Automatic Generation of Hardware for Custom Instructions

by

Philip Ioan Necsulescu

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Master of Science in Systems Science

Interdisciplinary studies

Faculty of Graduate and Postdoctoral Studies

Table of Contents

List of Figures

Acknowledgements

I acknowledge the opportunity given to me by my supervisor, Dr. Voicu Groza, to do Master's research work on a topic of great interest. I also acknowledge the cooperative spirit in the research group regarding the enthusiasm for this project, and in particular of the PhD student, D. Shapiro, who does a great job organizing the activities of the research group. The resources allocated by the Computer Architecture Research Group (CARG) to do this research, in particular the equipment, were essential for the success of my research work.

Abstract

The Software/Hardware Implementation and Research Architecture (SHIRA) is a C to hardware toolchain developed by the Computer Architecture Research Group (CARG) of the University of Ottawa. The framework and algorithms to generate the hardware from an Intermediate Representation (IR) of the C code is needed. This dissertation presents the conceiving, design, and development of a module that generates the hardware for custom instructions identified by specialized SHIRA components without the need for any user interaction. The module is programmed in Java and takes a Data Flow Graph (DFG) as an IR for input. It then generates VHDL code that targets the Altera FPGAs. It is possible to use separate components for each operation or to set a maximum number for each component which leads to component reuse and reduces chip area use. The performance improvement of the generated code is compared to using only the processor's standard instruction set.

Glossary

ASIP - Application Specific Instruction-set Processor

BSP – Board Support Package

CFG – Control Flow Graph

CI - Custom Instruction

COINS – COmpiler INfraStructure. A compiler infrastructure programmed in Java

Cygwin – A Linux emulator for Windows

DFG – Data Flow Graph

Eclipse IDE – A free Integrated Development Environment written in Java

FPGA – Field Programmable Gate Array

GCC – GNU Compiler Collection

GUI – Graphical User Interface

HDL - Hardware Description Language

IR – Intermediate Representation

ISA – Instruction Set Architecture

ISE – Instruction Set Extension

JAVA – An object oriented programming language

LE – Logic Element

LINDO – A commercial software for mathematical modeling which includes a model solver

LPM - Llibrary of Parameterized Modules

LPSOLVE – Open source software for mathematical modeling for optimization

LP – Linear Programming

MILP - Mixed Integer Linear Program

MinGW - Minimalistic GNU for Windows. A free and open source project that facilitates the use of gcc in windows

SHIRA - Software/Hardware Implementation and Research Architecture.

SING – Simulator and Interconnection Network Generator

Verilog – A Hardware Description Language

VHDL - VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuits

1 - Introduction

Technological advances in Field Program Gate Arrays (FPGAs) have opened the possibility for research in creating optimized custom hardware to run many algorithms at much greater speeds than possible on a standard computer processor. Significant research has been done in automating the creation of this hardware such that users do not need a high proficiency in hardware development to be able to benefit from it. Prior research has generally targeted specific applications such as image processing and digital signal processing. Many groups also use specialized languages such as TransmogrifierC and SystemC or require the original source code to be annotated. This requires knowledge of hardware design. This limits the applications of those tools.

The Computer Architecture Research Group (CARG) at the University of Ottawa is currently working on a tool chain called “The Software/Hardware Implementation and Research Architecture” (SHIRA) that automatically generates a hardware system (in VHDL) and the software that executes on that hardware without the need of specialized hardware development knowledge. This is accomplished by using sequential source code written in standard C language as input while also ensuring the tool is applicable to any field. The work to extend the functionality of the SHIRA toolchain to develop a module for the automatic generation of the hardware, incorporating identified performance improvements subject to various constraints is covered in this thesis.

The contribution of this thesis to the SHIRA toolchain is the development of the module for the automatic generation and implementation of the custom instructions that are identified by the prior work of Daniel Shapiro in [1] . The prior work done on the SHIRA toolchain provides an intermediate representation of the instruction to be implemented in the format of a Data Flow Graph (DFG). From the DFG, this new module analyses the data, separates the data into appropriate data types, and generates

the hardware code in VHDL. This module can be customized by the user, allowing the user to limit of the number of different hardware components such as adders and dividers to optimize space and speed requirements. The VHDL hardware code generated by this module is geared towards Altera FPGAs and the NIOS II generic processor. It was developed and extensively tested on an Altera Cyclone® II 2C35 FPGA mounted on the DE2 Development Board.

Custom instructions (CI) are used to add functionality in extension to the basic instruction set of a generic processor. In the SHIRA case, a NIOS II processor developed by Altera is used as the generic processor. The NIOS II processor allows for up to 255 CIs to be added, each with two 32bit inputs and a single 32 bit output. A single CI can have up to 2 * 255 inputs (32 bits), but this further limits the number of CIs that can be added to the NIOS II processor. The output in this implementation is always limited to the single 32 bit port.

CIs are implemented in the scope of increasing the performance of a specific task by using custom hardware instead of software for the identified parts of the program where it is possible and an increase in performance is expected. The identified custom instructions are provided in an intermediate representation (IR) before the hardware code can be generated. In the SHIRA case, the IR is a Data Flow Graph (DFG) such as the one shown in figure 1. To implement these instructions, a synthesizable Hardware Development Language (HDL) is used to generate the hardware, such as the synthesizable subset of VHDL. Generic algorithms to generate the VHDL code for the custom instructions are developed for the SHIRA tool chain such that the hardware can be implemented on Field Programmable Gate Arrays (FPGAs) provided by Altera.

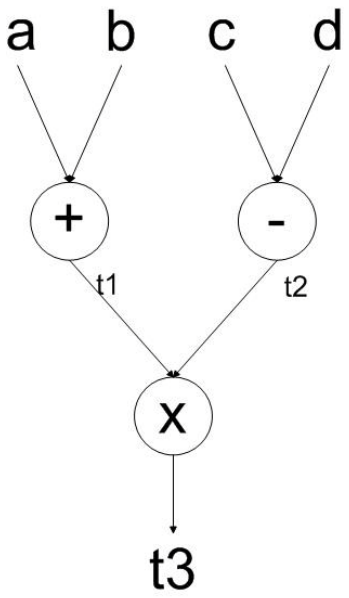

Figure 1 - Simple Data Flow Graph (DFG)

To provide easy adaptability of the module such that it can be modified to write valid VHDL code for FPGAs created by other manufactures, the standardized Library of Parameterized Modules (LPM) is used. LPMs are available in the development tools provided by most FPGA manufacturers and are partly standardized. Altera, through the use of their Quartus IDE software and compiler, also provides the ability to use them. These LPMs allow the HDL code for common hardware components, such as adders and dividers, to use vendor generated automatically according to the provided parameters. These modules are already vendor optimized for their FPGAs and saves the developers a lot of time. They also allow for the easy conversion of the VHDL code to be compatible with the different FGPA solutions from different vendors, such as Xilinx and Lattice

In this thesis, the module was developed and tested successfully that generates synthesizable VHDL code for the identified instructions without any interaction required by the end user. The generated VHDL code makes use of LPM functions and targets the NIOS II processor on Altera FPGAs by extending

its instruction set. These instruction set extensions of the NIOS II processor provide significant speed ups versus using none customized instruction sets of the processor without the requirement of hardware development knowledge.

2 - Background (Literature Review)

The automatic generation of hardware from a higher level computer programming language such as C has been studied by many different groups for the last 20 years. The advent of FPGAs and their development in recent years has made the use of custom hardware for specific tasks cheaper, quicker, and much easier to do than in the past. Many publications are already available with regard to the development and implementation of algorithms that generate custom hardware. The automatic hardware development does not generally create a whole new system, but expends on a system and processor easily generated by using the vendors' tools. The additional hardware is usually implemented as a custom instruction (CI) that is optimized for speed and/or chip area and is added to the instruction set of a general purpose CPU.

2.1 – Applications of Custom Instructions

There are many areas where custom instructions have been successfully implemented to provide very large speedups. Public-key cryptography is an area which stands to benefit from custom instructions (CI). The performance of having an Instruction Set Extension (ISE) for long integer modular arithmetic is studied in [2]. Grosschadl's group uses the LEON2 processor which is based on the SPARC v8 instruction set architecture. This is very similar to the CARG approach of using a generic processor and expending its instruction set with customized code.

With 1024 and 2048 bit encryption now being used, public-key cryptography is computationally intensive. This poses a problem in particular to small embedded devices such as smart cards or sensor nodes. To improve performance, the Cryptography Instruction Set was added to a generic processor, a LEON2. Two instructions were used from what was provided. The first one is the multiply-accumulate instruction (C = A * B + C) which multiples two 32 bit integers and then adds the result to C and stores

the result in C. This instruction is one of the most common custom instructions encountered. The other instruction used is shift right by 32 bit. This instruction is used since the multiply-accumulate instruction stores the result in three 32bit registers. If the result becomes too big, a shift to the right reducing the 32 LSB is used. The paper then presents the implementation of an efficient algorithm in assembly for modular multiplication that avoids the trial division in the reduction operation. Then they optimize the Montgomery reduction algorithm that is used for modular multiplication (that is C = A * B (mod N)) and also optimize exponentiation operations by using the two additional instructions and loop unrolling.

The conclusion of the experiment resulted in a 25% speedup over conventional methods with rolled loops and unoptimized exponentiation demonstrating the large and positive effect custom instructs can have on speed.

Their approach differs from that of CARG in the level of automation. The VHDL code used is not generated automatically from an intermediate representation (IR) but was part of a package. In addition, the code was hand written to use the custom instructions, whereas the SHIRA approach will be to use standard C source with the implementation of the use of the custom instructions done by the automatically by the tool. This work is interesting however as they use a similar method to increase the performance of mathematical operations. The inclusion and use of the multiply-accumulate and 32 bit right shift instructions were shown to be useful and could be some of the first custom instructions to be implemented and tested as part of SHIRA.

Similarly dealing with custom instruction use in encryption, [3] looks at the use of custom instructions to decipher messages using Data Encryption Standard (DES) encryption. It differs in scope from [2] since it deals with key extraction/recovery that requires computational power orders of higher magnitude using a network of processors versus the very limited smart cards targets [2].

DES uses 56 bit encryption leading to a large, but vulnerable number of possible keys (2^{56}). For key recovery, every key is checked for validity until one is found, a process that is very time consuming. Using an approach for linear cryptanalysis, custom instructions are added to a NIOS II processor provided by Altera to try and use only a single clock cycle per key trial. The instructions are added using VHDL and Altera's SOPC tool. Multiple processors can fit on a single FPGA such that the key ranges can be split across them. Also, custom instructions are also added to provide interaction across a TCP/IP network set up between multiple FPGA boards allowing them to communicate between each other and providing additional chip area for more processors.

A key benefit pointed out is the two levels of scalability provided to increase performance. Larger FPGA chips can be used to allow more processors to fit on the chip. Also, additional FPGA boards to the TCP/IP network. Currently they have implemented the custom instructions such that 2 processors can be embedded on the same FPGA. Their goal is to demonstrate a system running 4 processors amongst 8 boards networked together and communicated via the TCP/IP protocol.

This paper demonstrated the addition of custom instructions for a specific application and the use of custom instructions to allow scalability via a TCP/IP network. Although the VHDL code is not automatically generated, it demonstrates another beneficial application for the implementation of custom instructions. It also demonstrates the use of custom instructions to provide for network interconnectivity and easy scalability.

2.2 – Comparison of Automated HDL Generation Porjects

A comparison of automated HDL generation projects over the past 15 years is done in [4]. The current limitations described are the limited subset of supported high level language constructs, the lack of automation in the process leading to too much designer input and knowledge, and poor quality in

generated designs leading to poor performance gains. Three tools in particular are compared, SPARK, ROCCC, and DWARV according to these limitations.

Originally, the goal was to demonstrate the feasibility of generating gates and wires from high level language constructs. TransmogrifierC was one of the first attempts at automated hardware generation. It took simple integer arithmetic operations from C code and translated them to low level gate logic. Afterwards, the direction went to using extended C code which specified hardware before generating code in HDL. Current efforts are trying to use code written in a high level language without extensions for hardware.

In terms of automation, two of the tools require no additional information to the C source. Optimization is only done by two of the tools and both require user guidance.

Many High Level Constructs (HLC), such as floating point types and unions, are not supported by any of these tools. The amount of HLCs from C supported by each tool varied from 31-37 and the number of support applications between 6-50%. The main reasons for the limited support are given by the requirements for structures, perfectly nested loops, and constant sizes required by some of the tools. Also, even though a construct is supported, it might have severe restrictions on its use.

In terms of quality, only SPARK and DWARV were capable of successfully generating and synthesizing hardware but the majority of models generated by SPARK failed to simulate. Problems arose while simulating mainly due to latency estimations.

This paper compared the current state of projects that are similar in nature to SHIRA. The tools do not provide as much optimization as SHIRA attempts to and are a lot more limited in application scope. The paper points out many of the current limitations of the tool chains and identifies areas where VHDL code generation is limited, such as the limitations on array sizes, loops, and the limited high level constructs implemented.

In the SHIRA approach, the identified custom instructions are provided as a Data Flow Graph (DFG), an Intermediate Representation (IR). In [5] a similar approach to SHIRA is used but instead of using a DFG or other graph format, it uses the Language for Instruction Set Architectures (LISA) machine description language. From LISA, HDL code is generated for Application Specific Integrated Processors (ASIP) using a combination of automated code generation as well as hand written code.

LISA describes the instruction-set, the behavioral model, and the timing model of the hardware. LISA, like TMD, provides the link between software and hardware design, giving the developer all the required information to synthesis the architecture. The machine languages are the intermediary between the structural oriented HDL languages and the architecture exploration languages used for optimization.

The HDL code generated from LISA or another machine description language needs to minimize one or more of power consumption, chip area size, and execution speed. The hardware operations described can be grouped to functional units and are generated as wrappers.

The information given in LISA is the description of resources and operations. Resources are the storage elements whereas resources describe the timing and instruction set for the target. For the generation of custom instructions, the instruction set descriptions is of most interest. For parts that are unable to be synthesized to a HDL automatically, such as the data path, hand coding is required. One of the scopes of SHIRA is to eliminate any hand coding apart from the original C source.

Although an older work, this paper deals with a very similar task as the one at hand. It uses a machine transcription language to describe the instruction set extensions. Instead of targeting and FPGA however, the implementation targets ASIPs but the differences due to this are negligible.

Automatic generation of VHDL code has also been studied in [6]. The research group developed a method to generate VHDL code from an elementary operation graph with the help of a tool called PIPE and tested it a Xilinx SPARTAN II FPGA board.

The tool PIPE takes as input the elementary operation graph, which is a data-flow representation in a simple hardware development language. It produces as output the allocation, which is the collection of functional units that are implemented in common processors, the scheduling, and assignment of operations to the physical processors. PIPE also provides control flow information. Using the allocation and control-flow information from PIPE, VHDL code can then be generated.

The code generation is split into 3 steps:

- Modify the current data-flow graph.
- Construct the data-path.
- Realize the control path to activate the processors.

Four different types of hardware elements are used when the VHDL is generated. Code is also provided for the different components generated. Apart from the functional units that perform operations which are referred to as processors (adder, multiplier, etc…), this method requires buffers, multiplexers, and demultiplexers for the data-path. The buffers are used to allow for delays such that pipelining can be used. Multiplexers are used such that the input can be selected for the processor and demultiplexers are used to route the output of the functional unit. Two clock cycles are also used in the generated VHDL code. The select signal in the multiplexers is controlled by the falling edge of clock 1, clock 2 then triggers the multiplexers on its falling edge, finally, the buffers and processors are enabled on the rising edge of clock 1.

The VHDL code generation system developed in [6] is then benchmarked using a Taylor-sequence approximation to compute the cosine of a number. The benchmark shows that their system increases slightly the cost of the system over a human optimized implementation but on the other hand greatly reduces development time. Future research involves studying the implementation of a distributed control path to reduce the amount of control lines of different lengths and the number of crossings to avoid problems in FPGA and VLSI implementations. The method proposed also uses a very large number of multiplexers and demultiplexers. Using buses when possible to reduce the number of multiplexer and demultiplexers will help reduce chip size, allow for simpler control, less wiring, and a lower cost.

A lot of work has been done in identifying custom instructions that will increase performance of applications written in a high level language, such as C. There has also been a lot of work in generating instruction set extensions by hand for specific applications that demonstrate that vast improvements in performance are possible. There has been however limited work done on creating algorithms to generate HDL code from an intermediate representation (IR) without human intervention. There are many projects attempting to do a similar tool chain to SHIRA such as those compared in [4], but their generation of VHDL code is very limited and generally requires a lot of user intervention. The tool chains compared also implement only about 60-70% of the constructs in a C program while also imposing limitations on many of the constructs they do implement. The goal of this part of the SHIRA tool chain is to be able to implement more of the constructs, reduce the limitations, create generic algorithms to generate HDL code for custom instructions, and to remove user interventions for these stages. The processor used is the LEON3 based on the SPARC v8 ISA similar to [2] and the HDL code generated is planned to be synthesizable onto a variety of FPGAs, such as those provided by Altera and Xilinx.

2.3 – Improving Embedded System Performance

In [7], three techniques are explored to improve embedded system performance. The three techniques are: low level software-hardware tradeoffs between basic instructions, utility of Instruction Set Architecture ISA-specific features, and application specific register management. A MIPS processor and the gcc compiler were used for their implementations. For synthesizing, mapping, placement and routing, Quartus 5.0 was used.

In terms of low level software-hardware tradeoffs, parts of the base ISA that are not used in a certain application can be removed from the processor. This resulted in a 25-60% decrease in processor size for some applications. Two further opportunities to reduce to ISA are shown, removing the shift unit and the hazard detection logic. The removal of these instructions from the ISA requires software to be written so that these operations can be done using the remaining instructions available.

Unique ISA features are specific to a certain ISA. The MIPS ISA used in [7] has specific features for load and branch delay slots and hi/lo registers, which are absent from the NIOS II for example. The removal of some of these features might require extra instructions to be added however. The hi/lo register removal is an example of that. When two 32 bit integers are multiplied, their result is a 64 bit integer stored in two registers, requiring two load operations. The removal of the hi/lo registers requires two custom instructions to be added, one to calculate the 32 MSB, and one to calculate the 32 LSB.

The number of registers required and their management varies between tasks. Removing registers or modifying their management can increase performance. To customize the compiler's use of registers, two techniques were looked at, operand scheduling and limiting the use of architected registers.

Each of the three techniques provided speedups in general. The speedups varied by the amount of stages in the processors pipeline. The speedup was on average 13%.

Although this paper mostly deals with instruction set reductions, custom hardware sometimes needs to be added to deal with that reduction such that the hardware can still accomplish the task at hand. SHIRA

has currently not explored ISA reduction but could benefit from it. If ISA reduction is decided to be used, the VHDL code generation will have to be adapted.

2.4 –Automating Hardware Design Using Many Different Languages

The hArtes toolchain is a project with the aim of facilitating and automating the design of embedded systems for digital signal processing, general purpose, and reconfigurable systems. In [8], the authors evaluate the tools supporting profiling, compilation, and hardware description language (HDL). These tools are part of the DelftWorkBench framework.

The hArtes toolchain aims to accept many different languages as inputs, such as Matlab or C. The generated hardware is created to be platform independent. However, only a high end Xilinx Virtex-4 was used for prototyping. The generated software from the toolchain is C code that is annotated with pragmas.

The toolchain is composed of 3 specific tools:

- Algorithm Exploration and Translation Toolbox:

Translate the program (with the help of designers) from the input language to a unified internal description in the C language.

- Design Space Exploration Toolbox:

Divides the program into two parts, one for hardware, the other for software, in an optimal way by using different profilers and cost estimators to find an optimal design.

- System Synthesis Toolbox:

Receives the optimized partitions from the Design Space Exploration Toolbox and generates the hardware code (VHDL) and the program code that runs on it.

The HDL code generation borrows from research done over the past 15 years. It differs from other tools such as Handel-C, Catapult-C, and Impulse-C by targeting a different audience, which is the software designer instead of the hardware designer therefore not requiring in depth hardware knowledge to be able to use. hArtes also targets a broader range of application than previous research efforts and does not impose heavy restrictions on the subset of C that can be used.

The automated HDL generation is done using the DelftWorkBench Automated Reconfigurable VHDL Generator (DWARV). The tool takes as input the pragma annotated C program from the first 2 tools and implements it into hardware using VHDL. There are currently numerous syntax restrictions imposed on the C code but do not cause semantic limitations. The tool is split into two modules, one that generates an intermediate description, which is a dataflow graph (DFG). The other module takes the DFG and generates VHDL code for it. The scheduling used is limited to as soon as possible (ASAP) scheduling. The selected computation model is finite state machine (FSM) based.

In [8], two benchmarks are run to evaluate the toolchain. The first one is a multimedia application involving MJPEG encoding which performs image compression for streaming applications. The other test is a DSP application using G721 encoding for audio. Xilinx boards were used in both cases and the hardware generation was done using a Linux system. Hardware generation times were calculated using the Linux “time” command. The generated VHDL code provides a small enough design to be practical but is not as small as possible had the hardware been generated and optimized by hand. The generated hardware and software in this test provided a 36% speedup for the G721 encoding and 228% speedup for the MJPEG encoding over using the standard, non-extended, PowerPC processor.

Future work for the hArtes toolchain involves providing new profiling criteria such as power usage and memory access. Work will be done to extend the quantitative model with more targets such as delay, interconnect and power. For the compilation, optimizations to fully explore parallelism will be added as well as heavy optimizations to the VHDL generating tool.

The DelftWorkBench Automated Reconfigurable VHDL Generator (DWARV) described in [9] uses a pragma annotated C program as input and generates VHDL code for hardware implementation. DWARV has a broad range of applications and exploits when possible operation parallelism.

Numerous restrictions on the C source code currently apply but work is being done to remove them. The number of applicable domains is not affected by this limitation since algorithms with different characteristics can be generated. The benchmarks showed a 13% to 94% speedup of the maximum speedup possible according to Amdahl's law. Execution on real hardware was done using a MOLEN polymorphic processor.

DWRAV is composed of two modules: one that generates a data flow graph (DFG), while the other one generates VHDL from the DFG. Annotation is used in the C source code to specify which code segments are to be implemented in hardware.

The current limitations of DWARV are:

- Only 1 dimensional memory addressing.
- Structures, unions, and floating point types not supported.
- Iteration statements limited to "for".
- Selection statements limited to "if".
- Control jumps and function calls not implemented.
- No component hardware reuse.

- Pragma annotated C source code needed.

Some of the limitations of theses tool have been eliminated by other projects using the DWRAV tool as a part of their toolchains. This is the case in [8] where their tool accepts a C program (as well as Matlab and other languages) and outputs a pragma annotated C source code that is then used as input to the DWRAV toolchain to generate the VHDL code. The limitation of using only a subset of C constructs can be overcome by changing its construction, for example, rewriting a “while” loop to be a “for” loop.

The benchmark tests run in this paper are the same as in [8], a DSP application using the G721 audio encoding and a multimedia application using MJPEG picture compression. The tool managed to get speedups of 140% to 680% which corresponds to a 13% to 94% of the maximum possible speedup.

2.5 – Image Procession Application Using Automated Hardware Generation

In [10], an image processing application is explored as part of the Cameron project. The Cameron project uses a single-assignment subset of C called SA-C, generates dataflow graphs (DFG), and then generates the VHDL code. It is designed specifically for the image processing domain.

Image processing applications are ideal for parallelizing, both fine and coarse grained and thus ideal for configurable and reconfigurable systems. Developing specific hardware for this task requires in-depth knowledge of hardware design, specifically knowledge of a hardware development language along with knowledge to properly decide how to partition the program between hardware extensions and the main, general purpose processor. Programmers dealing with image processing generally deal solely with software development and have limited knowledge of hardware design. As in [8], the scope of this

project is to provide a tool for software designers and to make the tool usable by people with limited hardware development knowledge.

The Cameron project consists of a graphical programming environment, a high level language, an optimizing compiler, debugging tools, and performance monitoring tools. The project compiles the SA-C code into a DFG which is then compiled into VHDL code. This is the same procedure used by the CARG group with but CARG using C source code instead of SA-C.

The first step in translating the DFG to VHDL is the classification of the DFG nodes into various categories. The categories are:

- Run-time input nodes:

  These are not translated into VHDL directly but store the address of the location of the run-time data. These nodes are discovered at runtime and are therefore only used for reconfigurable systems, not the compile time configurable system COINS is targeting.

- Generator nodes:

  These nodes contain information on window size, shape, and step size and are used at compile time.

- Loop body nodes:

  These nodes specify the operations that are to be performed by the inner loop body and to generate its VHDL code.

- Reduction nodes:

  These nodes specify the parameters to select and instantiate the reduction nodes.

Using this information, there are three main parts for the translation process.

The first step is to identify the inner body loops which are the part of the DFG between the inputs and outputs of the loop body nodes. It is compiled into a VHDL component. Next, the loop generator and collection nodes are generated by using VHDL components from a library using appropriate parameters. The last step specifies the interconnections between the VHDL components from the first step and second step using two top-level VHDL modules. These two modules glue together the components of the final design.

The second step involves the translation of the inner body loops using a traversal of the DFG. A VHDL component is created using the loop generator output for input and connecting its output to the input of the data collector. Currently, the project can only produce entirely combinational inner loop bodies however plans for implementing multi-cycle functions have been made.

The third and final step is the implementation of the other components into the design. These components are the data generators and collectors. They are created from VHDL components that are selected from a module library and are parameterized with values from the DFG.

An example is provided using the Prewitt Algorithm described in [10] is a common algorithm used in image processing. It is inherently parallel due to each 3x3 convolution being independent, involves constant masks which allow significant optimization before hardware implementation, requires some operations that are difficult to do on FPGA such as square roots, and uses streaming data. The program is written in SA-C and consists of 19 lines of code. The generated VHDL code is over 5000 lines long. The program is tested on four different systems:

- Automatic VHDL code generation without stripmining resulting in a single inner loop body.
- Automatic VHDL code generation with stripmining resulting in two inner loop bodies
- Manual VHDL writing of a system equivalent to the stripmining automatic case
- On a 450Mhz Pentium PC computer system.

The manually written VHDL code was the fastest, running over 6x faster than any of the other cases requiring only 4.66ms. The Pentium system required 28.4ms whereas the two automatically generated VHDL code systems required 31.57ms with stripmining and 60.61ms without stripmining. The automatically generated VHDL code ran a lot slower than the hand written code, however, it took a few hours to generate the VHDL code automatically whereas multiple weeks were required to hand code the system.

Future work for the project involves optimizing the generated VHDL code as currently far more effort went into functionality over optimization and the VHDL code generation was secondary in importance to the compiler. Lookup tables and inner loop body pipelining are the main areas looked at for improving the performance.

2.6 – Performance of Automated Hardware Code

Generation of VHDL code for custom instructions has been studied and continues to be studied by many University research groups. Most groups however use special versions of the C language with annotations, such as in [9] and [10]. A way to use standard C with these tools is explained in [8], which takes regular C source code (and other languages such as Matlab code) and translates it into the version of C, with pragma annotations, which can be used as input by the DWARV that then generates the VHDL code for hardware synthesis.

Many of the tools created target specific applications, such as image processing in [10]. Other tools aim to be usable in all applications such as those described in [8] and [9]. Although the code generated by [10] is a lot slower than using a normal processor, this can be explained by the early state of the hardware generation tool. Other tools such as those tested in [6], [9], and [10] generate considerable

speedups. Although the speedups are slower than human generated and optimized VHDL code, the amount of time required to create the system take only a couple hours versus weeks for the code written by hand.

All the tools studied used a form of a graph as input for the VHDL code generation. This is the same as the SHIRA toolchain which uses DFGs. Different methodologies are discussed and will be explored in this project. There is however no component reuse implemented in any of the tools studied, which is one of the main objectives of the SHIRA toolchain. Component reuse is planned for the future however for most of the tools.

3 – The SHIRA Toolchain

3.1 –Overview of the SHIRA Toolchain Development

The Computer Architecture Research group at the University of Ottawa is currently working on Software/Hardware Implementation and Research Architecture (SHIRA) toolchain. The toolchain takes a C program as input, finds optimizations for it subject to constraints, and generates hardware for that C program. The generated hardware code is to be programmed onto an FPGA that then runs the C program compiled to use that hardware. An overview of the SHIRA toolchain is shown in Figure 2.

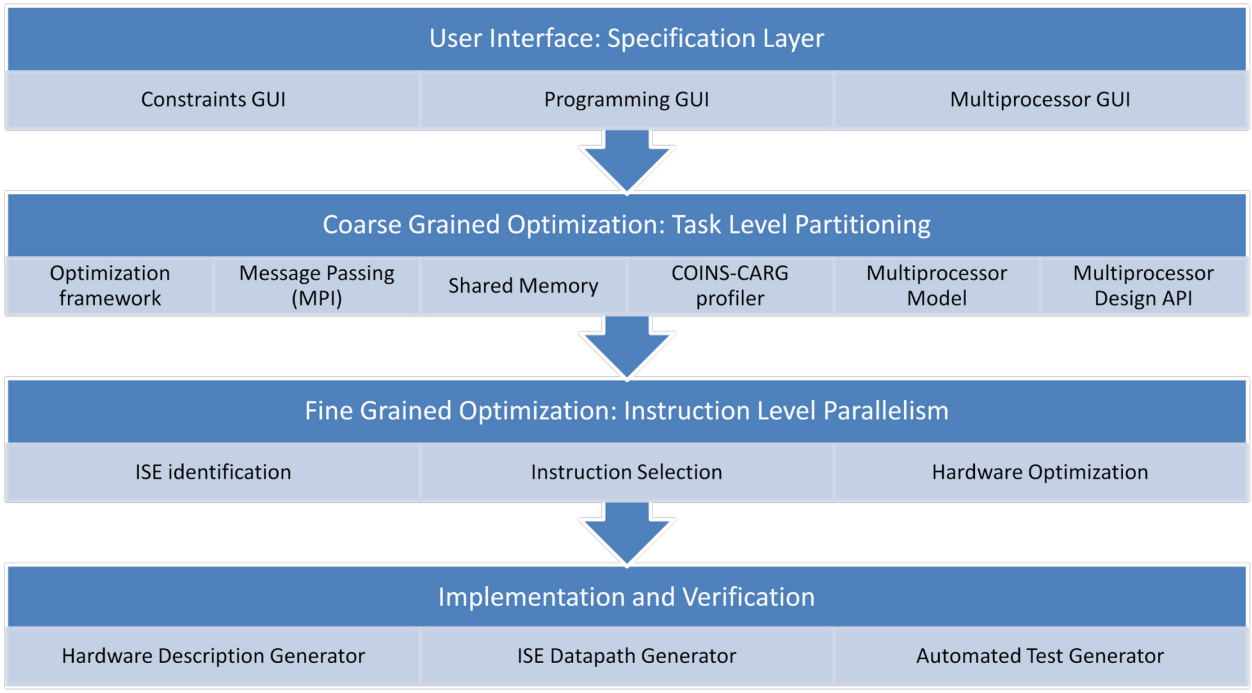

Figure 2 - SHIRA Toolchain [1]

The top layer of the toolchain is the Graphical User Interface (GUI) and is currently being worked on by John-Marc Desmarais. It is an Eclipse plugin and already partly implemented. It aims to accomplish the following goals:

- Define constraints including constraints on chip size usage, number of processors, memory size, I/O constraints, and power usage.
- Allow the user to decide what components should be used on the FPGA such memory, performance counters, JTAG interfaces, and others.
- Define the connection network of the processors and other components such as memory. This is currently accomplished by using the Altera SOPC Builder.
- Coprocessor usage for specialized tasks such as fuzzy logic.

The coarse grained and fine grained optimization is the work of Daniel Shapiro in his thesis work in (1). The module created for this thesis relies heavily on this work. The major work accomplished so far is the Instruction Set Extension (ISE) identification and selection. This is accomplished by performing a pass on the COINS low intermediate representation. This pass identifies and selects an optimal custom instruction set and orders by the level of importance of each identified CI. The basic blocks are converted to Dataflow Graphs (DFGs) which provide the information of a custom instruction. The DFGs are modeled using a Mixed Integer Linear Program (MILP). The MILP is used to model the following constraints [1]:

- Convexity
- Hardware Size
- Number of Nodes
- Disabled Nodes
- Disabled ISEs
- Constant Feeding ISEs
- I/O Constraints

The bottom level of SHIRA is the implementation and verification of the hardware and software. This module is developed and discussed in this thesis. It implements the CIs identified by the MILP. This module is integrated in the instruction selection section of the SHIRA toolchain. InstructionSelection.java was modified to include a section to call the functions for the hardware generation when it is enabled. It calls functions created for this module which are in the newly developed classes CIManipulation.java and CreateVHDL.java. CIManipulation.java was created to contain the functions required to extract and organize the DFG data as well as the functions for scheduling. CreateVHDL.java was created to contain the functions required for writing the different portions of the VHDL source file. The generated VHDL code is targeted for implementation as Custom Instructions (CI) on a NIOS II processor based system. The integration and development of this module was done for this thesis and allows the SHIRA toolchain to generate and verify the synthesizable VHDL code for the identified CIs to improve the performance of the C programs. This module is covered in more detail in chapter 4 of this thesis.

Other modules and additions currently being developed for SHIRA include are high level parallelism identification and the automated generation of a coprocessor for use in fuzzy logic applications.

3.2 – Software used in the SHIRA Toolchain

The SHIRA toolchain attempts to use free and open source software whenever possible. The toolchain uses the free Eclipse IDE as the front end for the GUI as well as our integrated development environment for developing and testing Java source code. The COINS C compiler, is written in Java, is also free and open source and was integrated into CARGs project. COINS is used to generate IRs of the C program. It will also be used to compile the C program for the customized hardware in the future.

Two separate programs are used for solving the Mixed Integer Linear Programs (MILPs) in [1]. Originally the commercial optimization software program LINGO was used with good results. Later on, the addition of the free LP Solve program was added to the toolchain and the solver to be used can be decided by the user.

The compilation of the VHDL code is done using Altera's Quartus II Software and NIOS II IDE. This software is commercial and is closed source. A limited free version is available for academic and trial use. Using the Quartus II program, a hardware system incorporating a NIOS II CPU, the CIs, and memory is generated using SOPC Builder. The Quartus II software can then compile the new system for the targeted FPGA and can be used to program the chip. The FPGA used in the development and verification of this module is the Cyclon II EP2C35F672C6N found on the DE2 development board. The NIOS II IDE is then used to compile the C code for the new hardware and to run the software on the FPGA.

4 – Automatic Generation of VHDL Code

4.1 – Overview of VHDL Generation System

This module of the SHIRA toolchain takes as input a data flow graph (DFG) and creates synthesizable VHDL code to create that DFG in hardware. The VHDL code generated is adapted to Altera's products. The generated VHDL code is to be added to a NIOS II processor as a custom instruction (CI). This allows the processor to call this instruction instead of using its own resources, therefore optimizing the system. The CI is implemented using Altera's library of parameterized (LPM) functions. LPM functions are provided by the manufacturer of the FPGA to speed up hardware development and include many common modules used in VHDL design for arithmetic operations, interfaces, storage elements, and many other elements. The flow of the hardware generation portion of the SHIRA toolchain is shown in Figure 3.

The software used for this project:

- Lpsolve for optimization.
- Eclipse IDE for Java development of the COINS C compiler
- Quartus II 9.1sp2 for embedded system generation and compilation of VHDL code
- SOPC builder (part of Quartus II) for generating a NIOS II system and adding the CI to the NIOS II processor.
- NIOS II IDE for creating the software to run the custom instruction.

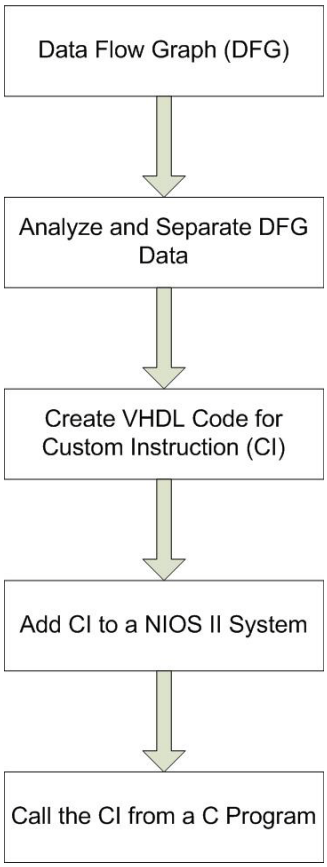

Figure 3 - **SHIRA Hardware Generation Flow Chart**

The following Java classes were added to the SHIRA toolchain or were extensively modified.

- CIManipulation.java was devised, implemented and added to the SHIRA toolchain to do analysis and classify the data of the DFG. It contains functions to separate the DFG data, analyze and catalogue the data, and functions to get the data.
- CreateVHDL.java was designed and added to the SHIRA toolchain to generate the VHDL code for different parts of the VHDL code.
- InstructionSelection.java was modified to get the DFG into a string form, call functions from CIManipulation.java to extract data and analyze it, and then call functions from CreateVHDL.java to create the VHDL file.

4.2 – Data Flow Graph Representation

The first step of the VHDL generation section is to retrieve and extract the DFG information. Figure 4 shows a DFG for the following simple operation: y = (a + b) * (c – d).

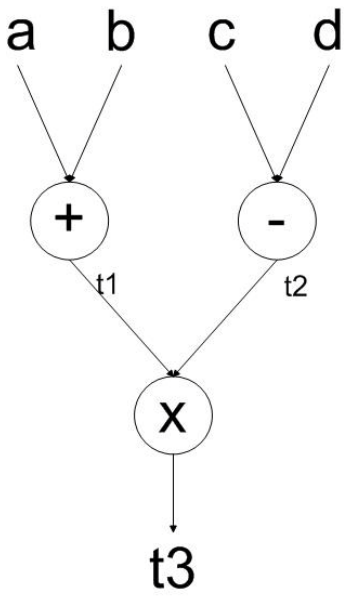

Figure 4 - Simple Data Flow Graph

This information however comes encoded in a string and must be analyzed and separated. The information about a node in the DFG is contained on a line in the string formatted as follows:

```
output = operation (inputA, inputB)
```

Each node is on a new line within the string and therefore, the string contains as many lines as there are nodes in the DFG.

The operations contained in the DFG string are not ordered in any way. The operation code is the instruction number of the operation on the processor, as shown in the first column of ht table below. The following operations are supported by SHIRA hardware generation:

Operation #	Operation Code	Operation
y = 9(a)	NEG	Negate: y = -a
y = 10(a,b)	ADD	Add: y = a + b
y = 11(a,b)	SUB	Subtraction: y = a - b
y = 12(a,b)	MUL	Multiply: y = a * b
y = 13(a,b)	DIVS	Signed Division: y = a/b
y = 14(a,b)	DIVU	Unsigned Division: y = a/b
y = 15(a,b)	MODS	Signed Modulus: y = a%b
y = 16(a,b)	MODU	Unsigned Modulus: y = a%b

The codes of the input and output operators are assigned by the optimization algorithms of SHIRA. Examples of codes assigned to the input and output operators are:

```
a: 109970892
b: 19111827
c: 148194546
d: 277277265
t1: 1685984552
t2: 1556901833
t3: 1427819114
```

The string of the encoded DFG, which is created by SHIRA for the DFG of Figure 4, is shown below.

```
1556901833=11(148194546,277277265)
1427819114=12(1685984552,1556901833)
1685984552=10(109970892,19111827)
```

4.3 – Analysis and Separation Data Flow Graph Information

The analysis and separation of the DFG data is done by functions from CIManipulation.java. These functions are called from a section added in InstructionSelection.java when the GENERATE_HARDWARE flag is set to true in CompilePareters.java. CIManipulation.java takes the DFG string, extracts, organizes

the information, and schedules the operations so that the DFG can be created in hardware. It is called by the InstructionSelection.java class with the DFG string as its variable. CIManipulation.java then separates the data into ArrayLists for inputs, operations, and an integer for the output. It then analyzes this data to find the global inputs to the custom instruction, the interior I/O registers, and the global output of the CI. CIManipulation.java also figures out what LPM components to define and how many input stages are needed.

The defining of each LPM component should only be done once. There are also some components that are used by multiple operations but have different generics. In this project, the LPM_ADD_SUB component is used for both the ADD and SUB instruction. The LPM_DIVIDE module is also used for DIVS, DIVU, MODS, and MODU. The operation for these components is specified when the generic and port map is specified. CIManipulation.java creates an Arraylist of the components that need to be defined. A form of memory storage is needed to store the value of an operation when components are being reused and the component for which the value is an input is busy. A 32 bit, positive edge triggered flip flop is used for the memory storage component. The table below shows what LPM component corresponds for each of the following operation codes used in the SHIRA toolchain:

Operation	LPM Component
9 - NEG	LPM_INV
10 - ADD	LPM_ADD_SUB
11 - SUB	LPM_ADD_SUB
12 - MUL	LPM_MULT
13 - DIVS	LPM_DIVIDE
14 - DIVU	LPM_DIVIDE
15 - MODS	LPM_DIVIDE
16 – MODU	LPM_DIVIDE
Memory - FlopFlop	LPM_FF

The NIOS II processor allows for CI's to be added to the processor's instruction set. 256 instructions can be added with 2 inputs and 1 output each. Instructions with more than 2 inputs can be added but reduce the total number of instructions that can be added. When more than 2 inputs are used, the inputs must be loaded in stages and the CI becomes a multicycle CI as explained in [11]. The number of input stages is required to be known before generating the VHDL code and is determined in CIManipulation. The formula used is:

```
(Number of Global Inputs to CI) /2
```

An example of calling a CI with 2 inputs (1 stage):

```
y = ALT_CI_CI_HARDWARE_INST(a,b)
```

An example of calling a CI with 4 inputs (2 stages):

```
ALT_CI_CI_HARDWARE_INST(0,a,b)
y =  ALT_CI_CI_HARDWARE_INST(1,c,d)
```

The order for inputting the values is provided the SHIRA toolchain as the order cannot be determined from the original C program.

4.4 – Creating VHDL Code for the Custom Instruction

The VHDL code is created by using the data and analysis information collected from the functions of CIManipulation.java and calling functions from CreateVHDL.java. CreateVHDL.java doesn't do any data processing; it is used solely for creating strings that are used for writing the VHDL file.

The header of the VHDL file does not change from CI to CI apart for n, which changes are needed when the CI has more than two inputs and the NIOS II multicycle mode is required. The header contains the libraries to include and the entity definition of the CI, which is the standard used for the NIOS II processor.

The signals used in the CI are then defined. The FSM states are represented by enumerated data type as suggested by [12]. This is done next, with the number of stages already determined and passed along as the function's variable. If there is no component reuse, the number of states will always be 2. The inputs, interior i/o, and outputs are then defined in VHDL code. These are defined by calling the appropriate function in CreateVHDL.java with the number of the input as its parameter. The number will be preceded with "io_" to help readability and because VHDL does not allow signals to start with a number. The null_io signals are created when division and modulo operations are present since the LPM_DIVDE module has 2 outputs, one for the remainder (modulus) and one for the quotient.

LPM components are then defined. Each component only needs to be defined once and can be shared by multiple operations. The LPM components to be defined is determined earlier using CIManipulation.java.

The LPM components can now be instantiated. One component is instantiated for each operation and the generics are created. The generics are done following the suggestions in [12] and all have 32 bit inputs and 32 bit outputs. The LPM_MULT must have an output double in size of the input, so it is defined as 64 bits, however, only the lower 32 bits are used because the size of the input and output port in the NIOS II processor are limited to 32 bits each. To allow for more than 32 bits, the output would have to be read twice and stored into to separate variables in the C program. The port map is also completed in this phase and the instantiated LPM ports are mapped to the corresponding I/O port. The function is called for each operation and has inputs for op number, inputA, inputB, and output of the LPM component. A null_io output might be required here if the LPM function has more than one output.

The finite state machine is generated now. It is separated into 4 sections. The 1st section is the header that doesn't change. The 2nd section generates the code for the different input stages. The 3rd section generates the code for the output, and the 4th section generates the ending part of code for the VHDL file. Only the 2nd and 3rd parts have variables, the 2nd part takes the inputs and the stage number as input, and the 3rd part takes the CI's output.

The order of providing the values for the inputs of the CI is given in the output console along with information on each stage of the VHDL code generation. The variable name and type is provided. An example for the DFG in figure 4 is provided below.

```
Started VHDL hardware generation
 - VHDL Header Generation
 - VHDL FSM States Generation
 - VHDL IO Signal Generation
 - VHDL Defining needed LPM Components Generation
   - VHDL Hardware Instantiation for Operation #12(MUL) Generation
   - VHDL Hardware Instantiation for Operation #11(SUB) Generation
 - VHDL Generic and Port Maps Generation
   - VHDL Generic and Port Map Generation for Operation #12(MUL)
   - VHDL Generic and Port Map Generation for Operation #11(SUB)
   - VHDL Generic and Port Map Generation for Operation #10(ADD)
 - VHDL FSM Generation
numberInputStages = 2
Custom Instruction Calling Order from Nios:
```

```
ALT_CI_CI_HARDWARE_INST(0,(REGI32"c_divexI32_1"),(REGI32"d_divexI32_1")
ALT_CI_CI_HARDWARE_INST(1,(REGI32"a_divexI32_1"),(REGI32"b_divexI32_1")
```

As explained before, the ADD and SUB operations use the same LPM module (LPM_ADD_SUB) but are two separate components. The actual operation is defined in the generic mapping. As an example, the code foe defining the component LPM_ADD_SUB is:

```
      COMPONENT lpm_add_sub
            GENERIC     (     lpm_direction           :     STRING;
                              lpm_hint                :     STRING;
                              lpm_representation      :     STRING;
                              lpm_type                :     STRING;
                              lpm_width               :     NATURAL
                        );
            PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                              datab :     IN    STD_LOGIC_VECTOR(31 DOWNTO
      0);
                              result      :     OUT   STD_LOGIC_VECTOR(31
      DOWNTO 0)
                        );
      END COMPONENT;
```

This is only done once in the VHDL source code. For each adder and subtractor, a generic map and port map must be created. The code for the generic map and port map of an adder and a subtractor is shown below :

```
lpm_add_0   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
                              lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                              lpm_representation      =>    "SIGNED",
                              lpm_type                =>    "LPM_ADD_SUB",
                              lpm_width               =>    32
                              )
      PORT MAP    (add_0_ina, add_0_inb, add_0_out);

lpm_sub_0   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "SUB",
                              lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                              lpm_representation      =>    "SIGNED",
                              lpm_type                =>    "LPM_ADD_SUB",
                              lpm_width               =>    32
                              )
      PORT MAP    (add_0_ina, add_0_inb, add_0_out);
```

If additional adders are needed, then they must be defined again as "`lpm_add_1 : lpm_add_sub`", "`lpm_add_2 : lpm_add_sub`", and so on, along with their generic map and port map. The same procedure is followed for all the other standard operations (SUB, MUL, DIVS, etc…).

The C code from which the DFG of Fig. 4 was derived, and subsequently is used to generate the VHDL code, is shown below:

```
int main(int argc, char *argv[])
{

    t1 = a+b;
    t2 = c-d;
    t3 = t1*t2;
    return t3;
}
```

The VHDL generated is shown in the appendix.

4.5 - Adding the Custom Instruction to a NIOS II System.

The generated VHDL code for the CI follows the specifications from [13] for the NIOS II processor. The CI input ports are named dataa and datab and the output port is named result. The generated CIs are multicycle therefore they use the clk, clk_en, reset, and start inputs along with the done ouput. If the CI has more than two inputs, an extended CI is required and the n variable is needed as an input. This naming convention is done automatically by the hardware generation software and requires no input from the user.

This VHDL code must be added to the instruction set of the NIOS II/e processor in a simple NIOS II system. To do this, first a new Quartus II project needs to be started. The VHDL file generated by SHIRA should then be added to the project and analyzed by selecting from the menu bar Processing -> "Analyze Current File" in Quartus II. Then, a simple NIOS II system needs to be created by the use and is composed of the following:

- NIOS II/e CPU with the CI added. The NIOS II/e CPU is the generic processor and the simplest one available.
- On chip memory of 46KB, used to store the software that runs on the NIOS II
- Sysid, used to verify that the hardware programmed on the FPGA is the expected version. In other words, to verify that the C program matches the version of the hardware it was made for.
- Performance Counter, used to calculated the performance of the system with and without the custom instructions
- JTAG UART, used for programming the software (C program) on the FPGA as well as allowing for communication between PC software and the program on the FPGA (i.e. for input of variables and output of results to the PC screen).
-

To add the NIOS II/e processor with a custom instruction, the other components should first be added. The components are added by finding each one in the component library on the left side of the SOPC Builder and double clicking on them. In the windows that pop up, parameters of the components must be set as follows:

- On-Chip Memory: “Total memory size” = 46KB, rest is default
- System ID Peripheral: default settings
- Performance Counter: “Number of simultaneously-measured sections” = 3
- JTAG UART: default settings

The system without the NIOS II processor is shown in Figure 5.

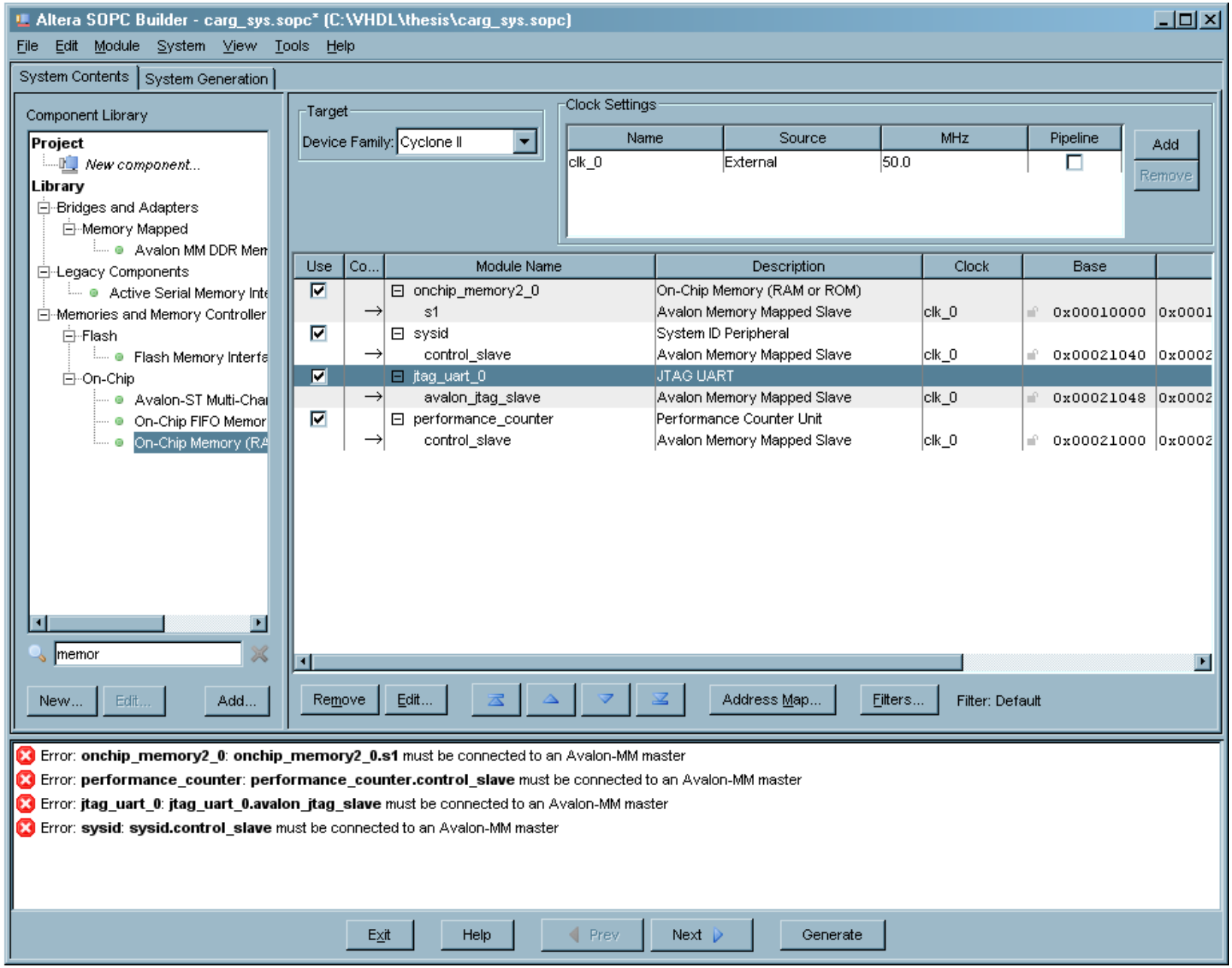

Figure 5 - SOPC Builder System without NIOS II CPU

To add the NIOS II/e processor with the CI, first find the “NIOS II Processor” on the left side of the SOPC builder and double click. Select the simplest processor, the NIOS II/e and set the Reset Vector and Exception Vector to the onchip_memory from the drop down menu. To add the CI to the instruction set of the processor, select “Custom Instructions” from the top menu bar and click Import to create the CI as a component to add to the processor. This will then open the “Component Editor” as shown in Figure 6. Click on Add and browse for the VHDL file created by the SHIRA module and click open, the name of the file is by default CI_Hardware.vhd. As the top level module, select CI_HARDWARE.

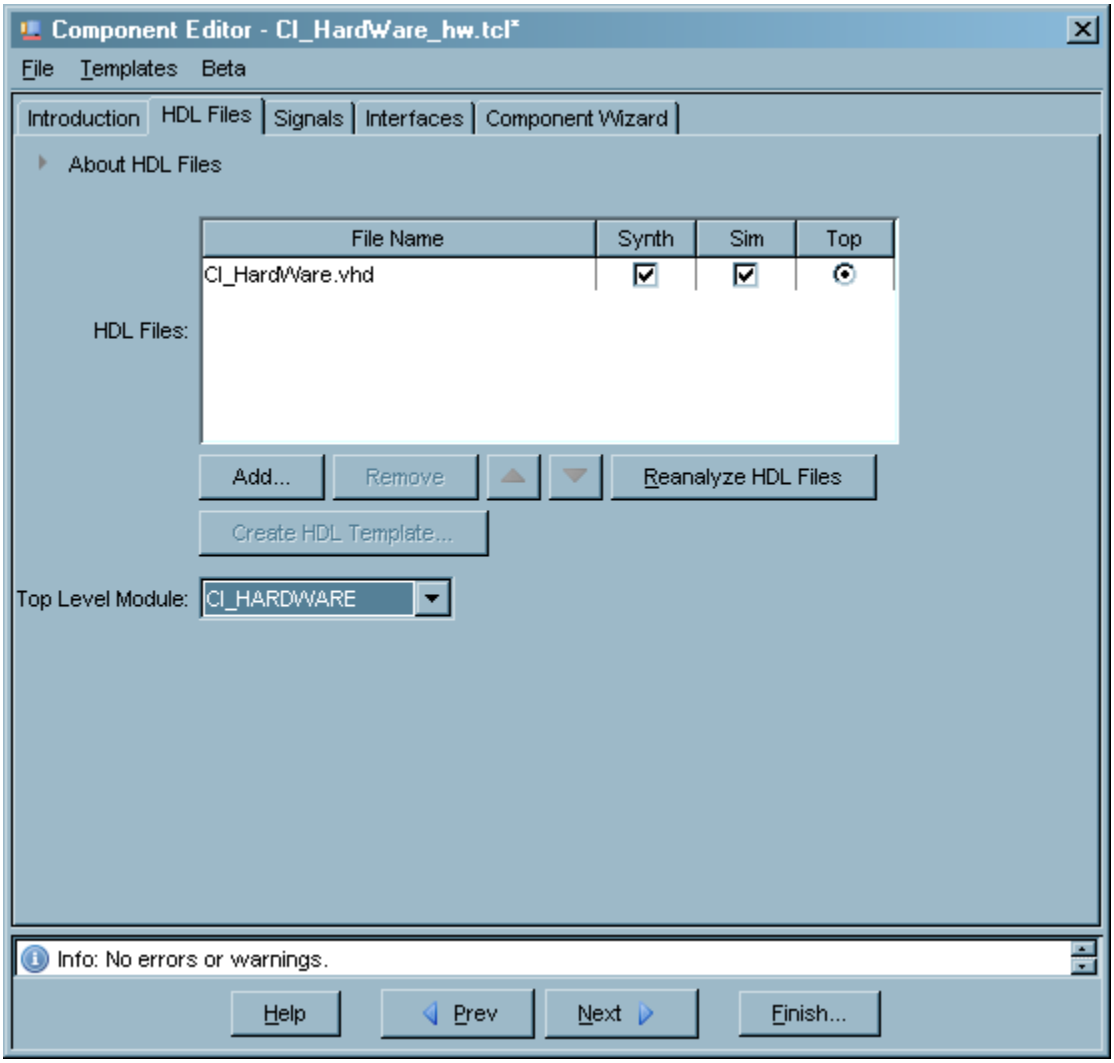

Figure 6 - Creating the CI Component

Click Finish when done. Now the CI can be added to the instruction set of the NIOS II/e processor. Select the CI from the menu on the left as shown in Figure 7 and click Add. Finish adding the NIOS II/e processor to the system by clicking Finish to return to the SOPC builder. The NIOS II system is now complete along with the NIOS II/e processor containing the CI. Auto-assign the base addresses and IRQ from the System menu. The NIOS II system can now be generated, compiled, and programmed on the Altera FPGA.

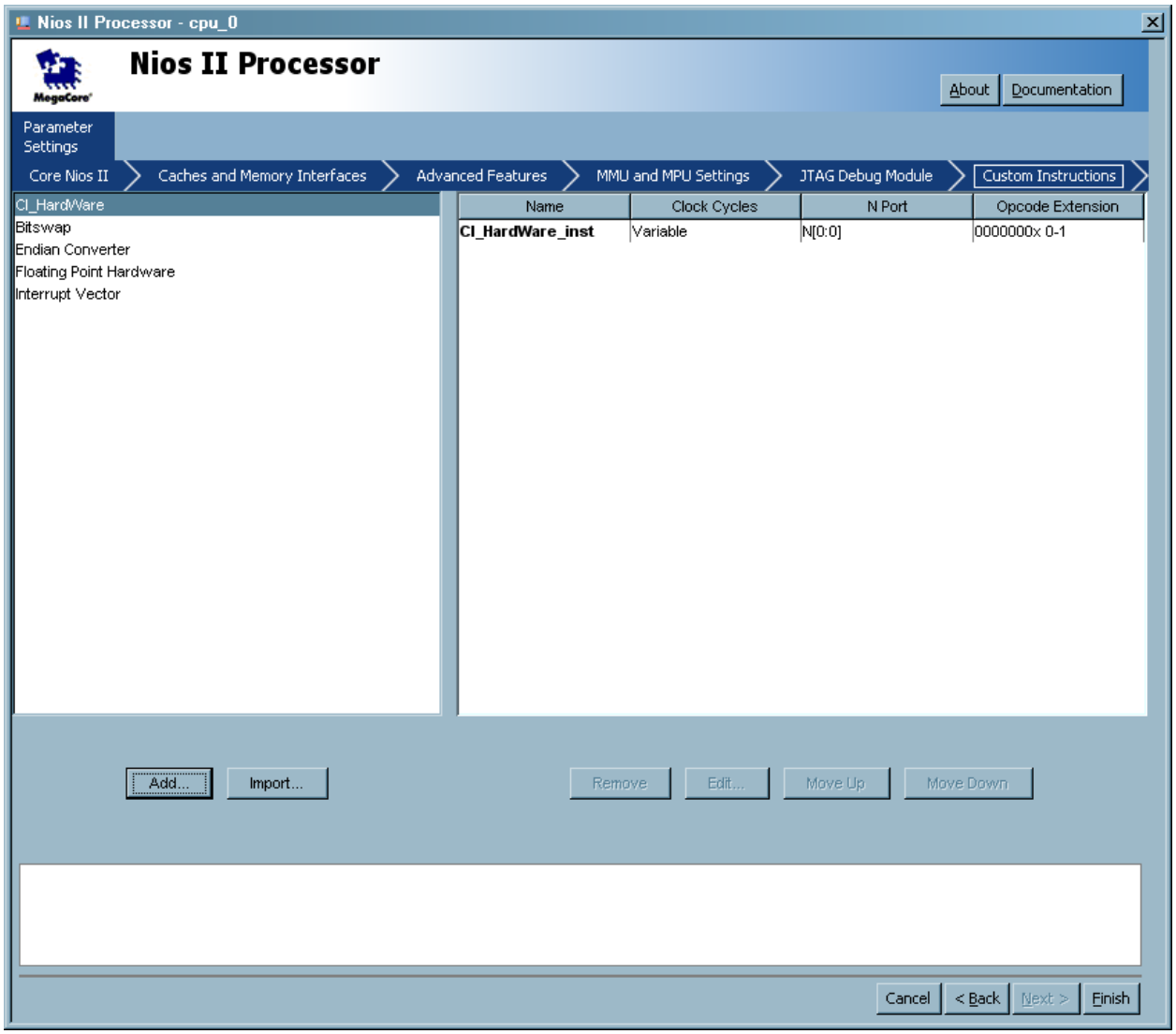

Figure 7 - Adding the CI to a NIOS II/e Processor

4.6 - Calling the Custom Instruction from a C program

Once the FPGA has been programmed, a C program is used to call the operation. The NIOS II IDE is used for developing the software to run on the new hardware system. A new NIOS II application and Board Support Package BSP from template is created using the Hello World template. The C file is then modified to call the CI as follows:

```
int main()
{
	int y = 0;

	int t1,t2;
	int a=6,b=3,c=10,d=5;


	ALT_CI_CI_HARDWARE_INST(0,c,d);
	y = ALT_CI_CI_HARDWARE_INST(1,a,b);

	printf("Result is %d \n", y);

	return 0;

}
```

The output of this code will give the result of the CI with the inputs shown above. The operation is y = (a+b)*(c-d) => y = (6 + 3) * (10 – 5). The output of the program gives:

```
Result is 45
```

4.7 – Component Reuse

The program created by the technique explained so far uses unconstrained scheduled and has no constraints on chip area used. A separate component is created for every operation and there is no component reuse. The toolchain has been expended to implement any data flow graph (DFG) using only one component for each operation. The as soon as possible (ASAP) scheduling algorithm as described In [14] was implemented.

When component reuse is selected, different functions are used for parts of the VHDL code generation. Component reuse is enabled by setting the maximum number of each component to 1 in CIManipulaton.java as shown in the code fragment below.

```
private static int maxInv = 1;
private static int maxAdd = 1;
private static int maxSub = 1;
private static int maxMul = 1;
private static int maxDivS = 1;
private static int maxDivU = 1;
private static int maxModS = 1;
private static int maxModU = 1;
```

Each component has to be declared as in the unconstrained case, but will also be instantiated only once in the generics VHDL generation section.

Since values have to be stored in between operations, a storage element is required to store intermediate values. A flipflop is used in this scope due to its small size and appropriate functionality. A flipflop is created for each input and output for the generic of each operation.

The input, output, and buffer generation is also changed to include the I/O for the flipflops. The inputs and outputs for the other components is no longer generated because the flipflops are mapped to those ports. The mapping is done in the generics and port map.

The finite state machine is significantly more complex when component reuse is enabled. More than the two states are now required. State 0 is used to input all the values from the program but is now also used to return the result of the previous operation when the first input of the new custom instruction call is done, that is when the state is 0 and n = 0. The result of the operation is retrieved by using the following c code.

```
y = ALT_CI_CI_HARDWARE_INST(1,a,b);
```

The scheduling is accomplished using the as soon as possible algorithm (ASAP). During the generation of the FSM, each operation for each inside the DFG is loaded into a priority queue. When a component for an operation becomes free, it then removes the inputs for the next node on the DFG from the priority queue and completes the operation.

5 – Generation and Results

5.1 – Testing single types of operations #1

5.1.1 – Test Program

This test program demonstrates the performance of the custom instruction that only uses one type of operation. It runs on the NIOS II/e system described in section 5.5 and the performance counter component provided by Altera to calculate the speed (that number of cycles). It is run on three different hardware platforms: Using only the standard NIOS II/e processor, the DFG to VHDL generation tool output with unlimited hardware space, and the DFG to VHDL tool generation tool output with a maximum of 1 component used. The arithmetic operation done for this experiment is:

y = (a+b) * (c+d)

The C source code for the operation above is:

//add3x.c

```
#include <stdio.h>

  int a = 0;
  int b = 0;
  int c = 0;
  int d = 0;
  int y = 0;
  int t1 = 0;
  int t2 = 0;
  void prof(int);

float main(int argc, char *argv[])
{
      t1 = a + b;
      t2 = c + d;
      y = t1 + t2;

      return y;
}

void prof(int num)
{
  printf("x%dx\n",num);
}
```

The first part of the SHIRA toolchain results in the following output of the intermediate representation (IR) of the program as a DFG.

```
1556901833=10(148194546,277277265)
1306952932=10(1685984552,1556901833)
1685984552=10(109970892,19111827)
```

The order for entering the inputs is:

```
ALT_CI_CI_HARDWARE_INST(0,(REGI32"c_divexI32_1"),(REGI32"d_divexI32_1")
ALT_CI_CI_HARDWARE_INST(1,(REGI32"a_divexI32_1"),(REGI32"b_divexI32_1")
```

This conversion is accomplished by using the conventions defined in [1]. It corresponds to the data flow graph shown in Figure 8.

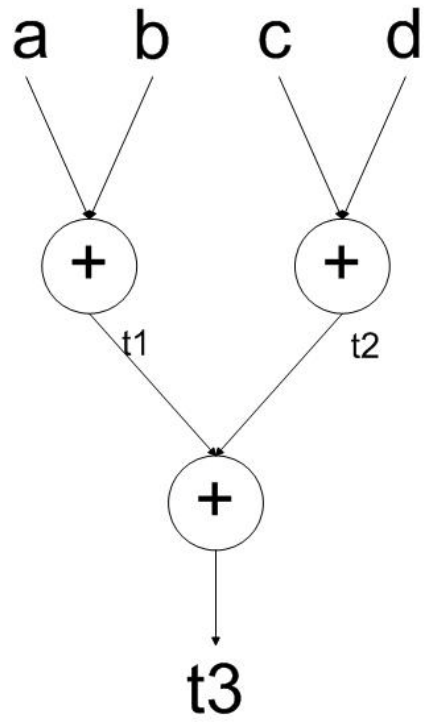

Figure 8 - Data Flow Graph for add3x.c

When using only one adder, and 2 registers, the DFG becomes as show in Figure 9 below:

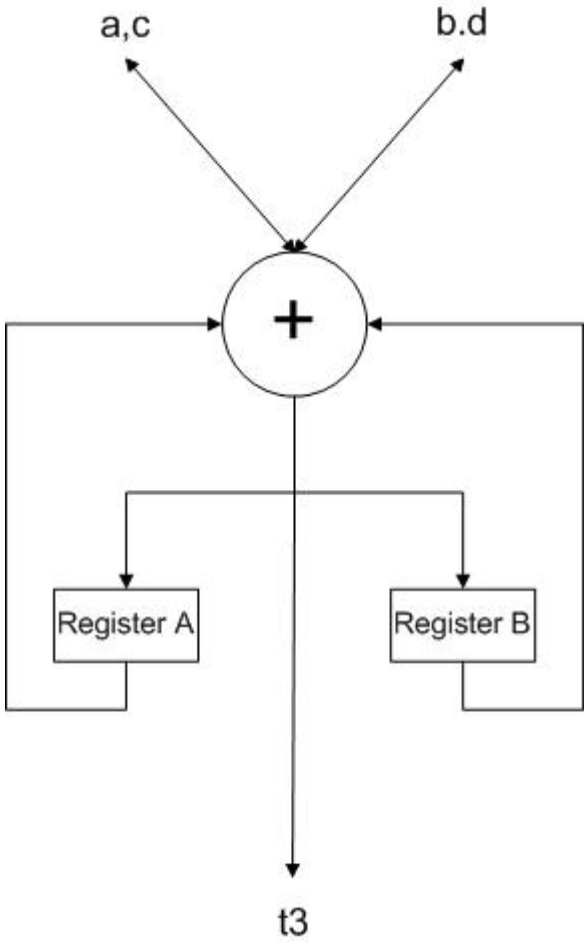

Figure 9 - Data Flow Graph add3x with Component Reuse

5.1.2 –Hardware System

The FPGA board used to run this experiment is an Altera DE2 development board using a Cycle II FPGA with 35k logic elements (EP2C35F672C6). The FPGA is programmed using the hardware system that is shown in Figure 10 below and explained in detail in section 5.5. The custom instruction does not appear in this list since it is added inside the configuration of the CPU.

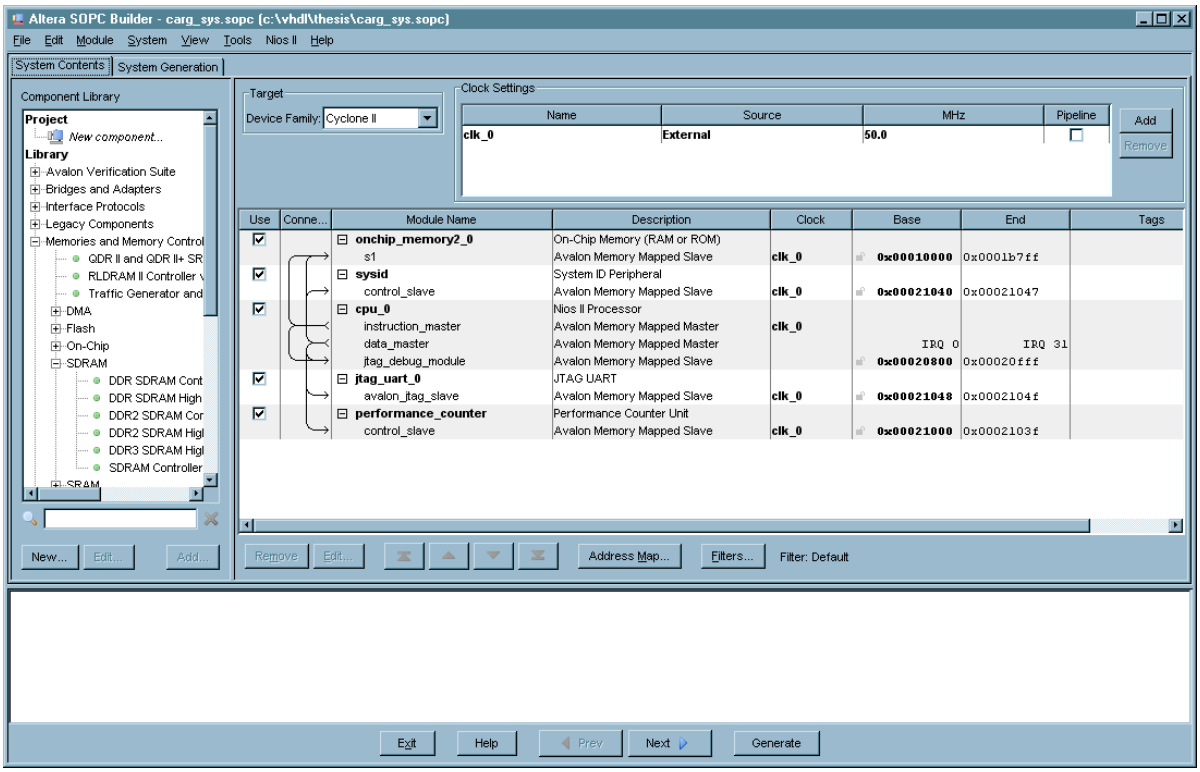

Figure 10 - NIOS II System for add3x.c

5.1.3 – Software running on the Hardware

The NIOS II IDE is used to develop a program written in C that runs on the newly created system on the FPGA. The main section of the source code is show below: The code first runs the instruction without using the custom instruction and then while using the custom instruction, finally printing the results. #1 time is using the native NIOS 2/e instruction set and the #2 time is while using the custom instruction. The operation is (6+3) + (10 + 5).

```
#include <stdio.h>
#include "system.h"
#include <altera_avalon_performance_counter.h>

int main()
{
      int y = 0;

      int t1,t2;
```

```
	int a=6,b=3,c=10,d=5;

	PERF_RESET(PERFORMANCE_COUNTER_BASE);
	PERF_START_MEASURING(PERFORMANCE_COUNTER_BASE);
	PERF_BEGIN(PERFORMANCE_COUNTER_BASE,1);

	t1 = a+b;
	t2 = c+d;
	y = t1 + t2;


	PERF_END(PERFORMANCE_COUNTER_BASE,1);
	printf("Result is %d \n", y);

	PERF_BEGIN(PERFORMANCE_COUNTER_BASE,2);

	ALT_CI_CI_HARDWARE_INST(0,c,d);
	y = ALT_CI_CI_HARDWARE_INST(1,a,b);


	PERF_END(PERFORMANCE_COUNTER_BASE,2);
	PERF_STOP_MEASURING(PERFORMANCE_COUNTER_BASE);

	printf("Result is %d \n", y);

	printf("#1 time - %d\n",
perf_get_section_time(PERFORMANCE_COUNTER_BASE,1));
	printf("#2 time - %d\n",
perf_get_section_time(PERFORMANCE_COUNTER_BASE,2));

	return 0;
}
```

5.1.4 – Tests and Results

The output without component reuse is:

```
Result #1 is 24
Result #2 is 24
#1 time - 110
#2 time – 73
```

The output with component reuse is:

```
Result #1 is 24
Result #2 is 24
#1 time - 110
#2 time – 74
```

The results of the test runs including logic elements used and number of cycles used is shown in the table below.

	No Custom Instruction	Custom Instruction with unlimited space	Custom Instruction with only 1 adder
Total Logic Elements	2130	2401	2317
- Total Combinational Functions	2041	2186	2188
- Dedicated Logic Registers	1127	1291	1293
- Total Registers	1127	1291	1293
- Total Memory Bits	387072	387072	387072
Number of Clock Cycles	110	75	74

The results show a an increase in speed when a custom instruction is used, using 75 cycles with no component reuse, 76 cycles when only one adder is being used, and 110 cycles when using the normal powers of the NIOS II/e CPU. This gives an increase in speed of (110/75) = 1.47 times. When using only 1 adder which forces component reuse, the increase in speed is slightly higher at 110/74 = 1.49 times. Figure 11 shows a graph of these results.

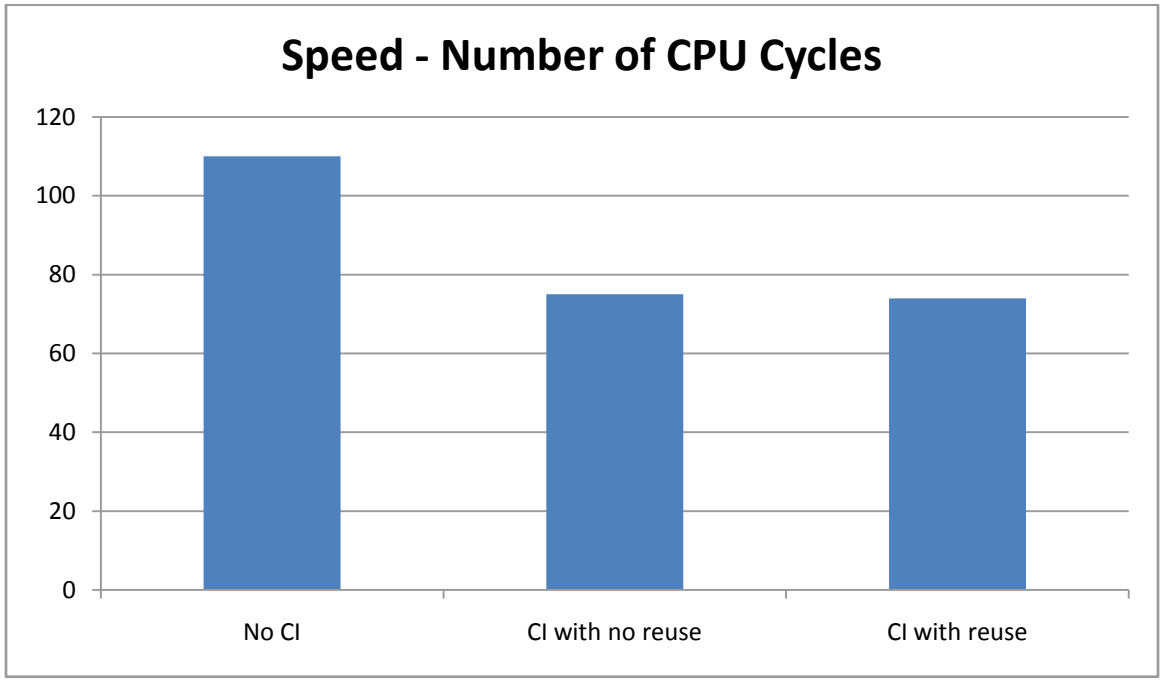

Figure 11 - Graph of CPU Cycles Used for Test #1

The chip area used increases when a custom instruction is added. 2130 logic elements are used for the simple NIOS II system when no custom instruction is added. The number of logic elements goes up to 2406 when the generated custom instruction is added. The number of logic elements goes down to 2333 when only one adder is re-used multiple times. Figure 12 shows a graph of the chip area used.

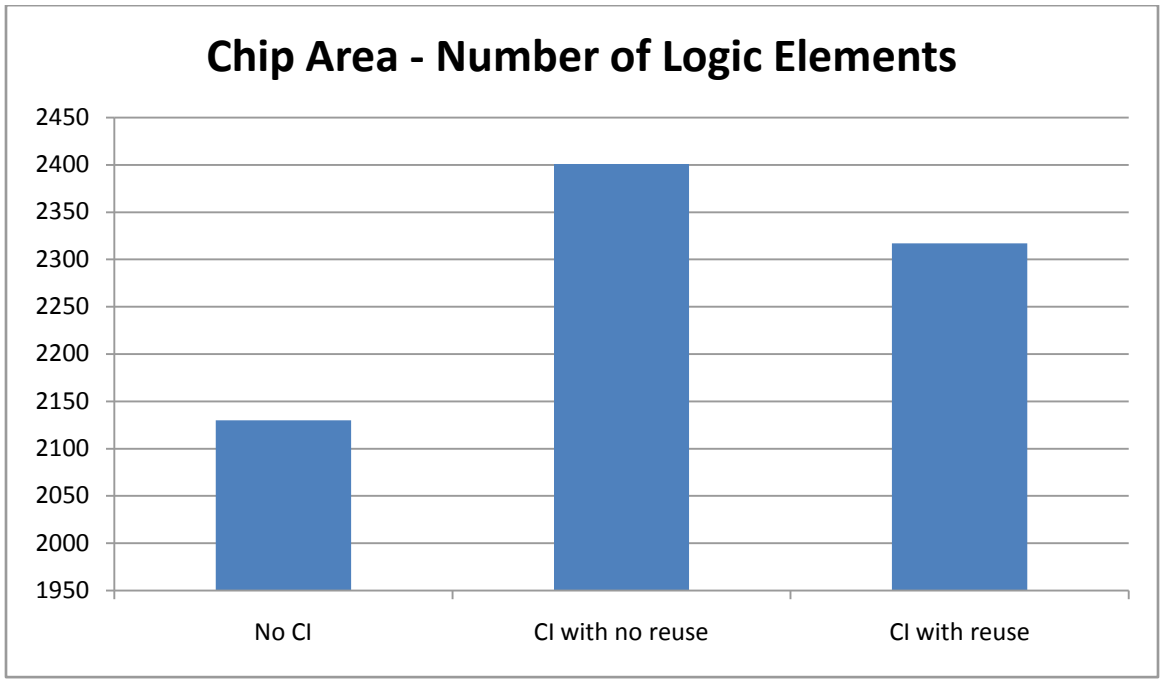

Figure 12 - Graph of CPU Chip Area for Test #1

5.2 – Testing single types of instructions #2

5.2.1 – Test Program

The operation to be completed is the following:

$$y = (a*b) + c$$

The operation has to be separated into steps of only two variables as required by the SHIRA toolchain. The C source code of the program is shown below.

//muladd.c

```
#include <stdio.h>

  int a = 0;
  int b = 0;
  int c = 0;
  int t1 = 0;
  int y = 0;



float main(int argc, char *argv[])
{
      t1 = a * b;
      y = c + t1;

      return y;
}


void prof(int num)
{
  printf("x%dx\n",num);
}
```

The output for this program from the SHIRA toolchain of the intermediate representation of data flow graph is shown below and its DFG is is shown in Figure 13

```
1306952932=10(148194546,1685984552)
1685984552=12(109970892,19111827)
```

The order for entering the inputs is:

```
ALT_CI_CI_HARDWARE_INST(0,(REGI32"c_divexI32_1"),(REGI32"c_divexI32_1")
ALT_CI_CI_HARDWARE_INST(1,(REGI32"a_divexI32_1"),(REGI32"b_divexI32_1")
```

The order of entering inputs shows that the user should enter the input c twice. This is how the format of the hardware targeted c program should look. The VHDL hardware code will automatically the output t1 instead of one of the inputs.

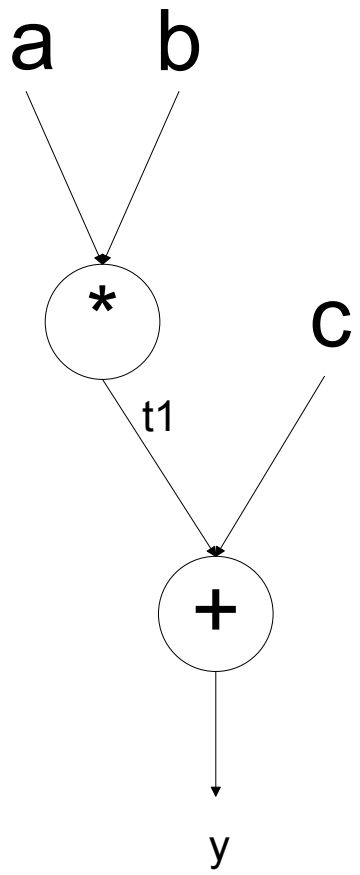

Figure 13 - DFG for Test #2

5.2.2 –Hardware System

The hardware system is identical to the one used for test #1. SOPC builder must however regenerate the hardware system because the number of input variables has changed.

5.2.3 – Software running on the Hardware

The order to input the operations is given by the SHIRA toolchain compiler during the generation of the VHDL code.

```c
#include <stdio.h>
#include "system.h"
#include <altera_avalon_performance_counter.h>

int main()
{
	int y = 0;

	int t1,t2,t3,t4,t5,t6;
	int a=6,b=55,c=10;

	PERF_RESET(PERFORMANCE_COUNTER_BASE);
	PERF_START_MEASURING(PERFORMANCE_COUNTER_BASE);
	PERF_BEGIN(PERFORMANCE_COUNTER_BASE,1);

			t1 = a * b;
			y = t1 + c;

	PERF_END(PERFORMANCE_COUNTER_BASE,1);
	printf("Result is %d \n", y);
	PERF_BEGIN(PERFORMANCE_COUNTER_BASE,2);

		ALT_CI_CI_HARDWARE_INST(0,c,c);
		y = ALT_CI_CI_HARDWARE_INST(1,a,b);



	PERF_END(PERFORMANCE_COUNTER_BASE,2);
	PERF_STOP_MEASURING(PERFORMANCE_COUNTER_BASE);

	printf("Result is %d \n", y);

	printf("#1 time - %d\n",
perf_get_section_time(PERFORMANCE_COUNTER_BASE,1));
	printf("#2 time - %d\n",
perf_get_section_time(PERFORMANCE_COUNTER_BASE,2));

	return 0;
}
```

5.2.4 – Tests and Results

The output without component reuse is:

```
Result #1 is 340
Result #2 is 340
#1 time - 212
#2 time – 73
```

The output with component reuse is:

```
Result #1 is 340
Result #2 is 340
#1 time - 212
#2 time – 74
```

The results of the test runs including logic elements used and number of cycles used is shown in the table below.

	No Custom Instruction	Custom Instruction with unlimited space	Custom Instruction with only 1 of each component
Total Logic Elements	2130	2270	2321
- Total Combinational Functions	2041	2150	2186
- Dedicated Logic Registers	1127	1195	1261
- Total Registers	1127	1195	1261
- Total Memory Bits	387072	387072	387072
Number of Clock Cycles	212	73	74

The time to complete the operation is increased due to the complexity of the operation. The longest time to complete the instruction occurs when no custom instruction is implemented. The performance increases by a significant amount. The increase in speed is 212/73 = 2.9 times when using a CI with no component reuse and 212/74 = 2.86 times faster when the custom instruction is added with component reuse. The speed results are shown in Figure 14.

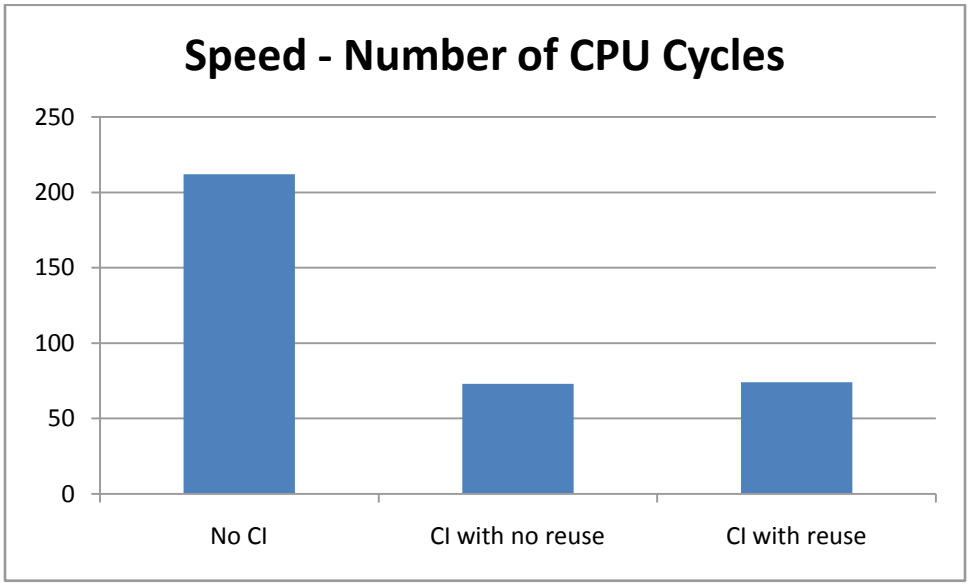

Figure 14 - Graph of CPU Cycles Used for Test #2

The hardware used when no custom instruction is used is the same as in test #1 since no extra hardware is created. When adding the custom instruction with no component reuse, the amount of logic elements it increased. This increase is far greater than that of test #1 since more components are required for the increased amount of components. The number of logic elements used is decreased significantly when component reuse is enabled. The chip area utilization is shown in Figure 15.

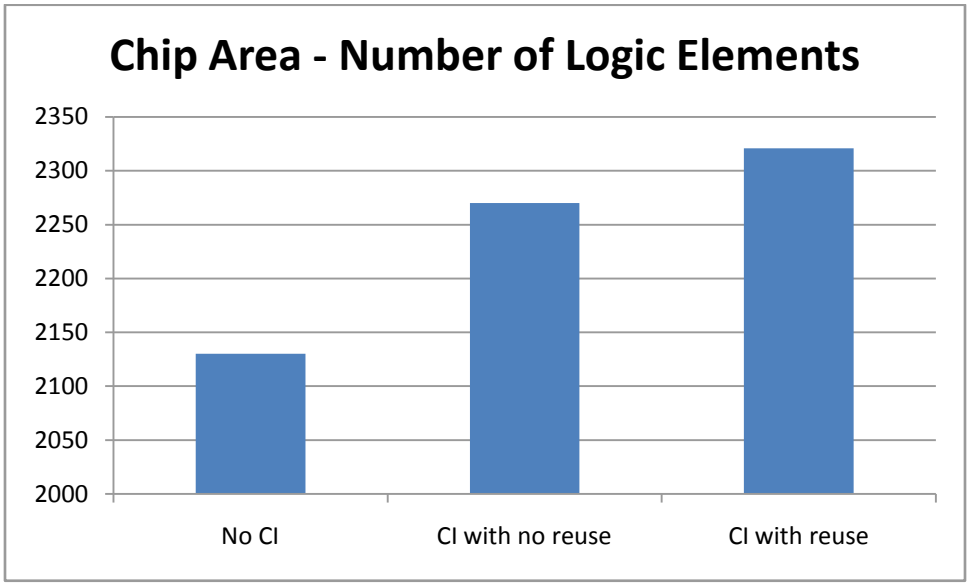

Figure 15 - Graph of Chip Area for Test #2

6 – Conclusions

6.1 - Summary

The research for this thesis contributed a module for the implementation of the identified custom instructions by the SHIRA toolchain into synthesizable hardware code that runs on Altera FPGAs. This extension of the toolchain generates synthesizable VHDL code from the intermediate representation as a data flow graph that is generated by the work done by Mr. Shapiro.

The hardware generated can either be unconstrained or resource constrained and use as soon as possible (ASAP) scheduling. The generated hardware code is then added to a simple NIOS II system using Altera Quartus software and the compiled to run on standard Altera FPGAs. The C code is then modified to use the custom instruction instead of the native NIOS II instruction set and compared. The test results compared the logic element utilization and number of clock cycles used in the unconstrained case, the resource constrained case using ASAP scheduling, and the native NIOS II instruction set.

The test results show a speed improvement in both cases when using a custom instruction, with the unconstrained case being slightly faster than the resource constrained solution. The hardware size increased when switching from the NIOS II instruction set to a NIOS II instruction set that included the custom instruction with the resource constrained solution using up the most logic elements on the FPGA.

The framework developed for generation of the VHDL code for the SHIRA toolchain can easily be expended to include other forms of resource constrained scheduling algorithms such As Late As Possible (ALAP) and other complex resource constrained scheduling algorithms using priority functions such as SPLICER. Time constrained algorithms can also be added to the developed framework to allow the tool to be used for real time applications. The most common time constrained scheduling algorithm is Force

Directed Scheduling (FDS). The framework also facilitates the future implementation of control flow optimizations on the datapath. Compatibility with FPGA produced by other manufactures is also foreseen by the SHIRA toolchain project and the framework layout already implements this ability.

The results of the test runs are as expected with significant speed improvements when the generated custom instruction is used. The chip size increase is minimal and the framework allows to select if the chip size should be constrained. The test results always provided identical results for the provided data flow graph operation.

6.2 – Future Work

The future work to be done on the automatic generation of the hardware code for custom instructions for the SHIRA toolcain is extensive. This can potentially provide many extra performance improvements for the user.

6.2.1 – Scheduling Improvements

Currently, only unconstrained scheduling and as soon as possible (ASAP) scheduling is being used and a maximum of one component for each operation is used. Other scheduling algorithms can be used to increase speed performance or further. Other algorithms can be used to balance speed performance with resource (logic element) utilization.

Resource constrained scheduling, such as ASAP, is used to maximize resource by limiting the number of functional units (LPM components) used. The ASAP implementation currently used can be expended to use more than one component of each type, either user selected or through an optimization algorithm. Other resource constrained scheduling types can also be implemented such as late as possible (ALAP)

scheduling, More complex resource constrained scheduling algorithms using priority functions can also further increase the performance of this tool. The SPLICER system by calculation the mobility of each operation is an example of this and is described in [14].

Time constrained scheduling algorithms can also be implemented to allow the SHIRA toolchain to be used for real time applications, such as digital signal processing (DSP). Real time applications have requirements that the constraints on the data sampling rate of the input data stream are met. This is accomplished without regard to the resource utilization and by limiting the number of control steps used for the custom instruction. The most popular time constrained scheduling algorithm is Force Directed Scheduling (FDS) which attempts to distribute uniformly the operations of the same type into all the available control steps. The FDS algorithm is described in [14].

Other scheduling algorithms combine resource and time constraints. These allow for limits the amount resources used as well as constraints on the number of cycles the custom instruction can use. These algorithms use optimization techniques generally requiring integer linear programming (ILP) and are much more complex than the cases where only one of resource or time is constrained. This also requires a lot more processing time for the compiler because many iterations have to be tried. Time and resource constrained scheduling is described in [14].

6.2.2 Control Flow Based Improvements

The custom instructions generated do not have any control information. The data flow graphs (DFG) that are created only provide data flow and no control flow information. With this additional data, a lot of further speed improvements can be achieved in cases where loops or real time data is used.

Control flow based scheduling can be implemented in this toolchain and more complicated datapaths can be used. Software optimization functions are needed to analyze and modify the C source code to

implement loop unrolling and loop splitting such that the generated hardware is used to its potential. This data processing should be done before the program beings to write the VHDL code. Further algorithms such as those described in [15] to optimize the datapats can also provide significant improvements.

6.2.3 Further Improvements

The VHDL generated is targeted for Altera FPGA boards and has not been tested on solutions provided by other FPGA manufacturers, such as Xilinx and Lattice. The framework is already implanted to allow the easy implantation of the changes needed to support the different FPGA solutions. Adapting the toolchain to create VHDL code for other manufacturers is simplified by the use LPM functions, the use of which is standardized across the most manufacturers. Modifying the code to use LPMs developed by the other manufactures is very small as some names and parameters will have to be modified to use those provided by the FPGA manufacturer. Other problems however might arise from other areas of the code and thorough testing on the different FPGA hardware will be needed. The performance of the VHDL code across the different FPGA platforms will also vary as the manufacturing and architecture of the FPGA is different amongst different models, with variances in number of cycles uses and number of logic elements used.

Further performance improvements can also be achieved by implementing more instructions, such as shifting operations. This can be further extended to detect when a simpler operation can be used instead of a more complicated one. For example, integer division by a factor of 2 can avoid the use of the complicated and time consuming divider and use a shift operation. This requires further inspection of the C source code as well as expansion of the hardware generation. Reduction of the instruction set of the generic CPU by removing unused instructions is also a possibility to reduce chip area and increase performance.

Flipflops are used at the moment for every input and output. The speed performance and resource utilization can be further improved by writing algorithms that remove flipflops that are not needed.

8 - References

1. **Shapiro, Daniel.** *Design and Implementation of Instruction Set Extension Identification for a Multiprocessor System-on-chip Hardware?Software Co-design Toolchain.* Ottawa : University of Ottawa, Faculty of Engineering, Jan 20/2009.

2. **Grosschadl, J, Tillich, S and Szekely, A.** *Performance Evaluation of Instruction Set Extensions for Long Modular Arithmetic on a SPARC V8 Processor.* s.l. : Digital System Design Architectures, Methods and Tools. DSD 2007. 10th Euromicro Conference on, Aug/2007. pp. 680-689.

3. **Lee, Tai-Chi, Roach, A and Robinson, P.** *DES Decoding Using FPGA and Custom Instructions.* s.l. : Information Technology: New Generations. ITNG 2006. Third Conference on, April/2006. pp. 575-577.

4. **Yankova, Y, et al.** *Automated HDL Generation: Comparative Evaluation.* s.l. : Circuits and Systems. ISCAS 2007. IEEE International Symposium on, May/2007. pp. 2750 – 2753.

5. **Hoffman, A, et al.** *A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA.* s.l. : Computer Aided Design. ICCAD 2001. IEEE/ACM International Conference on, Nov/2001. pp. 625 – 630.

6. **Arato, P and Kandar, T.** *VHDL code generation using pipeline operations produced by high level synthesis.* s.l. : Intelligent Signal Processing, 2003 IEEE International Symposium on, Sept/2003. pp. 191 - 196.

7. **Labrecque, Martin, Yiannacouras, Peter and Steffan, Gregory.** *Custom Code Generation for Soft Processors.* s.l. : ACM SIGARCH Computer Architecture News, Volume 35 Issue 3, June/2007.

8. **Bertels, K, et al.** *Hartes Toolchain Early Evaluation: Profiling, Compilation and HDL Generation.* s.l. : Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, Aug/2007. pp. 402-408.

9. **Yankova, Y, et al.** *DWARV: Delftworkbench Automated Reconfigurable VHDL Generator.* s.l. : Field Programmable Logic and Applications. FPL 2007. International Conference on, Aug/2007. pp. 697 - 701.

10. **Rinker, R, et al.** *An automated process for compiling dataflow graphs into reconfigurable hardware.* s.l. : Very Large Scale Integration (VLSI) Systems, IEEE Transactions on Volume 9, Issue 1,, Feb/2001. pp. 130 - 139.

11. **Shapiro, D, et al.** *Instruction Set Extension in the NIOS II: A Floating Point Divider for Complex Numbers.* s.l. : Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on, May/2010. pp. 1-5.

12. **Altera Corporation.** *Quartus II Handbook Version 10.1 Volume 1: Design and Synthesis.* Dec/2010.

13. **ALTERA Cooperation.** *Nios II Custom Instruction User Guide.* May/2008. http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf.

14. **Jerraya, Ahmed A, et al.** *Behavioral Synthesis and Component Resue with VHDL.* Norwell : Kluwer Academic Publishers, 1997. pp. 94-104.

15. **Arato, Peter, Visegrady, Tamas and Jankovits, Istvan.** *High Level Synthesis of Pipelined Datapaths.* Budapest : John Wiley and Sons Ltd.

16. **El-Rewini, H and Adb-El-Barr, M.** *Advanced Computer Architecture and Parallel Processing.* London : John Wiley, 2005.

17. **Pohl, Christopher, Paiz, Carlos and Porrmann, Mario.** *vMAGIC—Automatic Code Generation.* s.l. : International Journal of Reconfigurable Computing, 2009. http://www.hindawi.com/journals/ijrc/2009/205149.html.

18. **Leung, Man-Kit, Filiba, Terry Esther and Nagpal, Vinayak.** *VHDL Code Generation in the Ptolemy II Environment.* s.l. : Electrical Engineering and Computer Sciences, University of California at Berkeley, Oct/2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-140.html.

19. **Anand, Kapil and Gupta, Shivam.** *Designing of Customized Digital Signal Processor.* Dalhi : Indian Institute of Technology, May/2007. http://www.cse.iitd.ernet.in/esproject/homepage/release/or1200vector/pdfs/BTP_THESIS.pdf.

20. **Le Beux, Sébastien, et al.** *A Model Driven Engineering Design Flow to Generate VHDL.* s.l. : 2009 International Joint Conference on Neural Networks, 2009.

21. **Onoo, Akira, et al.** *On Automatic Generation of VHDL Code for Self-Organizing Map.* s.l. : Neural Networks, IEEE - INNS - ENNS International Joint Conference on, Neural Networks, 2009. pp. 2366 - 2373.

22. **Necsulescu, Philip and Voicu, Groza.** *Automatic Generation of VHDL Hhardware Code from Data Flow Graphs.* s.l. : Applied Computational Intelligence and Informatics (SACI), 2011 6th IEEE International Symposium on, May/2011. pp. 523-528.

Appendix A – Test #1 Generated VHDL Code

A.1 –add3x.vhd without Component Reuse

```
-- Generated VHDL code to implement Custom Instructions
-- CREATED: Sat Jan 10 22:01:48 EST 2011

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY CI_HARDWARE IS
PORT  (     SIGNAL clk        :     IN    STD_LOGIC;
                  SIGNAL reset      :     IN    STD_LOGIC;
                  SIGNAL clk_en     :     IN    STD_LOGIC;
                  SIGNAL start      :     IN    STD_LOGIC;
                  SIGNAL done       :     OUT STD_LOGIC;
                  SIGNAL dataa      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL datab      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL result     :     OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
                  SIGNAL n          :     IN    STD_LOGIC
            );
END CI_HARDWARE;

ARCHITECTURE behavior OF CI_HARDWARE IS


TYPE state_type is (s0,s1);
 SIGNAL STATE     :     state_type;

--inputs:
SIGNAL io_148194546     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL io_277277265     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL io_109970892     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL io_19111827      :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--InteriorIO:
SIGNAL io_1556901833    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL io_1685984552    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--output:
SIGNAL io_1306952932    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--null io:
--ff io:

COMPONENT lpm_add_sub
      GENERIC     (     lpm_direction           :     STRING;
                        lpm_hint                :     STRING;
                        lpm_representation      :     STRING;
                        lpm_type                :     STRING;
                        lpm_width               :     NATURAL
                  );
      PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        datab :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
                        result      :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

BEGIN
lpm_add_0   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
                              lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                              lpm_representation      =>    "SIGNED",
                              lpm_type                =>
      "LPM_ADD_SUB",
                              lpm_width               =>    32
                        )
      PORT MAP    (io_148194546, io_277277265, io_1556901833);

lpm_add_1   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
                              lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                              lpm_representation      =>    "SIGNED",
                              lpm_type                =>
      "LPM_ADD_SUB",
                              lpm_width               =>    32
                        )
      PORT MAP    (io_1685984552, io_1556901833, io_1306952932);

lpm_add_2   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
                              lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                              lpm_representation      =>    "SIGNED",
                              lpm_type                =>
      "LPM_ADD_SUB",
                              lpm_width               =>    32
                        )
      PORT MAP    (io_109970892, io_19111827, io_1685984552);


PROCESS (reset, clk)
BEGIN
if (reset = '1') then
state <= s0;
      ELSIF (clk'EVENT AND clk = '0') THEN
            CASE state IS
                  WHEN s0 =>
                  done <= '0';
                        IF (start = '1' AND  n = '0') THEN
                              io_148194546 <= dataa;
                              io_277277265 <= datab;
                              state <= s0;
                              done <= '1';
                        ELSIF (start = '1' AND  n = '1') THEN
                              io_109970892 <= dataa;
                              io_19111827 <= datab;
                              state <= s1;
                        ELSE
```

```
                              state <= s0;
                        END IF;
                  WHEN s1 =>
                        result <= io_1306952932;
                        done <= '1';
                        state <= s0;
                  WHEN OTHERS =>
                        report "Invalid State";
            END CASE;
      END IF;
END PROCESS;
END behavior;
```

A.2 - add3x.vhd with Component Reuse

```
-- Generated VHDL code to implement Custom Instructions
-- CREATED: Sat Jan 15 22:55:04 EST 2011

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY CI_HARDWARE IS
PORT  (     SIGNAL clk        :     IN    STD_LOGIC;
                  SIGNAL reset      :     IN    STD_LOGIC;
                  SIGNAL clk_en     :     IN    STD_LOGIC;
                  SIGNAL start      :     IN    STD_LOGIC;
                  SIGNAL done       :     OUT STD_LOGIC;
                  SIGNAL dataa      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL datab      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL result     :     OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
                  SIGNAL n          :     IN    STD_LOGIC
            );
END CI_HARDWARE;

ARCHITECTURE behavior OF CI_HARDWARE IS


TYPE state_type is (s0,s1,s2);
 SIGNAL STATE     :     state_type;

--inputs:
SIGNAL add_0_ina  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL add_0_inb  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--outputs:
SIGNAL add_0_out  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--ff io:
SIGNAL ff_add_0a_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0a_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0b_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0b_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
SIGNAL ff_add_0c_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0c_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);

COMPONENT lpm_add_sub
      GENERIC     (     lpm_direction           :     STRING;
                        lpm_hint                :     STRING;
                        lpm_representation      :     STRING;
                        lpm_type                :     STRING;
                        lpm_width               :     NATURAL
                  );
      PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        datab :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        result      :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

COMPONENT lpm_ff
      GENERIC (   lpm_fftype        : STRING;
                        lpm_type          : STRING;
                        lpm_width         : NATURAL
                  );
      PORT  (     clock :     IN    STD_LOGIC;
                        data  :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        q           :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

BEGIN
lpm_add_0   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
                                    lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                                    lpm_representation      =>    "SIGNED",
                                    lpm_type                =>
      "LPM_ADD_SUB",
                                    lpm_width               =>    32
                              )
      PORT MAP    (add_0_ina, add_0_inb, add_0_out);

lpm_ff_add_0a     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_add_0a_in, ff_add_0a_out);

lpm_ff_add_0b     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_add_0b_in, ff_add_0b_out);

lpm_ff_add_0c     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
```

```
                              )
      PORT MAP    (clk, ff_add_0c_in, ff_add_0c_out);



PROCESS (reset, clk)
BEGIN
if (reset = '1') then
state <= s0;
      ELSIF (clk'EVENT AND clk = '0') THEN
            CASE state IS
                  WHEN s0 =>
                  done <= '0';
                        IF (start = '1' AND  n = '0') THEN
                              add_0_ina <= dataa;
                              add_0_inb <= datab;
                              state <= s0;
                              done <= '1';
                        ELSIF (start = '1' AND  n = '1') THEN
                              add_0_ina <= dataa;
                              add_0_inb <= datab;
                              ff_add_0a_in <= add_0_out;
                              state <= s1;
                        ELSE
                              state <= s0;
                        END IF;
                  WHEN s1 =>
                        add_0_ina <= add_0_out;
                        add_0_inb <= ff_add_0a_out;
                        state <= s2;
                  WHEN s2 =>
                        result <= add_0_out;
                        done <= '1';
                        state <= s0;
                  WHEN OTHERS =>
                        report "Invalid State";
            END CASE;
      END IF;
END PROCESS;
END behavior;
```

Appendix B – Test #2 Generated VHDL Code

B.1 – addmul.vhd without Component Resuse

```
-- Generated VHDL code to implement Custom Instructions
-- CREATED: Fri Mar 04 01:26:37 EST 2011

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY CI_HARDWARE IS
PORT  (     SIGNAL clk        :     IN    STD_LOGIC;
                  SIGNAL reset      :     IN    STD_LOGIC;
                  SIGNAL clk_en     :     IN    STD_LOGIC;
                  SIGNAL start      :     IN    STD_LOGIC;
                  SIGNAL done       :     OUT STD_LOGIC;
                  SIGNAL dataa      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL datab      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL result     :     OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
                  SIGNAL n          :     IN    STD_LOGIC
            );
END CI_HARDWARE;

ARCHITECTURE behavior OF CI_HARDWARE IS


TYPE state_type is (s0,s1);
 SIGNAL STATE     :     state_type;

--inputs:
SIGNAL io_148194546     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL io_109970892     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL io_19111827      :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--InteriorIO:
SIGNAL io_1685984552    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--output:
SIGNAL io_1306952932    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--null io:
--ff io:

COMPONENT lpm_add_sub
      GENERIC     (     lpm_direction           :     STRING;
                        lpm_hint                :     STRING;
                        lpm_representation      :     STRING;
                        lpm_type                :     STRING;
                        lpm_width               :     NATURAL
                  );
      PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        datab :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        result      :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
```

```
END COMPONENT;

COMPONENT lpm_mult
      GENERIC (   lpm_hint                :     STRING;
                        lpm_representation      :     STRING;
                        lpm_type                :     STRING;
                        lpm_widtha              :     NATURAL;
                        lpm_widthb              :     NATURAL;
                        lpm_widthp              :     NATURAL
                  );
      PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        datab :     IN    STD_LOGIC_VECTOR  (31 DOWNTO 0);
                        result      :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

BEGIN
lpm_add_0   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
                                    lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                                    lpm_representation      =>    "SIGNED",
                                    lpm_type                =>
      "LPM_ADD_SUB",
                                    lpm_width               =>    32
                              )
      PORT MAP    (io_148194546, io_1685984552, io_1306952932);

lpm_mul_0   :     lpm_mult
      GENERIC MAP       (     lpm_hint                =>
      "MAXIMIZE_SPEED=5",
                                    lpm_representation      =>    "SIGNED",
                                    lpm_type                =>    "LPM_MULT",
                                    lpm_widtha              =>    32,
                                    lpm_widthb              =>    32,
                                    lpm_widthp              =>    64
                              )
      PORT MAP    (io_109970892, io_19111827, io_1685984552);


PROCESS (reset, clk)
BEGIN
if (reset = '1') then
state <= s0;
      ELSIF (clk'EVENT AND clk = '0') THEN
            CASE state IS
                  WHEN s0 =>
                  done <= '0';
                        IF (start = '1' AND  n = '0') THEN
                              io_148194546 <= dataa;
                              io_148194546 <= datab;
                              state <= s0;
                              done <= '1';
                        ELSIF (start = '1' AND  n = '1') THEN
                              io_109970892 <= dataa;
                              io_19111827 <= datab;
                              state <= s1;
```

```
                        ELSE
                              state <= s0;
                        END IF;
                  WHEN s1 =>
                        result <= io_1306952932;
                        done <= '1';
                        state <= s0;
                  WHEN OTHERS =>
                        report "Invalid State";
            END CASE;
      END IF;
END PROCESS;
END behavior;
```

B.2 – addmul.vhd with Component Reuse

```
-- Generated VHDL code to implement Custom Instructions
-- CREATED: Fri Mar 04 02:01:44 EST 2011

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY CI_HARDWARE IS
PORT  (     SIGNAL clk        :     IN    STD_LOGIC;
                  SIGNAL reset      :     IN    STD_LOGIC;
                  SIGNAL clk_en     :     IN    STD_LOGIC;
                  SIGNAL start      :     IN    STD_LOGIC;
                  SIGNAL done       :     OUT STD_LOGIC;
                  SIGNAL dataa      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL datab      :     IN    STD_LOGIC_VECTOR(31 DOWNTO
0);
                  SIGNAL result     :     OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
                  SIGNAL n          :     IN    STD_LOGIC
            );
END CI_HARDWARE;

ARCHITECTURE behavior OF CI_HARDWARE IS


TYPE state_type is (s0,s1);
 SIGNAL STATE     :     state_type;

--inputs:
SIGNAL add_0_ina  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL add_0_inb  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mul_0_ina  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mul_0_inb  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--InteriorIO:
--output:
SIGNAL add_0_out  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
SIGNAL mul_0_out  :     STD_LOGIC_VECTOR(31 DOWNTO 0);
--null io:
--ff io:
SIGNAL ff_add_0a_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0a_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0b_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0b_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0c_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_add_0c_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_mul_0a_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_mul_0a_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_mul_0b_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_mul_0b_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_mul_0c_in     :     STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ff_mul_0c_out    :     STD_LOGIC_VECTOR(31 DOWNTO 0);

COMPONENT lpm_add_sub
      GENERIC     (     lpm_direction           :     STRING;
                        lpm_hint                :     STRING;
                        lpm_representation      :     STRING;
                        lpm_type                :     STRING;
                        lpm_width               :     NATURAL
                  );
      PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        datab :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        result      :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

COMPONENT lpm_mult
      GENERIC (   lpm_hint                :     STRING;
                        lpm_representation      :     STRING;
                        lpm_type                :     STRING;
                        lpm_widtha              :     NATURAL;
                        lpm_widthb              :     NATURAL;
                        lpm_widthp              :     NATURAL
                  );
      PORT  (     dataa :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        datab :     IN    STD_LOGIC_VECTOR  (31 DOWNTO 0);
                        result      :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

COMPONENT lpm_ff
      GENERIC (   lpm_fftype        : STRING;
                        lpm_type          : STRING;
                        lpm_width         : NATURAL
                  );
      PORT  (     clock :     IN    STD_LOGIC;
                        data  :     IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
                        q           :     OUT   STD_LOGIC_VECTOR(31 DOWNTO 0)
                  );
END COMPONENT;

BEGIN
lpm_add_0   :     lpm_add_sub
      GENERIC MAP       (     lpm_direction           =>    "ADD",
```

```
                                    lpm_hint                =>
      "ONE_INPUT_IS_CONSTANT=NO, CIN_USED=NO",
                                    lpm_representation      =>    "SIGNED",
                                    lpm_type                =>
      "LPM_ADD_SUB",
                                    lpm_width               =>    32
                              )
      PORT MAP    (add_0_ina, add_0_inb, add_0_out);

lpm_mul_0   :     lpm_mult
      GENERIC MAP       (     lpm_hint                =>
      "MAXIMIZE_SPEED=5",
                                    lpm_representation      =>    "SIGNED",
                                    lpm_type                =>    "LPM_MULT",
                                    lpm_widtha              =>    32,
                                    lpm_widthb              =>    32,
                                    lpm_widthp              =>    64
                              )
      PORT MAP    (mul_0_ina, mul_0_inb, mul_0_out);

lpm_ff_add_0a     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_add_0a_in, ff_add_0a_out);

lpm_ff_add_0b     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_add_0b_in, ff_add_0b_out);

lpm_ff_add_0c     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_add_0c_in, ff_add_0c_out);

lpm_ff_mul_0a     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_mul_0a_in, ff_mul_0a_out);

lpm_ff_mul_0b     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_mul_0b_in, ff_mul_0b_out);

lpm_ff_mul_0c     :     lpm_ff
      GENERIC MAP       (     lpm_fftype => "DFF",
```

```
                                    lpm_type                =>    "LPM_FF",
                                    lpm_width               =>    32
                              )
      PORT MAP    (clk, ff_mul_0c_in, ff_mul_0c_out);


PROCESS (reset, clk)
BEGIN
if (reset = '1') then
state <= s0;
      ELSIF (clk'EVENT AND clk = '0') THEN
            CASE state IS
                  WHEN s0 =>
                  done <= '0';
                        IF (start = '1' AND  n = '0') THEN
                              add_0_ina <= dataa;
                              add_0_inb <= datab;
                              state <= s0;
                              done <= '1';
                        ELSIF (start = '1' AND  n = '1') THEN
                              mul_0_ina <= dataa;
                              mul_0_inb <= datab;
                              ff_mul_0a_in <= add_0_out;
                              state <= s1;
                        ELSE
                              state <= s0;
                        END IF;
                  WHEN s1 =>
                        result <= mul_0_out;
                        done <= '1';
                        state <= s0;
                  WHEN OTHERS =>
                        report "Invalid State";
            END CASE;
      END IF;
END PROCESS;
END behavior;
```