World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:10, No:11, 2016

Run-Time Customisation of Soft-Core CPUs on Field Programmable Gate Array

Rehab Abdullah Shendi

***Abstract*—**The use of customised soft-core processors in which instructions can be integrated into a system in application hardware is increasing in the Field Programmable Gate Array (FPGA) field. Specifically, the partial run-time reconfiguration of FPGAs in specialised processors for a particular domain can be very beneficial. In this report, the design and implementation for the customisation of a soft-core MIPS processor using an FPGA and partial reconfiguration (PR) of FPGA technology will be addressed to achieve efficient resource use. This can be achieved using a PR design flow that helps the design fit into a smaller device. Moreover, the impact of static power consumption could be reduced due to runtime reconfiguration. This will be done by configurable custom instructions implemented in the hardware as an extension on the MIPS CPU. The aim of this project is to investigate the PR of FPGAs for run-time adaptations of the instruction set of a soft-core CPU, including the integration of custom instructions and the exploration of the potential to use the MultiBoot feature available in Xilinx FPGAs to carry out the PR process. The system will be evaluated and tested on a Nexus 3 development board featuring a Xilinx Spartran-6 FPGA. The system will be able to load reconfigurable custom instructions dynamically into user programs with the help of the trap handler when the custom instruction is called by the MIPS CPU. The results of this experiment demonstrate that custom instructions in hardware can speed up a certain function and many instructions can be saved when compared to a software implementation of the same function. Implementing custom instructions in hardware is perfectly possible and worth exploring.

***Keywords*—**Customisation, FPGA, MIPS, partial reconfiguration.

I. Introduction

FPGAs have become popular over the last decade as they allow designers to create complex digital designs at a low implementation cost. Application Specific Circuits (ASICs), in contrast, introduce a high initial cost and require a large amount of resources to create complex designs. Modern FPGAs now occupy central positions in industry because of their capacity for over 1000 multipliers, megabytes of on-chip memory, hundreds of thousands of logic cells and clock speeds of up to half a gigahertz. Moreover, the cost per function in FPGAs decreases significantly over time [1].

PR is one of the most important features of modern FPGAs provided by the FPGA vendor Xilinx. It allows modules running on an FPGA to dynamically reconfigure and swap during execution while the remaining modules continue operating. PR is an interesting topic for research among students and researchers in the Reconfigurable Computing and Adaptive Hardware field. FPGAs are less efficient in area, power and speed than ASICs; however, it is possible to make them more efficient than a static system when all or parts of the hardware are reconfigured at run-time through the execution operation. The instruction set with user-defined instructions of a soft-core, that is used to speed up the execution of an application in a specific domain, can provide huge PR benefits. Such benefits include integrating different sizes of reconfigurable modules into the system to be placed on an FPGA at run-time, and being able to communicate efficiently with the rest of the system and avoiding additional delay.

In this paper, the extension of a MIPS soft-core, user-defined instruction set will be presented with the help of PR. The aim of this project is to investigate PR of FPGAs for run-time adaptations of the instruction set of a Soft-core CPU, including the integration of custom instructions by presenting a practical introduction to soft-core processor with extension design through the use of step-by-step integration of the system for PR using GoAhead tool flow. The powerful GoAhead tool supports all recent Xilinx FPGAs and includes some features that are not available in the other PR tools provided by the FPGA vendor Xilinx [2] as will be introduced in Section III. The objective of this project is to investigate a custom instruction module library that offers low latency performance; low implementation costs in terms of logic resources, and achieves high CPU clock cycle savings compared to software-only implementations.

Rehab A Shendi was a student at School of Computer Science, University of Manchester, UK. She is now with the Department of Computer Science, Taibah University, Saudi Arabia (e-mail: rshendi@taibahu.edu.sa).

II. Background

A. MIPS Architecture

MIPS overview

In this section, the MIPS architecture will be used as a demonstrator for the custom instruction implementation in hardware. It is used to implement a 32 bit embedded system. Moreover, it is one of the most widely supported RISC processors that have been used in research on efficient processor organisations to deliver the highest performance and high power efficiency.

The original MIPS architecture consists of the following functional blocks:

- **Instruction decoder**: It will decode the simple MIPS instructions since all instructions are the same size with only three different formats.
- **Programme Counter (PC):** It contains the address of the currently executed instruction and then increments the stored value address of the next instruction by 4. In the

case of there being a branch or jump instruction, a delayed branch will occur, which means one more instruction is performed and the value that is provided by the branch or jump instruction will be added to the instruction address.

- **Arithmetic Logic Unit (ALU):** It is a fundamental block of the CPU that performs arithmetic and logical operation on the operands, which are the data inputs to an ALU to be operated on, from register to register, memory to register or vice versa.
- **Registers**: the MIPS processor has 31 general purpose registers including register 0 that holds a constant zero. The other registers will be used by the compiler as outlined in the "MIPS32® Instruction Set Quick Reference"
- **Memory**: It will be only accessed via load and store instructions.

Pipeline registers are often placed between the functional blocks in order to allow the processor to run at high clock speeds and to minimise the delay. Basically, the MIPS processor has been designed to use pipelining to improve throughput and performance. It includes a 5-stage pipeline: Instruction Fetch, Instruction Decode, Execute, Memory access and Register Write Back.

MIPS Instruction Set

The MIPS instruction set is divided into three core groups of instructions. Each one of them has its own encoding, as illustrated in Table I.

TABLE I
TYPE OF MIPS INSTRUCTIONS [3]

Instructions type	BITS					
	31-26	25-21	20-16	15-11	10-6	5-0
R-type	opcode	rs	rt	rd	shamt	funct
I-type	opcode	rs	rt	immediate		
J-type	opcode	address				

Table I shows that each type has a 6-bit main opcode that can be used by the decoder to determine the instruction, while the other fields, rs, rt and rd, will be address vectors in the registers file. Those instructions are used for:

- R-type instructions are Arithmetic Instructions that use two operands from the register file, rs and rt, and the result of the operation will be returned to the register rd. The R-type instruction could share their opcode with other instructions and funct-code will determine the operation.
- I-type instructions are Load/Store Instructions that use a register, rs, with a constant value, coded as the immediate, the result will be returned to the register rt. The I-type instruction could be used for braches, so the immediate will be added to the current PC to perform a branch.
- J-type instructions are Jump instructions that provide a new address for the PC. This means moving the execution to a new code block.

B. Reconfigurable CPU Instruction Set Extensions

Many different applications could be handled by using only GPPs, General purpose processors. However, most of them could use only a small subset of all the available instructions in the GPP. Therefore, some small changes to dedicated hardware in any application could give a huge improvement in execution time. A compression algorithm, for example, would need to count the number of one-bits in a vector. By adding dedicated hardware instruction, the speed up of this algorithm will be increased. Extending the instruction set of a CPU could be one way to do this, allowing for hardware acceleration of small parts of an application. The Microblaze and the Nios soft-core CPUs from Xilinx and Altera are good examples of CPUs that allow custom instructions with the benefits of a fast RISC machine. The next section will highlight the interesting points regrading custom instructions.

1) Custom Instructions in Hardware

Custom instructions enable a designer to implement a complex sequence of standard instructions into a simpler and single instruction built in hardware. The simple description of implementing such a custom instruction in a MIPS CPU, that can access the register file in the same way as an ALU, is shown in Fig. 1.

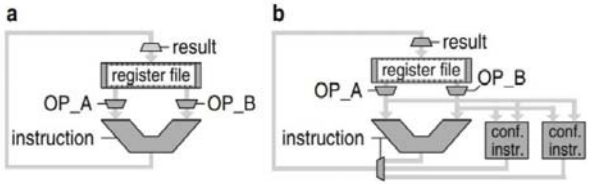

Fig. 1 (a) A typical CPU (b) Extensions CPU with Reconfigurable Instructions [1]

Fig. 1 shows that extending the CPU with exchangeable instructions could be done after decoding unused instruction in the original CPU ISA. Then, a multiplexer is used in order to select between normal ALU option and one or more user defined instructions. Then the configurable instruction can be integrated into the CPU [1].

The custom instruction logic block has two input ports and one output result, as shown in Fig. 4. Often, custom instructions operate in a single clock cycle. However, a multi-cycle operation can be considered for longer combinatory paths. Through the use of custom instructions, it becomes possible to tailor the processor core to a certain application.

One way to emulate the configuration instructions is by adding a large reconfigurable accelerator modules multiplexer that can be placed outside the CPU on the system bus. However, this approach will involve an additional cost. Another way to configure such a custom instruction in hardware is by using runtime partial reconfigurable. The custom instruction could be placed in small slots/islands close to the MIPS CPU, which could cause routing congestion because a high number of signals need to be entered inside the small area. Devices from Altera or Xilinx support design flow tools such as PlanAhead, Open PR and GoAhead flow can

communicate between the static system, which includes MIPS CPU, with custom instructions as they can implement the interface between the static and partial system. By using bus macros, proxy logic or direct mapping wired technique that are provided by PlanAhead, OpenPR and GoAhead flow tools respectively.

Reference [3], who proposed a fast dynamic PR system using GoAhead, argued that with a high number of signals and small islands/slots, design flows using bus macros or proxy logic could not give good results, considering the communication overhead. He shows that by using GoAhead with the direct wire approach, the implementation of the custom instructions can be very efficient in small islands/slots. Consequently, the modules can be relocated. The benefits of allowing the custom instruction to be relocated in more than one slot are the flexibility of slot utilization, the reduction of the external fragmentation and the removal of unnecessary reconfiguration calls, as mentioned by [4].

As a result, the processor will need a look-up table to store a location of a slot that has a custom instruction so that the decoder will know from which custom instruction slot the result should be routed [3].

2) Custom Instructions in Software

Reconfiguration of custom modules could be done either by run-time PR or by a multiplexer that emulates the configuration process, as already mentioned above, and the reconfiguration time could be the biggest overhead. So, in order to trigger the configuration process, there are two fundamental options:

- **Explicit approach:** The configuration instruction will be loaded during the execution time by the user or by the program, before the processor needs it. Reference [5] proposed this method as the configuration pre-fetch instructions before the instruction is called. This method could be fast. However, the speed of the configuration controller and the size of the bitstream will affect the time that the reconfiguration of the custom instruction takes. Consequently, the processor must be stalled, if the configuration of the custom instruction is not finished before the processor calls it.
- **Implicit approach:** An exception trap will be triggered when the processor detects that the custom instruction is not in hardware. The trap handler will handle the configuration process of the custom instruction that the processor needs. The trap handler could run a program [6] that the software function will be executed when the custom hardware is not configured. This approach could remove a lot of overheads by not stalling the CPU while the configuration is in progress. However, it could take time to handle the trap.

C. Design Considerations

The development of a customizable CPU on FPGAs requires the consideration of critical system factors in order to attain the desired performance. Some of the critical objectives that are normally taken into consideration include the speed of the CPU, the memory, the power required and the speed with which the CPU can access other components of the system. There is usually a trade-off between the performance and the power required to attain such performance [7].

The additional design considerations of a customisable configuration include the architecture of the processor and its suitability for the targeted application. This implies that the designer will have to take into consideration the size and type of memory and peripheral bus. In addition, the designer will have to decide on the model and size of the address space that is confined to the CPU, space and type of the caches and instruction and data caches. It is also important to give consideration to the type of controllers that are being used in the architecture. Optional accelerators might be used to speed up the CPU [8].

It should also be mentioned that the operating system and the design and development tools are part of the considerations that will have to be evaluated by the designer. The biggest advantage of implementing the soft-core CPU using FPGA lies in the fact that in the case of any mistake being committed during the development phase, there is the possibility of repeating the process to reconfigure the parameters afresh. There are no limits to the number of times the processor can be reconfigured. This provides designers with a degree of design flexibility [9].

The designer will have to take the development and design tools into consideration that will be used to develop the soft-core. Fig. 2 provides an illustration of the design and development tools. The design and development tools are considered to be responsible for the parameterisation of the soft-core and also the associated implementation of the peripherals [10].

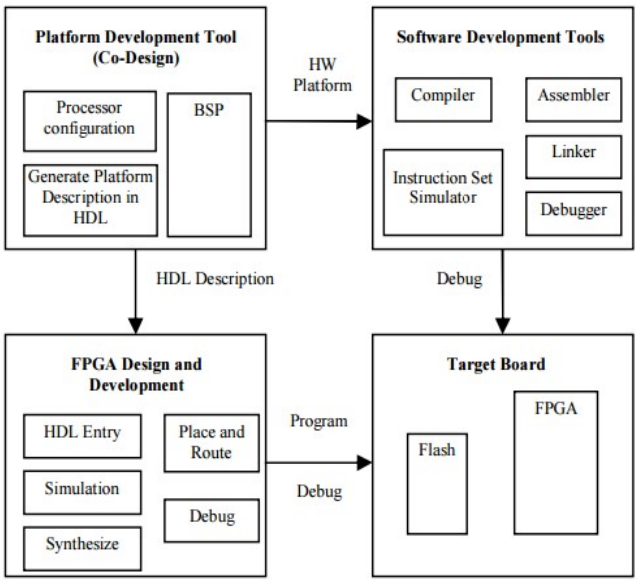

Fig. 2 Design and development tools [11]

FPGAs allow extensive customisation alternatives that are not found in other platforms such as ASIC. Additionally, an FPGA is also considered to have optimisation techniques that

help a designer to work towards achieving performance metrics faster [12]. The benefits of using an FPGA platform in customising soft-core CPUs have also be reviewed. The development of a customisable CPU on FPGA requires critical system factors in order to attain the desired performance [12].

Evaluation of the design and development tools will help the designer to easily and quickly attain the design requirements. It should also be noted that the wrong choice of design and development tools can lead to system inefficiencies. The design and development tools are considered to be responsible for the parameterisation of the soft-core and also the associated implementation of the peripherals [13].

III. Related Work

Related work that is relevant to this project can be categorized into two parts: instruction set extension and PR.

A. Instruction Set Extension

An example study regarding instruction set extension is that of [14]. This study demonstrates the ability to extend the NIOS-II CPU with custom instructions using the SOPC builder wizard of the Quartus design tool. Integrating custom instructions with a soft-core instruction set is a feasible way of speeding up application execution in specific domains such as cryptography [15]. Some of the issues involved in the customisation of an instruction set were analysed in detail by [16] who provided a comprehensive overview of instruction-set extensions.

B. PR

A fair amount of literature has been published on partial run-time reconfiguration in the soft-core CPUs of FPGA. These studies have shown that PR reduces the size, weight, power and cost of an FPGA system. The use of design techniques to increase performance and resource utilisation of reconfigurable soft CPUs was studied by [17]. They have investigated the appropriate instruction implementation technique for a soft CPU which can achieve a performance improvement, while at the same time reduce the resource requirement. It is a different task but fairly closely related to what this project is aiming at. Their goal is to improve soft CPUs for FPGAs using PR. For example, they presented a classification method that determined the parameters for selecting the most suitable instruction based on profiling. Instruction Set Extensions, Software Emulation, Reconfigurable Instructions and ISA Subsetting are the optimisation techniques used in their methodology. Reconfigurable instructions could result in a critical side effect in terms of the configuration time. An example of this could be stalling programme execution while waiting for the reconfiguration process to complete could cause an overhead [17]. Another study by [4] involved an approach to reduce this overhead. They examined the problem which occurs when the 36 communication needs an extra logic or the placement of reconfigurable modules needs to be restricted to the static system which causes an additional logic overhead. They reveal a novel tool called ReCoBus-Builder. In a case study, modules of different sizes and latency were integrated with soft CPUs without causing any logic overhead by using partial run-time reconfiguration. For this project, the newer tool GoAhead, which is a fully re-implemented issue of the tool ReCoBus-Builder, will be used. However, this study will be a library of dynamic instruction set extension.

IV. System Design and Methodology

A. System Development Methodology

System Development Methodology Designing and developing such an effective customization soft-core processor is a challenging task, especially with little experience in processor and system design. Therefore, a system development lifecycle method and a step-by-step design approach are appropriate. This can progressively develop a researcher's learning experience in this important computer engineering field and developing an effective system using PR field.

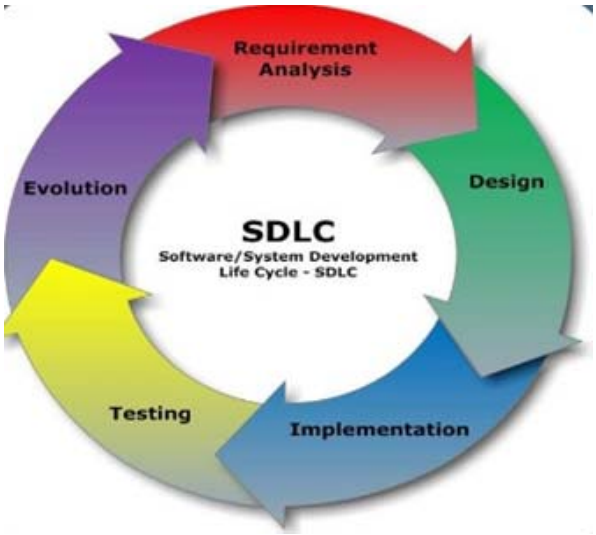

Fig. 3 The general approach of the system development stages [18]

Fig. 3 shows the general lifecycle stages that were used in this project in order to develop a processor. The requirement analysis stage has already been introduced in the objectives section of the Introduction. The design and implementation stages used a step-by-step design and implementation method [19], as shown in Fig. 4, and this will be discussed below in this section. The testing and evolution stages will be introduced in Section IV and will use an appropriate approach for FPGA Embedded Processors design and evaluation [20] such as comparing the system against a software implementation and comparing with the others real-world system. Finally, some techniques for optimizing the performance and cost in an FPGA MIPS processor system will be discussed.

When using such a step-by-step design and implementation method, the customizing soft-core processor has to be done by gradually integrating the processor module with other system

modules and developing other modules to get the final customization soft-core MIPS processor design with the help of the PR. Each of the steps is briefly described below.

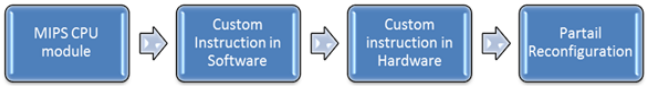

Fig. 4 A step-by-step design and implementation method

First Step: MIPS CPU: First of all, the soft-core is the brain of the system. A MIPS CPU has been implemented in one module, using an XOR gate in the top level in order to synthesize as shown in Fig. 8. The reason for the XOR gate is that the MIPS CPU used more interface wires than there are I/O pins available on the FPGA board. By XORing some of the CPU outputs, the CPU could be synthesized for test purpose (e.g. for data mining clock frequency and resources utilization). Testing MIPS instructions encoding and implementation module was done by using Test Bench in the Xilinx ISE package.

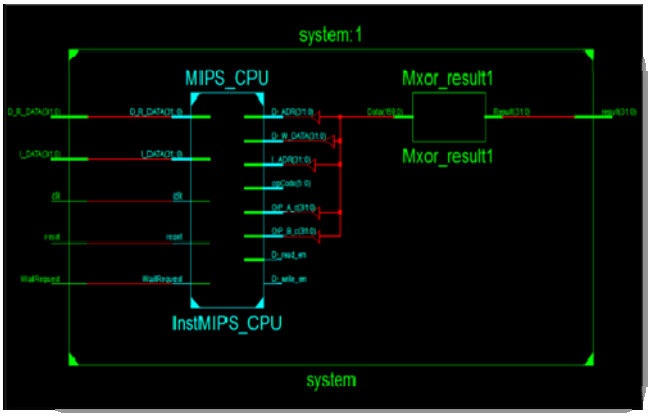

Fig. 5 First step, system overview

Second step: Custom instruction in software: A GCC cross compiler is used in order to compile the MIPS C code. This compiler is modified to include the custom instructions by assigning the custom instructions to unused opcodes. Accordingly, this will be used in the instruction decoder to select the instructions from the binary code. Installing the compiler was done using a virtual machine that was installed on the Windows operating system.

Third step: One custom instruction in hardware: A custom module that will be connected with the MIPS is chosen. Adding a "Counting One" function as a custom module component in the MIPS CPU. The MIPS will detect the custom instruction and return the result from the custom module. Moreover, the MIPS CPU module is connected with other modules such as ROM, RAM and GPIO via system bus.

Fourth step: Custom Instructions library in hardware: Four custom modules are implemented. In addition, a Trap handler that is based on a multiplexer (MUX) is developed in order to choose one custom instruction, the one that is called by the MIPS CPU. This approach has overhead logic costs.

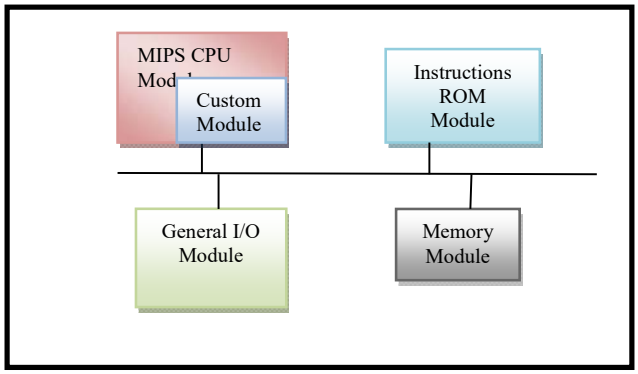

Fig. 6 Third Step, system overview

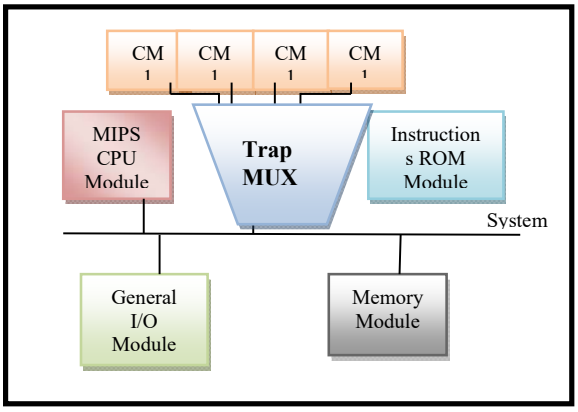

Fig. 7 Fourth step, system overview

Fifth step: Reconfiguration Custom instruction: There are different methods for implementing reconfigurable custom modules in hardware as already mentioned in the background section. In this project the following approaches have been implemented.

First approach step: Improving the Trap handler: The Trap handler based on a MUX is improved to handle the configuration process. In this approach, the trap handler will be based on ICAP. It is done by implementing the trap handler as a state machine which includes a table to save the addresses of the configuration bitstreams for the different custom instructions as will be introduced later and then uses the ICAP primitive in order to load the bitstreams into the device. We will exploit the fact that all academic boards come with serial SPI memory that is often not used. The **MultiBoot feature** is applied in this project; this allows the FPGA to load one of several configuration revisions. Spartan-6 FPGAs support two different configuration modes: BPI and SPI. The functionality of this feature is described in detail in [21]. The iMACT will be used to supply the starting address for each configuration revision in order to generate the MultiBoot SPI file [21]. SPI PROM is specified to store the configuration bitstream for the different custom modules. Consequently, if the custom instruction is needed by the MIPS CPU then the trap handler will check if the custom instruction is already configured otherwise a different bitstream will be loaded from an attached external memory (SPI PROM) into the FPGA. As a result, the FPGA will be reconfigured with a different configuration

bitstream. The whole process works with full reconfiguration, with respect to the MIPS and the extension. The reconfiguration will only make sense with partially reconfigurable custom instructions because rebooting the whole system each time when different custom instruction is called is not a good idea. So a different approach comes from investigating the MultiBoot can be used for PR.

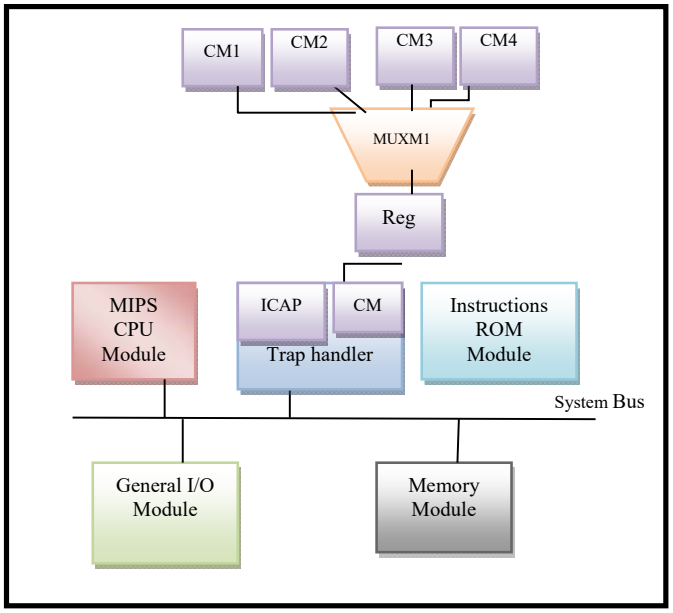

Fig. 8 Fifth step: System overview of the first approach

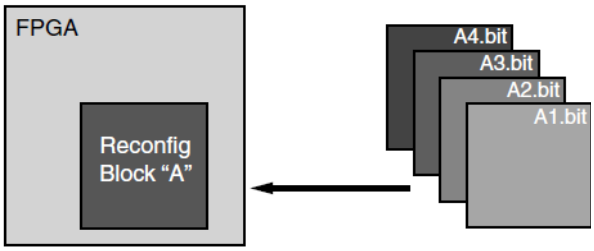

Fig. 9 Final step, system overview [22]

Second approach step: Exploiting the MultiBoot feature for PR. As stated in [22]. PR is a technique for modifying the operation of the FPGA by loading a different bitstream while it is performing its normal operation. The whole design in this technique is translated into different bitstreams or files, where each one defines a separate function and is loaded upon being required. Application Specific Integrated Chips (ASIC) are fabricated and designed to perform a fixed functionality. On the other hand, FPGAs offer the flexibility of being reprogrammed, and most modern FPGAs offer the capability of on-site programming. In PR, the operation of the FPGA is modified by programming a partial bitstream (also called bit files), which defines the operation of a subset of the programmable blocks while in this case the whole FPGA fabric is not reprogrammed. In such a scenario, first of all a full bit file is programmed into the FPGA, which defines the operation for the whole FPGA. Then afterwards, depending on the requirement of the operation, a partial bit file can be downloaded to modify the reconfigurable parts of the FPGA and the other parts continue to perform their operation without being affected. The conceptual diagram of the partial reconfigurable system is shown below.

It can be seen that there is a Reconfigurable Block A in the system, which can be loaded with one of the possible configurations defined by several BIT files, A1.bit, A2.bit, A3.bit, and A4.bit. The logic in the FPGA design is divided into two different regions: reconfigurable region and static region. The dark area of the FPGA block represents reconfigurable regions and the lighter area shows the static region. The functionality of the reconfigurable region is defined by the partial bit files and can be re-programmed by loading one of the partial configurations, while the static region continues to perform its operation and is not affected by the reprogramming of the reconfigurable region. The method of PR offers several advantages:

- This approach helps to reduce the area or size of the FPGA device required to implement a given function, which means fewer logic blocks are consumed; hence, as a result, it also reduces the cost and power consumption of the device.
- This approach helps to implement and test multiple algorithms or methods to perform a specific functionality. In such a case, multiple implementations can be loaded turn by turn and can be compared against each other.
- This technique enhances the design security as specific user dependent keywords or codes can be included into the reconfigurable region and reprogrammed by the end user.
- This approach enhances the fault tolerance in the FPGA design, where any malfunctioning regions or parts can be reprogrammed by the user and can be debugged.
- This approach enables the designer to divide the complete design into multiple regions or blocks, and these blocks can be added to the FPGA design incrementally; hence, it speeds up the FPGA design and verification process.

In our partially reconfigurable system, there is a PR controller implemented in the static region. This PR controller is used to retrieve the partial bitstreams from any memory connected to the FPGA, and then forwards it to a configuration port. There are two possibilities for the PR controller; either it is implemented in an external device such as a separate processor or in the static region of the FPGA design. In the case of the PR controller being located inside the static region of the FPGA, the partial bit files are loaded using ICAP interface. Like the other logic in the static region of the FPGA, the PR controller logic functions without being affected by the programming of partial bit files.

The fundamentals and the concepts of the PR for any system design are discussed above. However, nothing in the documentation provides information on using the ICAP primitive to send the command sequence for loading configuration bitsreams in MultiBoot feature for PR. Hence, PR is applied. The code will be changed to include a black box that presents the custom instruction wrapper later in order to

perform the down to top syntheses, which is the important concept when implementing PR.

B. System Design

The overall project is comprised of two parts. One is the implementation of a custom instruction module library, where we implement custom modules for different operations like CRC, Ones counter, parity etc. The other part is the implementation of the PR region of the FPGA, which is used to reconfigure the reconfigurable region according to the requirements.

C. System Architecture and Components

The overall system is divided into two main regions, the static region and reconfigurable region, as show in Fig. 14. The static part includes all of the 48 major logic and the reconfigurable region only includes the custom module. The MIPS CPU is the main controller processor of the system and it fetches instructions from the instruction ROM. The MIPS CPU decodes the instructions and performs the desired operations. When the MIPS CPU encounters an instruction which is not implemented in its datapath, it will start a hardware trap handler and send the opcode of the desired operation to the trap handler. The trap handler will look at the opcode and check if the desired instruction is already loaded into the custom module and performs the operation. If the desired instruction is not loaded into the custom module, then the configuration manager inside the trap handler will load the partial bitstream using the ICAP primitive and hence a new partial bit file will be loaded into the reconfigurable region and then the operation is performed. The whole process is carried out in hardware to achieve the lowest latency for reconfiguration.

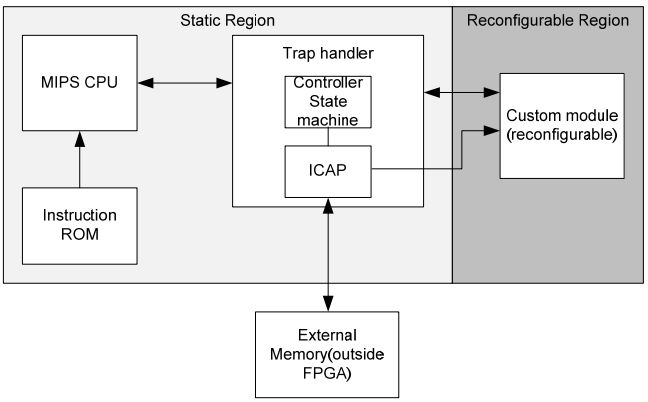

Fig. 10 The final system design

The system operates on a 50 MHz clock, that is derived internally from a top level clock using Global buffers BUFG to allow accessing of the clock in high speed and to provide the least amount of skew possible between the MIPS and the peripherals, which are connected to the bus that physically located in large distances.

D. MIPS Soft-Core Processor

The CPU core is based on the MIPS I instruction set and is built in the system as a soft-core processor. It is used as a platform demonstrator for reconfigurable instruction extensions. Moreover, it is the main module that will control all the different modules and it will run a trap when the custom instruction exception occurs.

E. Peripheral Component Modules

- Memory RAM: A static memory that provides write-before-read behavior. In other words, the data being returned, during a write-cycle, is the same as that being written. The memory module is synthesized into internal block memories in the Sparton-6 FPGA architecture [23].
- GPIO: General-purpose input/output (GPIO) that includes any connection with an input or output pin. The user at run-time can have control of them. GPIO pins such as LEDs and switches go OFF by default [3].
- ROM: This module will contain the machine code of the instructions, using the ROM's address as an index into this memory. The machine code will be generated with the help of a GCC cross compiler that compiles the C code and runs the assembly to produce the binary code that can be used in this array.
- UART: Universal Asynchronous Receiver/Transmitter. A UART module can be added to the system. This unit allows the user to control the operation of the MIPS CPU, the trap handler and other modules and allows them to check the status of the system. Additionally, the UART module can also be used to load the configuration required by the ICAP module.
- System Bus: All modules are connected via a baseline bus protocol, consisting of: Chip select (CS) input signal, Write enable (WR_en) input signal, Address input signal, Writedata input signal and Readdata output signal, with the MIPS as the only master module [3].

F. Configuration Controller Module

- The Trap Handler

The Trap Handler is a core module and is located in the static region of the FPGA design. The trap handler is directly connected with the MIPS CPU with a bus, this module can be easily modified such that multiple CPUs can use it to load the configuration at the desired places and run the operations. Whenever the MIPS CPU encounters an instruction which is not implemented in its datapath, then there are two options: either to have a stall or trigger the trap handler. The trap handler is implemented so as to avoid the malfunction of the CPU due to the non-implemented instruction.

The Trap handler is the module to handle the exception encountered by the MIPS CPU. The MIPS CPU reads the instructions from the instruction ROM and then decodes them. After this, it executes them. In the case that the instruction received is not implemented in the MIPS CPU, an exception is generated. Then the MIPS CPU requests the trap handler to handle the exception. The operation of the trap handler is controlled by a state machine. Fig. 11 shows the state machine diagram for the trap handler.

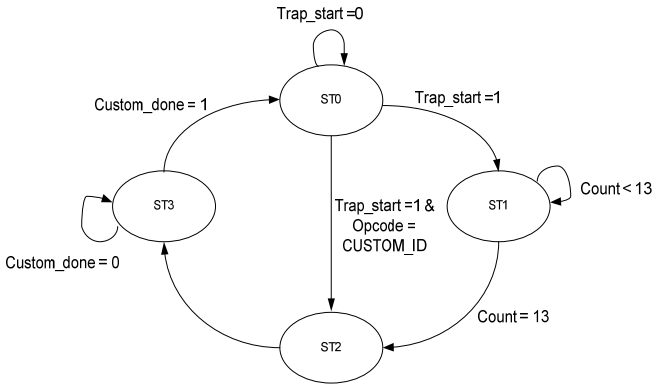

Fig. 11 Trap Handler State Machine

There are four states in the state machine. ST0 is the reset state and the system is normally in this state. Here it waits for the trap start signal, which comes from the MIPS CPU. When an exception occurs inside the MIPS CPU, it will send the trap start signal to the trap handler. On the reception of this signal, the state machine moves to either ST1 or ST2. If the requested Opcode is equal to the currently loaded CUSTOM ID, then there is no need to load the partial bit file so the state machine moves to ST2. In the other case, the state machine moves to ST1, where it sends the command to the ICAP primitive to load the partial bit file inside the custom module. The configuration process is typically thousands of cycles so we use a counter in order to monitor the configuration reading signal from ICAP before going to ST2. At ST2, the trap handler will send a start signal to the custom module and in ST3, it will wait for it to complete the operation.

Each custom module is assigned a unique opcode and the address, which are given in Table II.

TABLE II
CUSTOM INSTRUCTIONS' ADDRESS AND ID

Custom Module Name	Opcode	Address
CRC-32	010000	X"100000"
Ones Counter	100001	X"200000"
Parity	010001	X"300000"
Leading Zero Counter	100000	X"400000"

- The ICAP Primitive

As we are using Spartan-6 FPGA, the ICAP primitive is used to initiate the configuration process (called ICAP_SPARTAN6). It is implemented in the FPGA's fixed logic. This primitive can be used to program the FPGA logic by user control.

- Custom Modules

There are four custom instructions implemented in the design. The instructions are: CRC-32, Ones Counter, Parity flag and Leading zero counter. The concept of each custom instruction is taken from different sources, for example, using the CRC generator to generate the CRC-32 custom instruction module [24].

Each implemented module is assigned a CUSTOM ID, which makes it differentiable from the others. More custom instructions can be implemented and added to the systems by assigning a unique CUSTOM ID to each of the custom instruction as in Fig. 12.

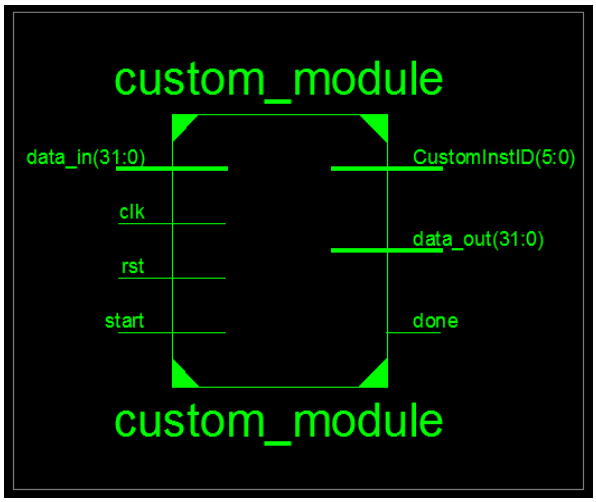

Fig. 12 Custom Module Logic

The CUSTOM ID is evaluated by the instruction decoder of the MIPS CPU in order to run the corresponding module or to trigger the configuration process through the hardware trap handler.

G. Custom Instructions as Partial Reconfigurable Modules

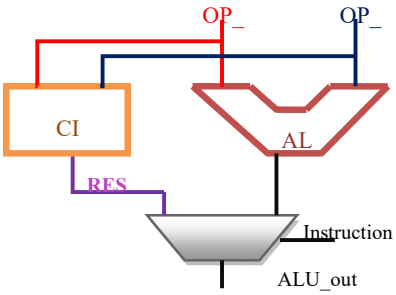

Fig. 13 Custom Instruction (CI) act as extension of the ALU

Fig. 13 shows the MIPS CPU with custom instructions as extensions to the original ALU. It could take one or two 32-bit input operands and one 32-bit output is computed. Adding custom instructions to the system can speed up the execution time of an application as mentioned above. Run-time reconfigurable accelerator modules in a PR region with a proxy logic approach for the communication have been implemented using the GoAhead tools.

Proxy logic will be used as a connection primitive which is nothing else than a look up table in route through mode. It acts as a placeholder for the non-existing part of the system; that is, it replaces the partial module when implementing the static system and it replaces the static system when implementing reconfigurable custom instruction accelerator. The same wires are used for the communication between the static system and the reconfigurable area.

H.Static System Implementation

A screenshot of the static system is shown in Fig. 14. It shows the operand signals (OP_A, OP_B) in the left side and the result signal is collected at the right. The amount of wires that are connected from the static part of the system to the PR region is four for the connection primitive. Consequently, it takes 8 connection primitives for each of the 32-bit interface signals (OP_A,OP_B and RES).

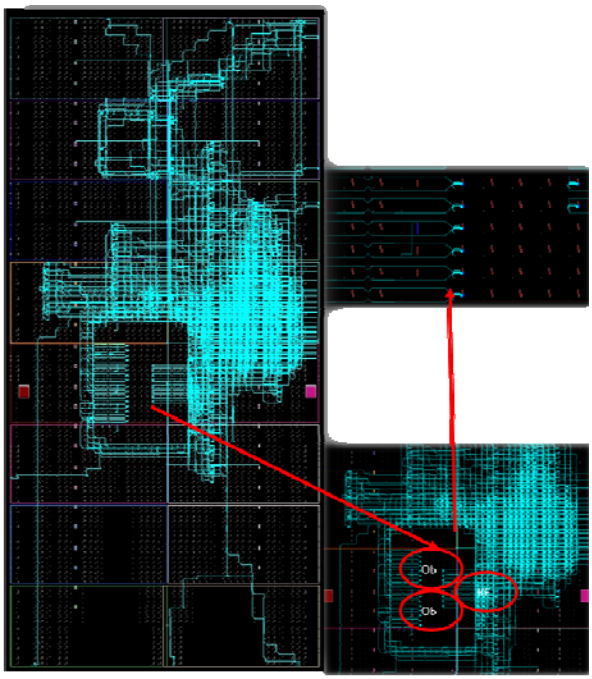

Fig. 14 Static implementation

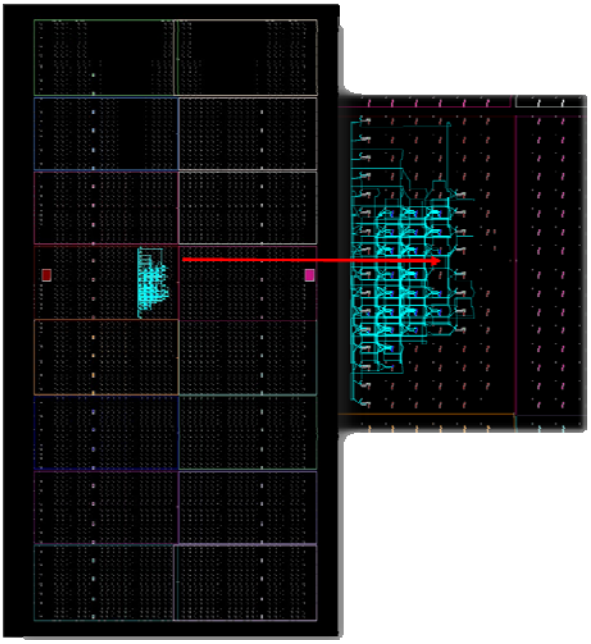

Fig. 15 Partial Part: The example shows the implementation CRC instruction

I. Reconfigurable Instructions

Implementing the reconfigurable modules in the absence of the static system is done as can be seen in the screenshot in Fig. 15. For the partial module implementation, the same primitive will be used with the other side which is not connected yet OP_A to CI and OP_B to Ci and RES_from CI. Fig. 15x shows the CRC module connects where the static design ends by the proxy logic. The custom instruction wrapper has been auto generated by the GoAhead tool.

As the output of the result is not connected to the outside word (i.e. the path ends at the connection primitives), the FPGA tools would typically remove all logic and routing to the output primitive. This will eventually result in an empty design to overcome this; all interface signals were set with a keep attribute (which is specific to the Xilinx vender tools).

V. Results and Evaluation

The cost of the system resources for the first approach and the cost of the system resources for the final system approach are outlined in Table III.

TABLE III
Resource Requirements for Configuration Controller

Approach	Nr of LUT	Nr.of Slices	Latency
MUX based trap handler	246	1798	20.011ns
ICAP based trap handler	438	1370	18.125ns

The results reveal that when using a MUX based trap handler. The system used less look up tables than the ICAP-based trap handler due to the simpler datapath in the ICAP variant. However, the slice resources that are used in the MUX-based trap handler system will be more than those used in the ICAP-based trap handler because the system uses more logic for the custom modules. Finally, the latency is higher in the case of the system that is based on the MUX-trap handler because of the trap overhead. However, in the ICAP system, unless one custom instruction is configured in the system and only in the case of the custom instruction not the desired one then the reconfiguration will be considered. Note that the delay in this table is for the whole implementation without considering the reconfiguration overhead. Only by introducing the ICAP-based trap handler, were we able to run the system at the target 50 MHz clock frequency

For the custom modules, Table IV shows the cost of the resources.

The results in Table IV show the implementation costs for the custom instructions. In the progress report, manual code optimization was performed in order to see if the tools recognize the optimization by itself or not and the result shows the tools do not do that. This point was considered when we implemented the custom module. Therefore, the result shown in Table IV shows the better use of the resources, delay and bitstream size for each custom module after manually optimising each module.

TABLE IV
RESOURCE REQUIREMENTS FOR CUSTOM MODULES

Custom Module	Nr. Of LUT	Nr.of Slices	Latency (Max/av)ns	Bitstream size (KB)
CRC 32	43	18	8.038/3.597	282
Counting One	39	19	15.717/14.35	263
Leading Zero	19	15	9.723/3.597	293
Parity (XOR)	7	6	3.618/3.597	282

A. System Performance

The whole system, including the configuration controller, can run at a system clock of 50 MHz. The first of the two biggest limitation factors is that the MIPS CPU runs a trap when a custom instruction exception occurs and traps have a tiny additional overhead which would not occur in case of a baseline MIPS implementation. The second factor is that the trap handler represents the configuration controller, which uses external flash memory.

For PR, one important benchmark is the response time that has to be considered for the reconfiguration process. Swapping instructions will obviously take a significant amount of time for loading the corresponding partial bitstream from an external SPI memory to the device. Moreover, the bitstream size would affect the speed of the configuration module.

In [3], the configuration controller for module relocation was designed to use two clocks, one clock running at 50 MHz for the part that was connected to the bus and the other one running at 100 MHz for the part that handled the configuration process. In our system, the trap handler will run at 50 MHz, which could slow down the configuration speed. Moreover, in [3] a decompression module is used to decompress the configuration data on the FPGA for faster reconfiguration. So, our predicted result of the reconfiguration time could be lower than what is achieved in that work. However, there are some techniques that could be applied to optimize the performance and cost in the system on the FPGA device. In this project, we used the FPGA MultiBoot feature that is slow, but that uses a serial configuration memory chip that is underutilized in most FPGA prototyping systems. This also separates the configuration bitstream storage from other memory which improves the security of the system.

B. Performance Enhancing Techniques

General speaking, performance techniques could be divided into: techniques that are not FPGA specific from compiler and memory usage to name a few; and techniques that are FPGA specific, such as increasing the operating frequency. As a rule of thumb, since optimizing configuration speed is a typical goal, an entire program should rarely be targeted at external [20] if so, then the use of another clock should be considered in order to handle the process faster than it would be.

C. Comparing the System to a Real-World System

The available embedded processors with the manufacturers quoted maximum frequency and our soft-core, included the extension with its maximum frequency are summarized in Table V. Despite the MIPS processor being the slowest in that table, it might outperform the others due to the use of custom instructions.

TABLE V
COMPARISON BETWEEN XILINX EMBEDDED PROCESSORS WITH OUR SOFT-CORE AND THEIR PERFORMANCE

Processor	Processor Type	Device Family used	Speed (MHz) Achieved
PowerPC$_{TM}$ 405	hard	Vritex-4	450
MicroBlaze	soft	Vritex-II Pro	150
MicroBlaze	soft	Spartan-3	85
MIPS	soft	Spartan-6	50

D. Hardware Acceleration

A soft-core on the FPGA will allow the designer to make a trade-off between hardware and software in order to maximize efficiency and performance. If there is a software function identified as a software bottleneck, then a custom module can be designed for this function in the FPGA. The device will then act as a coprocessor or, as in our case, as a custom instruction extension to the soft-core processor.

One way to evaluate custom instructions in hardware implementation is to compare them against software implementations of the functions running on the standard ISA of the MIPS CPU. The software functions that are used as a reference can be found on [25] Software evaluation for those four functions, which are written in C code, is compiled for the MIPS using a GCC cross-cross compiler. Using disassembly for the code in order to calculate how many instructions each function is consuming. Table VI shows how many CPU instructions are saved by using a custom instruction.

TABLE VI
SOFTWARE REQUIREMENTS

Software function	Instructions
CRC	262
Hamming weight	262
Leading Zero	294
Parity (XOR)	263

VI. FUTURE WORK

There are some improvements that can be done to the final implemented system and together these could be considered as the requirement analysis stage for the next lifecycle.

- In this project, Nexys3 has been used as a platform. However, the lack of external interfaces caused limitations in the usability of this device. Using another academic board which includes audio and video then could show the input and the output of the system and could design a complete digital system built around soft-core processor.
- The MIPS CPU that is used as soft-core is a very simple processor, is nonpipelined and uses BRAM as both program memory and data memory. These could be improved by implementing a pipelined processor also by implementing a simple cache controller that could be connected to DDRmemory. As a result of this, executing larger programs and storing large data structures such as frame buffers could be possible.
- The system uses the MultiBoot feature and the command sequence that is sent through the ICAP primitive to

support the read-back of configuration data from ICAP. However, there are two different ways for reading and writing the configuration data from ICAP. As illustrated in [3]. "Either clock is left toggling and clock enable is used to control throughput, or clock enable is kept high and the clock signal is controlled to achieve wanted throughput" with implementing ICAP interface.

- Adding more advanced modules for communication over COM-port.
- Measuring the clock cycle of the reconfiguration by using Log with a counter in the trap handler in order to reflect the number of clock cycles from the time the counter starts until it is stopped.
- The Nexus3 board has a seven segment electrical screen; it could be exploited for testing.
- Different benchmarks could be used to evaluate the soft-core on the FPGA. The most standard benchmark is Dhrystone MIPs (DMIPs) and the result from this could then be compared with the results we achieved with our system.

VII. Conclusion

The system is improved through the lifecycle that is presented in the methodology. The final system after all improvements had been done meets the objectives outlined in the introduction chapter. Moreover, learning the concepts and the fundamental features of FPGAs step by step is the biggest achievement. The previous chapters described those concepts in detail, the necessary components and tools and the implementation of a fully functional PR system. The dynamically run-time reconfigurable custom instruction set extension of a MIPS CPU can be replaced in the system. The most important part of the implemented system are:

1. MIPS CPU
2. Trap handler, included ICAP primitive.
3. The exploitation of the MultiBoot feature for the full and PR.

Acknowledgment

The author thanks supervisor, Dirk Koch, for giving the opportunity to work in Computer Sciences: Computer System Engineering.

References

[1] Koch, D., 2013. Partial Reconfiguration on FPGAs: Architectures, Tools and Applications. New York: Springer.

[2] Beckhoff, C., Koch, D. & Torresen, J., 2012. Go ahead: A partial reconfiguration framework. Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium, pp. 37-44.

[3] Fritzell, A., 2013. A System for Fast Dynamic Partial Reconfiguration using GoAhead Design and Implementation. Master's Thesis: University of Oslo.

[4] Koch, D., Beckhoff, C. & Torreson, J., 2010. Zero logic overhead integration of partially reconfigurable modules. Proceedings of the 23rd symposium on Integrated circuits and system design, pp. 103-108.

[5] Hauck, S., 1998. Configuration prefetch for single context reconfigurable coprocessors. In: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays. New York: ACM, pp. 65-74.

[6] Pittman, R. N., Lynch, N. L. & Forin, A., 2006. eMIPS, A Dynamically Extensible Processor, Redmond: Microsoft Research.

[7] Kulkarni, R., 2006. Disruptive Technology. Computing & Control Engineering Journal. vol. 17, no. 1, Feb.-Mar., pp. 32-35.

[8] Deschamps, Jean-Pierre, Sutter, Gustavo D., Cantó, Enrique "Guide to FPGA Implementation of Arithmetic Functions" Lecture Notes in Electrical Engineering, Volumen 149. Springer Netherlands 2012

[9] Kozyrakis, C. E. & Patterson, D. A., 2004. Scalable, vector processors for embedded systems. Micro, IEEE, 23(6), pp. 36-45.

[10] Kilts, S. (2007). Advanced FPGA design: architecture, implementation, and optimization. John Wiley & Sons.

[11] Minev, P. B. & Kukenska, V. S., 2007. Implemenation of Soft-core Processors in FPGAs. Gabrovo, International Scientific Conference.

[12] Gebotys, C. H., 2012. A network flow approach to memory bandwidth utilization in embedded DSP core processors. IEEE Transactions on Very Large Scale Integration (Vlsi) Systems, 10(4), pp. 390-398.

[13] Synopsys, 2010. SiliconBlue Selects Synopsys as FPGA Synthesis Partner for Its iCE65 mobileFPGA Family. (Online) Available at: http://news.synopsys.com/index.php?s=20295&item=123144 (Accessed 30 March 2015).

[14] Altera. (2011, Jan.) Nios II Custom Instruction User Guide. (Online). http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

[15] S. Majzoub, H. Diab, "Mapping and Performance Analysis of Lookup Table Implementations on Reconfigurable Platform," IEEE/ACS International Conference on Computer Systems and Applications (AICCSA'07), Le Méridien Amman Hotel, Amman, Jordan, May 13-16, 2007 pp.513-520.

[16] Galuzzi, C. & Bertels, K., 2011. The Instruction-Set Extension Problem: A Survey. ACM Transactions on Reconfigurable Technology and Systems. article 18, 4(2).

[17] Wold, A., Koch, D. & Torresen, J., 2012. Design techniques for increasing performance and resource utilization of reconfigurable soft CPUs. s.l., IEEE, pp. 50-55.

[18] Jo, J., 2013. 6 Basic Phases of Software Development Life Cycle (SDLC). (Online) Available at: http://www.techknol.net/2013/04/software-development-life-cycle.html (Accessed 15 August 2015).

[19] Elkateeb, A., 2011. A Processor Design Course Project: Creating Soft-Core MIPS Processor Using Step-by-Step Components' Integration Approach. International Journal of Information and Education Technology, 1(5), pp. 432-440.

[20] Fletcher, B., 2005. FPGA Embedded Processors Revealing True System Performance. In: Embedded Training Program Embedded Systems Conference.. (Online) Available at: http://www.xilinx.com/products/design_resources/proc_central/resource/ETP-367paper.pdf (Accessed 14 August 2015).

[21] Xilinx Inc, 2015. Spartan-6 FPGA Configuration User Guide. (Online) Available at: http://www.xilinx.com/support/documentation/user_guides/ug380.pdf (Accessed 11 August 2015).

[22] Xilinx, 2012. Partial Configuration User Guide. (Online) Available at: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf (Accessed 1 August 2015).

[23] Doulos.com, 2015. Simple Ram Model. (Online) Available at: https://www.doulos.com/knowhow/vhdl_designers_guide/models/simple_ram_model/ (Accessed 7 August 2015).

[24] OutputLogic.com, 2013. OutputLogic.com. (Online) Available at: http://outputlogic.com/ (Accessed 30 August 2015).

[25] Andersen, S. E., 2005. Bit Twiddling Hacks. (Online) Available at: http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive (Accessed 31 August 2015).

Rehab A Shendi was a student at School of Computer Science, University of Manchester, UK. She got a Master Degree in Computer science specialized in Computer system engineering from The University of Manchester. She is now with the Department of Computer Science, Taibah University, (e-mail: rshendi@taibahu.edu.sa).