ADAPTIVE EMBEDDED TECHNOLOGIES: HARDWARE ACCELERATION

By:

Ryan Meghdies-Vardeh

Computer Engineering, BEng

Ryerson University, Toronto, Canada, 2012

A thesis presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2015

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public

Adaptive Embedded Technologies: Hardware Acceleration

Master of Applied Science, 2015

Ryan Meghdies-Vardeh

Electrical and Computer Engineering

Ryerson University

ABSTRACT

This thesis establishes the benefits of multi-architecture systems by using reconfigurable modules in conjunction with a case integration strategy to improve system performance. The modules and strategies discussed in this thesis provide opportunities to the improve system performance of processing units designed for the consumer market.

The primary objective for this work is to improve the performance of consumer processors using programmable logic, while ensuring the changes are abstracted from operating systems and software applications. This thesis accomplishes this using specified integration strategies, protocols and through optimization of device drivers.

ACKNOWLEDGMENTS

First and foremost I thank my family for their motivation and pushing me to be the ambitious and creative man I am today.

There have been a number of notable educators who have inspired me and helped guide me through my undergrad and post-graduate studies. Dr. Vadim Geurkov has worked with me over the last 4 years. With his supervision, I was awarded a bronze medal for my undergraduate project. More recently, he has helped inspire and guide me with the work in this thesis and that found in [1]. Dr. Lev Kirischian provided an academic foundation for which my research began and provided opportunities which helped me pursue my areas of interest.

Finally, I would like to thank AMD who gave me an opportunity to work with cutting edge hardware design and design verification methodologies. In particular my mentor Darlington Opera who helped train me and challenge me and Behrooz Karimian who worked with me in endeavours related to the inventions found in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

INTRODUCTION

Intellectual Property

At the time in which this paper was written, Ryan Meghdies-Vardeh had submitted multiple patent applications to Advanced Micro Devices, Inc. (AMD). AMD has decided to pursue the protection of two of those filings [2] and [3]. A third filing, is discussed in the future work section is likely the most promising of the three: [1]. The research compiled for this thesis lead to the invention of these patents. See Appendix B to learn more about AMD.

Overview

This thesis provides an introductory analysis of General Purpose Processors (GPP) and Field Programmable Gate Arrays (FPGA) architectures, their benefits and drawbacks. In the past, consumer processors and reprogrammable logic, have not worked in conjunction with one another. Traditional processing architectures are considered to be the superior choice when there is enough volume demand and sufficient research and development resources. In contrast, FPGAs provided a platform that would significantly lower venture development costs and thereby make it possible for individuals and corporations to invest in, and more importantly afford to, develop new hardware solutions.

In this thesis, we identify that CPUs/GPUs and FPGAs have their own unique benefits and the architecture of the future will strategically approach the integration of these two technologies. It is important to note that this work targets consumer market devices. There are two key contribution to this future looking architecture presented in this thesis. First using through the “Hardware Acceleration Islands” (HWAI), a design strategy which strategically embeds DPR enabled FPGAs within exiting processor designs (without requiring software changes, minimizing risk and research and development costs). Secondly, by offering a technique that will enhance the HWAI modules and optional optimizations to processor device drivers.

CHAPTER I

1.1 MOTIVATION

With the end of Moore's Law [4] engineers must make use of smarter and more innovative processor architecture design concepts. Traditional brute force approaches translate into larger systems that are power hungry and more costly to manufacture. This is no longer a feasible technique as mobile processing and portable devices become the focus of the consumer market. As seen in the figure below [5], existing architecture designs are for the first time becoming smaller and engineers are looking for new ways to generate greater performance without proportional overhead.

1.2 OBJECTIVE

A successful solution would be one that minimizes the research and development risks, is easy to integrate within existing processor designs, and delivers worthwhile performance improvements using innovative system design. The "Hardware Acceleration Islands" in conjunction with the proposed case integration strategy offers such benefits, so as to make market adoption feasible and profitable for semiconductor manufacturing companies.

While this thesis targets consumer market processors, it is important to recognize that most modern supercomputers look to high-end consumer processors (GPUs and CPUs) [6].

1.3 EFFECTIVE CONTRIBUTIONS

There were a number of problems that arose in developing solutions for the aforementioned problems while staying within the constraints of the objective goals. As a result, it is recommended to use the hardware solution in conjunction with the protocol and strategy.

The hardware solution injects programmable logic into processor technology on the same die using similar strategies to embedded microprocessors. Using the proposed strategy, these hardware acceleration islands can absorb high-performance tasks and reduce area by also absorbing infrequent or temporal hardware logic. This methodology will not require any 3rd party software changes. Modifying device drivers will be sufficient to hide the suggested system changes. Future works may require optimizations to existing operating system settings.

There is also a protocol that hardware systems can take advantage of to improve the efficiency of the hardware acceleration islands. The protocol will work to minimize/eliminate delays in the pipeline and requires minimal hardware overhead (only 2 control signals).

1.4 THESIS ORGANIZATION

The first chapter will share the motivation of this research, what are the problems being solved and why they are necessary to be addressed. The objective of this thesis is clarified as it presents a unique area and set of goals.

The organization of the remainder of this thesis (chapter 2 and onwards) is structured to provide sufficient background to the build-up towards the primary design work found here. Chapter 2 will outline existing solutions similar to the work of this thesis or attempting to solve similar problems. The pros and cons of each will be evaluated. The following chapter will explain the new approach / primary work of this thesis. Chapter 3 will walk through the details of how to recreate this work.

With a strong understanding of the design work, chapter 4 will discuss the implementation and practical examination of the design work to evaluate the performance and tools to be used. In the final two chapters the obtained results will be compared with similar works and finally the future work for myself and hopefully to motivate other individuals to pursue this avenue of research and development.

CHAPTER II:

2.1 BACKGROUND

2.1.1 Insight Into Modern Applications

Since the introduction of FPGA technology, hardware developers have been working to make use of each architecture's relative benefits. Today these architectures are used in conjunction with each other. ASIC developers now commonly implement SOC designs on FPGAs, validating them thoroughly before investing the millions necessary to tape-out the product in the form of an ASIC. This is referred to as "FPGA Prototyping". Understanding such background and the progression of FPGA technology will provide a clearer image of the direction and impact of this architecture. While this thesis will not dive into this FPGA prototyping, it is helpful to understand 3 key benefits that this methodology provides.

1) Reliable Verification: Implementing designs on FPGAs is a reliable way to ensure the final ASIC will be functionally correct. Previously verification efforts were much more costly (in terms of research and development) or almost non-existent. Previous to FPGA prototyping, ASIC manufacturers relied primarily on software to verify their designs. About a third of all current SOC designs are fault-free during first silicon pass, with nearly half of all re-spins caused by functional logic errors [7].

2) Time-to-Market (TTM): FPGA prototyping has enabled hardware designers to shorten the TTM period. Since less verification resources are required and verification

accounts for as much as 80% of the design process, designers can save money and release new products faster, which in-turn increases the rate of innovation. In a consumer driven market that is keen on technology, meeting deadlines set according to demand peaks is critical to a corporations' success. These peaks present a small window in the market, and missing it could render a project useless thereby costing the company the much of the capital which was invested in the product (typically the research is reused for the progression of IP(s) [8].

3) Development Cost: By reducing the number of re-spins required through FPGA prototyping, hardware design corporations are able to save millions of dollars. These re-spins are caused by a number of factors including: firmware issues, power issues, mixed-signal interface related issues, race condition issues (that were not detected by PD (Physical Design) software, clocking domain issues, functional issues, and more [9]. In [10], there are five precautionary steps that are listed in which corporations take to minimize their risk of having a re-spin:

a) Constraint random verification
b) More effective block (IP) level verification
c) Verification reuse from block level to System level
d) Architecture of test bench using reusable methodologies
e) A reusable and scalable verification

2.1.2 Implementation

The implementation modules, located in Appendix A, demonstrate how "Constraint random verification" and "More effective block (IP) level verification" are achieved and being approached within the modern day hardware design industry. The technologies made use of are the latest solutions used by the most renowned hardware design corporations and design teams. In fact, many companies are still moving to this methodology. The demand for this can be seen by the sheer number of job postings to simply help convert/create design and design verification environments to make use of these techniques.

2.1.3 Verification Tools

It is well known that verification efforts can account for 60 to 80 percent of the hardware development process [11]; where larger designs typically requiring the higher percentages. There are a number of different stages when it comes to hardware verification, most stages run in parallel during the course of a project. In one of the first stages, engineers develop their concept design modules and algorithms using software. Software verification is highly effective at finding high level faults in a design due to two key characteristics: easy development (fast and low cost approach), as well as fast results (compile times for an FPGA can take hours when making even the smallest changes and can take an exorbitant amount of time to simulate). However, software verification is not without its limitations. Software is limited in its ability to mimic

hardware. While software models have made leaps and bounds in this area even capable of modeling the timing delay between modules, hardware and software are still inherently distinct and therefore limited in verifying the design. They are however an effective emulation tool for the conceptual design of the module (ex. testing algorithms which will be implemented in hardware).

In addition to software verification, engineering teams develop and implement their designs using “Hardware Description Languages” or HDLs. The two foundational HDLs include VHDL (VHSIC Hardware Description Language) & Verilog. The code found in these .vhdl and .v files (respectively to the languages above), are then translated to the required format as per a specific architecture/technology (an ASIC, Xilinx Kintex 7 FPGA, Altera Cyclone V FPGA, etc.). For example, a bit stream is generated based on the internal structure and resources available within the specific FPGA being used. This bit stream is used to program the FPGA being used by configuring the hardware, typically through JTAG interface, by filling in LUTs, configuring routing, and manipulating other resources.

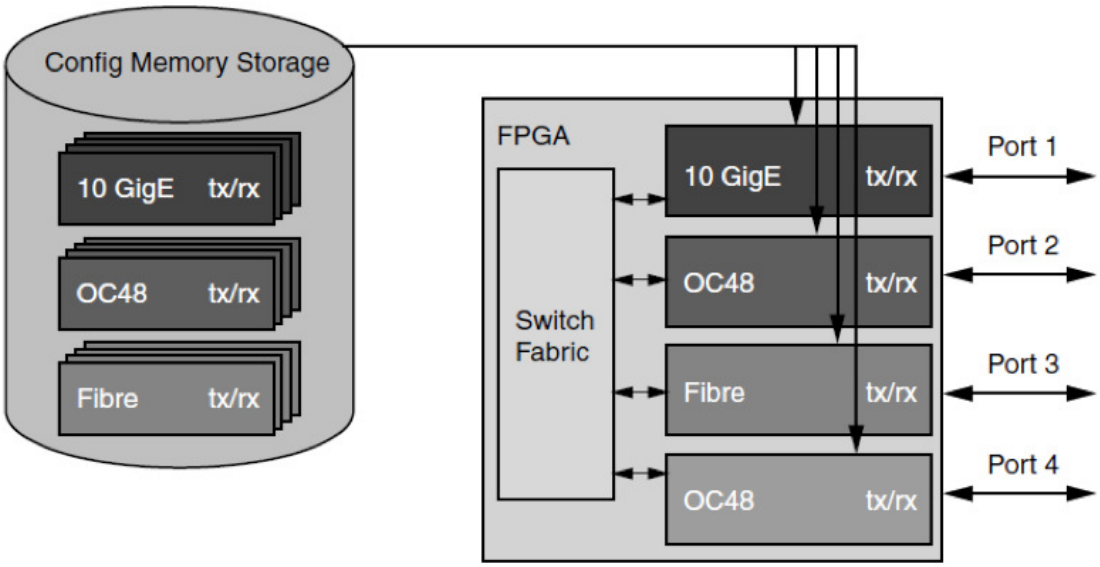


Figure 1 – Bit Stream Configuration [12]

From a verification perspective there are a number of options for engineers to verify their code at this stage, of which two fundamental concepts are discussed. First, engineers will often include additional verification oriented code that is built into the language (VHDL / Verilog) to ensure some conditions within respective states are met. Secondly, if the correct arguments are passed when invoking modern hardware simulators, such as VCS developed by Synopsys Inc., then files containing the values of registers and pins are dumped. These files can then be examined to determine if the behaviour is as expected.

Processor systems have become far more advanced, consequently hardware design files have grown exponentially larger over the last years. As a result, improved verification environments (speed, flexibility, features) have become a central focus in the hardware design industry. With this growing demand, two key issues were identified: a

limited library to verify hardware and manually analyzing the waves was extremely time consuming and difficult. When engineers looked at these limitations they had two goals: (1) To expand the available verification environment and enable hardware designers to build with ease similar to that which software provides. (2) To automate the manual processes that tended are labour intensive and repetitive (both characteristics for opportunities to develop a software solution).

In 2002, engineers released the revolutionary HDL (Hardware Description Language) - HVL (Hardware Verification Language) combination based on extensions to Verilog [13]. While there were predecessors which SystemVerilog inherited from, SystemVerilog provided superior functionality and modern features. As a result, SystemVerilog was selected as the IEEE standard in 2005 and by 2009 was merged with the base Verilog standard [14]. This decision demonstrates just how significant of a relationship existed between the hardware design and design verification world. To this day, SystemVerilog remains as the foundation to which more advanced tools are built on top of. It is not to be taken for granted by any means or compared against UVM (Universal Verification Methodology). UVM is built upon SystemVerilog, and as such should be identified as an extension. This is worth mentioning when examining the implementation found in Appendix A.

2.1.4 ASIC & FPGA Architectures

Field Programmable Gate Arrays have come a long way in recent years and continue to make leaps and bounds in the semiconductor industry. Their unique ability to be reconfigured is the reason why FPGAs are able to take on varying complex tasks that with an optimized hardware configuration. In order to accomplish hardware reconfigurability, FPGAs are designed using a distinct hardware layout, as seen in Figure 2 bellow. Logic blocks are LUT representations of the logic gates used inside ASICs (refer to Figure 2). These logic blocks are linked together using routing lines and switching blocks are used to select which logic blocks will be connected together.

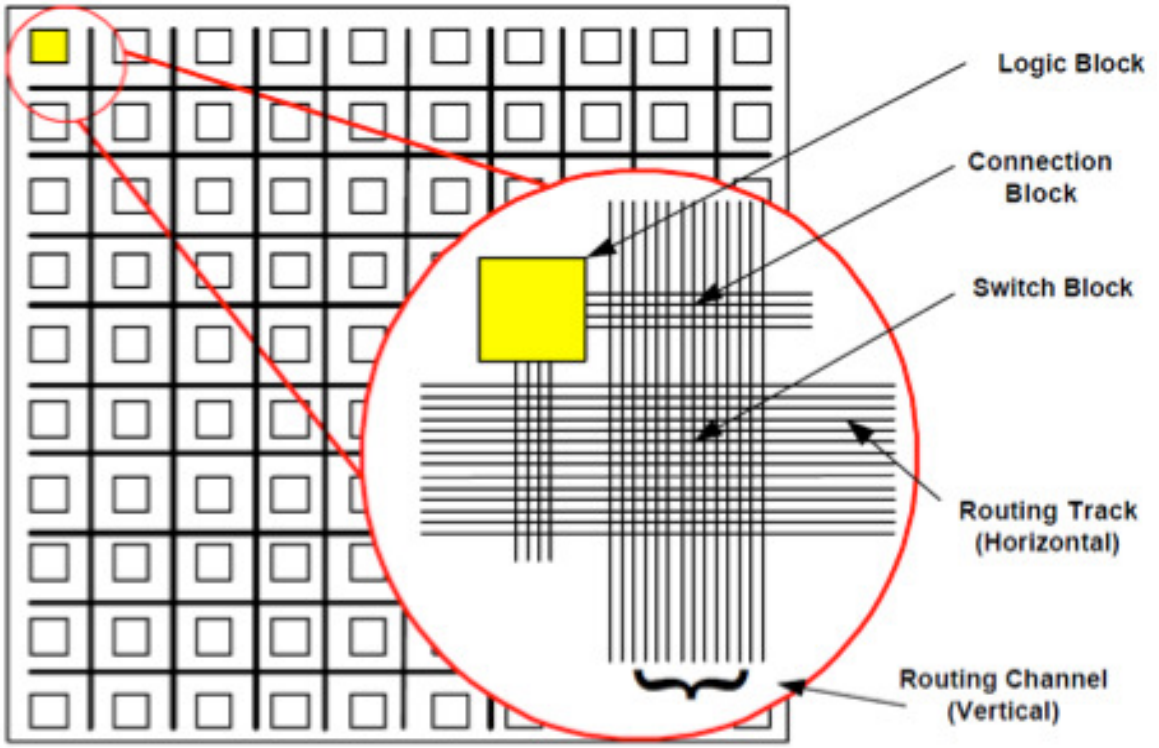
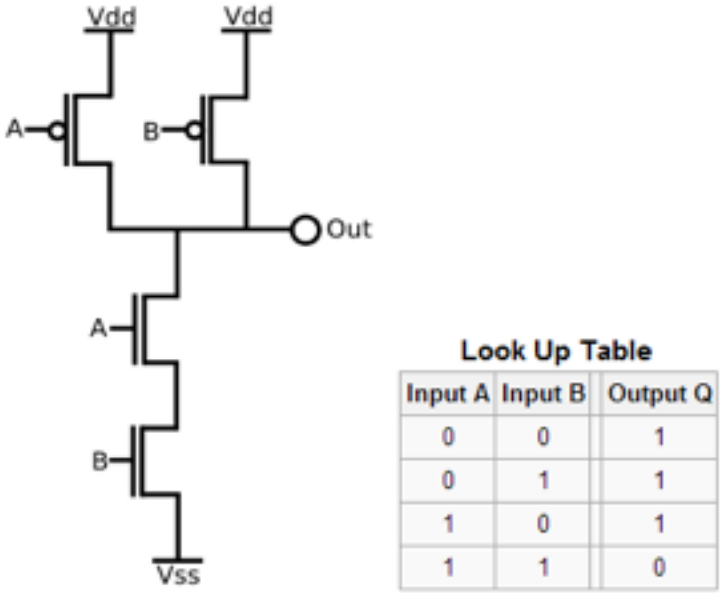


Figure 2 - Design of an FPGA [15]



Look Up Table

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0

ASIC **FPGA**

Figure 3 – The Operational Difference of ASICs and FPGAs [16]

An FPGA's ability to adapt to situations using its reconfigurable properties means that the hardware can be optimized based on the task at hand. These adaptive characteristics allow engineers to modify their designs (perhaps for optimizations or handling new protocols), or upload completely different designs onto a chip. For instance, an FPGA can be configured to be a router in one instance and a graphics processing unit in another. System architectures are classified with three parameters:

i. Components: the set of function made available by the hardware architecture
ii. Links: the interconnects between the components
iii. Procedures: the set of variations of functions and links in time

ASICs use a "fixed components, fixed links, variable procedures" architecture. While FPGAs on the other hand use a "fixed components, variable links, variable procedures" architecture [17]. This is also how our brains are designed, and despite running at an operating frequency of approximately 10 Hz, the human mind is the most advanced processing unit. This added variance allows us to optimize performance for various tasks, but also comes at a cost.

One question to be asking is why we still use ASICs if FPGAs have such a large advantage in being reconfigurable? It is important to note that although FPGAs and ASICs could theoretically be interchanged (for example ASIC designs are often implemented on FPGAs for prototyping and testing purposes), FPGAs will never replace

ASICs completely. This is because they serve different purposes and thus each has its own advantages, see Table 1 and 2:

Advantage	**Brief**
Faster time-to-market	No layout, masks or other manufacturing steps are needed
No upfront non-recurring expenses (NRE)	Costs typically associated with an ASIC design
Simpler design cycle	Due to software that handles much of the routing, placement, and timing
More predictable project cycle	Due to elimination of potential re-spins, wafer capacities, etc.
Field reprogramability	A new bitstream can be uploaded remotely

Table 1 – FPGA Design [17]

Advantage	**Brief**
Full custom capability	For design since device is manufactured to design specs
Lower unit costs	For very high volume designs
Smaller form factor	Since device is manufactured to design specs

Table 2 – ASIC Design [17]

The reconfigurable properties of FPGAs come at a cost however. FPGAs have a large hardware overhead in comparison with ASICs, thus increasing the size and cost of the units (see Figure 4). Additionally, since software handles most of the routing, placement, and timing within FPGAs, the layout is not the optimal solution (even though the software runs several optimization algorithms) [18]. So although this creates a simpler design cycle, ASICs have a performance advantage. As a consequence of not having a design implemented using the optimal layout, the clock frequency also has to be decreased according to the slowest path in the circuit [19].

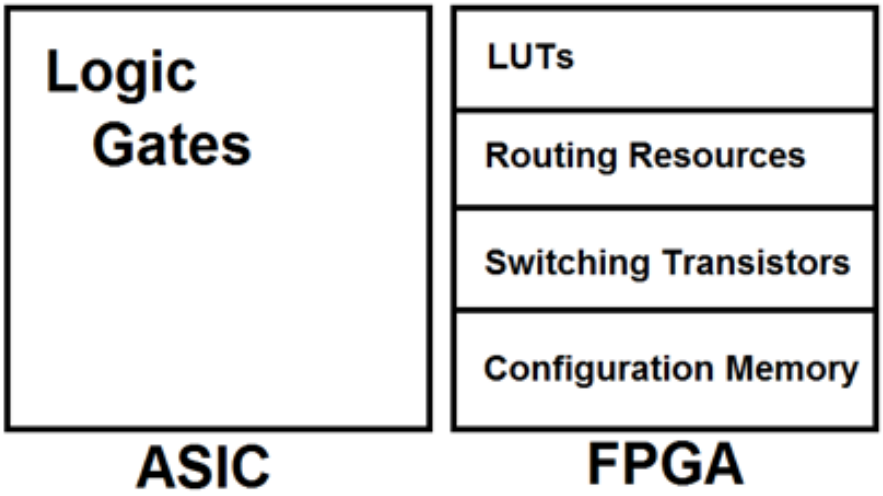


Figure 4 – FPGA Hardware Overhead

As the name states, ASICs are specific to an application and therefore are optimized for those task(s). Therefore, in systems where tasks are fixed, ASICs are the better solution. However, in situations where a systems' tasks are continually changing, a ASIC architectures may not be the ideal/optimized method of implementation. Despite the hardware overhead of an FPGA, the unique ability to be reconfigured is the reason why FPGAs are able to take on complex tasks with an optimal hardware configuration and outperform traditional processor architectures using less space and power consumption.

In the past, when a system [implemented on an FPGA] would switch tasks and require a different hardware configuration, this meant the FPGA had to be placed in shutdown mode while loading the new configuration file onto the chip (this is known as static reconfiguration). This limitation significantly impacts the performance of a system,

and hence the applications for which an FPGA's adaptive properties would be required. Engineers recognized this setback was restricting the performance capabilities that FPGAs had to offer. After several years of research, Xilinx discovered the solution: Dynamic Partial Reconfiguration.

2.2 FOUNDATION OF MODERN RESEARCH

2.2.1 Reconfigurable Technology

Dynamic Partial Reconfiguration takes the adaptive properties of an FPGA to a whole new level. DPR allows selected "parts" of an FPGA to be reprogrammed with new functionality while the remainder of the FPGA continues to operate (hence the word "partial" in dynamic partial reconfiguration). Observe Figure 3 bellow, "Part A" of the FPGA is going to be reconfigured. During this reconfiguration time, the module is shut down and the rest of the system continues under normal operation. When "Part A" has finished being reconfigured, it will turn on and resume operation with its new hardware configuration [20].

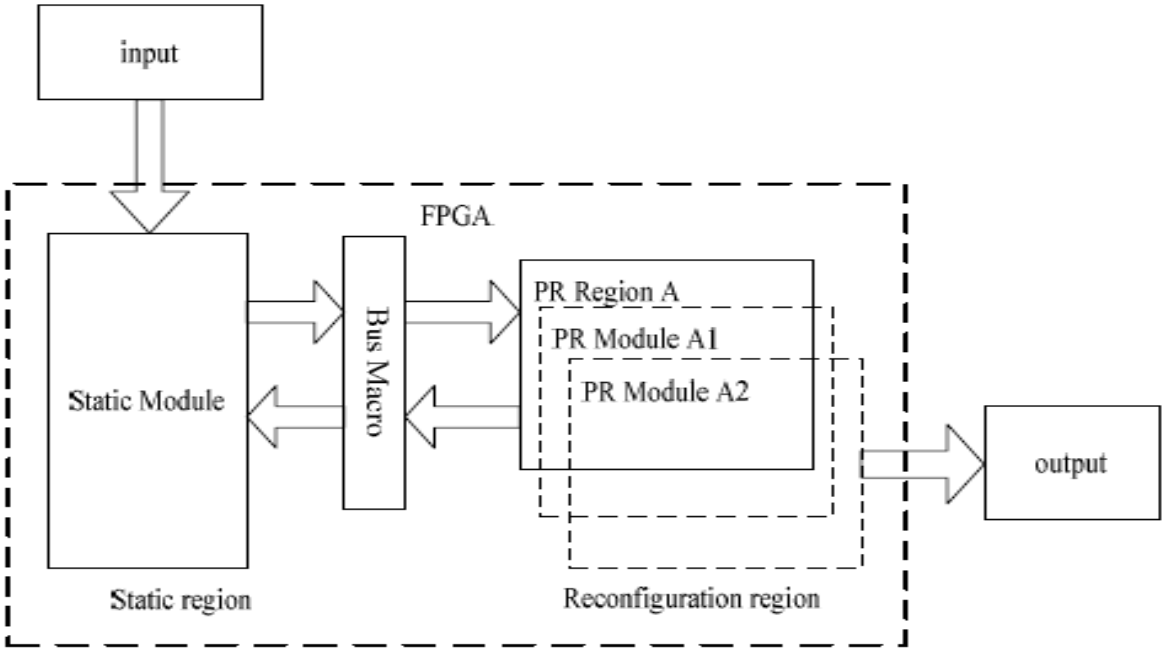


Figure 5 – Operation of Dynamic Partial Reconfiguration

Dynamic partial reconfiguration is not supported on all FPGAs. In fact, up until recently Altera (one of Xilinx's primary competitors) did not support partially reconfigurable FPGAs. The first FPGA to support DPR was Xilinx's Virtex II Pro. Partial reconfiguration addresses three fundamental needs by enabling the designer to:

1. Reduce cost and/or board space
2. Change a design in the field
3. Reduce power consumption

Dynamic partial reconfiguration enables designers to fit more logic into an existing device by time-multiplexing hardware dynamically on a single FPGA. This also translates into a smaller and less expensive device. The advantages of DPR are clearly seen when comparing Figures 6 and 7, which are two different implementations of the same hardware functionality [12].

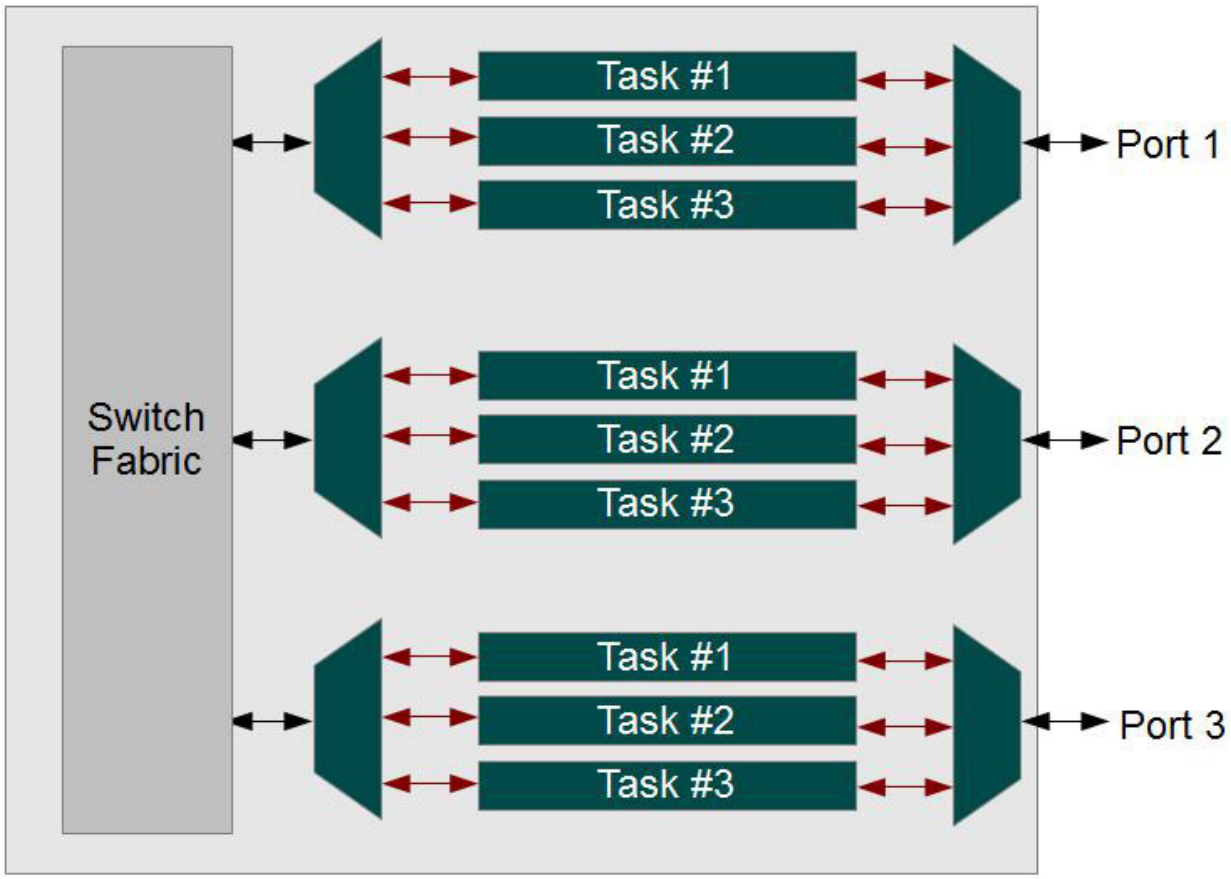


Figure 6 – Hardware Module without Partial Reconfiguration

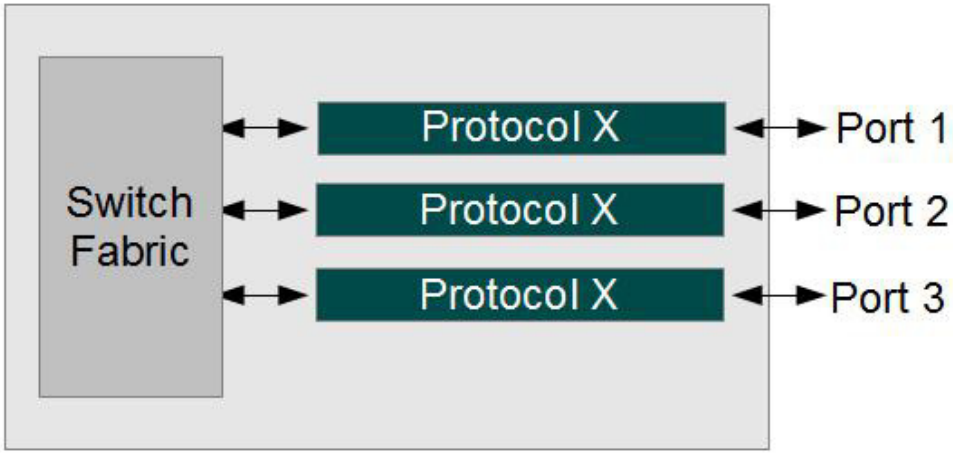


Figure 7 – Hardware Module with Partial Reconfiguration

DPR provides real-time flexibility for the protocols available in any given moment. This can translate into cost savings of several orders of magnitude compared to traditional processors, since the architecture cannot be reprogrammed to support a new protocol.

With partial reconfiguration, designers can time multiplex parts of their designs on the FPGA. As long as the stages of the design are independent, this can be done as many times as needed. Using this methodology then requires a much smaller FPGA chip since not every part of the design is needed 100% of the time. Using a smaller FPGA also means that the power consumption and timing (because everything is closer) is exponentially more optimal [21]. Additional advantages of partial reconfiguration include:

- Enables the use of new techniques in design security
- Improves FPGA fault tolerance
- Accelerates configurable computing
- Reduces bit stream storage requirements

2.2.2 Existing DPR Applications

Dynamic Partial Reconfiguration in FPGAs is still a fairly new concept that has not been taken advantage of fully. FPGAs that support DPR can be utilized in many applications to increase performance, while reducing costs, area consumption, and power consumption.

A hardware accelerator is an optimized functional block used to offload a specific task or set of tasks from a general purpose processor (GPP). Hardware accelerators are optimized frequently used in systems today to improve performance and decrease dynamic power consumption. However, according to Amdahl's Law hardware accelerating a task can only accelerate the overall performance according to how often that task is used during execution time. Figure 8 bellow further exemplifies this [22].

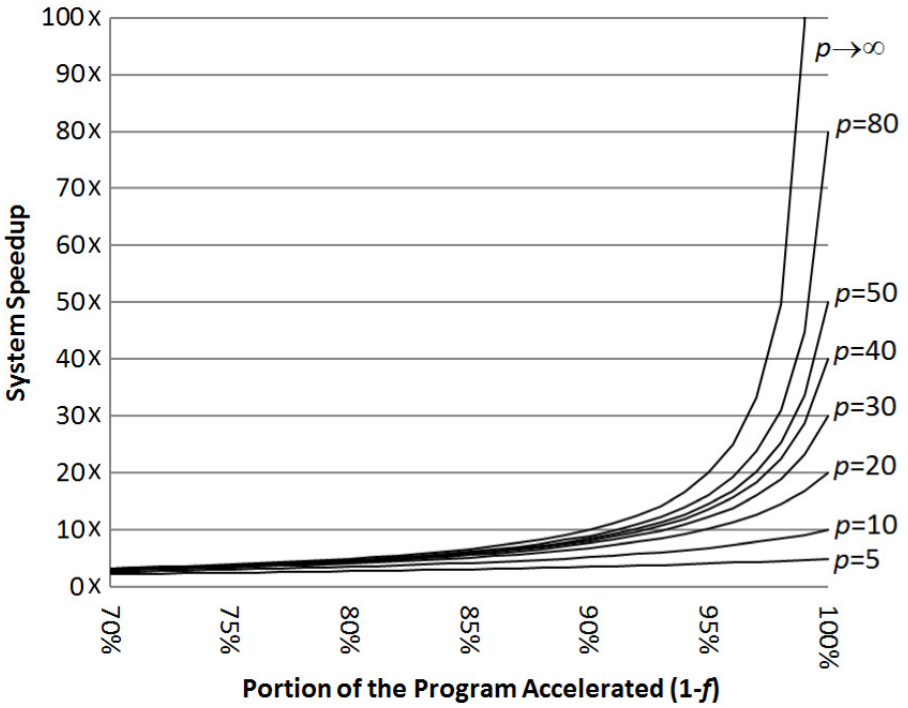


Figure 8 - Amdahl's Law: Speedup In Relation To Portion of Program Being Accelerated

Using dedicated hardware to accelerate a particular task has proven to be a very effective method for improving the performance of a system [23]. However, if every task had its own dedicated set of hardware the static power and area consumption would be astronomical. One may be able to see where DPR enabled systems can solve this problem, while maintaining the same performance benefits to the system. For instance, in most systems not all tasks are used in one instance. So if those tasks were loaded into configuration memory, then each module could be loaded onto chip (the FPGA) as needed. By following the suggested strategy being proposed here, only the tasks being used are loaded into hardware but the system still has access to all the dedicated hardware sets.

What [24] proposes is that hardware accelerators in the past can increase performance and reduce power consumption, but overlook the interface between CPU's and FPGA hardware accelerators. The model they present is to analyze the application running on the GPP (in this case a CPU), and profile the application to determine which time-critical functions should be accelerated. At this point the correct IP module will be loaded onto the FPGA. However, the way in which the CPU and the FPGA interface to each other will also affect the performance of the overall system. The results can be seen in Table 3 and will be further discussed below.

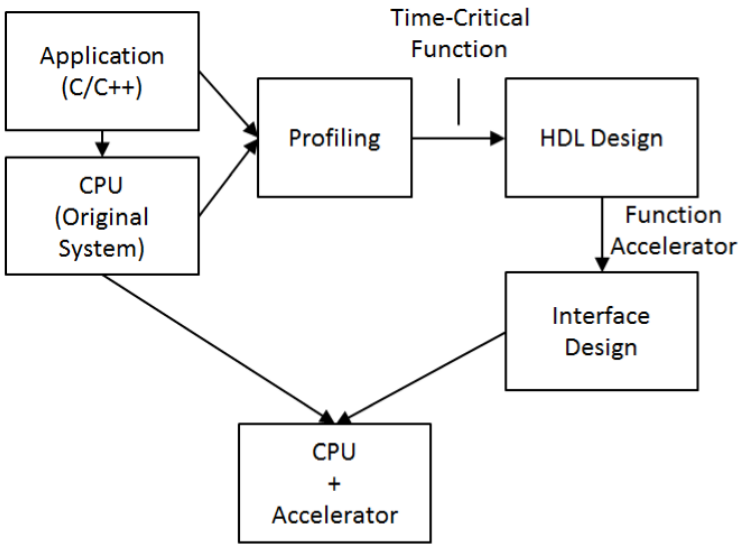


Figure 9 – Acceleration Design Flow **[24]**

The research and findings found in [25], compliments that of [24]. Using FPGAs to hardware accelerate embedded systems is a popular and promising area of research. The benefits and possibilities are promising and will truly expand the capabilities of present day embedded systems. The basic premise of [25] is that FPGAs in embedded systems can raise the abstraction level without imposing new tools and practices for design engineers and corporations. By taking an OOP approach (expressed in UML and implemented in C++) software coding efforts are significantly reduced. Not only provides a golden reference model, but may also be used in the actual implementation of the hardware design. This can prove especially useful in today's semiconductor industry. Since about 80% of the time needed to design an ASIC is spent in verification efforts, overlapping the time spent on developing Bus Function Models (BFMs) for the test bench and the actual RTL design could reduce the cost of processor design significantly [26]. For example, assuming 100% of the 30% design time could be

transferred to verification efforts that would mean a cost saving of 30%. The proposed methodology being indicated here is what top engineers are striving to achieve. However, what typically prevents them is the amount of resources and risk involved in making such drastic changes. With the approach discussed in this thesis and the tools discussed in Chapter one such results can be achieved. The results will be further discussed in the following section.

2.2.3 Theoretical DPR Gains

FPGAs are particularly suited for accelerating compute intensive algorithms that can take advantage of massive hardware parallelism. This is especially true for FPGAs that are DPR enabled, since the hardware can adapt to the exact compute operations being executed. Additionally, being able to reconfigure hardware during run time without stalling or impairing the performance of a system, makes this all the more promising. This in effect makes the hardware appear as if all tasks are in hardware and available to be called upon at any given time. The philosophy behind this appears to mimic that of virtual memory. In short, a system may only have a few megabytes of memory, but by virtualizing memory on external storage it appears to the user that the system memory is much greater.

This visualization of the hardware follows the same principles by context switching between hardware configurations. In multitasking contexts, virtualizing

hardware translates to superior hardware utilization and much greater performance can be achieved [27]. This is truly reflected in Table 3 bellow.

It can be seen that using a general processor architecture means the implementation must be completed using software. This abstraction means higher frequencies and hence power consumption. Using DPR to hardware accelerate this design proves to improve performance by factors greater than 50 times. The area in this design does increase; this is due to the initial hardware overhead that FPGAs introduce. However, as the design size increases, the initial investment of hardware overhead is disguised by the hardware virtualization capabilities.

FIR Version	CPU	Logic Cells	CPU @ 25MHz		CPU @ 50MHz		CPU @ 100MHz	
			Exec. Time (ms)*	Power (W)	Exec. Time (ms)*	Power (W)	Exec. Time (ms)*	Power (W)
SW	e	2795	276.80	0.06	138.40	0.11	69.20	0.19
	s	3768	156.00	0.08	78.00	0.13	39.00	0.25
	f	4309	125.00	0.08	62.50	0.14	31.25	0.25
Avalon	e	4781	4.24	0.07	2.12	0.11	1.06	0.20
	s	5850	1.68	0.08	0.84	0.13	0.42	0.25
	f	6390	1.36	0.08	0.68	0.14	0.34	0.25
Avalon (FIFO)	e	5087	4.25	0.07	2.12	0.11	1.06	0.20
	s	5967	1.68	0.08	0.84	0.14	0.42	0.25
	f	6498	1.36	0.08	0.68	0.14	0.34	0.26
CI	e	4720	3.32	0.07	1.66	0.11	0.83	0.20
	s	5660	1.32	0.08	0.66	0.13	0.33	0.25
	f	6189	1.32	0.08	0.66	0.13	0.33	0.25
TCM	f	6316	1.36	0.08	0.68	0.14	0.34	0.26
C2H	e	4992	44.40	0.07	22.20	0.11	11.10	0.21
	s	5895	18.22	0.08	9.11	0.14	4.56	0.25
	f	6452	14.64	0.08	7.32	0.14	3.66	0.26

* Execution Time for 1000 samples.

Table 3 - FIR Filter Evaluation **[24]**

Area cost with FPGAs is one of the biggest issues. The problem lies in what was discussed in the introduction and observed in Table 4. This is why existing FPGA solutions have not taken off in high volume design cases. As further discussed in [25], the capabilities to abstract hardware at a level where designers can overlap design and verifications efforts also come at a cost in the area designs consume. That being said, perhaps further research could uncover a more optimal method that will not increase the ASIC's area. Once again similar to the finding in [24], [25] has a small increase in hardware resources (area) required but performance increase of an order 10.

		all software	hard accelerated
Area	Slices	3,575	3,718
	BRAM	13	15
	DSP48	0	1
	PPC405	1	1
	DCM	1	1
Time (sec)	total	0.3831	0.0394
	Matrix::macc	0.1177	0.0224
	angleCalc::exec	0.2469	0.0082
Freq (Mhz)	of tests	300	PPC: 300, hard: 100
	max	300	PPC: 300, hard: 166

Table 4 – Hardware Accelerated Embedded Systems Comparison **[25]**

We have discussed and observed situations where FPGAs increase the area being consumed and yet other scenarios that decrease the area being consumed. The designer must truly consider the goals, meaning what the constraints and optimizations for the given project are. Once these functional specifications are gathered, the designer can analyze the technical specifications of the system and the resources available.

The conclusion of [23] supports the findings of this thesis, one can see that hardware accelerators can (but don't always) improve throughput and lower power consumption. When approaching the problem by proposing re-ordering the requests put on the accelerators and thereby decreasing the associated overhead with the DPR enabled system. What my work establishes and what [23] was working towards, is that an accelerator must be able to maintain performance between context switching, so by reducing this overhead and general switching occurrences, performance can be improved. Implementation results for a number of applications can be seen in table 5 bellow showing how such a strategy can improve performance:

Accelerator	Context Size in IBM WSP (byte)	Throughput Improvement (%)
Decompression	2500	26.7
XML	256	15.8
RegX	192	9.9
Cryptography	64	4.0

Table 5 - Hardware Accelerated Embedded Systems Comparison **[23]**

The designs he has implemented are larger than the ones found in previous papers, and provide a much better estimate as the FPGA overhead is better disguised. As seen the throughput has once again been improved (although not as much), but the area differences are equal or less than in the FPGA implementation. This is why the designer needs to follow careful methodology to determine the correct resources to implement the design on.

2.3 EXISTING METHODS & TECHNIQUES

The industry recognizes the powerful benefits of reprogrammable logic and FPGA-like resources. Many leading industry and academic research and development sources have concluded the benefits of converging the two worlds of processor and FPGA technology. However, thus far implemented systems and proposed architectures have failed to meet the objective goals found in this paper.

Due to a large granularity approach, such systems cannot strategically extract benefits of each technology. The cost, risk, and performance suffers for consumer related processing applications. Reprogrammable logic resources will always consume more area and be slower than ASIC logic, however it is also true that they consume less power and are faster at processing algorithms than processors and even DSPs. It is for this reason, the fusion of these technologies needs to follow a specific set of rules in order to extract the benefits of each technology and develop the innovative consumer processing architecture of tomorrow.

Below are examples of how researchers have approached this concept and what some semiconductor manufacturers have created.

2.3.1 Intel Develops FPGA-Based Coprocessors

Intel is one of the largest semiconductor manufacturers in the world. They have been a world leader in developing innovative processor technologies. Given their success they have invested significant resources into the research of architecture design improvements.

Intel has made progress in developing processors that make use of FPGA embedded technology [28]. As can be observed in the figure below, the approach is one that provides greater potential and flexibility, excluding the consumer market CPUs/GPUs. As mentioned in chapter 2, FPGAs makes the hardware design process far more affordable for companies to develop systems. However, what must be highlighted is how significant an impact FGPAs have had on accelerating the innovation within the hardware industry.

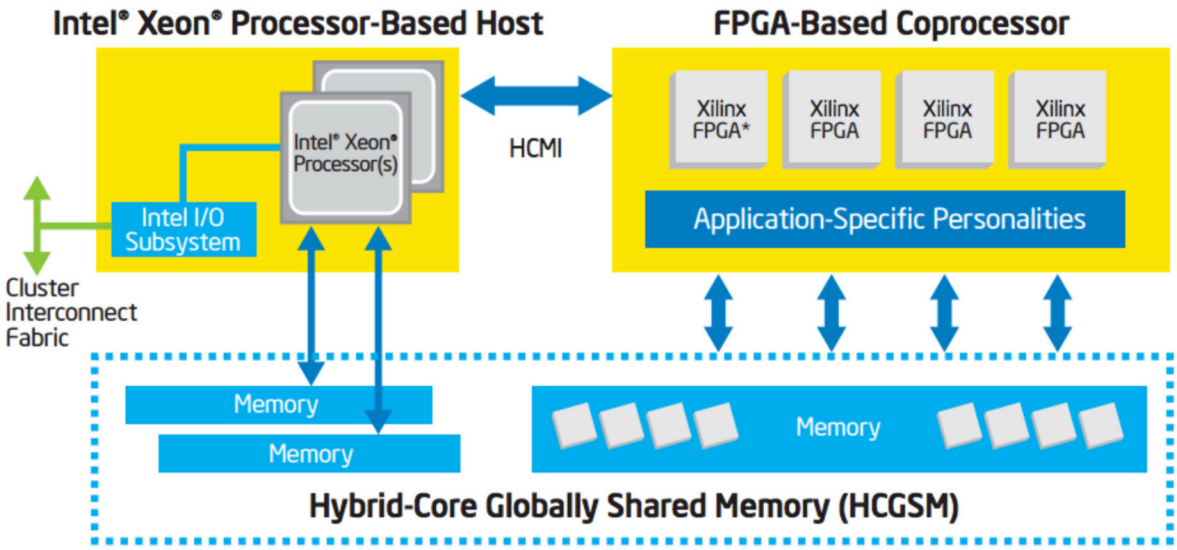


Figure 10 - The Coprocessor Approach [28]

Hardware is extremely expensive to manufacture using traditional semiconductor fabrication plants. The cost is so astronomical that unless millions of chips are being fabricated, it is typically not worthwhile or feasible to invest in exploring hardware solutions. With the introduction of FPGAs however, this all changed. Today developing a hardware module is an affordable option for fuelling growth in the hardware industry, through the enablement of small projects, small ventures and custom solutions. Moreover, as demonstrated in [29] FPGA fabrication technology and architecture advancements have further enabled engineers endeavours and expanded the realm of possibility. The compute capabilities in today's FPGAs are tremendous, depending on applications certainly surpassing the performance of other technologies as seen below.

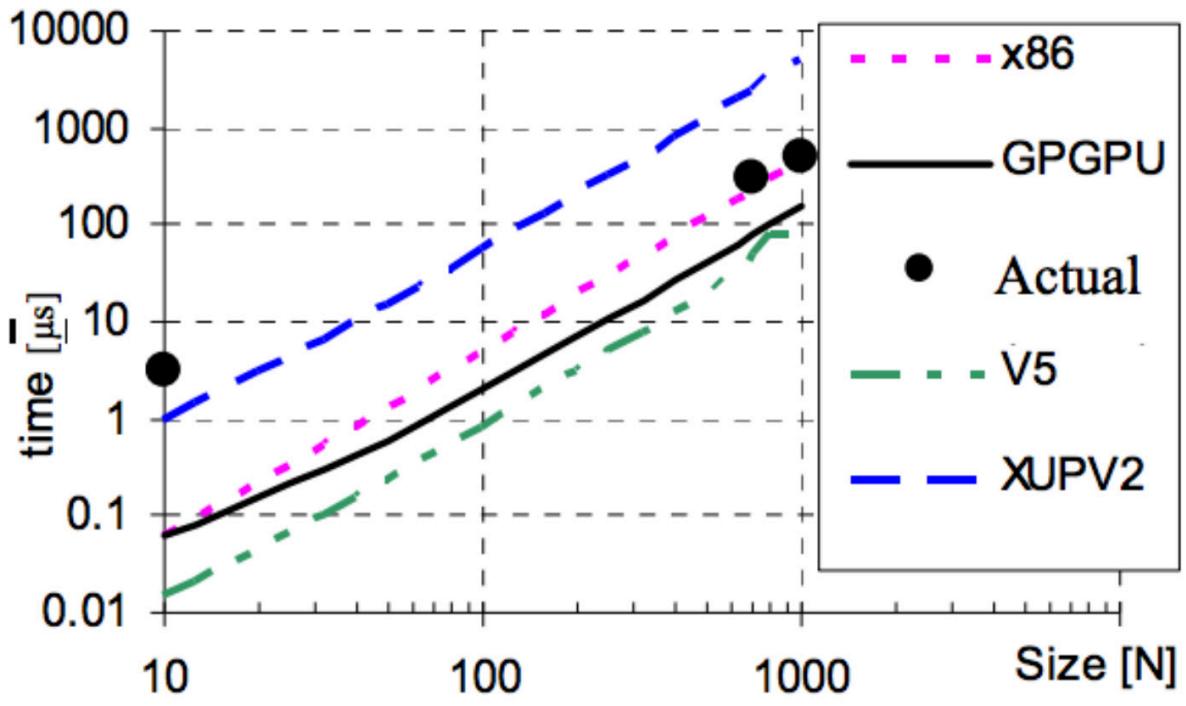


Figure 11 - Multi-Architecture Performance Comparison [29]

2.3.2 Existing Mixed Chips

It is commonly known that processors (including CPUs and GPUs) contain microcontroller(s) within their design. This embedded controller provides flexibility to a processor for particular set of tasks and frequently improves performance through measured analysis of a hardware pipeline. This can show up in area reductions, power reductions and other key performance metrics.

Heterogeneous compute devices such as this are become extremely popular, so much so in fact that the Heterogeneous System Architecture (HSA) Foundation was established in 2012 to focus on such efforts [30]. The HSA foundation is a not-for-profit

that develops industry standards to improve innovative efforts of heterogeneous computing devices. Unfortunately, thus far such efforts have primarily focused on large processor systems working together. As found in [31] other heterogeneous computing devices are emerging, the Intel Many Integrated Core (MIC) and the AMD Fusion technology. [31] is able to unlock additional performance using the microcontrollers found in GPUs. As seen in the figure below, the methods assist with the reduction of data transfer times through a method titled “Microcontroller-based data transfer”.

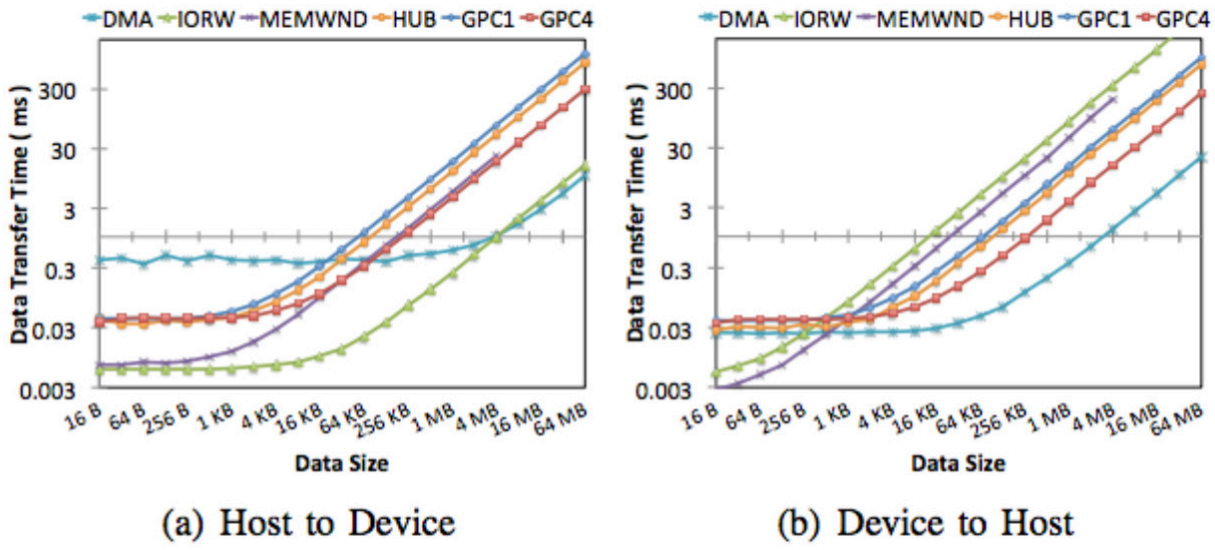


Figure 12 - Data Transfer Time Analysis

Altera and Xilinx have released the new generation FPD-Microprocessor mixed chips that consist of microcontrollers and programmable logic, though these ICs are aimed at embedded system markets only and provide limited amount of processing power compared to high-end processors. This idea has not been explored at high-end CPU/GPU chipset market. Solutions (such as netbooks, tablets, mobile devices and

embedded systems) that are power, cost, and space sensitive will benefit tremendously from the solution that is proposed here.

By introducing reconfigurable FPGA blocks within ASICs, systems would have far greater flexibility to the large main chips in order to satisfy the dynamic feature requirements of different O.E.Ms and system architects. The solution (embedding non-volatile programmable logic blocks) promises cost reduction, flexibility, performance enhancements, and size/power reduction from system engineering perspective. Such is the proposal of this thesis.

2.3.3 Embedding ARM Processors Into FPGAs

Semiconductor companies are making the opposite efforts to integrate traditional processor technologies (ARM processors) into their systems. Xilinx has been extremely successful at building processing systems that offer a number of benefits to the hardware development industry. Recall from previous chapters, FPGAs have enabled hardware companies and developers to perform research and development affordably. It was explained how this lead to improved innovation. To expand the level of system flexibility Xilinx has set a high priority towards integrating both technologies. Today's Xilinx Zynq systems are a testimony to the incredible performance and opportunities that these systems possess [32].

Xilinx Corporation markets the Zynq platform as an “All Programmable SOC”; this is an accurate term to describe the flexibility of the system platform. Developers are able to provide instructions through software, which can be run on the ARM core(s), while the programmable hardware can mange computation data processing. The system can execute applications in real time and optimize system interfaces through programmable I/O [33].

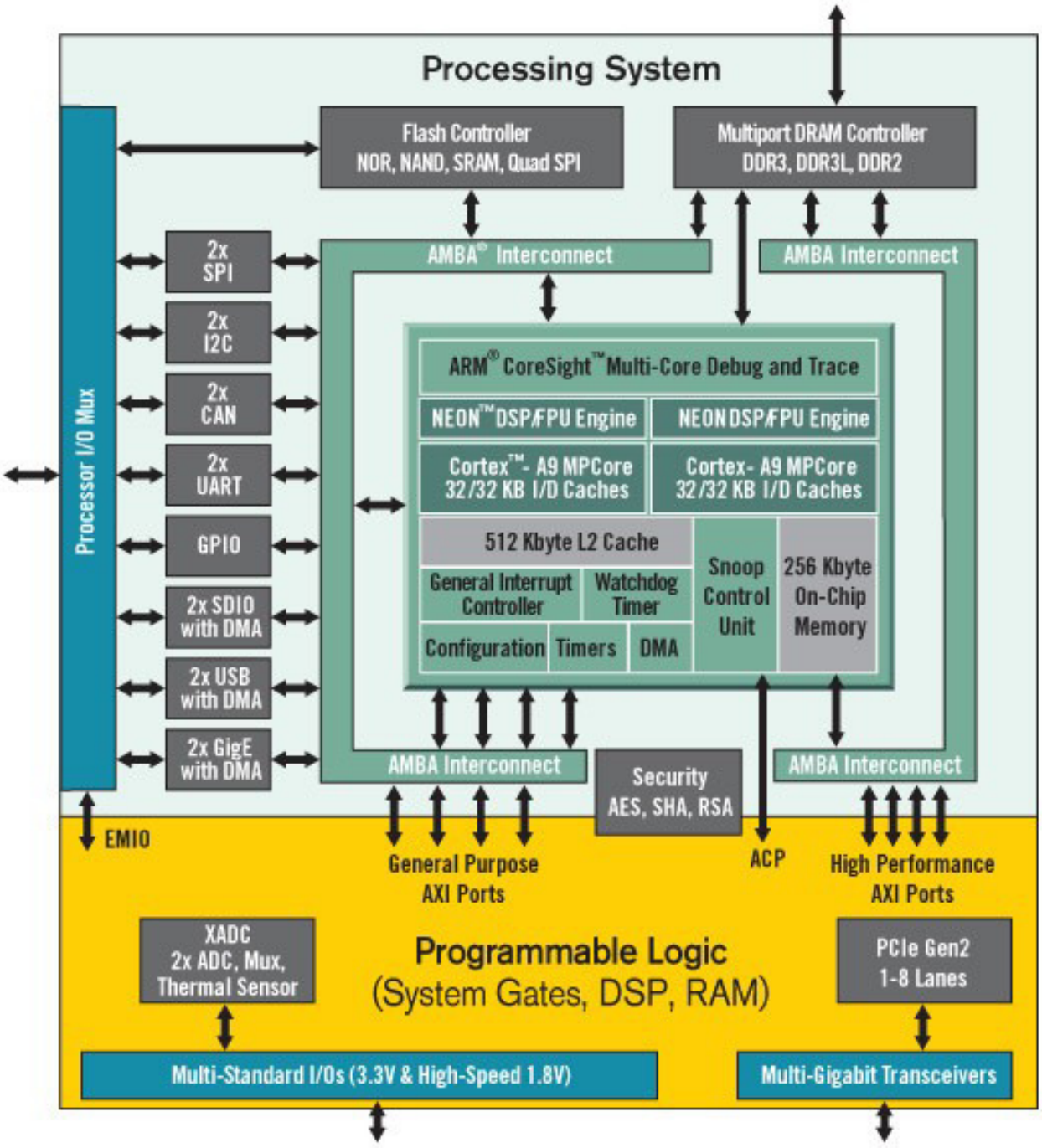


Figure 13 - Xilinx's All Programmable SOC [33]

CHAPTER III:

3.1 COMPARATIVE ANALYSIS OF EXITING SOLUTIONS

The problem solved in the proposed work is to improve the performance of existing main stream systems (CPUs, GPUs) by incorporating programmable embedded logic into each ASIC strategically. FPGAs are known for their performance in highly algorithmic contexts, which are especially common within GPUs. As computing takes on more complex algorithms, today's processors can benefit with the proposed design. It improves the overall performance by accelerating algorithms and offloading certain intensive CPU or GPU tasks. In an example used in section 4.1, we see an increase of performance of 0.5% at a system level and as much as 80% on a block level.

Due to the design strategy proposed here and the characteristics of the technology being used, the proposed solution would be a low risk endeavour that corporations could quickly adopt into upcoming technologies. Moving towards the proposed designs would be a step towards a future looking architecture that outperforms existing architectures. Additional benefits will be further discussed later.

The common execution processor stages (IF, ID, ED, MEM, WB) introduce high overhead in comparison with a stream processors, especially in terms of power and delay (see implementation results). Processors break up large tasks into small operations and take them on accordingly. This introduces hazards and high overheads,

in comparison to a hardware acceleration unit which reprogrammable logic could provide replicate within an ASIC.

3.1.1 Coprocessor Systems – FPGAs + CPUs/GPUs

As discussed above there are a number of semiconductor companies investing into solutions that integrate traditional processor technologies with FPGAs. This has been extremely effective at custom project solutions. As shown by [33], significant performance and flexibility can be achieved at low design costs. Xilinx brags about the lower BOM cost, higher systems performance, and lower system power they are able to provide with this type of architecture. Such an architecture is not far from the other methods proposed by Intel and Altera (at least in the system examples used here), however they have improved on the interface connections.

It should be understood, that the analysis of the pros and cons of this previous approach must be addressed in a context for which these systems were not designed. Therefore, it is not a criticism of the solution as it addresses a different problem all together. The objective of this thesis is, to the best of my knowledge conceptually different from all others.

When FPGAs and consumer processors (CPUs/GPUs) are connected to work together as coprocessors the system chip becomes very large. The cost of the chip significantly increases and large overhead IPs must be developed to manage the

connection. The two technologies stand alone in comparison with what this thesis proposes. Hence, the memory, contexts, tasks, and all other internal aspects of each technology are unknown to the other.

Moreover, due to the architecture the connections can become extremely slow in comparison the proposed design. Take for example the Altera system below.

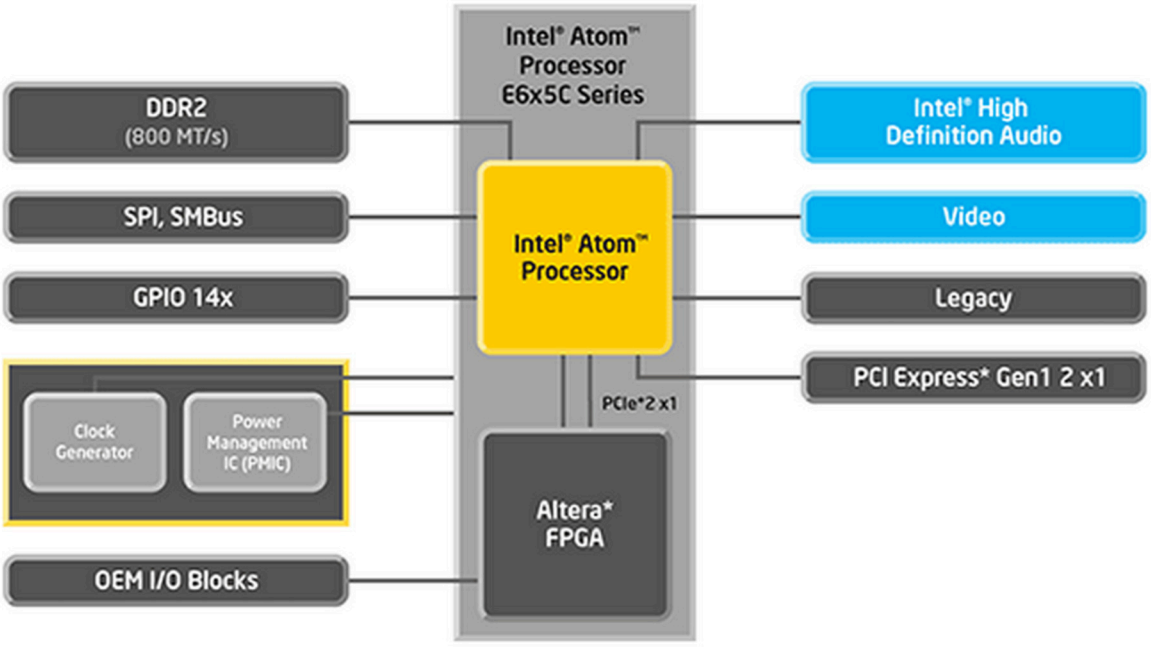


Figure 14 - Isolated Technologies, PCIE Connection [34]

The Altera FPGA and Atom processor are connected via PCIE. Such transfer speeds are considered to have extremely high latency. Communication is ineffective and it is desirable to avoid “team work” between the two technologies.

This isolation and lack of awareness are exactly what the “Hardware Acceleration Islands” and supporting strategies resolve. By removing this high latency connection,

and improving communication between the two technologies the new system can build upon the success of both CPUs/GPUs and FPGAs.

Moreover, the proposed designs accomplish something very unique to the hardware development industry. That is, that they do not require a new software to be developed within the consumer market. This system not only adds performance to existing CPUs and GPUs, but packages it in such a way that hides these changes from hardware and software layers above (with the exception of device drivers and potentially minimal operating system optimizations).

3.1.2 System Customization

The suggested design does not have as much customization and flexibility for after the market adjustments. Since this system is targeting consumer devices too much flexibility will start to slow down the system and loose the initial benefits purposed. That said, consumer CPUs and GPUs hardware design companies have incredibly large and robust systems that require lengthy development cycles. To ensure functionality, time is taken to verify each IP and final SOCs (see section 2.2). Therefore, it can be expected that such projects will not / should not add greater customization than necessary and according to the strategies outlined in this thesis.

The customization that is provided according to this work will be nearly or just as effective as the isolated task unit in the alternative hardware solutions. However, when

considering the context of the system the Hardware Acceleration Islands will be far more performant.

3.1.3 Hardware Acceleration

By embedding FPGAs into current processors (such as CPUs and GPUs), algorithms and tasks can be offloaded to the DPR enabled programmable logic.

The programmable logic can be modified using run time reconfiguration and a bit stream uploaded characterized by the context of the system during run-time without impacting the performance of the rest of the system. Moreover, due to the granularity of these embedded reprogrammable modules and the stream processing characteristics of FPGAs the performance benefits and flexibility is precisely what engineers are looking to achieve.

Looking at the system implementation example in the following chapter, it is evident that engineers could benefit by embedding an adaptive hardware acceleration unit. Moreover, when an FPGA logic block is introduced for hardware acceleration, depending on the interfaces, there may be opportunities to assist with varying tasks and secondary system requirements. While the FPGA may not be the most effective at routing and decision-making, there are areas within system processing where FPGAs would far surpass (such highly mathematical system contexts).

3.1.4 Protocols & Flashing

Custom SOCs are a growing market within the hardware design industry. The ability to add levels of flexibility and custom protocol support is a sought after trait. While it is costly, the return on investment is clearly justifiable.

There are a number of blocks within existing systems that have high demands in terms of the sheer number of protocols they are required to support. While some systems can afford to use software, most solutions require a hardware dedicated data path solution. Supporting a protocol is no small task; these dedicated pipelines are designed for each protocol which incoming data from the block interface will traverse.

The lists below are just basic protocols and codecs that many modern systems need to support. There are several others including a number of other categories that are not included here:

- H263
- H264
- VC1 Advanced
- VC1 Main (RCV)
- MPEG2 (DXVA)
- MPEG2 (Native)

- MPEG4

By using the proposed HWAI unit, not only could engineers target these with less hardware overhead but the complexity of the hardware would be far less. This would result in lower research and development costs and verification efforts. This is because by using a HWAI unit, the solutions become straightforward and simple problems that are less prone to errors. Currently engineers are faced with needing to come up with optimization strategies that, while ingenious, can be quite complex.

HWAI allows live system reprogramming. Moreover, new protocols could be downloaded post-silicon to consumer devices. Those bit streams can be implemented within the programmable logic constraints. This not only provides far-superior flexibility but also customizability. Hardware/software developers can create their own modules (ex. Apple/Intel's Thunderbolt protocol) without requiring extremely expensive hardware re-designs. The alternative for this today is to use slow software support.

3.1.5 Custom SOC

There is a growing demand for custom SOCs. As the use cases have grown and costs become difficult to manage, semi-conductor corporations are looking to build SOCs with greater and simpler customizability. ARM has been a leader in this regard; their success over these last years speaks for itself. ARM offers great performance using simple designs that are highly customizable.

The flexibility that reconfigurable systems offer is considered highly advantageous and powerful in today's industry. The reconfigurable nature of these blocks would allow CPU designers to create a new type of custom SOC that enables customers to flash their own configurations and modules. In specific, the internal FPGA could be flashed at any time, allowing for native hardware support for any task that can fit the logic cells available on the reprogrammable logic.

Using reconfigurable logic creates a unique opportunity to provide incredible flexibility while improving system performance. The simplicity and effectiveness of such a design truly stands out from existing custom IP solutions. The market this would attract and cost saving advantages that this design would provide to customers would be a unique and powerful combination.

3.1.6 Power Advantages

Currently semi-conductor corporations are investing a large amount of resources into power management. As an industry engineers are approaching this by developing new power efficient designs. To further those efforts, HWAI would allow design team to make use of embedded reprogrammable logic to implement power management strategies using a central logic unit.

As an added benefit for the custom SOC use case, customers can define and flash their own power templates.

Today, hardware accelerators inside processors are fixed. It is obvious that if one could decrease the execution time, the overall power consumption decreases. Therefore, by hardware accelerating portions of applications (such as an edge detection algorithm within a GPU) engineers can reduce power consumption. Consider the following general relationships:

if ↑ performance **by** ↑ frequency, **then** power consumption ↑

if ↑ performance **by** dedicated hardware, **then** power consumption ↓

if ↑ area, **then** ↑ cost

if ↑ area, **then** strong chance ↑ power consumption

3.1.7 Life Time

Given a system with such adaptive properties, as described in this work, design teams could reconfigure processors that have already been taped-out. This feature could be used to add dedicated hardware support for a new protocol that might be needed down the road, or to improve the performance of a task, power options and other customizations where allowable through the use of HWAI.

3.2 MODIFYING THE APPROACH

There are three parts that are necessary when considering the proposed design found in this thesis. Each aspect must be calculated and assessed carefully in order to achieve the objective goals.

The name of the proposed design work (as described in my patent filings) is ”Hardware Acceleration Islands” or sometimes abbreviated as HWAI.

In the past, corporations researching this area have only looked at involving FPGAs and ASICs through methods that are insufficient for the consumer market (as discussed in the previous chapter). As an example, simply stitching these large architectures together through a traditional bus produced high latency and did not allow for cooperation without significant overhead.

While this approach was necessary for the past, this will not be the case in the near future. Previous strategies were required because the technologies (transistor gate sizes) greatly differed. CPUs and GPUs are continuously leading the way making use of the latest transistor technology. FPGAs have remained behind on the advancements being made (due to unique issues that must be

overcome). However, that gap is shrinking as time passes and this approach becomes even more alluring.

The reason for this gap shrink is in the fact that current transistor technology is reaching its limitations. There are only a few atoms across the gate of a transistor. By decreasing it any further quantum physics becomes a factor affecting the behaviour of the transistor. Due to this hurdle and the increased number of resources being invested in recent years toward FPGAs, FPGA technologies have nearly closed the gap.

Therefore the previous strategy will no longer be necessary in order to avoid compromising. Moreover, the previous strategy contains 1 major flaw. FPGAs will never take the place of traditional processor architectures, and the proposed products have far too much overhead in the way they attempt to fuse the two technologies together. Rather than simply stitching these two technologies together I propose a far more strategic approach that will harness the benefits of both architecture individually. GPP systems currently dominate the market. The process of stitching an FPGA to a GPP costs a lot of money and the use reprogrammable use cases are limited in a large granularity implementation. This means high cost, high risk, and low gain. Alternatively, with my proposal, engineers take existing processor design and analyze the architectures strategically with a set of rules and strategies to identify locations to embed

HWAI. This is not a big task from a research and development point of view. During a design process there are design problems that simply highlight the benefits of the FPGA architecture. Similarly, most CPUs and GPUs already have microcontrollers embedded inside to help with regard to specific problems that would be solved more optimally by a microcontroller.

In recent years there have been a number of systems developed that contain a stitched CPU and FPGA architecture. The hardware design corporation leading the way in this regard is, Xilinx with their Zynq-7000 All Programmable SOC. The All Programmable SOC such as the Zynq-7000 has its place, however, the work discussed below targets hardware developers who are looking for innovative solutions within consumer processors. As they serve to improve the flexibility of modern day processors through a low cost, low risk approach that would be simple to implement across the industry. Providing a number of benefits that will be discussed below.

Previous to the Zynq-7000, Intel released the E600 series to harness the power of FPGAs, but they also simply adjoined an FPGA to an existing processor. This is not the strategy being proposing. Rather than trying to reinvent a new architecture, it is clear that ASICs are the superior architecture for CPU and GPU technologies. However, the proposed work compliments traditional processor architectures by harnessing the power of each technology in calculated locations.

There are a number of key benefits that this will offer to current semi-conductor design teams. The following sections are dedicated to the approach and benefits analysis of HWAI and what it offers.

3.2.1 Hardware Acceleration Islands

By embedding HWAI into current ASIC systems (such as CPUs and GPUs), algorithms and tasks can be offloaded to the DPR enabled programmable logic. The key for the success of this is to follow the rules closely.

The programmable logic can be modified using run time reconfiguration and a bit stream uploaded characterized by the context of the system during run-time without impacting the performance of the rest of the system. Moreover, due to the granularity of these embedded reprogrammable modules and the stream processing characteristics of FPGAs the performance benefits and flexibility is precisely what engineers are looking to achieve.

The rules to create system performance improvements are as follows:

1) Identify system blocks that:
    a. Contain logic which are infrequently active or infrequent enough that the a bit stream of comparable logic can be uploaded within comparable or better timing. These are especially common in multimedia and power logic.

b. Implement complex algorithms that do not require mainly interfaces/variables to be executed and do not have dependencies and are not atomic in nature.
c. Frequently are modified during custom SOC projects or that would benefit performance and effective system lifetime through adaptable logic.

2) Target the smallest division of logic. Remember bit stream sizes increase exponentially to the size of the design.
3) Determine routing overhead from FPGA to ensure the size will not push chip outside of specifications.
4) Compare the performance of the existing logic to a model of an optimized model (ideally developed as a streaming unit).
5) Power saving and cost benefits will be proportional to the decrease in frequency requirements, saved leakage current, area reduction, the simplification of the design (design, verification, research and development), and other factors.

Following this analysis and justification, the development is extremely simplified and the programmable logic can be embedded accordingly to replace existing blocks.

As previously suggested, a particular example where HWAI would be extremely beneficial would be within media blocks (see section 3.1.4). See the figure below and compare with Figures 6 and 7, focusing on the “Video Decoder / Encoder” block.

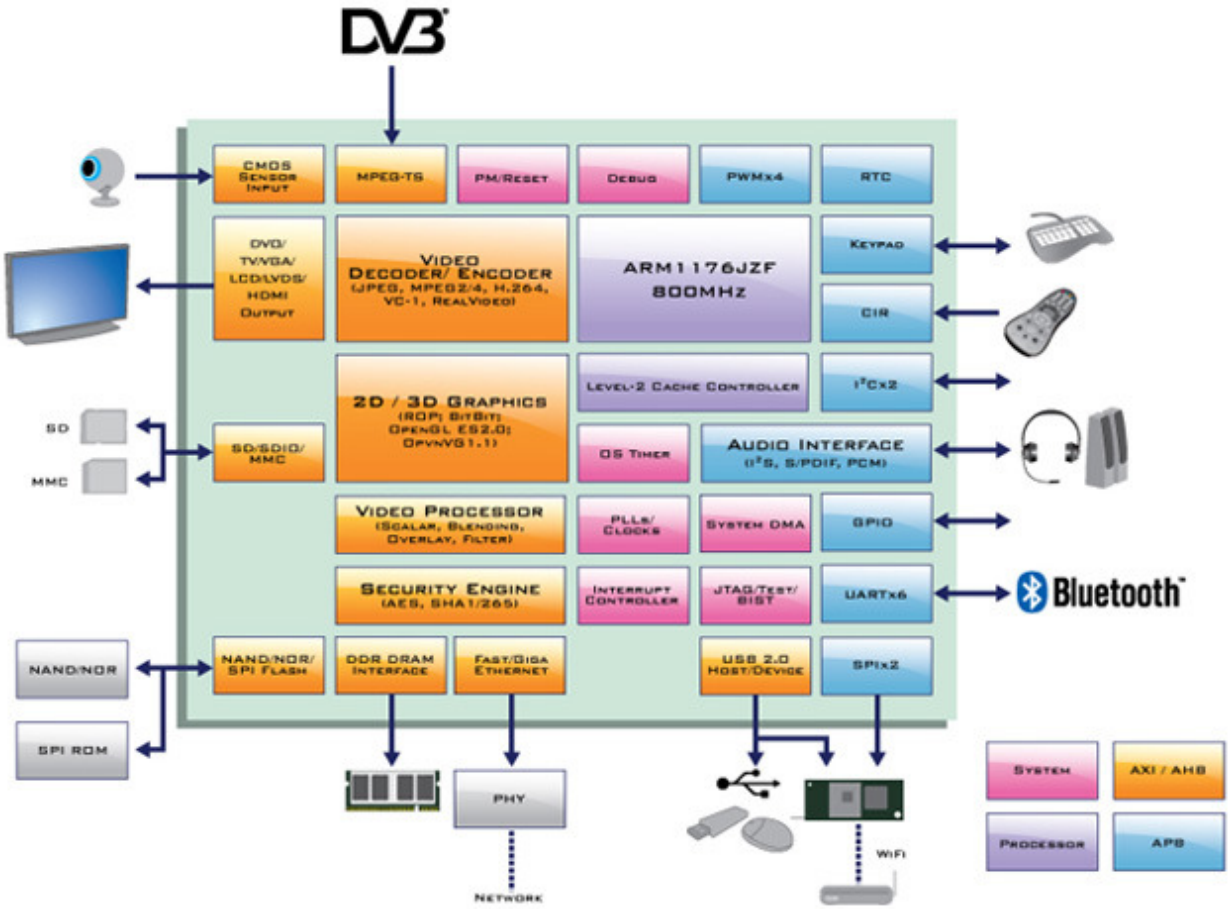


Figure 15 - Multimedia System with Block Diagram [35]

An alternative method to the above would be based on a bus system structure (see figure below). This method would be suitable for high performance computing. Today corporations such as NVIDIA have modified their processor designs to optimize for high performance computing systems. This is extremely expensive and has cause them to take a very reserved approach driven primarily by the following:

$$Justification = (Additional\ Cost\ Per\ Unit \times \#\ of\ Units\ Sold) - (Estimated\ \#\ of\ Additional\ Units\ Sold * Profits\ Per\ Unit)$$

While this does not account for indirect benefits (especially for marketing), in general as long as “Justification” is greater than 0 the project is justifiable.

In a CPU or GPU that follows a bus architecture, adding multiple Hardware Acceleration Islands (with a calculated number of LUTs) to the bus could provide dynamic support to the system.

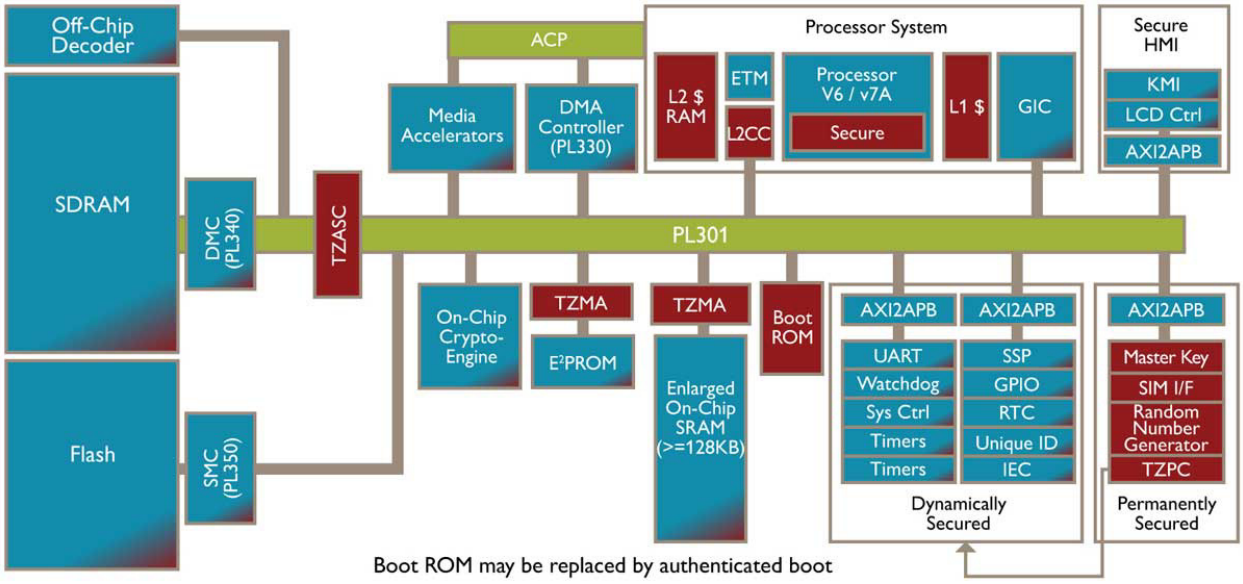


Figure 16 - BUS System Structure [36]

3.2.2 Internal Connections

The physical connections of HWAI to the system must be considered. Depending on the contextual placement of the programmable logic block and the support the unit will provide will impact how the connections will be established.

The control signals and data signals will have to remain consistent to duplicate the functionality. However, the method in which the module reprogrammability can be implemented will require a set of additional control signals. See the figure below:

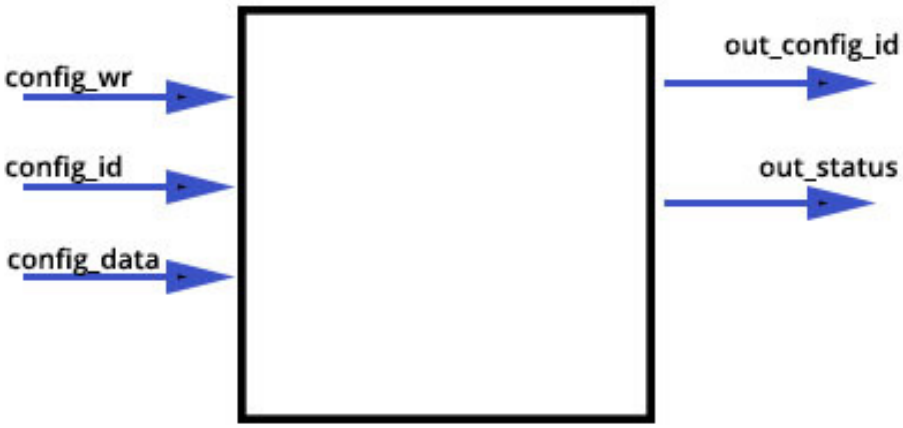


Figure 17 - Required Control Signals for HWAI

config_wr: Write enable for configuration bit stream.

config_id: *optional* by implementing cash into the block or nearby to be shared amongst other blocks (possibly other HWAI units), instead of transmitting the entire bit stream over a bus the id could be sent and the bit stream could be uploaded far faster.

config_data: The configuration bit stream data.

out_config_id: This signal can identify to other blocks what the current configuration inside the block is. This will be helpful for the system to determine the state of the block but also to determine if something is hung or there is a problem (both pre-silicon and live system, post-silicon).

out_status: Determine the internal state of the block.

Other control signals may be necessary or used to optimize the block.

In section 2.3.2, mixed chips were discussed as being an existing technology that conceptually looked at multiple chip designs into one final system. Semiconductor companies that design CPUs and GPUs already have a set of signals and associated strategies to incorporating microcontrollers into these processors. While this information is proprietary it can be expected that this information would be helpful in strategically designing hardware reprogrammable mixed chips architectures.

3.2.3 Pre-emptive Hardware Adaptation

The work in this section focuses on is providing simple adoption for the marketplace in the semiconductor industry and a feasible solution that does not interfere with current software applications. These are what successful inventions in the consumer market require to make industry adoption a feasible outcome. This objective is achieved by combining HWAI and the pre-emptive hardware adaptation strategy.

The “Hardware Acceleration Islands” was introduced in the previous section. However, adding hardware functions would be irrelevant if software cannot easily make use of the features. Ideally the hardware drivers will take care of this such that developers of operating systems and software applications don’t have to create separate code. User-friendly hardware is critical to success.

The work found in this section is titled “Pre-emptive Hardware Adaptation”, and is intended to compliment HWAI. It is helpful to note that although this work was intended for use in conjunction with HWAI, it can be applied to other hardware acceleration processes.

By taking advantage of embedded logic in existing processors (CPUs/GPUs), software drivers can pre-emptively prepare embedded hardware accelerators. FPGAs are known to shine in highly algorithmic contexts, which are especially common within GPUs. As computing takes on more complex algorithms a static hardware solution becomes difficult when trying to keep performance, power, area, and cost in balance.

In the system implementation example, we were forced to make two negative assumptions on the performance of the hardware accelerators. These assumptions can be eliminated through this work thus leading to a performance gain of as much as 0.1% on a system level and 7.5% on a block level. For reference purposes the assumptions made were as follows:

1) We will assume worst case scenario such that the DPR enabled hardware acceleration unit must be reconfigured every single time and that the same complex task is never invoked after each other."
2) "The time to reconfigure is an overhead of 3 c.c's"

However, what if we could remove these or at least minimize these similar to a compiler; that is what the "pre-emptive adaptive hardware" work accomplishes.

Unfortunately a compiler has the advantage of having all the instructions available, thereby optimizing previous to run time. In the case of hardware, we do not have this benefit. Moreover, there is an extremely high performance demand. What we do have available in hardware is an instruction buffer. Instructions are typically fetched / placed inside of a ring buffer (as seen in the figure below) before being assigned and executed.

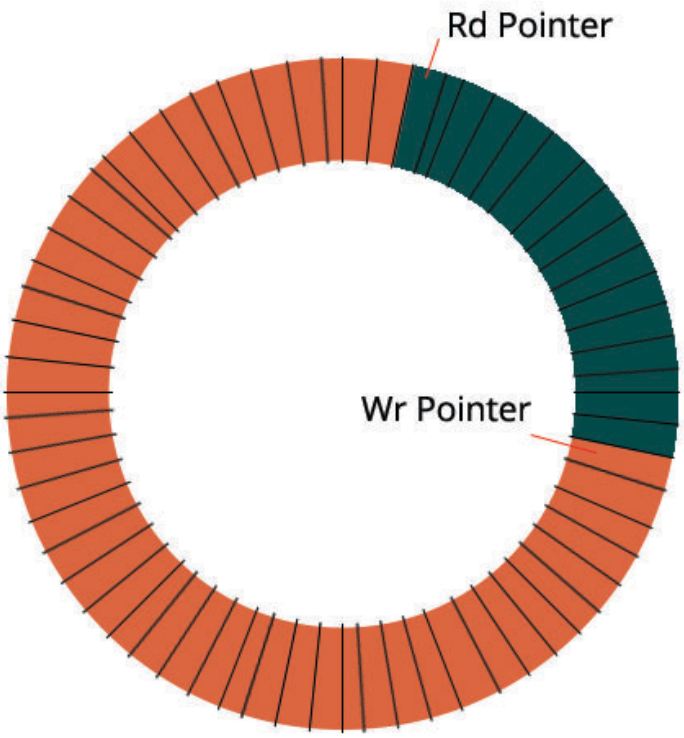


Figure 18 - Ring Buffer / Instruction Buffer

In order to achieve this additional acceleration and minimize delay we can do basic analysis of the packets coming and make simple actions to improve performance. Through my research I have been able to determine a feasible way of accomplishing this for existing and upcoming hardware systems. The proposed work here can work in conjunction with HWAI [2] to provide an even greater performance boost.

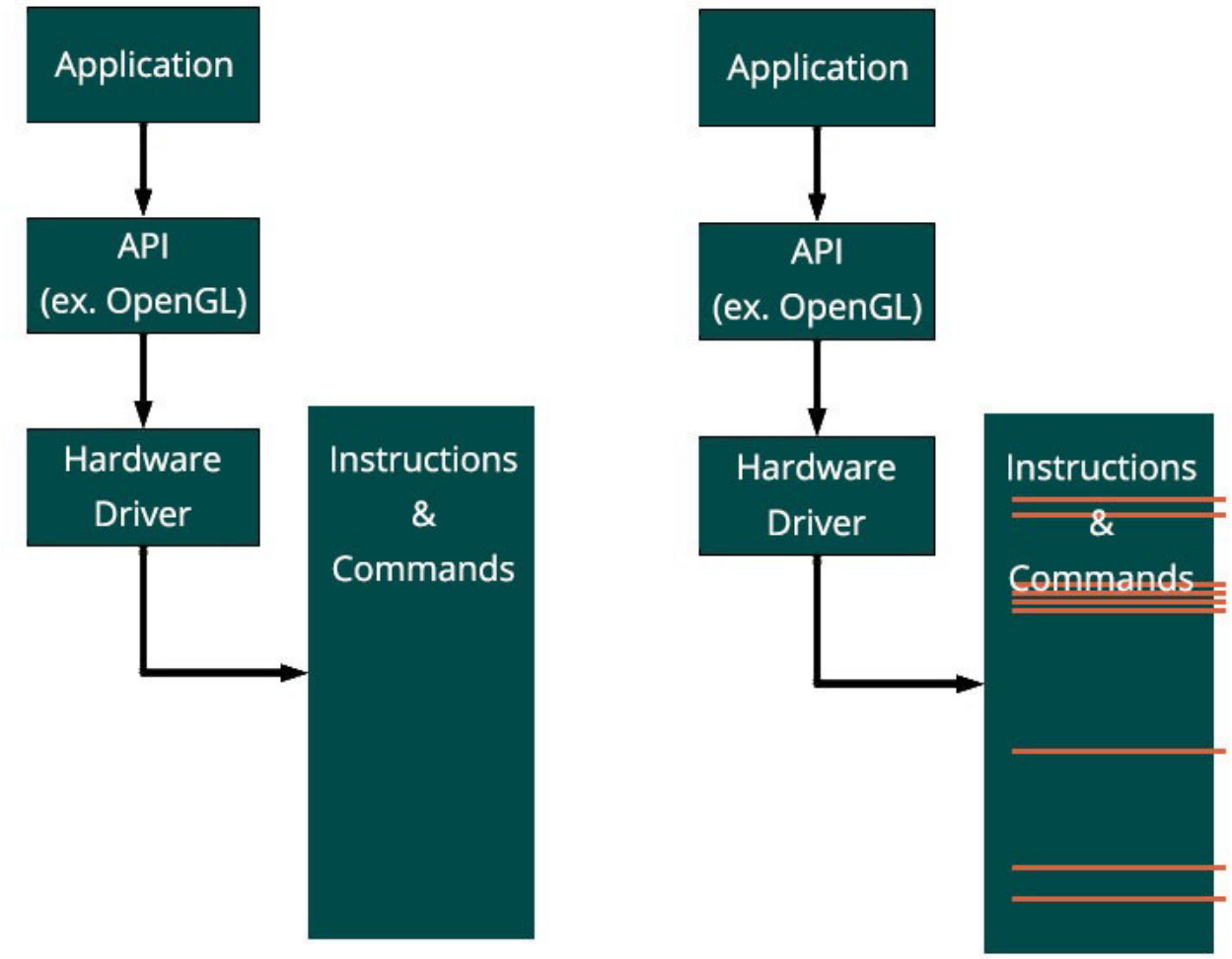


Figure 19 - Pre-emptive Adaptive Hardware

The traditional instruction set (on the left) can easily be modified using hardware drivers that interface with the hardware and the higher-level applications. The instruction set with pre-emptive hardware acceleration is seen on the right. Orange instructions are automatically inserted for pre-emptive preparation of hardware acceleration units. The implementation strategy behind this will be discussed below.

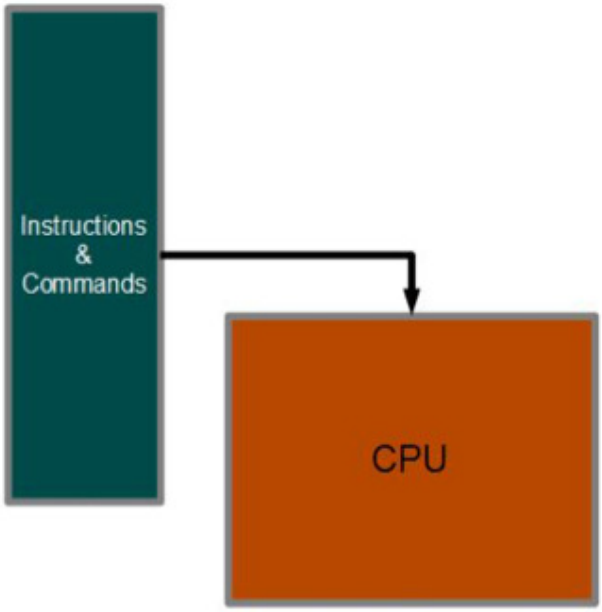


Figure 20 - Traditional System Architecture Overview

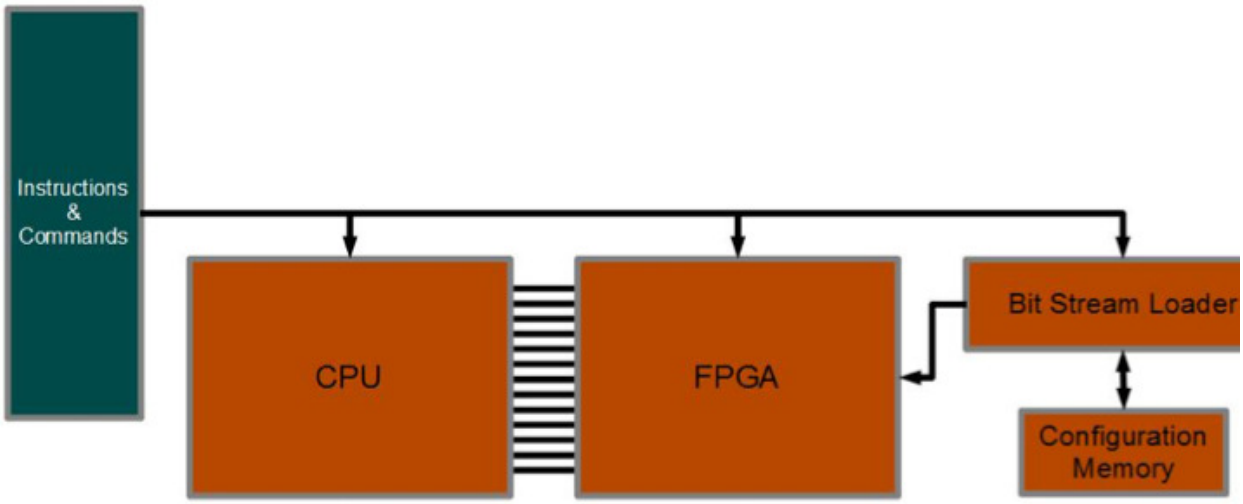


Figure 21 - Programmable SOC (ex Intel's E600 series)

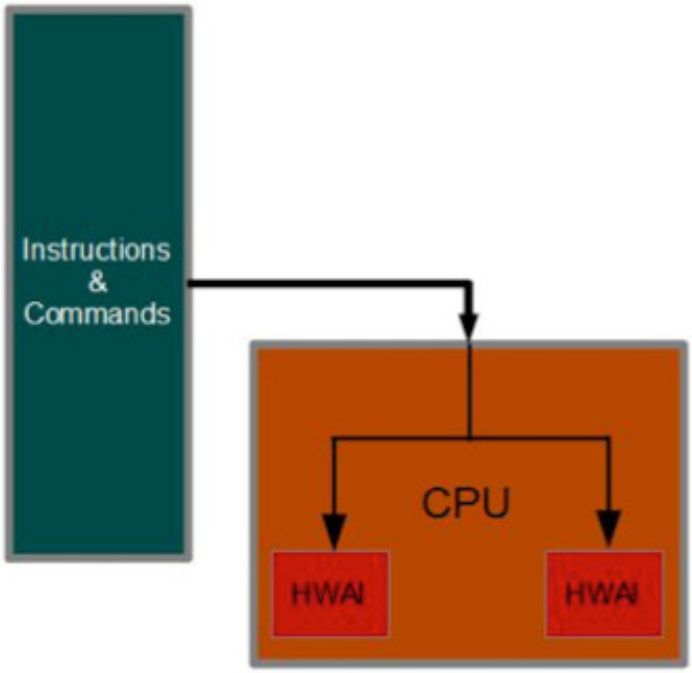


Figure 22 - CPU with HWAI

By modifying low-level hardware drivers, engineers can further improve hardware acceleration by preparing HWAI with the correct bit stream configuration before the task and data arrives. This abstracts hardware accelerators from programmers, which traditionally required separate instructions to create the same result. Many existing applications can take advantage of this feature and there is no need to change existing programs.

For example, when developing applications that need to perform graphics operations, developers make use of APIs such as OpenGL or DirectX to communicate with a GPU. However, between the hardware layer and these API's exists low-level drivers for the GPU. This standardized interface between the drivers and API's enables developers to make adjustments without affecting the interfaces between the hardware

and software. We can determine the exact hardware accelerator required for each call and pre-emptively load/prepare a hardware accelerator.

An alternative method of solving the objectives of the “Pre-emptive Adaptive Hardware” would be to introduce hardware overhead. If the system analyzes the instructions id, it can determine whether or not it would be worthwhile to make use of a hardware accelerator. Given a set of instructions that require hardware acceleration, data inside the instruction buffer will be reviewed to determine if this particular instruction fits into the set of predetermined instructions that can be optimized. These HWAI bit stream configurations will be available and uploaded to the hardware when the required conditions are met. As per the implementation, a small microcontroller is suggested should the software approach be insufficient. The microcontroller, will make the respective fetches and manage the preparation of the HWAI units. For power savings efforts, interrupts can activate the microcontroller.

Another note is that while operating system changes could lead to even greater performance gains, it is not necessary for the implementation and success of this work. This lines up with the objective goals set for this thesis.

CHAPTER IV:

4.1 Demonstrating Performance

4.1.1 Reconfigurable Hardware Benefits

Accelerating computationally intensive algorithms with custom hardware is an important area of application and one in which FPGAs really stand out. Many applications in image processing inherently have high parallelism demands. FPGAs have shown very high performance in spite of their low operating frequency by fully extracting the parallelism. This can be achieved through the featured adaptive procedures that were presented earlier. By operating at lower clock frequencies but increasing throughput, a system can operate with decreased power consumption.

When using an ASIC, which as the name indicates is application specific, to take on multiple applications we run into complexities and overheads that can impact the system design negatively. In this case we start to see technologies such as FPGAs really stand out.

With the power of reconfigurable logic, the system is able to dynamically modify/re-configure its' hardware to meet the needs of the system for the task at hand. This removes the hardware overhead that is required because of the static nature of ASICs. The hardware savings can improve the block latency and

power consumption. More importantly, the performance increase will improve execution time thus allowing for lower operating frequencies. This translates into even greater power saving opportunities.

To demonstrate some of the benefits, see the following example:

$$Y = \sum_{i=1}^{n} [(a_i + b_i)^2 + (c_i + d_i)^2]$$

In a traditional processor we would require approximately 14 instructions to complete the task above. Each instruction will have an fetch, decode, execute, memory, and write back stage. Moreover, the stages may have hazards that further delay the task. Assuming each instruction took 1 c.c and each multiply took 32 c.c's, the execution time per loop would be 76 c.c's. We can use booth multipliers to accelerate the task. Assuming 2 c.c's per multiply, we can bring the execution time to 16 c.c's.

However, using a DPR enabled hardware acceleration island (HWA), we could create a dedicated hardware pipeline (as seen in the figure bellow) during run-time. We can approximate the run time as follows:

$$Total\ Execution\ Time = 5 + (n - 1) * 2c.c$$

See Figure 23 below to understand what the dedicated hardware pipeline would look like.

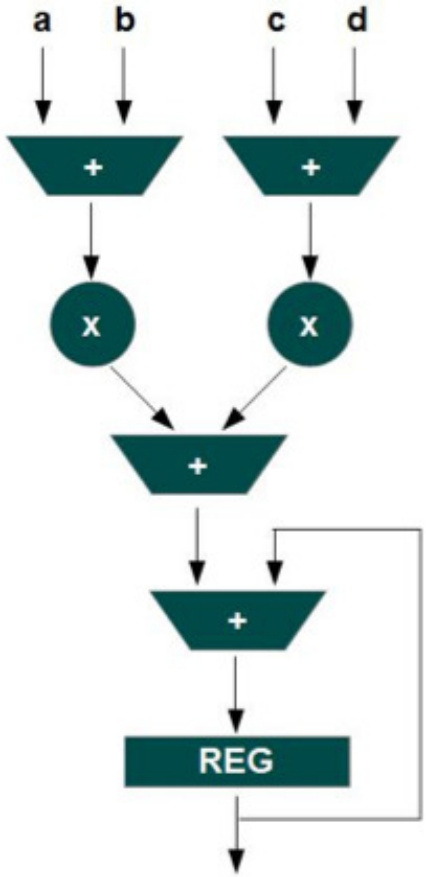


Figure 23 - Stream Processing Unit

4.1.2 Improved Application Performance

Traditional hardware accelerators are fixed components that are task/application specific. They consume minimal power, area, and offer superior performance. However, due to their fixed nature they cannot adapt to processor contexts, this is a key factor within the context of this thesis. Due to hardcoded nature of processors, it is challenging to satisfy the dynamic feature requirements of system developers in post-silicon. While

ASIC hardware acceleration units have their place and are not to be replaced, there are contexts where a reconfigurable hardware acceleration unit would be of greater use. To provide an example for which such context would be beneficial, we can consider the following scenario:

(1) A process consists of three hundred instructions including 10 core complex tasks
(2) There are 3 hardware platforms:
    a. A CPU without any hardware acceleration units
    b. A CPU with 4 ASIC hardware acceleration units
    c. A CPU with 3 ASIC hardware acceleration units and a single reconfigurable hardware acceleration unit
(3) Assume a simple instruction takes 1 clock cycle (or c.c.)
(4) Assume the core complex tasks each take 25 times longer than a typical instruction in the program (ie. 25 c.c.).
(5) If there are 10 complex tasks, let tasks 1-7 occur three times as often as tasks 8-10.

Therefore it would make sense that in our second hardware model to target our hardware accelerators toward the first 7 tasks. However, we are only able to improve 4 due the resource limitations. The speedup factor is 5 times (meaning the task now only takes 5 c.c's). In contrast, with the reconfigurable module we are able to target the remaining tasks. The time to reconfigure is an overhead of 3 c.c's and the speedup factor is slightly less such that the task takes 10 c.c's. In the first system, there is no

speedup however, the system will require less area. That means lower costs. And while this system would also save power without performance constraints, in order to compete with the performance of the 2nd and 3rd hardware platforms the frequency would need to be much higher resulting in far greater power consumption. From a performance perspective assuming tasks 8-10 occurred once within the 300 instructions and each of the 300 instructions, aside from the 24 tasks (7 complex tasks * 3 occurrences each + 3 complex tasks * 1 occurrence each), consume 1 c.c., we see the following performance:

System 1 = (300 instructions - 24 complex tasks) * 1 c.c + 24 complex tasks * 25 c.c.

= 276 c.c. + 600 c.c.

= 876 c.c.

System 2 = (300 instructions - 24 complex tasks) * 1 c.c

+ 12 accelerated complex tasks * 5 c.c

+ 12 complex tasks * 25 c.c.

= 276 c.c. + 60 c.c. + 300 c.c.

= 636 c.c.

We will assume worst-case scenario such that the DPR enabled hardware acceleration unit must be reconfigured every single time and that the same complex task is never invoked after each other.

System 3 = (300 instructions - 24 complex tasks) * 1 c.c

+ 9 accelerated complex tasks * 5 c.c

+ 12 dpr accelerated complex tasks * 3 c.c. to upload bit stream

+ 12 dpr accelerated complex tasks * 10 c.c.

= 276 c.c. + 45 c.c. + 36 c.c. + 120 c.c.

= 477 c.c.

Performance improvement:

System 2 —> System 1

= 876 / 636

= 1.377

System 3 —> System 1

= 876 / 477

= 1.836

System 3 —> System 2

= 636 / 477

= 1.333

This is a significant improvement in this context that System 3 was able to achieve. System 3 is an example of a system enabled with my proposed implementation. The work found in [2] and [3], enables this to be achieved and as shown in the following is an easy interface for both hardware and software developers to easily integrate this into

existing software without any change aside from the hardware driver and into upcoming hardware using the proposed strategy found here. However, this performance increase is not always true for every case, let us change one variable. Let us make the number of instruction 30,000 instead of 300.

System 1 = (30,000 instructions - 24 cmplx tasks) * 1 c.c + 24 cmplx tasks * 25 c.c.

= 29,976 c.c. + 600 c.c.

= 30,576 c.c.

System 2 = (30,000 instructions - 24 complex tasks) * 1 c.c

+ 12 accelerated complex tasks * 5 c.c

+ 12 complex tasks * 25 c.c.

= 29,976 c.c. + 60 c.c. + 300 c.c.

= 30,336 c.c.

We will assume the worst-case scenario such that the DPR enabled hardware acceleration unit must be reconfigured every single time and that the same complex task is never invoked after each other.

System 3 = (30,000 instructions - 24 complex tasks) * 1 c.c

+ 9 accelerated complex tasks * 5 c.c

+ 12 dpr accelerated complex tasks * 3 c.c. to upload bit stream

+ 12 dpr accelerated complex tasks * 10 c.c.

= 29,976 c.c. + 45 c.c. + 36 c.c. + 120 c.c.

= 30,177 c.c.

Performance improvement:

System 2 —> System 1

= 30,576 / 30,336

= 1.00791139241

System 3 —> System 1

= 30,576 / 30,177

= 1.01322199026

System 3 —> System 2

= 30,336 / 30,177

= 1.00526891341

In this case what we see is far less of a performance increase factor. To make a conclusion as to the correct system one must know what the constraints are or what one would choose to optimize for (area, power, performance, etc.).

Finally, consider this assume the area overhead of the CPU is 100 AU and each accelerator requires an additional 20 AU, in this case the area overhead is quite large for including these accelerators. However, if the CPU overhead is 10,000 then 20 AU

looks far more appealing. As a successful engineer all variables must be considered in order to build the appropriate system.

As depicted in the example above, in a system that takes on such a variety of complex algorithms it is impossible to hardware accelerate every single one using old design strategies. This can be solved using DPR enabled embedded logic.

The Field programmable logic blocks are in-system programmable by nature (unlike traditional accelerator logic components). This characteristic is not apparent in existing architectures. The HWAI design solves this problem with a superior architecture design. This flexibility can have various benefits during different stages of the IC lifetime including, smaller die size, less power, more versatility.

4.1.3 Modern Design Technique

As previously discussed, there a number of design and development tools being used in the industry today. The code found in the appendix contains the most modern languages and tools, including SystemVerilog and UVM. The modules developed can be implemented on an FPGA device or turned into an ASIC design. The compiler and tools will translate the design according to the requirements. This is one of the reasons that semiconductor companies can adopt HWAI without intensive research and development work.

The work also demonstrates verification principles that can simulate the full functionality of the modules. The test bench implements “constrained random verification” and provides easy support for the development of new tests using the Universal Verification Methodology.

4.2 Automated System Adaptation Capabilities

While adaptive hardware is a foreign concept to many hardware engineers, it is well known amongst those acquainted with FPGA technologies. As reconfigurable embedded technologies continue to gain momentum in the industry, the academic world has taken a number of extra steps in innovating even further. We introduced the concept of DPR and how Xilinx FPGAs even allow for run time hardware re-configuration. What cutting edge research and development teams (both in academia and industry) are currently looking at are self-adapting systems.

The demand for this exists in highly constrained systems. Typically constraints within regular consumer products are more targets and goals for marketing purposes. However, when we begin to introduce more significant constraints and advanced systems we require methodologies that are up to the challenges set by these leading technological needs.

The concept of self-adapting systems has empowered engineers to overcome many obstacles that technology has faced for years. Self-adapting systems expand the possibilities for which engineers can now successfully design.

Referring back to section 2.1.4 and 4.1.2, one is able to derive two categories of self-adapting systems: “Adaptive Procedures” (System 2) and “Adaptive Links & Procedures” (System 3). Within a processor system which has HWAI, as per the given rule sets and strategies proposed in previous chapters, such a system is primarily procedurally adaptive. In contrast, an FPGA is capable of link and procedure adaptation.

4.2.1 Manually Calculated Optimization Strategy

To demonstrate applications for self-adaptive systems it is important to lay a foundation of the algorithms and theory behind this methodology. To demonstrate the theorems a low pass FIR (Finite Impulse Response) Filter will be designed.

The FIR filter being designed was first invented eight years ago. This first implementation used 9 tap coefficients. This filter was designed for audio applications to filter out any frequencies above 20 kHz. Due to the structure of FIR filters, they can be modified to perform other filter functions simply by changing the coefficients. This ability to easily be modified is a huge advantage to system designers. Through the use of the design methodology presented in [37], the filter will be optimized to minimize area while meeting the performance and power constraints.

It is important to note that the specifications of the FPGA that this design would be implemented on as this will affect the resources available, potential constraints, and how the system will operate (for example the number of inputs on multiplexers inside the FPGA). Therefore, for the sake of this example we will consider the design to be implemented using the Xilinx Spartan 3E FPGA.

4.2.2 Specifications of System

To design the FIR filter, the signal processing toolbox was used. The FDA tool was given the following parameters: (1) Order of 8, (2) Sampling Frequency of 96000 Hz, and (3) Cut-off Frequency of 20,000 Hz. With these parameters a low pass filter was generated. The simulations yielded the following results: 8th order filter that was separated into four 2nd order blocks. The equation of the blocks is listed below in table 6. From the table below it is clear that there are 11 multiplications, 8 Additions and 8 Memory Location. Making use of the results from the simulation of that filter, we can obtain a more detailed set of specifications for our design. Since the cost of memory is significantly less than the cost of the functional units, the system transfer function was expanded and the delay weights from this expansion are listed in Table 7. This reduced the cost of the system to 8 Multiplications, 9 Additions and 9 storage locations

Block Number	Block Transfer Function
1	$\frac{z^2 - 28.83z + 97.99}{z^2} \cdot 0.0199$
2	$\frac{z^2 - 0.294z + 0.01}{z^2}$
3	$\frac{0.25z^2 - 0.5z + 0.85}{z^2}$
4	$\frac{0.25z^2 - 0.5z + 0.85}{z^2}$

Table 6 - 2nd Order Equation of Filter Blocks

Delay	**Weight**
z^0	0.1244
z^{-1}	-3.125
z^{--2}	-0.5
z^{-3}	28.12
z^{-4}	50.74
z^{-5}	28.12
z^{-6}	-0.5017
z^{-7}	-3.131
z^{-8}	0.1219

Table 7 - Transfer function Expansion Results

To evaluate the effectiveness of each variant it is necessary to evaluate the performance of the system using the following benchmarks: (1) Total Execution Time, (2) Area consumption and (3) Power consumption. To get some kind of figure of merit, the execution time is extracted after the system generates a schedule for the operations. For ease of reading the operations are assigned a variable that will be placed in the table below. Using Figure 24 below, the table was developed to provide a list of the operations and their assigned variable identifiers. It is important to note that

the system does not employ any advanced pipelining mechanism so the system will only be able to perform one operation per functional unit every twelve clock cycles (as per the performance limitations of the platform being considered).

Operation	$R0 \cdot a_1$	$R1 \cdot a_2$	$R2 \cdot a_3$	$R3 \cdot a_4$	$R4 \cdot a_5$	$R5 \cdot a_6$	$R6 \cdot a_7$	$R7 \cdot a_8$	$R8 \cdot a_9$
Variable	OP_1	OP_2	OP_3	OP_4	OP_5	OP_6	OP_7	OP_8	OP_9
Operation	$OP_2 + OP_3$	$OP_4 + OP_5$	$OP_6 + OP_7$	$OP_8 + OP_9$	$OP_1 + OP_{10}$	$OP_{11} + OP_{12}$	$OP_{13} + OP_{15}$	$OP_{16} + OP_{14}$	
Variable	OP_{10}	OP_{11}	OP_{12}	OP_{13}	OP_{14}	OP_{15}	OP_{16}	OP_{17}	

Table 8 - Operation Variable Assignment

Modelling the Area consumption of the system is a rather simple process. Since the number of each resource is already know, the equation is presented below, equation 1. The largest drawback of this model is that the system does not account for the area consumption of the interconnects. From this equation it is rather simple process to define the power. This is due to the fact that the power is linearly proportional to the CLB cost; the relationship is defined in equation (2) below. Please note that the power consumption is dependant of the clock frequency where the Area is multiplied by 9.6 to 10.4 μW.

$$\boldsymbol{A = (\#\ of\ Multipliers) * 102 + (\#\ of\ Adders) * 111} \quad \textbf{(1)}$$

$$\boldsymbol{P = \sum A(R_i) * P(f_{clk})} \quad \textbf{(2)}$$

4.2.3 System Sequencing Graph

Prior to the optimization process of our design, the systems' resource constraints must be determined. To obtain these specifications, an unconstrained Sequencing graph was generated. From the SG (sequencing graph) observed in the figure below, it is clear that the maximum number of adders (before total redundancy) that will affect the performance of the system is 4. Since the left most operation can be performed in T1 or T2, the maximum number of necessary multipliers is 8. By further analyzing the figure, it is clear that two multiplications are necessary for every initial addiction so the resource constrains should maintain a 2:1 ration between multiplier and adder resources, respectively.

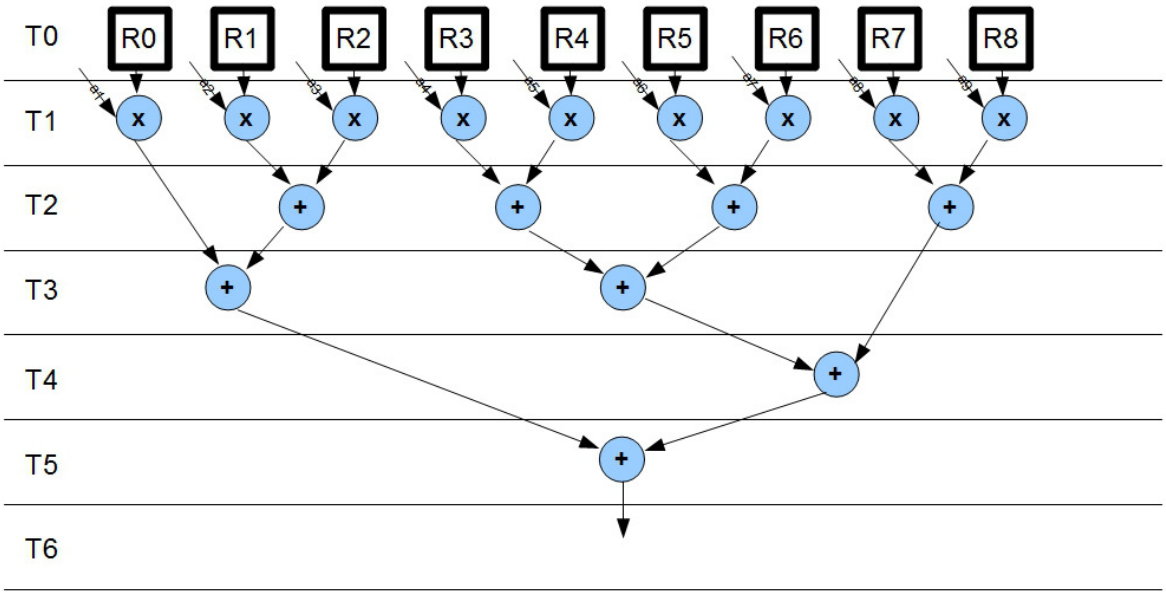


Figure 24 - Unconstrained Sequencing Graph

Before the resource constraints can be finalized it is also important to observe the sequencing graph for minimal resource usage. This sequencing graph is presented in Figure 25 below. From the figure, it is clear that the system can operate with 1 multiplier and 1 adder and no further restrictions need to be placed. Depending on the operations and pipeline structure, the typical rule of thumb is the fewer resources the lower the performance.

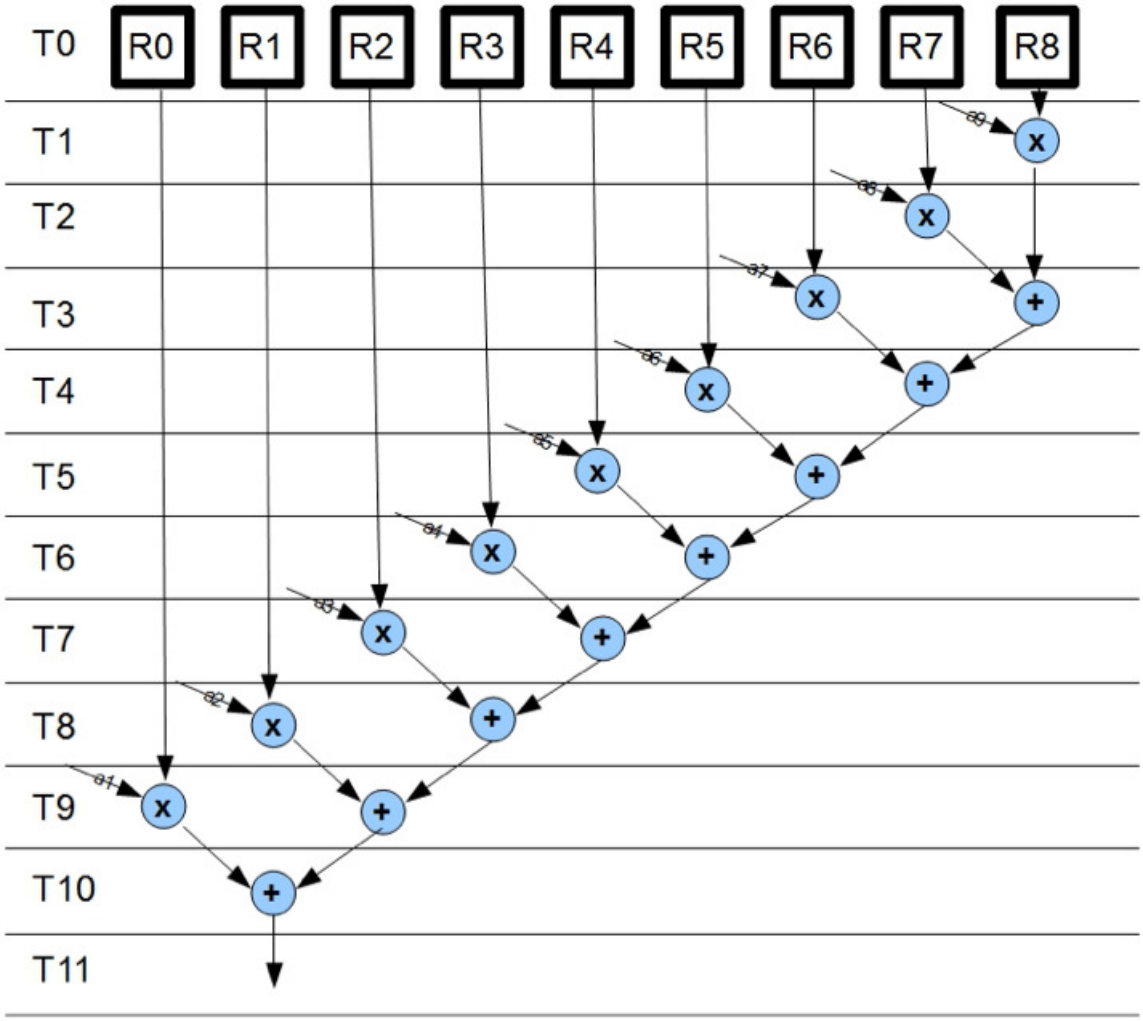


Figure 25 - Minimum Resource Sequencing Graph

4.2.4 Evaluating System Options

Since our goal is to minimize our area consumption, so long as our design meets the performance, power, and resource constraints the variant will be determined. The final resource that needs to be calculated is the clock frequency. From the sampling rate it is clear that the system must perform all relevant operation within 10.4 microseconds (based on the sampling rate). So a frequency of 10 MHz will be used (10 times the sampling rate), to provide some flexibility in the optimization 15 MHz will be available (15 times the sampling rate). A list of the resource options in this optimization are provided in the table below.

R1: Adder	**R2: Multiplier**	**R3: Clk Frequency**
R1.1 → 1 Adder	R2.1 → 1 Multipliers	R3.1 → 10 MHz
R1.2 → 2 Adders	R2.2 → 2 Multipliers	R3.2 → 15 MHz
R1.3 → 3 Adders	R2.3 → 4 Multipliers	
R1.4 → 4 Adders	R2.4 → 6 Multipliers	
	R2.5 → 8 Multipliers	

Table 9 - Resource Options

Before the optimization process can commence area, power estimations need to be performed. From [38] the relevant specification of floating point adders and multipliers are extracted. Tables 10 and Tables 11 present these findings. The costs that are provided in the above table however are in slices. Using [39] it is clear that every 4 slices equals one CLB. Using this conversion the cost of a multiplier is 102 CLBs and the cost of the adder is 111 CLBs. So far the only remaining performance benchmark is the power dissipation. Using source [40] the CLB cost is estimated at 40 μW at 200 MHz. To approximate the power consumption of the system the following

assumptions are made: (1) 20% of power dissipation is static, and (2) the dynamic power increases linearly with the clock frequency. Through these assumptions the power can be approximated. This yields a 9.6 μW power consumption per CLB at a frequency of 10 MHz and 10.4 μW at a frequency of 15 MHz.

Multiplier Cost	
Slices	408
Look-Up-Tables	646
Flip-Flops	703
Latency	6 C.C.

Table 10 - Multiplier Costs

Adder Cost	
Slices	441
Look-Up-Tables	600
Flip-Flops	590
Latency	12 C.C.

Table 11 - Adder Costs

The final task that needs to be performed prior to defining the optimization parameters and evaluations are the system constraints. It was decided that the system would accept a standard 32 bit floating point number (24 bit fraction and 8 bit exponential) for its input and all cost assumptions are based on this value. As for the system constraints the total execution time must be below the sample time and is restricted to a maximum delay of 10 μs. The power consumption is restricted to 8 mW and the optimization's objective is to minimize the area of the filter. These parameters are summarized in the table below.

Restriction	Condition
Execution Time	$T_{exe} \leq 10\mu S$
Power	$P \leq 8\, mW$
Area	$min\{A\}$

Table 12 - System Constraints

4.2.5 Mini-Max Variant

The optimization process begins with the evaluation of the design at its Minimum and maximum points. This is necessary to understand the boundaries of the system. This provides the opportunity of modifying the specifications due to unrealistic limits and lays the foundation for the critical variance analysis.

4.2.6 Min Resource Analysis

R1.1→R2.1→R3.1

This analysis performs a check to see the system behaviour with minimal resources, this implies that there is one adder, one multiplier and the clock frequency is operating at 10 MHz. The table below demonstrates the schedule of the system. From the schedule it is clear that the system can perform the necessary function within 10 clock cycles. The calculations for this variant are demonstrated below.

X	**Op_9**	**Op_8**	**Op_7**	**Op_6**	**Op_5**	**Op_4**	**Op_{93}**	**Op_2**	**Op_1**	**NOP**
+	**NOP**	**NOP**	**Op_{13}**	**Op_{12}**	**Op_{11}**	**Op_{10}**	**Op_{15}**	**Op_{14}**	**Op_{16}**	**Op_{17}**

Table 13 - Minimum Resource Schedule

Calculations:

$$T = 10 * \frac{12\ c.c.}{f_{clk}} = 120\ c.c. * \frac{1}{100\ MHz} = 12\ \mu s$$

$$A = (\#\ of\ Multipliers) * 102 + (\#\ of\ Adders) * 111 = 213$$

$$P = (213) * (9.6 \mu W) = 2\ mW$$

4.2.7 Max Resource Analysis

R1.4→R2.5→R3.2

This analysis performs a check to see the system behaviour with maximum resources, this means there are 8 multipliers, 4 adders and the running clock frequency is at 15 MHz. The table below presents the schedule for this resource schedule. The associated calculations are presented below in the calculations section.

X	**Op_9**	**Op_1**	**NOP**	**NOP**	**NOP**
X	**Op_8**	**NOP**	**NOP**	**NOP**	**NOP**
X	**Op_7**	**NOP**	**NOP**	**NOP**	**NOP**
X	**Op_6**	**NOP**	**NOP**	**NOP**	**NOP**
X	**Op_5**	**NOP**	**NOP**	**NOP**	**NOP**
X	**Op_4**	**NOP**	**NOP**	**NOP**	**NOP**
X	**Op_3**	**NOP**	**NOP**	**NOP**	**NOP**
X	**Op_2**	**NOP**	**NOP**	**NOP**	**NOP**
+	**NOP**	**Op_{13}**	**Op_{15}**	**Op_{16}**	**Op_{17}**
+	**NOP**	**Op_{12}**	**Op_{14}**	**NOP**	**NOP**
+	**NOP**	**Op_{11}**	**NOP**	**NOP**	**NOP**
+	**NOP**	**Op_{10}**	**NOP**	**NOP**	**NOP**

Table 14 - Maximum Resource Schedule

Calculations:

$$T = 5 * \frac{12\ c.c.}{f_{clk}} = 4$$

$$A = (8) * 102 + (4) * 111 = 1260$$

$$P = (1260) * (10.4 \mu W) = 13.1\ mW$$

4.2.8 Critical Variant

This section presents the calculations of the critical variants for all three resources. This step is necessary for determining the ACG placement. This step would help minimize any discontinuity between the ACG variants and provide a better optimization result. From the calculations that are presented in the following section it is possible to determine the order of the ACG for each performance check. Using the Time critical variance it is clear the resource order of the system is the following R3-R1-R2, where the resources are presented in descending order. The power ACG order is R2-R1-R3, this order is also applicable to the Area ACG.

4.2.9 Critical Variant, Adder

R1.1→R2.5→R3.2

To perform a critical variant check this system set the Adder resource to it's minimum value of 1 adder. Afterwards the operation for this system where schedule and these results are presented in Table 15. The resulting calculations for this variant are presented in the performance calculation section. From the calculated values the critical variance of the system can be calculated using the performance calculations from the maximum resource performance calculations. These results are presented in the critical variance calculation section.

	1	2	3	4	5	6	7	8	9
X	Op_9	Op_1	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_8	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_7	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_6	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_5	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_4	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_3	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
X	Op_2	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
+	NOP	Op_{13}	Op_{12}	Op_{11}	Op_{10}	Op_{15}	Op_{14}	Op_{16}	Op_{17}

Table 15 - Adder Critical Variant Schedule

Performance Calculation:

$$\boldsymbol{T = 9 * \frac{12\ c.c.}{15 * 10^{16}} = 7.2\mu S}$$

$$\boldsymbol{A = (8) * 102 + (1) * 111 = 927}$$

$$\boldsymbol{P = (927) * (10.4\mu W) = 9.64\ mW}$$

Critical Variance Calculation:

$$\boldsymbol{K_T = \frac{4\mu S - 7.2\mu S}{4-1} = -1.06\mu S}$$

$$\boldsymbol{K_A = \frac{1260 - 927}{4-1} = 111\ CLBs}$$

$$\boldsymbol{K_P = \frac{13.1mW - 9.64mW}{4-1} = 1.15mW}$$

4.2.10 Critical Variant, Multiplier

R1.4→R2.1→R3.2

The Multiplier resource is set to its minimum value of 1 multiplier this allows the system to perform a critical variance check on this resource. Afterwards the operation for this system were scheduled and these results are presented in Table 16. The calculations for this variant are presented in the performance calculation section. From the performance calculations the critical variance of the system can be determined. The results of the critical variance calculations are presented in the critical variance calculation section.

	1	2	3	4	5	6	7	8	9	10
X	Op_9	Op_8	Op_7	Op_6	Op_5	Op_4	Op_3	Op_2	Op_1	NOP
+	NOP	NOP	Op_{13}	Op_{12}	Op_{11}	Op_{10}	Op_{15}	Op_{14}	Op_{16}	Op_{17}
+	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
+	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP
+	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP

Table 16 - Multiplier Critical Variant Schedule

Performance Calculation:

$$\boldsymbol{T = 10 * \frac{12\ c.c.}{15\ MHz} = 8\mu S}$$
$$\boldsymbol{A = (1) * 102 + (4) * 111 = 546\ CLBs}$$
$$\boldsymbol{P = (546) * (10.2\mu W) = 5.57mW}$$

Critical Variance Calculation:

$$\boldsymbol{K_T = \frac{4\mu S - 8\mu S}{5-1} = -1\mu S}$$
$$\boldsymbol{K_A = \frac{1260 - 546}{5-1} = 178.5\ CLBs}$$
$$\boldsymbol{K_P = \frac{13.1mW - 5.57mW}{5-1} = 1.88\ mW}$$

4.2.11 Critical Variant, Clk

R1.4→R2.5→R3.1

The Clock resource is set to its minimum value of 10 Mhz. The schedule of the operation for this system is presented in Table 17. The calculations for this variant are presented in the performance calculation section. From the performance calculations the critical variance of the system can be determined. The results of the critical variance calculations are presented in the critical variance calculation section.

	1	2	3	4	5
X	Op_9	Op_1	NOP	NOP	NOP
+	Op_8	NOP	NOP	NOP	NOP
+	Op_7	NOP	NOP	NOP	NOP
+	Op_6	NOP	NOP	NOP	NOP
+	Op_5	NOP	NOP	NOP	NOP
+	Op_4	NOP	NOP	NOP	NOP
+	Op_3	NOP	NOP	NOP	NOP
+	Op_2	NOP	NOP	NOP	NOP
+	NOP	Op_{13}	Op_{15}	Op_{16}	Op_{17}
+	NOP	Op_{12}	Op_{14}	NOP	NOP
+	NOP	Op_{11}	NOP	NOP	NOP
+	NOP	Op_{10}	NOP	NOP	NOP

Table 17 - Clock Critical Variant Schedule

Performance Calculation:

$$\boldsymbol{T = 5 * \frac{12\ c.c.}{10\ MHz} = 6\mu S}$$

$$\boldsymbol{A = (8) * 102 + (4) * 111 = 546\ CLBs}$$

$$\boldsymbol{P = (1260) * (9.6\mu W) = 12.1\ mW}$$

Critical Variance Calculation:

$$K_T = \frac{4\mu S - 6\mu S}{2 - 1} = -1\mu S$$

$$K_A = \frac{1260 - 1260}{2 - 1} = 0\ CLBs$$

$$K_P = \frac{13.1mW1 - 12.1mW}{2 - 1} = 1.1\ mW$$

4.2.12 Execution Time Boundary

Before a search of the design space can be performed it is necessary to determine the boundaries that the system should not explore. To perform this check a dichotomy that is similar to a binary search is used by constantly dividing the design space in half and performing an execution time check on the variant. Figure 26 presents the ACG for the execution time. The calculations section presents the calculations of the search, in the interest of space the schedules are not provided. The syntax tree section provides a figure with the check that should be performed on each variant based on the resources to determine whether or not it is an acceptable design.

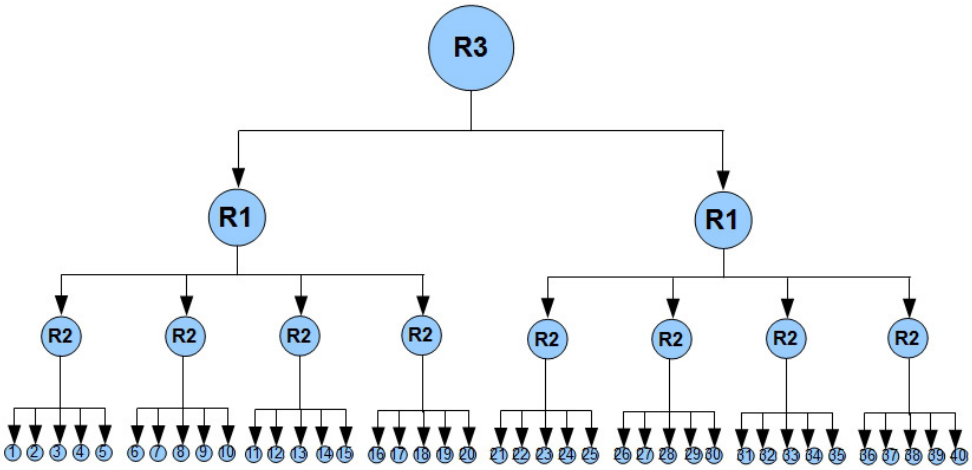


Figure 26 - ACG of Execution Time

Calculations:

Var #20 - R3.1→R1.4→R2.5

$$T = 5 * \frac{12\ c.c.}{10\ MHz} = 6\mu S$$

Var #10 - R3.1→R1.2→R2.5

$$T = 6 * \frac{12\ c.c.}{10\ MHz} = 7.2\mu S$$

Var #5 - R3.1→R1.1→R2.5

$$T = 9 * \frac{12\ c.c.}{10\ MHz} = 10.8\mu S$$

Var #7 - R3.1→R1.2→R2.2

$$T = 7 * \frac{12\ c.c.}{10\ MHz} = 8.4\mu S$$

Var #6 - R3.1→R1.2→R2.1

$$T = 10 * \frac{12\ c.c.}{10\ MHz} = 12\mu S$$

Syntax Tree:

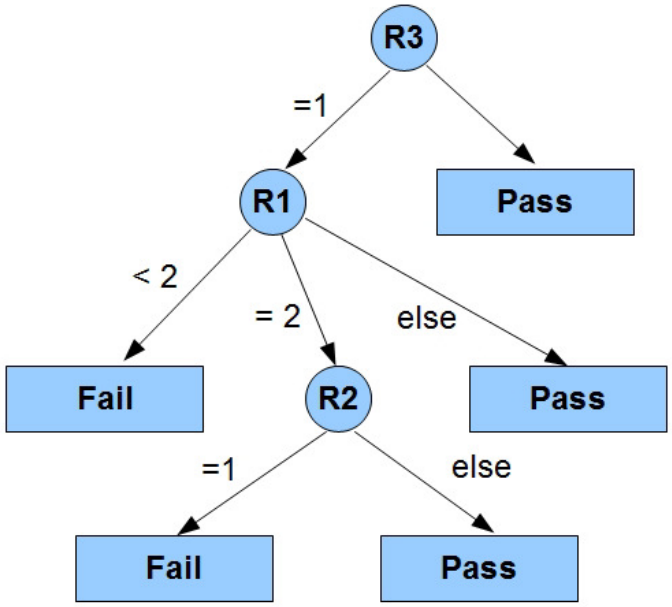


Figure 27 - Syntax Tree of Execution Time

4.2.13 Power Boundary

The calculations performed in this section are similar to the ones performed above. However since the power is dependent on the area of the system, both calculations are performed and provided in the calculations sections. Similarly the syntax tree of the power boundary is provided in the Syntax Tree section.

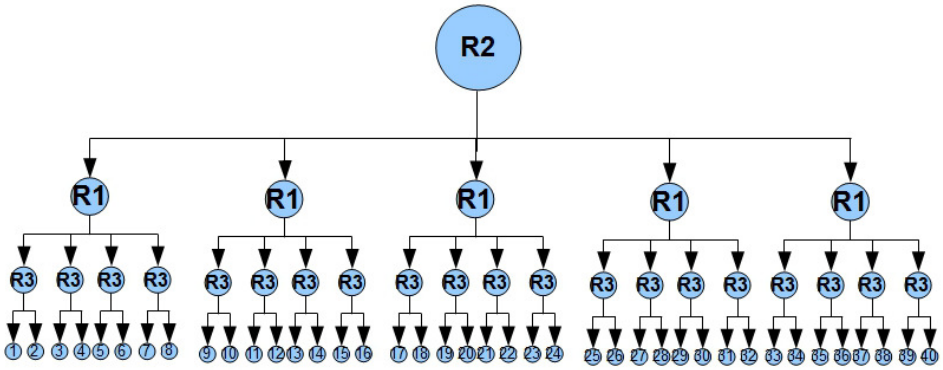


Figure 28 - Power ACG

Calculations:

Var #24 – R2.3→R1.4→R3.2

$$A = (4) * 102 + (4) * 111 = 852\ CLBs$$
$$P = (852) * (10.4\mu W) = 8.86\ mW$$

Var #32 – R2.4→R1.4→R3.2

$$A = (6) * 102 + (4) * 111 = 1056\ CLBs$$
$$P = (1056) * (10.4\mu W) = 10.98\ mW$$

Var #28 – R2.4→R1.2→R3.2

$$A = (6) * 102 + (2) * 111 = 834\ CLBs$$
$$P = (834) * (10.4\mu W) = 8.67\ mW$$

Var #30 – R2.4→R1.3→R3.2

$$A = (6) * 102 + (3) * 111 = 945\ CLBs$$
$$P = (945) * (10.4\mu W) = 9.83\ mW$$

Var #31 – R2.4→R1.4→R3.1

$$A = (6) * 102 + (4) * 111 = 1056\ CLBs$$
$$P = (1056) * (10.4\mu W) = 10.44\ mW$$

Syntax Tree:

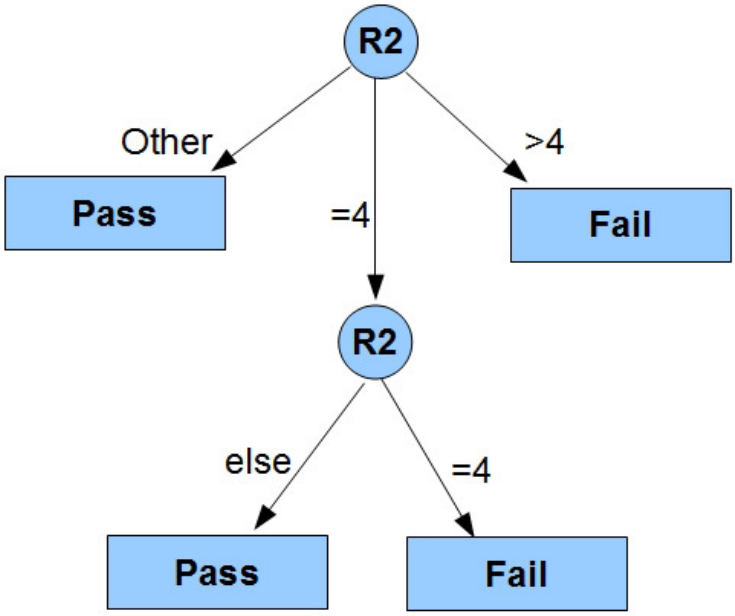


Figure 29 - Power Boundary Syntax Tree

4.2.14 Area Optimization

This step performs the optimization of the system. To perform the optimization the ACG is arranged in such a way that the left most variant (Figure 30) is the minimal area that could be used, and the area should increase as the variant moves to the right. The calculations are provided in the calculations section.

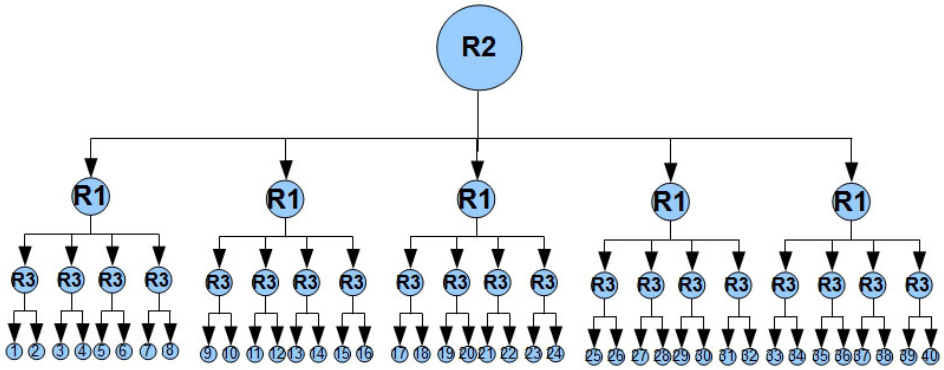


Figure 30 - Area ACG

Calculations:

Var #1 – R2.1→R1.1→R3.1

FAILED EXECUTION TIME SYNTAX CHECK

Var #2 – R2.1→R1.1→R3.2

PASSED ALL SYNTAX CHECKS PASS

$$\boldsymbol{T = 10 * \frac{12\ c.c.}{15\ MHz} = 8\mu S}$$

$$\boldsymbol{A = (1) * 102 + (1) * 111 = 213\ CLBs}$$

$$\boldsymbol{P = (213) * (10.4\mu W) = 2.2\ mW}$$

4.2.15 Resource Binding

From the above it is clear that the optimal variant is variant 2 from the area ACG. This means that only one multiplier and one adder are necessary to perform the necessary operation running at 15 MHz. The bound sequencing graph of the system is provided in Figure 31. Note that RI is an intermediate memory stage that is necessary so the multiplication performed during T0 and T1 is not lost. The schedule of this system

is provided in the table below. It is also important to note that the Difference between T0 and T1 is 12 clock cycles at 15 MHz.

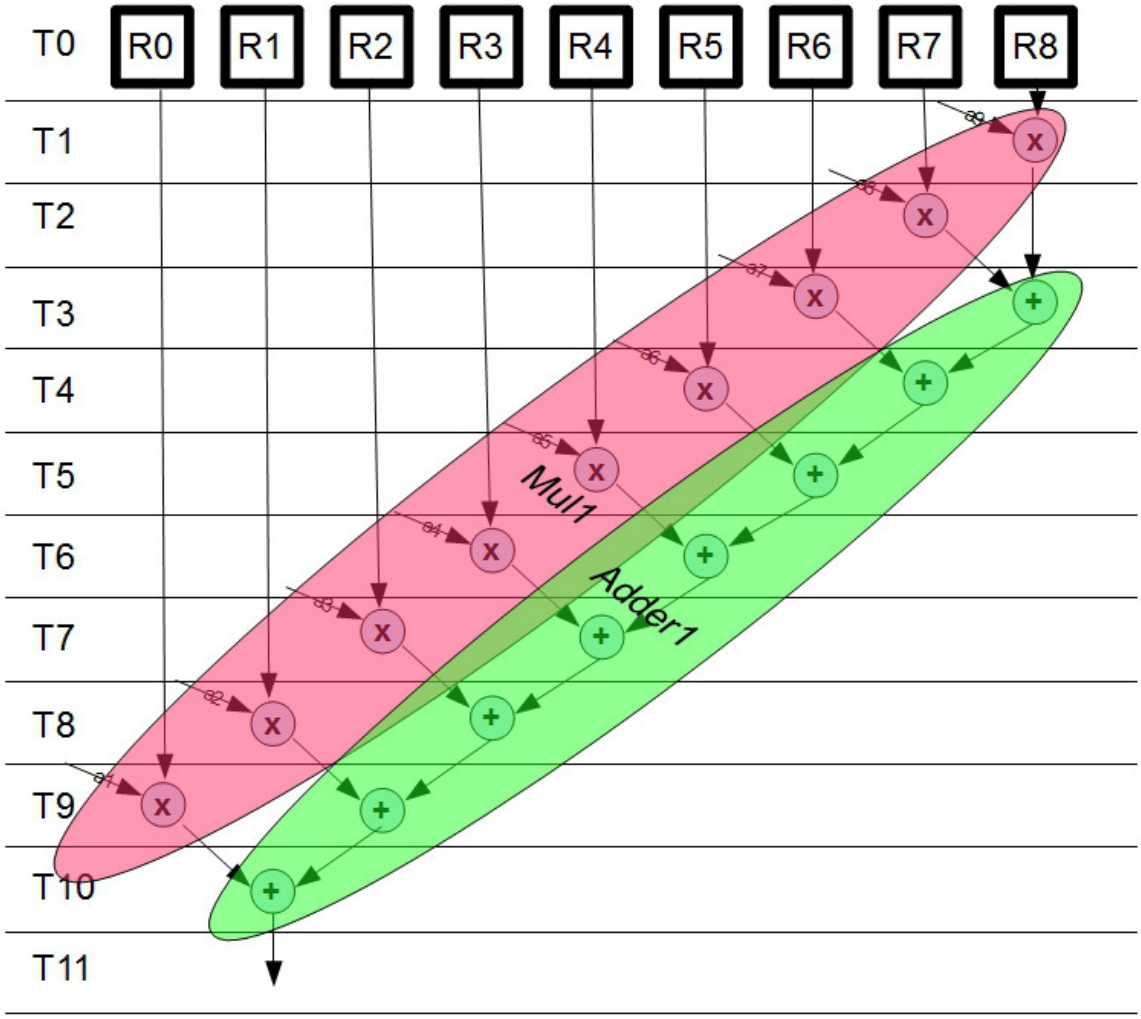


Figure 31 - Bound Sequencing Graph

X	**Op_9**	**Op_8**	**Op_7**	**Op_6**	**Op_5**	**Op_4**	**Op_{93}**	**Op_2**	**Op_1**	**NOP**
+	**NOP**	**NOP**	**Op_{13}**	**Op_{12}**	**Op_{11}**	**Op_{10}**	**Op_{15}**	**Op_{14}**	**Op_{16}**	**Op_{17}**

Table 18 - Schedule of Optimal Variant

4.2.16 Multiplexing Scheme

After the resource binding is determined the multiplexing scheme for the resources can be generated. This stage is important because it determines the data path of the system and ensures that the optimal resource usage. The multiplexing scheme of the Adder can be observed in Table 19, while the multipliers multiplexing scheme can be observed in Table 20. From the tables mentioned above the mux plans were generated. Figure 32 presents the mux plan of the adder while Figure 33 presents the mux plan of the multiplier.

Adder1	Op	In1	In2	Output
T0	NOP	-	-	-
T1	NOP	-	-	-
T2	NOP	MUL1	RI1	-
T3	+	MUL1	Adder1	Adder1
T4	+	MUL1	Adder1	Adder1
T5	+	MUL1	Adder1	Adder1
T6	+	MUL1	Adder1	Adder1
T7	+	MUL1	Adder1	Adder1
T8	+	MUL1	Adder1	Adder1
T9	+	MUL1	Adder1	Adder1
T10	+	-	-	Y

Table 19 - Adder Multiplexing Scheme

Multiplier1	Op	In1	In2	Output
T0	NOP	R8	A9	-
T1	X	R7	A8	RI
T2	X	R6	A7	Adder1
T3	X	R5	A6	Adder1
T4	X	R4	A5	Adder1
T5	X	R3	A4	Adder1
T6	X	R2	A3	Adder1
T7	X	R1	A2	Adder1
T8	X	R0	A1	Adder1
T9	X	-	-	Adder1
T10	NOP	-	-	-

Table 20 - Multiplier Multiplexing Scheme

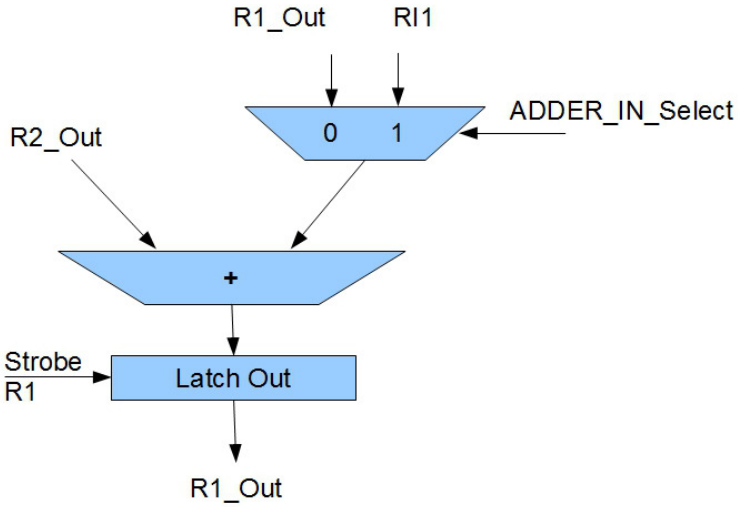


Figure 32 - Adder Mux Plan

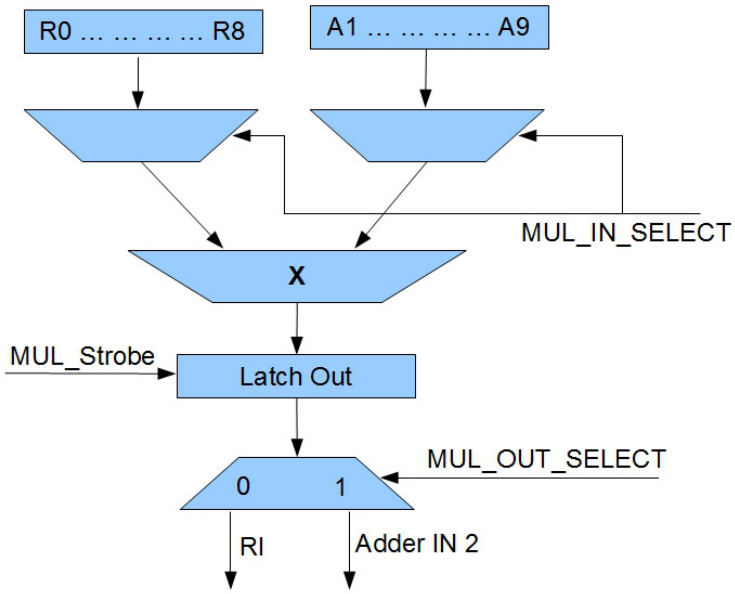


Figure 33 - Multiplier Mux Plan

4.2.17 Block Design Using Optimization Strategy

The block level symbol of the digital system is presented in Figure 34. The block has five input terminals they are: Data In, which is responsible for providing the input data; Data IN CLK, which provide a clock that is aligned with the input data; Reset, which is responsible for resetting the FIR BLOCK; EN, which enables or disables the FIR block; and CLK , which is the input clock for the system. On the other hand there are two output terminals: Data Out, where the data leaves the system; and Data OUT CLK, which is provides a clock that is aligned with the output data. All terminals are one bit with the exception of the Data IN and Data out which are left as 32 bit words. The reason the inputs are left as 32 bit buses is because the data arrives at 32 bits from the

ADC, and the Data out is usually transmitted to a DAC which will only support one word length.

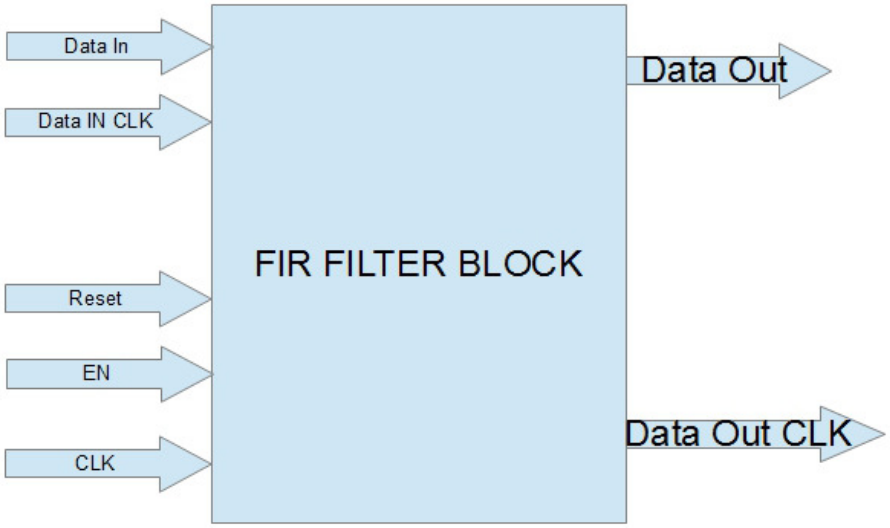


Figure 34 - FIR Block Symbol

CHAPTER V:

5.1 EFFECTIVE COMPARISON

The performance gains can manifest through a number of metrics and considerations (as discussed in section 3.1), including: faster execution time, area savings, reduced development costs, lower risks, customization enablement, and extended effective processor lifetime.

One of the challenges in this analysis is that the performance gains directly correlate to the system tasks being replaced with the work of this thesis. The rules found in 3.2.1 outline how to identify opportune logic. Moreover there is still a responsibility on the system developer(s) and engineer(s) to effectively develop and integrate these suggested changes. Due to the fact that performance metrics will be influenced proportional to the room for opportunity of each metric and the actual implementation it is difficult to standardize the performance gains.

That said, and as demonstrated in chapter 4, there is a room to achieve significant system performance gains. Knowing each case will achieve varying performance gains, the case in the previous chapter provided a performance gain of 1.32% at a system level and 183.60% on a block level.

CHAPTER VI: FUTURE WORK

6.1 SELF-ADAPTING SYSTEMS

Chapter V laid the foundation for a number of advancements to existing systems. A system that is capable of self-adaptation provides nearly endless areas of opportunity. [2] and [3] were just two examples of this.

This technology, for example, opens new horizons for FPGA testing, diagnosis and self-restoration (all together this is called “fault-tolerance”). It also makes it possible to easily apply test methods. FPGA fault-tolerance is an endless topic and is only growing in terms of demand as transistor technology continues to shrink and designs become more complex.

6.2 SELF-TESTING CAPABILITIES

Following the rules/guidelines set out from [2], we can use FPGA for other areas as well. For example, the work in [1] patent allows ASICs to recover from a radiation (power or other) event, which damages one or many transistors. This is going to be a bigger problem as semi-conductor corporations use smaller and smaller transistors and especially with a growing demand for cloud computing (as there is a higher risk of radiation damage).

Once a chip has been verified and tested it can be shipped to the end customer. Although these chips have successfully passed lab testing, there is still a possibility where transistors can be damaged during runtime as a result of alpha particles. If this happens the chip is now considered damaged. The issue is two-fold: (1) identifying the issue (2) working around the issue (if possible). There is a huge push for this within space exploration, and my work seeks to offer feasible solutions to these companies. There is still much to research and examine in this field.

APPENDIX A: UVM / SYSTEMVERILOG

RTL & Test Bench Code

Portions of the code below have been modified or removed so as to honour confidentiality.

Makefile

```
root       = $(STEM)/FIFO/src
rtl_path    = $(root)/rtl
tb_path     = $(root)/test
DUT      = $(tb_path)/fifo_tb_wrapper_io.sv $(tb_path)/fifo_tb_wrapper_io.sv $(rtl_path)/fifo.vh
$(rtl_path)/fifo.v
TB_TOP           = $(root)/test.sv
FSDB_NAME   = ${test}.fsdb

VCSFLAGS = +v2k -full64

TB_TOP      = $(root)/test.sv
HARNESS_TOP = $(tb_path)/fifo_tb.sv
UVM_HOME    = _____
defines            = UVM_NO_DEPRECATED+UVM_OBJECT_MUST_HAVE_CONSTRUCTOR
uvm_ver          = uvm-1.1
test       = fifo_sanity
seed               = 1
#verbosity       = UVM_HIGH
#option                    = UVM_TR_RECORD +UVM_LOG_RECORD
verbosity   = UVM_DEBUG

compile_dut: $(DUT)
        mkdir -p ./out
        vcs $(VCSFLAGS) -timescale="1ns/100ps" -l out/comp.log -debug_all ${SOURCES}
${DUT} -o out/simv

uvm_compile:
        mkdir -p ./out
        vcs $(VCSFLAGS) -sverilog ${UVM_HOME}/____/uvm_dpi.cc -CFLAGS -DVCS
${UVM_HOME}/___/uvm_pkg.sv +incdir+${UVM_HOME}/___/vcs -timescale="1ns/100ps" -l
out/comp.log -debug_all ${SOURCES} ${DUT} ${HARNESS_TOP} ${TB_TOP}
+define+${defines} -o out/simv

run_base_test: uvm_compile
        ./out/simv -l out/${test}_run.log +ntb_random_seed=$(seed) +UVM_TESTNAME=$(test)
+ntb_solver_mode=$(SOLVER) +UVM_VERBOSITY=$(verbosity) +${option}

$(FSDB_NAME): uvm_compile
```

```
	./out/simv -l out/${test}_run.log +ntb_random_seed=$(seed) +UVM_TESTNAME=$(test)
+ntb_solver_mode=$(SOLVER) +UVM_VERBOSITY=$(verbosity) +${option} +DEBUSSY=1
+DUMP=1 +dumpfile=$(FSDB_NAME) +fsdb+all=on +fsdbfile+$(FSDB_NAME)

verdi: $(SOURCES) $(FSDB_NAME)
	verdi -onfatalerrorcontinue $(NWAVE_FLAGS) -f $(SOURCES) ${DUT} -sv -ssf
$(FSDB_NAME)

nwave: $(FSDB_NAME)
	nWave -ssf $(FSDB_NAME)

clean:
	rm -rf out csrc *.rc *.key verdiLog vericomLog vc_hdrs.h $(test).log work.lib++ *.log
```

test.sv

```
`ifndef TEST__SV
`define TEST__SV

import uvm_pkg::*;
`include "test/test_top.sv"
`include "test/fifo_sanity.sv"

program automatic test;
    initial begin
                $timeformat(-9, 1, "ns", 10);
        run_test();
    end
endprogram

`endif
```

fifo.vh

```
`ifndef __FIFO_VH
`define __FIFO_VH

`define   QUEUE_DEPTH          8
`define   QUEUE_WIDTH          4
`define   MAX_DATA_SIZE        16

`endif
```

fifo.v

```
`include "rtl/fifo.vh"

module fifo (
```

```
    clk,
    rst,
    iEn,
    iData,
    oData,
    iPush,
    oSend,
    iPop,
    oFull,
    oAlmostFull,
    oEmpty
);

input clk, rst, iEn;
input [`QUEUE_WIDTH-1:0] iData;
output reg [`QUEUE_WIDTH-1:0] oData;
input iPush, iPop;
output oFull, oAlmostFull, oEmpty;    // Can't "assign" if reg
output reg oSend;

reg [`QUEUE_DEPTH-1:0] rdPtr;
reg [`QUEUE_DEPTH-1:0] wrPtr;
reg [`QUEUE_WIDTH-1:0] myQueue [`QUEUE_DEPTH-1:0];
reg loop;

assign oFull = ((wrPtr == rdPtr) && (loop == 1));
assign oAlmostFull = ((wrPtr == rdPtr - 1) && (loop == 1));
assign oEmpty = ((rdPtr == wrPtr) && (loop == 0));
//assign oSend = rst ? 0 : iPop;

always @(posedge clk) begin
    if (rst == 1) begin
        loop <= 0;
        wrPtr <= 0;
        rdPtr <= 0;
        oData <= 0;
        oSend <= 0;
    end
    else begin
        if (iEn == 1) begin
            // Push Logic
            if (iPush == 1) begin
                if ((wrPtr == rdPtr) && (loop == 1)) begin
                    // assert
                    $display ("ERROR: Push while full in %m\n");
                    $finish;
                end
                else begin
                    myQueue[wrPtr] <= iData;
```

```
            end
        end

        // Pop Logic
        if (iPop == 1) begin
            if ((wrPtr == rdPtr) && (loop == 0)) begin
                // assert
                $display ("ERROR: Pop while empty in %m\n");
                $finish;
            end
            else begin
                oData <= myQueue[rdPtr];
            end
            oSend <= 1;
        end
        else begin
            oSend <= 0;
        end

        // Pointer Update Logic
        if (iPush == 1) begin
            if (wrPtr == `QUEUE_DEPTH) begin
                wrPtr <= 0;
                if (!((iPop == 1) && (rdPtr == `QUEUE_DEPTH))) begin
                    loop <= 1;
                end
            end
            else begin
                wrPtr <= wrPtr + 1;
            end
        end
        if (iPop == 1) begin
            if (rdPtr == `QUEUE_DEPTH) begin
                rdPtr <= 0;
                loop <= 0;
            end
            else begin
                rdPtr <= rdPtr + 1;
            end
        end
    end
  end
end
endmodule
```

fifo_agent.svh

```
`ifndef _FIFO__AGENT__SVH
```

```
`define _FIFO__AGENT__SVH

`include "test/fifo_driver.svh"
`include "test/fifo_seq_item.svh"
`include "test/fifo_monitor.svh"
`include "test/fifo_scoreboard.svh"
`include "test/fifo_sequencer.svh"

//typedef uvm_sequencer #(Fifo_Data_inPacket) fifo_sequencer;

class fifo_agent extends uvm_agent;
    virtual fifo_tb_wrapper_io uFIFO_IO;
//        fifo_env _fifo_env;

          fifo_sequencer _fifo_sequencer;
          fifo_driver _fifo_driver;
          fifo_monitor _fifo_monitor;

    `uvm_component_utils(fifo_agent)

    function new(string name, uvm_component parent);
        super.new(name, parent);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction: new

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        _fifo_sequencer = fifo_sequencer::type_id::create("_fifo_sequencer", this);
        _fifo_driver = fifo_driver::type_id::create("_fifo_driver", this);
        _fifo_monitor = fifo_monitor::type_id::create("_fifo_monitor", this);

        if (!uvm_config_db#(virtual fifo_tb_wrapper_io)::get(this, "", "dut_io", uFIFO_IO))begin
            `uvm_fatal("CFGERR", "The interface has not been set!");
        end

        uvm_config_db#(virtual fifo_tb_wrapper_io)::set(this, "*", "dut_io", uFIFO_IO);
    endfunction: build_phase

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        _fifo_driver.seq_item_port.connect(_fifo_sequencer.seq_item_export);
    endfunction: connect_phase

    task main_phase(uvm_phase phase);
        forever begin
```

```
            @(uFIFO_IO.driver_cb);
            this.randomize();
        end
    endtask

endclass: fifo_agent
`endif
```

fifo_driver.svh

```
`ifndef _FIFO__DRIVER__SVH
`define _FIFO__DRIVER__SVH

`include "rtl/fifo.vh"
`include "test/fifo_seq_item.svh"

class fifo_driver extends uvm_driver #(Fifo_Data_inPacket);
    virtual fifo_tb_wrapper_io uFIFO_IO;

    `uvm_component_utils(fifo_driver)

    function new(string name, uvm_component parent);
        super.new(name, parent);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction: new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        uvm_config_db #(virtual fifo_tb_wrapper_io)::get(this, "", "dut_io", uFIFO_IO);
        if (uFIFO_IO == null) begin
            `uvm_fatal("CFGERR", "Interface for fifo_driver not set!");
        end
    endfunction

    virtual task reset_phase(uvm_phase phase);
        super.reset_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        // ---- RESET DUT ----
        phase.raise_objection(this);
        uFIFO_IO.driver_cb.rst       <=  1'b0;
        uFIFO_IO.driver_cb.iEn       <=  1'b0;
        uFIFO_IO.driver_cb.iData     <=  1'b0;
        uFIFO_IO.driver_cb.iPush     <=  1'b0;
        uFIFO_IO.driver_cb.iPop      <=  1'b0;
        repeat (10) @(uFIFO_IO.driver_cb);
        uFIFO_IO.driver_cb.rst       <=  1'b0;
```

```
        repeat (10) @(uFIFO_IO.driver_cb);
        phase.drop_objection(this);
        // ---- END RESET ----
    endtask: reset_phase

    virtual task run_phase(uvm_phase phase);
    forever begin
        Fifo_Data_inPacket req;
        seq_item_port.get_next_item(req);

        `uvm_info("DRV_RUN", req.sprint(), UVM_MEDIUM);

        send(req);
        seq_item_port.item_done();
    end
    endtask

   virtual task send(Fifo_Data_inPacket req);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        req.randomize();

        uFIFO_IO.driver_cb.rst   <= req.rst;
        uFIFO_IO.driver_cb.iEn   <= req.iEn;
        uFIFO_IO.driver_cb.iPush <= req.iPush;
        uFIFO_IO.driver_cb.iPop  <= 1'b1;
        uFIFO_IO.driver_cb.iData <= req.iData;
        @(uFIFO_IO.driver_cb);
        uFIFO_IO.driver_cb.iPush <= 1'b0;
   endtask: send

endclass: fifo_driver
`endif
```

fifo_env.sv

```
`ifndef _FIFO__ENV__SVH
`define _FIFO__ENV__SVH

`include "test/fifo_agent.svh"

class fifo_env extends uvm_env;

   fifo_agent          _fifo_agent;
   fifo_scoreboard     _fifo_scoreboard;

   `uvm_component_utils(fifo_env)

   function new(string name, uvm_component parent);
      super.new(name, parent);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
   endfunction: new

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

      _fifo_agent = fifo_agent::type_id::create("_fifo_agent", this);
      _fifo_scoreboard = fifo_scoreboard::type_id::create("_fifo_scoreboard", this);

   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);


_fifo_agent._fifo_monitor.mon_AnlysPort.connect(_fifo_scoreboard.fifo_iPort.analysis_export);

_fifo_agent._fifo_monitor.out_AnlysPort.connect(_fifo_scoreboard.fifo_oPort.analysis_export);
   endfunction: connect_phase

endclass: fifo_env

`endif
```

fifo_monitor.svh

```
`ifndef FIFO_MONITOR__SVH
`define FIFO_MONITOR__SVH

`include "test/fifo_seq_item.svh"

class fifo_monitor extends uvm_monitor;
    virtual fifo_tb_wrapper_io uFIFO_IO;
    uvm_analysis_port #(Fifo_Data_outPacket) out_AnlysPort;
    uvm_analysis_port #(Fifo_Data_inPacket) mon_AnlysPort;

    `uvm_component_utils(fifo_monitor);

    function new(string name, uvm_component parent);
        super.new(name, parent);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);


    endfunction: new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        uvm_config_db#(virtual fifo_tb_wrapper_io)::get(this, "", "dut_io", uFIFO_IO);
        if (uFIFO_IO == null) begin
            `uvm_fatal("CFGERR", "Interface for monitor not set!");
        end

        out_AnlysPort = new("out_AnlysPort", this);
        mon_AnlysPort = new("mon_AnlysPort", this);
    endfunction: build_phase

    virtual task run_phase(uvm_phase phase);
        Fifo_Data_inPacket in_pkt;
        Fifo_Data_outPacket out_pkt;

        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        fork
            forever begin
                in_pkt = Fifo_Data_inPacket::type_id::create("in_pkt");

                /*...get pkt...*/
                @(uFIFO_IO.driver_cb);
                    in_pkt.rst          = uFIFO_IO.front_monitor_cb.rst;
                    in_pkt.iEn          = uFIFO_IO.front_monitor_cb.iEn;
                    in_pkt.iData        = uFIFO_IO.front_monitor_cb.iData;
```

```
                in_pkt.iPush        = uFIFO_IO.front_monitor_cb.iPush;
                in_pkt.iPop         = uFIFO_IO.front_monitor_cb.iPop;

            `uvm_info("Collected In Pkt...", {"\n", in_pkt.sprint()}, UVM_MEDIUM);
            mon_AnlysPort.write(in_pkt);
        end

        forever begin
            out_pkt = Fifo_Data_outPacket::type_id::create("out_pkt");

            /*...get pkt...*/
            @(uFIFO_IO.driver_cb;
            wait (uFIFO_IO.front_monitor_cb.iPop);
                @(uFIFO_IO.driver_cb);
                out_pkt.oData       = uFIFO_IO.end_monitor_cb.oData;
                out_pkt.oFull       = uFIFO_IO.end_monitor_cb.oFull;
                out_pkt.oAlmostFull = uFIFO_IO.end_monitor_cb.oAlmostFull;
                out_pkt.oEmpty      = uFIFO_IO.end_monitor_cb.oEmpty;

            `uvm_info("Collected Out Pkt...", {"\n", out_pkt.sprint()}, UVM_MEDIUM);
            out_AnlysPort.write(out_pkt);
        end
      join_none
  endtask: run_phase

endclass: fifo_monitor

`endif
```

fifo_sanity.sv

```
`ifndef FIFO_SANITY__SV
`define FIFO_SANITY__SV

`include "test/fifo_sequence_library.svh"

class fifo_sanity extends test_base;
  `uvm_component_utils(fifo_sanity)

  Fifo_Data_inPacket iDataPkt;
  fifo_req_sequence req_seq;
  fifo_sequencer _fifo_sequencer;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
```

```
        super.build_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

            iDataPkt = Fifo_Data_inPacket::type_id::create("iDataPkt");
        req_seq = fifo_req_sequence::type_id::create("req_seq");

    endfunction: build_phase

    task main_phase(uvm_phase phase);
        uvm_component uvm_comp_ptr;

        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        phase.raise_objection(this);

                uvm_comp_ptr = uvm_top.find("*fifo_sequencer");       //  Cast method better
practice then (_env._fifo_agent._fifo_sequencer)

                // Assign to point to actual sequencer
                $cast(_fifo_sequencer, uvm_comp_ptr);

                // Randomize each sequence!
        if (!(iDataPkt.randomize() with {iData == 0;}))
            `uvm_fatal("RNDMERR", "Randomization failed!");

                // Start Sequencer
                req_seq.start(_fifo_sequencer);

        phase.drop_objection(this);
    endtask: main_phase

endclass: fifo_sanity

`endif
```

fifo_scoreboard.svh

```
`ifndef FIFO_SCOREBOARD__SVH
`define FIFO_SCOREBOARD__SVH

class fifo_scoreboard extends uvm_scoreboard;
    `uvm_component_utils(fifo_scoreboard)

    virtual fifo_tb_wrapper_io uFIFO_IO;

    uvm_tlm_analysis_fifo #(Fifo_Data_inPacket) fifo_iPort;              // Send Req
    uvm_tlm_analysis_fifo #(Fifo_Data_outPacket) fifo_oPort;              // Out Data


    Fifo_Data_inPacket _inQueue;
    Fifo_Data_outPacket _outQueue;

    Fifo_Data_inPacket in_queue[$];
    Fifo_Data_outPacket out_queue[$];


    int num_matches = 0, num_mismatches = 0;
    int num_elements = 0;
    int scan_pointer = -1;
    int cmp_ptr = 0;
    bit oBus_busy = 0;        // Ensure one data_out per c.c.
    bit ptr_update = 1;       // Pointer doesn't need to update on pop

    function new(string name, uvm_component parent);
        super.new(name, parent);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction : new

    virtual function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

       if (!uvm_config_db#(virtual fifo_tb_wrapper_io)::get(this, "", "dut_io", uFIFO_IO)) begin
           `uvm_fatal("CFGERR", "[SBD] INTERFACE IS NOT SET!");
       end

       fifo_iPort = new("fifo_iPort", this);
       fifo_oPort = new("fifo_oPort", this);
    endfunction


    virtual function void start_of_simulation_phase(uvm_phase phase);
       super.start_of_simulation_phase(phase);
       `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
```

```
        uvm_top.print_topology();
        factory.print();
    endfunction: start_of_simulation_phase

    task run_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        fork  // Record in -> DUT
            forever begin
                @(posedge uFIFO_IO.clk);
                if (~fifo_iPort.is_empty()) begin
                    fifo_iPort.get(_inQueue);
                    in_queue.push_back(_inQueue);
                    rcv_iWrite();
                end
            end
        join_none

        fork  // Record DUT -> Out
            forever begin
                @(posedge uFIFO_IO.clk);
                if (~fifo_oPort.is_empty()) begin
                    fifo_oPort.get(_outQueue);
                    out_queue.push_back(_outQueue);
                    rcv_oWrite();
                end
            end
        join_none
    endtask

    virtual function void final_phase(uvm_phase phase);
        super.final_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        `uvm_info("TRACE", $sformatf("[SBD] Comparing Tables!\n"), UVM_LOW);

        if (in_queue.size() != out_queue.size()) begin
            `uvm_error("TRACE",$sformatf("[SBD] Output size does not match expected
size!\nExpected: %d\nActual: %d\n",in_queue.size(),out_queue.size()));
        end
        else begin
            `uvm_info("TRACE",$sformatf("[SBD] Array Sizes...\nExpected: %d\nActual:
%d\n",in_queue.size(),out_queue.size()), UVM_DEBUG);

            `uvm_info("TRACE", $sformatf("[SBD] Sizes matched...Comparing Elements...\n"),
UVM_LOW);
            final_compare();
```

```
        end

        dump_buffers();
    endfunction

    virtual task run_time_compare();
        `uvm_info("TRACE", $sformatf("[SBD] RUN_TIME: Scoreboard Comparing Model Queue
with Actual Queue"), UVM_LOW);
        cmp_ptr = out_queue.size();

        if (!(in_queue[cmp_ptr].compare(out_queue[cmp_ptr]))) begin
            `uvm_error("TRACE", $sformatf("[SBD] Element does not match expected!\nExpected:
%s\nActual: %s\n",in_queue[cmp_ptr].sprint(),out_queue[cmp_ptr].sprint()));
        end
    endtask

    virtual function void final_compare();
        `uvm_info("TRACE", $sformatf("[SBD] FINAL: Scoreboard Comparing Model Queue with
Actual Queue"), UVM_LOW);

        num_elements = out_queue.size();
        for (int i = 0; i < num_elements; i++) begin
            if (in_queue[i].compare(out_queue[i])) begin
                num_matches++;
            end
            else begin
                `uvm_error("TRACE", $sformatf("[SBD] Element does not match
expected~\nExpected: %s\nActual: %s\n",in_queue[i].sprint(),out_queue[i].sprint()));
                num_mismatches++;
            end
        end

        if (num_mismatches > 0) begin
                `uvm_info("TRACE", $sformatf("[SBD] Dumping Buffers For Debug!"), UVM_LOW);
        end
        else begin
                `uvm_info("TRACE", $sformatf("[SBD] All elements matched!\nTEST
SUCCESSFULL"), UVM_LOW);
        end
    endfunction

    virtual task dump_buffers();
        `uvm_info("TRACE", $sformatf("[SBD] Dumping Buffers"), UVM_LOW);

        for (int i = 0; i < in_queue.size(); i++) begin
            `uvm_info("TRACE",$sformatf("[SBD] Expected[%d]: %s\n",i,in_queue[i].sprint()),
UVM_LOW);
        end
```

```
        for (int i = 0; i < out_queue.size(); i++) begin
            `uvm_info("TRACE",$sformatf("[SBD] Actual[%d]: %s\n",i,out_queue[i].sprint()),
UVM_LOW);
        end
    endtask

    virtual task rcv_iWrite();
        `uvm_info("TRACE", $sformatf("[SBD] Scoreboard observed an input to the DUT"),
UVM_LOW);

        `uvm_info("TRACE",$sformatf("[SBD] iWrite: %s\n",_inQueue.sprint()), UVM_DEBUG);
    endtask

    virtual task rcv_oWrite();
       `uvm_info("TRACE", $sformatf("[SBD] Scoreboard observed an output from the DUT"),
UVM_LOW);
        run_time_compare();

        `uvm_info("TRACE",$sformatf("[SBD] oWrite: %s\n",_outQueue.sprint()), UVM_DEBUG);
    endtask


endclass: fifo_scoreboard

`endif
```

fifo_seq_item.svh

```
`ifndef FIFO_SEQ_ITEM__SVH
`define FIFO_SEQ_ITEM__SVH

`include "rtl/fifo.vh"

class Fifo_Data_outPacket extends uvm_sequence_item;
    bit  [`QUEUE_WIDTH-1:0]  oData;
    bit                      oSend;
    bit                      oFull;
    bit                      oAlmostFull;
    bit                      oEmpty;

    `uvm_object_utils_begin(Fifo_Data_outPacket)
        `uvm_field_int(oData, UVM_ALL_ON)
        `uvm_field_int(oSend, UVM_ALL_ON)
        `uvm_field_int(oFull, UVM_ALL_ON)
        `uvm_field_int(oAlmostFull, UVM_ALL_ON)
        `uvm_field_int(oEmpty, UVM_ALL_ON)
    `uvm_object_utils_end

    function new(string name = "Fifo_Data_outPacket");
```

```
        super.new(name);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction
endclass: Fifo_Data_outPacket


class Fifo_Data_inPacket extends uvm_sequence_item;
    rand bit                    iEn;
    rand bit  [`QUEUE_WIDTH-1:0]  iData;
    rand bit                    iPush;
    rand bit                    iPop;
    rand bit                    rst;

    constraint rand_enable {
        iEn dist {
            0 := 1,
            1 := 20
        };
    }

    constraint rand_reset {
        rst dist {
            0 := 150,
            1 := 1
        };
    }

    constraint rand_sigs {
        iPush dist {
            0 := 1,
            1 := 100
        };

        iPop dist {
            0 := 1,
            1 := 200
        };

        iData inside {[0:`MAX_DATA_SIZE-1]};
    }

    `uvm_object_utils_begin(Fifo_Data_inPacket)
        `uvm_field_int(iEn, UVM_ALL_ON)
        `uvm_field_int(iData, UVM_ALL_ON)
        `uvm_field_int(iPush, UVM_ALL_ON)
        `uvm_field_int(iPop, UVM_ALL_ON)
        `uvm_field_int(rst, UVM_ALL_ON)
    `uvm_object_utils_end
```

```
    function new(string name = "Fifo_Data_inPacket");
        super.new(name);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction
endclass: Fifo_Data_inPacket

`endif
```

fifo_sequence_library.svh

```
`ifndef FIFO_SEQUENCE_LIBRARY__SVH
`define FIFO_SEQUENCE_LIBRARY__SVH

class fifo_base_sequence extends uvm_sequence #(Fifo_Data_inPacket);
        `uvm_object_utils(fifo_base_sequence)

        function new(string name = "fifo_base_sequence");
                super.new(name);
                `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
        endfunction

        virtual task pre_body();
                if(starting_phase!=null)
                        starting_phase.raise_objection(this);
        endtask

        virtual task post_body();
                if (starting_phase != null) begin
                        uvm_objection objection = starting_phase.get_objection();
                        #50;
                        starting_phase.drop_objection(this);
                end
        endtask
endclass: fifo_base_sequence

class fifo_req_sequence extends fifo_base_sequence #(Fifo_Data_inPacket);
        `uvm_object_utils(fifo_req_sequence)

        function new(string name = "fifo_request");
                super.new(name);
                `uvm_info("TRACe", $sformatf("%m"), UVM_HIGH);
        endfunction

        task body();
                `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
                `uvm_do(req);
        endtask
endclass: fifo_req_sequence
`endif
```

fifo_sequencer.svh

```
`ifndef _FIFO__SEQUENCER__SVH
`define _FIFO__SEQUENCER__SVH

class fifo_sequencer extends uvm_sequencer #(Fifo_Data_inPacket);
    virtual fifo_tb_wrapper_io uFIFO_IO;

    function new (string name, uvm_component parent);
        super.new (name, parent);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction: new

    `uvm_component_utils (fifo_sequencer)

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        if (!uvm_config_db#(virtual fifo_tb_wrapper_io)::get(this, "", "dut_io", uFIFO_IO)) begin
            `uvm_fatal ("CFGERR", "fifo_tb_wrapper_io is not set");
        end
    endfunction

    function void connect();
        super.connect();
    endfunction

endclass

`endif
```

fifo_tb_wrapper_io.sv

```
`ifndef FIFO_IO__SV
`define FIFO_IO__SV

`include "rtl/fifo.vh"

interface fifo_tb_wrapper_io(input logic clk);
    logic rst;
    logic iEn;
    logic [`QUEUE_WIDTH] iData;
    logic [`QUEUE_WIDTH] oData;
    logic iPush;
    logic oSend;
    logic iPop;
    logic oFull;
    logic oAlmostFull;
    logic oEmpty;

    clocking driver_cb @(posedge clk);
        output rst;
        output iEn;
        output iData;
        output iPush;
        output iPop;
    endclocking: driver_cb

    clocking front_monitor_cb @(posedge clk);
        input rst;
        input iEn;
        input iData;
        input iPush;
        input iPop;
    endclocking: front_monitor_cb

    clocking end_monitor_cb @(posedge clk);
        input oData;
        input oSend;
        input oFull;
        input oAlmostFull;
        input oEmpty;
    endclocking: end_monitor_cb

    modport TB(clocking driver_cb, clocking end_monitor_cb);
endinterface: fifo_tb_wrapper_io

`endif
```

fifo_tb.sv

```
// TOP level FIFO testbench
`timescale 1ns/10ps

module fifo_tb();
    parameter HALF_CLOCK_PERIOD = 50;

    logic clk;
    real half_clock_period;

    fifo_tb_wrapper_io uFIFO_IO(clk);                  // Interface

    //test fifo_sanity(uFIFO_IO);                                  // Instantiate Test Program

    fifo fifo(
      .clk                  (uFIFO_IO.clk),
      .rst                  (uFIFO_IO.rst),
      .iEn          (uFIFO_IO.iEn),
      .iData            (uFIFO_IO.iData),
      .oData            (uFIFO_IO.oData),
      .iPush            (uFIFO_IO.iPush),
      .oSend        (uFIFO_IO.oSend),
      .iPop             (uFIFO_IO.iPop),
      .oFull            (uFIFO_IO.oFull),
      .oAlmostFull      (uFIFO_IO.oAlmostFull),
      .oEmpty       (uFIFO_IO.oEmpty)
    );

        initial begin
       half_clock_period = HALF_CLOCK_PERIOD;
            $display ("Default Clock Period is twice: %0gns",half_clock_period);

            clk = 0;
            forever begin
                #(half_clock_period);
                clk <= ~clk;
            end
        end

        // FSDB dump

//      initial begin
//          #1;
//          $display("Dump all sub blocks");
//          $fsdbDumpvars(0, tb, "+all");
//      end

endmodule
```

test_top.sv

```
`ifndef TEST_TOP__SV
`define TEST_TOP__SV

`include "test/fifo_env.sv"

class test_base extends uvm_test;
        `uvm_component_utils(test_base)

    fifo_env _env;

        function new(string name, uvm_component parent);
      super.new(name, parent);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
   endfunction


    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

        _env = fifo_env::type_id::create("_env", this);

        uvm_config_db#(virtual fifo_tb_wrapper_io)::set(this, "_env._fifo_scoreboard", "dut_io",
fifo_tb.uFIFO_IO);
        uvm_config_db#(virtual fifo_tb_wrapper_io)::set(this, "_env._fifo_agent", "dut_io",
fifo_tb.uFIFO_IO);
    endfunction: build_phase

        virtual function void final_phase(uvm_phase phase);
                super.final_phase(phase);
                `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
                uvm_top.print_topology();

                factory.print();
        endfunction
endclass
`endif
```

APPENDIX B: ABOUT AMD

AMD is a global semiconductor company with facilities around the world. AMD was incorporated in 1969, establishing its headquarters in Sunnyvale, California. AMD designs and integrates technology that powers millions of intelligent devices, including personal computers, tablets, game consoles and cloud servers that define the new era of surround computing. AMD solutions enable people everywhere to realize the full potential of their favourite devices and applications to push the boundaries of what is possible.

AMD achieved recognition as a world leader in energy efficiency and compute power with AMD FirePro™ professional graphics being awarded the top spot on the Green500 List, a ranking of the world's most energy-efficient supercomputers, and the AMD Opteron™ server CPU receiving the number two spot on the latest TOP500 List, a ranking of the 500 most powerful supercomputers in the world. Strong demand continued for AMD-based game consoles, with Microsoft and Sony having shipped nearly 30 million AMD-based consoles through 2014.

GLOSSARY

ASIC – Application Specific Processor

AU — Area Units

C.C. — Clock Cycle(s)

DPR — Dynamic Partial Reconfiguration

FPGA — Field Programmable Gate Array

GPP — General Purpose Processor

HDL — Hardware Description Language

HVL — Hardware Verification Language

LUT — Look-Up Table

OOP — Object Oriented Programming

RTL — Register Transfer Logic

SW — Software

UML — Unified Markup Language

VHSIC — Very High Speed Integrated Circuits

REFERENCES

[1] Ryan Meghdies-Vardeh, "Adaptive ASIC Recovery," Hardware/ASIC Design, May 08, 2014.

[2] Ryan Meghdies-Vardeh, "Integrated Hardware Acceleration Islands," Hardware/ASIC Design, May 22, 2013.

[3] Ryan Meghdies-Vardeh, "Automated Preemptive Hardware Acceleration Block," Hardware/ASIC Design, May 22, 2013.

[4] A.A. Chien and V. Karamcheti, "Moore's Law: The First Ending and a New Beginning," *Computer*, vol. 46, no. 12, pp. 48-53, December 2013.

[5] Rachel Courtland. (2013, October) IEEE Spectrum. [Online]. http://spectrum.ieee.org/semiconductors/devices/the-status-of-moores-law-its-complicated

[6] R. Ammendola et al., "QUonG: A GPU-based HPC System Dedicated to LQCD Computing," in *Application Accelerators in High-Performance Computing (SAAHPC)*, 2011, pp. 113-122.

[7] Gary Stringham, *Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development*, 1st ed. Burlington, USA: Elsevier, 2010.

[8] P. Avss, S. Prasant, and R. Jain, "Virtual prototyping increases productivity - A case study," in *VLSI Design, Automation and Test*, 2009, pp. 96-101, 28-30.

[9] G. Venkataramani, K. Kintali, S. Prakash, and S. van Beek, "Model-based hardware design," in *Computer-Aided Design (ICCAD)*, 2013, pp. 69-73, 18-21.

[10] Ankit Gopani. (2013, Jan.) ASIC with Ankit. [Online]. http://asicwithankit.blogspot.ca/2013/01/most-of-re-spins-are-due-to-functional.html

[11] Marco Bernardo Alessandro Cimatti, *Formal Methods for Hardware Verification*, 6th ed. Bertinoro, Italy: Springer, 2006.

[12] Xilinx Inc. (2010, May) Xilinx. [Online]. http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf

[13] Synopsys Inc. (2014, Oct.) Synopsys. [Online]. http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx

[14] Dylan McGrath. (2005, Sep.) EE Times. [Online].

http://www.eetimes.com/document.asp?doc_id=1157473

[15] S. Oldridge, "A Novel FPGA Architecture Supporting," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, 2010, pp. 1-107.

[16] Justin Force. (2007, Sep.) CMOS NAND.svg. [Online]. http://en.wikipedia.org/wiki/File:CMOS_NAND.svg

[17] Lev Kirischian, Reconfigurable Computing Systems Engineering, 2012, Course Material.

[18] Yen-Tai Lai, Hsin-Ya Lai, and Chia-Nan Yeh, "Compiling for reconfigurable computing: A survey," *ACM Computing Surveys*, vol. 42, no. 4, pp. 1-65, 2010.

[19] R. V. Satish Ganesan, "An integrated temporal partitioning and partial reconfiguration technique for design latency improvement," *Automation & Test in Process Design*, 2000.

[20] Wang Lie and Wu Feng-yan., "Dynamic partial reconfiguration on cognitive radio platform," vol. 4, pp. 381-384, 2009.

[21] David. Dye. (2012, May) Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite. [Online]. http://www.xilinx.com/support/documentation/white_papers/wp374_Partial_Reconfig_Xilinx_FPGAs.pdf

[22] M. Edwards, "Software acceleration using coprocessors: is it worth the effort," *Proceedings of the Fifth International Workshop on Hardware/Software Codesign*, pp. 135-139, 1999.

[23] Xiaotao Chang et al., "Optimization of Stateful Hardware Acceleration in Hybrid Architectures," *Design, Automation & Test in Europe Conference & Exhibition*, 2011.

[24] P. Possa, D. Schaillie, and C. Valderrama, "Exploration, FPGA-based Hardware Acceleration: A CPU/Accelerator Interface," pp. 374-377, 2012.

[25] S. Pedre, T. Krajnik, E. Todorovich, and P. Borensztejn, "A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA," in *VIII Southern Conference on Programmable Logic (SPL)*, Buenos Aires, 2012.

[26] Wen Chen, Li-Chung Wang, J. Bhadra, and M. Abadir, "Simulation knowledge extraction and reuse in constrained random processor verification," in *Design Automation Conference (DAC)*, 2013, pp. 1-6.

[27] Ju Hwa Pan, T. Mitra, and Weng-Fai Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," *Computer Aided Design*, 2004.

[28] Intel Corporation. (2012) Intel. [Online]. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-phi-life-sciences-computing-paper.pdf

[29] Daniel L. Rosenband and Till Rosenband, "A design case study: CPU vs. GPGPU vs. FPGA," in *Formal Methods and Models for Co-Design*, 2009, pp. 69-72.

[30] HSA Foundation. (2012, June) HSA Foundation. [Online]. http://www.hsafoundation.com/hello-hsa-foundation/

[31] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data Transfer Matters for GPU Computing," in *Parallel and Distributed Systems (ICPADS*, 2013, pp. 275-282.

[32] R. Dobai and L. Sekanina, "Towards evolvable systems based on the Xilinx Zynq platform," in *Evolvable Systems (ICES)*, 2013, pp. 89-95.

[33] Xilinx Inc. (2014) Xilinx. [Online]. http://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf

[34] Intel, Block Diagram of Intel Stellarton, 2010, http://edc-cache.intel.com/App_Shared/Pix/Block-Diagrams/Stellarton-v2-Small.gif.

[35] http://gareth.halfacree.co.uk/pubimages/wm8710-blkgram.jpg, Multimedia System Block Diagram.

[36] ARM Holdings Plc, Trust Zone, 2015, http://www.arm.com/assets/images/TrustZone-Tier-3-lg.jpg.

[37] Lev Kirischian, Computer Aided Synthesis and Design of Digital Systems, 2013, Course Material.

[38] Xilinx Inc. (2011, January) Xilinx Inc. [Online]. http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf

[39] Xilinx Inc. (2008, May) Xilinx Inc. [Online]. http://www.xilinx.com/support/documentation/boards_and_kits/ug500.pdf

[40] Xilinx Inc. (2012, Oct.) Xilinx Inc. [Online]. http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf