computing. Reconfigurable processors provide us a great performance parameter over the traditional micro-processors. In such kind of processors the hardware changes according to the requirements of the active application. Hence the system follows the strategy of the demand-driven operators. The required hardware is swapped in and the unused hardware is swapped out and hence virtually providing more hardware resources than the physically available in the system during the execution of the application. Reconfigurable processors are very suitable processors for those applications where the different kinds of processing units are frequently required to boast up the performance of the application.

REFERENCES

[1] M.B. Taylor, W. Lee, J. Miller, D.Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams". Proceedings of the International Symposium on Computer Architecture 2004, June 2004.

[2] Miyamori, T. and Olukotun, K., REMARC: Reconfigurable Multimedia Array Coprocessor IEICE Transactions on Information and Systems E82-D, vol. pp. 389-397, Feb, 1999.

[3] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor. "PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology". Proceedings of Custom Integrated Circuits Conference (CICC) 2002, pages 63{66, May 2002.

[4] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P., "CHIMAERA: A High-Performance Architecture With a Tightly-Coupled Reconfigurable Functional Unit," Proceedings of the 27th International Symposium on Computer Architecture, pp. 225-235, 2000.

[5] TMS320C62x / C67x CPU and Instruction Set Reference Guide Literature Number: SPRU189C March 1998.

[6] R. W. Hartenstein, R. Kress, and H. Reinig. \A Reconfigurable Data-Driven ALU for Xputers". IEEE Workshop on FPGAs for Custom Computing Machines, April 1994.

[7] NEC Electronics. "Dynamically Reconfigurable Processor (DRP) - Architecture". 2004.

[8] Michael Taylor. "The Raw Processor - A Scalable 32-bit Fabric for Embedded and General Purpose Computing". Proceedings of Hot Chips 13, August 2001.

[9] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs". IEEE Micro, March April 2002.

[10] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. "PipeRench: A Reconfigurable Architecture and Compiler". IEEE Computer, Vol. 33(4):pp. 70{77, April 2000.

Second International Conference on Electrical Engineering
25-26 March 2008
University of Engineering and Technology, Lahore (Pakistan)

RISP Design with Most Optimal Configuration Overhead for VLIW Based Architectures

[1]M. Aqeel Iqbal, [1]Shoab Ahmed Khan, [2]Uzma Saeed Awan
[1]Center for Advanced Studies in Engineering (CASE), Islamabad, Pakistan
[2]International Islamic University (IIU), Islamabad, Pakistan
maiqbal_786pak@hotmail.com, shoab@case.edu.pk
uawan_80@hotmail.com

***Abstract*—In this research paper an alternative design for Reconfigurable Instruction Set Processor (RISP) has been proposed with the capability of the most optimal configuration overhead for Very Long Instruction Word (VLIW) based architectures. This processor supports the demand-driven modification of its instruction set during the program execution. The processor has been integrated with the high speed partially reconfigurable Field Programmable Gate Array (FPGA) cores as its Reconfigurable Functional Units (RFUs) in place of ALUs and it treats instructions as removable modules which can be paged in and paged out through the partial reconfigurations according to the requirements of the application being under execution. Instructions occupy the FPGA resources only when needed and FPGA resources can be released and reused at run-time on a fly for other kind of instructions belonging to the same or the different applications without affecting those who are currently under execution on the FPGA platform. RISPs are the next generation of processors which can adapt their instruction sets through a reconfiguration in their hardware according to the requirements of the applications being under execution on them. In this way the processor adapts its instruction set for the hardware design which is the most suitable for the application being executing on it, during the process of its execution and hence it accelerates the performance. RISPs are the programmable processors which contain the reconfigurable logic in one or more of their functional units. The hardware design of such a kind of processor can be categorized into two main tasks: The first task is to design the reconfigurable logic itself and the second task is to design the communication interface of reconfigurable logic with the remaining modules of the processor.**

Keywords— RISP, Configuration overhead, RFUs, FPGA, VLIW, Multi-port Memory, Configuration Controller.

I. INTRODUCTION

The Reconfigurable Instruction Set Processors (RISPs) combine a microprocessor core with a reconfigurable logic in one or more of their functional units. The reconfigurable logic provides hardware specialization to the application being under execution. The location of the reconfigurable logic in the architecture, relative to the microprocessor core affects the performance. The speed advantages achieved by executing a program in a reconfigurable logic depend on the type of the communication interfaces used between the reconfigurable logic and remaining modules of the processor [1] and the type of the configuration methods used. A reconfigurable functional unit can be placed in three different places, relative to the processor core [5]; first as an *Attached Processor* [2]; second as a *Coprocessor* [6]; and third as a *Functional Unit.* Reconfigurable logic loads its configuration from an external memory i.e. Configuration EPROM or main memory of the system etc. The configuration is loaded in the form of a bit stream either serially or parallelly, just like the bit stream loaded in an FPGA [6]. If we can configure the RFUs after initialization, the instruction set can be bigger than the size allowed by the reconfigurable logic. If we divide the application in functionally different blocks, the RFUs can be reconfigured according to the needs of the each individual block. Reconfiguration times depend on the size of the configuration bit stream, which is mostly quite large. Configuration times are critically dependant on the configuration methods and the configuration interfaces being used [6]. Configuration stream depends on the type of hard ware to be reconfigured and the type of the FPGA core being integrated in the RFUs. If the configurations load operation stops the system working while the loading of configuration stream, there is a great loss of performance. If the RFUs can be used during the loading of the new configurations, it will give a great performance boast up. If we divide the execution unit in different RFUs which can independently be configured, we will not have to reconfigure the all of RFUs at the same time, thus reducing the reconfiguration time. Configuration pre-fetching, configuration cloning and configuration context switching are other alternative techniques used to reduce the reconfiguration over head [6].

II. MOTIVATIONS

In future the interactive multimedia applications will be based on the standards like MPEG-4. Using an object-based approach to describe and composite an audio-visual scene, MPEG-4 combines many different coding tools not only for natural audio and video but also for synthetic objects and graphics. Objects are coded and transmitted separately and composed at the decoder side, letting the receiver interact and influence the way the scene is presented on the receiving display and speakers. Due to this user interaction, the number and the type of decoders that needs to be implemented on the system are not known at the design time, but rather at the run-time [4]. This fact forces the designers of the platforms for these applications to use the new design approaches. Traditionally, multimedia applications have been implemented on custom VLIW processors that provide enough parallelism to accelerate these computationally intensive applications [7], while at the same time retaining low power consumption. In order to increase even further the computational power of these devices, they have been enhanced with custom hard ware for acceleration of the most common multimedia operations. An example of this is the Trimedia Processor [7],

978-1-4244-2293-7/08/$25.00 ©2008 IEEE

vhich contains the specialized units for DCT (Discrete Cosine Transform) and motion estimation.

Jnfortunately, due to the variety of the algorithms that can be ised in new interactive multimedia applications and the fact hat the actual number and the type of the objects is not known ill run time, it is no longer economically viable to make pecialized functional units for each algorithm. The picture is urther complicated if we also take into account that a latform designed for these applications may have to decode in object encoded with an algorithm for which it was not onceived. Hence in order to maintain the power efficiency ind the real time constrains, we need a platform that can be pecialized at run-time to the algorithm at hand. A platform ased on RISP provides this type of run-time specialization.

III. RELATED WORK

Numerous reconfigurable hardware based architectures have ee proposed. Previously proposed reconfigurable processor rchitectures generally fit into one of two categories lepending on the size of the computations they map onto the econfigurable logic.

ine-grained Reconfigurable Processors, such as PRISC [2],)ISC [11], OneChip [1] and CHIMERAE [12] integrate the mall blocks of reconfigurable logic into superscalar processor rchitectures, treating the reconfigurable logic as rogrammable ALUs that can be configured to implement ipplication-specific instructions. CHIMERAE differs from ther systems primarily in that it supports a 9-input / 1-output nstruction model. These systems can achieve better erformance than conventional superscalar processors on a vide range of applications by mapping commonly-executed equences of instructions onto their reconfigurable units, but he maximum speedup they can achieve is limited by the small imount of logic in their reconfigurable units.

oarse-grained Reconfigurable Processors, such as REMARC [9], Garp [6], Napa [10], PipeRench [5], Rapid [3] ind RAW [8] provide larger blocks of reconfigurable logic hat are less tightly-coupled with the programmable portions f the processor. These architectures can achieve extremely ood performance on applications that contain long-running iested loops that can be mapped onto the processor's econfigurable arrays but perform less well on applications hat require frequent communication between programmable ind reconfigurable portions of the processor. Systems such as 'ilchard that integrates FPGAs into conventional workstations ver the processor's memory bus display similar behavior, ilthough the relatively low bandwidth of a processor's nemory bus makes them even more sensitive to the amount of ommunication that an application requires between the rocessor and the FPGA.

IV. PROPOSED RISP DESIGN

In this section the detailed architecture of the proposed RISP ias been discussed. The detailed architecture of the proposed rocessor is shown in the Fig.1 and different modules are liscussed below along with their functionality.

1. Input / Output Interface (IO Interface):

The IO interface of RISP is used to communicate with the external devices being interfaced with it. The first job of the I/O Interface is to load the configuration streams from external Configuration EPROM or main memory of system during the booting processes of the processor and it takes only a few clock cycles. These configuration streams contain the different hardware modules like Adders, Subtractors, Multipliers and Shifters etc. The second job of the IO interface is to load the instructions and their relevant data operands to be executed on the processor. The third job of the IO interface is to store the results of the computations performed on the processor in main memory of the system. The fourth job of the IO interface is to send and receive the control signals generated and acknowledged by the control unit of the RISP to the external devices.

2. Pre-fetch Unit (PFU)

The basic job of the PFU is to fetch or pre-fetch the instruction stream and the data stream of the application program being under execution. Fetched instructions are loaded in the *Instruction Pool* and then transferred into the *Instruction Cache*. Similarly the data stream is loaded into the *Data Pool* and then transferred into the *Data Cache*.

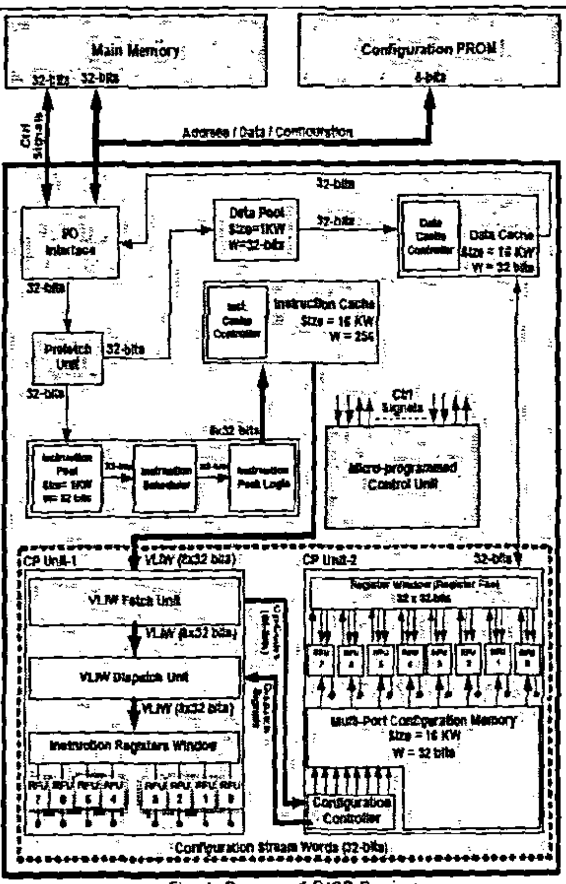


Fig. 1 Proposed RISP Design

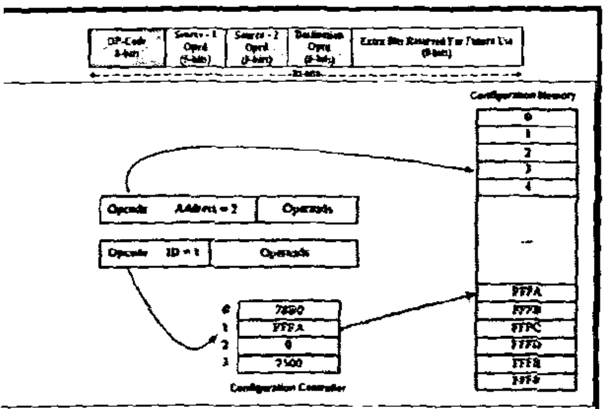


Fig. 2 RISP Instruction Format

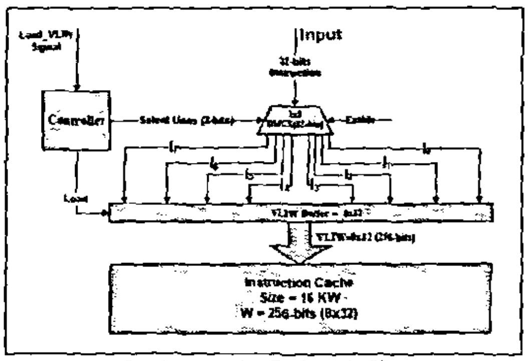


Fig. 3 Instruction Pack Logic (IPL)

ꞌ. Instruction Scheduler Unit (ISU)

The ISU is the micro-programmed implementation of the *'omasoulo's Algorithm* being used in VLIW and Super-scalar ›rocessors for the scheduling of the instructions. The nstruction scheduler reads instructions from the instruction ›ool and then it analyzes them for dependencies (if any) and esolves these dependencies. Dependencies being analyzed nclude Data Dependency, Control Dependency, Resource Conflicts and Data Hazards etc. Then it after analysis ISU ransfers these instructions to IPL.

I. Instruction Pack Logic (IPL)

The main job of the IPL is to pack the eight instructions in the orm of a VLIW. The 32-bits instructions transferred from the SU are given to the IPL. The IPL arranges these instructions n a buffer in a FIFO order on their arrival from the ISU. After he arrival of each instruction, the IPL increments its nstruction counter and checks either there are eight nstruction arrived from the ISU or not. If a total of eight nstructions have been arrived from the ISU then the IPL ransfers them into a VLIW buffer of size 8 x 32-bits. Then it enables this buffer to transfers this VLIW to instruction cache ›f the RISP if signal Load_VLIW =1. The same process is epeated constantly throughout the application execution. Consider the Fig. 3 of IPL.

ꞌ. Computational Pipeline-1 (CP-1)

CP-1 is consisting of a VLIW Fetch Unit (VFU) and a VLIW Dispatch Unit (VDU).

. *VLIW Fetch Unit (VFU):*

/FU is a *State Machine* based unit and works like a *Programmable Counter*. VFU fetches VLIW from the nstruction cache and the *Op-Codes of all instructions of the* /LIW are transferred to the *Configuration Unit* and the VLIW tself is transferred to VDU.

i. *VLIW Dispatch Unit (VDU):*

/DU is consisting of an array of eight De-MUXs whose select ines are controlled by the configuration controller. According o the select lines activated by the configuration controller all ›f the instructions of VLIW are dispatched or issued by *VDU* o their relevant RFUs. Consider the Fig. 4 of VDU.

6. Computational Pipeline-2 (CP-2)

The CP-2 is composed of a *VLIW Execution Unit (VEU)* which contains an array of eight RFUs and a Register Window of 32 registers (32-bits) and a *Configuration Unit* which contains a *Configuration Controller* and a *Multi-port Configuration Memory.*

i. *VLIW Execution Unit (VEU):*

VEU is the core component of the processor because it contains an array of RFUs being used for program execution. Consider the Fig. 5 of VEU. The VEU contains the following major modules.

a) External IO Logic (EIOL)
b) RFUs Data-in/Data-out Logic (RDIOL)
c) General-Purpose and Flag Registers (GFRs)
d) Registers Input/Output Logic (RIOL)
e) Reconfigurable Functional Units (RFUs)
f) Flags Generation Logic (FGL)

a) External IO Logic (EIOL)

The EIOL of the VEU is used to load instructions in the instruction register, source operands in general-purpose registers and the configuration stream in RFUs. The second job of the EIOL is to store the configuration stream being loaded in the RFUs for the analysis purpose and results being generated after the execution of VLIW.

The source operands Sr-1and Sr-2 are loaded into the internal general-purpose registers (GPRs) by the External De-MUX of size 1 x 24. The address given for the Data-in is connected to the select lines of De-MUX as well as to Decoder (5 x 24) input. De-MUX selects one of the general-purpose registers *for data* loading and the decoder enables its output channel connecting to the registers through the MUX of the size 2 x1. This MUX receives 32-bits data operand from External De-MUX at input "1" and receives 32-bits results from RFUs at *the input* "0". *If the* Ext_IO_En=0 then it selects the result coming from the RFUs and loads it in the register. If the Ext_IO_En=1 then it selects the data coming from the External De-MUX and loads it in the registers. Since there are *eight RFUs that can load their results in the same register,* hence in order to solve this problem an 8 x 1 MUX (32-bits) is interfaced with each register input. Each MUX is controlled

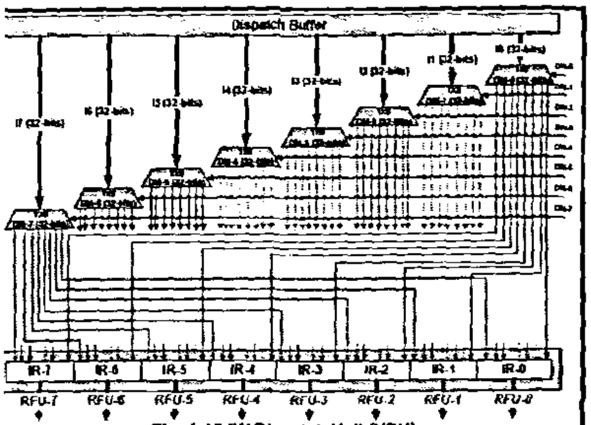
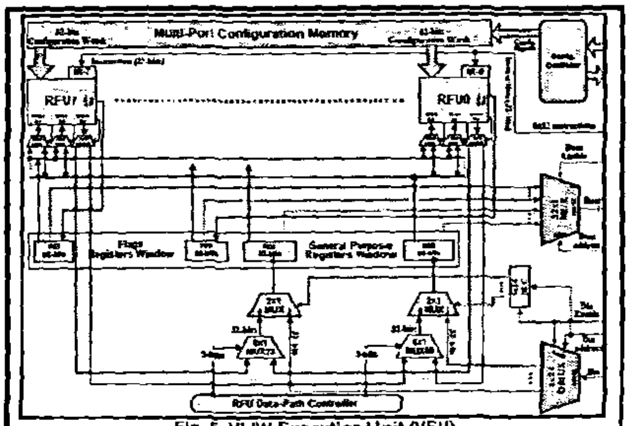


Fig. 4 VLIW Dispatch Unit (VDU)


Multi-Port Configuration Memory
RFU7
RFU0
Flags Registers Window
General Purpose Registers Window
RFU Data-Path Controller
Fig. 5 VLIW Execution Unit (VEU)

the *RFU Data-path Controller* which analyzes the stination Addresses of all the RFUs and selects only that U whose output is valid output. In order to store the results l the flags being available in the GPRs and flag registers Rs) into the data cache of the RISP, the 32 x 1 External JX (32-bits) is used which can read the contents of the ected register and sends it to the data cache of the RISP.

*b) RFUs Data-in / Data-out Logic (*RDIOL*)*
order to load/store the data across the RFUs there are two x 1 MUXs (32-bits) and one 1 x 24 De-MUX (32-bits) for :h RFU. Using the two MUXs the RFU is able to read the ırce data operands (Sr-1 and Sr-2) from any one of the 32 ;isters and using the one De-MUX it stores its results back any one of the GPRs. Flags generated during the execution the VLIW are loaded into the relevant FRs.

c) General-Purpose and Flag Registers (GFRs)
ere is an array of eight FRs (32-bits) and twenty four GPRs !-bits). GPRs can be read and written by the programmer t the FRs can only be read by the programmer and can not written. RFUs can read/write any one of these thirty two ;isters. More than one RFU can read the contents of the ne register at the same time but only one RFU can write in a ;ister at the same time because the read operation is ıreable but the write operation is not shareable.

d) Registers Input/Output Logic (RIOL)
s are loaded with the flags, being generated by the RFUs l can be read by the programmer through the External JX. In case of the GPRs, the programmer can read the isters through the External MUX but in order to write ntents into registers there is a 2 x 1 MUX (32-bits) which ects the data for the register either from some RFU output from data cache. The 8 x 1 MUX interfaced at the input of 2 x 1 MUX selects the valid RFU for the results to be red in the register. In order to select the valid RFU for ults, there is a RFU Data path Controller shown in Fig.6 is ıched with all MUXs. This controller reads the select lines all the De-MUXs of RFUs and after analysis it selects that U whose output is a valid output.

e) Reconfigurable Functional Units (RFUs)
Us are the computational units of RISP and can be reconfigured at any time according to the application demand. They have been tightly coupled in the form of an integrated FPGA core.

f) Flags Generation Logic (FGL)
The outputs generated by the RFUs are also read by the FGL and the flags are calculated for each RFU. Flag register is a 32-bits register but recently only Carry Flag, Sign Flag, Zero Flag, Overflow Flag and Equal Flag have been computed in the system and the remaining twenty-seven bits are available for the future extension.

ii. *VLIW Configuration Unit (VCU):*
VCU is composed of a *Configuration Controller* as shown in Fig.7 and a *Multi-port Configuration Memory* as shown in Fig.5. Configuration controller receives the op-codes of the eight instructions of the VLIW from the VFU and on the basis of these op-codes it decides to load one of the configuration blocks available in the memory for each RFU (if required). Also it checks if the op-code is a No Operation (NOP) or is same as that of any one of the existing op-codes. If so then the configuration controller does not load this new configuration

RFU Data-path Controller
Hardwired Algorithm

The Algorithm Initially Reads the Register Address (Rmn) and Destination Operand Addresses of all RFUs

```
if (RFU0-Dest-Address == Rmn Address)
  Then Sel_out = 0;
else
  if (RFU1-Dest-Address == Rmn Address)
    Then Sel_out = 1;
else
  if (RFU2-Dest-Address == Rmn Address)
    Then Sel_out = 2;
else
  if (RFU3-Dest-Address == Rmn Address)
    Then Sel_out = 3;
else
  if (RFU4-Dest-Address == Rmn Address)
    Then Sel_out = 4;
else
  if (RFU5-Dest-Address == Rmn Address)
    Then Sel_out = 5;
else
  if (RFU6-Dest-Address == Rmn Address)
    Then Sel_out = 6;
else
  if (RFU7-Dest-Address == Rmn Address)
    Then Sel_out = 7;
else Sel_out = Nil;
```

Reconfigurable Instruction Set Processor Design Using Software Based Configuration

M. Aqeel Iqbal and Uzma Saeed Awan
Faculty of Engineering and Information Technology
Foundation University, Institute of Engineering and Management Sciences
maiqbal_786pak@hotmail.com, uawan_80@hotmail.com

Abstract

Due to the potential enhancements in the execution of ftware based applications shown by Reconfigurable struction Set Processors (RISPs), reconfigurable computing is become a subject of great deal of research in the field of mputer sciences. Its key feature is the ability to perform the mputations in hardware to increase the performance on one nd while retaining much of the flexibility of the software on e other hand. The VLSI development is continuously iproving and new ways must be obtained to become able to lly take the advantages of the emerging technology. econfigurable hardware might be the next step which will ve computer performance a big leap forward. The idea is to e the now a day's high performance FPGA technology to lapt the hardware to the problem. This research paper esents an alternative design of a RISP which supports ultiple threads running concurrently, all with instant rdware support. Core of Xilinx FPGAs like Virtex series has en used to adapt the possibilities of loading partial rdware configurations while retaining the execution of the maining active parts of the application.

Index Terms — Fine-grain, Coarse-grain, Configurations, PGA, RFUs, RISP, Multi-port Configuration Memory.

Introduction

Reconfigurable Instruction Set Processors (RISPs) combine standard microprocessor core with a reconfigurable logic in ıe or more of their functional units [1]. The reconfigurable gic provides hardware specialization to the application being ıder execution. The location of the reconfigurable logic in e architecture, relative to the microprocessor core greatly fects the performance of the computational system. The ›eed advantages achieved by executing a program in a configurable logic depend on the type of the communication terfaces used between the reconfigurable core and rest of odules of the processor and the type of the configuration ethods being used. A Reconfigurable Functional Unit (RFU) ın be placed in three different places, relative to the ocessor core [6]; first as an *Attached Processor;* second as a *oprocessor;* and third as a *Functional Unit (FU).* econfigurable logic loads its configurations from an external emory like configuration EPROM or main memory of the stem. The configurations are loaded either serially or ırallelly, just like loaded in an FPGA [2]. If we can configure e RFUs after initialization, virtually the instruction set can be bigger than the actually available. If we divide the application in functionally different blocks, the RFUs can be reconfigured according to the needs of the each individual block. Reconfiguration times depend on the size of the configuration bit streams, which is mostly quite large. Configuration times are critically dependant on the configuration methods and the configuration interfaces being used. Configuration streams depend on the type of hardware to be reconfigured and the type of the FPGA core being integrated in the RFUs [2]. If the configuration load operation stops the working of the platform then during the loading of configuration stream, there is a great loss of performance. If the RFUs can be used during the loading of the new configurations, it will give a great performance boast up. If we divide the execution unit in different RFUs which can independently be configured, we will not have to reconfigure the all of RFUs at the same time, thus reducing the reconfiguration times. The Configuration Pre-fetching and the Configuration Cloning are other alternatives.

2. Related Work In Active Domain

Previously proposed reconfigurable architectures generally fit into one of two major categories depending on grain of computations they map onto the reconfigurable logic.

Fine-grained Reconfigurable Architectures, such as CHIMERAE [5] integrate the small blocks of reconfigurable logic into superscalar processor architectures, treating the reconfigurable logic as programmable ALUs that can be configured to implement application-specific instructions. These systems can achieve the better performance than the conventional superscalar processors on a wide range of applications by mapping commonly-executed sequences of instructions onto their reconfigurable units, but the maximum speedup they can achieve is limited by the small amount of logic in their reconfigurable units.

Coarse-grained Reconfigurable Architectures, such as REMARC [7], Napa [8] and PipeRench [6] provide larger blocks of reconfigurable logic that are less tightly-coupled with the programmable portions of the processor. These architectures can achieve extremely good performance on applications that contain long-running nested loops that can be mapped onto the processor's reconfigurable arrays but perform less well on applications that require frequent ommunication between programmable and the reconfigurable ortions of the processor.

. Software Support for Proposed RISP

The proposed RISP is a reconfigurable RISC architecture sing 32-bits instruction format. In order to manage the onfigurations for the applications, there is no need to have a pecially designed toolset but the existing Xilinx ISE Tools vill be used. There are two types of the instructions used for he processor i.e. Application Instructions and Configuration nstructions. The instruction formats for both types are shown n Fig. 1

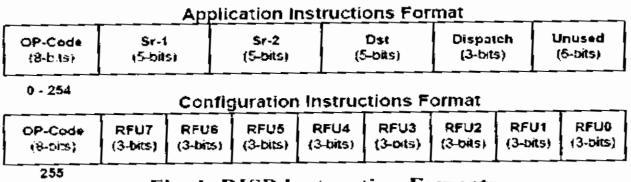


Fig. 1 RISP Instruction Formats

i. Application Instructions

These are the instructions of the application to be executed n the proposed RISP. These are 32-bits RISC instructions. mong these 32-bits, 8-bits on the most significant side of the nstructions represent to the operation codes (Op-Code). The p-codes ranging from 0-to-254 have been dedicated for pplication instructions while the Op-Code 255 has been ledicated for the configuration instructions. In order to access ach register a 5-bits address is required. Also the instruction ormat is a three-address instruction format containing two ddresses for sources and one address for the destination as hown in Fig. 1. On the least significant side of the application nstructions, there are 3-bits attached by the software (either ompiler or application layer) for the Program Dispatch Unit PDU). There are six unused bits which can be used in the uture to further enhance the instruction set.

ii. Configuration Instructions

Configuration instructions are also 32-bits RISC nstructions. These instructions are used by the Configuration Jnit of the RISP to configure the RFUs with a hardware that s the most suitable one for the Execute-Packet (EP) of eight nstructions of the application being under execution. Hence ncluding this configuration instruction in the Fetched Packet FP) it becomes a fetched-packet of nine instructions in it and ence sized as 9 x 32 = 288-bits. Configuration instruction ontains an 8-bits op-code (i.e. 255). Because there are a total f the eight RFUs inside the RISP, each capable of loading its wn configuration from the Multi-port Configuration Memory nd each requires a 3-bits address to mention which onfiguration is to be loaded from it, among a total of eight onfigurations available concurrently for the RFU. Hence here are 8 x 3 = 24-bits available on the least side of the op-ode for this information. Each FU (i.e. RFU) requires a -bits configuration address for its configuration from multi-ort configuration memory. For demonstration purpose the ested configuration streams have been shown in Table.1. These patterns are relocatable pointers, pointing to hardware onfiguration streams and new kind of the configurations can e allocated to these pointers by loading them into specific blocks of multi-port configuration memory. CONF instruction format is shown below

```
CONF (FU7, FU6, FU5, FU4, FU3, FU2, FU1, FU0);

11111111 XXX, XXX, XXX, XXX, XXX, XXX, XXX, XXX;
```

RISP Program Compilation Example:
Let Op-code ADD = 00000000 and SUB = 00000001

Machine Code before Compilation

```
00000000 00000, 00001, 00010, xxxxxxxxx;
00000000 00000, 00001, 00011, xxxxxxxxx;
00000000 00000, 00001, 00100, xxxxxxxxx;
00000000 00000, 00001, 00101, xxxxxxxxx;
00000001 00000, 00001, 00110, xxxxxxxxx;
00000001 00000, 00001, 00111, xxxxxxxxx;
00000001 00000, 00001, 01000, xxxxxxxxx;
00000001 00000, 00001, 01001, xxxxxxxxx;
```

Machine Code after Compilation

```
11111111 010, 010, 010, 010, 001, 001, 001, 001,
00000000 00000, 00001, 00010, 000, xxxxxx;
00000000 00000, 00001, 00011, 001, xxxxxx;
00000000 00000, 00001, 00100, 010, xxxxxx;
00000000 00000, 00001, 00101, 011, xxxxxx;
00000001 00000, 00001, 00110, 100, xxxxxx;
00000001 00000, 00001, 00111, 101, xxxxxx;
00000001 00000, 00001, 01000, 110, xxxxxx;
00000001 00000, 00001, 01001, 111, xxxxxx;
```

4. Hardware Support for Proposed RISP

The detailed architecture of the proposed RISP is shown in Fig.2. The functionality of each module is as given below.

A - Program and Data Memories:

The program memory is loaded with two kinds of the instructions i.e. the application instructions and configuration instructions. The data memory is a multi-purpose memory. Firstly; during the booting of the device it is loaded with the *Configuration Streams* either using the Configuration EPROMs or directly from the master system and then streams are moved into the multi-port configuration memory [1]. Secondly; the data memory is loaded with the data operands of the application program. These data operands are then moved into the *Register File* of the processor. Thirdly; the data memory is used to store the results being generated by the execution of the program. From this data memory the results are moved out into the main memory of the master system [1].

Table.1 Tested Configuration Stream Patterns

Bit Patterns	Referenced Configurations
000	No Change in configuration
001	Load ADDER configuration
010	Load SUBTRACTOR configuration
011	Load MULTIPLIER configuration
100	Load DIVIDER configuration
101	Load AND configuration
110	Load OR configuration
111	Load NOT configuration

B - Input / Output Interface (IO Interface):

The first job of the I/O Interface is to load the configuration ;treams from external Configuration EPROM or main memory)f master system during the booting processes of the processor ınd it takes only a few clock cycles. These configuration ;treams contain the different hardware modules like Adders, ;ubtractors, Multipliers and Shifters etc. The second job of he IO interface is to load the instructions and their relevant lata operands. The third job of the IO interface is to store the esults of the computations in main memory of the system. The fourth job of the IO interface is to send and receive the :ontrol signals generated and acknowledged by the control ınit of the RISP to the external devices.

C - Program Fetch Unit (PFU):

PFU is a *programmable controller which* fetches nstructions one by one from the program memory and loads hem into the VLIW Unit of the RISP. A pre-fetch unit is ntegrated inside the IO interface of the RISP to fetch or pre-'etch the instruction and data streams of the application)rogram. Fetched instructions are loaded in the program *nemory and* data are loaded into the *data memory*.

D - VLIW Unit (VLIWU):

The VLIWU arranges the instructions in a buffer in a FIFO)rder on their arrival from the PFU. After the arrival of each nstruction, the VLIWU increments its instruction counter and :hecks either there are nine instruction arrived from the PFU)r not. If a total of nine instructions have been arrived from he PFU then the VLIWU transfers them into a VLIW buffer)f size 9 x 32-bits. Then it enables this buffer to transfers this VLIW to Program Analyzer Unit (PAU) of the RISP if signal Load_VLIW =1. The same process is constantly repeated hroughout program execution.

E - Program Analyzer Unit (PAU):

Program analyzer unit receives the VLIW containing the ıine instructions in the form of a packet. It then analysis the)p-codes of instructions to check, which one is the :onfiguration instruction and which one is the application nstruction. After analysis it rearranges the VLIW in a pre-lefined order. A pre-defined op-code (i.e. 255) is dedicated 'or configuration instruction, which cannot be assigned to any)ther instruction of the application program. Consider the Fig. 3 for PAU. PAU receives the 9 x 32 = 288-bits long VLIW, from the VLIW Unit, in the *VLIW Receive Buffer*. Then the *Input-Inst MUX Controller* generates the select lines)ne by one for the *Input-Inst MUX* and instructions are loaded nto the *Analyzer Buffer* one by one. For each instruction, the Op-Code of the instruction is used to control either the *1x8 De-MUX* or the *I_0-Buffer*. If op-code is between 0 – 254, he instruction is loaded into the De-MUX otherwise it is oaded into the I_0-Buffer. Through the De-MUX it is loaded nto the proper instruction buffer from I_1 ___ I_8. At the end all nstruction buffers from I_0 ___ I_8 load their instructions into the 288-bits *Re-arranged VLIW Buffer*. Now VLIW is in the equired format, where the instruction on the least significant side i.e. I_0 is the *Configuration Instruction* and the remaining nstructions i.e. I_1 ___ I_8 are the application instructions. Then his VLIW is transferred to Program Schedule Unit (PSU) which sends configuration instruction to the *Configuration*

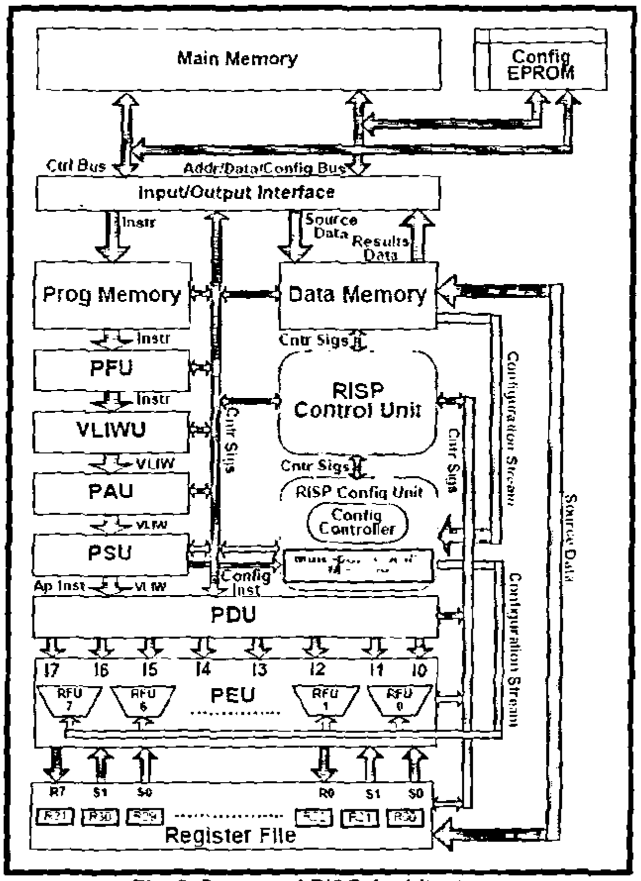


Fig. 2 Proposed RISP Architecture

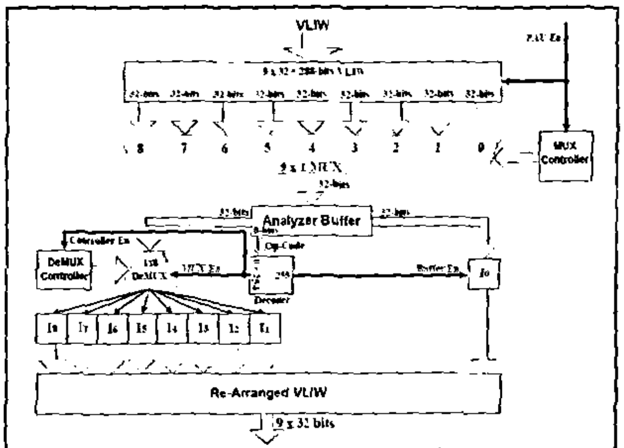


Fig. 3 Program Analyzer Unit (PAU)

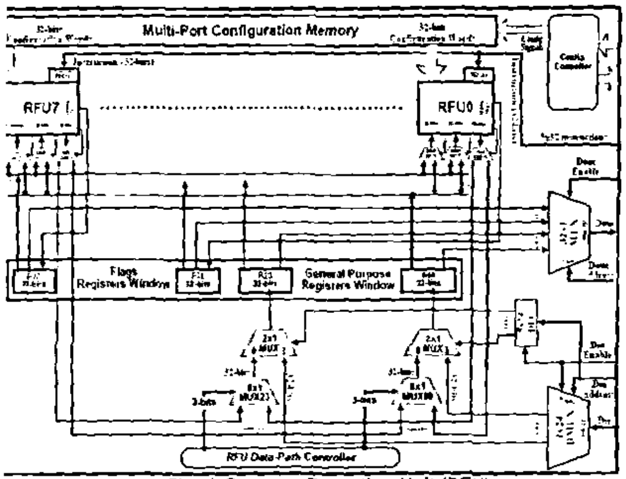
unit and the application instructions to the Program Dispatch Unit (PDU).

F - Program Dispatch Unit (PDU):

PDU is consisting of an array of eight De-MUXs whose select lines are controlled by the three configuration bits being attached with each instruction by the compiler or by the application layer. According to the select lines activated by these attached configuration bits, all of the instructions of VLIW are dispatched or issued by PDU to their relevant RFUs.

G - Program Execution Unit (PEU):

PEU is the main unit of RISP as it contains computational unctional units (RFUs) in it. The functionality of major nodules of PEU is as under. Consider the Fig. 4 of PEU.


Multi-Port Configuration Memory
RFU7
RFU0
Flags Registers Window
General Purpose Registers Window
RFU Data-Path Controller
Fig. 4 Program Execution Unit (PEU)

i. External IO Logic (EIOL)

The EIOL of the PEU is used to load instructions in the nstruction register, source operands in general-purpose egisters and the configuration stream in RFUs. The second ob of the EIOL is to store the results being generated after the xecution of VLIW. The source operands Sr-1and Sr-2 are oaded into the internal general-purpose registers (GPRs) by he External De-MUX of size 1 x 24. The address given for he Data-in is connected to the select lines of De-MUX as well s to Decoder (5 x 24) input. De-MUX selects one of the ;eneral-purpose registers for data loading and the decoder nables its output channel connecting to the registers through he MUX of the size 2 x1. This MUX receives 32-bits data ›perand from External De-MUX at input "1" and receives 32- ›its results from RFUs at the input "0". If the Ext_IO_En=0 hen it selects the result coming from the RFUs and loads it in he register. If the Ext_IO_En=1 then it selects the data oming from the External De-MUX and loads it in the egisters. Since there are eight RFUs that can load their results n the same register, hence in order to solve this problem an ; x 1 MUX (32-bits) is interfaced with each register input. ;ach MUX is controlled by the *RFU Data-path Controller* vhich analyzes the Destination Addresses of all the RFUs and elects only that RFU whose output is valid output. In order to tore the results and the flags being available in the GPRs and lag registers (FRs) into the data memory of the RISP, 32 x 1 ;xternal MUX (32-bits) is used which can read the contents of he selected register and sends it to the data memory of the ؛ISP.

ii. RFUs Data-in / Data-out Logic (RDIOL)

In order to load/store the data across the RFUs there are two ·2 x 1 MUXs (32-bits) and one 1 x 24 De-MUX (32-bits) for ach RFU. Using the two MUXs the RFU is able to read the ource data operands (Sr-1 and Sr-2) from any one of the 32 egisters and using the one De-MUX it stores its results back › any one of the General Purpose Registers (GPRs). Flags ;enerated are loaded into the relevant Flag Registers (FRs). There is an array of eight FRs (32-bits) and twenty four GPRs (32-bits).

iii. Registers Input/Output Logic (RIOL)

FRs are loaded with the flags, being generated by the RFUs and can be read by the programmer through the External MUX. In case of the GPRs, the programmer can read the registers through the External MUX but in order to write contents into registers there is a 2 x 1 MUX (32-bits) which selects the data for the register either from some RFU output or from data memory. The 8 x 1 MUX interfaced at the input of the 2 x 1 MUX selects the valid RFU for the results to be stored in the register. In order to select the valid RFU for results, there is a RFU Data path Controller attached with all MUXs. This controller reads the select lines of all De-MUXs of RFUs and after analysis it selects that RFU whose output is a valid output.

iv. Reconfigurable Functional Units (RFUs)

RFUs are the computational units of RISP and can be reconfigured at any time according to the application demand. They have been tightly coupled in the form of an integrated FPGA core. The outputs generated by the RFUs are also read by the Flag Generation Logic (FGL) and flags are calculated for each RFU.

H - RISP Configuration Unit (Config Unit):

Configuration Unit of the RISP is responsible for the configuration of the RFUs being integrated inside the Program Execution Unit [1]. The configuration instruction inserted by the application software or by the compiler is the control instruction for the configuration unit. Configuration instruction format is shown in Fig. 1. Configuration instruction has an 8-bits op-code (255). When this instruction reaches to the configuration unit of the RISP, it loads it into the *Configuration Buffer Register* (CBR). The op-code of the configuration instruction is decoded for the interpretation. If the op-code is the 255 then configuration controlling 24-bits on the least significant side are loaded into the *Configuration Analyzer Register* (CAR). After loading these 24-bits into the CAR, these are grouped into the 3-bits each and then are sent to the *Configuration Analyzer Unit* (CAU). The main job of the configuration analyzer unit is to analyze and update the configurations being running into the RFUs. Consider the Fig. 5 of the CAU. There are total eight configuration analyzer units inside the configuration unit of the RISP. Each one is responsible for updating the configurations of one RFU. The followings are the tasks performed inside the CAU.

i. It checks the incoming configuration control bits for the No Change Operation (NCO) of the currently running configuration. If the incoming configuration control bits are 000, then it means no change in the currently running configuration.

ii. It checks the incoming configuration control bits for the Same Configuration Operation (SCO). This operation occurs if the incoming configuration control bits are same as that of the currently running configuration. Hence in this case the configuration should also not be loaded into the RFU and hence it saves the configuration overhead of the device.

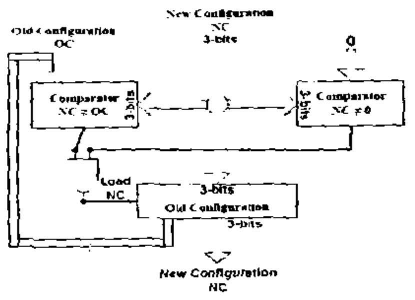


Fig. 5 Configuration Analyzer Unit (CAU)

, Mathematical Model for Analysis

Following is the mathematical formula being formulated for ıe calculations of the total no of cycles (T_{Total}), consumed for ıe execution of an application program. Consider the model arameters in Table.2.

$$_{Total} = \sum(T_C , T_D) + \Pi(N_{FP} , \beta_{FP}) + \sum(E_N (F_N + D_N)) \text{ Cycles}$$

Vhere $\beta_{FP} = \sum (T_{PFT} , T_{OFT})$

Table.2 Mathematical Model Parameters

Parameters	Possible Values
Io of Fetched Packets, N_{FP}	1, 2, 3, ...N / Program
acket Fetch Time, T_{PFT}	1-Cycle / Fetched Packet
)perand Fetch Time, T_{OFT}	1-Cycle / Fetched Packet
.xecute Packets, E_N	1 / Fetched Packet
)elay Slots, D_N	0 - to -1 Cycle
unctional Unit Latency, F_N	1-Cycle / Execute Packet
'onfiguration Time, T_C	0 - to -1 Cycle
)ispatching Time, T_D	1-Cycle / Execute Packet

It has been observed that the segments of codes of pplications containing loops of similar operations, like the ıperation of convolution in digital signal processing, will be lrastically boasted up by the proposed RISP as shown in the ;raph in Fig. 6. These results have been simulated by onfiguring the proposed RISP for the loops containing fixed ›oint arithmetic and logical operations.

i. Benchmark with Existing Architectures

In reference with ***Configuration Granularity***; the proposed ՚ISP is fine-grain architecture like CHIMERAE [5]. Using ine grain approach the system can be reconfigured at nstruction level and even at operator level [4]. But there exist nany other systems which use the coarse-grain architecture ınd can be reconfigured at ALU level like REMARC [7], Iapa [8] and PipeRench [6]. In reference with ***RFU Coupling***; the proposed RISP is a tightly-coupled architecture ike CHIMAERA [5]. Others may use a coprocessor approach ›r attached processor approach. Tightly-coupled designs have he small configuration overheads but are suffered by the lependant execution of RFU with standard CPU core. In eference with ***Operands Coding***; the proposed RISP is based ›n a *fixed-operand coding* scheme. But some of designs are ›ased on the *hardwired operand coding* scheme like CHIMAERA [5]. In reference with ***Configuration Memory***; the proposed RISP is using a multi-port *Configuration* memory unlike the existing architectures which are using the single-port configuration memory [3].

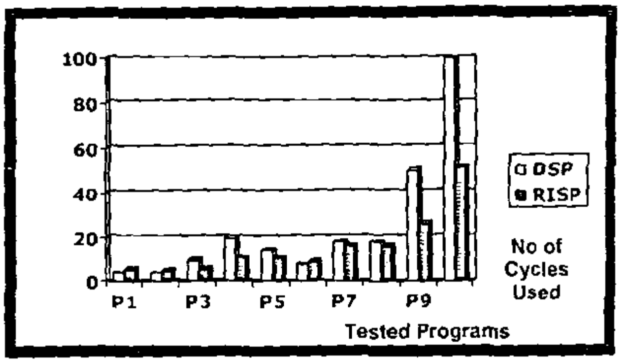


Fig. 6 Performance Analysis Graph

7. Conclusion

Reconfigurable Instruction Set Processors (RISPs) provides us a great performance parameter over the traditional micro-processors. In RISP the hardware changes according to the requirements of the application being under execution. Hence the system follows the strategy of the demand-driven operators. The required hardware is swapped in and the unused hardware is swapped out and hence virtually providing *more hardware than the physically available in the system* during the execution of the application. Reconfigurable Instruction Set Processors are very suitable processors for those applications where different kinds of processing units are frequently required to boast up the performance.

8. References

[1] Aziz-Ur-Rehman, Dr. Aqeel A. Syed and M. Aqeel Iqbal, 'Intelligent Reconfigurable Instruction Set Processor (IRISP) Design', Proceedings of IEEE computer society, 11th IEEE INMIC-2007 Conference, Dec 28-30, 2007 CIIT, Pakistan.
[2] X. Inc., "Virtex-E 1.8 v FPGAs." Xilinx DS022, 2001.
[3] Edson L. Horta and John W. Lockwood. PARBIT: A Tool to *Transform Bitfiles* to Implement Partial Reconfiguration of FPGAs. Washington University, Dep. of Computer Science Technical Report WUCS-01- 13. July 2001.
[4] Xilinx, Inc. Virtex II Configuration Architecture Advanced Users Guide'. March, 2000.
[5] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P., "CHIMAERA: A High-Performance Architecture With a Tightly-Coupled Reconfigurable Functional Unit," Proceedings of the 27th International Symposium on Computer Architecture, pp. 225-235, 2000.
[6] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R.Taylor, and R. Laufer. "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", in Proc. Intl. Symp. on Computer Architecture, May 1999.
[7] Miyamori, T. and Olukotun, K., REMARC: Reconfigurable Multimedia Array Coprocessor IEICE Transactions on Information and Systems E82-D, vol. pp. 389-397, Feb, 1999.
[8] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H.Holt, J.Arnold and M. Gokhale, "The NAPA Adaptive Processing Architecture", IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1998.