Middle-East Journal of Scientific Research 15 (8): 1125-1133, 2013
ISSN 1990-9233
© IDOSI Publications, 2013
DOI: 10.5829/idosi.mejsr.2013.15.8.1830

High Speed Computational Unit Design for Reconfigurable Instruction Set Processors

[1]*M. Aqeel Iqbal and Shoab* [2]*A. Khan*

[1]Department of Computer Engineering College of Electrical and Mechanical Engineering
National University of Sciences and Technology (NUST), Pakistan
[2]Department of CE, College of E and ME, NUST, Pakistan

Abstract: Reconfigurable computing is intended to fill the gape between hardware and software by providing an ideal platform for scientific and commercial computations. The platform tends to be as much flexible as that of general purpose processors (GPP) and also tends to be as much high speed as that of application specific integrated circuits (ASICs). Reconfigurable processors are the one of the most advanced solutions being proposed for reconfigurable computing. The performance of a reconfigurable processor is greatly dependant on the design of its reconfigurable computational unit. A large number of designs for such kind of computational units have already been proposed but majority of them are suffering from the problem of enormous configuration overheads. In this research paper a high speed computational unit design for reconfigurable processors has been proposed. The proposed design is using many emerging techniques like multi-port configuration memory, intelligent configuration updation, concurrent configuration mapping and multi-threaded configuration controller. The proposed design has been simulated for the execution of a variety of different application programs and the performance statistics so obtained have proved that it can be an excellent candidate for the design of high speed reconfigurable processors with the tendency of minimum possible configuration overheads and hence can be used to enhance the performance of high speed scientific and commercial applications.

Key Words: FPGAs · RFUs · Reconfigurable Computing · Reconfigurable Processors · RFU Coupling

INTRODUCTION

Reconfigurable Computing: General purpose processors (GPPs) have always been at the heart of the most of the current high performance computing platforms. They provide a flexible computing platform and are capable of executing a large class of applications. The software for general purpose processors is developed by implementing high level functions in the form of a tightly coupled software core using the instruction set of the under laying architecture. As a result, the same fixed hardware can be used for many general purpose applications. Unfortunately, this generality is achieved at the expense of performance. The software program stored in memory has to be fetched, decoded and executed. In addition, data is fetched from and stored back into memory. These conditions force the explicit sequentialization in the execution of the program. Casting all complex functions into simpler instructions to be executed sequentially on the under laying processor results in the degraded performance of the computation. Application Specific Integrated Circuits (ASICs) provide an alternate solution which addresses the performance issues of the general purpose processors. ASICs have fixed functionality and superior performance for a highly restricted set of the applications. However, ASICs restrict the flexibility of the mapped algorithms on the architecture.

A new computing paradigm using *Reconfigurable Computing (RC)* promises an intermediate trade-off between flexibility and performance [1]. Reconfigurable computing utilizes hardware that can be adapted at run-time to facilitate the greater flexibility without compromising performance gain [2].

Corresponding Author: M. Aqeel Iqbal, Department of Computer Engineering College of Electrical and Mechanical Engineering National University of Sciences and Technology (NUST), Pakistan.

Reconfigurable architectures can exploit fine-grain and coarse-grain parallelism available in the application because of the adaptability. Exploiting this parallelism provides significant performance advantages as compared to the conventional microprocessors. The reconfigurability of hardware permits adaptation of the hardware for specific computations in each application to achieve higher performance compared to software [3]. Complex functions can be mapped onto the architecture achieving higher silicon utilization and reducing instruction fetch, decode and execute bottleneck. Reconfigurable logic can be defined to consist of matrix of programmable computational units with a programmable interconnection network superimposed on the computational matrix. 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 [4].

Reconfigurable computing is based on a reconfigurable core being integrated inside the reconfigurable processor. The reconfigurable core is composed of many computational elements whose functionality is determined through the programmable configurations as shown in Figure 1. These elements some times known as *Configurable Logic Blocks (CLBs)* are connected by programmable routing resources as shown in Figure 2. The idea of configuration is to map logic functions of a design to the processing units within a reconfigurable device and use the programmable interconnects to connect processing units together to form the necessary circuit [5]. Huge flexibility comes from the programmable nature of processing elements and the routing resources. Performance can be much better than software based approaches due to the reduced execution overhead. Under this definition *Field Programmable Gate Array (FPGA)* is a form of reconfigurable computing device and is shown in.

Reconfigurable Processors (RPs): Combine a microprocessor core with a reconfigurable logic like FPGA [6], [7] as shown in Figure 3. The reconfigurable logic provides hardware specialization to application being under execution. Reconfigurable Processors can be adapted after the design in the same way as the programmable processors can adapt to the application changes [6]. The location of the reconfigurable logic in the architecture, relative to the microprocessor affects the performance. The speed advantages achieved by

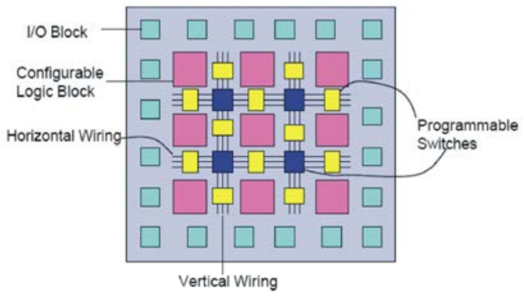


Fig. 1: A Typical FPGA Architecture

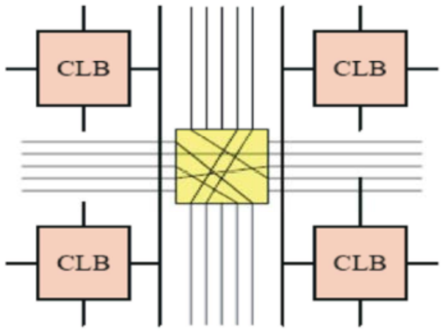
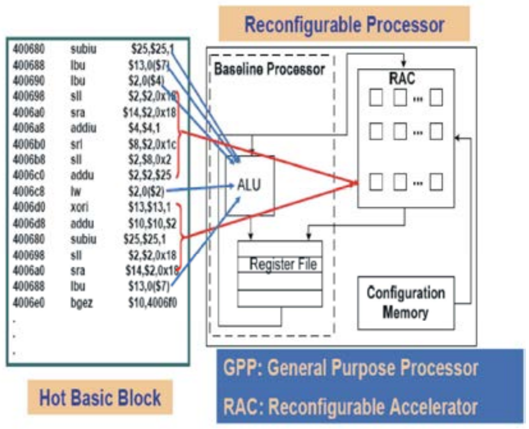


Fig. 2: Programmable Routing Resources for CLBs



```
400680   subiu   $25,$25,1
400688   lbu     $13,0($7)
400690   lbu     $2,0($4)
400698   sll     $2,$2,0x18
4006a0   sra     $14,$2,0x18
4006a8   addiu   $4,$4,1
4006b0   srl     $8,$2,0x1c
4006b8   sll     $2,$8,0x2
4006c0   addu    $2,$2,$25
4006c8   lw      $2,0($2)
4006d0   xori    $13,$13,1
4006d8   addu    $10,$10,$2
400680   subiu   $25,$25,1
400698   sll     $2,$2,0x18
4006a0   sra     $14,$2,0x18
400688   lbu     $13,0($7)
4006e0   bgez    $10,4006f0
.
.
.
```

Fig. 3: Typical Reconfigurable Processor Design

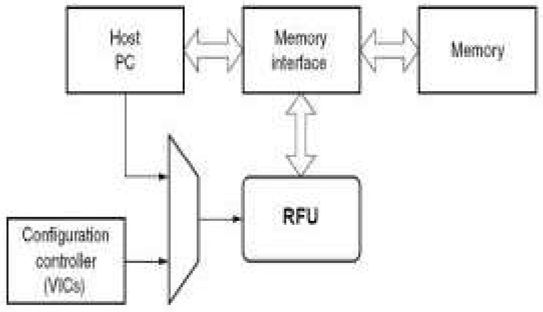


Fig. 4: RFU as Loosely Coupled Independant Unit

executing a program in a reconfigurable logic depend on the type of the communication interfaces used between the reconfigurable logic and the remaining modules of the system architecture. A reconfigurable functional unit (RFU) can be placed in three different places, relative to the processor [6]; As an *Attached Processor*, as a *Coprocessor* and as a *Functional Unit*. The reconfigurable logic loads its configurations from an external memory i.e. Configuration EPROM etc. The configuration is loaded in the form of a bit stream either parallelly or serially, just like the bit stream loaded in any FPGA. If we can configure the RFU 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. In this way the instruction adaptation is done in a per block basis.

RFU as Loosely Coupled Device: In such a kind of coupling of reconfigurable fabric with the host processor, the fabric is attached with main host processor via an external bus like serial port or parallel port etc. In such kind of loosely coupled systems, the reconfigurable functional unit (RFU) has no direct data transfer links to the main host processor as shown in Figure 4. Instead, all data communication takes place through main memory of the computing system. The host processor or a separate configuration controller is responsible to load a configuration stream into the RFU and places operands for the instruction into the main memory. Then RFU can perform the operations specified by the executing instruction and return the results back to main memory. Since such kind of loosely coupled and independent RFUs are separate from the traditional host processor, the integration of the RFU into existing computer systems is simplified. Unfortunately, this kind of loose coupling also limits the bandwidth and increases the latency of data transmissions between the RFU and traditional host processor. For this reason such kind of loosely coupled independent RFUs are well suited only to those applications where the RFU can work independently from the main host processor. Examples include data-streaming applications with significant digital signal processing, such as multimedia applications like image compression, decompression and data encryption.

RFU as Co-processor: As opposed to the loosely coupled independent processing model, other systems more tightly couple the RFUs with the main host processor on the same chip. In some cases, the RFU is loosely coupled with the processor as an independent functional unit. Such architectures typically allow direct access to the RFU from the main host processor as well as independent access to memory. Examples include the Garp and the Chameleon systems. Alternatively, the RFU can be coupled more tightly with the main host processor. For example, in Chimaera system the RFU is incorporated as a reconfigurable processing unit (RPU) within the main host processor itself.

RFU as Attached Processor: The commercial Reconfigurable Processor (RPs) was created by Chameleon Systems Incorporation [8]. It combined an embedded processor subsystem with an RFU via proprietary shared bus architecture. The RFU had direct access to the processor as well as direct memory access (DMA). The reconfigurable fabric also had a programmable I/O interface so that users could process off-chip I/O independent of the rest of the embedded on-chip processing system. This provided more flexibility for the RFU than in typical reconfigurable processing architectures, where the RFUs generally had access only to the processor and memory. The Chameleon architecture was able to provide an improved price to performance relative to highest performing commercially available DSPs, but its reconfigurable processing fabric consumed more power because of the RFU.

Rfu as Tightly Coupled Functional Unit: Such a kind of systems tightly couple the RFU to the central processing unit data-path in a same way as that of traditional CPU functional units such as the ALU, the multiplier and the FPU as shown in Figure 5. In some cases, these architectures only provide RFU access to input data from the register file in the same way as the traditional FUs of CPU. Other architectures allow the RFU to access data stored in the local cache memory directly. For reconfigurable computing architectures in which the RFU is tightly coupled with the processing core, the processor pipeline must be updated so that to dissolve the unit likes a traditional pipeline unit. RFU typically run during the execute stage and possibly the memory stage of the pipeline. Some of these processors are capable of running RFUs in parallel with instructions that use more traditional processor resources, such as the ALU or FPU and even support out-of-order execution like Chimaera [8].

Proposed Computational Unit: The flow chart showing the complete configuration and execution cycle for the proposed unit is shown in Figure 6. The proposed c*omputational unit* as shown in Figure 7 and simulated

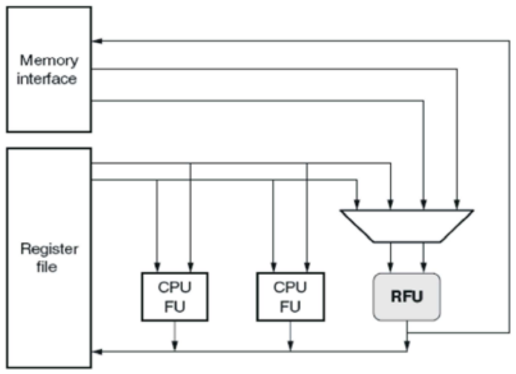


Fig. 5: RFU as Tightly Coupled Dependant Unit

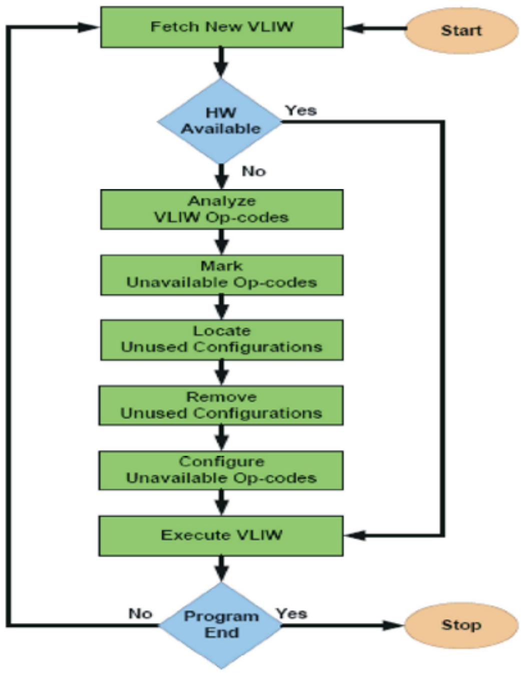

Start
Fetch New VLIW
HW Available
Yes
No
Analyze VLIW Op-codes
Mark Unavailable Op-codes
Locate Unused Configurations
Remove Unused Configurations
Configure Unavailable Op-codes
Execute VLIW
No
Program End
Yes
Stop

Fig. 6: Computational Unit Program Execution Chart

design using Proteus software is shown in Figure 8 is composed of many sub-modules. The *External Input/Output Logic (EIOL)* of the Bus Interface Unit (BIU) is used to load instructions in the instruction register, source operands in general-purpose registers and the configuration stream in Reconfigurable Functional Units (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 as shown in Figure 3. The address given for the Data-in is connected to the select lines of De-MUX as well as to Decoder

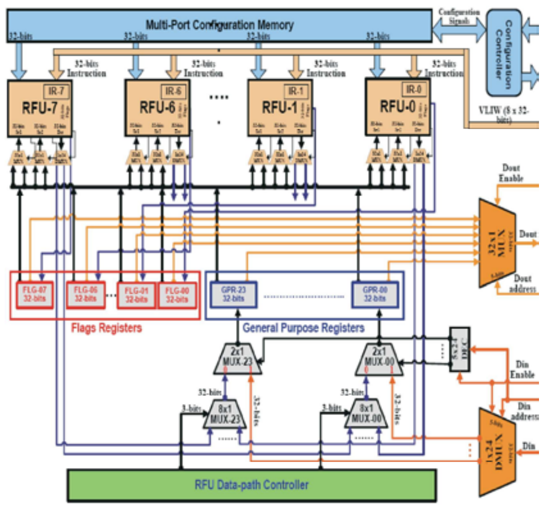


Fig. 7: Proposed Computational Unit Architecture

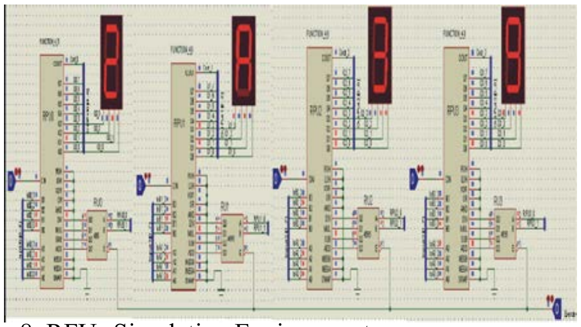


Fig. 8: RFUs Simulation Environment

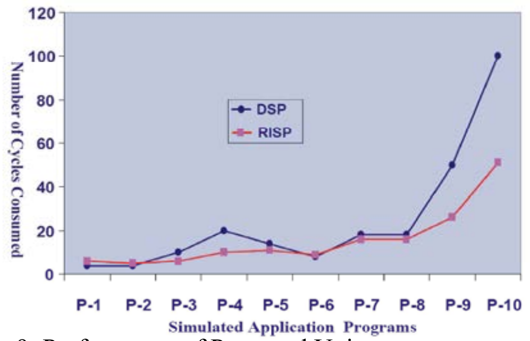


Fig. 9: Performance of Proposed Unit

(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 IO_En signal is OFF then it selects the result coming from the RFUs and loads it in the register. If the IO_En signal is ON 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 general purpose registers (GPRs) and *Flag Registers (FLGs)* into the data cache of processor, 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 reconfigurable processor.

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 as shown in Figure 6. Using 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 execution of the VLIW are loaded into relevant FLGs. There are eight FLGs (32-bits) and twenty four GPRs (32-bits) as shown in Figure 7. GPRs can be read and written by programmer but the FLGs 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. FLGs 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 as shown in Figure 3. In order to select the valid RFU for results, there is a RFU Data path Controller being 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.

Reconfigurable core unit is consisting of a layer of eight RFUs that are the computational units of reconfigurable processor and can be reconfigured at any time according to the demands of the running applications. They have been tightly coupled with integrated field programmable gate array (FPGAs) as shown in Figure 7. 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. *Field Programmable Gate Arrays* (FPGAs) consists of an array of *Configurable Logic Blocks (CLBs)* overlaid with an interconnection network of wires known as *Programmable Interconnect* as shown in Figure 1. Both the logic blocks and the interconnection network are configurable [7].

The configurability is achieved by using either anti-fuse elements or *Static Random Access Memory (SRAM)* memory bits to control the configurations of transistors. The Anti-fuse technology utilizes strong electric currents to create a connection between two terminals and is typically less reprogrammable [9], [10]. The SRAM based configuration can be reprogrammed on fly by downloading different configuration bits into the SRAM memory cells. Current and future generation of reconfigurable devices ameliorate the reconfiguration cost by providing partial and dynamic reconfigurability [11]. In partial reconfiguration, it is possible to modify the configuration of a part of the device while configuration of remaining parts is retained [12], [13]. In the dynamic reconfiguration, the devices permit this partial reconfiguration even while other logic blocks are performing computations. Devices in which multiple contexts of the configuration of logic block can be stored in the logic block and context switched dynamically have been proposed [12]. In order to further increase the performance of such devices by reducing the configuration overheads, the concepts of Configuration Cloning, Configuration Pre-fetching, Configuration Context-Switching, Configuration Compression and Intelligent Configuration techniques have also been proposed.

Performance Analysis Of Design

Typical DSP Performance: For the comparison of the performance of the proposed unit we selected the well-known DSP processor TMS320C6X [14] provided by Texas Instruments. It is a fixed-point VLIW architecture containing a total of eight functional units. The pipeline of the TMS320C6X can fetch a VLIW of eight instructions [14]. 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 to the functional

Table 1: Equation Parameters Description

Parameter	Values
No of fetch packets (FP)	1 - N / Program
Packet Fetch Time (T_{PFT})	1 Cycle/F. Packet
Operands Fetch Time (T_{OFT})	1 Cycle/F. Packet
Exe. packets (E_n= Fn + Dn)	1 - 4 /F. Packet
Delay Slots (Dn)	0 - 1Cycles/E.Packet
Functional Unit Latency (Fn)	1 Cycle/E. Packet
Total Execution Time (T_T)	M Cycles

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 or more execute packets and then executes. The following mathematical equation can be used for the calculation of execution times of programs. Consider Table 1 for equation parameters being used.

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

Consider the application programs P1, P2, P3, P4 and P5 for execution time calculations. The programs have following statistics when run on TMS320C6X [14].

Application Program (P-1):

```
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;
```

VLIW Fetch Time T_F = 1 Cycle
Operand Fetch Time T_O = 1 Cycle
No of execute packets = 1
Execute Time T_E = 1 x 2 Cycles
$T_{Total} = FP\,(T_{PFT} + T_{OFT}) + (F_0 + D_0)$
= 4 Cycles

Application Program (P-2):

```
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;
```

VLIW Fetch Time T_F = 1 Cycle
Operand Fetch Time T_O = 1 Cycle
No of execute packets = 2
Execute Time T_E = 2 x 1 Cycles
$T_{Total} = FP\,(T_{PFT} + T_{OFT}) + (F_0 + D_0) + (F_1 + D_1)$
= 4 Cycles

Application Program (P-3):

```
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;
```

VLIW Fetch Time T_F = 1 Cycle
Operand Fetch Time T_O = 1 Cycle
No of execute packets = 4
Execute Time T_E = 4 x 2 Cycles
$T_{Total} = FP\,(T_{PFT} + T_{OFT}) + (F_0 + D_0) + \ldots\ldots + (F_3 + D_3)$
= 10 Cycles

Application Program (P-4):

```
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;
```

VLIW Fetch Time T_F = 2 Cycle
Operand Fetch Time T_O = 2 Cycle
No of execute packets = 8
Execute Time T_E = 8 x 2 Cycles
$T_{Total} = FP\,(T_{PFT} + T_{OFT}) + (F_0 + D_0) + \ldots\ldots + (F_7 + D_7)$
= 20 Cycles

Application Program (P-5):

```
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;
```

VLIW Fetch Time $T_F = 2$ Cycle
Operand Fetch Time $T_O = 2$ Cycle
No of execute packets = 6
Execute Time $T_E = (2 \times 1) + (4 \times 2)$ Cycles
$T_{Total} = FP\ (T_{PFT} + T_{OFT}) + (F_0 + D_0) + \ldots\ldots + (F_5 + D_5)$
$= 14$ Cycles

B. Proposed Unit Performance: Consider the Figure 7 for proposed computational unit design. In a typical RISP using proposed unit the program fetch unit and VLIW unit take one CPU Cycle to fetch and deliver eight instructions based VLIW. The program analyzer unit, program schedule unit and program dispatch unit take one Cycle to analyze, schedule and dispatch eight instructions based VLIW. The execution time taken by program computational unit depends upon the type of the instructions to be executed. The configuration time required to configure different hardware modules needed by the running application varies from device to device and from hardware to hardware to be reconfigured. Configuration times mentioned here are for Vertex-E device based configuration cloning architectures. The operating speed assumed for the proposed unit has been taken as the fastest possible speed of Virtex-E series of field programmable Gate Array. So for, these devices are capable of operating at a high speed of more than 500MHz. Following is the mathematical model being formulated for the calculations of the total no of cycles (T_{Total}) consumed by proposed unit for the execution of an application program. The formulated mathematical model is based on an equation which calculates the total number of clock cycles being required by the unit to execute the said application program. Consider the Table.2 for the mathematical model parameters.

$$T_{Total} = N_{VLIW}\,(T_{VFT} + T_{OFT}) + T_D + E_{VLIW} + \beta_{CNF}$$

Where $\beta_{CNF} = (N_{CNF} \times T_{CNF})$,
$N_{VLIW} = (N_{INST} + N_{NOP}) / 8$
$E_{VLIW} = \Sigma\,(E_{VLIW\text{-}0}, E_{VLIW\text{-}1}, E_{VLIW\text{-}2} \ldots E_{VLIW\text{-}N})$

Consider same application programs P1, P2, P3, P4 and P5 for execution time calculations when run on a RISP using the proposed computational unit. The programs have following statistics.

Application Program (P-1):
No of Long words $N_{VLIW} = 1$
VLIW Fetch Time $T_{VFT} = 1$ Cycle
Operand Fetch Time $T_{OFT} = 1$ Cycle
Dispatch Time $T_D = 1$ Cycle

Table 2: Equation Parameters Description

Parameters Description	Possible Values
Total Instructions (N_{INST})	1 – J / Program
Total NOPs Used (N_{NOP})	0 – 7 / VLIW
Total VLIWs (N_{VLIW})	1 – K / Program
VLIW Fetch Time (T_{VFT})	1 Cycle / VLIW
Operand Fetch Time (T_{OFT})	0 – 1 Cycle / VLIW
VLIWs Exe. Time (E_{VLIW})	1 – L Cycles/ Program
Total Config. Time (β_{CNF})	0 – M Cycles/ Program
Total Config. (N_{CNF})	0 - N_{VLIW}/ Program
Configuration Time (T_{CNF})	1 – N Cycles / VLIW

Config Time $\beta_{CNF} = 1$ Cycle
No of execute packets =1
Execution time $E_{VLIW} = 1 \times 2 = 2$ Cycle
$T_{Total} = N_{VLIW}\,(T_{VFT} + T_{OFT}) + T_D + E_{VLIW} + \beta_{CNF} = 6$ Cycles

Application Program (P-2):
No of Long words $N_{VLIW} = 1$
VLIW Fetch Time $T_{VFT} = 1$ Cycle
Operand Fetch Time $T_{OFT} = 1$ Cycle
Dispatch Time $T_D = 1$ Cycle
Config Time $\beta_{CNF} = 1$ Cycle
No of execute packets =1
Execution time $E_{VLIW} = 1 \times 1 = 1$ Cycle
$T_{Total} = N_{VLIW}\,(T_{VFT} + T_{OFT}) + T_D + E_{VLIW} + \beta_{CNF} = 5$ Cycles

Application Program (P-3):
No of Long words $N_{VLIW =}$ 1
VLIW Fetch Time $T_{VFT} = 1$ Cycle
Operand Fetch Time $T_{OFT} = 1$ Cycle
Dispatch Time $T_D = 1$ Cycle
Config Time $\beta_{CNF} = 1$ Cycle
No of execute packets =1
Execution time $E_{VLIW} = 1 \times 2 = 2$ Cycle
$T_{Total} = N_{VLIW}\,(T_{VFT} + T_{OFT}) + T_D + E_{VLIW} + \beta_{CNF} = 6$ Cycles

Application Program (P-4):
No of Long words $N_{VLIW} = 2$
VLIW Fetch Time $T_{VFT} = 1$ Cycle
Operand Fetch Time $T_{OFT} = 1$ Cycle
Dispatch Time $T_D = 1$ Cycle
Config Time $\beta_{CNF} = 1$ Cycle
No of execute packets =2
Execution time $E_{VLIW} = 2 \times 2 = 4$ Cycle
$T_{Total} = N_{VLIW}\,(T_{VFT} + T_{OFT}) + T_D + E_{VLIW} + \beta_{CNF} = 11$ Cycles

Application Program (P-5):
No of Long words $N_{VLIW} = 2$
VLIW Fetch Time $T_{VFT} = 1$ Cycle
Operand Fetch Time $T_{OFT} = 1$ Cycle
Dispatch Time $T_D = 2$ Cycle

Config Time β_{CNF} =2 Cycle
No of execute packets =2
Execution time $E_{VLIW} = (1 \text{ x } 1) + (1 \text{ x } 2) = 3$ Cycle
$T_{Total} = N_{VLIW}(T_{VFT} + T_{OFT}) + T_D + E_{VLIW} + \beta_{CNF} = 11$ Cycles

A variety of programs have been executed and benchmarked on the both processors. The theoretically calculated and simulated performance statistics have been shown in the form of a comparison graph as shown in Figure 9. It has been observed that the segments of codes of an application containing loops of similar or repeated operations will be drastically boasted up on a reconfigurable processor using the proposed unit.

Related Research Work: The following topics outline the different aspects of reconfigurable computing that research has been addressing in the past several years and still there is a lot of research work required to explore new ideas that can further enhanced the performance of reconfigurable computing systems.

A. 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 and are candidates for mapping to reconfigurable hardware.

B. Emerging Architectures: As far as the reconfigurable architectures are concerned, a variety of reconfigurable devices and system architectures have been developed which propose the various ways of organizing and interfacing the configurable logic resources. Certain architectures are based on fine-grain functional units and some are based on coarse-grain functional units [15].

Coarse-grain architectures are configured on the fly to execute an operation from a given set of the operations. Also the commercially available reconfigurable architectures are exploring the integration of the reconfigurable logic and the microprocessors on the same chip.

C. 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 the 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. There have been several research projects that focused on developing architectures and the associated software tools for mapping onto their specific architecture. Such projects include Berkeley Garp, National Semiconductor NAPA and CMU PipeRench.

D. Configuration Pipelining: Pipelined designs have been studied by several researchers in the configurable computing domain. The concept of virtual pipelines and their mapping onto physical pipelines has also been analyzed. A group has addressed some of the issues in mapping virtual pipelines onto a physical pipeline by using incremental reconfiguration in the context of PipeRench. Yet another group described the pipeline morphing and virtual pipelines as an idea to reduce the reconfiguration costs. A pipeline configuration is morphed into another configuration by incrementally reconfiguring the stage by stage while computations are being performed in the remaining stages.

CONCLUSION

The proposed computational unit provides us a great performance parameter over the traditional processor computational units. In the proposed computational unit the hardware changes according to requirements of applications being under execution. The required hardware is swapped in and the unused hardware is swapped out and hence providing more hardware than that actually available in the system during the execution of the application. This reconfiguration of the hardware does not stop the application being under execution. Due to the partial reconfiguration of device, the time overheads required to reconfigure device are compensated because the application keeps continue its operation during reconfiguration of the device. Such kind of computational units are very suitable for those applications where different kinds of processing units are frequently required to boast up the performance of executing application.

REFERENCES

1. Benkrid, Khaled, 2008. “High Performance Reconfigurable Computing: From Applications to Hardware” IAENG International Journal of Computer Science, 1: 35.

2. Philip Garcia and Katherine Compton, 2006. Michael Schulte, Emily Blem and Wenyin Fu, "An overview of reconfigurable hardware in embedded systems", EURASIP Journal on Embedded Systems, pp: 1-19.
3. Todman, T.J., G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung, 2005. "Reconfigurable computing: architectures and design methods," IEE Proceedings: Computers and Digital Techniques, 152(2): 193-207.
4. Hartenstein, R., 2002. "Trends in reconfigurable logic and reconfigurable computing," in Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems (ICECS '02), Dubrovnik, Croatia, September, pp: 801-808.
5. Hartenstein, R., 2001. A decade of reconfigurable computing: a visionary retrospective. In DATE '01: Proceedings of the conference on Design, automation and test in Europe, pages Piscataway, NJ, USA, IEEE Press, pp: 642-649.
6. Francisco Barat, R. Lauwereins and G. Deconinck, 2002. "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective"; IEEE Transactions on Software Engineering, 28(9): 847-862.
7. Hauck, S., 1998. "The Roles of FPGAs in Reprogrammable Systems" Proceedings of the IEEE, 86(4): 615-638.
8. Hauck, S., T. Fry, M. Hosler and J. Kao, 2004. "CHIMAERA: Integrating a Reconfigurable Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 12: 2.
9. Kuon, I. and J. Rose, 2006. "Measuring the gap between FPGAs and ASICs," in Proceedings of the ACM/SIGDA 14th International Symposium on Field-Programmable Gate Arrays (FPGA '06), Monterey, Calif, USA, pp: 21-30.
10. Sima, M., S. Vassiliadis, S.D. Cotofana, J.T.J. van Eijndhoven and K.A. Vissers, 2002. Field-programmable custom computing machines-a taxonomy. In Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 2002). Reconfigurable Computing Is Going Mainstream, pp: 79-88.
11. Xilinx, 2001. Virtex Series FPGAs, http://www.xilinx.com,
12. Philip James-Roxby and A. Steven, 2000. Guccione, Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configurations. In Proceedings of IEEE Workshop on Field Programmable Custom Computing Machines, pp: 153-161.
13. Edson L. Horta and John W. Lockwood, 2001. 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 (Available at http:// www.arl.wustl.edu/ arl/ projects/fpx/parbit.
14. TMS320C62x/C67x CPU and Instruction Set Reference Guide Literature Number: SPRU189C March 1998.
15. Peck, W., E. Anderson, J. Agron, J. Stevens, F. Baijot and D. Andrews, 2006. Hthreads: A computational model for reconfigurable devices. In 16th International Conference on Field Programmable Logic and Applications, Madrid, Spain, August.