Reconfigurable Co-design of a Computationally Intensive Mathematical Problem

by

Beatriz Chiavegatto Iaderoza
B.Sc., University of Victoria, 2004

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Reconfigurable Co-design of a Computationally Intensive Mathematical Problem

by

Beatriz Chiavegatto Iaderoza
B.Sc., University of Victoria, 2002

Supervisory Committee:

Dr. Micaela Serra, Supervisor (Department of Computer Science)

Dr. Jon Muzio, Member (Department of Computer Science)

Dr. Sudhakar Ganti, Member (Department of Computer Science)

Supervisory Committee

Dr Micaela Serra
Supervisor (Department of Computer Science)

Dr Jon Muzio
Department Member (Department of Computer Science)

Dr Sudhakar Ganti
Department Member (Department of Computer Science)

Abstract

A reprogrammable hardware platform is used for the Co-design and implementation of a computationally intensive mathematical problem, namely the listing of irreducible polynomials over Galois fields of order 3 (GF(3)). The main goal is to accelerate the performance compared to an existing software implementation. This project uses hardware/software Co-design methodologies and techniques, and it is designed, implemented and evaluated on two distinct platforms, not simply by simulations. FPGAs are used as part of the reconfigurable hardware in both a PCI-based environment and in a more successful System-on-Chip (SOC) platform, which takes advantage of the closely-coupled interconnection between the hardware and software, thus minimizing the communication overhead. The case study, findings and general analysis lead to a possible ideal architecture for future approaches. Moreover, a more general detailed strategy can be seen for the transformation from software to a Co-design paradigm, maximizing parallelism.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Thanks to my friends and colleagues in Digital Systems Lab and Hannes Prokop, all of whom have helped make this research a very pleasant journey. Special thanks to Adam Backer for his constant support while I undertook some of the research at the University of New Brunswick. I will miss our enjoyable times at the lab.

Special thanks to both of my supervisors Dr Micaela Serra and Dr Ken Kent for their supervision and support.

Many warm thanks to my family for their endless love and encouragement.

Chapter 1: Introduction

Embedded systems are implemented by using a mix of software and hardware systems. Most commonly, software is used for features and flexibility, while hardware is used for performance. Design of embedded systems can be subject to many different types of constraints, including timing, size, weight, power consumption, reliability and cost. Current embedded systems are designed by specifying the software and hardware design completely separately causing incomparability of the design in the final stage of the implementation. Hardware/Software Co-design can be described as the cooperative design and development of the hardware and software partitions of a system. One of the main-purposes of Co-design is to facilitate the design of embedded systems. Such systems consist of a microprocessor, for which software is required, plus one or more custom ASICs or Field Programmable Gate Arrays (FPGAs). This methodology is extensively used in this work by providing a hardware/software Co-design for the generation of irreducible polynomials over the finite field of order 3, with two distinct hardware implementations.

This particular problem was chosen due to its utility in cryptography and also due to the research conducted by the Huskey and Lee at the University of Victoria in this area. This multiplication of polynomials is a computationally intensive problem bringing a slow and unfeasible execution of high degrees such as degree 20 up. Partitioning it by moving functionality from software to hardware is an attempt to accelerate the performance of an existing software solution. An FPGA is used for the hardware partition, providing possibilities for reconfigurability. If a hardware solution is developed for the multiplication of order 3, it would be a template for new implementation developed such as multiplication of order 5 and 7.

Initially, the software implementation is analyzed to try and identify computationally intensive modules which are possible candidates for hardware. For these modules a new Co-design infrastructure is derived, together with the overall control for

communication and synchronization. After the design and verification phases, the problem is implemented in two distinct platforms. In the first environment, the FPGA (the reconfigurable hardware element) is located on a board connected to the host computer via a PCI bus. The software component exchanges data with the hardware via the PCI bus, introducing substantial communication delays. The second implementation uses a System-on-Chip environment, where the processor is directly embedded within the FPGA itself. This allows the software to be executed from inside the FPGA, eliminating the use of the PCI bus. In both cases, the hardware modules in the FPGAs always execute much faster than the same modules in their software implementation. Communication between the hardware and software partitions continues to present the major challenge, as, in this problem, it occurs very intensely.

Chapter 2 presents an overview of background concepts, including hardware/software Co-design, reconfigurable computing, Field Programmable Gate Arrays, hardware accelerators and System-on-Chip. These are intended to help the reader understand the hardware implementation better.

Chapter 3 briefly describes the computational problem to be implemented and previous work as found in [15] and [10]. Also, an analysis of the existing software solution and implementation is undertaken. This analysis allows decisions to be made about possible partitioning strategies, that is, which software modules are the best candidates for hardware acceleration.

Chapter 4 gives a detailed description of the new Co-design solution. First, a description of the algorithm is presented in a newly derived form with explicit parallelism to exploit the hardware paradigm. A full verification of correctness is performed by comparing the solution against the initial software algorithm. Its design using VHDL, the choice of hardware description language, is also developed. This design remains the same for the hardware modules in both platforms used in the later

experiments, namely a PCI-based environment and the System-on-Chip. A full description of the implementations on each platform can also be found here.

Chapter 5 summarizes results of the experiments from the two implementations. An additional software comparison between a Pentium and a PowerPC is given to allow somewhat more accurate conclusions to be drawn from the performance data. The data is based entirely from direct execution on the hardware and software platforms, and not simply by simulation, as often found in other research projects.

Finally, chapter 6 presents the conclusions and conjectures which can be derived from the experiments together with comments on potential directions for future work.

Chapter 2: Background

A number of topics need to be combined in order to provide an interesting solution through hardware acceleration for the enhanced performance of an algorithm. This chapter provides a brief overview of the most relevant aspects. The topics are the following, together with the section number in which they are described.

- 2.1 Hardware/Software Co-design;
- 2.2 Reconfigurable Computing;
- 2.3 Field Programmable Gate Arrays (FPGA);
- 2.4 Hardware Accelerators;
- 2.5 System-on-Chip.

2.1 Hardware/Software Co-design

Hardware/Software (HW/SW) Co-design can be defined in many ways depending on which aspect is emphasized. The overall description includes all of the following items:

- the cooperative design of hardware and software components;
- the unification of currently separate hardware and software paths;
- the movement of functionality between hardware and software;
- the meeting of system-level objectives by exploiting synergy between hardware and software through their concurrent design.

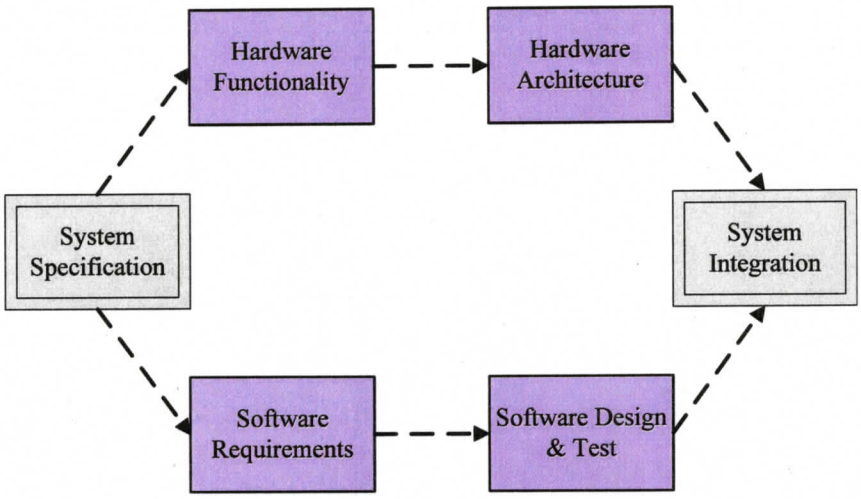


Figure 1: Conventional System Design

Typically, developers approach system design through two separate paths: software design and hardware design. As illustrated in Figure 1, the specifications are partitioned into hardware and software portions early in the design process. This can often lead to difficulty when integrating the entire system at the end of the process and finding incompatibilities across the boundaries. This can directly impact time-to-market of the product, delaying its deployment. Most of all, this design process restricts the ability to explore hardware and software trade-offs, such as the movement of functionality from hardware to software and vice-versa, and their respective implementation, from one domain to the other. This exploration is particularly important for legacy products, when new and improved hardware elements (e.g. a DSP with extra functionalities) become available.

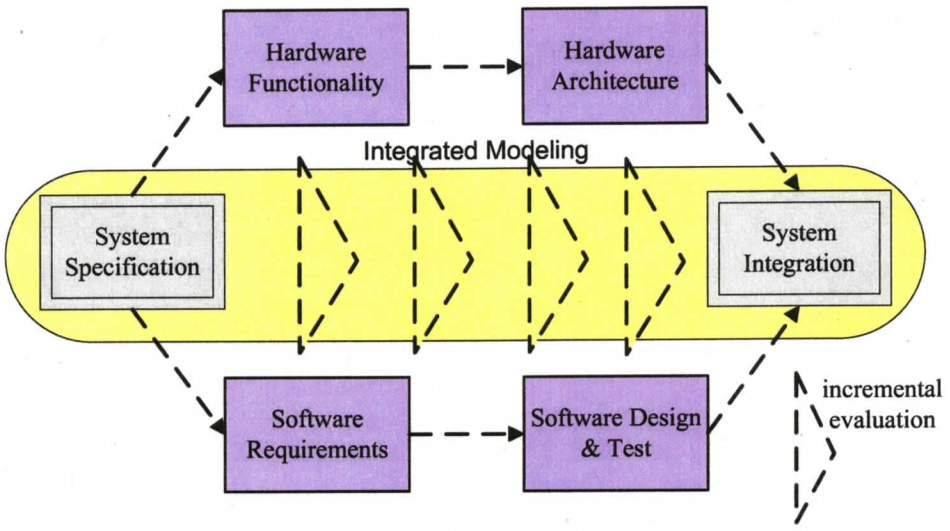


Figure 2: Unified Design Environment (HW/SW Co-design)

Hardware/Software Co-design attempts to integrate the hardware and software paths by envisioning a common platform, which should in turn increase the possibility of interaction between the hardware and software development. The interaction focuses on keeping the software and hardware spaces unified at all times as much as possible, as illustrated in Figure 2, to avoid a sudden complete integration in the last phase. The cooperative approach is modeled as a constant incremental evaluation of the two paths. Incremental evaluation does not simply involve meetings between developers, but employs more formal methods such as: modeling schemas, common development languages, co-simulations, common database and common specifications.

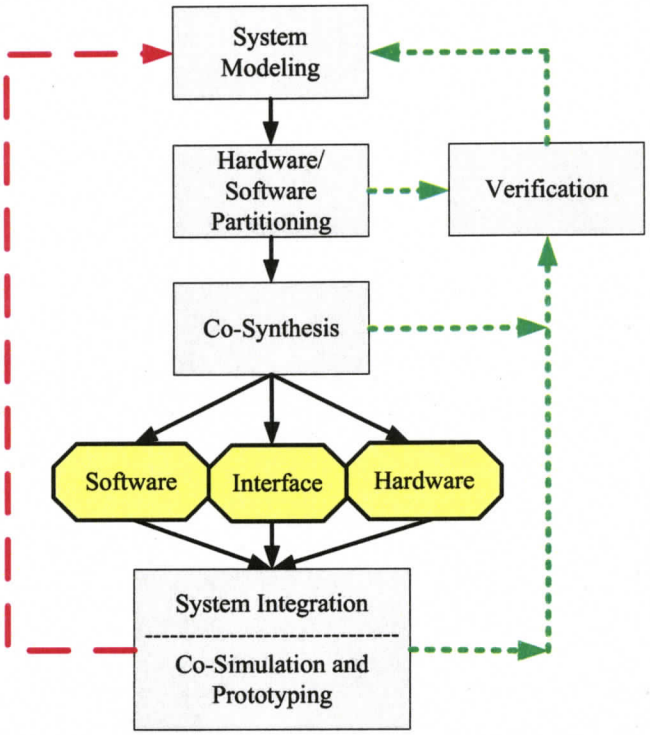


Figure 3: Co-design System Flow

The HW/SW Co-design of systems may be viewed as being composed of five main phases [16], as illustrated in Figure 3.

1. **Modeling** involves specifying the concepts and the constraints of the system to obtain a refined specification. This phase of the design also specifies the software and hardware models. There are three different paths the modeling process can take according to three possible starting points.
   i. An existing software implementation of the problem.
   ii. An existing piece of hardware, or its synthesizable description.
   iii. Neither of the above, allowing the developer to choose a general model as a starting point.

   It is important for existing tools to support all three paths, since different process efforts in design and development are required in each path. In the current market, there is still an absence of tools that can support all three design starting points. A model or a language that handles all possible paths to

development does not exist, forcing a designer to acquire multiple tools and making any easy integration more challenging.

2. **Partitioning** identifies the parts in the model which should be best suited in hardware or in software, given constraints such as time, size, cost and power. These constraints may change as the design develops (e.g. new hardware becomes available).
3. **Verification** ensures that the system constraints have been analyzed, that there is a certain amount of confidence that the system will work as designed, and that it follows its specification.
4. **Co-synthesis** uses the available tools to synthesize the software, hardware and interface implementation. This is done concurrently with as much interaction as possible between the three implementations.
5. **Co-simulation** executes all three components, functioning together in real time verifying that the original design and implied constraints by verifying if input and output data are presented as expected.

Although this system flow is suited for Co-design, it still shows some pitfalls in the process. If a new partition is necessary during the iterations, then a modification to the interface is required. This is very challenging and time consuming for developers. A typical example could imply a change in the details of a driver for a bus interface. This type of modification is usually not supported by existing tools; it is done manually and is left to the competence and experience of the designers.

HW/SW Co-design is applied mostly in the development of embedded systems. In this work, however, it is used as a platform for algorithm acceleration using reconfigurable hardware.

2.2 Reconfigurable Computers

Reconfigurable computers[1] are systems that use reprogrammable logic hardware, usually a Field Programmable Gate Array (FPGA), to implement specialized units for

[1] Configurable and reconfigurable are often used interchangeably in the literature.

a particular application [13]. This characteristic allows parallel computation of specific, configured set of operations, just like a custom Application-Specific Integrated Circuits (ASIC), with the advantage that such a system can also be reconfigured. In contrast, a custom hardware circuit can only provide the precise functionality needed for a specific task. The location of the reconfigurable component within the system is very important to achieve maximum performance and efficiency. This aspect is shown in this work by using the reconfigurable hardware in different environments.

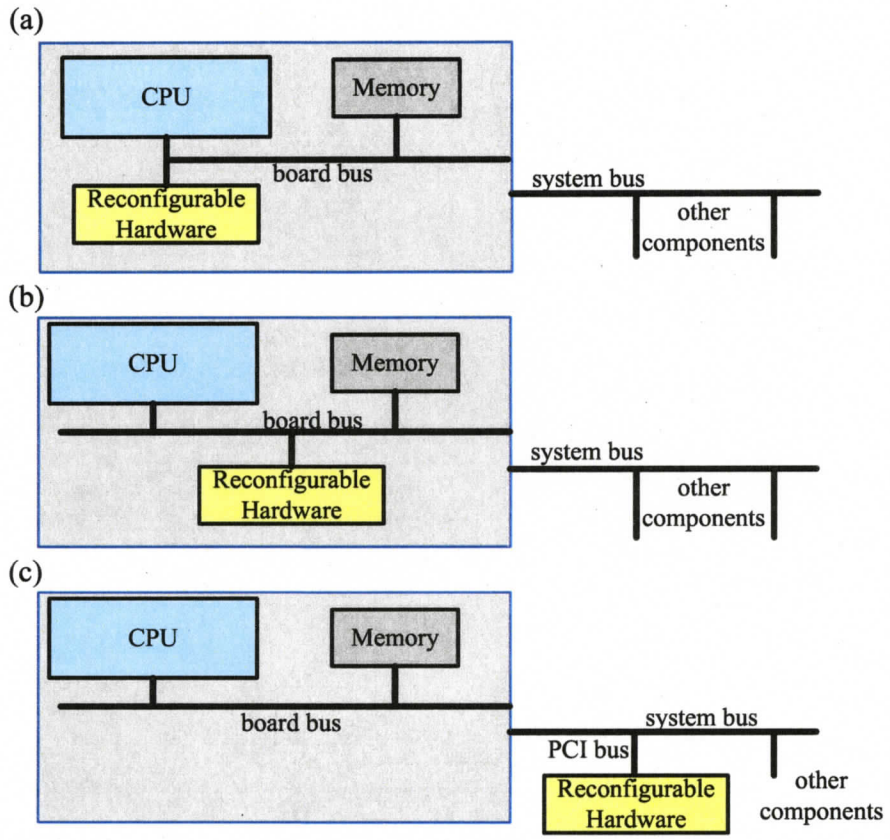


Figure 4: (a) Reconfigurable Processing Unit (b) Tightly-Coupled Reconfigurable System (c) Loosely-Coupled Reconfigurable System

A typical reconfigurable architecture consists of general purpose hardware, which is fixed, and an application specific hardware component, which is reconfigurable. These two components can have bindings at different levels such as instruction level, system bus level and peripheral local bus level, as shown graphically in Figure 4.

Reconfigurable computers can be classified into three categories: *Reconfigurable Processing Units (RPUs), tightly-coupled* systems and *loosely-coupled* systems [7]. A computer that couples the CPU with the programmable hardware at the instruction-level is called a *Reconfigurable Processing Unit.* A RPU has reconfigurable hardware in the same board as the CPU or even within the CPU chip, as in Figure 4 (a). If the system is coupled at a system bus level, it is defined as *tightly-coupled*, as in Figure 4 (b). Finally, a computer that couples the CPU with the programmable hardware at the peripheral bus level is referred to as *loosely-coupled*, as in Figure 4 (c).

Loosely-coupled systems are the most commonly available in a research environment. In these systems, the bandwidth connecting the reconfigurable hardware to the rest of the system is almost always too restrictive for production purposes. Tightly-coupled and RPU systems have only appeared recently in the market. These systems should help overcome the communication overhead between the CPU and the reconfigurable hardware, which is one of the main obstacles. In a lab research environment, this overhead can be evaluated separately and later removed from the overall run time, allowing for more varied types of experiments.

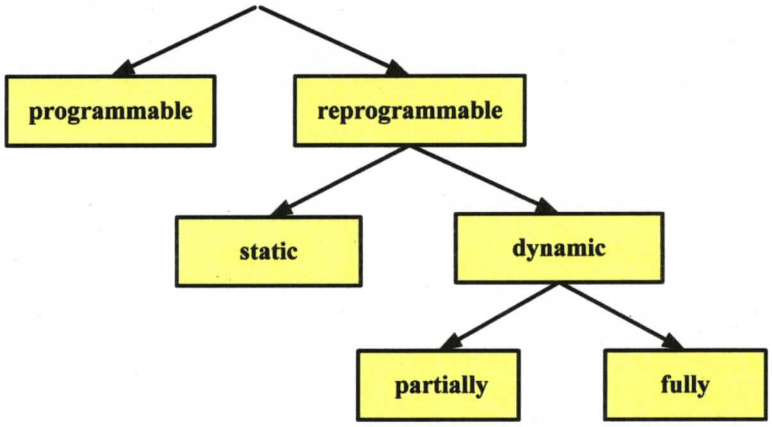


Figure 5: Taxonomy of reconfigurable components

From a general point of view, reconfigurable components can be programmed in several ways as shown in Figure 5. Currently, most reprogrammable devices are configured differently for each application at the start of a given computation. This is

termed *static configuration*. It also requires the component to be isolated from the running system during its configuration cycle. *Dynamic* configuration allows reconfiguration to happen during the execution of an application. *Partially* reconfigurable components allow a portion of the current configuration to remain fixed while another portion can change dynamically. *Fully* reconfigurable components require an entire new configuration.

Reconfiguration can provide some advantages over conventional computing in the following areas.

- **Performance**: Circuits can be customized for the task at hand as desired by the developer.
- **Time-To-Market**: Shorter development time allows faster time-to-market, especially through the intensive use of prototyping. Manufacturing of circuits and their development are not necessary, eliminating much overhead going to and from the manufacturer. Also, the logic design remains flexible due to its re-programmability, allowing changes until the product is shipped.
- **Efficiency**: There are opportunities for increased parallel computation and increased throughput.
- **System cost**: Reconfigurable computers can reduce the overall system cost through hardware reuse.

Reconfigurable computing has attracted a lot of attention among developers because it provides the flexibility of a general-purpose computer with the performance of an application-specific hardware. These systems can be customized at the hardware level to perform a task in hardware, overcoming the obstacle of conventional systems where the configuration of the hardware is fixed.

2.3 Field Programmable Gate Array (FPGA)

The most commonly used enabling device for configurable computing is the *Field Programmable Gate Array* (FPGA). Field Programmable Devices (FPD) have changed significantly in the last few years, increasing in complexity, efficiency, and

diminishing in cost. Field programmability means the logic function of the device can be modified easily by the user after the manufacturing process, enabling hardware reuse and a more flexible design. The FPGA itself is presented as a blank set of hardware units on a chip, where a user can program the function of each computing unit and the connections between them. By doing so, the FPGA chip *becomes* the equivalent of an ASIC chip, custom made for that given application. FPGAs can also be reprogrammed an unlimited number of times. This is a great advantage over ASICs, and while this may represent a cost advantage (depending on volume), the resulting product is not usually as efficient in performance.

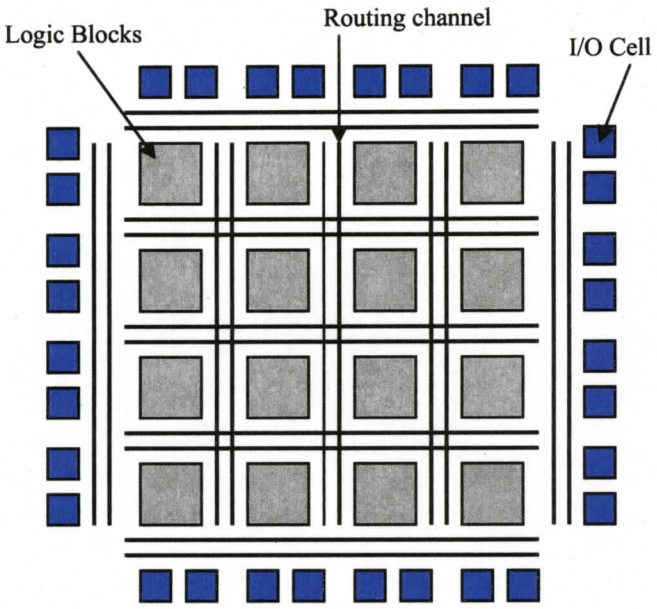


Figure 6: FPGA Structure

FPGAs are becoming more complex as technology evolves. The elements shown in Figure 6 are the basic ones that are present in a general FPGA. Most of the FPGAs fabricated today contain extra features. For example, the Xilinx FPGA used in this project has an embedded processor. These basic elements are [13]:

1. **Control Logic Blocks (CLB)** or **Logic Elements (LE)**. These components, shown as grey squares, are blocks of logic, mostly based on lookup tables (LUT) and storage components. Each block can be programmed to implement

all Boolean functions of n inputs and m outputs. Typically, a LE has $n = 4$ or 5 and $m = 1$ or 2.

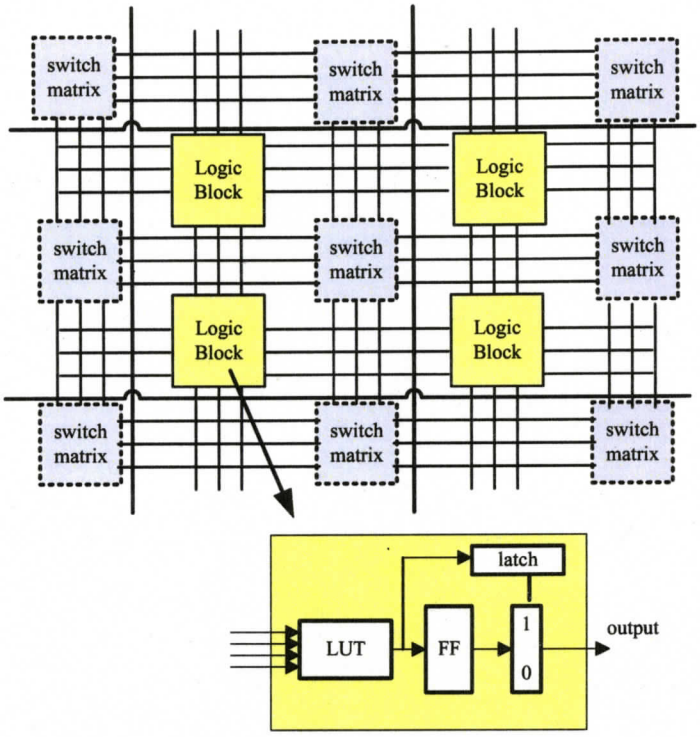


Figure 7: FPGA architecture and its routing resources

2. **Routing Resources** or **Routing Channels**. These are wires of various lengths to interconnect the logic blocks, as shown schematically in Figure 7. There are also several programmable switch matrices to connect the routing resources together in flexible paths.
3. **I/O Cells**. These, shown in dark blue squares in Figure 6, are logic cells at the periphery of the device for external connections.

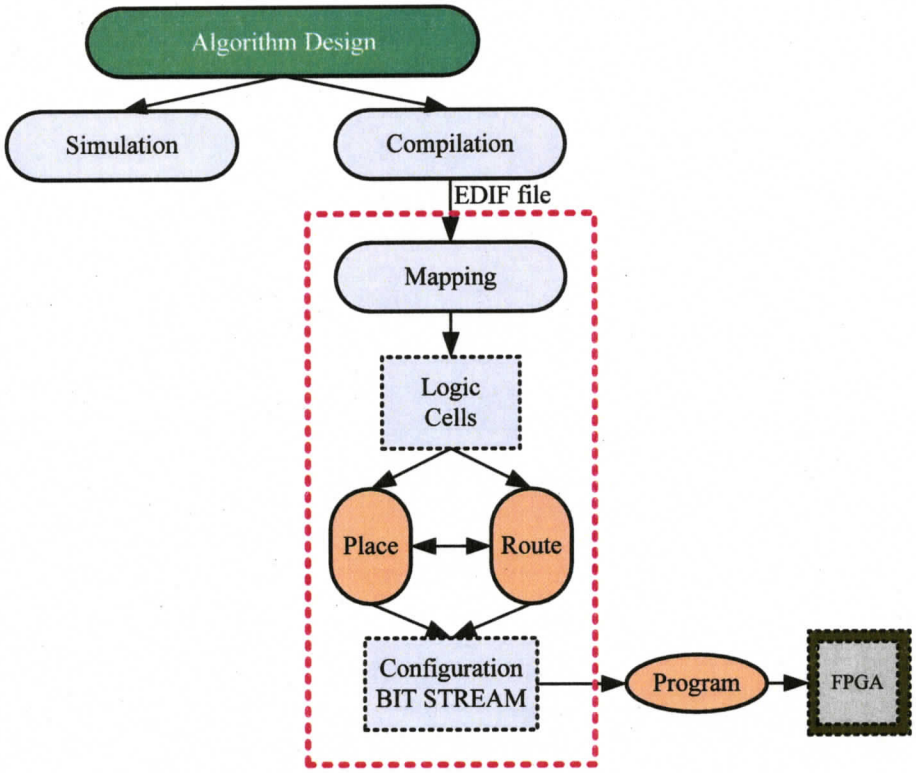


Figure 8: FPGA Programming Flow

A FPGA can be programmed by mapping the user design to the logic blocks and routing resources. This process is briefly illustrated in Figure 8. The top part of the design flow can be seen as common for all types of hardware design involving a resulting EDIF (Electronic Design Interchange Format) file. The process is analogous to the generation of object files by a software compiler. The bottom portion of the diagram, delimited by the dashed rectangle, shows the steps for an FPGA implementation usually with proprietary CAD tools (e.g. ISE by Xilinx and Quartus by Altera). It differs from general hardware in that there is a decomposition of the EDIF circuit into blocks, each of which can fit into the FPGA CLBs, followed by assignment of blocks and corresponding routing. The dashed red box is considered to be vendor-specific, since mapping, placement and routing processes are proprietary. In this work, this characteristic is exemplified by developing a design with both Altera's and Xilinx's tools. The resulting EDIF file is considered to be vendor independent since many tools can use the EDIF file as a starting point for the design.

An example is Quartus II by Altera, which takes an EDIF file designed in a Xilinx environment and feeds to the compiler to create an Altera-specific BIT file. The final result is then mapped into the actual chip for execution.

Although FPGAs consume more power than custom logic and are relatively expensive, they provide compensating advantages such as the following [21].

- **Testing Efficiency**. To test the system, the Co-designer does not have to wait until the end of the overall process, after obtaining a working chip.
- **Reuse of Hardware**. The same FPGA can be reused in multiple designs.
- **Prototyping**. FPGAs can be programmed quickly and also can be used as parts in a final design.

Realistically, programming FPGAs still requires that designers have a certain amount of knowledge of hardware languages. Vendors such as Xilinx provide tools to map such a language to an actual chip. New development platforms tools are now available to allow developers to program in a more software-oriented language. Handel-C contained in the DK platform by Celoxica is one such example of this programming language.

2.4 Hardware Accelerators

Hardware acceleration involves replacing a software algorithm with a hardware module in an attempt to accelerate a certain application. It is important to recognize that hardware accelerators are not co-processors. Co-processors are connected internally to a CPU and execute special instructions as dispatched by the CPU. Hardware accelerators, on the other hand, are connected to the CPU bus and communicate with the CPU via data and control registers located inside the accelerator, as shown in Figure 9. A typical example might be represented by a video card for games.

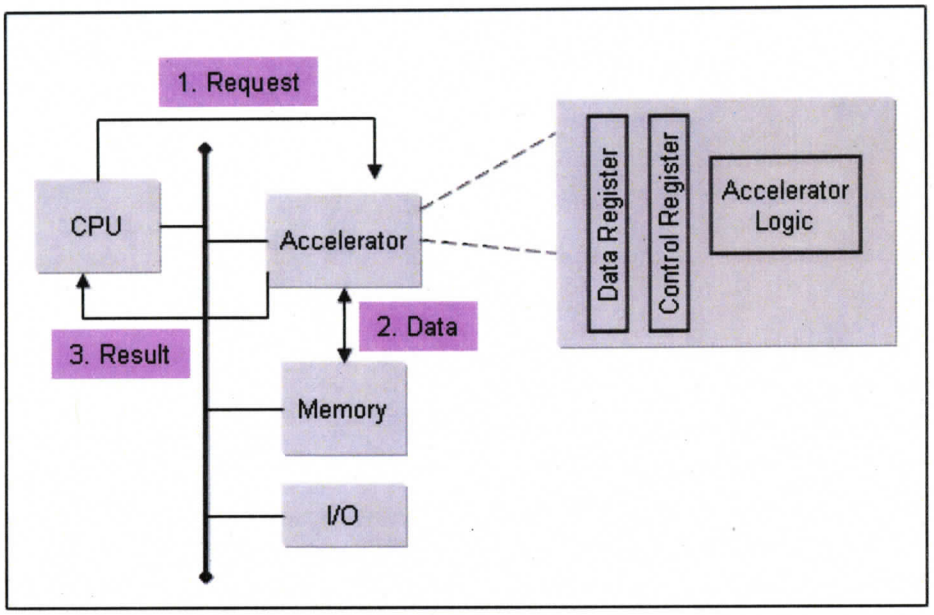


Figure 9: Accelerator architecture in a system

There are several reasons to add accelerators to a computing system [20], such as:

1. **Cost and performance**. Custom logic performs operations with greater speed than a CPU of equivalent cost. Because fast processing elements are very costly, splitting the application to perform in several elements is much more efficient.
2. **Parallelism**. Having another module facilitates the exploitation of executing blocks in parallel when possible and when desirable.

FPGAs provide an example of a programmable logic device which can be used as an accelerator.

2.5 System-on-Chip

System-on-Chip (SoC) refers to the technology that integrates various components of various functions of a computer into a single chip. Some of these components include.

- Memory blocks such as random access memory (RAM), read-only memory (ROM) and electronically erasable programmable read-only memory (EEPROM);

- Analog interfaces such as analog-to-digital converters (ADC) and digital-to-analog converters (DAC);
- External interfaces such as Universal Serial Bus (USB) and Ethernet;
- Other peripherals such as timers and power-on reset that ensure the power starts when the operating system is in a known state.

SoC designs offer the opportunity to migrate functionality initially implemented in software into hardware to achieve acceleration. SoCs can be designed with reconfigurable FPGAs which allow an attractive compromise between flexibility and efficiency [14]. Modern FPGAs equipped with embedded Reduced Instruction Set Computing (RISC) processor cores allow efficient designs of hardware with cooperation of software. This FPGA-based SoC architecture joint, with Intellectual Property Cores, allows rapid development with available predefined software drivers and libraries. The design of SoC consists of a hardware component, for example an FPGA, and a software module that controls the hardware component and/or executes extra functionality. The design flow of SoC aims to develop those two components in parallel.

The major concepts introduced above are important for the explanation of the following design solution work. In the analysis that follows, a Co-design solution using reconfigurable hardware as an accelerator is applied to a mathematical problem.

Chapter 3: Finding Polynomials over GF(3)

This section presents a brief introduction of the problem to be redesigned and implemented. The overall mathematical theory for the generation of polynomials is beyond the scope of this work.

3.1 Finite Fields of Characteristic 3

In algebra, a finite field is defined as a field containing only a finite number of elements. These fields are also called Galois Fields in honor of Evariste Galois, a 19th century French mathematician. The Galois Fields are usually referred to GF(p) where p denotes the number of elements in the field. Finite Fields of the type GF(p^n) are said to be of characteristics p, where p is always a prime. Elements of the field GF(p^n) can be represented as polynomials with degrees less or equal to n and coefficients less than p.

The finite field of characteristic 3, referred as GF(3), is used, among other applications, in cryptography for security aspects. Maintaining the integrity of the encryption key is essential to building a secure system. To do so, the computation requires a significant amount of resources due to the cryptographic key length and the large number of iterations necessary to transform these keys into valid ones. One approach to accelerate a cryptographic problem is to develop an ASIC that executes a specific process at hardware speed.

3.2 Existing Solutions

The narrow goal of the problem attacked here is to determine the number of irreducible polynomials in GF(3) given a degree as input. Figure 10 and Figure 11 show graphically the number of polynomials for degrees 2 to 18, where the exponential increase is evident[2]. A complete table with all the exact values is presented in section 5.1 in Table 3.

---

[2] The result of the degrees is shown in two separate graphs due to large data range of the number of irreducible polynomials.

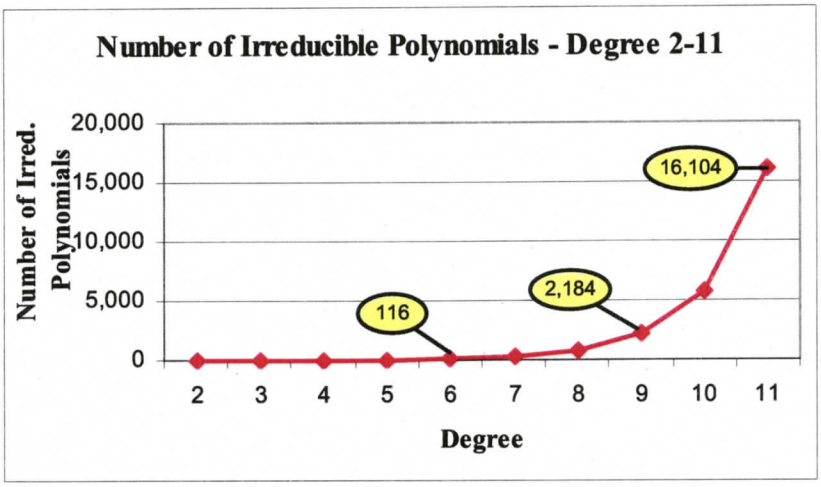


Figure 10: Number of generated irreducible polynomials over GF(3) for degrees 2 to 11

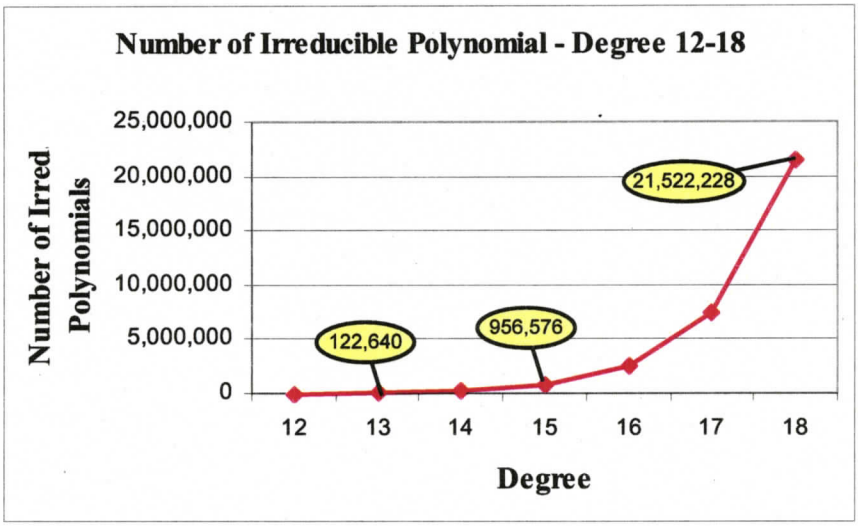


Figure 11: Number of generated irreducible polynomials over GF(3) for degrees 12 to 18

As the coefficients in polynomials over GF(3) can have three different values, namely 0, 1 or 2, they cannot be stored directly as bit values. In a software implementation, the most efficient solution seems to be using an array to hold the coefficients of the polynomials. In [10], Harrison uses two bit-words of length *n* called *bottom* and *top,* where *n* is the number of coefficients. Each coefficient *ci* allocates a bit with index *i* in each word. The values of the bits *i* in the two words can represent *ci* as in:

Coefficient	0	1	2
High word	0	0	1
Low word	0	1	0

This is called a *bit-sliced representation* and it has its benefits when storing coefficients. In software, word-oriented instructions can be used to process n coefficients in parallel. In hardware, efficient circuits can be built to implement multiplication and addition.

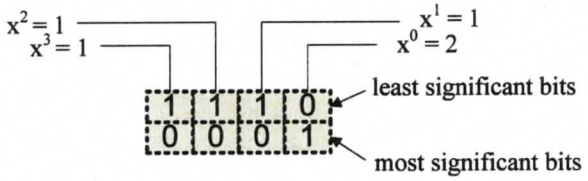


Figure 12: Bit-Slice Representation

The most interesting previous solution to this problem can be found in a reconfigurable hardware implementation presented in [15]. Both hardware and software implementations were proposed by Page and Smart. In their solution, with the objective of showing that arithmetic in GF(3) is suitable in terms of performance and size for use in cryptographic problems, a hardware implementation of the problem is described. This implementation was based on the Handel-C language using version 2.1 of Celoxica Handel-C hardware compilation system with a Xillinx4000XL FPGA. The bit-slicing approach was used to represent the polynomials as illustrated in Figure 12. Polynomial coefficients are represented as two values, w_1 and w_2 as shown in the example of the trinomial x^3+x^2+x+2. The value held by w_1 represents the least significant bits of all the coefficients in the polynomial while the value held by w_2 represents the most significant bits.

In a yet unpublished paper, Lee and Ruskey present a new solution with a new representation, where the change is in the bits for the coefficients with value 2:

Coefficient	0	1	2
High word	0	0	1
Low word	0	1	1

The combination with both bits set to 1 is used to represent coefficient of value 2, while the representation of high-bit set and low-bit clear is considered invalid. Using this representation Lee and Ruskey implemented a software version of the same problem in C. This work aimed at gaining theoretical insight into the properties of such polynomials over GF(3). The computational effort for finding these polynomials is high. This would be a good starting reason to an attempt of acceleration of the problem. Moreover, there are other applications that make hardware accelerated arithmetic over GF(3) an interesting field of study.

Extensive software timing was obtained as a reference point from the original code developed by G. Lee. The C code was run in a 1.8 GHz Intel Xeon processor and 1 Giga-Byte of RAM machine using an existing freeware tool to measure the execution time. This tool, called *ptime*[3], is a program timer used for benchmarking purposes. This command line software measures the execution time of programs passed as an argument to an accuracy of 5ms or better. The software was used to find all irreducible polynomials over GF(3) of degrees 2 to 18. The runtime to generate the required number of irreducible polynomial for the C implementation can be observed in Figure 13 and Figure 14. These graphs provide a general overview of the runtime exponential growth. A complete table with all the results is presented in section 5.1 in Table 3.

[3] http://www.pc-tools.net/win32/ptime/, last accessed: 5 January 2007

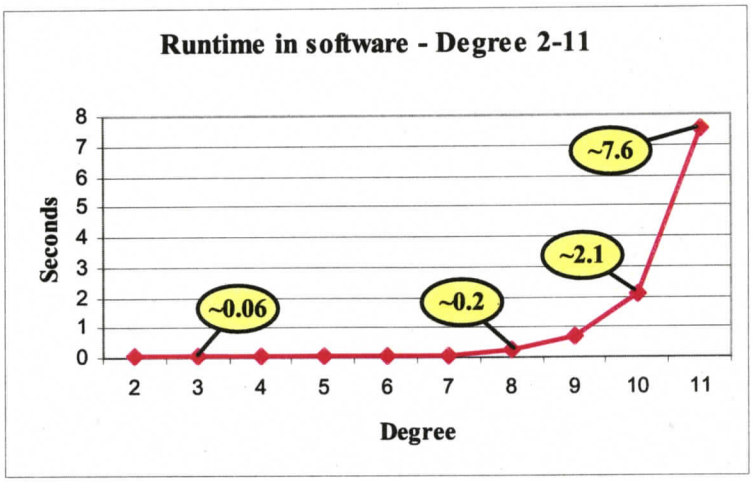


Figure 13: Runtime of software for degree 2 to 11

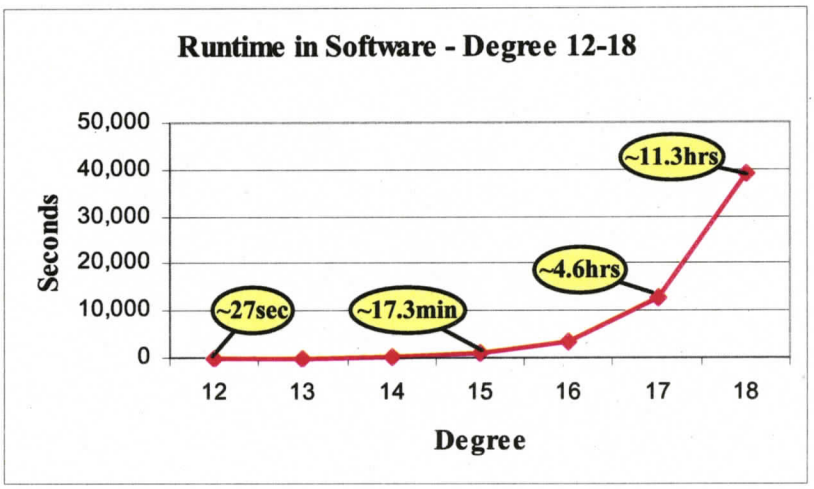


Figure 14: Software runtime for degree 12 to 18

Since the goal for this project is to accelerate the existing software implementation by applying HW/SW Co-design techniques, the main idea involved is to move sections of the software code to hardware and create a communication path between the software and hardware. As an initial approach, an analysis of the C code is done to identify sections that most affect the performance of the generation of irreducible polynomials. To do so, the program was run using *Gprof* UNIX profiler on a Sunblade Solaris 8 machine with optimization 3. *Gprof* profiler is a time profiler that

measures the execution paths of the application on a method level. This profiler provides an estimate execution time for each method. An example of its output by running the code with degree 11 is shown in Figure 15.

granularity: each sample hit covers 4 byte(s) for 0.07% of 14.83 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.2	4.93	4.93				internal_mcount [7]
32.8	9.79	4.86	2736896	0.00	0.00	multmod [6]
14.4	11.93	2.14	45996934	0.00	0.00	add [8]
6.3	12.86	0.93				__ashldi3 [12]
6.0	13.75	0.89	2032605	0.00	0.00	cmp [13]
1.9	14.03	0.28	273793	0.00	0.00	subtract [14]
1.8	14.29	0.26	16104	0.02	0.45	minpoly [5]
1.6	14.53	0.24				_mcount (38)
1.1	14.70	0.17	159892	0.00	0.01	mod [10]
0.3	14.74	0.04	16104	0.00	0.00	PrintCoeff [16]
0.3	14.78	0.04	16104	0.00	0.09	gcd [9]
0.3	14.82	0.04	16104	0.00	0.08	powmod [11]
0.1	14.83	0.01	16104	0.00	0.54	PrintIt [1]
0.0	14.83	0.00	16104	0.00	0.00	OutputPoly [15]
0.0	14.83	0.00	1	0.00	8730.00	Gen [2]
0.0	14.83	0.00	1	0.00	0.00	ProcessInput [17]
0.0	14.83	0.00	1	0.00	8730.00	main [3]

Figure 15: UNIX Profiler output for degree 11

The execution time for the application increases by an approximate value of 30% when using the profiler. This happens because the profiler adds extra code to the original implementation to obtain the precise timing, causing some delay. In Figure 15, for degree 11, the method *multmod* takes 32.8% of the total execution time and the method *internal_mcount* takes 33.2%. This latest is an overhead given by the profiler. To obtain a more realistic percentage of the *multmod* and *add* function, this value (33.2%) is re-distributed between the original functions of the C code.

Although the *Gprof* profiler provides precise information of how many times a certain function is called, it is in fact not a good measurement of the information sent between the functions. For example, in this project the a portion of the multiplication of polynomials is implemented in hardware in a attempt of acceleration. It is know that this portion of the code communications often with the software portion causing a lot of information to be transferred a time. Therefore it is also important to address

the communication aspect of an application but the profiler used in this thesis does not address this issue.

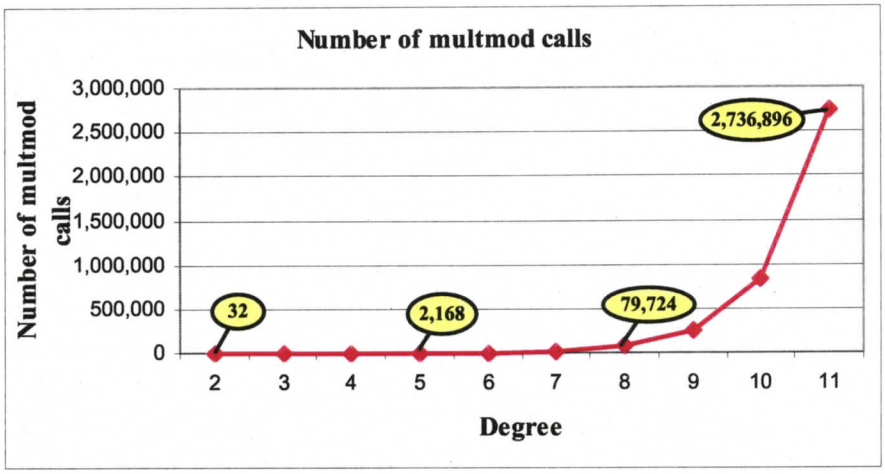


Figure 16: *Multmod* calls in GF3 for degree 2 - 11

Another aspect to consider is that approximately 96.4% of the *add* calls are done by *multmod*. Out of the 45 million times that *add* is executed in this example, approximately 44 million are calls from *multmod*[4]. Also, one must consider that *multmod* is called 2,736,896 times in this case. Figure 16 illustrates the number of *multmod* calls for each degree varying from 2 to 11. Note that the number of *multmod* operations increases rapidly for each degree. A complete table with the results for number of multmod calls generated is presented in section 5.1 in Table 3.

This analysis leaves one to conclude that the total percentage taken by the *multmod* function is 62.1% of the total execution runtime of the program indicating this function is a good candidate for hardware acceleration. This was calculated the following way:

62.1 % (total *multmod* %) =

1.332 (re-distribution of *internal_mcount %*) *

[32.8% (*multmod*) + **13.83%** (96 % of 14.4 for *add*)]

[4] This value was obtained by running the *Gprof* profiler.

Two different Co-design implementations for this problem are explored, implemented and timed in this work. In an attempt to accelerate the solution, the multiplication of polynomials is implemented in hardware interacting with the existing software in two different environments. First there is a Co-design on a PCI-based platform, followed by a Co-design in a System-on-Chip platform. For fast prototyping, FPGAs are used as a reconfigurable hardware and VHDL, a hardware description language, as the programming language.

Chapter 4: Co-design Solutions

Timing analysis of the software implementation identified the *multmod* function, which is the multiplication between polynomials, as the most CPU intensive portion of the code. An approach to improve the execution time is to implement this function using programmable hardware as an accelerator. The motivation for moving functionality to a reconfigurable hardware involves the creation of a custom operations for the *multmod* function. For a better understanding of this concept, it is useful to consider an example. The following instruction is present in *multmod*:

```
result = result + 2 * t
```

In a general microprocessor, depending on the instruction set available, this line of code would be translated into two instructions, as in:

```
SHL t, t, 1                ; shift left by 1 bit to multiply by 2
ADD result, result, t      ; add the result to the running sum
```

There are more recent exceptions, as in the ARM architecture where it would be translated into a single instruction, as in:

```
ADD result, result, t LSL #1
```

Moreover at execution time, the whole process of Fetch/Decode/Execute and transfers between the CPU and memory or cache has to be executed. The idea in acceleration through Co-design with reconfigurable hardware is that one would design a "custom mini processor" in the reconfigurable hardware to implement optimally only the mini CPU required for the particular application. Building an actual custom chip would be infeasible, in terms of cost and time, while this reconfigurable hardware-based solution offers a possible alternative.

In the *multmod* function examined here, the complete computational unit is analyzed in detail in order to extract the "native" (operating system specific) instructions which need to be implemented and optimized. The design steps to arrive at such a solution are now described, including the migration from the control flow inherent in the software, shown as a flow chart, towards the hardware-oriented design using a Finite State Machine (FSM) diagram.

The concept of hardware implementation in HW/SW Co-design is best shown using block diagrams or such equivalent structures. The main transitions from software to hardware are to be found in the sections where parallelism can be exploited [18]. The first step, as shown in Figure 17, is the precise analysis of the software implementation using a flowchart.

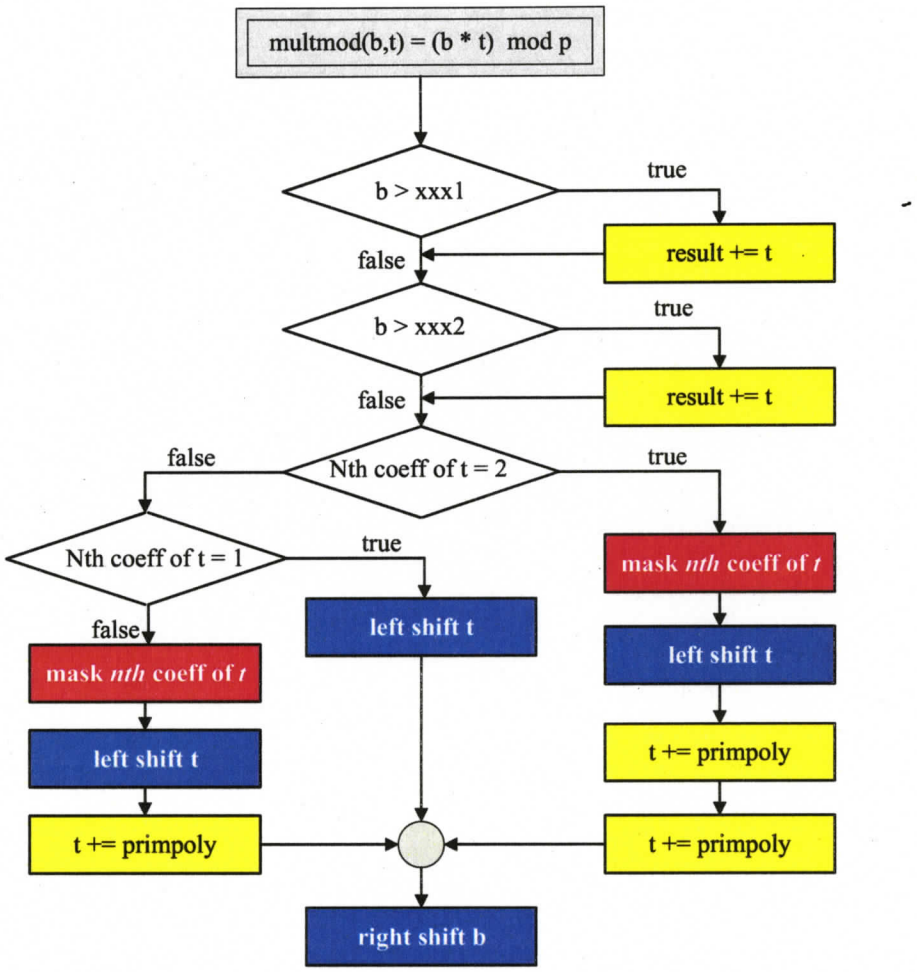


Figure 17: Software Algorithm for multiplication of two polynomials over GF(3)

This flowchart illustrates one single call of the *multmod* function in software, but this is repeated many times during the execution. For example, at degree 10, it is called 847,268 times.

For consistency with the software implementation, the input polynomials to be multiplied are called *b* and *t*. The degree entered by the user is denoted as *N* and the primitive polynomial is denoted as *primpol*. By careful analysis of Figure 17, one can see the three main operations in *multmod*.

1. Addition of two polynomials (illustrated in yellow). These are "*result* += *t*" and "*t* += *primpoly*".
2. Left or right shift (illustrated in blue). These are "*left shift t*" and "*right shift b*".
3. Masking of individual bits (illustrated in red), *"mask nth coeff of t"*.

Based on the software flowchart, two distinct implementations of *multmod* into two different hardware platforms are presented in this work[5]. The main reasons two distinct platforms were used are that:

- The two platforms, with different reconfigurable hardware and microprocessors, were available.
- It is always extremely useful to experiment on multiple platforms for more accurate conclusions and comparisons.
- After the first implementation was completed, a bottleneck of the communication between hardware and software was identified, leading towards a second environment where this issue could be overcome, to a certain extent.

The FPGA was chosen as the reconfigurable hardware in both boards. Moreover, the two platforms represent very closely the models of loosely-coupled and integrated System-on-Chip (SoC), as described earlier in sections 2.2 and 2.5.

Section 4.1 describes the design decisions from the migration of software to hardware. A Finite State Machine is presented as the core of the computation.

---

[5] The development on the first platform was done in collaboration with Hannes Prokop from the Technical University of Vienna, who worked temporarily as a member of the Digital Systems Design (DSD) lab research group at the University of Victoria.

Section 4.2 describes the implementation of the problem in a loosely-coupled environment, denoted as *Part 1*. Section 4.3, denoted as *Part 2*, presents the design on a SoC environment.

4.1 Decisions in Design and Implementation

The first high level decision is achieved by analyzing the software flowchart and separating the operations in the design flow. For example, operations such as "*result += t*" is placed in one module, which in turn contains only the custom operations for it. In this work, this module is called the Arithmetic Computation Unit (ACpU). The ACpU can be seen as similar to an Arithmetic Logic Unit (ALU) in a general processor. However, it is restricted in its operations to only those elements required by the current application.

In a general processor, there is also a control unit, an FSM, which directs the flow of operations. The same unit is required here. This consists of another module called Arithmetic Control Unit (ACtU) in this work. Both units are now described in detail.

4.1.1 Arithmetic Computation Unit

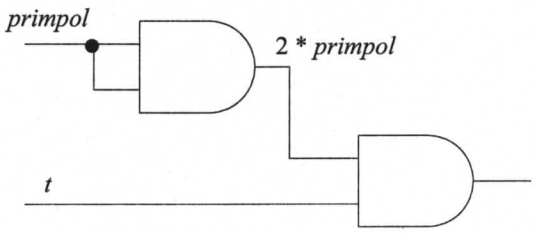
The Arithmetic Computation Unit (ACpU) implements the arithmetic and logic operations for the multiplication of polynomials. The ACpU design follows quite closely the original software implementation.

Due to the bit-sliced representation of the polynomials, each operation is performed on two 64-bit words in parallel in the same cycle. Thus it can be said that the design uses the single instruction, multiple data (SIMD) paradigm. In the software implementation the operations used are operations provided by the general microprocessor CPU. As seen in Figure 17, the operations used from the general CPU might be *add*, *move*, *shift*, *xor*, *compare* and *branch*, leaving all other operations available and redundant. Ideally, it would be very efficient if a custom and optimized hardware were available, yet such a custom chip is not realistic. Instead the implementation of this customized module with the ACpU using configurable

hardware is completely feasible. The following custom operations have been designed:

1. **Addition**: This consists of the addition of two 64-bit words representing the two polynomials. The types of additions that can be computed by ACpU are:
   a. ADD(*result, t*) – adds *result* polynomial to input variable *t*.
   b. ADD(*result, 2 * t*) – adds *result* polynomial to the result polynomial of 2 * *t*.
   c. ADD(*t, primpol*) – adds the input *t* polynomial to primitive polynomial *primpol*. This primitive polynomial is a fixed value for each degree.
   d. ADD(*t, 2 * pimpol*) – adds the input *t* polynomial to the result polynomial of 2 * *primpol*.

   While in a standard binary arithmetic "2 * *t*" could be done by simply shifting t by one, in a ternary arithmetic this is not possible. An additional ADD operation is performed that does the following ADD(*t, t*). The same idea applies to "2 * *primpol*" as shown in the circuitry below:



ADD(*t*, 2 * *primpol*)

2. **Shift**: The shifting of bottom and top words is done in parallel, in a single clock cycle. This differs from the software implementation where the top word is shifted in one clock cycle and the bottom in another due to the sequential execution. There are two main shift operations required by this algorithm as can be seen in Figure 17: shift right *b* and shift left *t* denoted as *shift_r_b* and *shift_l_t* in the hardware implementation. The first operation assigns the bits 63 through 1 to the input signals 62 through 0 of the *b* flip-flop. The two input lines for top and bottom at bit 63 are assigned to 0. The *shift_l_t* operates in a similar way but assigning the input lines 0 to zero value.

3. **Mask**: Masking of an individual bit is denoted as *mask_t_coeff*, where the position of the bit being masked is data dependent. Masking is done in the top bit variable of the polynomial, called *top_bit*. The following equation shows the calculation of the top_bit variable in software:

```
long long top_bit = (long long)1 << (N-1);
```

It can be noted that the only variable on which this calculation is dependent is the input degree *N*. The input degree is sent by the ACtU to this module. When masking a variable, the SIMD approach is used by the top and bottom word in at the same time.

Before any of these operations can take place, the program needs to send the input values to the hardware. This is better described in the later sections since the transfer of input and output varies from one platform to the other.

4.1.2 Arithmetic Control Unit

The Arithmetic Control Unit (ACtU) consists of an FSM that coordinates the operation flow for the multiplication of the polynomials. This FSM must be derived from the flowchart illustrated in Figure 17. The ACtU can be described as a high level design of the multiplication between polynomials separating the multiplication control flow from its operations. Essentially, the *operations* (the coloured boxes in the flowchart) are implemented in the ACpU as previously described, and the *flow of control* (the arrows) is implemented in the ACtU. The ACtU is responsible for:

- Loading input polynomials.
- Storing output polynomials.
- Controlling the multiplication operations.
- Implementing the reset logic.
- Providing accurate time measurements.

These requirements were analyzed and manually redesigned into an FSM which later led to the VHDL implementation. The outcome, achieved following many iterations for refinement and verification, is shown in

Figure 18. The general logic design is eventually divided into five main processes (in VHDL) and it is useful to describe them as such.

Finite state machine (*FSM process*). **As shown in**

1. Figure 18, the FSM implements the core steps for the polynomial multiplication and exploits the parallelism of the computation, as in:
    a. Defining states.
    b. Defining transitions between states.
    c. Optimizing/minimizing the FSM.

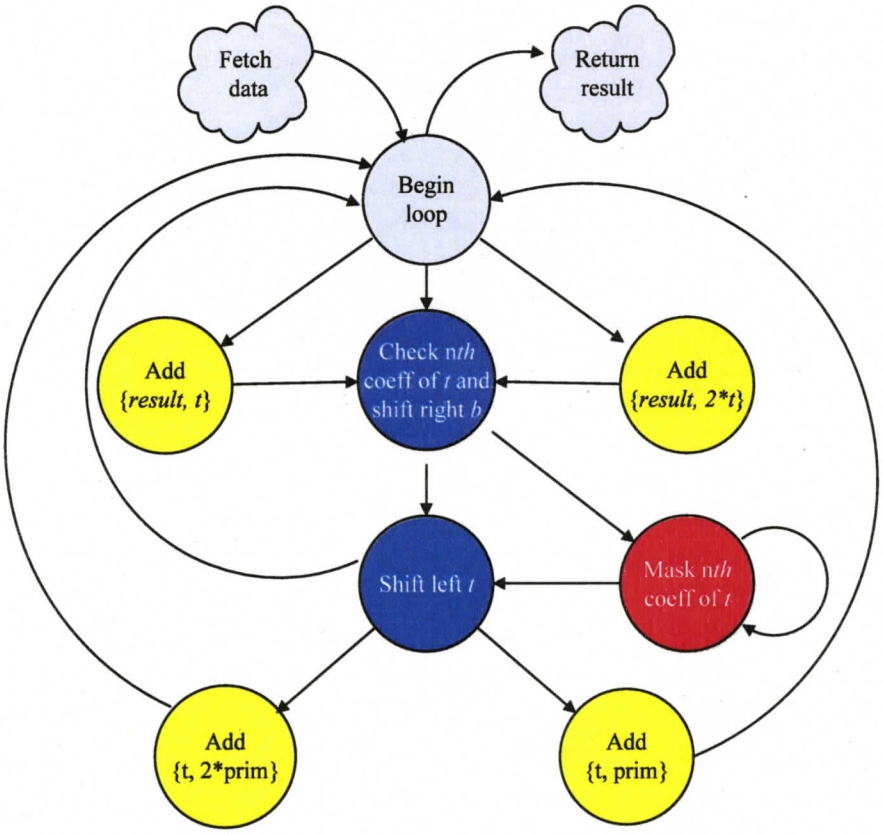


Figure 18: Arithmetic Control Unit Finite State Machine

To define the states, a detail analysis of the flowchart for the software code (Figure 17) was performed. The first attempt did not include any parallelism and simply assigned statements to states. After this sequential design, a second approach considered parallelism. Thus statements without any inter dependencies were combined in the same state. As an example, one can consider the last statement in the flowchart, namely "*right shift b*", in Figure 17. Since variable *b* is not dependent on any previous statement, it can be included in computations in earlier states. This appears in Figure 18, where the right shift of *b* is done in the same state where the coefficient of *n* is also checked (shown in the centre blue circle). The analysis to include parallelism exploited in hardware was repeated for the whole algorithm.

After a design phase, verification for correctness is crucial. In this work, verification was done manually using a complete trace of the software code and comparing it to the flow of the FSM. This is done to ensure that every state is followed by the correct instruction just as performed by the software. This certainly is a case when automated tools could be very useful, beyond full simulation and signals tracing.

2. **Synchronization process (*Sync* process).** This process of the design synchronizes the various signals.

3. **Interrupt handling process (*Int* process).** Interrupts signal the completion of a multiplication, so the result can be transferred back to software. For example, for the PCI-based environment (*Part 1*), the ACtU sends an interrupt signal to the PCI control unit and waits for the interrupt acknowledge signal from the system.

4. **Timing process (*Timer* process).** This process calculates the total number of clock cycles required to execute *multmod* in hardware.

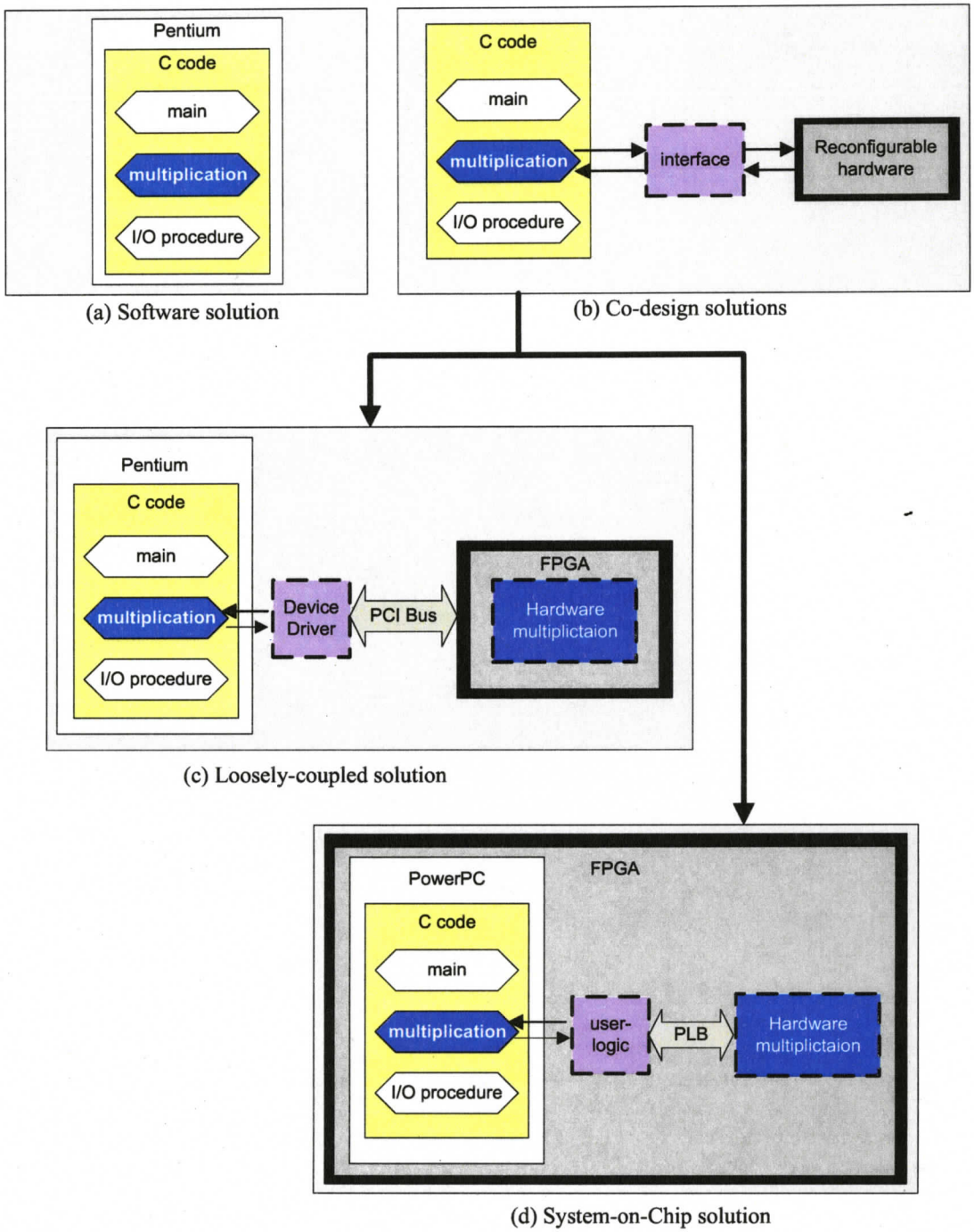


Figure 19: Solutions

After a general design is created, specific design decisions for the two different Co-design solutions are implemented in the two platforms available, as shown in Figure

19. Figure 19(a) illustrates the software solution, where the entire program is executed in software. Figure 19(b) shows the overall Co-design paradigm, while 19(c) and 19(d) illustrate the two HW/SW Co-design solutions in a loosely-coupled and integrated SoC environment. In 19(c), the interface and software implementation for the problem is located outside the reconfigurable hardware and communicates with the hardware multiplication via a PCI bus. In 19(d), a SoC environment is used where the software implementation and the interface are located on a processor embedded on the FPGA itself.

4.2 Part 1: Loosely-coupled Co-design Implementation

In a loosely-coupled environment, the reconfigurable hardware receives and sends information to the software application via a Peripheral Component Interconnects (PCI) bus. The general paradigm was discussed in Section 2.2. The hardware accelerator described here is implemented in VHDL and it is executed on an FPGA.

4.2.1 Hardware Environment

The first Co-design implementation is done on a Windows 2000 workstation equipped with a 1.8 GHz Intel Xeon processor and 1 Giga-Byte of RAM. The workstation has an Altera PCI Development Kit Version 2.1.0 that is used as the reconfigurable hardware platform. This development kit consists of the APEX PCI Development board with an Altera Apex 20KC1000CF672 FPGA [1]. Figure 20 shows a schematic view of the workstation and the development board. This board supports 32 bit and 64 bit PCI communication at both 33 and 66 MHz [2].

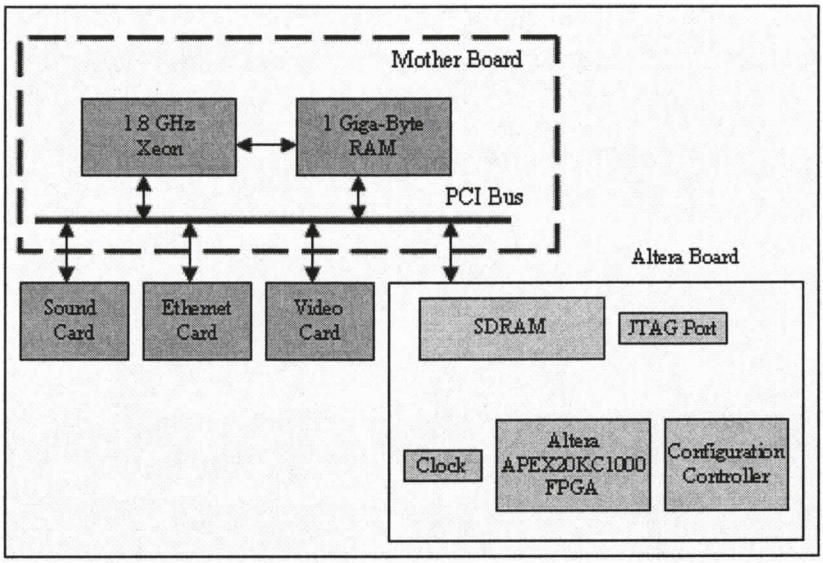


Figure 20: Development Hardware Semantic with Altera Board

The Altera APEX FPGA offers 38,400 logic elements (LE) and can be programmed in different ways, including a flash memory card that can be inserted to an onboard slot. The programming method used in this project was the ByteBlaster download cable connected to the parallel port of the host computer and using the JTAG standard to write configuration data to the device.

4.2.2 Software Environment

Altera's Quartus II [3] design software version 4.1 is used with Altera PCI Compiler 2.1.1 for the development of the multiplication. This tool provides a design entry, synthesis, place-and-route, verification and an FPGA programming environment. As a development environment for the Windows device driver, Jungo's WinDriver[6] software Version 8.0.2 was used. Debugging PCI bus communication was done using Jungo's diagnosis tools and Altera's PCI kit test application. The software module of this project was written in C and compiled using Microsoft Visual Studio 2003.

4.2.3 The Loosely-Coupled Design Structure

The main goal of this design is to implement the multiplication algorithm in hardware in an attempt to accelerate the existing software algorithm. To do so, the *multmod* is

[6] http://www.jungo.com

implemented in VHDL and it is executed on a FGPA that connected via the PCI bus to the host system. The software code consists of the original software implementation with modifications to support the hardware functionality. The high level view of the structure of the hardware and software interaction is illustrated in Figure 21.

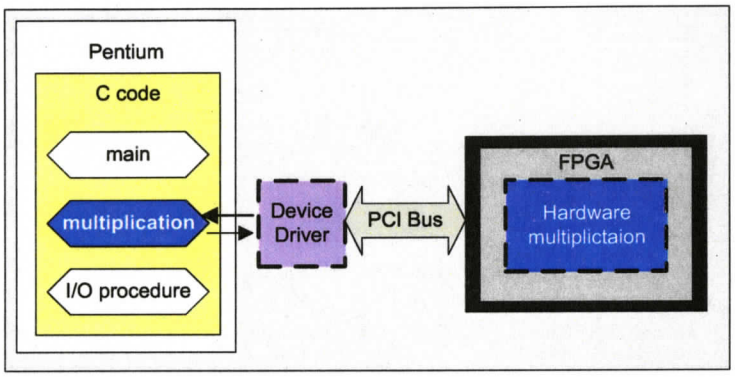


Figure 21: Design structure for Part 1 implementation

To interface the hardware accelerated *multmod* to the software implementation of the polynomial generation, a device driver is developed. Since developing device drivers manually is very difficult and error-prone, Windriver provided by the Californian vendor Jungo[7] is used. This tool generates Windows drivers based on a Wizard type interface[8].

The C code is modified to redirect all the calls made to *multmod* to hardware. To do so, these calls are sent to the device driver which redirects the input data to the FPGA. The opposite path is executed to return the results to the C code. This implementation is static; its configuration cannot be changed at runtime. The hardware implementation has to be downloaded into the FPGA before it can be used.

The design of the multiplication consists of the two modules described earlier (ACpU and ACtU) and two additional modules to implement the PCI functionality. These are

[7] http://www.jungo.com/windriver_usb_pci_driver_development_software.html, last accessed: 12 October 2007.
[8] The interface file (Interface32.vhd) was created by Jason Hannula in [9] as part of a M.Sc. thesis.

the PCI Core (PCIC) and PCI Control Unit (PCIU). Their interconnections are illustrated in Figure 22.

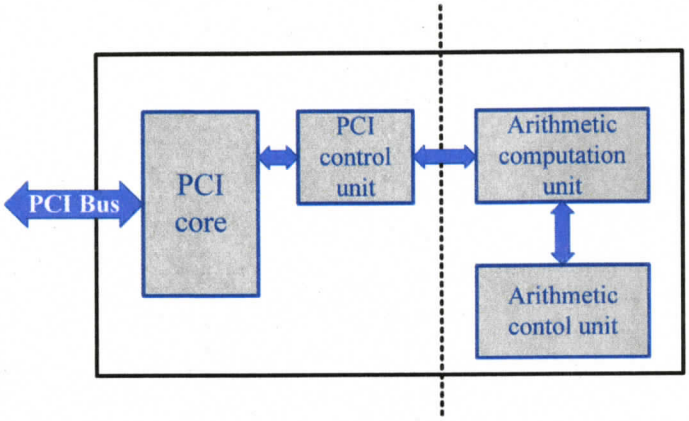


Figure 22: Multiplication hardware module structure

Each multiplication call sends input and output data through the PCI bus. Assuming a data word is 64-bit in length, the data transfer between hardware and software is composed of the following:

- 2 input polynomial *b* and *t*. Each polynomial consists of a top and bottom word (4 data words in total).
- Degree, ranging from 2 to 18 (1 data word).
- Primitive polynomial (top and bottom word, 2 data words in total).
- Output polynomial (top and bottom word, 2 data words in total).

4.2.3.1 Hardware Component

The hardware component consists of the *multmod* function implemented in VHDL. The ACpU, described in Section 4.1.1, occupies 1972 logic units on the FPGA. This module also contains several flip-flops provided by Quartus LPM library to store the operands of the computation.

To speed the hardware programming phase, two LPM Modules[9] provided by Altera was added to assist the implementation of ACpU.

[9] Library of Parametrized Modules. LPM modules can be parametrized and instantiated in designs to serve common tasks

a. **Decoder**: This module is denoted as *lpm_decode* and it is an auto generated file. In depth details can be viewed at page 8 of the LPM Quick Reference Guide [4]. This module takes 76 logic units on the FPGA and it converts the lower 6 bit of the loop counter signal that is supplied by the Arithmetic Control Unit into a 64 bit wide mask. This mask represents the degree in a one-hot encoding, having exactly one bit of the mask set to one at all times.
b. **D flip-flop**: This module is denoted as *lpm_ff* and it is also an auto generated file. In depth details can be viewed at page 26 of the LPM Quick Reference Guide [4]. Several flip-flops were allocated to provide storage for the operands of the computation. A total of nine 64-bit D-type flip-flops are used to hold the input values *b* and *t*, as well as the primitive polynomial *primpol* and the result value. One additional flip-flop is used for intermediate storage of input values during store operations.

The ACtU, described in section 4.1.2, occupies 374 LEs on the FPGA. For this part of the experiment the input and output polynomials are store in a FIFO buffer that is then used by the PCIU, described in the next section.

The *Timer process* is used to estimate the runtime of the program. To obtain accurate timing, a measurement of the execution including PCI bus transfer time was obtained at first and then a new measurement was acquired without the PCI bus transfer time. This allows further comparisons with the SoC Co-design. This timing is obtained by using a variable that acts as a counter accumulating clock cycles.

4.2.3.2 Interface Component

To allow communication between hardware and software, the PCI functionality is implemented with PCIC and the PCIU. The PCIC was provided as an Intellectual Property core (IP core) by Altera together with the PCI card. This core is available in four different variations.

1. **pci_mt32**: Master-core and target-core for 32 bit bus width.
2. **pci_mt64**: Master-core and target-core for 64 bit bus width.

3. **pci_t32**: Target-core for 32 bit bus width.
4. **pci_t64**: Target-core for 64 bit bus width.

All the operations in this project are implemented in target mode, meaning that the initiation of the data transfer is done by the driver software, not by the PCI core on the FPGA. Although, for this experiment, only target functionality is necessary, used *pci_mt64* is used for flexibility in future expansion of the problem. The *pci_mt64* function provides a fully compliant, 64-bit, 66-MHz PCI bus interface with bus mastering support. There is an extra overhead due to this larger IP, but the compiler removes most of the unused logic during synthesis process

The PCIU connects the PCI core to the ACtU as shown in Figure 22. This component is based on the Altera's reference design [5]. This unit takes 176 logical units in the FPGA. The PCIU implements the PCI functionality that is necessary to perform proper communication between the FPGA and the software code and requires two FIFO buffers where the input and output data is stored.

At power-up, the system enters an *Init* state. After all signals have been initialized, the PCIU waits until a request from the PCI core or the ACtU arrives. The PCIU can perform two different kind of operations; *input-write* and *result-read.* The *input-write* transaction is used to send all data to the FPGA that is needed to start the multiplication of the two polynomials. The data transmitted consists of the following: (i) input polynomial b (2 words), (ii) input polynomial t (2 words),(iii) primitive polynomial *primpol* (2 words) and (iv) the degree of the polynomial that is to be computed N (1 word). Due to the restrictions of the PCI bus, only one word can be transmitted at a time, therefore the transmission of one polynomial takes 2 bus transactions, one for the top word and one for the bottom word.

Before all the input data is sent, the PCIU checks to ensure sufficient space on the input buffer. If so, then it starts the writing process by performing one of the two transactions present: *input-write* or *result-read.* If unsuccessful, the transaction is aborted. *Input-write* sends the input data to the FPGA. *Result-read* sends back the

multiplication result to the software code running on the host system. This implementation uses interrupts to indicate when the multiplication is finished and the transfer of the result can be initialized. This transaction is implemented as a target read in terms of the PCI standards and can only be accepted by the PCIU when the PCI bus is in *idle* mode.

4.2.3.3 Software Component

The original software implementation (gf3.c) is used for this Co-design. The algorithm is based on an extension of a GF(2) algorithm published in [8]. This C code was modified to call the hardware implementation when performing the multiplication in an attempt to accelerate the polynomial generation. The multiplication function in software is called *multmod* and now re-routes to the FPGA for the computation.

4.3 Part 2: System-on-Chip Co-design Implementation

In the System-on-Chip (SoC) environment used for the second part of the research, a general purpose processor is closely-coupled with the custom hardware, providing a complete computing platform within an FPGA. This differentiates it from a loosely-coupled implementation, where the reconfigurable hardware communicates with the software via a PCI bus. The loosely-coupled Co-design (Part 1) showed a very large bottleneck for the movement of data on the PCI bus. To overcome this obstacle, this implementation is done by bringing the processor and execution engine closer together in an attempt to reduce the communication overhead. In this approach a different board is used where a PowerPC is directly embedded in the reconfigurable hardware device (the FPGA) in a SoC platform.

The addition of polynomials (*add*) is included in the hardware computational engine, in addition to the multiplication of polynomials (*multmod*). The *add* function is called by the *multmod* function as well as by other functions (e.g. *minpoly*). The *add* function itself uses less than 5% of the total software runtime. However, by doing this the design has been modified to handle two components in hardware, and the process of adding further components has been set. The movement of additional software

modules to hardware is facilitated by this initial addition. Another important aspect is the decrease in communication between software and hardware is accomplished by moving *add* to hardware. For simplicity, and for better understanding of the next few sections, the design of the hardware *multmod* is used as the main reference. The design of add in hardware follows similar steps.

4.3.1 Hardware Environment

The development was done on an Intel Xeon based PC running Windows XP Professional with an Intel Xeon 3.4 GHz processor with 3GB of RAM. The reconfigurable hardware used was an Amirix AP1100 FPGA development board [6]. The schematic for the Amirix board is illustrated in Figure 23.

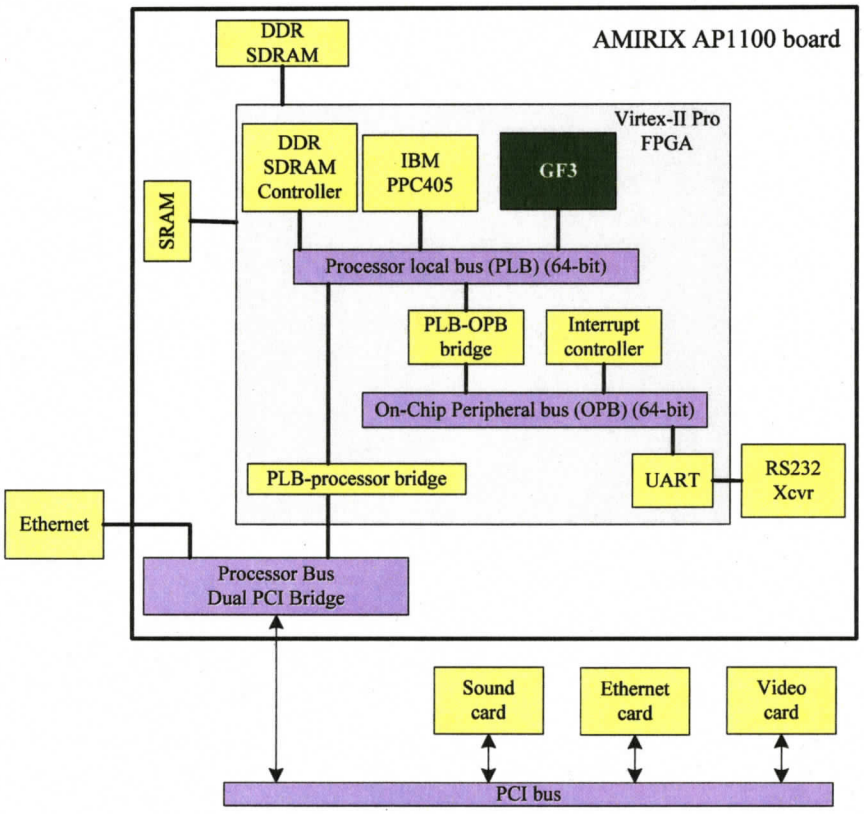


Figure 23: Amirix Platform Environment

This board is equipped with a Xilinx VirtexII Pro part XC2VP100 with an embedded IBM PowerPC 405 processor. The FPGA has the following characteristics:

- 99,216 logic elements;
- 444 18X18 bit multiplier blocks;
- 7,992 Kb of block RAM on chip;
- 12 Digital Clock Managers (DCMs).

The PowerPC embedded in the FPGA has a 64KB of direct accessible block memory that can be initialized during the programming of the FPGA. The main components of this board are connected using the CoreConnect Processor Local Bus (PLB); an 80 MHz high speed shared medium bus. Since speed is important to this work having a faster FPGA and extra area is advantageous.

4.3.2 Software Environment

The software used for this part of the experiment consists of the Xilinx ISE Foundation 8.1i CAD and the Xilinx Embedded Development Kit (EDK) 8.1i packages. These tools provide a complete development environment, which also provides a set of IP cores that can be integrated in the design. These cores are generated using Xilinx Core Generator Version 8.1i. For simulation, two tools are used: ModelSim [12] version 6.1f by Mentor Graphics and the ISE 8.1 Simulator.

4.3.3 The System-on-Chip Design Structure

The SoC Co-design was developed in three phases based on the components created.

1. **Hardware component**: This involves creating a Xilinx implementation for the functions implemented in hardware in order to place them in the FPGA. This step was accomplished by making modifications to the top level design previously used in Part 1 towards an Altera implementation.
2. **Interface component**: This component contains the 'interface', labeled *user-logic* in this design, used to communicate between software and hardware.
3. **Software component**: Modification to the original software code is needed to support the new hardware modules. Also, extra changes became necessary to avoid the use of the standard C library – this is due to memory limitation that is explained below.

Figure 24 gives a schematic of how these components interconnect. The crucial difference from the loosely-coupled environment is that the software module here executes on a processor (the PowerPC) which is located within the FPGA itself.

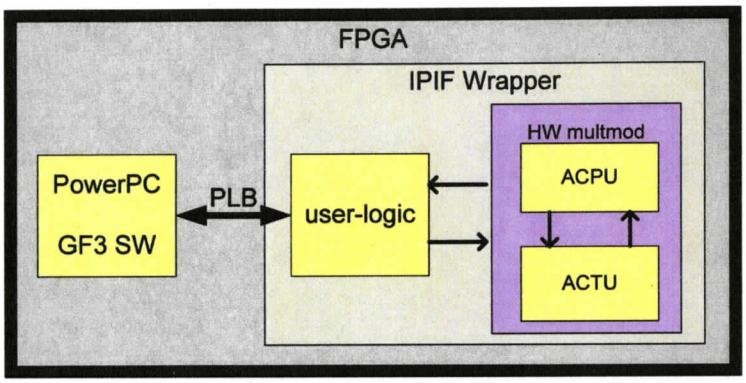


Figure 24: Components Interconnection inside the FPGA

The *add* function hardware implementation follows a very similar pattern. The *user-logic* component for the interface is duplicated with minor modifications, such as the number of registers used, since the *add* requires only two inputs: *b* and *t*. A total of 5 registers are used, 2 for each input and 1 for the total number of clock cycles required to execute the *add* operation.

The total hardware implementation, including *add* and *multmod* occupies 10,680 logic elements (LEs) from the 88,192 available, that is, approximately 12% of the FPGA area.

4.3.3.1 Hardware Component

Xilinx and Altera are competing companies, and as such, their design tools are not compatible, nor are the files directly transferable. Therefore, the hardware implementation, currently in an Altera design, needs to be modified and ported to a Xilinx environment. Fortunately, the hardware description language, VHDL, was

used for design entry and is a standard language that can be recompiled to other platforms. However, modifying a hardware design from one environment to another still requires that a few key items be considered. First, any previous IP core must be replaced. Secondly, the PCI bus used for the Altera Co-design is eliminated. Thirdly, a new interface must be developed.

As stated in section 4.2, the implementation of the problem in the Altera environment (Part 1) used three Altera LPM IP modules. These modules are provided by Altera and are only compatible with the Quartus CAD tool [4]. For this second design, the Xilinx LogiCOREs[10] cores are substituted for the Altera ones. The complete list is as follows:

- **PCI core**: The PCI core (*pci_mt64*) was not replaced in the new design as the new board communicates differently than the Altera board. The Amirix Platform development board uses a Tundra PowerSPAN II PCI bridge and does not have the PCI edge connector directly connected to the FPGA as the Altera APEX development board does [19]. In this environment the communication of the PowerPC and hardware is done via a shared medium bus IBM CoreConnect Processor Local Bus (PLB).
- **Decoder**: The decoder (*lpm_decode*) module was replaced by Xilinx Binary Decoder 7.0 with 64-bit outputs [23].
- **Flip-Flop**: The flip-flop (*lpm_ff*) core module had to be replaced by a new custom written module, since Xilinx does not provide an equivalent one.

Once the IP cores are replaced, several modifications to the hardware design are also necessary to support the removal of the PCI bus. The main change to the design is the absence of *input-read* and *input-write* functionalities, where input is sent to the hardware via the PCI bus and result output is read by the software. As explained in section 4.1, the initial design supported PCI reading/writing and each input/output required a clock cycle to complete the operation. In the current architecture, the

[10] Xilinx LogiCORE libraries can be parametrized and instantiated in designs to serve common tasks

input/output values are stored in a register by the software. The hardware reads the registers in a single step.

After adding functionality to allow the hardware to read/write to/from registers, the hardware component also needs full testing. First, a full simulation is performed using two simulators, the Mentor Graphics ModelSim and the ISE 8.i Simulator. ModelSim was used initially. Due to the fact that it does not support 64-bit inputs, it needed to be incremented by the ISE simulator (which has the same functionality). Data was collected from the software implementation, such as expected input/output pairs for a few sets of *multmod* calls. These sets of results were then added as input to the registers allowing the simulator to be run. The registers holding the resulting values were verified to ensure the values produced matched the software values collected by running the whole program in software.

4.3.3.2 Interface Component

To allow the communication between the hardware and the software component, an interface component is needed. The Xilinx Intellectual Property Interface (IPIF) provides a means to connect a user-logic component to a bus in the IBM CoreConnect architecture. The IPIF abstracts much of the details of the bus away from the software, including bus timing. The IPIF provides a master or slave attachment to the bus or both if required. The master attachment allows the IPIF to access other devices directly on the PLB, while the slave attachment allows other devices to access the IPIF directly. In this design the master attachment is not used. The slave attachment provides access to software addressable registers and an interrupt service.

Slave Register Name	Value
slv_reg0	Input - Indicates if input is ready to start multiplication Output - User logic returns the number of clock cycles
slv_reg1	Degree of polynomials to be computed
slv_reg2	Primitive polynomial *primpol* top word
slv_reg3	Primitive polynomial *primpol* bottom word
slv_reg4	Input polynomial *b* top word
slv_reg5	Input polynomial *b* bottom word
slv_reg6	Input polynomial *t* top word

slv_reg7	Input polynomial *t* bottom word
slv_reg8	Result polynomial top word
slv_reg9	Result polynomial bottom word

Table 1: Register assignment

For the multiplication of polynomials, a total of 10 registers are used in the design for input/output data storage purposes. As noted in section 4.1, the polynomials are described as two 64-bit words. For this reason, each register holds a 64-bit input or output. Register 1 is the *Go* register. When the polynomial generation problem on the PowerPC writes the start key (*0xAAAA*) to the *Go* register, it indicates that all the registers are loaded and the hardware execution can be started. This register is also used as storage for the total number clock cycles taken to execute the hardware execution. The clock cycle count is later used to calculate the total hardware runtime. All register descriptions are listed in Table 1.

An additional register is defined by the IPIF and not by the software. The *Interrupt Status* register is used to observe when an interrupt has occurred and to clear the interrupt state.

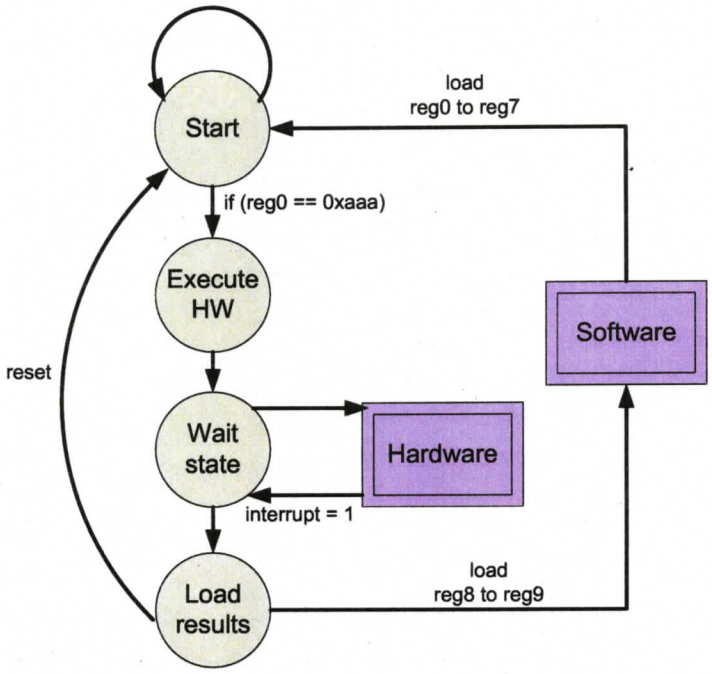


Figure 25: User-logic FSM for hardware and software communication

The user-logic component is responsible for sending and receiving data to/from the hardware. To do so, it contains an FSM that controls the flow of which module should be executing at a given time. For example, the control should only be sent back to the software module on the PowerPC when the hardware calculation is completed. Also, the hardware should only start the multiplication when all the input registers are loaded. The FSM for the multiplication user-logic is illustrated in Figure 25.

The *user-logic* component stays idle until the hardware finishes its operation. The hardware sends an interrupt once the result registers are loaded with proper values, indicating to the *user-logic* that the software component can access the result registers.

4.3.3.3 Software Component

The third step of this development phase is the implementation of extra functionality for the software component. The software implementation used here is an extension

of the original code. The current software executes most operations in C, except for the multiplication and addition of polynomials which are moved to hardware.

As shown in

Figure 24, the software module is located in an IBM PowerPC embedded in the VirtexII FPGA with 64 KB of direct memory. The polynomial generation software program occupies approximately 53KB of the memory available. For future reference, if the C code is extended and if the size is greater than 64KB then the software code will have to be placed in main memory. If main memory is used then master functionality must be implemented to allow the communication of the software located memory with PowerPC via serial port. Due to the size limitation of the software code, some custom made C functionality was added. Standard C library functions were not used because they offer more functionality and occupy more space than necessary in the 64KB available. The new custom functions added are listed below in Table 2.

Function	Purpose
NS16550_getc	Reads a character
NS16550_putc	Writes a character
NS16550_puts	Write a string
NS16550_gets	Reads a string
Reverse	Reverses the bytes in a string
strlength	Finds the length of a string
Itoa	Converts integer to string
Ulltoa	Converts unsigned long long to string
Axtoi	Converts string to unsigned integer

Table 2: Software library functions

Two Xilinx library functions are included to deal with interrupts and exceptions on the PowerPC. These libraries, *xintc* and *xexception*, are designed for embedded systems and are small enough to fit in the available space. The software code for is

compiled using the *powerpc-eabi-gcc*[11] cross compiler. This compiler is contained in the Xilinx EDK environment and allows programs for the PowerPC to be written and compiled on a Windows platform. Debugging and testing is done using the Xilinx Microprocessor Debugger (XMD). This allows the testing of the software code on the PowerPC without having a full operating system and development tools running on it.

The software executes as the original version until it reaches *multmod* or *add*, when it starts loading the input registers. While the hardware executes, the software loops and monitors to determine if the *user-logic* has signaled any interrupt. When the hardware execution finishes, the software reads the result words from registers 8 and 9. The number of total clock cycles required to run the hardware multiplication or addition is read from register 0, which was loaded by the *user-logic*.

[11] Compiler used with Optimization 3 (O3)

Chapter 5: Results

With two designs in different environments, experiments were performed with the following objectives:

- to compare the performance of *multmod* between hardware and software;
- to compare the software, hardware and interface modules between the two environments.

The reader may find it useful to review a summary of all the modules, illustrated below in Figure 26. The dashed boxes indicate the portions for which experiments were conducted and timing collected, as explained in detail in the following sections. The numbers used as labels refer to the corresponding tables of results. The experiments performed fall into these main categories:

1. The application is timed using the original software implementation (Section 5.1).
2. The *multmod* function is timed in software for both hardware portion of the Co-designs (Section 5.2).
3. Timing for the Co-design for Part 1 is collected (Section 5.3).
4. Timing for the Co-design for Part 2 is collected (Section 5.4).

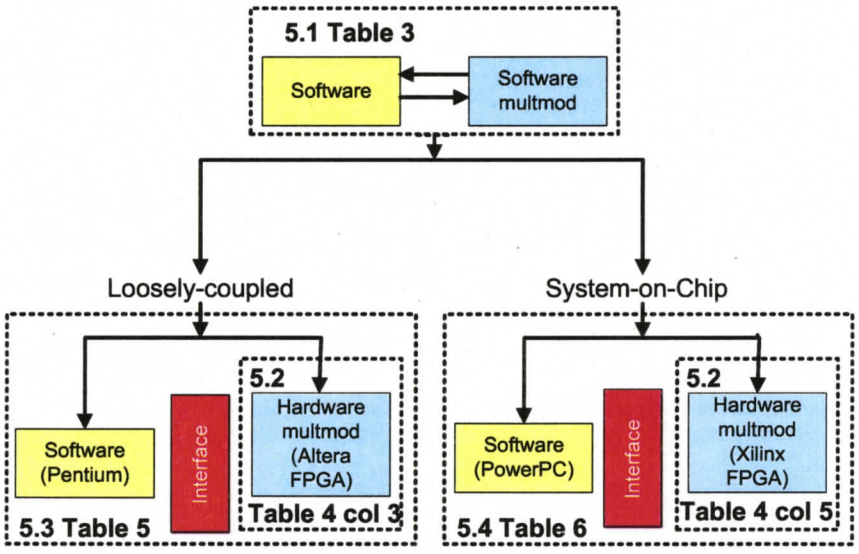


Figure 26: Co-design solutions summary

5.1 Original Software Runtime Results

The running time of the design implemented 100% in software provides a reference point for any accelerated version. The results in Table 3 are organized as follows:

1. **(Col. 1) Polynomial degree**: Input degree entered by the user.
2. **(Col. 2) Number of irreducible polynomials**: This indicates the resulting output of the number of irreducible polynomials.
3. **(Col. 3) Number of *multmod* calls**: Output reflecting the number of calls to the function *multmod.*
4. **(Col. 4) All-in-software runtime**: Output runtime in seconds obtained by executing the software program[12] in the Pentium.

Col. 1	Col. 2	Col. 3	Col. 4
Polynomial Degree	**# of irreducible polynomials**	**# of *multmod* calls**	**All-in-software runtime (sec)**
2	3	32	0.059
3	8	160	0.061
4	18	566	0.064
5	48	2,168	0.072
6	116	7,056	0.075
7	312	24,536	0.085
8	810	79,724	0.241
9	2,184	262,640	0.681
10	5,880	847,268	2.102
11	16,104	2,736,896	7.583
12	44,220	8,747,364	26.603
13	122,640	27,924,008	88.175 (1.47 min)
14	341,484	88,640,036	318.941 (5.31 min)
15	956,576	280,719,824	1,036.520 (17.27 min)
16	2,690,010	885,978,776	3,624.602 (60.41 min)
17	7,596,480	2,789,886,944	16,560.152 (4.6 hrs)
18	21,522,228	8,763,137,088	40,716.203 (11.31 hrs)

Table 3: Output values for the initial software program where the entire application is implemented in C on the Pentium

5.2 Multmod Module only Results

To detect whether the multiplication of polynomials is accelerated, three different running times are collected for the *multmod* function on its own. These *multmod*

[12] The profiling tool *ptime* is used for timing results.

modules can be viewed in Figure 26 (blue blocks). The results in Table 4 are organized as follows:

1. **(Col. 2) *Multmod* software runtime**: Time in seconds for the execution of only the *multmod* function.
2. **(Col. 3) *Multmod* Altera FPGA runtime (Part 1: loosely-coupled Co-design)**: Time in seconds for the *multmod* hardware implementation on the Altera Apex20K FPGA. This number is calculated by dividing the number of clock cycles, collected for the execution, by the design's maximum frequency, in this case 66MHz.
3. **(Col. 5) *Multmod* Xilinx FPGA runtime (Part 2: System-on-Chip Co-design)**: Time in seconds for the *multmod* hardware implementation on the VirtexII Pro FPGA. This number is calculated by dividing the number of clock cycles, collected for the execution, by the design's maximum frequency, in this case 80MHz.
4. **(Col. 4 and 6) Percentages**: They refer to the ratio with the software runtime of column 2. For example, for degree 10, the Altera FPGA runtime (Col. 3) of 0.89412 seconds is divided by the 1.898 seconds (Col. 2) which gives 0.471, implying that the execution time of column 3 is 47.1% of the software execution of column 2. Similarly, column 6 indicates that the execution time for column 5 is 35% of the software execution in column 2.

Therefore, the hardware implementation on the Altera FPGA executes *multmod* almost two times faster than the software implementation. As for the Xilinx FPGA (Col. 5), the hardware implementation for degree 10 takes 35% of the software time (Col. 6 = Col. 5 divided by Col. 2), i.e. *Multmod* executes approximately 3 times faster than the software *multmod* (Col. 2). Thus it appears that the hardware implementation of the Xilinx design (Col. 5) is faster than the Altera design (Col. 3) for *multmod.*

Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6
Degree	Pentium @1.8GHz runtime (sec)	Altera FPGA@66MHz runtime (sec)	%	Xilinx FPGA@80MHz runtime (sec)	%
2	0.00004	0.00002	*37.6*	0.00001	*24.8*
3	0.00018	0.00009	*46.6*	0.000056	*30.7*
4	0.00076	0.00034	*44.2*	0.000228	*30.0*
5	0.00298	0.00145	*48.6*	0.001005	*33.7*
6	0.01495	0.00519	*34.7*	0.003664	*24.5*
7	0.04043	0.02005	*49.6*	0.01439	*35.6*
8	0.14300	0.07123	*49.8*	0.051788	*36.2*
9	0.54600	0.25595	*46.9*	0.188174	*34.5*
10	1.89800	0.89412	*47.1*	0.663513	*35.0*
11	6.24000	3.08083	*49.4*	2.302203	*36.9*
12	22.30800	10.58020	*47.4*	7.963267	*35.7*
13	74.78900	35.12896	*47.0*	26.53804	*35.5*
14	263.53600	120.09697	*45.6*	91.323996	*34.7*
15	888.30300	400.77343	*45.1*	306.075094	*34.5*
16	2985.50200	1343.56091	*45.0*	1049.41517	*35.9*
17	10192.85900	4424.87400	*43.4*	n/a	n/a
18	32658.34090	14642.10675	*44.8*	n/a	n/a

Table 4: *Multmod* runtime in seconds and its percentage related to the original software implementation

Obviously the different clock speed is a factor for the improved performance of *multmod* in the Xilinx design (Col. 5) over the Altera design (Col. 3), since the Xilinx FPGA runs at 80MHz and the Altera FPGA runs at 66MHz for this Co-design.

Timing for all the desired degrees could not be obtained for the Xilinx Co-design (see Table 4, rows 17 and 18) due to the license restrictions for the serial port controller on the board (which can only be used for 48 hours after programming the device).

5.3 Part 1: Loosely-Coupled Co-design Result

Timing for the loosely-coupled environment (Part 1) is collected as illustrated in Figure 27, showing the different sections where timing was obtained, relating them to the columns in Table 5.

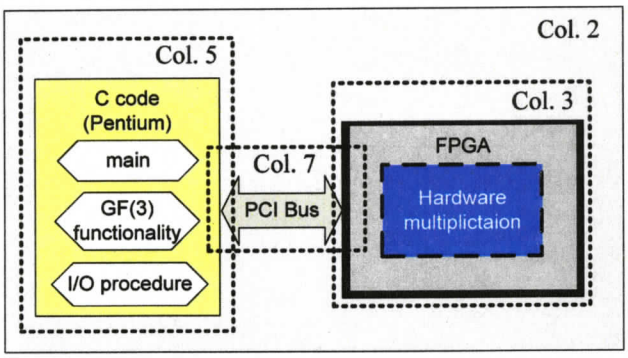


Figure 27: Timing result sections for Part 1

Table 5 is organized as follows:

1. **(Col. 2) Total runtime**: Number in seconds representing the execution of the entire Co-design including software, hardware and communication in the PCI-based environment.
2. **(Col. 3) Hardware runtime**: Time in seconds for the *multmod* hardware implementation executed on the Altera Apex20K FPGA. These are the same as in column 3 of Table 4.
3. **(Col. 5) Software runtime**: Number in seconds for the software as shown in Figure 27.
4. **(Col. 7) Communication runtime**: This number reflects the runtime to transfer the input/output to and from the software and hardware via the PCI bus.
5. **(Col 4, 6 and 8) Percentages**: The percentages refer to the ratio related to the total runtime of column 2. For example, for degree 10, the hardware runtime (Col. 3) of 0.89412 seconds is divided by the 12.871 seconds (Col. 2) which gives 0.06948, implying that the execution time of column 3 is 6.95% of the total execution of column 2. Similarly, the software execution for degree 10 takes 2.28% and the communication between software and hardware takes 90.75% of the total execution time of in column 2.

Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6	Col. 7	Col. 8
Polyn. degree	Total runtime (sec)	HW runtime (sec)	HW %	SW runtime (sec)	SW %	Comm. runtime (sec)	Comm. %
2	0.094	0.00002	*0.02*	0.004	*4.22*	0.090	*95.74*
3	0.074	0.00009	*0.11*	0.009	*11.6*	0.060	*81.08*
4	0.076	0.00034	*0.44*	0.009	*11.8*	0.070	*92.11*
5	0.100	0.00145	*1.45*	0.010	*10.0*	0.090	*90.00*
6	0.179	0.00519	*2.90*	0.011	*6.15*	0.160	*89.39*
7	0.446	0.02005	*4.49*	0.012	*2.69*	0.410	*91.93*
8	1.272	0.07123	*5.60*	0.034	*2.67*	1.170	*91.98*
9	4.069	0.25595	*6.29*	0.095	*2.33*	3.720	*91.42*
10	12.871	0.89412	*6.95*	0.294	*2.28*	11.680	*90.75*
11	41.790	3.08083	*7.37*	1.062	*2.54*	37.650	*90.09*
12	133.814	10.58020	*7.91*	3.724	*2.78*	119.510	*89.31*
13	440.515	35.12896	*7.97*	12.345	*2.80*	393.041	*89.22*
14	1380.442	120.0969	*8.70*	44.652	*3.23*	1,215.690	*88.07*
15	4388.144	400.7734	*9.13*	145.113	*3.31*	3,842.260	*87.56*
16	13899.938	1,343.568	*9.67*	507.501	*3.65*	12,048.870	*86.68*
17	44447.226	4,424.870	*9.96*	1,825.061	*4.11*	38,197.290	*85.94*
18	139500.798	14,642.185	*10.5*	5,476.691	*3.93*	119,382.010	*85.58*

Table 5: Loosely-coupled environment (Part 1) results

It can be observed that the software execution is considerably fast since it is still being run in a Pentium at 1.8GHz. The communication takes an average of 90.03% of the total execution time due to the PCI bus that only allows one word to be on the bus at a time creating a communication bottleneck.

5.4 Part 2: SoC Co-design Results

Timing for the Co-design on the SoC environment is collected as illustrated in Figure 28, which also shows the different sections where timing was obtained relating them to the columns in Table 6.

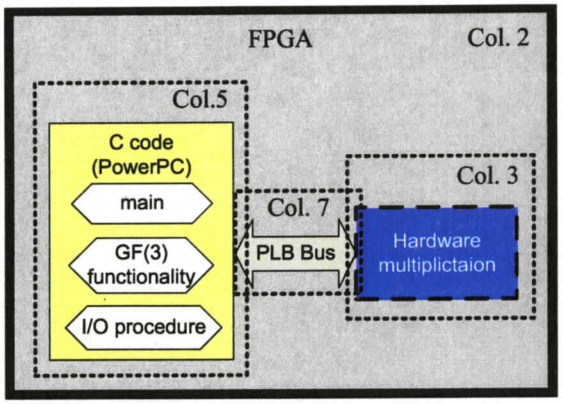


Figure 28: Timing result sections for Part 2

Table 6 is organized as follows:

1. **(Col. 2) Total time**: Number of seconds representing the execution of the entire Co-design implementation including software, hardware and communication in the SoC environment.
2. **(Col. 3) Hardware runtime**: Time in seconds for the *multmod* only hardware implementation executed on the Xilinx FPGA. These are the same as in column 5 of Table 4
3. **(Col. 5) Software runtime**: Runtime of software module only on the PowerPC. As it can be observed in Figure 28, this timing value does not include the *multmod* function.
4. **(Col. 7) Communication runtime**: This number reflects the runtime to transfer the input/output to and from the software/hardware.
5. **(Col 4, 6 and 8) Percentages**: The percentages refer to the ratio related to the total runtime of column 2. For example, for degree 10, the hardware runtime (Col. 3) of 0.66351 seconds is divided by the 51.023 seconds (Col. 2) which gives 0.013, implying that the execution time of the multmod function on the Xilinx environment is 1.3% of the total Co-design execution. Similarly, the software execution for degree 10 takes 75.59% and the communication between software and hardware takes 19.12% of the total execution.

Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6	Col. 7	Col. 8
Polyn. degree	Total runtime (sec)	HW runtime (sec)	HW %	SW runtime (sec)	SW %	Comm. runtime (sec)	Comm. %
2	0.002	0.00001	*0.55*	0.0012	*63.63*	0.0004	*22.09*
3	0.008	0.00006	*0.66*	0.0058	*68.51*	0.0020	*23.55*
4	0.029	0.00023	*0.78*	0.0192	*65.73*	0.0070	*24.00*
5	0.116	0.00101	*0.86*	0.0766	*65.83*	0.0250	*21.47*
6	0.377	0.00366	*0.97*	0.2756	*73.16*	0.0810	*21.50*
7	1.407	0.01439	*1.02*	1.0934	*77.72*	0.2820	*20.05*
8	4.507	0.05179	*1.15*	3.4563	*76.69*	0.9180	*20.37*
9	15.683	0.18817	*1.20*	11.652	*74.30*	3.0240	*19.28*
10	51.023	0.66351	*1.30*	38.687	*75.59*	8.7540	*19.12*
11	172.684	2.30220	*1.33*	129.901	*75.19*	31.5090	*18.25*
12	538.842	7.96327	*1.48*	417.856	*77.55*	100.7040	*18.69*
13	1,865.070	26.53804	*1.42*	1423.871	*76.34*	321.4750	*17.24*
14	5,985.520	91.32400	*1.53*	4650.526	*77.70*	1,020.4680	*17.05*
15	19,273.601	306.01750	*1.59*	14595.685	*75.73*	3,231.7870	*16.77*
16	66,421.321	1049.41517	*1.58*	51177.627	*77.05*	11,059.1593	*16.75*

Table 6: System-on-Chip environment (Part 2) results

The software (Col. 5) and the communication (Col. 7) timings are obtained by accumulating their respective clock cycles, using a Xilinx profiling library named *xtime_l* [22]. This library provides access to the 64-bit time base counter on the PowerPC. This counter increases by one at every processor cycle. The standard C *time* library was not used because it is not supported by the PowerPC.

5.5 Part 1 and Part 2 Comparison

Using the results for degree 10 as an example for clarification, (see Table 7), one can make a few observations:

	Total runtime	HW runtime	SW runtime	Communication runtime
Part 1	12.871 sec	0.894 sec (6.95%)	0.294 sec (2.28%)	11.68 sec (90.75%)
Part 2	51.023 sec	0.663 sec (1.3%)	38.687 sec (75.59%)	8.754 sec (19.12%)

Table 7: Timing results for both Co-designs for degree 10

- The total runtime for Part 2 is higher than for Part 1. This increase can be attributed to the significant increase in the software module runtime (from 0.294 to 38.687 seconds). The software runtime for Part 2 (running at

240MHz on a PowerPC) is higher than Part 1 (running at 1.8GHz on a Pentium).

- The communication in Part 2, accounting for 19.12% of the total runtime, is faster than the communication in Part 1of approximately 90% of the total runtime. The communication path is much shorter since the PowerPC is embedded within the FPGA itself. All the data is transferred in one single clock cycle by loading the assigned registers. The same happens on the hardware side, where it reads all the registers at the same time.

5.5 Results for 100% Software

To obtain a more detailed comparison between the Pentium and the PowerPC, timing of the application running in software only was obtained in both platforms as shown in Table 8. In this case, the entire application was executed in software with no hardware calls to *multmod*. The *ratio* of the two (Col. 4) reflects the rate of the PowerPC runtime (Col. 3) over the Pentium runtime (Col 2). The positive ratio indicates the Pentium is faster than the PowerPC and the negative ration shows how many times slower the Pentium is compared to the PowerPC. For example, for degree 11, the PowerPC runtime is 80.69 times slower than the runtime of the software on the Pentium.

Col. 1	Col. 2	Col. 3	Col. 4
Polynomial degree	**Pentium @1.8GHz runtime (sec)**	**PowerPC @240MHz runtime (sec)**	**Pentium vs. PPC (times faster)**
2	0.059	0.003	*- 19.75*
3	0.061	0.014	*- 4.35*
4	0.064	0.055	*- 1.16*
5	0.072	0.255	*+ 3.54*
6	0.075	0.906	*+ 12.08*
7	0.085	3.750	*+ 44.12*
8	0.241	17.334	*+ 71.93*
9	0.681	51.474	*+ 75.59*
10	2.102	176.555	*+ 83.99*
11	7.583	611.835	*+ 80.69*
12	26.603	2,116.979	*+ 79.58*
13	88.175	7,070.623	*+ 80.19*
14	318.941	24,721.771	*+ 77.51*
15	1,036.520	82,975.982	*+ 80.05*

Table 8: Software application runtime with no hardware functionality

For degrees 2, 3 and 4, the PowerPC is faster than the Pentium. This is possible due to various reasons; for example, when the execution of the software implementation is requested, the PowerPC is idle waiting for a task. The opposite happens with the Pentium where other tasks are being executed when the request for the polynomial generation code occurs. To run the C program, the Operating System must halt the task in hand, save data if necessary and then does a context switch to the C application. Another factor to consider is that the PowerPC already contains the code to be executed in local memory. In a Pentium machine, the execution code needs to be fetched into local memory bits before it can be run.

The raw data of the experiments can be evaluated from many perspectives. Final answers are not absolutely definitive, yet some analysis and ideas are discussed in the next chapter.

Chapter 6: Analysis, Conclusions and Future Directions

This work presents an overall Co-design for a computationally intensive algorithm over the finite field GF(3), together with two distinct FPGA-based implementations. A derivation was done both from an existing software solution and from a similar FPGA-based solution, while exploiting a more efficient internal representation. Here, analysis and comments on the Co-design and the two implementations are presented, followed by a view for future directions.

6.1 Observations on Timing

Differences in timing are found in different portions of the Co-design and for the two platforms, so it is useful to analyze them separately. Based on timing results presented earlier, it can be observed that the multiplication of polynomials was accelerated when implemented in a reconfigurable hardware. If the whole application is considered, the acceleration is not present due to platform restrictions described next.

Acceleration *of Multmod*

In section 5.2 (Table 4) timing results show that acceleration was achieved for *multmod* when executing in a PCI-based environment, with a speed of approximately 2 times faster than the software version. Furthermore, the implementation of *multmod* in a SoC environment is approximately 3 times faster than the software version. Some observations can be made on how this acceleration over the software implementation was achieved.

- **Data structure**: A custom data structure is used to represent the polynomials in a bit-sliced manner. Two 64-bit words are used for the coefficients of the polynomials, such that operations can be carried out on both words/slices in parallel, top and bottom words, whenever necessary.
- **Custom instructions**: The operations used often in *multmod*, for example "$a = a + 2b$" are not generally available in the basic instruction set of a general microprocessor as such. Such relatively simple operations still require more

than one instruction. Here, all such operations are analyzed and implemented as custom modules, basically designing a pre-defined computational unit with a custom instruction set, as explained in details in section 4.1.

- **Parallelism**: One of the main features of any hardware design, easily expressed in a hardware language, is the parallelism available for the processes. Section 4.1 describes some of the parallelism used in the creation of the ACtU FSM (the control unit). Conversely, software implementations remain limited to the capabilities offered by a sequential target processor. The parallelism introduced was achieved by doing a re-design starting from the flowchart of the software-based solution, towards an isomorphic FSM, exploiting the inherent parallelism of the algorithm (see section 4.1).

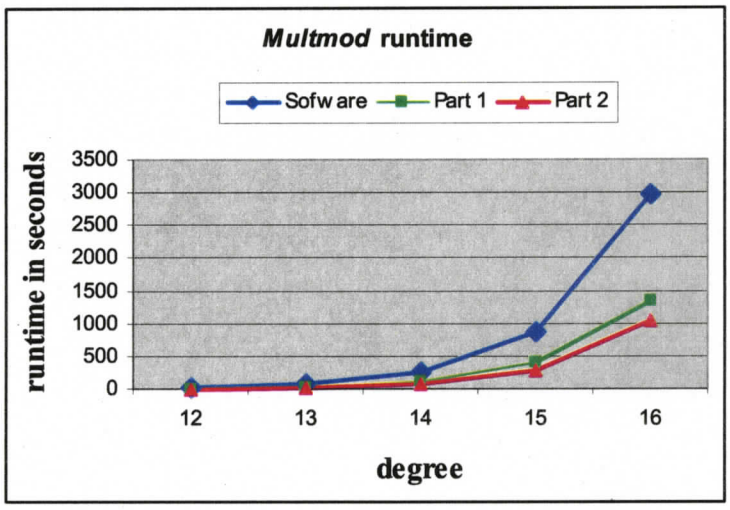


Figure 29: *Multmod* runtime graph

When comparing the *multmod* hardware implementation between the two platforms, it can be observed that the Co-design on the SoC platform executes the hardware faster than on the loosely-coupled platform. The data from Table 4 in section 5.2 is shown graphically in Figure 29, only for degrees 12 to 16. As mentioned before, the difference in speed between the FPGAs contributes to the additional acceleration of Part 2. We also have to consider that some modifications to the *multmod* code were necessary from Part 1 to Part 2. For example, functionality for the PCI-bus was omitted and the reading/writing of data is done in a single clock cycle for Part 2 by

loading the input/output registers. This simplified the code and several clock cycles for the FSM no longer needed execution.

Total running time

When comparing the implementations of Part 1, the PCI-based environment, and Part 2, the SoC environment, the total running time, including software, hardware and communication, shows that Part 1 executes faster than Part 2. This is most certainly due to the software portion, running on a PowerPC at 240MHz in Part 2, slowing the application overall.

Communication time

Communication between hardware and software demonstrates an extreme bottleneck for Part 1, due to the PCI bus. This issue is much better resolved in Part 2 where the software module is closely-coupled to the hardware module. For Part 1, the communication takes, on average, 90% of the total running time. For example, for degree 10, the communication running time corresponds to 90.8% of the total runtime (11.68 out of 12.8 seconds). Conversely, for Part 2, the communication between software and hardware takes, on average, approximately only 19.12% of the total running time.

The communication of hardware and software is high due to the number of data transferred to/from the two. The problem presented in this thesis could be considered a not ideal example when trying to minimize communication timing. In the multiplication of polynomials, there are many calls to multmod forcing the implementation to migrate from hardware and software excessive amount of times.

Running time of the software module in the Co-design

The software module in the Co-design was first executed on a Pentium at 1.8GHz for the original version. For Part 1, this module running time remains the same, since the same processor is used. For Part 2, the software instead executes on a PowerPC

running at 240MHz, which is the processor available in the SoC platform. Therefore, Part 1 executes the software module faster than Part 2, on average 80 times faster. For example, for degree 10, the software module is executed in 0.29 seconds, in contrast to Part 2 where the software module is executed at 38.7 seconds.

As a separate experiment, the entire application was run on the PowerPC with no hardware portion, in order to compare more directly the software speed between the two processors. This again showed that the PowerPC is on average 80 times slower than the Pentium. This confirmed the conjecture that the PowerPC performance indeed is the culprit in slowing down the total running time of the application. However, the issue must be based on more than the raw clock speed, since 240MHz is not 80 times slower than 1.8GHz.

6.2 Observations on Architectural Issues and Tools

One of the main obstacles encountered when moving from Part 1 to Part 2 of this Co-design is the fact that the PowerPC performance slows down the total running time. Not only the software is affected by the speed of the PowerPC, but also the communication between hardware and software, since the software is responsible for loading the input registers. Listed below are some possible reasons which we deduced based on the findings.

- **Raw clock speed**: The application on the PowerPC is running at a rate of 240MHz and on the Pentium at 1.8GHz. This should account for at least 7.5 times speedup for the Pentium.
- **Cache storage**: The size of the software module in the Co-design, after compiling the code with the ISE Xilinx tool for the PowerPC, is approximately 58Kb. Unfortunately, the available cache is only16Kb, forcing the PowerPC to communicate back and forth with its memory. With a much larger cache of 2 MB, the Pentium can instead load the whole software portion to cache.

- **Instruction set**: It is not clear what effect there may be in the reduced instruction set (RISC) of the PowerPC compared to the more complex instruction set (CISC) architecture of the Pentium
- **Memory type**: The Pentium memory access achieves greater bandwidth than the single-data-rate (SDRAM) in the PowerPC by transferring data on both the rising and falling edges of the clock signal. The richer interface with memory of the Pentium, called double-data-rate synchronous dynamic random access memory (DDR-SDRAM) [11], effectively nearly doubles the transfer rate without increasing the frequency of the front side bus.

Partitioning

For this project the partitioning between hardware and software was done based on the results of a profiler as described in chapter 3, such that the *multmod* method was chosen to be implemented in hardware. Although this method was accelerated by the hardware implementation, there is a large amount of interaction between hardware and software, sending input/output to/from the two portions millions of times. One approach to make the application faster would be to move more functionality to hardware, minimizing the traffic between hardware and software. It is possible that for this particular application, a partitioning where almost all the modules are in hardware might be better. However the Co-design paradigm remained the first goal.

CAD tools

Available CAD tools for the hardware and IDEs for the software were useful and adequate for this work. On the other hand, in order to evolve towards a really efficient Co-design solution in a reasonable amount of time, with less direct human effort of an experienced designer and with much more control over correctness, CAD tools need to evolve to integrate more fully the creation of interfaces between the hardware and software portions. In the migration from Part 1 to Part 2, a new interface needed to be designed, created and implemented manually in all details. Instead the hardware/software interface should allow system designers to move functions and services gracefully and efficiently between the hardware and software portions,

without requiring a major reengineering effort. It is an essential ingredient towards an effective exploration of possible solutions. Therefore, the main need in this area is to automate as much as possible the entire process of code generation, compilation, and implementation with the emphasis on "co-synthesis" as shown originally in Figure 3 in Chapter 2. This could be done, for example, by incorporating more formal specifications, both in functionality and communication, in system designs languages [17]. Also tools that support an efficient Co-design simulation are necessary. For this work the hardware module was simulated separately from the software module due to the restrictions of the simulators available.

The ideal architecture

From the work in this Co-design, some interesting ideas about the best features towards an ideal architectural platform can be considered.

1. Since the PowerPC slowed the application down, a more powerful processor would be ideal in the SoC environment, making it competitive with other platforms.
2. One needs the closely-coupled communication scheme from Part 2 together with the microprocessor speed from Part 1.
3. It would be desirable to place a processor with the same power as a Pentium on a Soc environment. However, this would be impractical due to the power consumption limitations for the overall embedding.
4. One approach to test the solution before a complete Co-design execution is to test each module, software, hardware and the communication between the two. Ideally it would be better to simulate all the modules together, but simulation tools that support a complete Co-design are not available yet.
5. The need for co-simulation, co-synthesis and interface design within a CAD platform to enhance productivity remains the ideal goal to enable Co-design to its full capability.

6.3 Future Work

At the end stages of this project it is apparent that more avenues for future work are still open. The following is a list of potential research projects that could provide further insights into the dynamics of both the designs and the implementations.

1. **All-in-hardware implementation**: In the current implementation, both the *multmod* and *add* functions were accelerated in hardware to exploit HW/SW Co-design concepts. A different approach would be to implement the system entirely in hardware using the FPGA.
2. **More portions of the code in hardware:** In the current implementation the portion implemented in hardware consists only of the multiplication of polynomials. To minimize the amount of times the hardware and software sends information to each other, it would be adequate to move more portions of the software to hadware.
3. **Re-implement with improved CAD tools**: When more complete CAD tools become available, it would be useful to re-develop this project by automating the majority of the design. This would then allow performance comparisons among implementations to be possible on a faster turnaround, thus providing further insights to various solutions.
4. **On-chip storage for primitive polynomials**: In the current implementation the primitive polynomials are transmitted by the software at the beginning of the multiplication cycle. A potentially better approach would be to store them permanently on the FPGA using local memory.
5. **Wider data words**: In the research of finding primitive and irreducible polynomials, even with hardware support, it is not feasible to find all irreducible polynomials of higher degrees (for example, degree 100). At the same time it would be interesting to explore certain classes of these degrees by allowing wider data words.
6. **Expansion of test cases**: Testing was limited due to restrictions to the software used for the SoC implementation. If a faster processor became available, testing of the higher degrees would provide additional data.

7. **High-Level language implementation**: While Page and Smart presented a Handel-C implementation in [15] of multiplication of polynomials over GF(3), the algorithm and representation used there are slightly different from those used in this project. In order to attribute the performance gain to either improvements in algorithm and representation or to VHDL implementation details, a direct reference implementation using a high level language such as Handel-C or SystemC is recommended.
8. **Expand implementation to GF(5) and GF(7)**: Until the conclusion of this thesis the implementation of the multiplication of polynomials of order 5 and 7 were not available. When these become available, it worthwhile to develop a codesign solution for performance comparison.

Bibliography

[1] "APEX 20KC Programmable Logic Device", *Data Sheet Version 2.2*. Altera Corporation, 2002.

[2] "APEX PCI Development Board Data Sheet", *Data Sheet Version 2.1.* Altera Corporation, 2002.

[3] "Quartus II PLD Design", *User Manual Version* 2. Altera Corporation, 2002.

[4] "LPM Quick Reference Guide", Altera Corporation, 2003.

[5] "PCI_mt64 MegaCore function reference design", *Functional Specification Version 1.2*. Altera Corporation, 2003.

[6] "AP1000 FPGA Development Board User Guide", *User Guide Manual Version 2*. AMIRIX Systems Inc., 2005.

[7] W. H. Wolf, *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann Publishers, California, 2001.

[8]] K. Cattell, F. Ruskey, J. Sawada, C. R. Miers and M. Serra, "Fast Algorithm to Generate Unlabeled Necklaces". *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, p.256-262. 2004.

[9] J. Hannula, "Is high-level design representation worthwhile?", M.Sc. dissertation. University of Victoria, Victoria, BC, Canada, 2004.

[10] K. Harrison, D. Page and N. Smart, "Software Implementation of Finite Fields of Characteristic Three, for use in Pairing Based Cryptosystems", *LMS Journal of Computation and Mathematics,* vol. 5, pp. 181–193, 2003.

[11] C. May, E. Silha, R. Simpson and H. Warren, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers Inc.1994.

[12] "ModelSim SE User Manual", Mentor Graphics Corp., 2005.

[13] J. Moris, D. Newport D., D. Bouldlin, R. Gonzalez and A. Wang, *The Computer Science and Engineering Handbook: Chapter 37 Reconfigurable Comupting.* CRC Press, Inc. USA, 2002.

[14] G. Noll, "Application specific eFPGAs for SoC platforms", *Proceedings of the 2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test.* April 2005.

[15] D. Page and N. P. Smart, "Hardware implementation of finite fields of characteristic three", *Proceedings of the CHES 2002,* p.529-539, 2002.

[16] M. Serra, W. Gardner,"An Object-Oriented Layered Approach to Interfaces for Hardware / Software Codesign of Embedded Systems.", *Thirty-First Annual Hawaii International Conference on System Sciences- 7, Software Technology.* p. 197-203, 1997.

[17] Sam Siewert, "Soc drawer: SoC design for hardware acceleration, Part 1 volume A". [Online] Available: http://www-128.ibm.com/developerworks/power/library/pa-soc8/.

[18] J. Staunstrup and W. Wolf, *Hardware/Software Co-design: Principles and Practice.* Kluwer Academic Publishers, 1997.

[19] "PowerSpanII User Manual", Tundra Semiconductor Corp., 2004

[20] W. H. Wolf, *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann Publishers, California, 2001.

[21] W. Bishop, "Configurable computing for mainstream software applications", Ph.D. dissertation. University of Waterloo, Waterloo, ON, Canada, 2003

[22] "Xilinx Standalone Board Support Package - EDK 8.2i", Xilinx Inc., 2006.

[23] "Xilinx LogiCore DataSheet - Decoder v7.1", Xilinx Inc., 2005.

Appendix A: IP cores

To remove the burden on the system designer of developing all the necessary components, the concept of Intellectual Properties (IP) Cores emerged. These are previously tested and optimized predefined libraries of complex functions and circuits used to simplify and speed the design of complex designs in FPGA. These circuits are essentially a block of logic or data that is used in making an FPGA. In ideal situations IP Cores should be completely portable allowing the designer to insert them in any vendor technology or design methodology. Some of the advantages of IP Cores are listed below:

- *Increase in productivity*. Pre-defined instructions allow the designer to save time and concentrate in other tasks of the design.
- *Increases in reliability*. IP cores have likely been used by numerous designers offering a better solution to a given problem compared to an initial solution developed by a single designer.
- *Simplified testing*. IP Core vendors usually provide extensive testbenches for their design. Designing good testbenches can be very time consuming and involves a significant amount of work. These testbenches can help the designer in testing other parts of the project.

Although the use of IP Cores can bring productivity enhancements to the developer, it also has its disadvantages:

- *Overhead introduced.* The code provided may contain more functionality than needed, leading to overhead in the final design (area and energy) and to increased design effort for the surrounding logic.
- *Licensing & 3rd party dependencies*. IP Cores are usually black-boxes. Due to encryption and license conditions, "bugs" can only be fixed by the vendor. Changes in design can require different IP-core licensing.

In conclusion, IP Cores have benefits, but they don't come without a price. A precise evaluation is required for each project to determine if the benefits will exceed the risks.

Appendix B: gf3.c – SoC environment

```
/*
Author: Beatriz Iaderoza
        based on gf3.c by Gilber Lee

Last update: October 21, 2006

Platform: IBM PowerPC405 (Board located at University of New Brunswick)

This program calculates the irreducible/primitive polynomials of Galois Fields.
The degree of the calculation is entered by the user as an input.
*/

#include "xparameters.h"
#include "ns16550.h"
#include <xexception_l.h>
#include <xintc.h>
#include <xintc_l.h>
#include <GF3_MULTIPLIER.h>
#include <GF3_ADDER.h>
#include <xtime_l.h>

/*------definitions------*/
typedef unsigned long long ullong;
typedef struct{
  ullong top;
  ullong bot;
} Poly_GF3;

#define MAX 64

/*---methods---*/
void MultIntHandler(void *);
void AddIntHandler(void *);
void resetGF3();
void itoa(int, char *);
void reverse(char *);
int strlength(char *s);
int axtoi(char *hex);

/*---GF3 methods---*/
void PrintIt();
Poly_GF3 minpoly(Poly_GF3 necklace);
Poly_GF3 powmod(Poly_GF3 a, Poly_GF3 power);
Poly_GF3 multmod(Poly_GF3 a, Poly_GF3 b);
Poly_GF3 gcd(Poly_GF3 a, Poly_GF3 b);
Poly_GF3 mod(Poly_GF3 a, Poly_GF3 b);
Poly_GF3 subtract(Poly_GF3 a, Poly_GF3 b);
int cmp(Poly_GF3 a, Poly_GF3 b);
Poly_GF3 add(Poly_GF3 a, Poly_GF3 b);
int density(Poly_GF3 a);
int bitcount(ullong x);
```

```
void Gen(int t, int p);
void ulltoa(ullong n, char *s);

/* Each polynomial is expressed in two words.  00 = 0, 01 = 1, 11 = 2 */

/* A list of primitive polynomials
   poly_table[i] contains what x^i is equivalent to.
   E.g. given primitive polynomial x^3 + 2x + 1:
        x^3 ~= x + 2 --> {01,11} = {1,3}
*/

Poly_GF3 poly_table[MAX+1] = {
  /*  0 */ { 0 , 1 },
  /*  1 */ { 0 , 1 },
  /*  2 */ { 0 , 3 },
  /*  3 */ { 1 , 3 },
  /*  4 */ { 0 , 3 },
  /*  5 */ { 1 , 3 },
  /*  6 */ { 0 , 3 },
  /*  7 */ { 1 , 5 },
  /*  8 */ { 0 , 9 },
  /*  9 */ { 1 ,15 },
  /* 10 */ { 0 ,11 },
  /* 11 */ { 1 , 5 },
  /* 12 */ { 2 ,31 },
  /* 13 */ { 1 , 3 },
  /* 14 */ { 0 , 3 },
  /* 15 */ { 1 , 5 },
  /* 16 */ {24 ,27 },
  /* 17 */ { 1,  3 },
  /* 18 */ {32, 39 },
  /* 19 */ { 1,  5 },
  /* 20 */ {34, 35 },
  /* 21 */ { 1, 15 },
  /* 22 */ { 2, 15 },
  /* 23 */ { 1,  9 },
  /* 24 */ { 8, 27 },
  /* 25 */ { 1,  9 },
  /* 26 */ { 0, 11 },
  /* 27 */ {49, 59 },
  /* 28 */ {48, 51 },
  /* 29 */ {21, 21 },
  /* 30 */ { 0,  3 },
  /* 31 */ { 3, 15 },
  /* 32 */ {24, 27 }
#if (MAX > 32)
 ,
  /* 33 */ {33, 37 },
  /* 34 */ { 0, 11 },
  /* 35 */ { 1,  5 },
  /* 36 */ {48, 57 },
  /* 37 */ { 9, 15 },
  /* 38 */ { 6, 15 },
  /* 39 */ {33, 45 },
  /* 40 */ { 0,  3 },
  /* 41 */ { 1,  3 },
```

```
 /* 42 */ {62, 63 },
 /* 43 */ {11, 11 },
 /* 44 */ { 0,  9 },
 /* 45 */ { 5, 13 },
 /* 46 */ {18, 23 },
 /* 47 */ {29, 29 },
 /* 48 */ {32, 61 },
 /* 49 */ { 5, 13 },
 /* 50 */ { 8, 31 },
 /* 51 */ { 1,  3 },
 /* 52 */ {32, 51 },
 /* 53 */ { 1, 29 },
 /* 54 */ { 2,  3 },
 /* 55 */ { 1, 15 },
 /* 56 */ { 8,  9 },
 /* 57 */ {35, 47 },
 /* 58 */ {18, 23 },
 /* 59 */ { 9, 15 },
 /* 60 */ {34, 35 },
 /* 61 */ { 9, 15 },
 /* 62 */ {18, 23 },
 /* 63 */ { 1, 67108865 },
 /* 64 */ { 8,  9 }
#endif
};

/*------variables------*/
static int interrupt = 0; /*set to 1 when hardware is done*/
static int addInterrupt = 0; /*set to 1 when hardware is done*/
int N;
ullong ONE = (ullong)1;
ullong TWO = (ullong)2;
ullong multcount = 0, addcount = 0;
ullong addCycles, multCycles;

int N,D;
ullong maxNumber;
ullong ir_count = 0, pr_count = 0;
int printed = 0;
Poly_GF3 threeN_1;
int a[MAX+1] = {0};
XTime mult_time = 0, add_time = 0, multTransfer_time = 0, addTransfer_time = 0;
/*-----------------------*/

int main(void)
{
        int cmd, i;
        char cmd_buffer[80];
        char output[80];
        int cmd_degree;
        char* temp;
        XTime total_time, start_time, end_time, sw_time;

        /* Initialize the Console */
        NS16550_t console = (NS16550_t) (XPAR_OPB_UART16550_I_BASEADDR + 0x1000);
        int clock_divisor = CFG_NS16550_CLK / 16 / CFG_BAUD;
```

```
NS16550_init(console, clock_divisor);

//ask the user for the degree until 0 is entered (value must be entered in HEX)
NS16550_puts(console, "Enter degree: ");
N = axtoi(NS16550_gets(console, cmd_buffer, 79));

NS16550_puts(console, "\r\nCurrent degree is: ");
NS16550_puts(console, cmd_buffer);

/* Welcome Message */
NS16550_puts(console, "\r\n---------- GF3 Program ----------\r\n");
/* Reset the Board and enable Interrupts */
GF3_MULTIPLIER_mReset(XPAR_GF3_MULTIPLIER_0_BASEADDR);
GF3_ADDER_mReset(XPAR_GF3_ADDER_0_BASEADDR);

/*-------GF3 main------*/
XTime_GetTime(&start_time);

D = N+1;
for(i = 0; i < N; i++)
        threeN_1.top |= (ONE<<i);
threeN_1.bot = threeN_1.top;
Gen(1,1);

XTime_GetTime(&end_time);

total_time = end_time - start_time;
sw_time = total_time - mult_time - add_time;

//display info
NS16550_puts(console, "Number of irreducible polynomials: ");
itoa(ir_count, output);      NS16550_puts(console, output);
NS16550_puts(console, "\r\nNumber of primitive polynomials: ");
itoa(pr_count, output);      NS16550_puts(console,output);
NS16550_puts(console, "\r\nNumber of multiplications: ");
itoa(multcount, output); NS16550_puts(console,output);
NS16550_puts(console, "\r\nNumber of addition: ");
itoa(addcount, output); NS16550_puts(console,output);
NS16550_puts(console, "\r\nNumber of clock cycles on the FPGA: ");
ulltoa(multCycles, output); NS16550_puts(console,output);

NS16550_puts(console,"\r\n Time of total execution (HW+Transfer+SW):  ");
ulltoa(total_time, output); NS16550_puts(console, output);
NS16550_puts(console,"\r\n Time for HW execution only (-SW-Transfer):  ");
ulltoa(mult_time, output); NS16550_puts(console, output);
NS16550_puts(console,"\r\n Time for SW execution only (-HW-Transfer):  ");
ulltoa(sw_time, output); NS16550_puts(console,output);
NS16550_puts(console,"\r\n Time for TRANSFER execution only (-HW-HW):  ");
ulltoa(multTransfer_time, output); NS16550_puts(console,output);

NS16550_puts(console,"\r\n ****Add functionality**** ");
NS16550_puts(console,"\r\n (ADD function) Clock cycles for ADD:  ");
ulltoa(addCycles, output); NS16550_puts(console,output);
NS16550_puts(console,"\r\n (ADD function) Time for TRANSFER execution only (-HW-
HW):  ");
ulltoa(addTransfer_time, output); NS16550_puts(console,output);
```

```
	return 0;
}

void resetGF3(){
	/* Reset Board */
	GF3_MULTIPLIER_mReset(XPAR_GF3_MULTIPLIER_0_BASEADDR);
	GF3_ADDER_mReset(XPAR_GF3_ADDER_0_BASEADDR);

	/* Initialize exception-handler table */
	XExc_Init();

	/* Register the default interrupt handler with the non-critical interrupt pin */
	XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
		(XExceptionHandler)XIntc_DeviceInterruptHandler, (void *)0);

	XIntc_RegisterHandler(XPAR_OPB_INTC_I_BASEADDR,
		XPAR_OPB_INTC_I_GF3_MULTIPLIER_0_IP2INTC_IRPT_INTR,
(XExceptionHandler)MultIntHandler, (void *)0x2a000000);
	XIntc_RegisterHandler(XPAR_OPB_INTC_I_BASEADDR,
		XPAR_OPB_INTC_I_GF3_ADDER_0_IP2INTC_IRPT_INTR,
(XExceptionHandler)AddIntHandler, (void *)0x2b000000);

	/* Enable the OPB_INTC to generate interrupts*/
	XIntc_mMasterEnable(XPAR_OPB_INTC_I_BASEADDR);

	/* Enable JE interrupts in the interrupt controller */
	XIntc_mEnableIntr(XPAR_OPB_INTC_I_BASEADDR,
		XPAR_GF3_MULTIPLIER_0_IP2INTC_IRPT_MASK);
	XIntc_mEnableIntr(XPAR_OPB_INTC_I_BASEADDR,
		XPAR_GF3_ADDER_0_IP2INTC_IRPT_MASK);

	/* Enable PPC non-critical interrupts */
	XExc_mEnableExceptions(XEXC_NON_CRITICAL);

	GF3_MULTIPLIER_EnableInterrupt((void *)XPAR_GF3_MULTIPLIER_0_BASEADDR);
	GF3_ADDER_EnableInterrupt((void *)XPAR_GF3_ADDER_0_BASEADDR);
}

void AddIntHandler(void * baseaddr_p)
{
	Xuint32 baseaddr;
	Xuint32 IntrStatus;

	baseaddr = (Xuint32) baseaddr_p;
	addInterrupt++;

	/*
	* Get status from IP Interrupt Status Register.
	*/
	IntrStatus = GF3_ADDER_mReadReg(baseaddr, GF3_ADDER_INTR_ISR_OFFSET);

	/*  xil_printf("IP Interrupt! Status register (ISR) value : 0x%08x \n\r", IntrStatus);
	*/
	/*
	* Clear IP interrupts by toggle write back to IP ISR register.
```

```
        */

        GF3_ADDER_mWriteReg(baseaddr, GF3_ADDER_INTR_ISR_OFFSET, IntrStatus);
}

void MultIntHandler(void * baseaddr_p)
{
        Xuint32 baseaddr;
        Xuint32 IntrStatus;

        baseaddr = (Xuint32) baseaddr_p;
        interrupt++;

        /*
        * Get status from IP Interrupt Status Register.
        */
        IntrStatus = GF3_MULTIPLIER_mReadReg(baseaddr,
GF3_MULTIPLIER_INTR_ISR_OFFSET);

        /*  xil_printf("IP Interrupt! Status register (ISR) value : 0x%08x \n\r", IntrStatus);
        */
        /*
        * Clear IP interrupts by toggle write back to IP ISR register.
        */

        GF3_MULTIPLIER_mWriteReg(baseaddr, GF3_MULTIPLIER_INTR_ISR_OFFSET,
IntrStatus);
}

void ulltoa(ullong n, char *s)
{
        ullong i, sign;
        if ((sign = n) < 0)
                n = -n;
        i = 0;
        do {
                s[i++] = n % 10 + '0';
        } while ((n /= 10) > 0);

        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}

void itoa(int n, char *s)
{
        int i, sign;
        if ((sign = n) < 0)
                n = -n;
        i = 0;
        do {
                s[i++] = n % 10 + '0';
        } while ((n /= 10) > 0);
        if (sign < 0)
                s[i++] = '-';
```

```
        s[i] = '\0';
        reverse(s);
}


void reverse(char *s)
{
        int c, i, j;
        for (i = 0, j = strlength(s) - 1; i < j; i++, j--) {
                c = s[i];
                s[i] = s[j];
                s[j] = c;
        }
}

int strlength(char *s)
{
  int i, len;
  i = len = 0;
  while (s[i++] != '\0')
    len++;
  return len;
}

int axtoi(char *hex)
{
 int n = 0;         // position in string
 int m = 0;         // position in digit[] to shift
 int count;         // loop index
 int intval = 0;  // integer value of hex string
 int digit[9];      // hold values to convert
 while (n < 8) {
   if (hex[n]=='\0')
     break;
   if (hex[n] >= '0' && hex[n] <= '9' )          //if 0 to 9
     digit[n] = hex[n] & 0x0f;            //convert to int
   else if (hex[n] >='a' && hex[n] <= 'f') //if a to f
     digit[n] = (hex[n] & 0x0f) + 9;      //convert to int
   else if (hex[n] >='A' && hex[n] <= 'F') //if A to F
     digit[n] = (hex[n] & 0x0f) + 9;      //convert to int
   else return 0; /* return zero if we get a bad digit */
  n++;
 }
 count = n;
 m = n - 1;
 n = 0;
 while(n < count) {
   // digit[n] is value of hex digit at position n
   // (m << 2) is the number of positions to shift
   // OR the bits into return value
   intval = intval | (digit[n] << (m << 2));
   m--;   // adjust the position to set
   n++;   // next digit to process
 }
 return (intval);
}
```

```
/* ----------GF3 Functionality----------*/

/* Counting bits in O(number of bits) */
int bitcount(ullong x){
 int c = 0;
 while(x){
  c++;
  x &= x-1;
 }
 return c;
}

/* Density - (By our encoding, we count number of bits in the bottom word) */
int density(Poly_GF3 a){
 return bitcount(a.bot);
}

/* Returns a + b */
Poly_GF3 add(Poly_GF3 a, Poly_GF3 b){
        Poly_GF3 sw_result, hw_result;
        Xuint64 value, value1, start_key;
        Xuint64 *data;
        char output[80];
        XTime hw_start, hw_end, transfer_start, transfer_end;

        addcount ++;
        XTime_GetTime(&hw_start);

        GF3_ADDER_mReset(XPAR_GF3_ADDER_0_BASEADDR);
        start_key.Lower = 0xaaaa;
        start_key.Upper = 0;
        XTime_GetTime(&transfer_start);

        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG1
_OFFSET, (long)(a.top >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG1
_OFFSET+0x4, (long)(a.top & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG2
_OFFSET, (long)(a.bot >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG2
_OFFSET+0x4, (long)(a.bot & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG3
_OFFSET, (long)(b.top >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG3
_OFFSET+0x4, (long)(b.top & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG4
_OFFSET, (long)(b.bot >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG4
_OFFSET+0x4, (long)(b.bot & 0xffffffff));

        XTime_GetTime(&transfer_end);
        addTransfer_time += (transfer_end - transfer_start);

        // Write register0 -> inputrdy, it will activate the software
        GF3_ADDER_WriteSlaveReg0(XPAR_GF3_ADDER_0_BASEADDR, &start_key);
```

```
        GF3_ADDER_ReadSlaveReg0((void *)XPAR_GF3_ADDER_0_BASEADDR, &value);
        while (value.Lower == 0xaaaa);

        addCycles +=  value.Lower;

        hw_result.top = 0;
        hw_result.bot = 0;

        //Get result from multmod
        XTime_GetTime(&transfer_start);
        value.Upper = XIo_In32((void
*)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG5_OFFSET);
        value.Lower = XIo_In32((void
*)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG5_OFFSET+0x4);
        value1.Upper = XIo_In32((void
*)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG6_OFFSET);
        value1.Lower = XIo_In32((void
*)XPAR_GF3_ADDER_0_BASEADDR+GF3_ADDER_SLAVE_REG6_OFFSET+0x4);

        XTime_GetTime(&transfer_end);
        addTransfer_time += (transfer_end - transfer_start);

        hw_result.top = value.Upper;
        hw_result.top = (hw_result.top << 32) | value.Lower;
        hw_result.bot = value1.Upper;
        hw_result.bot = (hw_result.bot << 32) | value1.Lower;

        XTime_GetTime(&hw_end);
        add_time += (hw_end - hw_start);

        /* Software add */
        /*ullong t1 = a.bot^b.top;
        ullong t2 = a.top^b.bot;
        sw_result.top = t1&t2;
        sw_result.bot = (a.top^t1)|(b.top^t2);
        return sw_multmod*/

        return hw_result;
}

/* Returns -1 if a < b, 0 if a == b, +1 if a > b */
int cmp(Poly_GF3 a, Poly_GF3 b){
 int i, x, y;
 for(i = N-1; i >= 0; i--){
  x = (a.top & (ONE<<i)) ? 2 : (a.bot & (ONE<<i)) ? 1 : 0;
  y = (b.top & (ONE<<i)) ? 2 : (b.bot & (ONE<<i)) ? 1 : 0;
  if(x < y) return -1;
  if(x > y) return 1;
 }
 return 0;
}

/* Returns a - b, (assuming a > b and a,b are ternary numbers */
Poly_GF3 subtract(Poly_GF3 a, Poly_GF3 b){
 int i, borrow = 0, x, y;
```

```
 Poly_GF3 r = {0,0};
 for(i = 0; i < N; i++){
  x = ((a.top & (ONE<<i)) ? 2 : (a.bot & (ONE<<i)) ? 1 : 0) - borrow;
  y =  (b.top & (ONE<<i)) ? 2 : (b.bot & (ONE<<i)) ? 1 : 0;
  if(x < y){
   borrow = 1;
   x += 3;
  } else {
   borrow = 0;
  }
  switch(x-y){
  case 1: r.bot |= (ONE<<i); break;
  case 2: r.bot |= (ONE<<i); r.top |= (ONE<<i); break;
  }
 }
 return r;
}

/* Returns a % b (a and b treated as ternary numbers) */
Poly_GF3 mod(Poly_GF3 a, Poly_GF3 b){
 Poly_GF3 r = {0,0};
 int i;
 for(i = N-1; i >= 0; i--){
  r.top <<= 1; r.bot <<= 1;
  if(a.top & (ONE<<i)) r.top++;
  if(a.bot & (ONE<<i)) r.bot++;
  while(cmp(b,r) <= 0){
   r = subtract(r, b);
  }
 }
 return r;
}

/* Returns the gcd of a and b (a and b treated as ternary numbers */
Poly_GF3 gcd(Poly_GF3 a, Poly_GF3 b){
 if(!b.top && !b.bot) return a;
 return gcd(b, mod(a, b));
}

/* Returns (a * b) mod p (where p = poly_table[N]) */
Poly_GF3 multmod(Poly_GF3 a, Poly_GF3 b){
        Poly_GF3 hw_result = {0,0};
        ullong temp;
        Xuint64 value, value1, start_key;
        Xuint64 *data;
        Poly_GF3 t = a, sw_result = {0,0};
        ullong top_bit = ONE << (N-1);
        int i, z=0;
        char output[80];
        XTime hw_start, hw_end, transfer_start, transfer_end;

        multcount ++;
        XTime_GetTime(&hw_start);//start multCycles

        /*************HW multmod*************************/
        GF3_MULTIPLIER_mReset(XPAR_GF3_MULTIPLIER_0_BASEADDR);
```

```
        start_key.Lower = 0xaaaa;
        start_key.Upper = 0;

        XTime_GetTime(&transfer_start);//start multCycles

        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG1_OFFSET, (long)z);
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG1_OFFSET+0x4, (long)N);
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG2_OFFSET, (long)((poly_table[N].top) >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG2_OFFSET+0x4, (long)(poly_table[N].top & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG3_OFFSET, (long)(poly_table[N].bot >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG3_OFFSET+0x4, (long)(poly_table[N].bot & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG4_OFFSET, (long)(b.top >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG4_OFFSET+0x4, (long)(b.top & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG5_OFFSET, (long)(b.bot >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG5_OFFSET+0x4, (long)(b.bot & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG6_OFFSET, (long)(t.top >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG6_OFFSET+0x4, (long)(t.top & 0xffffffff));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG7_OFFSET, (long)(t.bot >> 32));
        XIo_Out32((Xuint32)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SL
AVE_REG7_OFFSET+0x4, (long)(t.bot & 0xffffffff));

        XTime_GetTime(&transfer_end);//start multCycles
        multTransfer_time += (transfer_end - transfer_start);

        // Write register0 -> inputrdy, it will activate the software
        GF3_MULTIPLIER_WriteSlaveReg0(XPAR_GF3_MULTIPLIER_0_BASEADDR,
&start_key);

        GF3_MULTIPLIER_ReadSlaveReg0((void *)XPAR_GF3_MULTIPLIER_0_BASEADDR,
&value);
        while (value.Lower == 0xaaaa) {
                GF3_MULTIPLIER_ReadSlaveReg0((void
*)XPAR_GF3_MULTIPLIER_0_BASEADDR, &value);
        }

        //add multCycles to calculate clock cycles
        multCycles +=  value.Lower;

        hw_result.top = 0;
        hw_result.bot = 0;
        XTime_GetTime(&transfer_start);

        //Get result from multmod
```

```
	//GF3_MULTIPLIER_ReadSlaveReg8((void *)XPAR_GF3_MULTIPLIER_0_BASEADDR,
&value); //result.top
	value.Upper = XIo_In32((void
*)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SLAVE_REG8_OFFSET);
	value.Lower = XIo_In32((void
*)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SLAVE_REG8_OFFSET+0x4)
;
	value1.Upper = XIo_In32((void
*)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SLAVE_REG9_OFFSET);
	value1.Lower = XIo_In32((void
*)XPAR_GF3_MULTIPLIER_0_BASEADDR+GF3_MULTIPLIER_SLAVE_REG9_OFFSET+0x4)
;

	XTime_GetTime(&transfer_end);
	multTransfer_time += (transfer_end - transfer_start);

	hw_result.top = value.Upper;
	hw_result.top = (hw_result.top << 32) | value.Lower;
	hw_result.bot = value1.Upper;
	hw_result.bot = (hw_result.bot << 32) | value1.Lower;

	XTime_GetTime(&hw_end); //start multCycles
	mult_time += (hw_end - hw_start);

	/**************SW multmod*************************/
	/*for(i = 0; i < N; i++){
		if(b.bot & ONE) sw_result = add(sw_result, t);
		if(b.top & ONE) sw_result = add(sw_result, t);
		if(t.top & top_bit){
			t.top = (t.top & ~top_bit) << 1;
			t.bot = (t.bot & ~top_bit) << 1;
			t = add(t, poly_table[N]);
			t = add(t, poly_table[N]);
		} else if(t.bot & top_bit){
			t.top <<= 1;
			t.bot = (t.bot & ~top_bit) << 1;
			t = add(t, poly_table[N]);
		} else {
			t.top <<= 1;
			t.bot <<= 1;
		}
		b.top >>= 1;
		b.bot >>= 1;
	}
	return sw_result;
	*/

	return hw_result;
}

/* Returns (a^power) mod p (where p = poly_table[N]) */
Poly_GF3 powmod(Poly_GF3 a, Poly_GF3 power){
 Poly_GF3 t = a, result = {0,ONE};
 while(power.top || power.bot){
	 if(power.bot & ONE) result = multmod(result, t);
	 if(power.top & ONE) result = multmod(result, t);
```

```
    t = multmod(multmod(t, t), t);
    power.top >>= 1;
    power.bot >>= 1;
  }
  return result;
}

/* Calculates the minimum polynomial, given a necklace */
Poly_GF3 minpoly(Poly_GF3 necklace){
  Poly_GF3 root = {0,TWO}, result = {0,0};
  Poly_GF3 f[MAX]; /* f[i] contains the polynomials listing the
                      coefficient of x^i in terms of the primitive root */
  int i, j;

  f[0].top = 0;
  f[0].bot = ONE;
  for(i = 1; i < N; i++)
    f[i].top = f[i].bot = 0;

  root = powmod(root, necklace);

  for(i = 0; i < N; i++){
    if(i) root = multmod(multmod(root, root), root);
    for(j = N-1; j >= 1; j--)
      f[j] = add(f[j-1], multmod(f[j], root));
    f[0] = multmod(f[0], root);
  }

  /* Because we compute (x+a)(x+a^2)...(x+a^n) instead of (x-a)(x-a^2)...(x-a^n)
     we now have to inverse all odd positions */
  for(i = N-1; i >= 0; i--){
    if(f[i].top == ONE){
      if(i % 2 == N % 2) result.top |= ONE<<i;
      result.bot |= ONE<<i;
    } else if(f[i].bot == ONE){
      if(i % 2 != N % 2) result.top |= ONE<<i;
      result.bot |= ONE<<i;
    }
  }
  return result;
}


void PrintIt(){
  int i;
  Poly_GF3 necklace = {0,0};
  Poly_GF3 gcdResult;
  Poly_GF3 mpoly;

  for(i = 1; i <= N; i++){
    necklace.top <<= 1;
    necklace.bot <<= 1;
    if(a[i]) necklace.bot++;
    if(a[i] == 2) necklace.top++;
  }
```

```
  mpoly = minpoly(necklace);
  if(density(mpoly)+1 > D) return;

  ir_count++;
  gcdResult = gcd(necklace, threeN_1);
  if(gcdResult.top == 0 && gcdResult.bot == ONE){
   ++pr_count;
   ++printed;
  }
  if(maxNumber && printed >= maxNumber) exit(1);

}

void Gen(int t, int p){
  int j;
  if(t > N){
   if(p == N) PrintIt();
  } else {
   a[t] = a[t-p];
   Gen(t+1,p);
   for(j=a[t-p]+1; j<=2; j++) {
    a[t] = j;
    Gen(t+1,t);
   }
  }
}
```

Appendix C: user_logic.vhd – SoC environment

```
------------------------------------------------------------------------------
-- user_logic.vhd - entity/architecture pair
-- by Beatriz Iaderoza
        Based on user_logic.vhd by Adam Baker
------------------------------------------------------------------------------
--
-- ***************************************************************************
-- ** Copyright (c) 1995-2006 Xilinx, Inc.  All rights reserved.          **
-- **                                          **-- ** Xilinx, Inc.
**
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"       **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
**
-- ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,      **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,      **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION         **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
**
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
**
-- ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY           **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE             **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
**
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF     **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
**
-- ** FOR A PARTICULAR PURPOSE.                                          **
-- **                                          **
-- ***************************************************************************
--
------------------------------------------------------------------------------
-- Filename:        user_logic.vhd
-- Version:         1.00.a
-- Description:     User logic.
-- Date:            Sat Sep 30 13:50:27 2006 (by Create and Import Peripheral Wizard)
-- VHDL Standard:   VHDL'93
------------------------------------------------------------------------------
-- Naming Conventions:
--   active low signals:                  "*_n"
--   clock signals:                       "clk", "clk_div#", "clk_#x"
--   reset signals:                       "rst", "rst_n"
--   generics:                            "C_*"
--   user defined types:                  "*_TYPE"
--   state machine next state:            "*_ns"
--   state machine current state:         "*_cs"
--   combinatorial signals:               "*_com"
--   pipelined or register delay signals:  "*_d#"
--   counter signals:                     "*cnt*"
--   clock enable signals:                "*_ce"
--   internal version of output port:     "*_i"
--   device pins:                         "*_pin"
```

```
--  ports:                          "- Names begin with Uppercase"
--  processes:                       "*_PROCESS"
--  component instantiations:           "<ENTITY_>I_<#|FUNC>"
------------------------------------------------------------------------------

-- DO NOT EDIT BELOW THIS LINE ----------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
-- DO NOT EDIT ABOVE THIS LINE ----------------

--USER libraries added here

------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
-- Definition of Generics:
--   C_DWIDTH                -- User logic data bus width
--   C_NUM_CE                -- User logic chip enable bus width
--   C_IP_INTR_NUM             -- User logic number of interrupt event
--
-- Definition of Ports:
--   Bus2IP_Clk             -- Bus to IP clock
--   Bus2IP_Reset            -- Bus to IP reset
--   IP2Bus_IntrEvent          -- IP to Bus interrupt event
--   Bus2IP_Data            -- Bus to IP data bus for user logic
--   Bus2IP_BE              -- Bus to IP byte enables for user logic
--   Bus2IP_RdCE              -- Bus to IP read chip enable for user logic
--   Bus2IP_WrCE              -- Bus to IP write chip enable for user logic
--   Bus2IP_RdReq             -- Bus to IP read request
--   Bus2IP_WrReq             -- Bus to IP write request
--   IP2Bus_Data            -- IP to Bus data bus for user logic
--   IP2Bus_Retry            -- IP to Bus retry response
--   IP2Bus_Error           -- IP to Bus error response
--   IP2Bus_ToutSup           -- IP to Bus timeout suppress
--   IP2Bus_Busy            -- IP to Bus busy response
--   IP2Bus_RdAck             -- IP to Bus read transfer acknowledgement
--   IP2Bus_WrAck             -- IP to Bus write transfer acknowledgement
------------------------------------------------------------------------------

entity user_logic is
 generic
 (
  -- ADD USER GENERICS BELOW THIS LINE ---------------
  --USER generics added here
  -- ADD USER GENERICS ABOVE THIS LINE ---------------

  -- DO NOT EDIT BELOW THIS LINE ----------------------
  -- Bus protocol parameters, do not add to or delete
  C_DWIDTH                : integer          := 64;
  C_NUM_CE                : integer          := 10;
  C_IP_INTR_NUM             : integer          := 1
```

```
  -- DO NOT EDIT ABOVE THIS LINE ----------------------
 );
 port
 (
  -- ADD USER PORTS BELOW THIS LINE ------------------
  --USER ports added here
  -- ADD USER PORTS ABOVE THIS LINE ------------------

  -- DO NOT EDIT BELOW THIS LINE ----------------------
  -- Bus protocol ports, do not add to or delete
  Bus2IP_Clk                : in  std_logic;
  Bus2IP_Reset              : in  std_logic;
  IP2Bus_IntrEvent          : out std_logic_vector(0 to C_IP_INTR_NUM-1);
  Bus2IP_Data               : in  std_logic_vector(0 to C_DWIDTH-1);
  Bus2IP_BE                 : in  std_logic_vector(0 to C_DWIDTH/8-1);
  Bus2IP_RdCE               : in  std_logic_vector(0 to C_NUM_CE-1);
  Bus2IP_WrCE               : in  std_logic_vector(0 to C_NUM_CE-1);
  Bus2IP_RdReq              : in  std_logic;
  Bus2IP_WrReq              : in  std_logic;
  IP2Bus_Data               : out std_logic_vector(0 to C_DWIDTH-1);
  IP2Bus_Retry              : out std_logic;
  IP2Bus_Error              : out std_logic;
  IP2Bus_ToutSup            : out std_logic;
  IP2Bus_Busy               : out std_logic;
  IP2Bus_RdAck              : out std_logic;
  IP2Bus_WrAck              : out std_logic
  -- DO NOT EDIT ABOVE THIS LINE ----------------------
 );
end entity user_logic;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is

  --USER signal declarations added here, as needed for user logic
--top level component from Gf3
component mult_ctr_unit
        port (
                clk                                     : IN  STD_LOGIC;
                reset                   : IN  STD_LOGIC;
                start                                   :in std_logic;--input is read to be loaded
                slv_reg1                                :in std_logic_vector(63 downto 0);--
degree N
                slv_reg2                                :in std_logic_vector(63 downto 0);--
poly_table top
                slv_reg3                                :in std_logic_vector(63 downto 0);--
poly_table bot
                slv_reg4                                :in std_logic_vector(63 downto 0);--b
top
                slv_reg5                                :in std_logic_vector(63 downto 0);--b
bot
                slv_reg6                                :in std_logic_vector(63 downto 0);--a
top
```

```
                slv_reg7                                    :in std_logic_vector(63 downto 0);--a
bot
                slv_reg8                                    :out std_logic_vector(63 downto 0);--
result top register
                slv_reg9                                    :out std_logic_vector(63 downto 0);--
result bottom register
                interrupt                           :out std_logic
                );
end component;
 ------------------------------------------
 -- Signals for user logic slave model s/w accessible register example
 ------------------------------------------
 signal slv_reg0              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg1              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg2              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg3              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg4              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg5              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg6              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg7              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg8              : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_reg9              : std_logic_vector(0 to C_DWIDTH-1);
 signal m_slv_reg8              : std_logic_vector(0 to C_DWIDTH-1);
 signal m_slv_reg9              : std_logic_vector(0 to C_DWIDTH-1);
 signal start                                                                    :
std_logic_vector(0 to 63);
 signal slv_reg_write_select       : std_logic_vector(0 to 9);
 signal slv_reg_read_select        : std_logic_vector(0 to 9);
 signal slv_ip2bus_data          : std_logic_vector(0 to C_DWIDTH-1);
 signal slv_read_ack            : std_logic;
 signal slv_write_ack           : std_logic;
 signal gf3_state                                                : std_logic_vector(0 to 1)  :=
"00";
 signal gf3_reset                                                : std_logic;
 signal gf3_start                                                : std_logic;
 signal gf3_interrupt                                            : std_logic;
 ------------------------------------------
 -- Signals for user logic interrupt example
 ------------------------------------------
 signal interrupt             : std_logic_vector(0 to C_IP_INTR_NUM-1);

 -- timer for counting clock cycles
        signal timer                                : std_logic_vector(0 to C_DWIDTH-1) ;
        signal timer_run                    : std_logic  := '0';
        signal timer_reset                  : std_logic;
begin

 --USER logic implementation added here

        MULT_INST:  mult_ctr_unit
                port map (
                        clk => Bus2IP_clk,
                        reset => gf3_reset,
                        start => gf3_start,
                        slv_reg1 => slv_reg1,
                        slv_reg2 => slv_reg2,
```

```
                slv_reg3 => slv_reg3,
                slv_reg4 => slv_reg4,
                slv_reg5 => slv_reg5,
                slv_reg6 => slv_reg6,
                slv_reg7 => slv_reg7,
                slv_reg8 => m_slv_reg8,
                slv_reg9 => m_slv_reg9,
                interrupt => gf3_interrupt
        );
--State Machine to operate gf3 Core
        ENABLE_GF3 : process (Bus2IP_clk, gf3_state, gf3_interrupt, timer, timer_run) is
        begin
                if (Bus2IP_clk'event AND Bus2IP_clk = '1') then
                        if (timer_run = '1') then
                                timer <= timer + '1';

                        end if;
                        case (gf3_state) is
                                when "00" =>
                                        timer <= (others => '0');
                                        if (slv_reg0 = X"0000AAAA") then        -- Starting
the hardware
                                                timer_run <= '1'; -- start timer
                                                gf3_state <= "01";
                                                gf3_reset <= '1';
                                                gf3_start <= '1';
                                        else
                                                gf3_state <= "00";
                                                gf3_reset <= '0';
                                                gf3_start <= '0';
                                        end if;
                                when "01" =>
                                        gf3_state <= "11";
                                        gf3_start <= '1';
                                when "11" =>
                                        gf3_start <= '0';
                                        if (gf3_interrupt = '1') then
                                                interrupt <= (others => '1');
                                                --stop the timer
                                                timer_run <= '0';
                                                gf3_state <= "10";

                                        end if;
                                        if (Bus2IP_Reset = '1') then
                                                gf3_state <= "00";
                                                gf3_reset <= '0';
                                        end if;
                                when "10" =>
                                        interrupt <= (others => '0');
                                        if (Bus2IP_Reset = '1') then
                                                gf3_state <= "00";
                                                gf3_reset <= '0';
                                        end if;
                                when others => null;
                        end case;
                end if;
```

```
        end process ENABLE_GF3;
 ------------------------------------------
 -- Example code to read/write user logic slave model s/w accessible registers
 --
 -- Note:
 -- The example code presented here is to show you one way of reading/writing
 -- software accessible registers implemented in the user logic slave model.
 -- Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
 -- to one software accessible register by the top level template. For example,
 -- if you have four 32 bit software accessible registers in the user logic, you
 -- are basically operating on the following memory mapped registers:
 --
 --   Bus2IP_WrCE or   Memory Mapped
 --     Bus2IP_RdCE   Register
 --         "1000"   C_BASEADDR + 0x0
 --         "0100"   C_BASEADDR + 0x4
 --         "0010"   C_BASEADDR + 0x8
 --         "0001"   C_BASEADDR + 0xC
 --
 ------------------------------------------
 slv_reg_write_select <= Bus2IP_WrCE(0 to 9);
 slv_reg_read_select  <= Bus2IP_RdCE(0 to 9);
 slv_write_ack        <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or
Bus2IP_WrCE(3) or Bus2IP_WrCE(4) or Bus2IP_WrCE(5) or Bus2IP_WrCE(6) or Bus2IP_WrCE(7)
or Bus2IP_WrCE(8) or Bus2IP_WrCE(9);
 slv_read_ack         <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or
Bus2IP_RdCE(3) or Bus2IP_RdCE(4) or Bus2IP_RdCE(5) or Bus2IP_RdCE(6) or Bus2IP_RdCE(7)
or Bus2IP_RdCE(8) or Bus2IP_RdCE(9);

 -- implement slave model register(s)
 SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
 begin

  if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
   if Bus2IP_Reset = '1' then
    slv_reg0 <= (others => '0');
    slv_reg1 <= (others => '0');
    slv_reg2 <= (others => '0');
    slv_reg3 <= (others => '0');
    slv_reg4 <= (others => '0');
    slv_reg5 <= (others => '0');
    slv_reg6 <= (others => '0');
    slv_reg7 <= (others => '0');
    slv_reg8 <= (others => '0');
    slv_reg9 <= (others => '0');
   else
                        slv_reg8 <= m_slv_reg8;
                slv_reg9 <= m_slv_reg9;
                        --load reg0 with the timer value
                        slv_reg0 <= timer;
    case slv_reg_write_select is
     when "1000000000" =>
      for byte_index in 0 to (C_DWIDTH/8)-1 loop
       if ( Bus2IP_BE(byte_index) = '1' ) then
        slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
```

```
          end if;
         end loop;
        when "0100000000" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0010000000" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg2(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0001000000" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg3(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0000100000" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg4(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0000010000" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg5(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0000001000" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg6(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0000000100" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg7(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when "0000000010" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg8(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
```

```
          end if;
         end loop;
        when "0000000001" =>
         for byte_index in 0 to (C_DWIDTH/8)-1 loop
          if ( Bus2IP_BE(byte_index) = '1' ) then
           slv_reg9(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
byte_index*8+7);
          end if;
         end loop;
        when others => null;
       end case;
     end if;
   end if;

 end process SLAVE_REG_WRITE_PROC;

 -- implement slave model register read mux
 SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0, slv_reg1, slv_reg2, slv_reg3,
slv_reg4, slv_reg5, slv_reg6, slv_reg7, slv_reg8, slv_reg9 ) is
 begin

  case slv_reg_read_select is
   when "1000000000" => slv_ip2bus_data <= slv_reg0;
   when "0100000000" => slv_ip2bus_data <= slv_reg1;
   when "0010000000" => slv_ip2bus_data <= slv_reg2;
   when "0001000000" => slv_ip2bus_data <= slv_reg3;
   when "0000100000" => slv_ip2bus_data <= slv_reg4;
   when "0000010000" => slv_ip2bus_data <= slv_reg5;
   when "0000001000" => slv_ip2bus_data <= slv_reg6;
   when "0000000100" => slv_ip2bus_data <= slv_reg7;
   when "0000000010" => slv_ip2bus_data <= slv_reg8;
   when "0000000001" => slv_ip2bus_data <= slv_reg9;
   when others => slv_ip2bus_data <= (others => '0');
  end case;

 end process SLAVE_REG_READ_PROC;

 ----------------------------------------
 -- Example code to generate user logic interrupts
 --
 -- Note:
 -- The example code presented here is to show you one way of generating
 -- interrupts from the user logic. This code snippet infers a counter
 -- and generate the interrupts whenever the counter rollover (the counter
 -- will rollover ~21 sec @50Mhz).
 ----------------------------------------
 -- INTR_PROC : process( Bus2IP_Clk ) is
 --  constant COUNT_SIZE   : integer := 30;
 --  constant ALL_ONES     : std_logic_vector(0 to COUNT_SIZE-1) := (others => '1');
 --  variable counter      : std_logic_vector(0 to COUNT_SIZE-1);
 -- begin
 --
 --  if ( Bus2IP_Clk'event and Bus2IP_Clk = '1' ) then
 --   if ( Bus2IP_Reset = '1' ) then
 --    counter := (others => '0');
 --    interrupt <= (others => '0');
```

```
--    else
--      counter := counter + 1;
--      if ( counter = ALL_ONES ) then
--        interrupt <= (others => '1');
--      else
--        interrupt <= (others => '0');
--      end if;
--    end if;
--  end if;

-- end process INTR_PROC;

 IP2Bus_IntrEvent <= interrupt;

 ----------------------------------------
 -- Example code to drive IP to Bus signals
 ----------------------------------------
 IP2Bus_Data       <= slv_ip2bus_data;

 IP2Bus_WrAck      <= slv_write_ack;
 IP2Bus_RdAck      <= slv_read_ack;
 IP2Bus_Busy       <= '0';
 IP2Bus_Error      <= '0';
 IP2Bus_Retry      <= '0';
 IP2Bus_ToutSup    <= '0';

end IMP;
```

Appendix D: mult_contr_unit.vhd – SoC environment

```
--------------------------------------------------------------------------------
-- Title      : mult_contr_unit
-- Project    : gf3_board
--------------------------------------------------------------------------------
-- File       : mult_ctr_unit.vdh
-- Author     : Beatriz Iaderoza
--                                      based on implementation by Hannes Prokop
-- Last update: 2006-03-24

-- Description: Finite State Machine (FSM) containing 3 processes:
--          multmod:        The main FSM, controlling the fetch and store
--                          operations of data to/from the pci_ctr_unit
--                          and the computation in mult_arith_unit.
--          interrupt:      requests a pci interrupt being raised by the
--                          pci_ctr_unit
--          sync:           updates values for multmod at every rising clk
--                          edge.
--
--          Polynoms are represented in a bit-sliced way where
--          00 equals 0
--          01 equals 1
--          11 equals 2
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- LIBRARY
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
USE work.gf3_board_pkg.all;
--------------------------------------------------------------------------------
-- ENTITY
--------------------------------------------------------------------------------
ENTITY mult_ctr_unit IS
 PORT(
     clk :in std_logic;
         reset   :in std_logic;
         start       :in std_logic;--input is read to be loaded
         slv_reg1:in std_logic_vector(63 downto 0);--degree N
         slv_reg2:in std_logic_vector(63 downto 0);--poly_table top
         slv_reg3:in std_logic_vector(63 downto 0);--poly_table bot
         slv_reg4:in std_logic_vector(63 downto 0);--b top
         slv_reg5:in std_logic_vector(63 downto 0);--b bot
         slv_reg6:in std_logic_vector(63 downto 0);--a top
         slv_reg7:in std_logic_vector(63 downto 0);--a bot
         slv_reg8:out std_logic_vector(63 downto 0);--result top register
         slv_reg9:out std_logic_vector(63 downto 0);--result bottom register
         interrupt:out std_logic
     );
```

```
END mult_ctr_unit;

-------------------------------------------------------------------------------
-- END ENTITY
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- ARCHITECTURE
-------------------------------------------------------------------------------
architecture behaviour of mult_ctr_unit is

COMPONENT mult_arith_unit
PORT
(
        clk             : IN    STD_LOGIC;
        reset           : IN    STD_LOGIC;
        op_code         : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
        data            : IN    STD_LOGIC_VECTOR(63 DOWNTO 0);
        loop_counter    : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        t_coeff_n       : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
        b_coeff_zero    : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
        result_top      : OUT   STD_LOGIC_VECTOR(63 DOWNTO 0);
        result_bot      : OUT   STD_LOGIC_VECTOR(63 DOWNTO 0)
);
END COMPONENT;

subtype STATE_TYPE is std_logic_vector(5 downto 0);

constant res_state          : STATE_TYPE := "000000";--0
constant idle               : STATE_TYPE := "000001";--1
--fetch states
constant fetch_degree       : STATE_TYPE := "000010";--2
constant fetch_degree_fin   : STATE_TYPE := "000011";--3
constant fetch_primpol_top: STATE_TYPE := "000100";--4
constant fetch_primpol_top_fin: STATE_TYPE := "000101";--5
constant fetch_primpol_bot: STATE_TYPE := "000110";--6
constant fetch_primpol_bot_fin: STATE_TYPE := "000111";--7
constant fetch_t_top        : STATE_TYPE := "001000";--8
constant fetch_t_top_fin    : STATE_TYPE := "001001";--9
constant fetch_t_bot        : STATE_TYPE := "001010";--10
constant fetch_b_top        : STATE_TYPE := "001011";--11
constant fetch_b_top_fin    : STATE_TYPE := "001100";--12
constant fetch_b_bot        : STATE_TYPE := "001101";--13
constant fetch_b_bot_fin    : STATE_TYPE := "001110";--14
--computation states
constant begin_loop         : STATE_TYPE := "001111";--15
constant add_res_t          : STATE_TYPE := "010000";--16
constant add_res_2t         :STATE_TYPE := "010001";--17
constant check_coeff        : STATE_TYPE := "010010";--18--does shift b also
constant mask_coeff         : STATE_TYPE := "010011";--19
constant shift_t    : STATE_TYPE := "010100";--20
constant add_t_prim : STATE_TYPE := "010101";--21
constant add_t_2prim        : STATE_TYPE := "010110";--22
--send states
constant req_send_res_top : STATE_TYPE := "010111";--23
constant send_res_top       : STATE_TYPE := "011000";--24
```

```
constant req_send_res_bot : STATE_TYPE := "011001";--25
constant send_res_bot     : STATE_TYPE := "011010";--26
constant req_send_timer : STATE_TYPE := "011011";--27
constant send_timer       : STATE_TYPE := "011100";--28

----

SIGNAL state     : STATE_TYPE;
SIGNAL next_state: STATE_TYPE;
SIGNAL last_state: STATE_TYPE;
SIGNAL new_last_state   : STATE_TYPE;
SIGNAL loop_counter             : std_logic_vector(7 downto 0);
SIGNAL new_loop_counter         : std_logic_vector(7 downto 0);
SIGNAL arith_unit_loop_counter_sig        :std_logic_vector(7 downto 0);
signal arith_unit_data : std_logic_vector(63 downto 0);
signal arith_unit_op_code : STD_LOGIC_VECTOR(3 DOWNTO 0);
signal arith_unit_result_top:  STD_LOGIC_VECTOR(63 DOWNTO 0);
signal arith_unit_result_bot:  STD_LOGIC_VECTOR(63 DOWNTO 0);
signal arith_unit_b_coeff_zero:STD_LOGIC_VECTOR(1 DOWNTO 0);
signal arith_unit_t_coeff_n: STD_LOGIC_VECTOR(1 DOWNTO 0);
signal arith_unit_loop_counter: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL int_state: std_logic;
SIGNAL int_req  : std_logic; --used by mult_ctr FSM

SIGNAL t_coeff_n        : std_logic_vector(1 downto 0);--used in shift_t state and check_coeff state
SIGNAL new_t_coeff_n    : std_logic_vector(1 downto 0);--used in shift_t state and check_coeff state

SIGNAL new_degree       : std_logic_vector(7 downto 0);

--signals for controlling the timer process
SIGNAL timer            : std_logic_vector(31 downto 0);
SIGNAL timer_run        : std_logic;
SIGNAL timer_reset      : std_logic;

begin

        mult_arith_unit_inst : mult_arith_unit
        PORT MAP
        (
        clk       => clk,--inc_clk
        reset             => reset,
        op_code           => arith_unit_op_code,
        data              => arith_unit_data,--data from PCI-buffer
        loop_counter      => arith_unit_loop_counter_sig,
        t_coeff_n         => arith_unit_t_coeff_n,
        b_coeff_zero      => arith_unit_b_coeff_zero,
        result_top        => arith_unit_result_top,--data to mult_ctr_unit
        result_bot        => arith_unit_result_bot--data to PCI-buffer
        );


multmod:
PROCESS (clk, reset, state, next_state, start,
        arith_unit_b_coeff_zero, arith_unit_t_coeff_n,
        arith_unit_result_top,      arith_unit_result_bot, loop_counter, t_coeff_n,
        last_state, arith_unit_loop_counter_sig, timer)
```

```
BEGIN
        --default assignments, changed in FSM
        new_loop_counter <= loop_counter;
        new_t_coeff_n <= t_coeff_n;
        new_last_state <= state;
        new_degree <= arith_unit_loop_counter_sig;
        int_req <= '0';
        timer_run <= '1';
        timer_reset <= NOT RES_ACT;

--slv_reg8(5 downto 0) <= state;

CASE state IS

        WHEN res_state =>
                arith_unit_op_code <=  op_idle;
                interrupt <= '0'; --INTERUPT reset
                new_loop_counter <= (others => '0');
                next_state <= idle;

        WHEN idle => -- idle
                timer_reset <= RES_ACT;
                timer_run <= '0';
                arith_unit_op_code <=  op_idle;
                if start = '1' then   -- get start from software
                        next_state <= fetch_degree;
                else
                        next_state <= state;
                end if;

        --******* GET OPERANDS STATES *********

        WHEN fetch_degree =>
                arith_unit_op_code <= op_idle;
                new_loop_counter <= (others => '0');
                new_loop_counter <= slv_reg1(7 downto 0);
                new_degree <= slv_reg1(7 downto 0)-1;
                next_state <= fetch_primpol_top;

        WHEN fetch_primpol_top =>
                arith_unit_op_code <= op_ld_prim_top;
                arith_unit_data <= slv_reg2;
                next_state <= fetch_primpol_bot;

        WHEN fetch_primpol_bot =>
                arith_unit_op_code <= op_ld_prim_bot;
                arith_unit_data <= slv_reg3;
                next_state <= fetch_b_top;

        WHEN fetch_b_top =>
                arith_unit_op_code <= op_ld_b_top;
                arith_unit_data <= slv_reg4;
                next_state <= fetch_b_bot;

        WHEN fetch_b_bot =>
```

```
        arith_unit_op_code <= op_ld_b_bot;
        arith_unit_data <= slv_reg5;
        next_state <= fetch_t_top;

WHEN fetch_t_top =>
        arith_unit_op_code <= op_ld_t_top;
        arith_unit_data <= slv_reg6;
        next_state <= fetch_t_bot;

WHEN fetch_t_bot =>
        arith_unit_op_code <= op_ld_t_bot;
        arith_unit_data <= slv_reg7;
        next_state <= begin_loop;

--******* COMPUTATION STATES *********

WHEN begin_loop =>
        arith_unit_op_code <=  op_idle;
        -- check loop counter
        if loop_counter > 0 then
                -- check b_coeff_zero
                CASE arith_unit_b_coeff_zero IS
                        WHEN "00" =>
                                next_state <= check_coeff;
                        WHEN "01" =>
                                next_state <= add_res_t;
                        WHEN "11" =>
                                next_state <= add_res_2t;
                        WHEN OTHERS =>
                                --error
                                --fsm_error <= "0001";--simulation only
                                next_state <= res_state;
                END CASE;
        else
                --finished multiplication, output
                next_state <= req_send_res_top;
        end if;

        WHEN add_res_t =>       --compute result+=t
        arith_unit_op_code <=  op_add_res_t;
        next_state <= check_coeff;

        WHEN add_res_2t =>      --compute result+=2t
        arith_unit_op_code <=  op_add_res_2t;
        next_state <= check_coeff;

        WHEN check_coeff => --check t_coeff_n AND shift right b
        arith_unit_op_code <=  op_shift_r_b;
        new_t_coeff_n <= arith_unit_t_coeff_n;
        CASE arith_unit_t_coeff_n IS
                WHEN "00" =>
                        next_state <= shift_t;
                WHEN OTHERS =>
                        next_state <= mask_coeff;
        END CASE;
```

```
                WHEN shift_t =>
                        arith_unit_op_code <=  op_shift_l_t;
                CASE t_coeff_n IS
                        WHEN "00" =>
                        new_loop_counter <= loop_counter - '1'; --decrement loop counter
                                                        next_state <= begin_loop;
                        WHEN "01" =>
                                next_state <= add_t_prim;
                        WHEN "11" =>
                                next_state <= add_t_2prim;
                        WHEN OTHERS =>
                                --error
                                --fsm_error <= "0011";--simulation only
                                next_state <= res_state;
                END CASE;
                WHEN mask_coeff => --masks the Nth coefficient of t
                arith_unit_op_code <=  op_mask_t_coeff;
                if arith_unit_t_coeff_n = "00" then
                        next_state <= shift_t;
                else
                        next_state <= mask_coeff;
                end if;
                WHEN add_t_prim =>
                arith_unit_op_code <=  op_add_t_prim;
                next_state <= begin_loop;
                new_loop_counter <= loop_counter - '1'; --decrement loop counter

        WHEN add_t_2prim =>
                arith_unit_op_code <=  op_add_t_2prim;
                next_state <= begin_loop;
                new_loop_counter <= loop_counter - '1'; --decrement loop counter

                        --******* SEND RESULT STATES *********

                        when req_send_res_top =>
                                slv_reg8 <= arith_unit_result_top;
                                slv_reg9 <= arith_unit_result_bot;
                                arith_unit_op_code <=  op_idle;
                                next_state <= idle;
                                interrupt <= '1'; -- INTERRUPT!!!!<-done calculations

                        WHEN OTHERS =>
                                timer_run <= '0';
                                --error: unknown state
                                arith_unit_op_code <=  op_idle;
                --fsm_error <= "0100"; --simulation only
                next_state <= res_state;
                END CASE;
END PROCESS multmod;

--process updates signals at every rising clock edge
sync:
process (clk, reset)
        begin
                if clk'event and clk = '1' then
                        if reset = RES_ACT then --sync. reset
```

```
                        state <= res_state;
                    else
                        state <= next_state;
                    end if;
                    loop_counter                    <= new_loop_counter;
                    arith_unit_loop_counter_sig <= (new_degree(7 downto 0));
                    t_coeff_n                       <= new_t_coeff_n;
                    last_state                      <= new_last_state;

            end if;
        end process sync;


--simple timer with reset and run input
--used to measure execution time
--result is transmitted over the PCI bus at the end
--of each multmod operation
        timer_proc : PROCESS (clk, reset, timer, timer_reset, timer_run) is
        BEGIN
            IF reset = RES_ACT OR timer_reset = RES_ACT THEN
                timer <= (others => '0');
            ELSIF clk'EVENT AND clk = '1' THEN
                if timer_run = '1' then
                    timer <= timer + '1';
                end if;
            END IF;
        END PROCESS;

end behaviour;
```

Appendix E: mult_arith_unit.vhd – SoC environment

```
--------------------------------------------------------------------------------
-- Title      : mult_arith_unit
-- Project    : gf3_board
--------------------------------------------------------------------------------
-- File       : mult_arith_unit.vhd
-- Author     : Beatriz Iaderoza and Hannes Prokop
-- Company    : University of Victoria - DSD Lab
-- Created    : 2006-01-08
-- Last update: 2006-03-24

-- Description: Performs arithmetical operations for multiplication of
--                                  polynomials over GF3, controlled by mult_ctr_unit
--                                  uses 5 add_64 circuits, a decoder and 9 64-bit Flip-Flops
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- LIBRARY
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
use work.gf3_board_pkg.all;
--------------------------------------------------------------------------------
-- ENTITY
--------------------------------------------------------------------------------
ENTITY mult_arith_unit IS
        PORT
        (
                clk                                                     : IN
        STD_LOGIC;
                reset                                                   : IN
        STD_LOGIC;
                op_code                                         : IN
        STD_LOGIC_VECTOR(3 DOWNTO 0);
                data                                                    : IN
        STD_LOGIC_VECTOR(63 DOWNTO 0);
                loop_counter                                : IN
        STD_LOGIC_VECTOR(7 DOWNTO 0);
                t_coeff_n                                       : OUT
        STD_LOGIC_VECTOR(1 DOWNTO 0);
                b_coeff_zero                                : OUT
        STD_LOGIC_VECTOR(1 DOWNTO 0);
                result_top                                      : OUT
        STD_LOGIC_VECTOR(63 DOWNTO 0);
                result_bot                                      : OUT
        STD_LOGIC_VECTOR(63 DOWNTO 0)
        );
END mult_arith_unit;
--------------------------------------------------------------------------------
-- END ENTITY
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
-- ARCHITECTURE
--------------------------------------------------------------------------------
ARCHITECTURE behaviour OF mult_arith_unit IS
        --adder in/output signals
        SIGNAL adder_res_t_in_a_top      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_in_a_bot      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_in_b_top      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_in_b_bot      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_out_res_top   : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_out_res_bot   : std_logic_vector(63 downto 0);

        SIGNAL adder_res_2t_in_a_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_in_a_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_in_b_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_in_b_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_out_res_top : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_out_res_bot : std_logic_vector(63 downto 0);

        SIGNAL adder_t_prim_in_a_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_in_a_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_in_b_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_in_b_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_out_res_top : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_out_res_bot : std_logic_vector(63 downto 0);

        SIGNAL adder_t_2prim_in_a_top    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_in_a_bot    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_in_b_top    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_in_b_bot    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_out_res_top : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_out_res_bot : std_logic_vector(63 downto 0);

        SIGNAL two_prim_top    : std_logic_vector(63 downto 0);
        SIGNAL two_prim_bot    : std_logic_vector(63 downto 0);

        --enable lines for FFs
        SIGNAL prim_top_ena             : std_logic;
        SIGNAL prim_bot_ena             : std_logic;
        SIGNAL result_top_ena   : std_logic;
        SIGNAL result_bot_ena   : std_logic;
        SIGNAL b_top_ena                : std_logic;
        SIGNAL b_bot_ena                : std_logic;
        SIGNAL t_top_ena                : std_logic;
        SIGNAL t_bot_ena                : std_logic;

        --output lines for FFs
        SIGNAL prim_top                 : std_logic_vector(63 downto 0);
        SIGNAL prim_bot                 : std_logic_vector(63 downto 0);
        SIGNAL result_top_sig   : std_logic_vector(63 downto 0);--result output port
        SIGNAL result_bot_sig   : std_logic_vector(63 downto 0);
        SIGNAL b_top                    : std_logic_vector(63 downto 0);
        SIGNAL b_bot                    : std_logic_vector(63 downto 0);
        SIGNAL t_top                    : std_logic_vector(63 downto 0);
        SIGNAL t_bot                    : std_logic_vector(63 downto 0);
```

```
--data lines for FFs
SIGNAL result_top_new  : std_logic_vector(63 downto 0);
SIGNAL result_bot_new  : std_logic_vector(63 downto 0);
SIGNAL b_top_new       : std_logic_vector(63 downto 0);
SIGNAL b_bot_new       : std_logic_vector(63 downto 0);
SIGNAL t_top_new       : std_logic_vector(63 downto 0);
SIGNAL t_bot_new       : std_logic_vector(63 downto 0);
SIGNAL t_top_masked    : std_logic_vector(63 downto 0);
SIGNAL t_bot_masked    : std_logic_vector(63 downto 0);

SIGNAL t_coeff_n_sig   : std_logic_vector(1 downto 0);
SIGNAL mask            : std_logic_vector(63 downto 0);
SIGNAL op_code_sig     : std_logic_vector(3 downto 0);
SIGNAL data_sig        : std_logic_vector(63 downto 0);
signal ff_clear        : std_logic; --synch. clear for all flip flops

--******* COMPONENT DECLARATIONS *********

COMPONENT add_64
        PORT (
a_top     : IN  std_logic_vector(63 downto 0);
a_bot     : IN  std_logic_vector(63 downto 0);
b_top     : IN  std_logic_vector(63 downto 0);
b_bot     : IN  std_logic_vector(63 downto 0);

sum_top   : OUT  std_logic_vector(63 downto 0);
sum_bot   : OUT  std_logic_vector(63 downto 0)
        );
END COMPONENT;

COMPONENT arith_unit_ff IS
PORT (
        clock     : IN STD_LOGIC ;
        enable    : IN STD_LOGIC ;
        sclr      : IN STD_LOGIC ;
        data      : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        q         : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
);
END COMPONENT;

component decode
PORT (
        S         : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
        O         : OUT std_logic_vector (63 downto 0)
);

end component;


BEGIN
--debug/simulation only port map
--debug_b_bot <= b_bot;
--debug_t_bot <= t_bot;

--******* COMPONENT INSTANTIATION *********
```

```
-- Adders
--add_64_prim_prim : add_64
        --PORT MAP (
        --      a_top   =>      adder_prim_prim_in_a_top,
        --      a_bot   =>      adder_prim_prim_in_a_bot,
   -- b_top     =>      adder_prim_prim_in_b_top,
   -- b_bot     =>      adder_prim_prim_in_b_bot,

    --    sum_top =>      adder_prim_prim_out_res_top,
         --      sum_bot =>      adder_prim_prim_out_res_bot
        --);

add_64_t_prim : add_64
        PORT MAP (
                a_top   =>      adder_t_prim_in_a_top,
                a_bot   =>      adder_t_prim_in_a_bot,
    b_top       =>      adder_t_prim_in_b_top,
    b_bot       =>      adder_t_prim_in_b_bot,
    sum_top =>  adder_t_prim_out_res_top,
                sum_bot =>      adder_t_prim_out_res_bot
        );

add_64_t_2prim : add_64
        PORT MAP (
                a_top   =>      adder_t_2prim_in_a_top,
                a_bot   =>      adder_t_2prim_in_a_bot,
    b_top       =>      adder_t_2prim_in_b_top,
    b_bot       =>      adder_t_2prim_in_b_bot,
    sum_top =>  adder_t_2prim_out_res_top,
                sum_bot =>      adder_t_2prim_out_res_bot
        );

add_64_res_t : add_64
        PORT MAP (
                a_top   =>      adder_res_t_in_a_top,
                a_bot   =>      adder_res_t_in_a_bot,
    b_top       =>      adder_res_t_in_b_top,
    b_bot       =>      adder_res_t_in_b_bot,
    sum_top =>  adder_res_t_out_res_top,
                sum_bot =>      adder_res_t_out_res_bot
        );

add_64_res_2t : add_64
        PORT MAP (
                a_top   =>      adder_res_2t_in_a_top,
                a_bot   =>      adder_res_2t_in_a_bot,
    b_top       =>      adder_res_2t_in_b_top,
    b_bot       =>      adder_res_2t_in_b_bot,
    sum_top =>  adder_res_2t_out_res_top,
                sum_bot =>      adder_res_2t_out_res_bot
        );

--Flip-Flops
t_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
```

```
                enable  => t_top_ena,
                sclr    => ff_clear,
                data => t_top_new,
                q               => t_top
        );
t_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => t_bot_ena,
                sclr    => ff_clear,
                data    => t_bot_new,
                q               => t_bot
        );

b_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => b_top_ena,
                sclr    => ff_clear,
                data    => b_top_new,
                q               => b_top
        );
b_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => b_bot_ena,
                sclr    => ff_clear,
                data    => b_bot_new,
                q               => b_bot
        );

prim_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => prim_top_ena,
                sclr    => ff_clear,
                data    => data_sig,
                q               => prim_top
        );
prim_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => prim_bot_ena,
                sclr    => ff_clear,
                data    => data_sig,
                q               => prim_bot
        );

result_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => result_top_ena,
                sclr    => ff_clear,
                data    => result_top_new,
                q               => result_top_sig
        );
```

```
result_bot_ff : arith_unit_ff
        PORT MAP (
                clock    => clk,
                enable   => result_bot_ena,
                sclr     => ff_clear,
                data     => result_bot_new,
                q                => result_bot_sig
        );

--incoming data ff
data_ff : arith_unit_ff
        PORT MAP (
                clock    => clk,
                enable   => '1',
                sclr     => ff_clear,
                data     => data,
                q                => data_sig
        );

--decoder for getting the bitmask used in mask_t_coeff
decode_inst : decode PORT MAP (
                S                => loop_counter(5 downto 0),
                O                => mask
        );

--Operations using the adders:
--add result t
--add t prim

--add result 2t
--add t 2prim

--adder for prim and prim
--adder_prim_prim_in_a_top <=    prim_top;
--adder_prim_prim_in_a_bot <=    prim_bot;
--adder_prim_prim_in_b_top <=    prim_top;
--adder_prim_prim_in_b_bot <=    prim_bot;

--adder for t and result
adder_res_t_in_a_top <=  t_top;
adder_res_t_in_a_bot <=  t_bot;
adder_res_t_in_b_top <= result_top_sig;
adder_res_t_in_b_bot <= result_bot_sig;

--adder for t and result + t
adder_res_2t_in_a_top <= t_top;
adder_res_2t_in_a_bot <= t_bot;
adder_res_2t_in_b_top <= adder_res_t_out_res_top;
adder_res_2t_in_b_bot <= adder_res_t_out_res_bot;

--adder for t and primitive
adder_t_prim_in_a_top <= t_top;
adder_t_prim_in_a_bot <= t_bot;
adder_t_prim_in_b_top <= prim_top;
adder_t_prim_in_b_bot <= prim_bot;
```

```
--adder for t and 2*primitive
adder_t_2prim_in_a_top <=       t_top;
adder_t_2prim_in_a_bot <=       t_bot;
adder_t_2prim_in_b_top <=       two_prim_top;
adder_t_2prim_in_b_bot <=       two_prim_bot;


--Operations on b:
--Shift right b
--load b top
--load b bottom
with op_code_sig select
        b_top_new (63) <=       '0'                     when op_shift_r_b,
                                                data_sig (63)   when op_ld_b_top,
                                                'X'             when others;
with op_code_sig select
        b_top_new (62 downto 0) <=      b_top(63 downto 1)              when op_shift_r_b,
                                                                        data_sig (62 downto 0)
        when op_ld_b_top,
                                                                        (others => 'X')
        when others;

with op_code_sig select
        b_bot_new (63) <=       '0'                     when op_shift_r_b,
                                                data_sig (63)   when op_ld_b_bot,
                                                'X'                             when others;
with op_code_sig select
        b_bot_new (62 downto 0) <=      b_bot(63 downto 1)              when op_shift_r_b,
                                                                        data_sig (62 downto 0)
        when op_ld_b_bot,
                                                                        (others => 'X')
        when others;
with op_code_sig select
        b_top_ena       <=      '1' when op_shift_r_b,
                                        '1' when op_ld_b_top,
                                        '0' when others;
with op_code_sig select
        b_bot_ena       <=      '1' when op_shift_r_b,
                                        '1' when op_ld_b_bot,
                                        '0' when others;


--Operations on t:
--shift_left_t
--load_t_top
--load_t_bot
--add_t_prim
--mask_t_coeff

--this process selects the inputs to the t-register (MUX)
assign_t_input:
PROCESS (op_code_sig, t_top, t_bot, data_sig, t_bot_masked, t_top_masked,
adder_t_prim_out_res_top,
                adder_t_prim_out_res_bot, adder_t_2prim_out_res_top, adder_t_2prim_out_res_bot)

BEGIN
```

```
        CASE op_code_sig IS
        WHEN op_shift_l_t =>
                t_top_new(0) <= '0';
                t_top_new(63 downto 1) <= t_top(62 downto 0);
                t_bot_new(0) <= '0';
                t_bot_new(63 downto 1) <= t_bot(62 downto 0);
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN op_ld_t_top =>
                t_top_new <= data_sig;
                t_bot_new <= (others => 'X');
                t_top_ena <= '1';
                t_bot_ena <= '0';

        WHEN op_ld_t_bot =>
                t_top_new <= (others => 'X');
                t_bot_new <= data_sig;
                t_top_ena <= '0';
                t_bot_ena <= '1';

        WHEN op_mask_t_coeff =>
                t_top_new <= t_top_masked;
                t_bot_new <= t_bot_masked;
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN op_add_t_prim =>
                t_top_new <= adder_t_prim_out_res_top;
                t_bot_new <= adder_t_prim_out_res_bot;
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN op_add_t_2prim =>
                t_top_new <= adder_t_2prim_out_res_top;
                t_bot_new <= adder_t_2prim_out_res_bot;
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN OTHERS =>
                t_top_new <= (others => 'X');
                t_bot_new <= (others => 'X');
                t_top_ena <= '0';
                t_bot_ena <= '0';
        END CASE;
END PROCESS assign_t_input;


-- Operations on primitive polynomial
-- store
with op_code_sig select
        prim_top_ena  <='1' when op_ld_prim_top,
                                                '0' when others;
with op_code_sig select
        prim_bot_ena  <='1' when op_ld_prim_bot,
                                                '0' when others;
```

```
-- Operations on result
-- add result t
-- add result 2t
-- reset on OP: load t top
with op_code_sig select
        result_top_new  <=      adder_res_t_out_res_top  when op_add_res_t,
                                adder_res_2t_out_res_top when op_add_res_2t,
                                (others => '0')                  when
op_ld_t_top, --reset result
                                (others => 'X')          when others;
with op_code_sig select
        result_bot_new  <=      adder_res_t_out_res_bot  when op_add_res_t,
                                adder_res_2t_out_res_bot when op_add_res_2t,
                                (others => '0')                  when
op_ld_t_top, --reset result
                                (others => 'X')          when others;
with op_code_sig select
        result_top_ena  <=      '1' when op_add_res_t,
                                '1' when op_add_res_2t,
                                '1' when op_ld_t_top, --reset result
                                '0' when others;
with op_code_sig select
        result_bot_ena  <=      '1' when op_add_res_t,
                                '1' when op_add_res_2t,
                                '1' when op_ld_t_top, --reset result
                                '0' when others;

-- Operations on t coeff port
t_coeff_n_sig(1) <= t_top(conv_integer(loop_counter));
t_coeff_n_sig(0) <= t_bot(conv_integer(loop_counter));

--assign output pin
t_coeff_n <= t_coeff_n_sig;

--assign input for FFs
t_top_masked <= t_top and (not mask);
t_bot_masked <= t_bot and (not mask);

result_top <= result_top_sig;
result_bot <= result_bot_sig;

--in our arithmetic 2 x prim is computed like that
two_prim_top <= (NOT prim_top) AND prim_bot;
two_prim_bot <= prim_bot;

b_coeff_zero(1) <= b_top(0);
b_coeff_zero(0) <= b_bot(0);

sync:
process (clk)
        begin
                if clk'event and clk = '1' then
                        if reset = RES_ACT then
                                ff_clear <= '1';
                        else
```

```
                    ff_clear <= '0';
                end if;
                op_code_sig <= op_code;
            end if;

end process sync;
END behaviour;
```

Appendix F: gf3.c – Loosely-coupled environment

```
/*
  Program to enumerate all irreducible and/or primitive polynomials
  over GF(3) using FPGA support.
  Hannes Prokop and Beatriz Iaderoza, University of Victoria - DSD Lab, 2006

  Version: 1.0; final version for performance measurement

  Based on:
  Program to enumerate all irreducible and/or primitive polynomials
  over GF(3).
  by Gilbert Lee

  Modifications:
                - Included driver for interfacing an FPGA for multmod computation
                - Added methodes for measuring HW and SW execution times
                - Removed HTML output options

        Remarks:
                - This code is for compilation with MS Visual Studio.
                  To compile using gcc the ullong data type should be
                  declared as unsigned long long, the format string for printf is then
                  %llu instead if %I64u
*/
#define VERSIONSTRING "GF3 FPGA Version 1.1\n"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef unsigned __int64 ullong;
/* Change MAX to size of unsigned long long */
#define MAX 64
#define MAX_ITERATIONS 10000
#define MEASURE_SW_TIMING 0 //increases runtime by factor 10,000

//FPGA board driver libraries
#include "gf3_board_v3_lib.h"
#include "d:/windriver/samples/shared/pci_diag_lib.h"
#include "d:/windriver/include/windrvr_int_thread.h"
#include "d:/windriver/include/windrvr_events.h"
#include "d:/windriver/include/status_strings.h"

//FPGA related declarations
#define WR_BANK 0
#define RD_BANK 1
#define CTL_BANK 2
static int use_fpga;

//definitions for HW execution time measurement
ullong duration = 0;
ullong totalduration = 0;

//definitions for SW execution time measurement
```

```
#if (MEASURE_SW_TIMING)
double totalelapsed = 0;
#endif

// input command from user
static char line[256];
GF3_BOARD_V3_HANDLE hGF3_BOARD_V3 = NULL;
HANDLE hWD;
DWORD dwAction = 0;
BOOL fRegisteredEvent = FALSE;

ullong ONE = (ullong)1;
ullong TWO = (ullong)2;
ullong counts[3][3][3];
ullong iter_limit = 0;
ullong multcount = 0;

typedef struct{
 ullong top;
 ullong bot;
} Poly_GF3;
/* Each polynomial is expressed in two words.  00 = 0, 01 = 1, 11 = 2 */

/* A list of primitive polynomials
   poly_table[i] contains what x^i is equivalent to.
   E.g. given primitive polynomial x^3 + 2x + 1:
      x^3 ~= x + 2 --> {01,11} = {1,3}
*/

Poly_GF3 poly_table[MAX+1] = {
 /*  0 */ { 0 , 1 },
 /*  1 */ { 0 , 1 },
 /*  2 */ { 0 , 3 },
 /*  3 */ { 1 , 3 },
 /*  4 */ { 0 , 3 },
 /*  5 */ { 1 , 3 },
 /*  6 */ { 0 , 3 },
 /*  7 */ { 1 , 5 },
 /*  8 */ { 0 , 9 },
 /*  9 */ { 1 ,15 },
 /* 10 */ { 0 ,11 },
 /* 11 */ { 1 , 5 },
 /* 12 */ { 2 ,31 },
 /* 13 */ { 1 , 3 },
 /* 14 */ { 0 , 3 },
 /* 15 */ { 1 , 5 },
 /* 16 */ {24 ,27 },
 /* 17 */ { 1,  3 },
 /* 18 */ {32, 39 },
 /* 19 */ { 1,  5 },
 /* 20 */ {34, 35 },
 /* 21 */ { 1, 15 },
 /* 22 */ { 2, 15 },
 /* 23 */ { 1,  9 },
 /* 24 */ { 8, 27 },
 /* 25 */ { 1,  9 },
```

```
  /* 26 */ { 0, 11 },
  /* 27 */ {49, 59 },
  /* 28 */ {48, 51 },
  /* 29 */ {21, 21 },
  /* 30 */ { 0,  3 },
  /* 31 */ { 3, 15 },
  /* 32 */ {24, 27 }
#if (MAX > 32)
 ,
  /* 33 */ {33, 37 },
  /* 34 */ { 0, 11 },
  /* 35 */ { 1,  5 },
  /* 36 */ {48, 57 },
  /* 37 */ { 9, 15 },
  /* 38 */ { 6, 15 },
  /* 39 */ {33, 45 },
  /* 40 */ { 0,  3 },
  /* 41 */ { 1,  3 },
  /* 42 */ {62, 63 },
  /* 43 */ {11, 11 },
  /* 44 */ { 0,  9 },
  /* 45 */ { 5, 13 },
  /* 46 */ {18, 23 },
  /* 47 */ {29, 29 },
  /* 48 */ {32, 61 },
  /* 49 */ { 5, 13 },
  /* 50 */ { 8, 31 },
  /* 51 */ { 1,  3 },
  /* 52 */ {32, 51 },
  /* 53 */ { 1, 29 },
  /* 54 */ { 2,  3 },
  /* 55 */ { 1, 15 },
  /* 56 */ { 8,  9 },
  /* 57 */ {35, 47 },
  /* 58 */ {18, 23 },
  /* 59 */ { 9, 15 },
  /* 60 */ {34, 35 },
  /* 61 */ { 9, 15 },
  /* 62 */ {18, 23 },
  /* 63 */ { 1, 67108865 },
  /* 64 */ { 8,  9 }
#endif
};

/* Global Variables */
int N,D;
int outformat, type;
ullong maxNumber;
ullong ir_count = 0, pr_count = 0;
int printed = 0;
Poly_GF3 threeN_1;
int a[MAX+1] = {0};

void GF3_BOARD_V3_IntHandlerRoutine(GF3_BOARD_V3_HANDLE hGF3_BOARD_V3,
GF3_BOARD_V3_INT_RESULT *intResult)
{
```

```
  printf ("Got Int number %u\n", (UINT)intResult->dwCounter);
}


/* FPGA initialization */
GF3_BOARD_V3_HANDLE GF3_BOARD_V3_LocateAndOpenBoard(DWORD dwVendorID,
DWORD dwDeviceID)
{
  DWORD cards, my_card;
  GF3_BOARD_V3_HANDLE hGF3_BOARD_V3 = NULL;

  if (dwVendorID==0)
  {
    printf ("Enter VendorID: ");
    fgets(line, sizeof(line), stdin);
    sscanf (line, "%x",&dwVendorID);
    if (dwVendorID==0) return NULL;

    printf ("Enter DeviceID: ");
    fgets(line, sizeof(line), stdin);
    sscanf (line, "%x",&dwDeviceID);
  }
  cards = GF3_BOARD_V3_CountCards (dwVendorID, dwDeviceID);
  if (cards==0)
  {
    printf ("%s", GF3_BOARD_V3_ErrorString);
    return NULL;
  }
  else if (cards==1)
    my_card = 1;
  else
  {
    UINT i;

    printf ("Found %u matching PCI cards\n", (UINT)cards);
    printf ("Select card (1-%u): ", (UINT)cards);
    i = 0;
    fgets(line, sizeof(line), stdin);
    sscanf (line, "%d",&i);
    if (i>=1 && i <=cards) my_card = i;
    else
    {
      printf ("Choice is out of range\n");
      return NULL;
    }
  }
  if (!GF3_BOARD_V3_Open (&hGF3_BOARD_V3, dwVendorID, dwDeviceID, my_card - 1))
  {
    printf ("%s", GF3_BOARD_V3_ErrorString);
    return NULL;
  }
  //write reset command to control bank
        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, CTL_BANK, 0, 3);
        printf ("GF3_BOARD_V3 PCI card found!\n");

  return hGF3_BOARD_V3;
```

```
}


/* Printing routines */
void PrintString(Poly_GF3 a){
 int i;
 for(i = N-1; i >= 0; i--)
  if(a.top&(ONE<<i)) printf("2");
  else if(a.bot&(ONE<<i)) printf("1");
  else printf("0");
 printf("\n");
}

void PrintCoeff(Poly_GF3 a){
 int i;
 printf("%d_1", N);
 for(i = N-1; i >= 0; i--){
  if(a.bot & (ONE << i)){
   printf(" %d_%d", i,(a.top & (ONE << i)) ? 2 : 1);
  }
 }
 printf("\n");
}

void PrintPoly(Poly_GF3 a){
 int i;
 printf("x^%d + ", N);
 for(i = N-1; i > 0; i--){
  if(a.top & (ONE << i)){
   if(i != 1){
        printf("2x^%d + ", i);
   } else {
                printf("2x + ");
   }
  } else if(a.bot & (ONE << i)){
   if(i != 1){
                printf("x^%d + ", i);
   } else printf("x + ");
  }
 }
 if(a.top & ONE){
  printf("2");
 } else if(a.bot & ONE){
  printf("1");
 }
 printf("\n");
}

void PrintBit(Poly_GF3 a){
 int i;
 for(i = N-1; i >= 0; i--)
  printf("%d", a.top&(ONE<<i) ? 1 : 0);
 printf("\n");
 for(i = N-1; i >= 0; i--)
  printf("%d", a.bot&(ONE<<i) ? 1 : 0);
 printf("\n");
```

```
}

/* Counting bits in O(number of bits) */
int bitcount(ullong x){
 int c = 0;
 while(x){
  c++;
  x &= x-1;
 }
 return c;
}

/* Density - (By our encoding, we count number of bits in the bottom word) */
int density(Poly_GF3 a){
 return bitcount(a.bot);
}

/* Returns a + b */
Poly_GF3 add(Poly_GF3 a, Poly_GF3 b){
 Poly_GF3 res;
 ullong t1 = a.bot^b.top;
 ullong t2 = a.top^b.bot;
 res.top = t1&t2;
 res.bot = (a.top^t1)|(b.top^t2);
 return res;
}

/* Returns -1 if a < b, 0 if a == b, +1 if a > b */
int cmp(Poly_GF3 a, Poly_GF3 b){
 int i, x, y;
 for(i = N-1; i >= 0; i--){
  x = (a.top & (ONE<<i)) ? 2 : (a.bot & (ONE<<i)) ? 1 : 0;
  y = (b.top & (ONE<<i)) ? 2 : (b.bot & (ONE<<i)) ? 1 : 0;
  if(x < y) return -1;
  if(x > y) return 1;
 }
 return 0;
}

/* Returns a - b, (assuming a > b and a,b are ternary numbers */
Poly_GF3 subtract(Poly_GF3 a, Poly_GF3 b){
 int i, borrow = 0, x, y;
 Poly_GF3 r = {0,0};
 for(i = 0; i < N; i++){
  x = ((a.top & (ONE<<i)) ? 2 : (a.bot & (ONE<<i)) ? 1 : 0) - borrow;
  y =  (b.top & (ONE<<i)) ? 2 : (b.bot & (ONE<<i)) ? 1 : 0;
  if(x < y){
   borrow = 1;
   x += 3;
  } else {
   borrow = 0;
  }
  switch(x-y){
  case 1: r.bot |= (ONE<<i); break;
  case 2: r.bot |= (ONE<<i); r.top |= (ONE<<i); break;
  }
```

```
 }
 return r;
}

/* Returns a % b (a and b treated as ternary numbers) */
Poly_GF3 mod(Poly_GF3 a, Poly_GF3 b){
 Poly_GF3 r = {0,0};
 int i;
 for(i = N-1; i >= 0; i--){
  r.top <<= 1; r.bot <<= 1;
  if(a.top & (ONE<<i)) r.top++;
  if(a.bot & (ONE<<i)) r.bot++;
  while(cmp(b,r) <= 0){
   r = subtract(r, b);
  }
 }
 return r;
}

/* Returns the gcd of a and b (a and b treated as ternary numbers */
Poly_GF3 gcd(Poly_GF3 a, Poly_GF3 b){
 if(!b.top && !b.bot) return a;
 return gcd(b, mod(a, b));
}


/* FPGA-supported version or multmod */
Poly_GF3 hw_multmod(Poly_GF3 a, Poly_GF3 b)
{
        Poly_GF3 result = {0,0};
  int killtime = 0;
        int i =0;
        DWORD data = 0;
        DWORD dump = 0;
 // WD_PCI_SLOT pciSlot;

    if (hGF3_BOARD_V3)
    {
      //write input values to FPGA board
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
N);//degree
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
(int)poly_table[N].top);
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
(int)poly_table[N].bot);
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
(int)b.top);
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
(int)b.bot);
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
(int)a.top);
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, WR_BANK, 0,
(int)a.bot);

                        while (killtime < 3000) {
                                killtime++;
```

```
                        }

                        //write a 0 to bank 2 to get that last value out of the FIFO
                        //printf ("writing zero padding...\n");
                        GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, CTL_BANK, 0, 0);


                        //read results
                        while (i < 4) {
                                data = GF3_BOARD_V3_ReadDword(hGF3_BOARD_V3,
RD_BANK, 0);
                                //printf ("Value %d read: %u\n", i, data);
                                i++;
                                if (i==1) dump = data;
                                if (i==2) result.top = data;
                                if (i==3) result.bot = data;
                                if (i==4) duration = data;
                        }
                        //HW execution time measurement
                        totalduration += duration;
                }
        return result;
}

/* Returns (a * b) mod p (where p = poly_table[N]) */
Poly_GF3 multmod(Poly_GF3 a, Poly_GF3 b){
 Poly_GF3 hw_result = {0,0};
 Poly_GF3 t = a, result = {0,0};
 ullong top_bit = ONE << (N-1);
 int i;

 //for SW timing
 #if (MEASURE_SW_TIMING)
        clock_t start, end;
        double elapsed;
        int j;
        start = clock();
 #endif

        multcount ++;

 //for SW timing
 #if (MEASURE_SW_TIMING)
 for (j=0; j<10000; j++) {
 #endif

                if (use_fpga > 0) {//hardware multmod
                        hw_result = hw_multmod(a, b);
                }

                if (use_fpga != 1) {//software multmod
                        for(i = 0; i < N; i++){
                                if(b.bot & ONE) result = add(result, t);
                                if(b.top & ONE) result = add(result, t);
                                if(t.top & top_bit){
                                t.top = (t.top & ~top_bit) << 1;
```

```
                        t.bot = (t.bot & ~top_bit) << 1;
                        t = add(t, poly_table[N]);
                        t = add(t, poly_table[N]);
                        } else if(t.bot & top_bit){
                        t.top <<= 1;
                        t.bot = (t.bot & ~top_bit) << 1;
                        t = add(t, poly_table[N]);
                        } else {
                        t.top <<= 1;
                        t.bot <<= 1;
                        }
                        b.top >>= 1;
                        b.bot >>= 1;
                }
        }

        if (use_fpga == 2) {//compare HW and SW results

                if (((int)result.top == (int)hw_result.top) & ((int)result.bot ==
(int)hw_result.bot)) {
                        //results match
                }
                else {
                        printf(" HW-SW MISMATCH!!!\n");
                        printf("SW result: %I64u, ", result.top);
                        printf("%I64u\n",                    result.bot);
                        printf("HW result: %u, ",  hw_result.top);
                        printf("%u\n",                                  hw_result.bot);
                }
        }
    #if (MEASURE_SW_TIMING)
    }//end for
    end = clock();
    elapsed = ((double) (end - start)) / (CLOCKS_PER_SEC);
    totalelapsed += elapsed;
    #endif

    if (use_fpga ==1) return hw_result;
    else return result;
}

/* Returns (a^power) mod p (where p = poly_table[N]) */
Poly_GF3 powmod(Poly_GF3 a, Poly_GF3 power){
  Poly_GF3 t = a, result = {0,ONE};
  while(power.top || power.bot){
    if(power.bot & ONE) result = multmod(result, t);
    if(power.top & ONE) result = multmod(result, t);
    t = multmod(multmod(t, t), t);
    power.top >>= 1;
    power.bot >>= 1;
  }
  return result;
}

/* Calculates the minimum polynomial, given a necklace */
Poly_GF3 minpoly(Poly_GF3 necklace){
```

```
 Poly_GF3 root = {0,TWO}, result = {0,0};
 Poly_GF3 f[MAX]; /* f[i] contains the polynomials listing the
                     coefficient of x^i in terms of the primitive root */
 int i, j;

 f[0].top = 0;
 f[0].bot = ONE;
 for(i = 1; i < N; i++)
  f[i].top = f[i].bot = 0;

 root = powmod(root, necklace);

 for(i = 0; i < N; i++){
  if(i) root = multmod(multmod(root, root), root);
  for(j = N-1; j >= 1; j--)
   f[j] = add(f[j-1], multmod(f[j], root));
  f[0] = multmod(f[0], root);
 }

 /* Because we compute (x+a)(x+a^2)...(x+a^n) instead of (x-a)(x-a^2)...(x-a^n)
    we now have to inverse all odd positions */
 for(i = N-1; i >= 0; i--){
  if(f[i].top == ONE){
   if(i % 2 == N % 2) result.top |= ONE<<i;
   result.bot |= ONE<<i;
  } else if(f[i].bot == ONE){
   if(i % 2 != N % 2) result.top |= ONE<<i;
   result.bot |= ONE<<i;
  } else if(f[i].bot) fprintf(stderr, "TROUBLE\n");
 }
 return result;
}

void OutputPoly(Poly_GF3 poly){
 switch(outformat){
 case 0: PrintBit(poly); break;
 case 1: PrintString(poly); break;
 case 2: PrintPoly(poly); break;
 }
 printed++;
}

void PrintIt(){
 int i;
 Poly_GF3 necklace = {0,0};
 Poly_GF3 gcdResult;
 Poly_GF3 mpoly;

 for(i = 1; i <= N; i++){
  necklace.top <<= 1;
  necklace.bot <<= 1;
  if(a[i]) necklace.bot++;
  if(a[i] == 2) necklace.top++;
 }

 mpoly = minpoly(necklace);
```

```
  if(density(mpoly)+1 > D) return;

  ir_count++;
  gcdResult = gcd(necklace, threeN_1);
  if(gcdResult.top == 0 && gcdResult.bot == ONE){
   ++pr_count;
   ++printed;
     switch(outformat){
     case 0: PrintString(mpoly); break;
     case 1: PrintCoeff(mpoly); break;
     case 2: PrintPoly(mpoly); break;
     }

  } else if(type == 0){
     switch(outformat){
     case 0: PrintString(mpoly); break;
     case 1: PrintCoeff(mpoly); break;
     case 2: PrintPoly(mpoly); break;
     }
  }
  if(maxNumber && printed >= maxNumber){
   exit(1);
  }
}

void Gen(int t, int p){
  int j;
  if(t > N){
   if(p == N) PrintIt();
  } else {
   a[t] = a[t-p];
   Gen(t+1,p);
   for(j=a[t-p]+1; j<=2; j++) {
     a[t] = j;
     Gen(t+1,t);
   }
  }
}

void ProcessInput(int argc, char *argv[]){
  int i;
  outformat = 0;
  type = 0;
  maxNumber = 0;
  if (argc > 2) {
            use_fpga = atoi(argv[1]);
            N = atoi(argv[2]);
  }
  if (argc > 3) D = atoi(argv[3]); else D = N+1;
  if (argc > 4) outformat = atoi(argv[4]); else outformat = 0;
  if (argc > 5) type = atoi(argv[5]);
  if (argc > 6) maxNumber = atoi(argv[6]);
  if (argc > 7 || N < 2 || N > MAX || use_fpga > 2){
   fprintf(stderr,"Usage: gf3_fpga fpgaMode n d format type number\n");
   fprintf(stderr,"       All params except 'fpgaMode' and 'n' are optional\n");
   fprintf(stderr,"       use_FPGA = 0,1,2 = no, FPGA only, compare FPGA and SW\n");
```

```
        fprintf(stderr,"        n = degree (2-%d)\n", MAX);
  fprintf(stderr,"        d = density (2-%d)\n", MAX);
  fprintf(stderr,"        format = 0,1,2 = string, ceoffs, poly\n");
  fprintf(stderr,"        type = 0/1 = all/primitive only\n");
  fprintf(stderr,"        num = max to find (0 for all)\n");
  exit( 1 );
 }
 if (use_fpga == 0) printf("using SW only\n");
 else if (use_fpga == 1) printf("using FPGA only\n");
 else if (use_fpga == 2) printf("using FPGA and comparing results to SW\n");
 else printf("invalid value for use_fpga: %d", use_fpga);

 for(i = 0; i < N; i++)
  threeN_1.top |= (ONE<<i);
 threeN_1.bot = threeN_1.top;
}


int main(int argc, char **argv){
 unsigned int test = 0;

 fprintf(stdout, VERSIONSTRING);
 if (MEASURE_SW_TIMING) printf("SW execution time measurement\n");
 ProcessInput(argc, argv);

 //Initialize the FPGA
 if (use_fpga > 0) {
        // Make sure WinDriver is loaded
  if (!PCI_Get_WD_handle(&hWD))
     return 0;
  WD_Close (hWD);

        hGF3_BOARD_V3 =
GF3_BOARD_V3_LocateAndOpenBoard(GF3_BOARD_V3_DEFAULT_VENDOR_ID,
GF3_BOARD_V3_DEFAULT_DEVICE_ID);

        //enable interrupt
        if (!GF3_BOARD_V3_IntEnable(hGF3_BOARD_V3,
GF3_BOARD_V3_IntHandlerRoutine))
            printf ("%s", GF3_BOARD_V3_ErrorString);

  printf ("Application accesses hardware using WinDriver.\n");
        //printf ("generating interrupt request.\n");
        //GF3_BOARD_V3_WriteDword(hGF3_BOARD_V3, CTL_BANK, 0, 5);
 }

 Gen(1,1);

 if(type == 0){
  printf("Number of irreducible polynomials = %I64u\n", ir_count);
 }
 printf("Number of primitive polynomials  = %I64u\n", pr_count);
 printf("Number of multiplications        = %I64u\n", multcount);
 #if (MEASURE_SW_TIMING)
 printf ("The total time spent in multmod is %e\n", totalelapsed);
 #endif
```

```
 if (use_fpga == 0) printf("using SW only\n");
 else if (use_fpga == 1) printf("using FPGA only\n");
 else if (use_fpga == 2) printf("using FPGA and comparing results to SW\n");
 if (use_fpga >0  && hGF3_BOARD_V3) {
          printf("Number of FPGA cycles = %I64u\n", totalduration);
          GF3_BOARD_V3_Close(hGF3_BOARD_V3);
 }
 return 0;
}
```

Appendix G: mult_contr_unit.vhd – Loosely-coupled environment

```
--------------------------------------------------------------------------------
-- Title      : mult_contr_unit
-- Project    : gf3_board
--------------------------------------------------------------------------------
-- File       : mult_ctr_unit.vdh
-- Author     : Hannes Prokop and Beatriz Iaderoza
-- Company    : University of Victoria - DSD Lab
-- Created    : 2005-12-21
-- Last update: 2006-03-24

-- Description: Finite State Machine (FSM) containing 3 processes:
--                                  multmod:        The main FSM, controlling the fetch and store
--                                                  operations of data to/from the
pci_ctr_unit
--                                                  and the computation in mult_arith_unit.
--                                  interrupt:      requests a pci interrupt being raised by the
--                                                  pci_ctr_unit
--                                  sync:           updates values for multmod at every rising clk
--                                                  edge.
--
--                                  Polynoms are represented in a bit-sliced way where
--                                  00 equals 0
--                                  01 equals 1
--                                  11 equals 2
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- LIBRARY
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
USE work.gf3_board_pkg.all;
--------------------------------------------------------------------------------
-- ENTITY
--------------------------------------------------------------------------------
ENTITY mult_ctr_unit IS

    PORT(
        clk                                     : IN  STD_LOGIC;
                reset                   : IN  STD_LOGIC;

                        --ports connected to PCI controller
                        resultrdy                       : IN    STD_LOGIC;
                        resultreq               : OUT   STD_LOGIC;
                        resultrec               : IN    STD_LOGIC;
                        inputrdy                        : IN    STD_LOGIC;
                        inputreq                : OUT   STD_LOGIC;
                        inputrec                : IN    STD_LOGIC;
                        pci_result                      : OUT   STD_LOGIC_VECTOR(63
DOWNTO 0);
```

```
                        data                                    : IN    STD_LOGIC_VECTOR(63
DOWNTO 0); --from PCI bus
                        debug_state                             : OUT   STD_LOGIC_VECTOR(4
DOWNTO 0);
                        debug_data                              : OUT   STD_LOGIC_VECTOR(7
DOWNTO 0);
                        pci_intreq                              : OUT   STD_LOGIC;
                        pci_intreqack           : IN    STD_LOGIC;
                        --data signals from mult_arith_unit
                        arith_unit_result_top           : IN STD_LOGIC_VECTOR(63
DOWNTO 0);
                        arith_unit_result_bot           : IN STD_LOGIC_VECTOR(63
DOWNTO 0);

                        --control signals for arithmetical_unit
                        arith_unit_op_code                              : OUT
        STD_LOGIC_VECTOR(3 DOWNTO 0);
                        arith_unit_loop_counter         : OUT   STD_LOGIC_VECTOR(7
DOWNTO 0);
                        arith_unit_t_coeff_n            : IN    STD_LOGIC_VECTOR(1
DOWNTO 0);
                        arith_unit_b_coeff_zero         : IN    STD_LOGIC_VECTOR(1
DOWNTO 0)
    );
END mult_ctr_unit;

--------------------------------------------------------------------------------
-- END ENTITY
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- ARCHITECTURE
--------------------------------------------------------------------------------
architecture behaviour of mult_ctr_unit is

subtype STATE_TYPE is std_logic_vector(5 downto 0);

constant res_state                      : STATE_TYPE := "000000";--0
constant idle                                   : STATE_TYPE := "000001";--1
--fetch states
constant fetch_degree                   : STATE_TYPE := "000010";--2
constant fetch_degree_fin       : STATE_TYPE := "000011";--3
constant fetch_primpol_top              : STATE_TYPE := "000100";--4
constant fetch_primpol_top_fin  : STATE_TYPE := "000101";--5
constant fetch_primpol_bot              : STATE_TYPE := "000110";--6
constant fetch_primpol_bot_fin  : STATE_TYPE := "000111";--7
constant fetch_a_top                    : STATE_TYPE := "001000";--8
constant fetch_a_top_fin        : STATE_TYPE := "001001";--9
constant fetch_a_bot                    : STATE_TYPE := "001010";--10
constant fetch_b_top                    : STATE_TYPE := "001011";--11
constant fetch_b_top_fin        : STATE_TYPE := "001100";--12
constant fetch_b_bot                    : STATE_TYPE := "001101";--13
constant fetch_b_bot_fin        : STATE_TYPE := "001110";--14
--computation states
constant begin_loop                             : STATE_TYPE := "001111";--15
constant add_res_t                      : STATE_TYPE := "010000";--16
```

```
constant add_res_2t                    : STATE_TYPE := "010001";--17
constant check_coeff                   : STATE_TYPE := "010010";--18--does shift b also
constant mask_coeff                    : STATE_TYPE := "010011";--19
constant shift_t                       : STATE_TYPE := "010100";--20
constant add_t_prim                    : STATE_TYPE := "010101";--21
constant add_t_2prim                   : STATE_TYPE := "010110";--22
--send states
constant req_send_res_top              : STATE_TYPE := "010111";--23
constant send_res_top                  : STATE_TYPE := "011000";--24
constant req_send_res_bot              : STATE_TYPE := "011001";--25
constant send_res_bot                  : STATE_TYPE := "011010";--26
constant req_send_timer          : STATE_TYPE := "011011";--27
constant send_timer                    : STATE_TYPE := "011100";--28

----

SIGNAL state                     : STATE_TYPE;
SIGNAL next_state                : STATE_TYPE;
SIGNAL last_state                : STATE_TYPE;
SIGNAL new_last_state   : STATE_TYPE;
SIGNAL loop_counter              : std_logic_vector(7 downto 0);
SIGNAL new_loop_counter          : std_logic_vector(7 downto 0);
SIGNAL arith_unit_loop_counter_sig       :std_logic_vector(7 downto 0);
SIGNAL int_state                 : std_logic;
SIGNAL int_req                   : std_logic; --used by mult_ctr FSM
--SIGNAL timer_int_req   : std_logic; --used by the timer FSM

--data output signals for flip-flops
--SIGNAL result_top                      : std_logic_vector(63 downto 0); --d lines for output FFs
--SIGNAL result_bot                      : std_logic_vector(63 downto 0);
SIGNAL t_coeff_n                       : std_logic_vector(1 downto 0);--used in shift_t state and
check_coeff state
SIGNAL new_t_coeff_n             : std_logic_vector(1 downto 0);--used in shift_t state and
check_coeff state

SIGNAL new_degree                      : std_logic_vector(7 downto 0);

--signals for controlling the timer process
SIGNAL timer                           : std_logic_vector(31 downto 0);
SIGNAL timer_run                       : std_logic;
SIGNAL timer_reset                     : std_logic;

begin

--simulation only
        --debug_state <= state;
        debug_data <= data (7 downto 0);
        arith_unit_loop_counter <= arith_unit_loop_counter_sig;
--/simulation only

multmod:
PROCESS (clk, reset, state, next_state, inputrdy, inputrec, data,
                resultrec,      resultrdy, arith_unit_b_coeff_zero, arith_unit_t_coeff_n,
                arith_unit_result_top,      arith_unit_result_bot, loop_counter, t_coeff_n,
                last_state, arith_unit_loop_counter_sig, timer)
```

```
BEGIN
        --default assignments, changed in FSM
        inputreq <= '0';
        resultreq <= '0';
        pci_result <= (others => '0');
        new_loop_counter <= loop_counter;
        new_t_coeff_n <= t_coeff_n;
        new_last_state <= state;
        new_degree <= arith_unit_loop_counter_sig;
        int_req <= '0';
        timer_run <= '1';
        timer_reset <= NOT RES_ACT;

CASE state IS

                WHEN res_state =>
                        arith_unit_op_code <=  op_idle;
                        inputreq <= '0';
                        resultreq <= '0';
                        new_loop_counter <= (others => '0');
                        next_state <= idle;

                WHEN idle => -- idle
                        timer_reset <= RES_ACT;
                        timer_run <= '0';
                        arith_unit_op_code <=  op_idle;
                        if inputrdy = '1' then

                                next_state <= fetch_degree;
                        else
                                inputreq <= '0';    --wait for input
                                next_state <= state;
                        end if;

                --******* GET OPERANDS STATES *********

                WHEN fetch_degree =>
                        inputreq <= '1';
                        arith_unit_op_code <= op_idle;

                        if inputrec = '1' then
                                --set loop counter
                                new_loop_counter <= data (7 downto 0);
                                new_degree <= data(7 downto 0)-1;
                                inputreq <= '0';
                                next_state <= fetch_degree_fin;
                        else
                                next_state <= state; --wait
                                timer_run <= '0';
                        end if;

                WHEN fetch_degree_fin =>
                        arith_unit_op_code <= op_idle;
                        inputreq <= '0';
                        if inputrec = '0' then
                                next_state <= fetch_primpol_top;
```

```
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_primpol_top =>
        arith_unit_op_code <= op_ld_prim_top;
        inputreq <= '1';
        if inputrec = '1' then
                next_state <= fetch_primpol_top_fin;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_primpol_top_fin =>
        arith_unit_op_code <= op_idle;
        inputreq <= '0';
        if inputrec = '0' then
                next_state <= fetch_primpol_bot;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_primpol_bot =>
        arith_unit_op_code <= op_ld_prim_bot;
        inputreq <= '1';
        if inputrec = '1' then
                next_state <= fetch_primpol_bot_fin;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_primpol_bot_fin =>
        arith_unit_op_code <= op_idle;
        inputreq <= '0';
        if inputrec = '0' then
                next_state <= fetch_b_top;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_b_top =>
        arith_unit_op_code <= op_ld_b_top;
        inputreq <= '1';
        if inputrec = '1' then
                next_state <= fetch_b_top_fin;
        else
                arith_unit_op_code <= op_idle;
                next_state <= state;
                timer_run <= '0';
        end if;
```

```
WHEN fetch_b_top_fin =>
        arith_unit_op_code <= op_idle;
        inputreq <= '0';
        if inputrec = '0' then
                next_state <= fetch_b_bot;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_b_bot =>
        arith_unit_op_code <= op_ld_b_bot;
        inputreq <= '1';
        if inputrec = '1' then
                next_state <= fetch_b_bot_fin;
        else
                arith_unit_op_code <= op_idle;
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_b_bot_fin =>
        arith_unit_op_code <= op_idle;
        inputreq <= '0';
        if inputrec = '0' then
                next_state <= fetch_a_top;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_a_top =>
        arith_unit_op_code <= op_ld_t_top;
        inputreq <= '1';
        if inputrec = '1' then
                next_state <= fetch_a_top_fin;
        else
                arith_unit_op_code <= op_idle;
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_a_top_fin =>
        arith_unit_op_code <= op_idle;
        inputreq <= '0';
        if inputrec = '0' then
                next_state <= fetch_a_bot;
        else
                next_state <= state;
                timer_run <= '0';
        end if;

WHEN fetch_a_bot =>
        arith_unit_op_code <= op_ld_t_bot;
        inputreq <= '1';
        if inputrec = '1' then
```

```
                    next_state <= begin_loop;
                    inputreq <= '0';
            else
                    arith_unit_op_code <= op_idle;
                    next_state <= state;
                    timer_run <= '0';
            end if;
--dont need fin state as begin_loop will deassert inputreq

--******* COMPUTATION STATES *********

WHEN begin_loop =>
            arith_unit_op_code <=  op_idle;
            -- check loop counter
            if loop_counter > 0 then
                    -- check b_coeff_zero
                    CASE arith_unit_b_coeff_zero IS
                            WHEN "00" =>
                                    next_state <= check_coeff;
                            WHEN "01" =>
                                    next_state <= add_res_t;
                            WHEN "11" =>
                                    next_state <= add_res_2t;
                            WHEN OTHERS =>
                                    --error
                                    --fsm_error <= "0001";--simulation
only
                                    next_state <= res_state;
                    END CASE;
            else
                    --finished multiplication, output
                    next_state <= req_send_res_top;
            end if;

WHEN add_res_t =>       --compute result+=t
            arith_unit_op_code <=  op_add_res_t;
            next_state <= check_coeff;

WHEN add_res_2t =>      --compute result+=2t
            arith_unit_op_code <=  op_add_res_2t;
            next_state <= check_coeff;

WHEN check_coeff => --check t_coeff_n AND shift right b
            arith_unit_op_code <=  op_shift_r_b;
            new_t_coeff_n <= arith_unit_t_coeff_n;
            CASE arith_unit_t_coeff_n IS
                    WHEN "00" =>
                            next_state <= shift_t;
                    WHEN OTHERS =>
                            next_state <= mask_coeff;
            END CASE;

WHEN shift_t =>
            arith_unit_op_code <=  op_shift_l_t;
            CASE t_coeff_n IS
                    WHEN "00" =>
```

```
                                        new_loop_counter <= loop_counter - '1'; --
decrement loop counter
                                        next_state <= begin_loop;
                                WHEN "01" =>
                                        next_state <= add_t_prim;
                                WHEN "11" =>
                                        next_state <= add_t_2prim;
                                WHEN OTHERS =>
                                        --error
                                        --fsm_error <= "0011";--simulation only
                                        next_state <= res_state;
                        END CASE;

                WHEN mask_coeff => --masks the Nth coefficient of t
                        arith_unit_op_code <=  op_mask_t_coeff;
                        if arith_unit_t_coeff_n = "00" then
                                next_state <= shift_t;
                        else
                                next_state <= mask_coeff;
                        end if;

                WHEN add_t_prim =>
                        arith_unit_op_code <=  op_add_t_prim;
                        next_state <= begin_loop;
                        new_loop_counter <= loop_counter - '1'; --decrement loop counter

                WHEN add_t_2prim =>
                        arith_unit_op_code <=  op_add_t_2prim;
                        next_state <= begin_loop;
                        new_loop_counter <= loop_counter - '1'; --decrement loop counter

                --******* SEND RESULT STATES *********

                when req_send_res_top =>
                        pci_result <= arith_unit_result_top;
                        arith_unit_op_code <=  op_idle;
                        if resultrdy = '1' then
                                -- send top result word
                                next_state <= send_res_top;
                                resultreq <= '1';
                        else
                                next_state <= state; --wait until clear to send
                                resultreq <= '0';
                                timer_run <= '0';
                        end if;

                when send_res_top =>
                        pci_result <= arith_unit_result_top;
                        arith_unit_op_code <=  op_idle;
                        if resultrec = '1' then
                                next_state <= req_send_res_bot;
                                resultreq <= '0';
                        else
                                --wait until output is accepted
                                next_state <= state;
                                resultreq <= '1';
```

```
                timer_run <= '0';
        end if;

when req_send_res_bot =>
        pci_result <= arith_unit_result_bot;
        arith_unit_op_code <=  op_idle;
        if resultrdy = '1' then
                next_state <= send_res_bot;
                resultreq <= '1'; --begin output handshake
        else
                next_state <= state; --wait until clear to send
                resultreq <= '0';
                timer_run <= '0';
        end if;

when send_res_bot =>
        --int_req <= '1'; --request pci interrupt
        pci_result <= arith_unit_result_bot;
        arith_unit_op_code <=  op_idle;
        if resultrec = '1' then --output handshake complete
                next_state <= req_send_timer;
                resultreq <= '0';
        else
                next_state <= state;
                resultreq <= '1';
                timer_run <= '0';
        end if;

when req_send_timer =>
        timer_run <= '0';
        pci_result (63 downto 32) <= (others => '0');
        pci_result (31 downto  0) <= timer;
        arith_unit_op_code <=  op_idle;
        if resultrdy = '1' then
                next_state <= send_timer;
                resultreq <= '1'; --begin output handshake
        else
                next_state <= state; --wait until clear to send
                resultreq <= '0';
        end if;

when send_timer =>
        timer_run <= '0';
        --int_req <= '1'; --request pci interrupt
        pci_result (63 downto 32) <= (others => '0');
        pci_result (31 downto  0) <= timer;
        arith_unit_op_code <=  op_idle;
        if resultrec = '1' then --output handshake complete
                next_state <= idle;
                resultreq <= '0';
        else
                next_state <= state;
                resultreq <= '1';
        end if;

WHEN OTHERS =>
```

```
                                timer_run <= '0';
                                --error: unknown state
                                arith_unit_op_code <=  op_idle;
                                --fsm_error <= "0100"; --simulation only
                                next_state <= res_state;
                END CASE;
END PROCESS multmod;

--process updates signals at every rising clock edge
sync:
process (clk, reset)
        begin
                if clk'event and clk = '1' then
                        if reset = RES_ACT then --sync. reset
                                state <= res_state;
                        else
                                state <= next_state;
                        end if;
                        loop_counter                                    <= new_loop_counter;
                        arith_unit_loop_counter_sig <= (new_degree(7 downto 0));
                        t_coeff_n                                       <= new_t_coeff_n;
                        last_state                                      <= new_last_state;

                end if;
        end process sync;

--process receives interrupt request from mult_ctr FSM,
--and forwards the request to the pci_ctr_unit.
--waits then for pci_ctr_unit's acknowledge before a new interrupt can be sent
interrupt :     PROCESS (clk, reset) is
        BEGIN
                IF reset = RES_ACT THEN
                        int_state <= '0';
                ELSIF clk'EVENT AND clk = '1' THEN
                        CASE int_state IS
                                WHEN '0' => --idle
                                        pci_intreq      <= '0';
                                        IF int_req = '1' THEN
                                                int_state <= '1';
                                        END IF;

                                WHEN '1' => --waiting for pci ACK
                                        pci_intreq      <= '1';
                                        IF pci_intreqack = '1' THEN
                                                int_state <= '0';
                                        END IF;
                        END CASE;
                END IF;
        END PROCESS;

--simple timer with reset and run input
--used to measure execution time
--result is transmitted over the PCI bus at the end
--of each multmod operation
        timer_proc : PROCESS (clk, reset, timer, timer_reset, timer_run) is
        BEGIN
```

```
        IF reset = RES_ACT OR timer_reset = RES_ACT THEN
                timer <= (others => '0');
        ELSIF clk'EVENT AND clk = '1' THEN
                if timer_run = '1' then
                        timer <= timer + '1';
                end if;
        END IF;
END PROCESS;

end behaviour;
```

Appendix H: mult_arith_unit.vhd – Loosely-coupled environment

```
--------------------------------------------------------------------------------
-- Title      : mult_arith_unit
-- Project    : gf3_board
--------------------------------------------------------------------------------
-- File       : mult_arith_unit.vhd
-- Author     : Hannes Prokop and Beatriz Iaderoza
-- Company    : University of Victoria - DSD Lab
-- Created    : 2006-01-08
-- Last update: 2006-03-24

-- Description: Performs arithmetical operations for multiplication of
--                                  polynomials over GF3, controlled by mult_ctr_unit
--                                  uses 5 add_64 circuits, a decoder and 9 64-bit Flip-Flops
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- LIBRARY
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
use work.gf3_board_pkg.all;
--------------------------------------------------------------------------------
-- ENTITY
--------------------------------------------------------------------------------
ENTITY mult_arith_unit IS
        PORT
        (
                clk                                                     : IN
        STD_LOGIC;
                reset                                           : IN
        STD_LOGIC;
                op_code                                         : IN
        STD_LOGIC_VECTOR(3 DOWNTO 0);
                data                                            : IN
        STD_LOGIC_VECTOR(63 DOWNTO 0);
                loop_counter                            : IN
        STD_LOGIC_VECTOR(7 DOWNTO 0);
                t_coeff_n                                       : OUT
        STD_LOGIC_VECTOR(1 DOWNTO 0);
                b_coeff_zero                            : OUT
        STD_LOGIC_VECTOR(1 DOWNTO 0);
                result_top                                      : OUT
        STD_LOGIC_VECTOR(63 DOWNTO 0);
                --debug, simulation only ports
                debug_t_bot                                     : OUT
        STD_LOGIC_VECTOR(63 DOWNTO 0);
```

```
            debug_b_bot                                    : OUT
        STD_LOGIC_VECTOR(63 DOWNTO 0);
            result_bot                                     : OUT
        STD_LOGIC_VECTOR(63 DOWNTO 0)
        );
END mult_arith_unit;
--------------------------------------------------------------------------------
-- END ENTITY
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- ARCHITECTURE
--------------------------------------------------------------------------------
ARCHITECTURE behaviour OF mult_arith_unit IS
        --adder in/output signals
        SIGNAL adder_res_t_in_a_top      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_in_a_bot      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_in_b_top      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_in_b_bot      : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_out_res_top   : std_logic_vector(63 downto 0);
        SIGNAL adder_res_t_out_res_bot   : std_logic_vector(63 downto 0);

        SIGNAL adder_res_2t_in_a_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_in_a_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_in_b_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_in_b_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_out_res_top  : std_logic_vector(63 downto 0);
        SIGNAL adder_res_2t_out_res_bot  : std_logic_vector(63 downto 0);

        SIGNAL adder_t_prim_in_a_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_in_a_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_in_b_top     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_in_b_bot     : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_out_res_top  : std_logic_vector(63 downto 0);
        SIGNAL adder_t_prim_out_res_bot  : std_logic_vector(63 downto 0);

        SIGNAL adder_t_2prim_in_a_top    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_in_a_bot    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_in_b_top    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_in_b_bot    : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_out_res_top : std_logic_vector(63 downto 0);
        SIGNAL adder_t_2prim_out_res_bot : std_logic_vector(63 downto 0);

        --SIGNAL adder_prim_prim_in_a_top      : std_logic_vector(63 downto 0);
        --SIGNAL adder_prim_prim_in_a_bot      : std_logic_vector(63 downto 0);
        --SIGNAL adder_prim_prim_in_b_top      : std_logic_vector(63 downto 0);
        --SIGNAL adder_prim_prim_in_b_bot      : std_logic_vector(63 downto 0);
        --SIGNAL adder_prim_prim_out_res_top : std_logic_vector(63 downto 0);
        --SIGNAL adder_prim_prim_out_res_bot : std_logic_vector(63 downto 0);
        SIGNAL two_prim_top    : std_logic_vector(63 downto 0);
        SIGNAL two_prim_bot    : std_logic_vector(63 downto 0);

        --enable lines for FFs
        SIGNAL prim_top_ena             : std_logic;
        SIGNAL prim_bot_ena             : std_logic;
        SIGNAL result_top_ena   : std_logic;
```

```
SIGNAL result_bot_ena   : std_logic;
SIGNAL b_top_ena                : std_logic;
SIGNAL b_bot_ena                : std_logic;
SIGNAL t_top_ena                : std_logic;
SIGNAL t_bot_ena                : std_logic;

--output lines for FFs
SIGNAL prim_top                 : std_logic_vector(63 downto 0);
SIGNAL prim_bot                 : std_logic_vector(63 downto 0);
SIGNAL result_top_sig   : std_logic_vector(63 downto 0);--result output port
SIGNAL result_bot_sig   : std_logic_vector(63 downto 0);
SIGNAL b_top                    : std_logic_vector(63 downto 0);
SIGNAL b_bot                    : std_logic_vector(63 downto 0);
SIGNAL t_top                    : std_logic_vector(63 downto 0);
SIGNAL t_bot                    : std_logic_vector(63 downto 0);

--data lines for FFs
SIGNAL result_top_new   : std_logic_vector(63 downto 0);
SIGNAL result_bot_new   : std_logic_vector(63 downto 0);
SIGNAL b_top_new                : std_logic_vector(63 downto 0);
SIGNAL b_bot_new                : std_logic_vector(63 downto 0);
SIGNAL t_top_new                : std_logic_vector(63 downto 0);
SIGNAL t_bot_new                : std_logic_vector(63 downto 0);
SIGNAL t_top_masked             : std_logic_vector(63 downto 0);
SIGNAL t_bot_masked             : std_logic_vector(63 downto 0);

SIGNAL t_coeff_n_sig    : std_logic_vector(1 downto 0);
SIGNAL mask                             : std_logic_vector(63 downto 0);
SIGNAL op_code_sig              : std_logic_vector(3 downto 0);
SIGNAL data_sig                         : std_logic_vector(63 downto 0);
signal ff_clear                 : std_logic; --synch. clear for all flip flops

--******* COMPONENT DECLARATIONS *********

COMPONENT add_64
        PORT (
a_top     : IN  std_logic_vector(63 downto 0);
a_bot     : IN  std_logic_vector(63 downto 0);
b_top     : IN  std_logic_vector(63 downto 0);
b_bot     : IN  std_logic_vector(63 downto 0);

sum_top           : OUT  std_logic_vector(63 downto 0);
                  sum_bot     : OUT  std_logic_vector(63 downto 0)
        );
END COMPONENT;

COMPONENT arith_unit_ff IS
PORT (
        clock           : IN STD_LOGIC ;
        enable          : IN STD_LOGIC ;
        sclr            : IN STD_LOGIC ;
        data            : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        q               : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
);
END COMPONENT;
```

```
component decode
PORT (
        data        : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
        eq0         : OUT STD_LOGIC ;
        eq1         : OUT STD_LOGIC ;
        eq2         : OUT STD_LOGIC ;
        eq3         : OUT STD_LOGIC ;
        eq4         : OUT STD_LOGIC ;
        eq5         : OUT STD_LOGIC ;
        eq6         : OUT STD_LOGIC ;
        eq7         : OUT STD_LOGIC ;
        eq8         : OUT STD_LOGIC ;
        eq9         : OUT STD_LOGIC ;
        eq10        : OUT STD_LOGIC ;
        eq11        : OUT STD_LOGIC ;
        eq12        : OUT STD_LOGIC ;
        eq13        : OUT STD_LOGIC ;
        eq14        : OUT STD_LOGIC ;
        eq15        : OUT STD_LOGIC ;
        eq16        : OUT STD_LOGIC ;
        eq17        : OUT STD_LOGIC ;
        eq18        : OUT STD_LOGIC ;
        eq19        : OUT STD_LOGIC ;
        eq20        : OUT STD_LOGIC ;
        eq21        : OUT STD_LOGIC ;
        eq22        : OUT STD_LOGIC ;
        eq23        : OUT STD_LOGIC ;
        eq24        : OUT STD_LOGIC ;
        eq25        : OUT STD_LOGIC ;
        eq26        : OUT STD_LOGIC ;
        eq27        : OUT STD_LOGIC ;
        eq28        : OUT STD_LOGIC ;
        eq29        : OUT STD_LOGIC ;
        eq30        : OUT STD_LOGIC ;
        eq31        : OUT STD_LOGIC ;
        eq32        : OUT STD_LOGIC ;
        eq33        : OUT STD_LOGIC ;
        eq34        : OUT STD_LOGIC ;
        eq35        : OUT STD_LOGIC ;
        eq36        : OUT STD_LOGIC ;
        eq37        : OUT STD_LOGIC ;
        eq38        : OUT STD_LOGIC ;
        eq39        : OUT STD_LOGIC ;
        eq40        : OUT STD_LOGIC ;
        eq41        : OUT STD_LOGIC ;
        eq42        : OUT STD_LOGIC ;
        eq43        : OUT STD_LOGIC ;
        eq44        : OUT STD_LOGIC ;
        eq45        : OUT STD_LOGIC ;
        eq46        : OUT STD_LOGIC ;
        eq47        : OUT STD_LOGIC ;
        eq48        : OUT STD_LOGIC ;
        eq49        : OUT STD_LOGIC ;
        eq50        : OUT STD_LOGIC ;
        eq51        : OUT STD_LOGIC ;
        eq52        : OUT STD_LOGIC ;
```

```
                eq53            : OUT STD_LOGIC ;
                eq54            : OUT STD_LOGIC ;
                eq55            : OUT STD_LOGIC ;
                eq56            : OUT STD_LOGIC ;
                eq57            : OUT STD_LOGIC ;
                eq58            : OUT STD_LOGIC ;
                eq59            : OUT STD_LOGIC ;
                eq60            : OUT STD_LOGIC ;
                eq61            : OUT STD_LOGIC ;
                eq62            : OUT STD_LOGIC ;
                eq63            : OUT STD_LOGIC
        );
end component;


BEGIN
--debug/simulation only port map
debug_b_bot <= b_bot;
debug_t_bot <= t_bot;

--******* COMPONENT INSTANTIATION *********

-- Adders
--add_64_prim_prim : add_64
        --PORT MAP (
        --      a_top   =>      adder_prim_prim_in_a_top,
        --      a_bot   =>      adder_prim_prim_in_a_bot,
   -- b_top       =>      adder_prim_prim_in_b_top,
   -- b_bot       =>      adder_prim_prim_in_b_bot,

    --      sum_top =>      adder_prim_prim_out_res_top,
          --      sum_bot =>      adder_prim_prim_out_res_bot
        --);

add_64_t_prim : add_64
        PORT MAP (
                a_top   =>      adder_t_prim_in_a_top,
                a_bot   =>      adder_t_prim_in_a_bot,
    b_top       =>      adder_t_prim_in_b_top,
    b_bot       =>      adder_t_prim_in_b_bot,

        sum_top =>      adder_t_prim_out_res_top,
                sum_bot =>      adder_t_prim_out_res_bot
        );

add_64_t_2prim : add_64
        PORT MAP (
                a_top   =>      adder_t_2prim_in_a_top,
                a_bot   =>      adder_t_2prim_in_a_bot,
    b_top       =>      adder_t_2prim_in_b_top,
    b_bot       =>      adder_t_2prim_in_b_bot,

        sum_top =>      adder_t_2prim_out_res_top,
                sum_bot =>      adder_t_2prim_out_res_bot
        );
```

```
add_64_res_t : add_64
        PORT MAP (
                a_top   =>      adder_res_t_in_a_top,
                a_bot   =>      adder_res_t_in_a_bot,
    b_top       =>      adder_res_t_in_b_top,
    b_bot       =>      adder_res_t_in_b_bot,

        sum_top =>      adder_res_t_out_res_top,
                sum_bot =>      adder_res_t_out_res_bot
        );

add_64_res_2t : add_64
        PORT MAP (
                a_top   =>      adder_res_2t_in_a_top,
                a_bot   =>      adder_res_2t_in_a_bot,
    b_top       =>      adder_res_2t_in_b_top,
    b_bot       =>      adder_res_2t_in_b_bot,

        sum_top =>      adder_res_2t_out_res_top,
                sum_bot =>      adder_res_2t_out_res_bot
        );

--Flip-Flops
t_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => t_top_ena,
                sclr    => ff_clear,
                data    => t_top_new,
                q               => t_top
        );
t_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => t_bot_ena,
                sclr    => ff_clear,
                data    => t_bot_new,
                q               => t_bot
        );

b_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => b_top_ena,
                sclr    => ff_clear,
                data    => b_top_new,
                q               => b_top
        );
b_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => b_bot_ena,
                sclr    => ff_clear,
                data    => b_bot_new,
                q               => b_bot
        );
```

```
prim_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => prim_top_ena,
                sclr    => ff_clear,
                data    => data_sig,
                q               => prim_top
        );
prim_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => prim_bot_ena,
                sclr    => ff_clear,
                data    => data_sig,
                q               => prim_bot
        );

result_top_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => result_top_ena,
                sclr    => ff_clear,
                data    => result_top_new,
                q               => result_top_sig
        );
result_bot_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => result_bot_ena,
                sclr    => ff_clear,
                data    => result_bot_new,
                q               => result_bot_sig
        );

--incoming data ff
data_ff : arith_unit_ff
        PORT MAP (
                clock   => clk,
                enable  => '1',
                sclr    => ff_clear,
                data    => data,
                q               => data_sig
        );

--decoder for getting the bitmask used in mask_t_coeff
decode_inst : decode PORT MAP (
                data    => loop_counter(5 downto 0),
                eq0             => mask(0),
                eq1             => mask(1),
                eq2             => mask(2),
                eq3             => mask(3),
                eq4             => mask(4),
                eq5             => mask(5),
                eq6             => mask(6),
                eq7             => mask(7),
```

```
eq8             => mask(8),
eq9             => mask(9),
eq10    => mask(10),
eq11    => mask(11),
eq12    => mask(12),
eq13    => mask(13),
eq14    => mask(14),
eq15    => mask(15),
eq16    => mask(16),
eq17    => mask(17),
eq18    => mask(18),
eq19    => mask(19),
eq20    => mask(20),
eq21    => mask(21),
eq22    => mask(22),
eq23    => mask(23),
eq24    => mask(24),
eq25    => mask(25),
eq26    => mask(26),
eq27    => mask(27),
eq28    => mask(28),
eq29    => mask(29),
eq30    => mask(30),
eq31    => mask(31),
eq32    => mask(32),
eq33    => mask(33),
eq34    => mask(34),
eq35    => mask(35),
eq36    => mask(36),
eq37    => mask(37),
eq38    => mask(38),
eq39    => mask(39),
eq40    => mask(40),
eq41    => mask(41),
eq42    => mask(42),
eq43    => mask(43),
eq44    => mask(44),
eq45    => mask(45),
eq46    => mask(46),
eq47    => mask(47),
eq48    => mask(48),
eq49    => mask(49),
eq50    => mask(50),
eq51    => mask(51),
eq52    => mask(52),
eq53    => mask(53),
eq54    => mask(54),
eq55    => mask(55),
eq56    => mask(56),
eq57    => mask(57),
eq58    => mask(58),
eq59    => mask(59),
eq60    => mask(60),
eq61    => mask(61),
eq62    => mask(62),
eq63    => mask(63)
```

```
);

--Operations using the adders:
--add result t
--add t prim

--add result 2t
--add t 2prim

--adder for prim and prim
--adder_prim_prim_in_a_top <=    prim_top;
--adder_prim_prim_in_a_bot <=    prim_bot;
--adder_prim_prim_in_b_top <=    prim_top;
--adder_prim_prim_in_b_bot <=    prim_bot;

--adder for t and result
adder_res_t_in_a_top <=  t_top;
adder_res_t_in_a_bot <=  t_bot;
adder_res_t_in_b_top <= result_top_sig;
adder_res_t_in_b_bot <= result_bot_sig;

--adder for t and result + t
adder_res_2t_in_a_top <= t_top;
adder_res_2t_in_a_bot <= t_bot;
adder_res_2t_in_b_top <= adder_res_t_out_res_top;
adder_res_2t_in_b_bot <= adder_res_t_out_res_bot;

--adder for t and primitive
adder_t_prim_in_a_top <= t_top;
adder_t_prim_in_a_bot <= t_bot;
adder_t_prim_in_b_top <= prim_top;
adder_t_prim_in_b_bot <= prim_bot;

--adder for t and 2*primitive
adder_t_2prim_in_a_top <=        t_top;
adder_t_2prim_in_a_bot <=        t_bot;
adder_t_2prim_in_b_top <=        two_prim_top;
adder_t_2prim_in_b_bot <=        two_prim_bot;


--Operations on b:
--Shift right b
--load b top
--load b bottom
with op_code_sig select
        b_top_new (63) <=        '0'                    when op_shift_r_b,
                                 data_sig (63)          when op_ld_b_top,
                                 'X'                    when others;

with op_code_sig select
        b_top_new (62 downto 0) <=      b_top(63 downto 1)          when op_shift_r_b,
                                        data_sig (62 downto 0)

        when op_ld_b_top,
                                        (others => 'X')

        when others;
```

```
with op_code_sig select
        b_bot_new (63) <=       '0'                     when op_shift_r_b,
                                                data_sig (63)   when op_ld_b_bot,
                                                'X'                     when others;
with op_code_sig select
        b_bot_new (62 downto 0) <=      b_bot(63 downto 1)              when op_shift_r_b,
                                                                data_sig (62 downto 0)
        when op_ld_b_bot,
                                                                (others => 'X')
        when others;
with op_code_sig select
        b_top_ena       <=      '1' when op_shift_r_b,
                                        '1' when op_ld_b_top,
                                        '0' when others;
with op_code_sig select
        b_bot_ena       <=      '1' when op_shift_r_b,
                                        '1' when op_ld_b_bot,
                                        '0' when others;


--Operations on t:
--shift_left_t
--load_t_top
--load_t_bot
--add_t_prim
--mask_t_coeff

--this process selects the inputs to the t-register (MUX)
assign_t_input:
PROCESS (op_code_sig, t_top, t_bot, data_sig, t_bot_masked, t_top_masked,
adder_t_prim_out_res_top,
                adder_t_prim_out_res_bot, adder_t_2prim_out_res_top, adder_t_2prim_out_res_bot)

BEGIN
        CASE op_code_sig IS
        WHEN op_shift_l_t =>
                t_top_new(0) <= '0';
                t_top_new(63 downto 1) <= t_top(62 downto 0);
                t_bot_new(0) <= '0';
                t_bot_new(63 downto 1) <= t_bot(62 downto 0);
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN op_ld_t_top =>
                t_top_new <= data_sig;
                t_bot_new <= (others => 'X');
                t_top_ena <= '1';
                t_bot_ena <= '0';

        WHEN op_ld_t_bot =>
                t_top_new <= (others => 'X');
                t_bot_new <= data_sig;
                t_top_ena <= '0';
                t_bot_ena <= '1';

        WHEN op_mask_t_coeff =>
```

```
                t_top_new <= t_top_masked;
                t_bot_new <= t_bot_masked;
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN op_add_t_prim =>
                t_top_new <= adder_t_prim_out_res_top;
                t_bot_new <= adder_t_prim_out_res_bot;
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN op_add_t_2prim =>
                t_top_new <= adder_t_2prim_out_res_top;
                t_bot_new <= adder_t_2prim_out_res_bot;
                t_top_ena <= '1';
                t_bot_ena <= '1';

        WHEN OTHERS =>
                t_top_new <= (others => 'X');
                t_bot_new <= (others => 'X');
                t_top_ena <= '0';
                t_bot_ena <= '0';
        END CASE;
END PROCESS assign_t_input;


-- Operations on primitive polynomial
-- store
with op_code_sig select
        prim_top_ena  <='1' when op_ld_prim_top,
                                                '0' when others;
with op_code_sig select
        prim_bot_ena  <='1' when op_ld_prim_bot,
                                                '0' when others;

-- Operations on result
-- add result t
-- add result 2t
-- reset on OP: load t top
with op_code_sig select
        result_top_new  <=      adder_res_t_out_res_top  when op_add_res_t,
                                                adder_res_2t_out_res_top when op_add_res_2t,
                                                (others => '0')                 when
op_ld_t_top, --reset result
                                                (others => 'X')         when others;
with op_code_sig select
        result_bot_new  <=      adder_res_t_out_res_bot  when op_add_res_t,
                                                adder_res_2t_out_res_bot when op_add_res_2t,
                                                (others => '0')                 when
op_ld_t_top, --reset result
                                                (others => 'X')         when others;
with op_code_sig select
        result_top_ena    <=    '1' when op_add_res_t,
                                                '1' when op_add_res_2t,
                                                '1' when op_ld_t_top, --reset result
                                                '0' when others;
```

```
with op_code_sig select
        result_bot_ena    <=        '1' when op_add_res_t,
                                    '1' when op_add_res_2t,
                                    '1' when op_ld_t_top, --reset result
                                    '0' when others;


-- Operations on t coeff port
t_coeff_n_sig(1) <= t_top(conv_integer(loop_counter));
t_coeff_n_sig(0) <= t_bot(conv_integer(loop_counter));

--assign output pin
t_coeff_n <= t_coeff_n_sig;

--assign input for FFs
t_top_masked <= t_top and (not mask);
t_bot_masked <= t_bot and (not mask);


result_top <= result_top_sig;
result_bot <= result_bot_sig;

--in our arithmetic 2 x prim is computed like that
two_prim_top <= (NOT prim_top) AND prim_bot;
two_prim_bot <= prim_bot;

b_coeff_zero(1) <= b_top(0);
b_coeff_zero(0) <= b_bot(0);

sync:
process (clk)
        begin
                if clk'event and clk = '1' then
                        if reset = RES_ACT then
                                ff_clear <= '1';
                        else
                                ff_clear <= '0';
                        end if;
                        op_code_sig <= op_code;
                end if;

end process sync;
END behaviour;
```