ARCHITECTURE SYNTHESIS METHODOLOGY FOR RUN-TIME RECONFIGURABLE MULTI-TASK AND MULTI-MODE SYSTEMS WITH SELF-ASSEMBLING MICRO-ARCHITECTURE

by

Pil Woo (Peter) Chun

B. Eng. Hon., Ryerson University, Toronto, Canada, 2002

M.A.Sc., Ryerson University, Toronto, Canada, 2004

A dissertation

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2009

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-58995-3
Our file Notre référence
ISBN: 978-0-494-58995-3

AUTOUR'S DECLARATION

I hereby declare that I am the sole author of this thesis.

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

Pil Woo (Peter) Chun

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.

Pil Woo (Peter) Chun

ABSTRACT

Architecture Synthesis Methodology for Run-time Reconfigurable Multi-task and Multi-mode Systems with Self-assembling Micro-architecture

Pil Woo (Peter) Chun
Doctor of Philosophy
Graduate Program of Electrical and Computer Engineering
Ryerson University
2009

Despite the success that programmable devices have enjoyed in the last two decades, architecture synthesis methodologies for Run-Time Reconfigurable (RTR) systems are still in their infancy. As the majority of consumer devices integrate multiple-functionality, the cost-effectiveness becomes the main focus of computing systems design. This thesis presents a novel architecture synthesis methodology for the cost-effective implementation of a multi-task and multi-mode workload. The proposed methodology creates a RTR system that changes its functionality in response to a dynamic environment and enables on-chip assembly of pre-constructed components by synthesizing a workload-specific static architecture. The proposed methodology presents novelties in design abstraction, partitioning method and in the procedure of deciding reconfiguration granularity. The experimental results show the cost benefits of the proposed architecture synthesis methodology saving 73% of area and 29.8% of power compared to fixed design approach.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ACG Architecture Configuration Graph

ASIC Application Specific Integrated Circuit

ASMBL Advanced Silicon Modular Block

ATPG Automatic Test Pattern Generator

ATM Asynchronous Transfer Mode

BRAM Block selectRAM

ISA Instruction Set Architecture

CB Cell Based

CMOS Complementary Metal Oxide Semiconductor

CLB Configurable Logic Block

CPLD Complex Programmable Logic Device

CPU Central Processing Unit

DAC Digital-to-Analog Converter

DCM Digital Clock Manager

DSM Deep Sub-Micron

DSP Digital Signal Processor

EDA Electronic Design

EHW Evolvable Hardware

ESL Electronic System Level

FPGA Field Programmable Gate Array

FF Flip Flop

FIFO First In First Out

FPS Frames Per Second

GSM General Switch Module

GUI Graphical User Interface

QAM Quadrature Amplification Modulation

I/O Input/Output

IC Integrated Circuit

IP Intellectual Property

LAN Local Area Network

LED Light Emitting Diode

LUT Look Up Table

LVDS Low Voltage Differential Signal

NRE Non-Recurring

NOC Network on Chip

MB Mega Byte

MPEG Moving Picture Experts Group

MHz Mega Hertz

PCI Peripheral Component Interconnect

RE Revenue Expenditure

RC Reconfigurable Computing

PR Partial Reconfiguration

RAM Random Access Memory

RFM Reconfigurable Functional Module

RGB Red Green Blue

RTL Transfer level

HOS Hardware Operating System

SOC System On a chip

SONET Synchronous Optical Network

SCM Stereo Capture Module

SRAM Static RAM

USB Universal Serial Bus

VB Virtual Bus

VCL Virtual Component Library

VGA Video Graphics Array

VHC Virtual Hardware Component

VHDL Very High Speed Integrated Circuit Hardware Description Language

VME Versa Modular Eurocard

WLAN Wireless Local Area Network

ACKNOWLEDGEMENTS

First of all I would like to express my sincere gratitude to Dr. Lev Kirischian, Associate Professor of Electrical and Computer Engineering, Ryerson University, who has been my supervisor since the beginning of my study. He provided me with many helpful suggestions, important advice and constant encouragement during the course of this work.

My keen appreciation goes to Valeri Kirischian, Jamin Islam and Sergiy Zhelnakov for their valuable assistance in the field. Without their help, the field work would not have been accomplished in time.

I'd like to acknowledge the National Science and Engineering Research Council (NSERC), the Ontario Centres of Excellence (CITO), MDA Space Missions, CMC Microsystems, Unique Broadband Systems (UBS) and the Department of Electrical and Computer Engineering at Ryerson University for their financial contributions to this work.

I am also indebted to Soo Moon for editing the English of this document. My special appreciation goes to my parents, Son-ok Lee and Ki-young Chun, who introduced me into the world. I would like to express my heartiest thanks to my children, Monet, Adele, Jin-sol and Min-sol for their support and inspiration, and for always letting me feel that I am worthy.

Finally, I would like to express special thanks to my wife Cheng-hyun (Elena) Kim. She helped me to concentrate on completing this dissertation and supported me mentally during the course of this work. Without her help and encouragement, this study would not have been completed.

Chapter 1

Introduction

The advent of silicon technology has enabled many hardware evolutions in the computing industry in the last half century. However, the increase of computation capacity has mostly relied on increasing computation frequency of Instruction Set Architectures (ISAs). As we face the era of deep submicron process technologies, it is hard not only to increase the operating frequency of computing systems, but also to build the computing systems that can satisfy complex and multiple functionalities of today's applications. Because the dynamic partial reconfiguration in field programmable devices can change their internal structure and behaviour in response to a dynamic environment, reconfigurable computing systems allow system designers to employ more complex systems. This thesis recognizes the cost benefits that such run-time adaptability can provide and proposes a novel reconfigurable architecture synthesis methodology to achieve a cost-effective system solution. The proposed architecture synthesis methodology establishes the design steps starting from recognition of the environment to on-chip assembling of a complete micro-level system.

1.1 Motivation

As the flood of Eastern products as well as the rise of Eastern philosophical approaches inundates the industrialized part of the world, we look into the doctrine of

"Form follows function" which strongly exists in Eastern philosophies such as Oullium[1] [**1**] to shed light on the quest for a new Reconfigurable Architecture synthesis methodology. We strongly believe that reconfigurable systems can be more cost-effective by employing programmable fabrics and reusing them according to the applications needs in today's environment.

"Form follows function" is a dictum that represents the core of functionalism that Bauhaus[2] pioneered in 20th century. In functionalism, function is dynamic. Thus, form can only unveil through function. Form should stay shapeless unless indulged by function. In other words, *"Form follows function"* can be interpreted as deconstructionism[3] denying the subjective interpretation of the world. Functionalism is presented in Eastern philosophical world view – 諸法無我, there is no *'I'(subjective being)*. When there is proper understanding and interpretation of environment, then only form can follow function. Form should represent lively aspects of environment.

For example, in nature, giraffes have evolved to be tall in the attempt to get to tree leaves. Flowers have beautiful colors and shapes in order to be pollinated. In human society, forks have been elaborated to have tines in order to pierce and hold the meat. Particularly there has been no unique solution to the elementary problem of food utensils as Petroski [**2**] states in his book. Chopsticks are evolved from Eastern cuisine as the meat is cut before cooking and forks are developed in Western cuisine to cut the meat

[1] Oullium tries to extract the ideas that coexist in Buddhism, Taoism and Christianity.

[2] Bauhaus is a term used for a school in Germany that combines the skill sets that train craftsmen and artists in one umbrella. The movement started from the school became the foundation of modernist architecture of 20th century.

[3] Is a challenge to the attempt to "deconstruct" the ideological biases to establish any ultimate or secure meaning of "truth".

after it is cooked. The designs of things such as houses, cars, clothes, computers, utensils, etc. are not direct results of solving a particular task, but result from more complicated issues.

Particularly applicable to the computing industry, form is the product of design. It should thrive to be minimal (i.e., cost-effective). Form should not be limited as physical characteristics but should reflect specification of the running functions. According to the Bauhaus' functionalism, we need a computing system architecture (i.e., form) that reflects dynamic changes in applications and implements the applications as they are represented (i.e., function).

Moreover, some changes in environment do not happen very often borrowing the explanations from biological evolution. Therefore, for some biological beings, the environment feels like *"static"*. However, the environment is a dynamic entity. Thus, the perceived time scaling of changes determines how the system (e.g., biological being) needs to reflect them into the architecture (e.g., macroevolution and microevolution [**3**]).

For instance, high biological beings (e.g., mammals) are normally comprised of two types of forms: sea of homogenous elements (e.g., 100 millions neurons in brain) and a particular set of heterogeneous components (e.g., body parts and its structure). While homogenous elements possess only preliminary functionality compared to heterogeneous components, they can accomplish complex functionality by interconnecting their behaviours. The system is ready to adapt its functionality to *"dynamic"* changes of the environment that might occur. Thus, they represent flexibility at large. On the contrary, heterogeneous components try to achieve best results by knowing its requirements and tailoring the system for particular task(s). There is no imminent change to be updated.

The environment seems to be *"static"* to the system. Because form is the product of design that thrives to be minimal, both cases achieve the minimal cost from their perspectives of a *"dynamic"* or *"static"* environment. The perceived time scaling difference results in two distinctive ways of implementing forms.

The proposed reconfigurable system needs to identify functionalities in the application(s) and associate them with events that are identifiable at the system level. Once all functionalities are organized and the reusable areas are recognized, a suitable time scaling (i.e., reconfiguration granularity) is determined to differentiate between dynamic and static components of the system as the responses of the environmental events. The reconfigurable system does not only save hardware area by reconfiguring "dynamic" requests of the environment, but also maintains reliable communication between "static" components to carry on their functionalities. Conventional Instruction Set Architectures (ISAs) can not guarantee the required performance of concurrently running multiple functionalities because the system is designed to produce the results on the fixed granularity of its instructions. However, the proposed reconfigurable design does not need to compromise the performance of required computations because it implements only *currently* needed functionalities in hardware, thus reflecting lively aspects of the environment. Due to this reason, the reconfigurable systems can reduce hardware cost by adapting its form for the tasks that are imminent from its time scale, where the remote or static part of tasks can remain to be optimized and unchanged throughout the life time of the running applications.

1.2 Background

This section studies the uprising reconfigurable computing systems from various aspects: trends in embedded systems, target applications, and design methodology.

1.2.1 Embedded systems

In general, embedded systems are designed to perform a specific task in the most effective way. Thus, embedded systems should be cost conscious. To name a few embedded systems, they are:

- Terminals for wireless communication – termination for cellular, WLAN or mobile satellite access, QAM based terminals [**4**];
- Setup boxes for fixed network access – X-DSL twisted pair, CATV or LAN access;
- Video and image processing applications – MPEG-1/2/4 [**5**], teleconferencing and HDTV).

Today, these types of applications tend to co-exist in a consumer device. For example, a cellular phone can typically carry all of the above tasks. The same trend is increasingly found in many systems, standards and protocols.

The design applications (i.e., functions) which are targeted for the embedded systems increasingly require more complex functionalities with real-time and computation-intensive operations that can not be fulfilled by instruction-driven architectures (i.e., ISAs). At the same time the functionalities often become too diverse to be satisfied by Application Specific Integrated Circuits (ASICs) because of high cost associated with ASIC implementation[4].

As the advent of process technology reaches the level where the programmable fabrics can consist of billons of logic gates, they become feasible to implement large and complex System-on-Chip (SoC) designs which were only possible by ASIC implementations. Furthermore, the **programmable fabric-based devices** can *provide a form that reflects the complex (dynamic) nature of today's applications*.

1.2.1.1 Electronic system level design

Electronic System Level (ESL) design methodology is in critical need of tackling complex design problems and improving design productivity. To speed up high-quality hardware implementation using billons of logic gates and enable fast and accurate design space exploration, the research efforts should be devoted into developing an ESL methodology. To address the speed, power, and interconnect challenges, the ESL methodology has to take meaningful physical information from potential hardware and plan the physical layout of a functional implementation. One direction is to incorporate physical planning in as early as possible, thus resulting a combined synthesis and layout optimization. Our proposed **reconfigurable system's design abstraction** that *takes the multiple-functionalities of the applications on to the planning stage of the system's architecture* can achieve this goal.

[4] Cost analysis of ASIC vs. FPGA is given in the Chapter 3.

1.2.1.2 Field Programmable Gate Array

The most successful example of a programmable fabric is the Field Programmable Gate Array (FPGA). Because of diverse functionalities and genuine structure that an FPGA provides, the design applications may have dramatically different user requirements in terms of performance, power, computational throughput, etc. To satisfy the requirements of various applications, the generations of FPGA devices have been developed to contain and support a large number of heterogeneous IP cores. As the programmable devices evolve to meet the requirements of today's applications, the layout needs to deal with the heterogeneity of the reconfigurable devices, core topology, and the specific memory and interconnect structures.

Nonetheless, due to the homogeneous logic structure that the **programmable fabrics** (e.g., run-time reconfigurable FPGAs) employ, they inherit a few advantages that can be beneficial for the ESL designs:

- *Reusability* – synthesized blocks can be reused in different designs without recurring additional design costs, thus creating higher design productivity. Conversely, processing resources can be reused for various functionalities that are executed at different times to reduce the system costs.
- *Adaptability/flexibility* – allows coping with design errors and error detection and correction and accommodates last-minute changes and updates that are frequently occurring in communication and multi-media standards.

1.2.2 Target Applications

Even though programmable fabrics provide very attractive advantages, the design costs and efforts associated with realizing a reconfigurable system is not trivial. Therefore, it is crucial to mainly focus on the class of applications with which the hardware design costs are justifiable and where the performance demands can not be met with other alternatives.

1.2.2.1 Real-time applications

A real time system is where its timing behaviour is as important as its functional behaviour. A real-time computing system is always associated with the real-time constraint that is a short time during when the system must produce the results as the response to an event.

Due to its *predictable timing behaviour*, the **real-time system** is used at every corner of all industries: an anti-brake system, ATM [**6**] and SONET network nodes [7], MPEG-4/7/21 encoders [**8**], an airplane controller and a cellular node [9]. They are normally classified as one of the following categories: control, adaptive and reactive systems. These applications have different timing requirements due to the environment that they interact with.

1.2.2.2 Stream applications

The computing system with which this thesis concerns is characterized by the stream data. Due to the complexity of the data set that the system deals with (i.e., multi-dimensional arrays), parallel (or concurrent) computation must be utilized to produce the expected results within a given time.

1.2.2.3 Characteristics of real-time and stream systems

The embedded systems that deal with real-time and stream applications show a subset of the following characteristics.

Integration of multiple functionalities. It summarizes the trends of SoC that implements multiple functionalities to realize a complete electronic system that can be compact and portable by reducing area, power and cost. Because of the complexity that today's systems deal with, the system design methodology needs to reuse invariable task level designs and raise design abstraction.

Limited dynamism. Real-time systems show only a limited dynamic character within calculated variations (e.g., buffers). Since prior system knowledge makes an estimation of system resources possible, the system resources can be allocated at compile time, resulting in an optimized solution.

Timeliness. Real-time systems can be associated with timing constraints that are different in nature. They are concerned with system response time interacting with the environment. The response time often decides the bandwidth of I/O signals that the system needs to allocate. Their characteristics can be represented as input-to-output latency and throughput. The interactions between the environment and the system as well as the communication among internal components are subject to timeliness of the real-time characteristics.

Structured Data. Regarding the timeliness of the real-time system, one assumes that our application has the associated multi-array structure that is bounded by the timing constraints. Structured data, so-called *"stream"* works as prior knowledge to estimate the computational requirements of the system.

The real-time and stream systems can be much more cost-effective than ASIC counterparts when they are designed with a novel architecture synthesis methodology that reflects the multiple functionalities of applications at the system level. The combination of the aforementioned traits makes the real-time and stream applications more applicable to be designed with reconfigurable systems.

1.2.3 Reconfigurable systems

One recognizes that each application is unique in its requirements. Thus, instead of providing a fixed architecture that is unique for a particular application (i.e., ASIC) or a fixed processor that is optimized for a set of general instructions (i.e., ISA), programmable fabrics (e.g., FPGA) are utilized to provide a system that can accommodate multiple functionalities of particular applications via reconfiguration.

1.2.3.1 Reconfigurable system design flow

To facilitate the programmable fabrics effectively for multi-task and multi-mode applications, there is a need for modeling. Because a symbol cannot represent all the aspects of the element, modeling inevitably involves a process of abstraction. One describes the design flow of the reconfigurable system by four abstraction levels.

System level. In the first stage of the design flow, one focuses on correct identification of system interaction with the environment. This stage determines what Input/Output ports are necessary and how they should be characterized (e.g., rate, respond time, deadline, latency, protocols, etc). The system ports are tied with the responses that are considered in the algorithm level.

Algorithm level. After the boundary of the system is decided, the reactions of the system are determined depending on the characteristics of real-time and stream data. Then we focus on a correct functional specification of the design. In this stage there still can be design exploration relating to transformations and refinement of the data and their interpretation by the system.

Task level. After the algorithm level, the designer ends up with an optimized system specification consisting of a number of concurrent tasks, annotated with a number of timing requirements which define the boundaries for temporal correctness. The concurrent tasks are interconnected by the fixed interfaces that can tolerate limited dynamism occurring due to the timing variation of results. Then, temporal boundaries are assigned by the *system level control data* that differentiates temporal functionality of the system.

Network level. Each level should be concerned with technological limitations. Due to the physically imposed constraints under deep submicron geometry devices, today's systems put utmost importance in physical limitation of the system such as placement, routing, layout, etc. The existing systems (e.g., micro-network, network-on-a-chip, etc.) attempts to solve the physical problems of communication by imposing network-level and pre-constructed architecture. Our approach does not suggest any novel solution to the physical constraints but uses an ad-hoc approach to provide a *"micro-network"* solution. One assumes that the system consists of two routing structures that are distinctively built for global communication between coarse tasks and local communication designed for concurrent operations within the tasks. Reconfigurable system design flow should be able to account for the physical constraints and to reflect the physical specifications into the system level design.

1.2.3.2 Reconfigurable system architecture

The architecture of a computer illustrates what the computer does. From the architecture of the computer, the user should be able to perceive the conceptual structure of the computer. For this reason, the system is considered as the layered structure of subsystems keeping the hierarchical view of functionalities. The layered structure of intercommunication permits the network of elements to be viewed as one logical entity. The layered structure provides independent services that allow the insertion/omission of layers based on the necessity of services [**10**] without affecting the overall functionality of the system. Services can be tackled at each layer to employ optimal and tractable solution [**11**].

To provide the flexibility of implementing multiple functionalities and the reusability of processing resources, the service that the system needs is dynamic reconfiguration. Since the flexibility and reusability requires built-in service hardware and interconnects between components, the system is implemented into three layers: System-level, Network-level and Micro-level architecture.

- **System-level architecture**: specifies the reconfiguration mechanism and places the necessary service hardware (e.g., configuration loader, component library, configuration interfaces) for reconfiguration. Thus, it can estimate the reconfiguration overhead that can be used for deciding applicable applications.
- **Network-level architecture**: deploys appropriate network elements and layout interconnections to ensure reliable communication through specifications of modular blocks.
- **Micro-level architecture**: creates and establishes modular block components in a library for a given technology, thus enabling model-driven virtual assembly of components.

There are a few differences that the proposed reconfigurable system makes compared with others. The network architecture is unique in a sense that it is task-driven. As tasks are inserted or removed from the scene of workload, the network architecture itself reflects interconnections between tasks (and/or modular blocks constituting a task). As long as the tasks stay in the same format, the network architecture preserves its topology throughout its execution. Hence, the dynamic partial reconfiguration is applied to the region of a reconfigurable block that requires altering the structure of logics (e.g., Sobel, Laplacian or Canny algorithms for edge detection). This approach ensures the minimum reconfiguration overhead while guaranteeing on-going operations of other blocks.

When the architecture is modeled based on immediate application and associated requirements, the reconfigurable system can reflect the optimal solution for the given application which is interpreted by the hierarchical structure of multiple functionalities in the application. The proposed system is elaborated under the context of predefined architectural synthesis that is task-driven. The design procedure such as run-time Place-and-Route is excluded to keep the complexity of hardware operating system low and to allow heterogeneous structure of network architecture that is workload-driven.

1.2.3.3 Reconfigurable system model

The most common way is to think of the system as a collection of simpler subsystems. If the subsystem represents a particular application, then a group of subsystems can create a complex system that has multiple functionalities. A subsystem should possess no ambiguity and it also should be complete, comprehensive and easy to parameterize.

The (sub) systems should be able to generate specifications by being described in a particular language. A language can capture many different models, and a model can be captured in many different languages. Our models provide abstractions of a physical system and applications that allow system designers to partition the system by ignoring extraneous details while focusing on only relevant ones. Extraneous details constitutes building modular blocks in a size (i.e., <50,000 ~ 100,000 gates [**12**]) on a specific technology to ensure that the module achieves required functionality without facing timing-closure problem and deep-submicron (DSM) effects (e.g., RC delay, noise, interconnect delay and drive current). Relevant details signify the ongoing efforts of SoC to employ network into a system. However, there is significant difference in their objective between network in LAN or WAN level and network in a chip level. LAN or WAN network is designed for the systems to reduce the number of interconnections because the system cost heavily depends on the number of interconnects. The *"micro-network"* mainly concerns how reliably network elements can communicate because of problems raised from characteristics of data path and data pattern within its medium. Traditionally the main design technology for real-time and stream application has been ASIC whose main focus had been to minimize gate resources by applying global optimization. However, the technology seems to push towards subsystem level taxonomy rather than global optimization when there are many logics available and verification is hard to come by.

The purpose of the reconfigurable system model is to imprint the high-level structural view of applications onto physical aspects of a system by dynamic reconfiguration. The reconfigurable system model consists of a set of configurations and associated system events that trigger the transition between configurations. The system's temporal behaviour is explicitly expressed by applications, threads, tasks and modes[5] and implemented by reconfiguration via system events.

[5] Refer to section **2.5** for more information.

1.3 Objectives

Traditionally, the reconfigurable systems (e.g., FPGAs) have been used as the glue logic or prototyping system mainly due to its logic density. As the logic density increases there seems a broadening niche market that exists for the reconfigurable systems. However, for the idea of reconfigurable systems to be successful, one needs to accomplish the following:

- Adapting run-time programmable fabrics to reflect the demands for multi-function capability for cost-sensitive devices;
- Instituting Virtual Hardware Component (VHC) Library to enable fast and correct assembly of reconfigurable systems;
- Establishing the *system-level* design methodology to recognize multi-function applications that can benefit from the reconfigurable design approach;
- Developing systematic methods that enable the construction of workload-specific multi-task and multi-mode architecture;
- Enabling on-chip assembly mechanism for dynamic adaptation of multi-task and multi-mode workload;
- Determining a cost-effective reconfigurable granularity depending on the dynamic nature of the multi-function applications.

Many of these objectives bring the fundamental changes into the existing system design flow. By investigating many aspects of electronic digital systems and market trends[6], this thesis attempts to provide the reason(s) why it is necessary to use dynamic reconfiguration to express systems' multi-functionality and why a new architecture synthesis methodology is necessary to realize their counterparts in hardware.

Because the thesis aims to establish a new architecture synthesis methodology for building a cost-effective reconfigurable system, the author uses the system cost matrices to prove the advantage of reconfigurable design approach throughout the thesis. As the thesis proceeds, the architecture of the reconfigurable system will begin to emerge with more details being added to each chapter of the thesis.

1.4 Contributions

Reconfigurable computing, with the usage of programmable fabrics as the platform, offers a much more effective solution to address the multi-functionalities of today's applications and to reduce the system cost by reusing costly hardware. However, the reconfigurable system design encompasses much more than just reflecting dynamic changes in the applications.

In this thesis, a novel architecture synthesis methodology is developed to cost-effectively implement a multi-task and multi-mode workload. The methodology identifies the multi-task and multi-mode workload and divides the workload hierarchically according to the dynamic characteristics of the environment. The dynamic characteristics

[6] To find out what aspects and market trends are considered, refer to Chapter 3.

of the environment are modeled as the configuration events[7]. Then the methodology brings the identification to structural and physical representations of the reconfigurable system which associates the physical layout of the programmable device with the multi-task and multi-mode workload. The main novel contributions of this thesis are as follows:

- A new system-level design abstraction of a multi-task and multi-mode workload to bring structural, physical and behavioural specifications together;
- A novel partitioning method to construct an optimal system-level architecture that integrates static and dynamic components together;
- A new procedure to estimate an appropriate reconfiguration granularity.

The novel architecture synthesis methodology starts from the concept of virtualization of programmable hardware that enables the re-usability of pre-designed components. The cost-effective reconfigurable system can be realized based upon the availability of Virtual Hardware Component (VHC) library where the parametric search of components for the reconfigurable system can be systematically accomplished for a large pool of VHCs. With the suggested parametric search method[8], the reconfigurable system can construct an optimal system that is assembled with pre-synthesized VHCs in the library. To complete the assembling steps of the reconfigurable system methodology, the system needs to provide detailed parameters that result from the steps of the architecture synthesis methodology.

[7] The configuration events presumably contain the dynamic request of the environment for different functionalities.

[8] It offers faster computation than other search methods. For details, refer to the Chapter 4

Based on the detailed specifications of a multi-task and multi-mode workload and the conditions of real-time stream applications, the thesis proposes the novel architecture synthesis methodology to achieve a cost-effective system solution. The Multi-task Adaptive Reconfigurable System (MARS) platform is constructed to analyze the implementation of dynamic functional changes and to analyze the cost benefits of the proposed methodology. In the implementation:

- The feasibility of on-chip assembly via run-time reconfiguration is demonstrated;
- The procedures for on-chip self-assembly is developed and implemented;
- The concept of run-time reconfigurable system based on static architecture is created;
- The dynamic reconfiguration of various tasks (i.e., VHCs) driven by configuration events is demonstrated.

1.5 Organization of Thesis

The remainder of this thesis is structured as follows. **Chapter 2** conducts a broad survey of reconfigurable approaches that classifies the reconfigurable systems based on their view point. As a necessary supplement, **Chapter 2** investigates the architectural evolution of programmable devices using Xilinx Dynamic Reconfigurable FPGAs as a case study. **Chapter 3** explores the current market trends and correlates system costs of ASICs and FPGAs. It also shows how FPGA systems become more cost-effective as the market trends become more aggressive. **Chapter 4** proposes a methodology that explores various architectural spaces and demonstrates how this method obtains faster results for building a large amount of Virtual Hardware Components (VHCs) for the VHC library. **Chapter 5** presents a new architecture synthesis methodology that introduces a new design flow for identifying a multi-task and multi-mode workload, constructing a static architecture and defining the optimal reconfiguration granularity. **Chapter 6** encompasses the implementation details of a reconfigurable system using the optimized static architecture and acquired reconfiguration granularity via a Xilinx Virtex-4 FPGA platform. **Chapter 7** analyzes the experimental results. Then, **Chapter 8** summarizes the final outcomes and reports on possible future research directions.

Chapter 2

Related Works and Overviews of Reconfigurable Systems

2.1 Introduction

There are many computing approaches that can be classified as Reconfigurable Computing (RC). Many of the RC research have focused on providing the hardware solutions for different architectural problems. However, their attention was not paid on how to optimally design a RC system for a particular application. As the applications increasingly become computing intensive and complex, the cost of a computing system becomes too expensive to implement or manufacture in a conventional platform. It becomes necessary that a new system level design methodology needs to be combined with reconfigurable devices to provide a cost-effective system solution.

In the course of searching for an application-centric RC design methodology, one must first take a look at the broad classification of RCs, and then define some terminologies used for classification and the underlying domains of applications. To summarize the architectural centric research efforts, the specific comparisons of the RC platforms are presented according to their architectural characteristics. Finally, we explore a system design methodology of how to optimize a particular set of applications for the given RC platform.

2.2 Motivation

Moore's prophecy was widely adapted to justify the success of the semiconductor industry for the past 30 years. It states that functionality of chip should increase twice every 1.5 to 2 years. His prophecy was very attractive not only because of increase in logic density but also because of "*doing it at the same price"*. Entering Deep Submicron process technology acclaims the difficulties of keeping the manufacturing costs the same, which are associated with new lithography tools, unreliable mask accuracy and multiple emerging technologies [**13**]. Moreover shortened time-to-market and hastened product life cycle put pressure on reducing test and verification, and R&D periods for increasingly complex systems [**14**]. Figure **2-1** shows a typical example of high-end design costs associated with the process technology that shrinks as the time progresses [15].

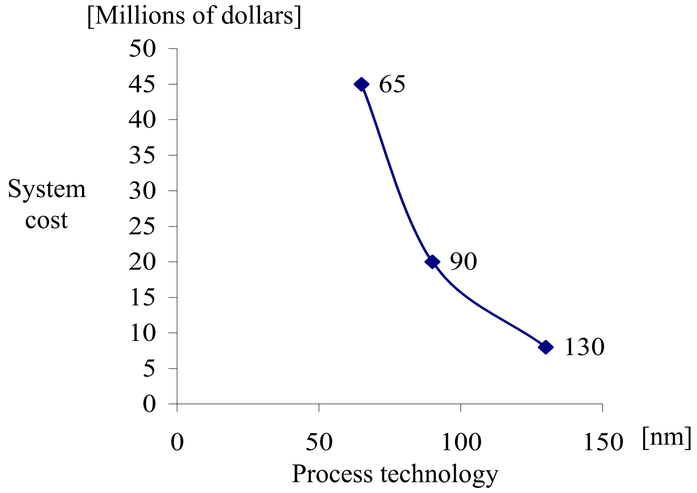


Figure **2-1**: Process technology vs. System cost

We recognize that there are the hurdles to provide the promise of Moore's law with the existing computing systems. In order to solve the riddle of *"doing it at the same price"*, we look into an unconventional solution, *Reconfigurable computing*.

2.3 Reconfigurable Computing

Reconfigurable computing (RC) can carry many different meanings. However, we believe RC is a computing paradigm that can integrate the flexibility of software into the parallel computing fabrics. The main attraction of the approach is the capability of *"tailoring"* reconfigurable computing fabrics while exploiting the performance of the available algorithms from utilization of parallel processing resources. The system can also reduce the costs by reusing valuable processing resources at run-time. The current application trends of continuously evolving multi-media and network standards showcase the benefits that are applicable via reconfigurability.

However, reusing hardware requires *"virtualizing"* processing resources – more precisely modeling. Virtualization in modeling requires establishing higher levels of abstraction (i.e., system-level abstraction). The approach with a *system-level abstraction* allows the reconfigurable system to *reuse hardware spaces via coarse control flow* while accommodating fine data flow with continuous execution of programmable parallel fabrics.

The system-level design should be written with the implementation concerns that satisfy the requirements of the underlying applications. In order to reflect technological and algorithmic limitations onto the system at early stage of the system design, it first needs to be concerned with the following topics and recognize the differences that exist between the reconfigurable systems:

- How to describe major blocks (hardware, netlist or hard/soft description);
- How big they should be (gate-level as fine granularity, function level – coarse granularity, 50K gates – physical limitation, Network node – abstraction level or application specific);
- How they are connected (SoC BUS, NoC or P-to-P) and they need to communicate.

The goal of our approach is to provide a task-specific hardware system that can respond to the current requirements of applications while reducing system cost by reusing the processing resources for multiple tasks over times. Even though our solution focuses on its effectiveness towards computing intensive applications such as real-time and stream applications, we will explore various types of the RC systems. Each type of RC renders the different merits and employs the architectural choices that target different types of applications. We will start by categorizing the RCs by widely renowned fields.

2.4 Taxonomy of Reconfigurable Computing

Many different areas of applications have adapted reconfigurable computing as a tool to create a new computing paradigm. The classification of these approaches leads us into a focused study of RC's promises and problems associated with the underlying applications. The taxonomy of RC is depicted empirically as in Figure **2-2**.

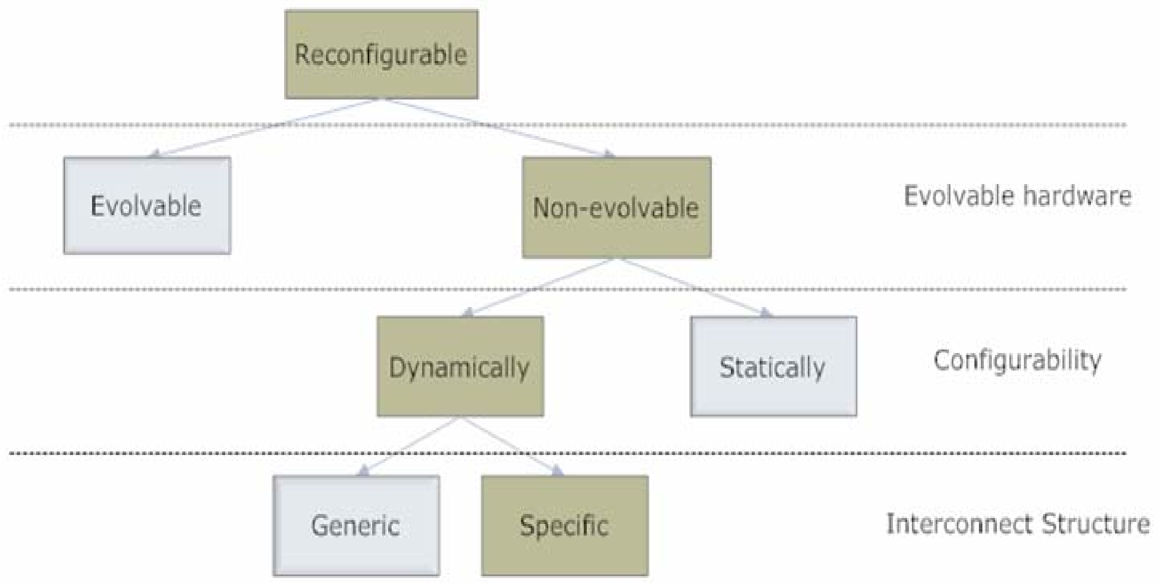


Figure **2-2**: Empirical Taxonomy of RC

Under the umbrella of reconfigurable computing, the reconfigurable systems can be divided into evolvable and non-evolvable hardware systems. When a system is evolvable, it generally contains homogeneous cells that reconfigure their own interconnects by simple rules to achieve the desired functionality. Conversely, non-evolvable systems have pre-fixed structure that is dictated by human guidance. Non-evolvable hardware systems can be further divided with respect to how the system applies changes – reconfigurability. Once reconfigurability is chosen, we pay a visit to the structures of communication – generic or specific structure. It is our interest to investigate the reconfigurable computing systems that fall in the path of the gray shaded boxes.

2.4.1 Evolvable Hardware

The RC can be first divided into evolvable and non-evolvable hardware (EHW and non-EHW). The EHW is a research area that looks for a methodology to obtain an optimal presentation by adapting its own structure to the environment [16] [**17**]. The main focus of the EHW is the evolutionary computation to design specialized circuits without explicit guidance of users [**18**], so-called embryonics [**19**].

The EHW borrows the conceptual understanding of how biological beings evolve and adapt to the environment – so-called Bio-inspired systems. Especially, it identifies that the growth and the operation of all living beings are the interpretations of how their unit cells react to one another (e.g., genome). The start of the embryonic is based on the hypothesis of the environment that is assumed to be homogeneous cellular array that evolve its state and values over time [**20**].

Generally speaking, the EHW starts by embedding fundamental rules that all entities of the EHW follow. The EHW possesses an efficient mechanism to adapt cell configuration and interconnections via enabling dynamic and autonomous reconfigurations. Then, based on the specifications of the desired circuit, the EHW starts to evolve towards the optimal circuit via self-reproduction. The EHW also can explore fault-tolerance architectures via self-healing (or self-repair) mechanisms [**21**].

However, the complexity of circuits that are synthesized using the EHW is normally limited at the functional level due to the time that takes to render the optimal results that are significantly inferior with other approaches. Additionally, unpredictability of time and quality of the design raises the reluctance to use the approach in many areas of applications. On the other hand, the non-EHW approaches normally result in predictable performance and timing closure depending on the complexity of the algorithms and the communication bandwidth. We pay attention to non-EHW (man-guided) systems that can be reconfigured to achieve higher cost effectiveness.

2.4.2 Non-EHW systems

Once the EHW is excluded, the non-EHW reconfigurable systems can be boldly divided into two categories, *dynamically* reconfigurable and *statically* reconfigurable systems. In order to distinguish how these categories play a role in dealing with tasks, we illustrate the hierarchy of the application as shown in Figure **2-3**.

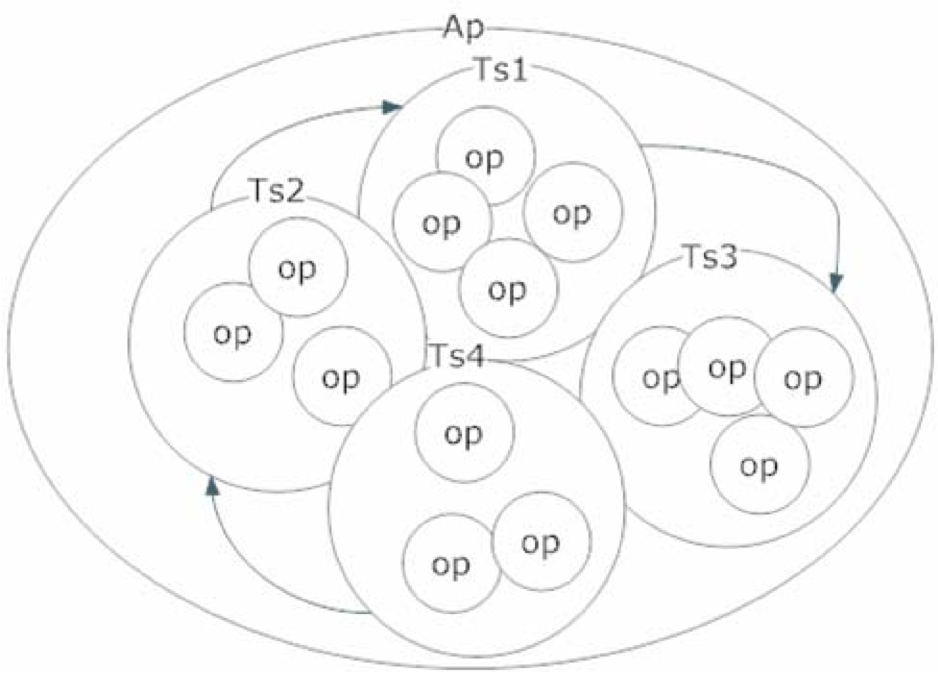


Figure **2-3**: A typical example – Hierarchy of Spatial Computation (**Ts** denoting a task, **Ap** denoting an application and **op** denoting an operation)

It is first assumed that every application is fundamentally comprised of operations that are arithmetic or logical. Associated operations are grouped together to create a *task*. Each task is independent in a sense that they do not use output(s) of operations from other task(s). Yet, they are linked by input(s)/output(s) of other task(s) or the system to map functional requirements of the application. At the same time, excessive data flow might demand adequate control by implementing internal or external memories to contain required data by tasks. Once all interactions and communication between tasks are arranged, a higher level of computation can be formed. An application is a set of tasks that are connected (or disjointed) for the system at times. The application essentially portrays a snapshot of currently required operations. The connected set of tasks in an application is referred to as a *thread*. The application may contain many threads.

Complex systems generally require multiple applications to be available as shown in Figure **2-4** . Depending on the physical and timing requirements, there can be different approaches to arrange the applications on a device – where the arrows indicate the time displacement of applications where they are functionally different. Depending on the level of change necessary, the designers must navigate between the choices of systems to reflect the requirements and produce an optimal solution.

As the name – Application Specific Integrated Circuit (ASIC), states, the ASIC is the best solution when it is *"application specific"* and fixed. Because the ASICs physically implement all applications and optimize the common operations that are timely shared among the applications, there should be virtually no application change(s) necessary to minimize the hardware idling within the ASICs. If there is a considerable amount of changes necessary between applications, the ASIC can become too expensive to manufacture.

However, alternatively RCs can use *static (full)/dynamic (partial)* reconfigurations to reflect the necessary changes. If the timing allowance in the applications is more than the configuration time of a full device, then the full reconfiguration can be employed. If the applications have stricter timing constraint but require small changes, the RC can use *dynamic* reconfiguration to echo the changes in the application without halting the rest of system's functionality.

This thesis focuses on Dynamic reconfiguration that reconfigures a portion of the device for task-level changes without disruption of links between tasks.

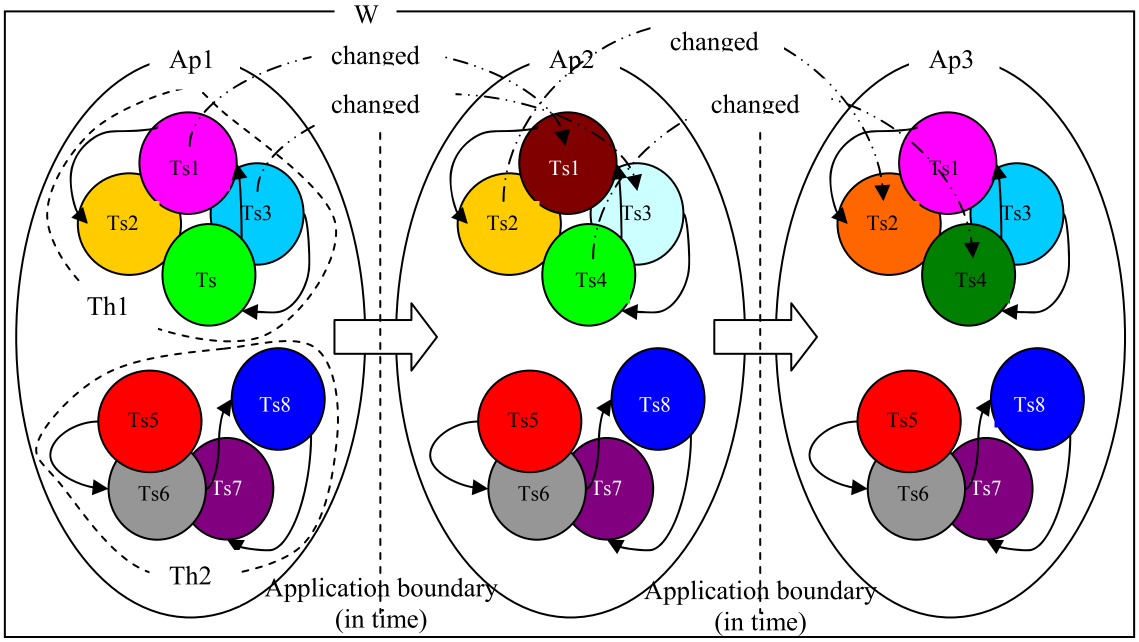


Figure **2-4**: Temporal Computation (**Th** denoting a thread and **W** denoting a workload)

While an application represents a snapshot of the system's functionality, a *workload* assembles all applications that the system requires over time. For example, let the workload consist of three applications and each application employs two threads as shown in Figure **2-4**. As for many cases, the applications do not get changed much. Shifting from Ap1 to Ap2, illustrated by color changing, only Ts1 and Ts3 need to be changed and the interconnect structure within the thread (e.g., Th1) remains unchanged. In this case, if the tasks occupy a designated area, the shifting between applications can be very easily obtained without disturbing operation of other threads. Since the tasks in Th1 change their functionalities depending on the conditions given for the application, they are called *multi-mode* tasks.

In a nutshell, the timing and physical constraints given for the system determine the arrangement of operations that are hierarchically organized as *modes*, *tasks, threads, applications* and *workload*. The reconfigurable computing paradigm can utilize dynamic nature of the applications to maximize system efficiency using temporal and spatial redundancies, particularly in terms of hardware resource utilization via reconfiguration.

2.5 Terminology and notation

In order to clearly outline the benefits of reconfiguration under different system conditions, the terminology of system and applications play an important role. One tries to define terminologies related with hierarchical organization of reconfiguration and terminologies that are used for the underlying applications in the following sections.

2.5.1 System for reconfiguration

The essential terminology and notation are defined and described in this section to express the associated applications and data, system model and their relationships. First, the system is represented by a set of operations. Based on temporal or spatial relationships among operations, they form *tasks, threads, applications* and *workload*.

Definition 1 A *task, **Ts*** is a group of arithmetic (and/or logical) operations that are necessarily interconnected to perform the described computation.

Definition 2 A *mode, **Md*** is one of ways executing a task that is bounded by constraints, and specifications.

While a *mode* is bounded by available resources and technology specifications, a *task* is a description that can be implemented by any system. For example, if the task is mathematical formulas that represent a computational algorithm, the *mode* is a placed and routed hard IP core for a certain system.

Definition 3 An *application, **Ap*** is a group of *task*(s) that are interconnected (or disjointed) to carry out computational requirements for a system given at time(s).

An *application* is the spatial expression of the system defined by the computational requirements of the time. Depending on the degree of computational changes required by the system, *applications* tend to be relatively static over short period of times.

Definition 5 A *thread,* ***Th*** is a connected set of task(s) given in an application.

Definition 6 A *workload,* ***W*** represents all application(s) that a system needs to implement at different times.

Using *mode, task, thread* and *workload,* $W \supseteq \mathrm{A}p \supseteq Th \supseteq Ts \supseteq Md$, the system can express the temporal or spatial computational requirements to reflect the necessary changes by different granularities. For detail graphical explanations, refer to the section **2.4.2**.

One acknowledges that applications that reconfigurable systems deal with are quite wide and complex. However, due to the embedded nature of computation-intensive applications on which reconfigurable systems focus, the instance of the applications that are deployed with the system is very specific and narrow. Specifically, the categories of real-time and stream applications show the promises of rendering the benefits of cost savings via exploiting redundancies available in multi-task and multi-mode applications.

2.5.2 Real-time applications

The basic terminology of real-time tasks sheds some light on the context of the applications that reconfigurable systems can effectively deal with. Their definitions are based on [**22**] and adapted to fit with the previous definitions given in this chapter**.**

Definition 7 A real-time task is an executable entity of work which is characterized by a minimum (and maximum) execution time and a time constraint.

Definition 8 A job is an instance of a task in time.

The time constraints can be release times or deadlines, or both.

Definition 9 A release time, t_r, is a point in time at which a real-time job becomes ready to execute.

Definition 10 A deadline, t_d, is a point in time by which the job (which is the instance of a task) must complete.

Real-time applications can be categorized into one of three types: periodic, aperiodic, and sporadic.

Definition 11 Periodic tasks are real-time tasks which are triggered (released) consistently at a fixed interval (period).

The notation of the period and deadline is designated to be $\boldsymbol{T}$ and $\boldsymbol{t_d}$ **,** respectively**.** Periodic tasks are associated with time constraints – the instances of a periodic task must execute once every ***T.*** The deadline of a periodic task, $\boldsymbol{t_d}$ normally equals to $\boldsymbol{T}$.

Definition 12 Synchronous periodic tasks are a set of periodic tasks where all first instances are triggered at the same time.

Definition 13 Asynchronous periodic tasks are a set of periodic tasks where all first instances are triggered at different times.

Definition 14 Aperiodic applications are real-time applications which are triggered inconsistently at some unknown bounded rate.

The bounded rate is characterized by a minimum inter-arrival period – a minimum interval of time between two successive activations.

Definition 15 Sporadic tasks are real-time tasks which are activated irregularly with some known bounded rate.

Definition 16 A hard deadline means that it is vital for the existence of the system that the deadline is met all the times.

Definition 17 A soft deadline means that it is permissible that the job is completed after the deadline, but the usefulness of a computation becomes enumerated (normally decreased) after the deadline expires.

Definition 18 A firm deadline means that a task should complete by the deadline. There is no use of completing the task after the deadline.

The deadlines of real-time applications are typically categorized as no, soft, firm and hard deadlines. While a firm deadline specifies the time point when the computation is realized to be futile, a soft deadline can formulate the usefulness of a computation by the time passed after the deadline expires.

With the above definitions, the notation used to represent the jth instance (i.e., job[9]) of ith task, Ts_i is denoted by $J_{i,j}$ or simply J_j . Each task, Ts_i is coupled with maximum execution time, C_i.

Definition 19 A job has release time , $\boldsymbol{t_r}$ if its execution can begin only at time $t \geq t_r$.

Definition 20 A job has deadline, $\boldsymbol{t_d}$ if its execution must complete by $\boldsymbol{t_d}$.

[9] Job and mode can be used interchangeably

Because the performance of computing systems greatly depends on what it deals with, we explore the traits of typical data with which the reconfigurable systems are associated. One of them is real-time applications. The real-time applications put great attention to the notion of time [**23**] [**24**]. The real-time applications specify deadlines and real-time arrivals of input signals. Based on the specifications of real-time applications such as $\boldsymbol{t_d}$**,** one attempts to identify the boundary that enables lowering the unit costs of the reconfigurable system.

Not all real-time applications provide the predictability to acquire allocation of necessary resources. Aperiodic tasks that are unpredictable are taken out from consideration. The application which does not have deadline can not take advantage of reconfigurable system because it is unsure of what configuration it needs. Thus, one only considers the real-time applications that are periodic or sporadic tasks bounded by a soft, firm or hard deadlines.

First let us specify the timing constraints of periodic[10] applications. The release time and deadline of the j^th^ job of the periodic task, $\boldsymbol{Ts_i}$ is quoted as:

$t_r(i,j)=(j-1)T_i$ and $t_d(i,j)=t_r(i,j)+T_i=jT_i$ respectively.

As noted, the deadline of one instance is the release time of the next instance. For sporadic tasks, the release time of two consecutive instances must be distanced by at least $t_d(i,j)=t_r(i,j)+T_i=jT_i$. The deadline is equal to the next release time: $t_d(i,j)=t_r(i,j)+T_i$.

[10] Periodic tasks can be either synchronous or asynchronous.

Let us assume that task $\boldsymbol{Ts_i}$ is characterized by period (or minimum inter-arrival rate), T_i, and the execution time, C_i. Then, the task (or application) can be reconfigured when:

For a task:

$t_{rec_i} \le T_i - C_i$, where t_{rec_i} is the time that takes to reconfigure $\boldsymbol{Ts_i}$

For an application or thread:

$\sum t_{rec_i} \le \min\left(T_i - C_i\right)$, where $\sum$ indicates the sum of all changing entities. $\sum t_{rec_i}$ can be further reduced by only taking the differences for reconfiguration. However, in this thesis we calculate the reconfiguration time by the area that the tasks occupy. As long as the proceeding changes are indicated at the beginning of the previous release time, the reconfiguration of task (or application) can be successfully conducted upon the above condition. Whether the system is associated with a hard, soft or firm deadline is a question of scheduling that should be dealt with at the system-level.

2.5.3 Stream applications

When the real-time applications focus on the timing notation of computation entities, the stream applications identify the data structure and determines the data access necessary for the given system architecture.

Definition 21 *Stream* is a continuous list of elements that bears a particular structure of data of which the system is interested in processing.

Stream can carry various meanings. For instance, it can refer to regular structure of high-speed communication protocols such as SONET and ATM cells, compression payloads such as JPEG, TIFF, signal processing algorithm such as DFT and DCT, multimedia protocol such as MPEG4, MPEG7 and MPEG21, etc. All the computations are determined by concatenated header or input data structure – that is known to algorithms or protocols.

Definition 22 *Stream processing* is a processing of high amount of regularly structured data in a limited time, $\boldsymbol{t_d}$.

While the real-time applications put attention to the notion of time, the emphasis of stream applications is structured around data and data access pattern that determine the requirements of system resources. We assume that the reconfigurable systems deal with data that is *stream*. Stream data is typically [**25**]:

- Organized as multi-dimensional array;
- Unbounded along the temporal domain;
- Embedded with diverse information that is structured.

The stream applications normally utilize the structure of multi-dimensional data array mapped into one-dimensional data sequence – the data sequence can be realized as serial or parallel lines [**26**].

Stream algorithm is particularly interested in processing structure of stream data where the information can be reorganized (e.g., encoded, compressed and converted). Stream processing algorithms normally require:

- Independent processing of locally limited dimensional data;
- Data access pattern that are typically fixed within the data set.

The structure of the stream application also provides the control for operations to differentiate processing. Because the structure of stream data underlines the data format and data sequence, it provides a way to estimate the reconfiguration overhead in time. However, because different applications employ different combinations of functional units and different functional units consume the data in a different way, the access pattern of the data decides how to implement an algorithm in the system.

Stream application requires the same processing to be repeated in a given time. Due to the repeated processing limited by the structure of stream data, the processing is executed independently every time when a new unit of stream is supplied. Thus, the failure of processing current stream does not influence the processing of the next stream unit.

Based on the above assumptions, the reconfiguration time would generally be smaller than the period subtracted by execution time. However, if the application can tolerate the loss of a stream unit, then the reconfiguration constraint based on real-time applications can be relaxed up to the point of the application tolerance that is the multiple of periods. The study of the underlying applications provides the way to estimate the reconfiguration constraint based on the characteristics of the applications.

When the system can reconfigure various functionalities by satisfying the reconfiguration constraint, there is a considerable amount of hardware that can be saved. These savings have inspired many companies and researchers to construct reconfigurable systems using generic interconnect structure.

2.6 Generic Reconfigurable Systems

As we enter deep sub-micron (DSM) process technology to deliver a great number of logic gates on a chip, the focus of micro-architecture has moved away from processing towards communication that is referred as *micro-network*. As the advents of the process technology decrease the gate delay comparable with wire delays under deep submicron technology, interconnection designs of reconfigurable devices are increasingly considered as signal integrity problems of high speed communication. These new challenges will require fundamental changes how system designs are preceded.

For instance, integration of complex system on a chip demands complex intercommunication between modules – block of gates that are locally connected. The increase of system complexity requires longer global wire interconnects. As a result, the scaling down of wire length through the advances of technology does not hold at the global level. Especially, rising of RC wire delays in the DSM adds “reverse scaling” of interconnects. As global wire delay surpasses gate delay, interconnects between modules surfaces as a difficult task to solve by ad-hoc methods. Additionally, signal transmission on a chip faces an increasingly noisy environment, where noise introduces undesirable effects such as timing variations, cross-talk and interference. Such problems are not only application-driven but also physically bounded ones. The problems need to be dealt with in a multi-level solution to mandate the fixes in a systematic way. This will tend to move computer architecture in the direction of locally-connected, reconfigurable hardware meshes that implement communication networks between modules.

Instead of constructing application specific *micro-networks* some approaches solve the interconnection problems by imposing the generic structure of System-on-Chip BUS or Network-on-Chip (NoC) architectures.

2.6.1 System-on-Chip BUS architecture

The *BUS* was conventionally coined as a term that defines a set of physical wires that is shared among various entities which intend to communicate one another as shown in Figure **2-5** .

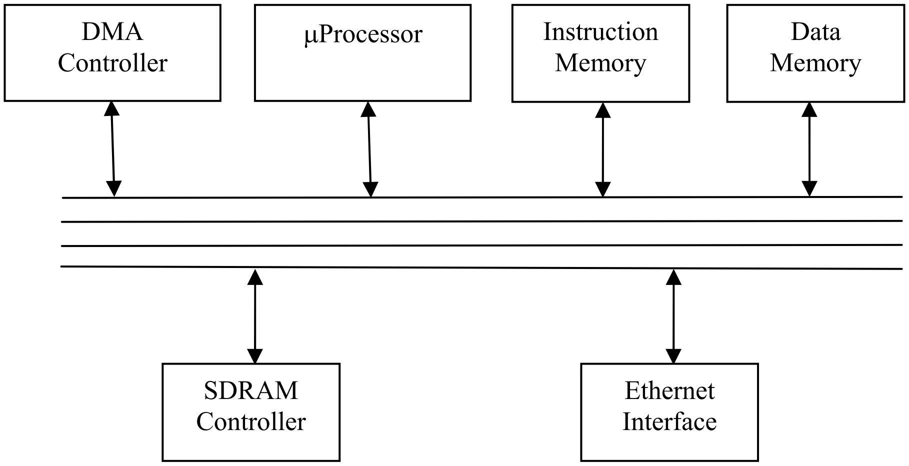


Figure **2-5**: Conventional BUS – an example

Nevertheless, today's BUS used in SoC carries a different meaning. It comprises of multiplexers that establish necessary interconnections among entities and an arbiter that decides the precedence of communication to grant access as shown in Figure **2-6**

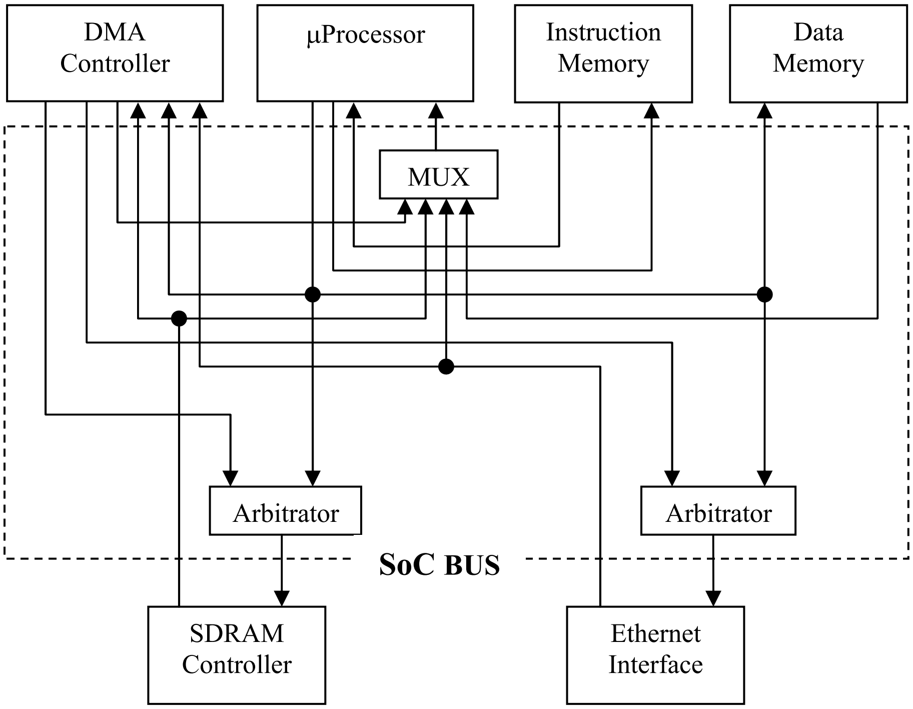


Figure **2-6**: SoC Bus Module – an example

The success of SoC BUS architecture sprouts from popular processor centered designs such as CoreConnect BUS architecture [**27**] with IBM PowerPC processors and AMBA architecture [**28**] with ARM processors. The introduction of SoC BUS allows designers to use SoCs as processor centered collections of IP blocks. Then again SoC BUS architecture has advanced to include more features than the initial architecture envisioned.

In SoC BUS architecture, once the required behaviour of the blocks is defined by transactions of BUS protocol, the level of abstraction to organize the system is elaborated. Thus, the interconnect problem falls under interactions between BUS layers where generic approach allows systematic verification of interconnect architecture. SoC BUS architectures have evolved by embracing the mixture of various traffic types by the means of segmented and tiered BUS architecture. For instance, AMBA deploys four interface protocols called, Advanced eXtensible Interface (AXI), Advanced High performance Bus (AHB), Advanced Peripheral Bus (APB) and Advanced System Bus (ASB) to satisfy various traffic requirements.

As it is shown in AMBA case, the SoC BUS architectures can consist of many local buses. Each local BUS essentially contains a master, slave(s) and an arbiter. The IBM's CoreConnect shown in Figure **2-7** shows a typical architecture of SoC BUS architecture where inter communication is achieved through arbiters and intra communication between local buses via bridges that enable data transfer by master(s) to and from slave(s).

Dynamic reconfiguration with SoC BUS architecture can provide *"plug-and-play"* capability to IP cores that need to be implemented. SoC BUS architectures can:

- Provide technology independent interconnection solutions
- Encourage system level modular designs
- Allow reusable IP cores that are targeted for the same BUS architecture

The above advantages can be generally identified when there is virtual/physical separation of computations and links between them. Because all the communications between IP cores are bounded to work through BUS architecture, it allows technological independence. It also promotes the usage of modular design where modules are the IP cores that are specially developed to be compatible for the BUS transactions.

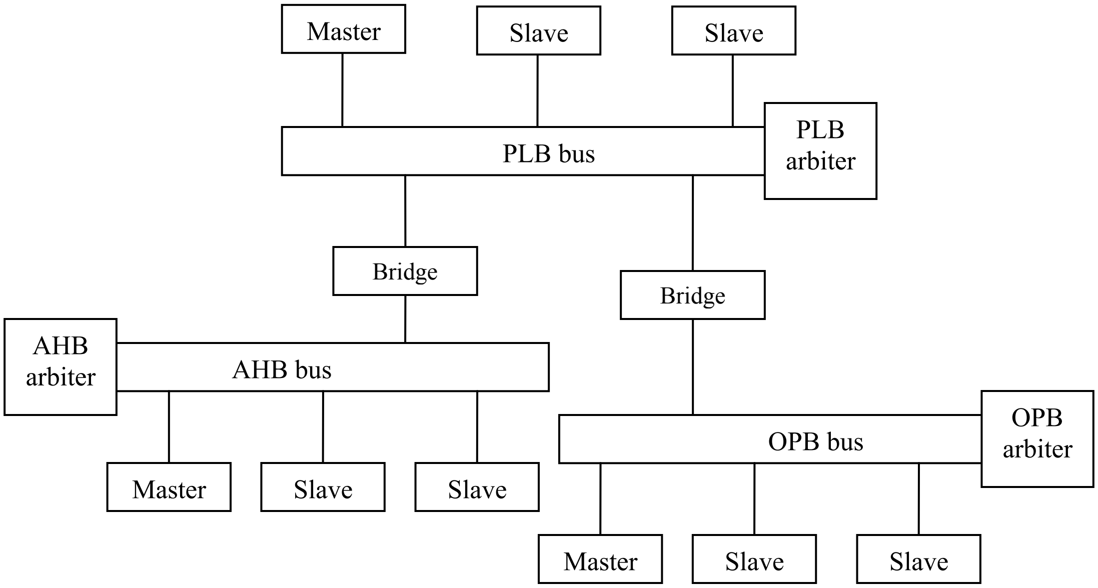


Figure **2-7**: A example of CoreConnect bus architecture

SoC BUS architectures intend to find an optimal way to utilize available bandwidth when the communication requirements of the applications are uncertain. In this case the algorithm deployed on the arbiter plays an important role to achieve the desired performance. Therefore, SoC BUS architectures are mainly concerned with achieving optimal temporal sharing of physical wires (i.e., BUS).

2.6.2 Network On Chip

Once the layering of SoC BUS architecture has become rather complex and the number of IP cores increases beyond typical bandwidth of SoC BUS architectures, the idea of Network-on-Chip (NoC) started to emerge. While the SoC BUS architecture started with the intention of providing the bridge to narrow the gap of realizing the processor-centric SoC designs, the NoC was initiated to provide a solution for new challenges in silicon technology. The core of these challenges is how to successfully implement on-chip applications that require interconnecting more than billion transistors running at GHz. There are several trends that are worthy of noting:

- Increasing operating frequency due to shrinking fabrication technology
- Rising susceptibility of long wires to signal integrity problems (e.g., crosstalk, fabrication uncertainties, noise sensitivities, etc.)
- Difficulty of achieving global synchrony due to growing clock skew
- Increasing power requirements to drive long wires
- Diverging wire delay between local and global communication

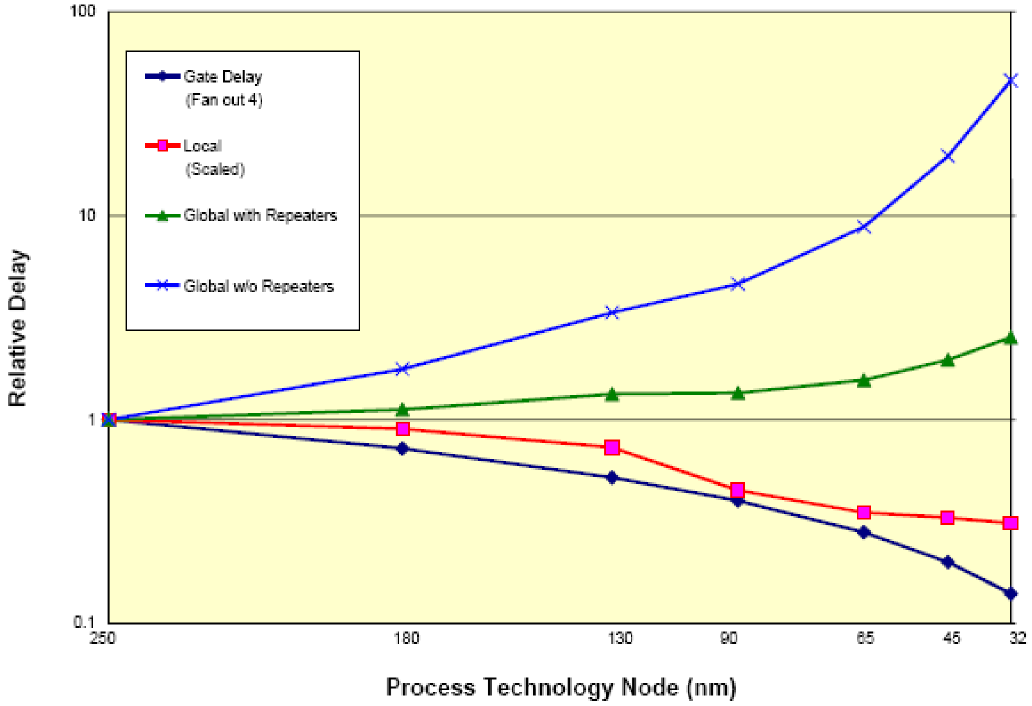


Figure **2-8**: Projected relative delays of local and global wires [**29**]

Figure **2-8** illustrates a growing concern of global communication from the perspective of wire delays [**29**]. The main theme of these trends reveals the difficulties in communicating globally in SoC as process technology shrinks.

The distinctive feature of NoC architectures is the adaptation of network abstraction model (i.e., OSI model) from area networks. The network abstraction model traces the flow of data and works as a vehicle to place the data onto a particular data unit associated with the layer. The OSI network model portrayed on NoC can be shown as Figure **2-9**.

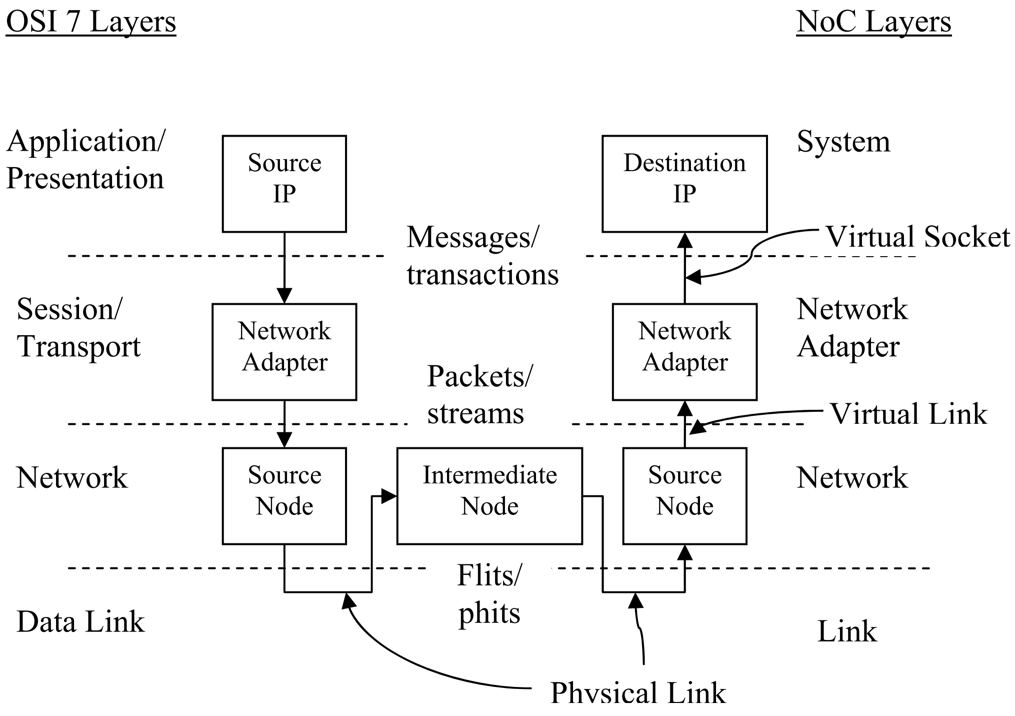


Figure **2-9**: NoC layers vs. OSI 7 layers: redrawn from [**30**]

The seven Open System Interface (OSI) layers are grouped into four layers [**30**] to map the abstraction level of currently undergoing NoC researches. From the top of the OSI model, the roles of the OSI layers are compared with NoCs.

First, the presentation layer is responsible of establishing a context between application entities. Then, using the customary context, the application layer interfaces directly to application services – the context is the format the data is produced and consumed by applications. Therefore, these layers create *virtual* point-to-point communication between end applications. In NoC, IP cores carry the same rationality as the end applications. The cores are constructed without knowing the bandwidth or limitation of links. They are normally constructed by the specifications of a certain communication protocol or generic communication scheme available (e.g., OCP-IP [**31**], VSIA [**32**], etc.). Unlike the application/presentation layers that use *"soft"* approach in the OSI, the system layer uses "hard" approach in the NoC. It also needs a matching communication scheme – so-called *virtual socket*. Otherwise, the NoC can suffer from unreliable data losses during the operation or waste a valuable time at the design stage. Depending on the nature of IP cores, the system layer should create the different types of context (e.g., simple, handshake, burst, pipelined write/read, etc. [**33**]). Since the NoC does not deal with *"real"* application/presentation interfaces – I/O peripherals of the SoC are mostly dealing with transport or network layer data, the construction of IP cores based on the determined and characterized context is the highest abstraction model that belongs to the system layer.

Secondly, the session/transport layer mainly concerns managing the established sessions. The intention of the session layer is to control the connections and then the transport layer provides the reliable data between the opened sessions. In NoC, the session/transport layer can be viewed as the gateway placed to and from the end of network adapter. Therefore, the session/transport layer does not heed what type of IP core it is connected with. When the IP cores are plugged into the network adapter, a session (i.e., *virtual link*) is established via on-chip network.

Thirdly, the network layer provides the procedural ways to transfer data from a source to a destination. The existence of this layer starts from the concept of *"networking"*. The networking is generally accepted as a resource-binding idea. The network model is the idea to optimize resource for establishing communication among multiple entities. With cost and performance considerations, the network model settles with a topology that results in limited (but sufficient) interconnects between network entities. The area network employs the means of extending the reachability of the data by extending networks that have limited interconnects. When there is no direct route between the source and destination, routers operate to relay the data via the extended network to reach the destination. The syndicate of the network is achieved by routers who manage a portal to the extended network. In NoC, the network layer focuses on reach-ability of network limited by the links of the deep submicron process technology. As long as the data is concerned, the network layer only inserts intermediate nodes to extend the links that are bounded by physical limitation.

In addition to all the above, there are several conceptual and real gaps that distinguish the OSI model from the NoC. Due to the embedded nature of applications that are dealt in the NoC:

- The area of applications in a device is very narrow.

- The NoC is not a general purpose system, but is rather an application-specific device.
- Except for the cases of upgrades and mode changes, the structure of applications are mostly static not dynamic.
- The number of network entities that require communication does not scale much for the NoC as it does for the area network.
- As the number of gates increases, the NoC can implement more threads to utilize available areas but the number of entities in a thread would not increase much.

Unlike the area network mediums (e.g., optical fiber, coaxial cables, etc.):

- The operating frequencies of cores (e.g., network entities) are very close to the bandwidth of physical link.
- To be attractive for computation-intense applications, the NoC utilizes largely available link resources to massively parallelize computations.

The area network utilizes high bandwidth through strictly technology dependant mediums. Since the NoC shares fundamentally the same fabrication technology, it is technically more challenging to manufacture the links that are superior to the computational resources. There is much research to look for physical link capability to achieve a NoC solution that is comparable with the area networks. One of these solutions is Globally Asynchronous Locally Synchronous (GALS).

A GALS system contains complex computational blocks that are synchronous by themselves. These coarse blocks are interconnected by globally asynchronous communication. The globally asynchronous communication requires several analog blocks (e.g., PLL, A/D and D/A converters) to recognize the data and establish the communication. Due to the operating frequency that GALS promises, it provides the benefits that the area network offers. They are such as reduction of power consumption due to globally clockless designs and shrinking the number of wires employed for global communication due to serialization of links.

While the SoC BUS architectures employ shared medium approach that requires arbitrations and control transactions, the NoC utilize point-to-point network that implement its responsibility into different layers of the architectures.

The NoC will provide the communication structure for easy integration of various applications if the costs of the Reconfigurable SoC system become cheap enough. Yet, the majority of today's applications that are implemented in SoC systems are quite narrow. They are typically timing-critical and computation-intensive applications that are identified as real-time and stream applications. Other applications are normally implemented in micro-processor systems. The system of real-time and stream applications ought to maximize the capability of existing hardware to achieve the desired performance of the applications.

In area network, the unprecedented choice of medium (e.g., optical fiber) allows the communication between systems to withstand the overhead resulted by protocols. On the contrary, the SoC systems always struggle to squeeze out every ounce of bandwidth that it can reliably provide.

As we witnessed in the previous sections, the main focus of the RC research have been the study of hardware limitation and how to resolve them using architectural solutions. To study the main trends in RC industry, one first takes a look at the existing RC systems from their architectural point of view.

2.7 Reconfigurable Computing Systems: Architectural point of view

Hartenstein [**34**] presents his paper by classifying how hardware of each reconfigurable system is organized. Since the information of each system is presented by the merits of combined choices (e.g., granularity, topology and architecture) of the reconfigurable system, it is referred as *subjective -view*. The architecture-centric view of systems identifies the principal computation requirements and tries to bring the optimal solution for the limited set of operations. The reconfigurable systems tend to focus on the effectiveness of hardware solutions. Thus, the performance of the system greatly depends on how applications can be converted for the given hardware solution.

One of ways to look at the system is by their granularity. The increase of granularity is rooted from the computational datapaths that are naturally more than 1-bit. Due to the overheads incurred by configuration time and memory and the complexity associated with place and route problem, it is encouraged to utilize the hardware that is closely matched with the granularity of datapath.

From the beginning of the RC history, the main debate of the RC architecture was focused on what granularity the RC systems should have. Due to many reasons such as routing areas, power consumption, routability and configuration time, they are mostly implemented as coarse-grained architectures. The summary of these systems is listed in Table **2-1** in terms of their granularity.

Table **2-1**: Granularity of coarse-grained reconfigurable architectures

Project	Granularity
DReAM	8/16 bit
PADDI	16 bit
PADDI-2	16 bit
RaPID	16 bit
REMARC	16 bit
MorphoSys	16 bit
CS2000 family	16/32 bit
PipeRench	128 bit

When the Table **2-1** represents link/datapath granularity, there can be the granularity of computation unit. Since the granularity of the model is bounded by the granularity of links, the granularity of a computation becomes the same as link size. For example, DReAM [**35**] employs 8-bit Reconfigurable Arithmetic Processing Units. CS2000 family [**36**] offers the multi-protocol multi-application reconfigurable platform with a 32 bit RISC core as a host. MorphoSys [**37**] has “TinyRISC” with 16-bit interface. As it is specified in Table **2-1**, the system employs the granularity that is optimized the link capacity of the system except the cases of Pleiade [**38**] where the system is synthesizable by the applications.

Topology of the systems also demonstrates the variety that exists in communication infrastructure of the RCs as shown in Table **2-2**

Table **2-2**: Communication topology of reconfigurable architectures

Project	Topology
PADDI	central crossbar
PADDI-2	multiple crossbar
DP-FPGA	inhomogenous routing channels
KressArray	multiple NN and bus segments
RaPID	segmented buses
Matrix	8NN, length 4 and global lines
RAW	8NN, switched connections
Garp	global and semi-global lines

The topology of reconfigurable systems shows a mixed reflection of homogenous [**39**] [**40**], hierarchical [**41**] [**42**] [**43**] [**44**] and switched [**45**] routing structures. Table **2-2** indirectly shows that there is no one topology that meets the requirements of all applications. However, there is a strong trend to divide global and local communication to overcome the implementation constraints laid by deep submicron effects. It is very interesting to look at the interconnection structure changes occurred in Programmable Arithmetic Device for DSP (PADDI) [**39**] [**40**] from a full crossbar switch, to restricted crossbar with a hierarchical interconnects.

The hardware specific view of reconfigurable computers reveals the architectural differences among the RC systems. There is no universal agreement on which combination of architectural approaches is better for certain applications. However, one believes that these RCs will continuously evolve and reflect the trends of today's applications by applying different granularity, communication and network topology and adapting to a new architectural solution.

In terms of hardware platform, one does not assume that the proposed system employs one of discussed platforms. Instead one uses one of the commercially available Field Programmable Gate Arrays (FPGAs) to implement the discussed class of applications (e.g., real-time and stream). One assumes that the chosen FPGA can accommodate anticipated performance of underlying applications.

In the subsequent section, one will look for an efficient way to implement the trends of today's applications as they become multi-task and multi-mode.

2.8 Optimizing Reconfiguration

Many researchers in the name of multi-tasking have investigated on the subject of reconfiguration [**46**][**47**][**48**][**49**][**50**]. Their research focuses on continuation of tasks that are segmented (e.g., multi-task) in the time domain. Specifically, the papers [**46**][**47**][**49**] present the operating system and platform support for preemption and recovery of states between tasks. The papers [**48**][**50**] guides one through the sketch of the RCs from the operating system (OS)'s point of view. These papers assume that the reconfigurable device does not have enough logic density or its cost becomes too expensive to implement all in hardware. Once these assumptions are in place, task scheduling and inter-task communication along with preemption/restoration becomes an essential part of the OS tasks [**51**].

Conversely, one assumes that our RC platform (e.g., FPGA) is able to accommodate all the requirements at the times, but not over all times. Multi-task in our assumption is an application-driven phenomenon, not an OS-driven task. Reconfiguration of processing resources would bring the same cost benefits that the above approaches mention. Invoking of reconfiguration is also not initiated by internal trigger but serviced by external system events. Because data units in stream are independently executed, there is no need to restore the state of the system. If the above RC systems use the architectural specific reconfiguration, our approach attempts to discover the natural flow of multi-tasks in the applications and then matches it with the reconfiguration capability of our RC system.

2.9 Reconfigurable Computing System: Application point of view

If hardware specific reconfigurable computing looks for a new hardware solution that is powerful enough to meet the requirements of target applications, application specific reconfigurable computing – so-called *predicative view* of RCs, focuses on searching more efficient way of designing RC systems.

Because each application is different, we need a model to describe the undermining applications and study the effectiveness of our approach based on the model. The modeling taxonomy provides a means to categorize applications according to a set of attributes depending on their characteristics. Thus, a modeling of the applications that share the common traits can be constructed.

We adapt the model taxonomy used in [**52**] that is widely accepted by Virtual Socket Interface Alliance (VSIA) [**32**] and Open Core Protocol Intellectual Protocol (OCP-IP) [**31**] community. The model taxonomy in [**52**] consists of four main axes that are not completely orthogonal to each other. The model can have both internal and external resolutions. However we focus on external resolution of a RC system as they should define interfaces between reconfigurable blocks.

The model taxonomy has four axes. These axes are expressed in resolutions that differentiate the applications. The axes are described in temporal, data, functional and structural domain. The temporal axis defines the timing in the model. Its resolution can be divided into seven different categories, “partially ordered events”, “system event”, “token cycle”, “instruction cycle”, “cycle approximate”, “cycle accurate” and “gate propagation accurate”. Because we are interested in modeling the communication between cores on a chip, the temporal resolution should be fairly high. Additionally the granularity of on-chip core processing and their precedence in today’s applications are normally described in high temporal resolution. We only focus on high temporal resolution such as “partially ordered events”, “system event” and “token cycle”. For instance, “partially ordered events” can describe the systems that can consist of many threads that are independent and concurrently executable. There is the precedence of execution among tasks without exclusive timing assignment. “system event” can specify start and end times of system functions. When the operations are executed by the inputs of the same data format not by the content of data, it is referred to as “token cycle”. The low resolutions, such as “cycle approximate”, “cycle accurate” and “gate propagation accurate”, are used to describe for a particular RTL or internal resolution of IP blocks. The high temporal resolutions can be precisely specified by the definitions used in the section **2.5.2** and available precedence of processing core blocks.

The data resolution represents the precision, with which the formats of values are specified in a model. Considering high temporal resolutions “token” should be assigned as the data resolution. “Token” is considered to be the highest abstraction level for data representation without implementation information such as structure, size, value and so on. However, “token” in our RC model, carries the structure of stream application that are defined under the section **2.5.3**. The structure of stream application defines the order of coarse computing operations. The content of data does not change the order of operations unless it is classified as the system events.

The functional and structural resolutions do not play an important role in finding an optimal design solution for RC systems because they do not contribute to the organization of computing operations. These resolutions tend to show how core blocks are internally represented and described.

As it is seen in Figure **2-10**, each temporal resolution can de described by an associated timing scale [**52**].

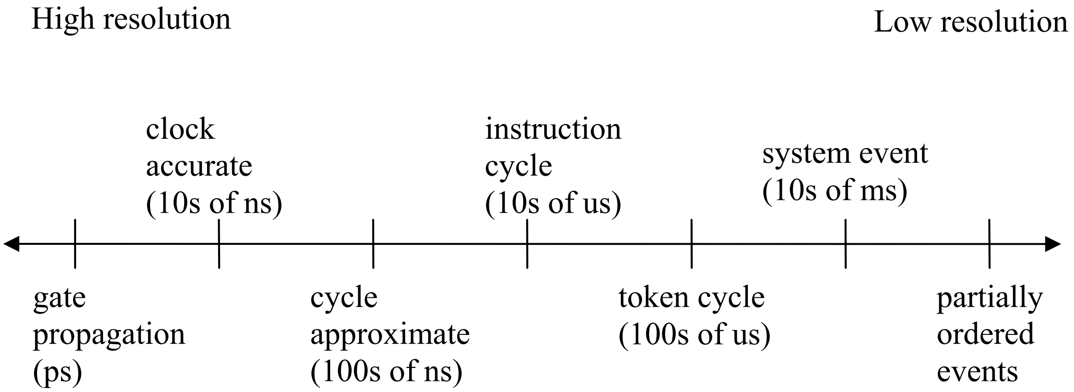


Figure **2-10**: Temporal Resolution Axis

Especially when the temporal resolution is "token cycle" and less, the timing allowance is in the same timing scale as reconfiguration time of RC systems. This combination allows the reconfigurable systems to reuse processing resources that are available between core operations via reconfiguration. All of these are achieved by raising temporal abstraction level as mentioned in section **2.3**. Because of non-orthogonality between temporal and data resolutions, they should linger in the lower region. The functional and structural resolutions depend on how the system can be described by smaller elements.

In order to achieve the goal of designing a cost-effective reconfigurable system, we explore the redundancies that are available from the applications and utilize the timing redundancies as system reconfiguration. The first step is to recognize what are these redundancies. Today's applications have two faces: one is static face that requires faster execution of a fixed set of operations; the other is dynamic face that demands coarse changes between static faces. To exploit the benefits of the above traits, one introduces the 5th model taxonomy axis – i.e., reconfiguration resolution. The reconfiguration resolution is both physical constraint and the specification of the system. Physically the reconfiguration resolution is deeply coupled with the types of computing system. For instance, the reconfiguration resolution of microprocessor is one clock cycle that allows the change(s) of hardware while the some of the RC can change "context" of the system within a clock cycle. For FPGAs, the reconfiguration resolution is generally accepted by the time with which it takes to program a tile of reprogrammable logics. However, the granularity of the tile seems to change as the FPGA systems try to reflect the current needs of the electronic market. All of these are based on the assumption that the physical configuration speed meets the fundamental unit of data that the system deals with. Thus, it leaves with the question, what granularity of reconfigurable system is best suitable for a particular *workload*? The answer should not be a unique one.

The designing RC system with the fine-tuned reconfigurable/temporal resolution results in a cost-effective solution compared with ASIC approaches where the detail timing of temporal resolution is determined by system events and reconfiguration resolution is set by RC technology.

2.10 Summary of RC taxonomy

The subjective view of the RCs generally looks at the maximum performance/area that hardware can provide and accommodates the applications onto the platform to carry out. Conversely, the predicative view of the RCs analyzes the applications and partitions them into tasks that can be reconfigured with the given platform. Otherwise, they are laid as parallel processing tasks. In order to do so, the predicative view of the RCs requires identifying two faces of applications. Once they are clearly separated, the reconfiguration resolution of the chosen RC hardware system (e.g., FPGA) can be matched with other modeling axes, especially temporal and data axes. In this case, we can reuse the processing resources that reduce the overall cost of system without compromising performance.

It is difficult to compare the impact of the various RC techniques mentioned in this chapter because the target architecture and the requirements of underlying applications vary. This thesis makes an attempt to assess the benefits of application centric RC design methodology with a particular set of applications (i.e., real-time and stream applications). Moreover, it examines in detail the dependence of our methodology on the relevant characteristics of the underlying FPGA architecture. The following section will study a particular FPGA architecture on which our experiments are based.

2.11 Field Programmable Gate Array

As consumer devices evolve towards increasingly complex integrated circuits and time-to-market pressure continues relentlessly, re-usage of pre-verified component (e.g., IP cores) becomes mandatory. In this chapter we studied the RC systems in the hope that they can provide the gains in productivity and the savings in system costs through reconfiguration. Using RC systems we thrive for a result in which the system is shared across multiple applications and its overall cost becomes lower than the ASICs.

2.11.1 Reconfigurable Hardware based on FPGAs

One of the most commercially successful reconfigurable devices is Field Programmable Gate Array (FPGA). An FPGA consists of arrays of programmable logic blocks whose functionality can be determined by configuration streams. The configurable logic blocks are also connected by a set of programmable routing resources. FPGAs can be programmed to map custom logic circuits. FPGAs can be programmed in three different ways. They are: full static configuration; partial static reconfiguration; partial dynamic reconfiguration.

However, not all FPGAs can support all modes because of how configuration memory is connected and what memory technology is used in the FPGA. They are three different types of programmable technology that are readily available: anti-fuse; flash; SRAM.

As the technology states, anti-fuse based technology is one-time programmable. It has negligible delay and relatively small area overhead. Actel Corporation [**53**] is one of the leading producers of anti-fuse FPGA that are widely used in space applications. Flash based technology can maintain programming even when turned off, but requires more manufacturing processes. Flash based FPGA are produced by many companies. However Lattice semiconductor Corporation [**54**] is one of first who was successful on the market.

SRAM-based FPGAs can provide faster reconfiguration when they are compared to FLASH-based FPGAs. Additionally SRAM-based FPGAs can provide dynamic reconfiguration. The dynamic reconfiguration allows configuring a part of logics when others are still in operation. They are two dominant commercial companies who make SRAM-based FPGAs. They are Xilinx Incorporated [**55**] and Altera Corporation [**56**].

In this thesis we implement the system using SRAM-based FPGAs. Due to the support for dynamic reconfiguration, we decide to use Xilinx FPGA families, particularly Virtex-4 FPGA [**57**].

2.11.2 History of run-time reconfigurable Xilinx FPGAs

In order to understand the rationale behind reconfigurable technology and architecture that Virtex-4 FPGA has, we need to study the history of FPGA development.

One will investigate the construction of homogenous logic blocks used in FPGA, and routing structure that interconnects logic blocks and how reconfiguration mechanism has evolved to reflect the above developments in FPGAs. Our investigation will be limited to Xilinx Virtex FPGA family, not across multiple FPGA platforms, because of different strengths that they appeal for different tasks. One starts with the predecessor of Xilinx Virtex FPGAs that is the XC6200 FPGA [**58**]. Xilinx XC6200 FPGAs allow direct access of configuration memory for static or dynamic reconfiguration via parallel configuration interface that is very similar with SRAM memory access. Because of homogeneity, the FPGA is composed with identical basic cells. Switching fabrics consists of basic cell blocks and routing multiplexers shown in Figure **2-11**.

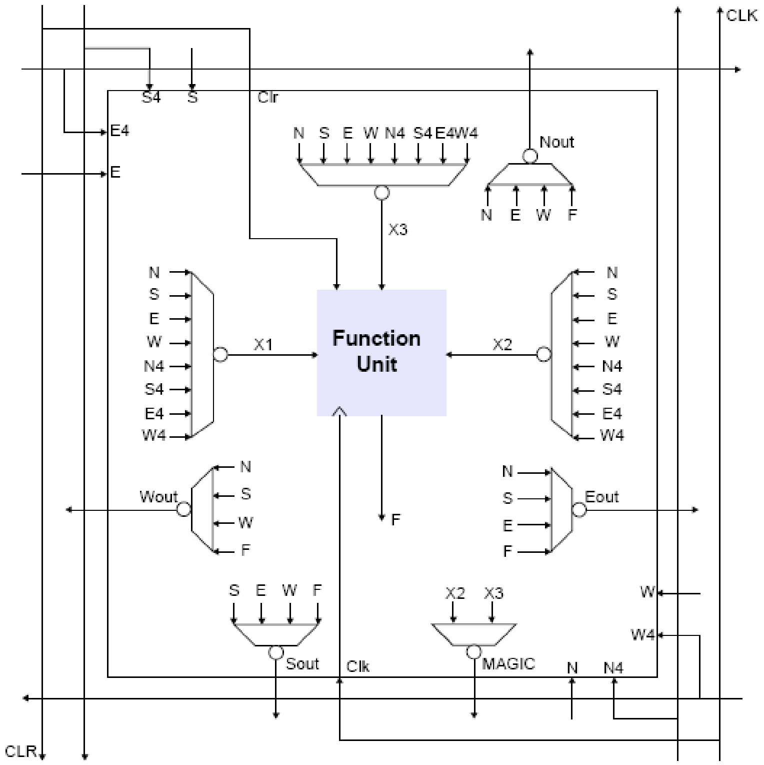


Figure **2-11**: XC6200 Basic Cell [**58**]

The basic cells in the array have inputs from the length 4 wires associated with 4×4 cell blocks as well as their nearest neighbor cells as the name North (N), South (S), East (E) and West (W) stands out. The multiplexers within the cell are controlled by bits within the configuration memory. The function unit is a simple 2-input combinatorial logic block with a flip flop as shown in Figure **2-12**. The XC6200 FPGAs hierarchically cascade routing resources by expanding from 4×4, 16×16 and to chip-length lanes.

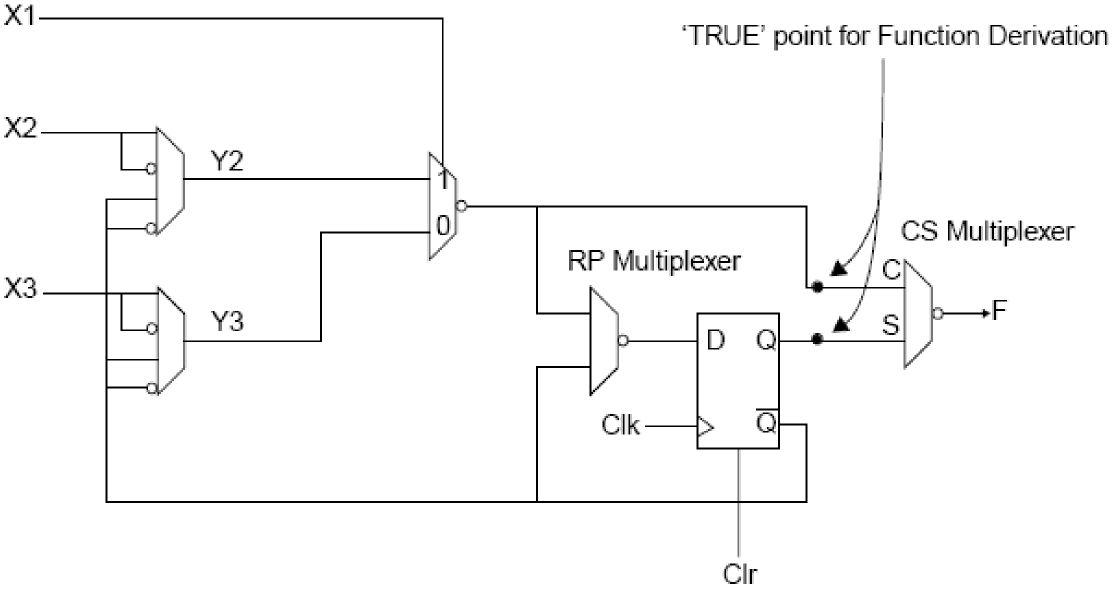


Figure **2-12**: XC6200 Function Unit [**58**]

In XC6200 FPGA series [**58**], the configuration of each cell was accessible as if one accesses RAM memory with a valid address and the routing structure grew consistently as a group of 4 multiple cell blocks. As one of the first, the XC6200 FPGA series allow the user to selectively modify the contents of configuration memory, so-called partial reconfiguration. There were several applications that utilize the partial reconfiguration feature of XC6200 [**59**] [**60**] [**61**].

After the XC6200 FPGA series, similar FPGA architectures started to appear. The Virtex family FPGA is the first case as shown in Figure **2-13**.

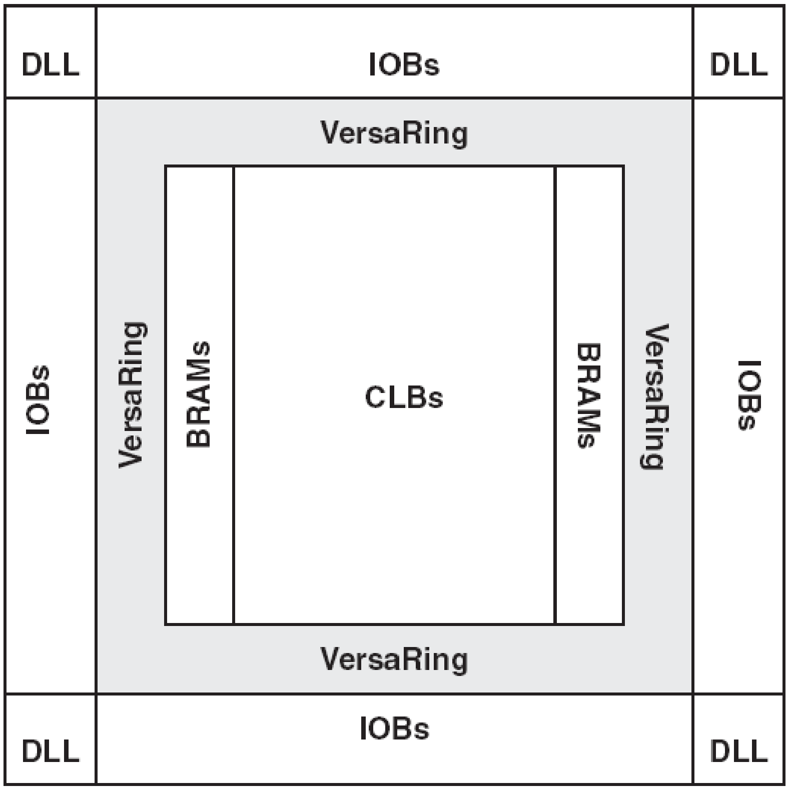


Figure **2-13**: Virtex Architecture Overview [**62**]

Starting from Virtex FPGA family, each Xilinx FPGA device started to use the terminology of Configurable Logic Blocks (CLB), Input/Output Blocks (IOB) and Block RAMs (BRAM). CLB is used as the equivalent terminology with cell blocks in XC6200. The Virtex FPGAs also include clock resources, local/global routing and configuration SRAM and configuration controller. Figure **2-13** illustrates the organization of these resources in a device (i.e., Virtex FPGA Family). While the BRAM and clock resources have its own routing to interconnect with the rest of the device, the CLBs need to communicate via a general routing matrix. This switch matrix comprises an array of routing switches located at the intersections of horizontal and vertical routing channels.

2.11.3 Evolution of Virtex FPGAs

As the process geometry shrinks, there is a growing possibility to implement more complex applications. While maintaining the overall FPGA architecture, the details of **CLB, routing** and **configuration** have been evolved to accommodate increases in power consumption, wire delay, etc. Associating with the complexity of embedded systems, there have been needs for architectural changes.

2.11.3.1 Configurable Logic Block

The major merit of the CLB is homogeneous structure. Because of the uniform structure of the CLB, it is possible to implement configurable/swappable hardware components that are virtual. “Virtual” components represent hardware components that are synthesized using Configurable Logic Blocks (CLBs). Thus, the same CLBs can employ different “virtual” components – so-called, Virtual Hardware Components (VHCs). The CLBs in Virtex FPGAs started to be implemented as 4-input Look-Up Tables (LUT) as shown in Figure **2-14**.

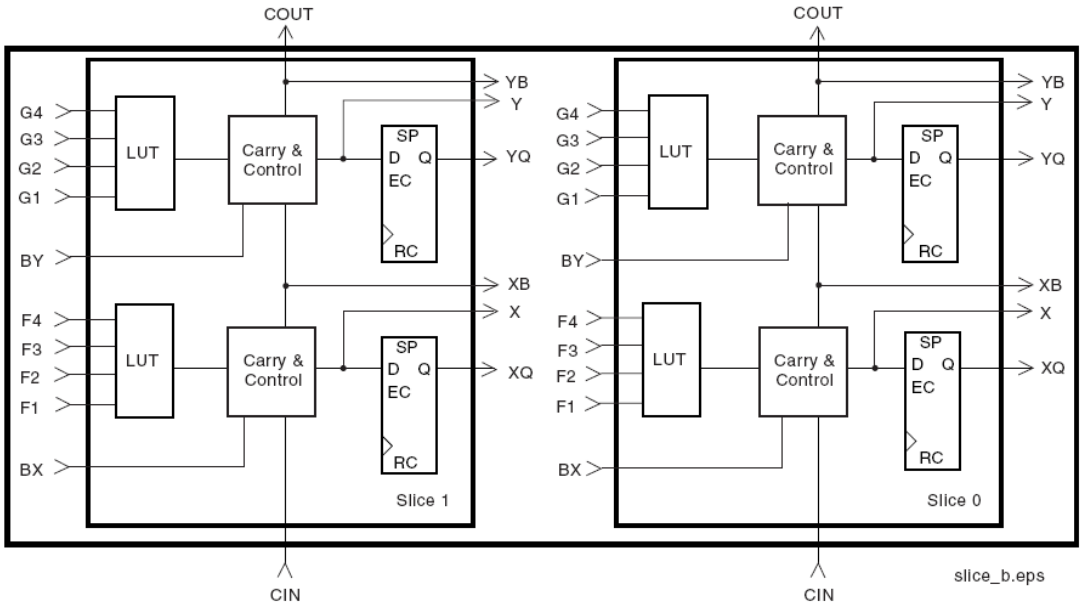


Figure **2-14**: 2-slice Virtex CLB [**62**]

Figure **2-14** from [**62**] illustrates a CLB of Virtex FPGAs that can work as logic cells, LUTs and storage elements. The CLB of Virtex-2 FPGAs [**63**] consists of 4 slices which can serve more functionality shown in Figure **2-15**

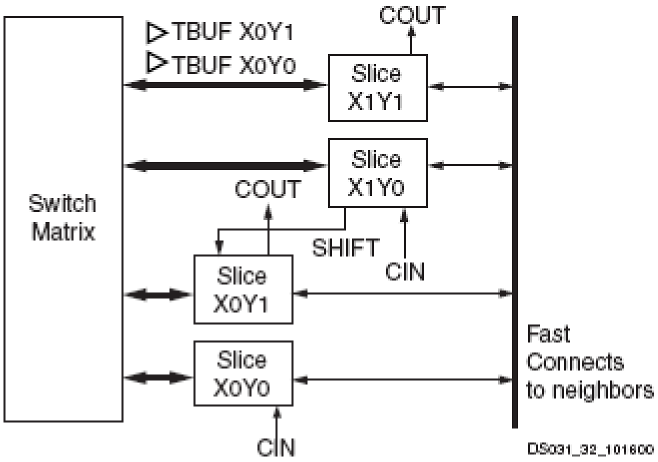


Figure **2-15**: Virtex-2 CLB element [**63**]

Each slice is capable of working as functional generator, 4-input LUT, 16-bit distributed RAM, 16-bit shift register. Associated multiplexers allow the CLBs to behave as wide multiplexer and fast lookahead carry logic and arithmetic gates. Each CLB employs two 3-state drivers that can drive chip-wide buses. The structure of CLB in Virtex-4 FPGA appears to be the same as in Virtex-2 devices. However as the process geometry decreases 3-state buffers were taken out from CLBs in Vritex-4. As the logic density increases for a chip, the chip length communication lines became too difficult to achieve in terms of their power consumption and wire delay. For example, Virtex-2 FPGA has maximum 104,882 logics cells in FF1517 while Virtex-4 FPGA has maximum 200,448 logic cells in FF1513 package. Twice the logic is available in Virtex-4 FPGAs than Virtex-2 FPGAs in the comparable packages. Virtex-5 FPGA [**64**] took a leap toward implementing 6-input lookup table by following the market trend.

2.11.3.2 Routing

When there are more logics to be connected in the given area, it has become increasingly difficult to construct routing structure that spans through an entire FPGA as in XC6200 series. The routing structure is highly hierarchical to balance the overhead. Xilinx divides the routings into two categories: general purpose routing and dedicated routings. General purpose routing uses the connections between horizontal and vertical switch matrix to connect adjacent CLBs, hex lines and longlines. The hex lines connect to a CLB that is located 6 positions away. The longlines span a full vertical and horizontal length of the device. The construction of routing structure in FPGAs is possible using switches. The switches that are used in Virtex FPGAs are called as subset or disjoint ones. These switches connect to four ports that are incident on a point. By providing configuration information to these points the switches can be differently connected.

The initial Virtex FPGAs implement all of the above to provide fairly straightforward interconnection strategy. However in the subsequent family of FPGAs the lots of routings become dedicated due to insertion of many custom hardware blocks (e.g., multiplier, processor, DSP slices and memory blocks). One of changes is that longlines do not get to span over a full device. It becomes inevitable to go through a few switches to reach from an input of one side to an output of the other side. Additionally, because of reach-ability of global clocks, there are designated areas that are confined with a particular clock in the system. It becomes impossible for all processing blocks in a FPGA to share the same high-speed clock.

2.11.3.3 Configuration

The first configuration mechanism was brought by Xilinx was XC4000 FPGA devices [**65**] that provide a serial access to the configuration memory. The serial access allows incremental writing of configuration data. Hence, the XC4000 FPGA requires programming the entire chip. Then there was XC6200 FPGA family. Due to the increase of configuration memory size, the reconfiguration mechanism is changed to combined address and data type. The configuration memory organization of XC6200 FPGAs is a reflection of SRAM memory access. Due to random access capability it allows partial reconfiguration to be carried for the devices. Not only how to access the data is changed but also the unit of reconfiguration is changed to reduce the size of configuration stream. If the unit of reconfiguration was one basic cell in XC6200, it is changed to be a frame[11] in both Virtex and Virtex-2 FPGAs [**66**]. Unlike previous Virtex FPGAs in which the frame spans over the entire height of the devices, it is reduced to be 16 vertically aligned rows of a frame in Virtex-4. The contents of configuration frames for a CLB can be represented graphically as in Figure **2-16**.

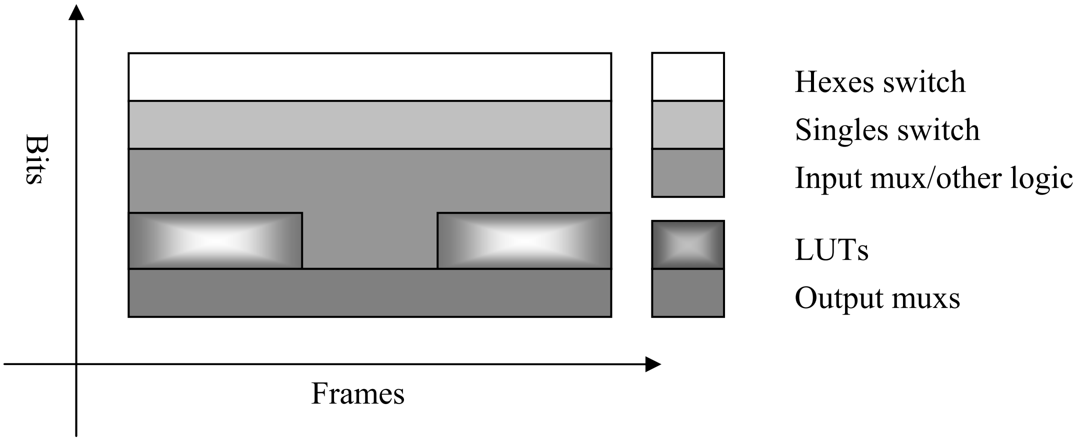


Figure **2-16**: Graphical representation of one CLB worth configuration data for Virtex FPGA

We expect the structure of CLB, routing and configuration to be evolved by reflecting the needs of market.

[11] Frame is a subsection of one column in Xilinx Virtex FPGAs.

Moreover, there have been specialized platforms that provide other means of configuration. One of them is multi-context FPGAs. The multi-context FPGAs have more than one configuration memory plane. By activating one plane at a time the multi-context FPGAs can switch configuration ideally in one cycle. There is also a technique called configuration caching. As the name indicates, the technique reserves a part of configuration file in a separate memory to use it for configuration of later circuits. The reconfigurable platforms such as Garp [**41**] and Chimaera [**67**] use the technique in their implementation. Architectural techniques such as pipelined reconfiguration [**68**] and wormhole reconfiguration [**69**] are also available.

All of these approaches are the attempts to reduce the reconfiguration time. Due to continuous efforts to increase configuration speed with partial reconfiguration, the configuration time has reduced considerably. However, because the amount of logic density available in FPGAs is continuously increasing, the configuration time of the entire chip should take longer (e.g., 100ms to 10s seconds). The only way to reduce or optimize the reconfiguration time of the applications is to employ partial reconfiguration. Because of known and fixed size of functions in the applications, reconfiguration of these functions becomes feasible within their operations to achieve the maximum efficiency of reconfigurable computing.

It seems that evolution of Xilinx FPGAs show not only the advancements in silicon technology but also its limitations. Accommodation of these changes is an expected process of system design where the ideas meet practical resolution.

2.12 Summary

In this chapter, we studied various hardware approaches that the RC investigates, but, these approaches were focused on hard fixes of the problems through architectural solutions. Because of the inflexibility of the solutions, they tend to be too expensive to fabricate the solution for a specific application. Using reconfigurable devices and implying multi-task and multi-mode conscious design methodology from the application's point of view, it becomes feasible to reuse processing resources without jeopardizing the performance of applications.

Nonetheless, a new system design methodology requires several design steps such as recognizing system events, partitioning tasks and identifying reconfigurable blocks before it proves to be useful. The subsequent chapter tries to suggest more intriguing reasons behind the FPGA based designs as a reconfigurable system. Then, the thesis will explore the strategies which investigate the method of each step.

Chapter 3

Overview and Analysis of Effectiveness of Computing Systems

This chapter focuses on the qualitative estimation of trends and technologies for effective implementation of data processing systems. Though the estimation does not provide a quantitative analysis of economical point of views, the qualitative examples are added to provide better illustrations and understandings of current market trends and tendencies. This chapter guides the readers to the qualitative analysis that discovers the idea of cost-effectiveness for reconfigurable computing systems by virtualizing data processing resources for a multi-task and multi-mode workload.

3.1 Introduction

In recent years, changing electronic *market trends* as well as fast shifting *fabrication technology* have challenged conventional computing approaches (e.g., microprocessors and ASICs). Upon the rising of *new applications* that demand integration of multiple functionalities, there is a certain quality (i.e., virtualization of processing resources) that advocates reconfigurable computing to be more competitive.

This chapter provides a general overview of system effectiveness by materializing changing electronic market trends and shifting fabrication technology into the system costs. When the degree of an application’s feature such as multi-functionality becomes more apparent, reconfigurable computing can be more cost-effective than conventional computing approaches.

Considering high-speed requirements of the underlying applications, the cost-effectiveness of two systems that are capable of providing the services for the applications are discussed. One is an Application Specific Integrated Circuit (ASIC) system. The other is a Field Programmable Gate Array (FPGA) system as an example of reconfigurable computing.

3.1.1 Cost-effectiveness of computing system

If the cost effectiveness of a computing system is the degree of difference between the total cost to construct the system and the services received from the investment, one should evaluate the components and conditions that impact their values. First, one can formulate the cost-effectiveness of the system as the difference between the services provided per workload, $S_{workload}$ and the total cost, C_{total} to build the system as in Eq. **3-1** .

$$CE = S_{workload} - C_{total} \qquad \textbf{3-1}$$

The number of services provided per workload, $S_{workload}$ is counted by the instances of applications that are different over time – Refer to section **2.4.2**. In ASIC approaches, $S_{workload}$ does not signify any particular benefit because it represents as a choice of multiplexer occupied in hardware of coarse-grain architecture (e.g., for multi-task and multi-mode applications). The ASIC approaches do not impose any distinction implementing the choices in coarse or fine-grain architecture. However, in FPGA approaches $S_{workload}$ might be recognized as the sequence of reconfigurable blocks that are organized in time domain.

3.1.2 Three levels of cost-effectiveness

The total cost to construct a computing system, C_{total} can depend on various conditions. To make their contributions clear, one first divided the cost-effectiveness into three different levels. They are:

- Field level;
- Design level;
- Manufacturing level.

The cost evaluation at the field level conventionally requires analyzing the effectiveness of the system architecture to deal with a given application. For the instruction based systems, it means how fast the system switches to carry out the given set of instructions. For an ASIC, it means how much resources it employ to process the required functionality. For an FPGA, due to a unique feature of some FPGAs (i.e., field reconfigurability), the FPGA platform can reuse limited processing resources where the hardware depicts *"function(s)"* in a given time slice. So, referring to section **2.4.2** it means how effectively the system utilizes the given resources to adapt for the workload that consists of timely spaced applications. As a result, it can *decrease the functional cost per transistor* by reusing the transistors in the temporal domain. At the field level, the types of applications should be researched to find out their applicability (how to measure the value of $S_{workload}$) on to the FPGA system.

At the design level the FPGA design looks already promising because of its low developing cost compared with the ASIC. Since the design level cost are mostly test and verification related with hardware, the IP core-centric design becomes mandatory, which pays most attentions to the human resource for creating pre-verified and socket standardized cores. The IP core centric design allows the associated costs to be redistributed over many different projects. At the same time it diverts the designer's attention to implementing communication by excluding low-level functional verification. Moreover, the FPGA designs can adapt to a new product life cycle much easier than the ASIC designs because of their shorter time-to-market time – by excluding test, verification and hardware manufacturing phase out of the design cycle.

The cost evaluation at the manufacturing level starts to appear more important due to deep-sub micron effects that play a major role under smaller geometry. Hence, the issue becomes how to control the increasing cost of smaller geometry while decreasing functional cost per transistor – higher density. However there are still remaining problems even after the cost is controlled. One of them is reliability. Beckett in [**70**] states that the system within 1.3 years would exhibit 90% failure rate for transistor densities of 10^9. As a result, testing and verification will be a major cost in manufacturing integrated circuit devices. The architectural level implementation should be considered to guarantee better yielding rate. The regularly structured devices such as an FPGA can easily encapsulate defect-safe implementation to manufacture the devices more successfully. The failures at the field can also be avoided by excluding the defected area and reconfiguring the functionality into the rest of chip.

3.2 Motivation

3.2.1 A perspective of computing history

There have been many different systems that were successful in the history of computing industry because the systems were able to reflect the trends and demands of the market at the times as shown in Table **3-1** .

Table **3-1**: Architectural eras

Type	Example	Start
Pioneer computer	Mark I, ENIAC	1940
Classical computer	Univac, IBM 704	1950
Supervised computer	IBM Stretch	1955
Supercomputer	CDC 6600	1960
Time-shared computer	GE 645	1965
Minicompuer	DEC PDP8, PDP11	1970
Microprocessor	Intel 8080A	1975
Workstation	Motorola MC68000	1980

Let us look at the trends of current market to *anticipate a perspective of future systems and a new design methodology*. The best way to predict where one is going is by looking at what the history shows. Makimoto's wave [**71**] shown in Figure **3-1** successfully exhibits the past trends, which layout large repetitive cycles that bounce between standardization and customization. Makitomo's wave moves towards standardization when large numbers of new inventions appear. Once the standardization is mature, the need for product differentiation appears, throwing the swing towards customization [**72**].

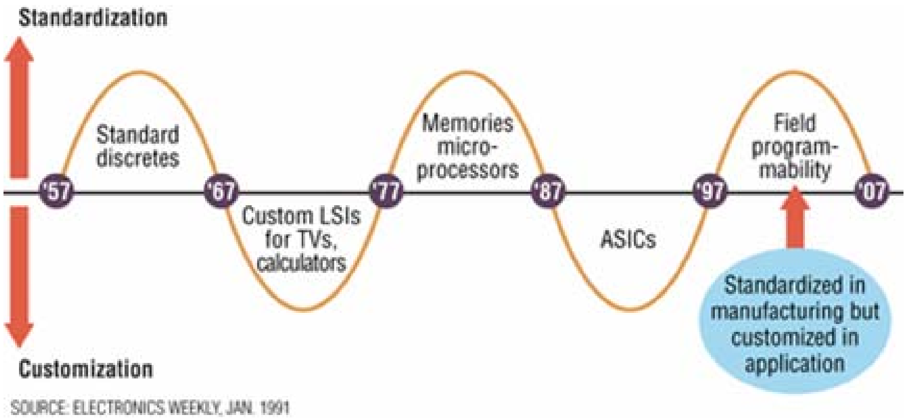

Standardization
Standard discretes
Memories micro-processors
Field program-mability
'57
'67
'77
'87
'97
'07
Custom LSIs for TVs, calculators
ASICs
Standardized in manufacturing but customized in application
Customization
SOURCE: ELECTRONICS WEEKLY, JAN. 1991

Figure **3-1**: Makitomo's Wave [**71**]

Currently one lives in the outskirt of standardization of field programmable devices (or systems) plunging into customization era. It means that one does not expect to see many new inventions. Though, there will be plenty of differentiations attempted by providing added values from the existing devices and architectures [**73**]. Similarly Keutzer in [**74**] portraits another historical perspective on design evolution to exclaim that there is a need for a new approach as shown in Figure **3-2** .

Electronic device/system designers periodically change their tools throughout history and migrate to a new tool sets that offer a higher level design abstraction. As the s-curve [**74**] indicates that there are leaps of improvements (in Y-axis) happening as the efforts of R&D in EDA tools (in X-axis) continue. One can also observe that there are design level changes occurring where the leaps of design productivity are achieved.

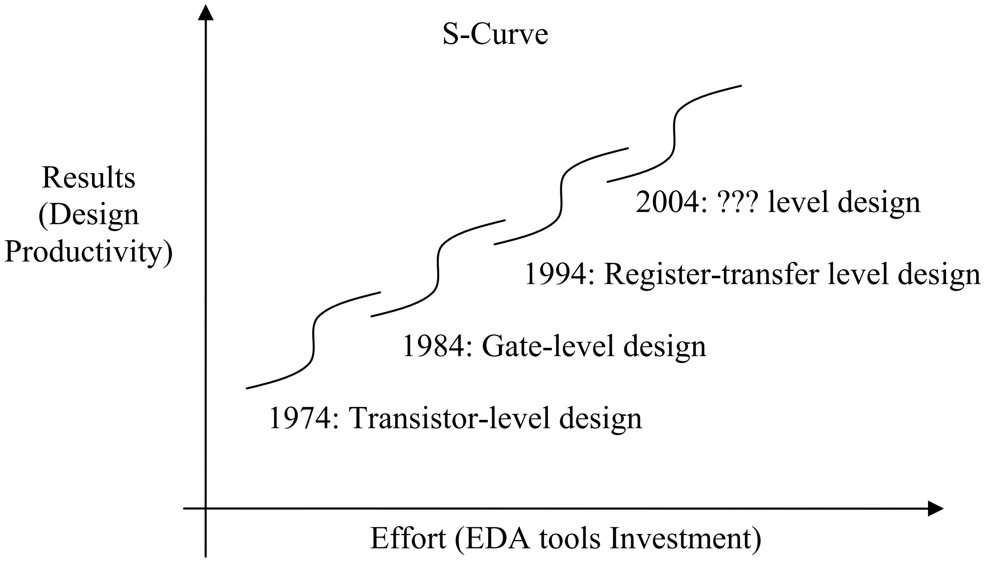

S-Curve
Results (Design Productivity)
2004: ??? level design
1994: Register-transfer level design
1984: Gate-level design
1974: Transistor-level design
Effort (EDA tools Investment)

Figure **3-2**: Design discontinuities in EDA [**74**]

The first design revolution was initiated by Spice circuit simulation developed in 1973. The transistor-level design treated as an artistic skill was embraced by systematic procedures of Spice circuit simulation. When the density of integrated circuit increases, the productivity of transistor-level design lingered at the same level. The second leap was started by assembly of libraries that consist of standard-cells that are verified. Even though they are considered to be area inefficient, the charm of modularity and improvement of productivity persuaded the most of designers to move their design practices. The library components are portrayed as gate-level symbols in schematics, then placed and routed by automatic design tools. By 1990's the schematic based gate-level design became saturated because the number of modules (e.g., >10,000 components) increased up to the level that is extremely time-consuming. Led by Hardware Description Language (HDL) the designs were able to be expressed by register-transfer level (RTL) where logical transfer functions between registers are automatically generated. The combined capability of HDL based entry and automated logic synthesis could produce the gate-level netlists.

By 2000's one should expect:

- customary field programmable system tuned to a specific application ;
- using high-level design abstraction (*system*-level),

according to Makimoto and Keutzer.

3.2.2 Changing economics

The success of the semiconductor industry in the last 30 years can be measured by continuous innovation in technology and increasing R&D investments in design and manufacturing to keep up with the growing demand for semiconductor devices. Out of many predictions, Moore's law has been a main observation that accurately predicts the future of the semiconductor market and a key driver of how the semiconductor industry should respond for market demands. So far the expectation from Moore's law has always been satisfied by implementing higher number of transistors (i.e., Tx/cm2) through smaller geometric processes. Moore predicted that Tx/cm2 would double every 1.5 to 2 years. In addition to Moore's Law, Cost-Per-Function (i.e., microcents per bit or transistor) is a good measure of competitiveness.

Figure **3-3** illustrates the anticipated view of these trends.

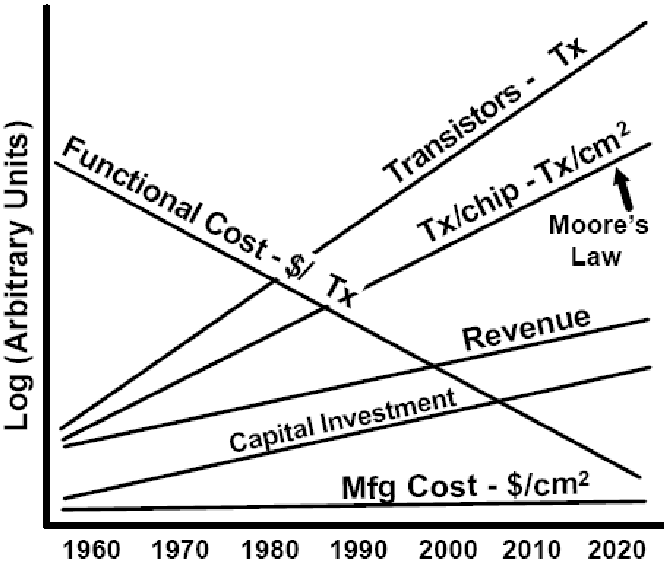


Figure **3-3**: Functional form of key semiconductor industry business trends (Tx=transistor) [**75**]

History shows that if the functionality doubles every 1.5 years then the reduction requirement (i.e., -29%) for Cost-Per-Function can be sustained by doubling Cost-Per-Chip – packaged unit, every six years [**75**]

However, if functionality doubles every three years then the manufacturing cost per chip must remain flat according to the International Technology Roadmap for Semiconductors (ITRS) 2003 [**76**]. These trends show how important improving the cost-effectiveness is for the products to be competitive. In other words, there is a greater chance that the *increasing manufacturing cost of DSM devices might outrun the increase in the number of transistors* under smaller geometry process (< 22nm).

Combining lengthening R&D periods and shorter product life cycle – illustrated in Figure **3-5** , the cost of manufacturing (e.g., new lithography tools, unreliable mask accuracy, multiple emerging technology, etc.) is more likely to increase radically in the near future.

When there is no technological innovation to pull the expectation of Moore's law through, there must be an architectural evolution and/or methodological advancement to push the semiconductor market forward by adding features to meet the anticipated growth.

Today's changing market shows a few distinctive characteristics:

- Shortened time-to-market and critical time-in-market;
- Continuous demand for decrease of functional-cost-per-transistor;
- Explosion of stream application (e.g., multi-media, communication, digital processing, etc.).

Facing these rather demanding trends asks us to look at unconventional solutions. One looks at field reconfiguration as a new direction to expand the usage of hardware by exploiting the spatial and temporal redundancies of running applications. The solution becomes very cost effective. One will compare the costs associated with an ASIC and a FPGA platform to analyze what components are most crucial to make the system more cost effective.

3.2.3 Changing Market

In today's electronic market one can observe prospering consumer devices that are an ensemble of many features. Even though current devices do not reach the level of services that application specific devices can provide, the integration of many complex digital functions led by consumer devices demonstrates the efforts of adding more values for customization. Consumers endeavour to seek more and better services, triggers rapid changes in standards, protocols and I/O technology, which in turn causes the industry to reflect the changes that are consumer's demands. In a worse situation crucial standard deployed in a device is modified to include more features and more values. The device manufacturer who risked capturing early market share would suffer from the losses which include all reoccurring cost of designing the device. The aforementioned case can be empirically illustrated from the difference between Figure **3-4** and Figure **3-5**.

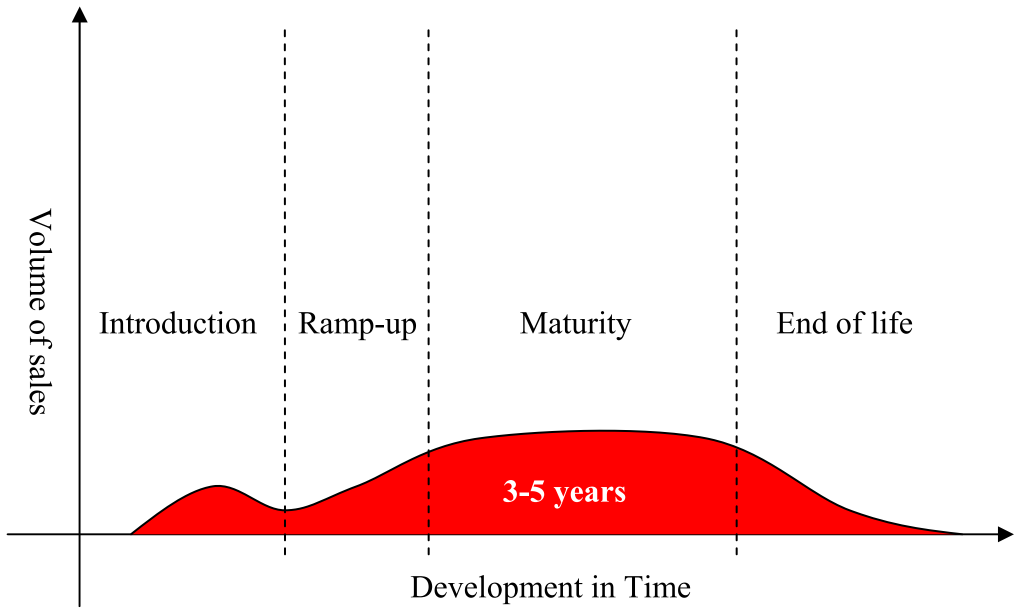


Figure **3-4**: Conventional Product Life Cycle [**72**]

Figure **3-4** from [**72**] illustrates a ballpark figure of conventional product life cycle from introduction to the market to end of device life. The volume which reflects the cost recovery shows steady changes compared with Figure **3-5**.

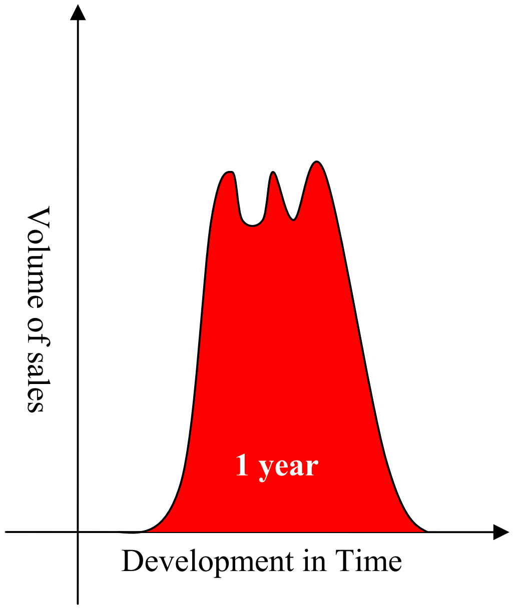


Figure **3-5**: New Product Life Cycle [**72**]

Figure **3-5** illustrates new product life cycle. It has shorter-time-to-market decreasing from 3-5 years to 1 year, higher volume for shorter period of time and dramatic end of life [**72**]. These are new changes occurring in a new semiconductor market.

Rapid rise in manufacturing costs with smaller fabrication technology has become an issue at manufacturing plants. To overcome rising costs at manufacturing level associated with shorter-time-to-market, the field programmability becomes an asset because the functionality of the device can be determined after the device is produced.

3.2.4 FPGA: the alchemist of performance and flexibility

Someone might argue, why not to use stored program architecture (SPA) if the field programmability is an issue to meet performance requirements of the applications? Since many of today's applications rely on stream processing (e.g., audio, video standards, communication protocols, digital processing, etc.), it becomes very difficult to obtain the desired performance without exploiting inherited parallelism of the algorithms in the applications.

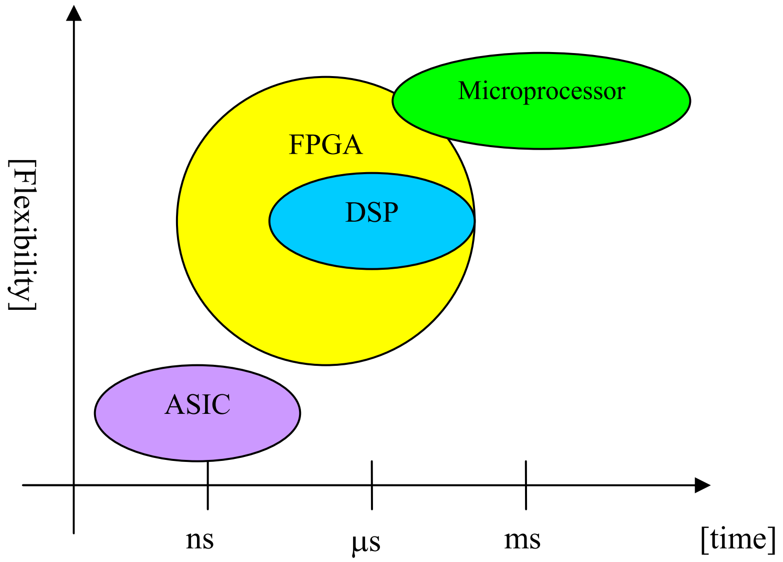


Figure **3-6**: Flexibility vs. Operating time window

An SPA shown as Microprocessor in Figure **3-6** is very flexible and cheap in the performance per dollar ratio [**77**]. However many of them are not powerful enough to process stream applications due to their instruction-based latency. On the contrary, the application specific system with a dedicated hardware solution (i.e., ASIC) can be 10 to 20 times more efficient in silicon area and in speed than a programmable solution. There might also be a need to explore the efficiency of architectures as a function of the application set and implementation models (i.e., Digital Signal Processor – DSP and Application Specific Instruction Processor – ASIP). Among available choices given in Figure **3-6** FPGA might be suitable for that changing market that demands a new design methodology to efficiently reflect the integration of multiple features and to easily meet the performance of embedded applications.

In summary, the previous sections explore various motives that demand a new system design approach for field programmable devices. Makitomo's wave illustrates the customization of field programmable technology that would be specialized in various applications. Keutezs's s-curve points out the necessity of a new design methodology that uses higher design abstraction. Furthermore changing economics is pushing computing systems to be more cost-effective by decreasing development costs and changing market is forcing the systems to be more flexible to adapt for shorter time-to-market trends.

In section **3.3**, the formal quantitative analysis should be conducted to find out what choice is suitable for surfing through the second tide of Makimoto's wave especially in terms of cost-effectiveness.

3.3 Total system Costs

The cost-effectiveness according to the Eq. **3-1** increases by how many services the system does provide, $S_{workload}$ and decreases by the total system cost, C_{total}. The total system cost can have many faces depending on the view of analysis. One facilitates two views in our analysis.

The first view is called unilateral view that looks at overall cost of developing a batch of the same systems before going to market. The second view is called collateral view that looks at the temporal relationship of expenditure and revenue according to the anticipated market trends.

The analysis might be the simplification of real cases overlooking some impacts and issues. However, it would provide the reasons behind the choices that are made in this dissertation.

The following sections clearly define the individual components that are needed to obtain the total system costs in both views. In order to effectively compare, the costs of commercially available reconfigurable device (i.e., FPGA) and a typical ASIC design are analyzed.

3.4 Unilateral System Costs

With the unilateral view one assumes Eq. **3-2**

$$C_{total} = C_{development} + \alpha C_{unit} \qquad \textbf{3-2}$$

where α is the number of units projected to sell, C_{unit} is the unit cost of the system and $C_{development}$ is the development cost of the system. The development cost involves many different aspects of designs.

3.4.1 Personnel Costs

The personnel costs, $C_{personnel}$, depends on the size of workload to be designed. If one assumes that a full device is utilized, the number of gates given for a device is counted as a size of the workload. For FPGA designs, the above assumption is applied to find out what size of a device needs to be purchased while for ASIC designs, the size of workload determines the size of ASIC chips. There are a few other assumptions to be made. The first of them is that the workload can be directly translated into the number of engineers (or the number of months) needed. Since the design engineer is tied with their ability to design and is paid in accordance with their ability, one assumes that the average engineer is capable of developing a certain amount of Register and Transfer Level (RTL) design with a particular salary.

The second assumption is that the differences in resource requirements for ASIC and FPGA designs. To materialize the differences, the personnel costs, $C_{personnel}$ is divided into $C_{development}$ and $C_{verification}$.

The personnel costs, $C_{personnel}$ can be expressed as Eq. **3-3**

$$C_{personnel} = C_{development} + C_{verification} + C_{fix} \tag{3-3}$$

where $C_{fix} = time\_spend \times \wp$ which is the bug fix and code verification process that happens throughout the life time of the design, $time\_spend$ that is assumed to be long in ASIC design (e.g., 16 months) and short in FPGA design (e.g., 4 months).

The development and verification cost, $C_{devleopment\,(or\ verification)}$ can be expressed as Eq. **3-4**.

$$C_{devleopment\,(or\ verification)} = \frac{\partial \times \wp(1+\gamma)}{\nu_{dev\,(or\ ver)}} \qquad \textbf{3-4}$$

where ∂ is the size of workload [gate], ν is the required human resources that it takes to develop logic gates in ASIC (or CLBs in FPGA) [month/gate], $\wp$ is the salary of the average engineer [\$/month] and γ is the average overhead of an engineer [%].

3.4.2 Supply Costs

The considerable amount of development cost is denoted as the supply costs. The supply costs comprise of the hardware and software costs as shown Eq. **3-5**.

$$C_{subply} = C_{software} + C_{hardware} \qquad \textbf{3-5}$$

where $C_{hardware}$ only occurs in ASIC design where the services consist of mask set & prototype wafers, respins, hardware Simulation tools, support and services, etc.

The supply costs are mainly associated with software tool costs. Each software tool is evaluated based on its list price and yearly maintenance fee. Depending on the usage of software, one assumes that the software tools can be shared among users resulting, *cost per seat*. There is also software that needs to be purchased and is specific to hardware (or technology) that the design needs to deal with. In this case the software tool is purchased and used without being shared resulting *program cost*. The life time for supplies including software and workstation is projected to be 3 years (or 36 months).

3.4.3 Software Tools Costs

According to the above facts, the software tool costs are calculated as the summation of tool costs as shown Eq. **3-6**.

$$C_{software} = \sum_{i=1}^{M}(k_i T_i^{floating} + P_i^{floating}) + \sum_{k=1}^{N}(T_k^{fixed} + P_k^{fixed}) + C_{workstation} \qquad \textbf{3-6}$$

where k_i is the Per Seat Usage (0<k_i<1), $T_i^{floating}$ is the price of the ith software. P_i is the yearly maintenance cost that is involved with the ith software. The second summation adds node locked software costs. $C_{workstation}$ represents the workstations (and servers) that are necessary to execute the software. To result in the monthly cost of software tools, the Eq. **3-6** is divided with their life expectancy of 36 months. Where ASIC and FPGA designs differentiates starts by looking at their design flow.

As the name states ASIC is an application specific device. Each time there is a new application (or change in the application), the device needs to be implemented (or modified), verified and tested. The design flow of ASIC is more complicated compared to FPGA due to the verifications and tests required in the hardware level. Figure **3-7** illustrates the typical steps involved in the design flow of ASIC and FPGA.

Figure **3-7** depicts the steps involved to produce the designs in the FPGA and ASIC. Because hardware verifications and tests are proprietary procedures involved in each application, the cost of developing software in ASIC design tends to be expensive. One considers the following software tools to be included for the ASIC design flow:

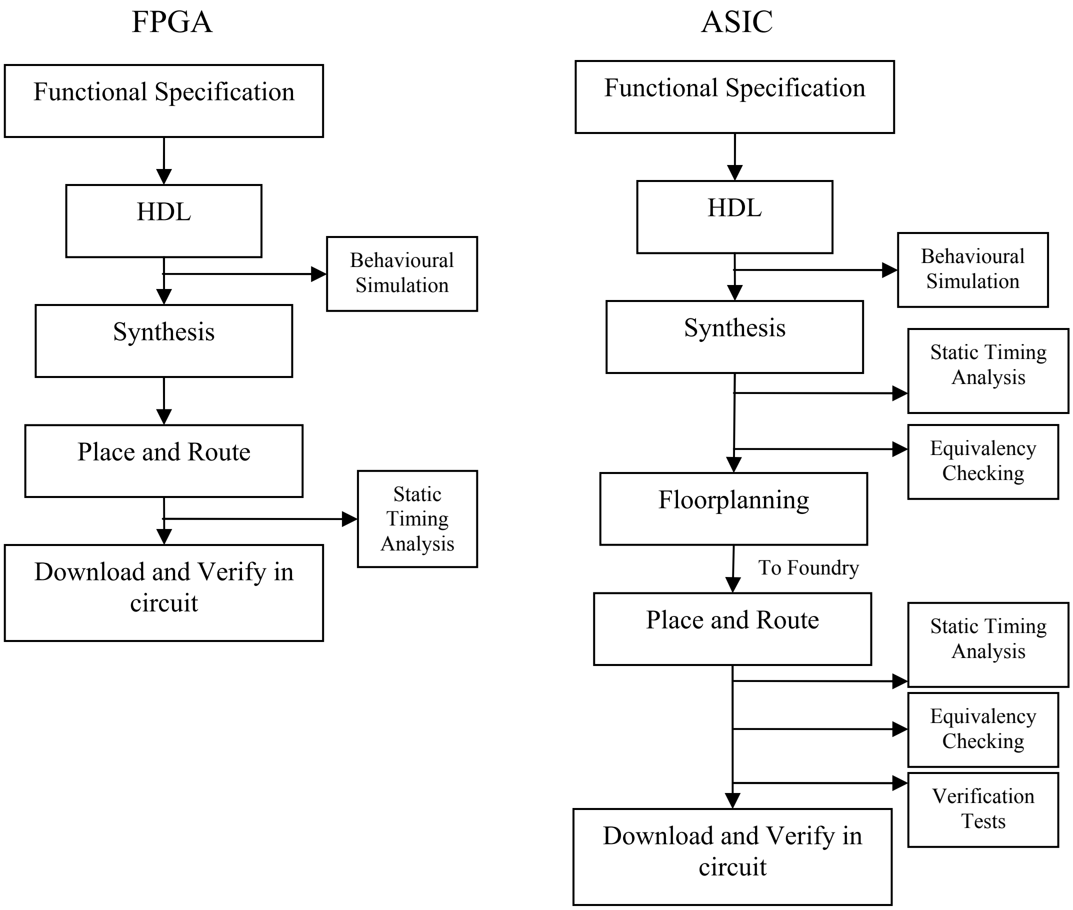


Figure **3-7**: Design Flow FPGA vs. ASIC

- ASIC synthesis software;
- Optimization software;
- ASIC schematic debugging software;
- ASIC Static Timing Analysis;
- ATPG Test Pattern Generator;
- RTL simulator;
- Testbench Automation software;
- Memory Design/simulation software;

- Design maintenance software (e.g., LINT, Code, Coverage, Revision, Control).

The tools[12] are considered to have a floating license. Thus, these software tools are assigned with the street price $T_i^{floating}$ and *per seat usage*, k_i. On the other hand, ATPG Test Pattern Generator, Memory Design/simulation software and Design maintenance software (e.g., LINT, Code, Coverage, Revision, Control) are considered as the node-locked software. Thus they are assigned with the street price of T_i^{fixed}. These attributes provide sufficient information to estimate the overall cost of software tools. Conversely the FPGA design only requires FPGA synthesis software and VHDL simulator that belong to the category of floating software with $T_k^{floating}$. The number of software related with ASIC design tends to be larger than FPGA design, $M_{ASIC} > M_{FPGA}$ and the most of cases, there is no necessity of purchasing a node locked software for FPGA designs, $N_{FPGA} = 0$.

3.4.4 Unit Cost

If the development cost decides what price the device begins with, the unit cost determines how the price of the device increases.

The unit cost of ASIC involves many aspects of technical details and depends on the technology used to fabricate the device. Nonetheless one uses a simplistic view of ASIC fabrication to extract an approximate unit cost. The unit cost of ASIC mainly

[12] ASIC synthesis software, Optimization software, ASIC schematic debugging software, ASIC Static Timing Analysis, RTL simulator and Testbench Automation software

depends on the number of gates to be used, which determines the raw die cost, $C_{raw\,die}$ in our estimation. One assumes that $C_{raw\,die} = \varphi N_{gate}$ has the linear relationship with the number of gates used. Then there is a package cost, $C_{package}$. The $C_{package} = \lambda N_{pin}$ decides the type of package to be used and the number of pins for the device. The ASIC needs to go through assembly test and final test. The cost of tests is estimated to be $C_{test} = \vartheta N_{pin}$. They incur the yielding rate such as κ_{test} and $\kappa_{assembly}$ respectively. Finally the ASIC manufacturing would expect to have the overhead that might be related with logistics, royalty, etc. Eq. **3-7** represents the unit cost that the ASIC requires

$$C_{unit} = \kappa_{test}\left(\kappa_{assembly}\left(C_{raw\,die} + C_{package}\right) + C_{test}\right) + C_{overhead} \qquad \textbf{3-7}$$

3.5 Unilateral System Cost Graph

Looking back at Eq. **3-2**, $C_{total} = C_{development} + \alpha C_{unit}$, the system cost shows the linear relationship. $C_{development}$ indicates the starting point and C_{unit} implies the slope of the cost line. Figure **3-8** illustrates the above traits.

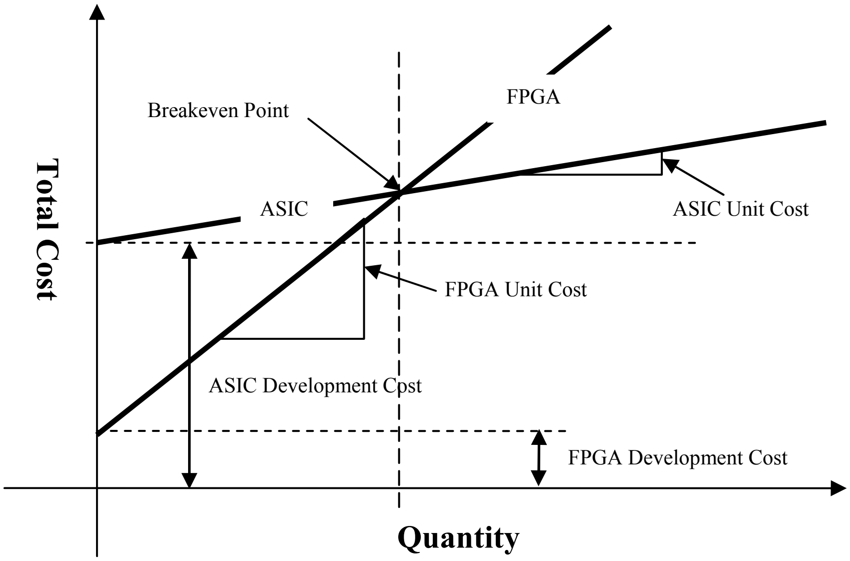


Figure **3-8**: Conventional unilateral System Cost ASIC vs. FPGA

Figure **3-8** informs us that

- ASIC has higher development cost compared to FPGA.

The trait is backed by the various ASIC system costs, $C_{personnel}$ and $C_{software}$ that are larger than ones for FPGA due to longer design time and larger number of software associated the design.

- FPGA has higher unit cost compared to ASIC.

The direct comparison between the street prices of compatible FPGA and ASIC shows a large difference in their unit cost:

- If the anticipated quantity of devices is less than the breakpoint, it is cheaper to implement the design in FPGA;
- If the anticipated quantity of devices is more than the breakpoint, it is cheaper to implement the design using ASIC approach.

This is how a FPGA device is determined to be the choice of the system by analyzing the breakpoint in low-volume applications.

3.5.1 An example: unilateral system costs

The system design environment is assumed to possess the following characteristics:

- Average annual salary of en engineer is \$100,000 and requires at least additional 65% incurred overhead;
- There are 3 man/months required to develop and 6 man/months required to verify 50K gates of ASIC designs[13]. In comparison there are 3 man/months required to develop and 2 man/months required to verify 1.39K CLBs[14] for FPGA designs[15];
- The device targets 0.18μm fabrication technology and requires 960 I/O pins;
- ASIC Software tools are discounted by 45%;

FPGA-equivalent applications need to be implemented in an entire ASIC design that amount up to 5.4K gates.

[13] For ASIC, $v_{dev_{ASIC}} = 3/50K$, $v_{ver_{ASIC}} = 6/50K$

[14] 1.39K Configurable Logic Block (CLB) in FPGA is equivalent with 50K gate in ASIC

[15] For FPGA, , $v_{dev_{FPGA}} = 3/1.39K$, $v_{ver_{FPGA}} = 2/1.39K$

3.5.1.1 Unit Cost

The advantage of ASICs is low unit cost for high performance. Since there is relatively large development costs associated with manufacturing the ASICs, the large volume of sales should be expected to amortize the development costs over the number of chips. On the other hand, the FPGAs save considerably in development costs because they are pre-manufactured programmable chips that can be reconfigured when the unit cost of the FPGAs can be very high compared to the ASIC.

In order to calculate the unit cost, one needs to assume some technical details [**78**]. The real costs of die, packaging, outputs and testing are quoted by UMC in Taiwan[16]. These technical details are exemplified in Table **3-2** for ASIC approach.

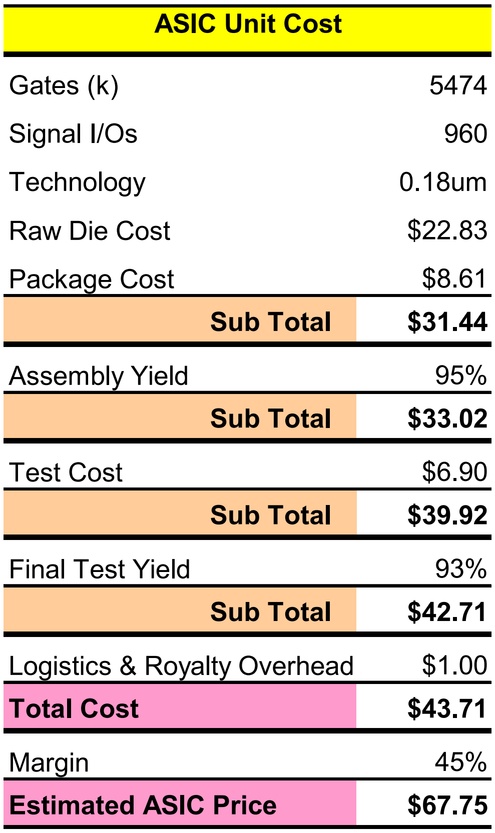
With similar details, the unit cost of FPGA should be very similar to the ASIC counterparts. However, the unit cost of FPGA available from vendors is not what is expected.

For instance to determine the unit cost of ASIC that is similar with the FPGA device, one first looks into the typical metrics used to calculate the gate average of Xilinx FPGA XC4000 series. Then it is used to estimate the gate usage in Virtex-4 devices. To extract the average gate usage of the FPGA device, the number of CLBs (i.e., 152064 for the target device) is applied with the linear relationship found in the Appendix A. Then, the typical gate for XC4VLX160 is calculated to be 5474 [Kgate]. Assuming the linear

[16] The given assumptions and the following table templates are quoted from “FPGA vs. ASIC project cost calculator” which uses the dollar figures provided by the UMC in Taiwan.

die cost increases with the increase of gate numbers, the ASIC that is equivalent to the target device costs $67.75.

Table **3-2**: ASIC Unit Cost with 21888K gate capacity



ASIC Unit Cost	
Gates (k)	5474
Signal I/Os	960
Technology	0.18um
Raw Die Cost	$22.83
Package Cost	$8.61
Sub Total	**$31.44**
Assembly Yield	95%
Sub Total	**$33.02**
Test Cost	$6.90
Sub Total	**$39.92**
Final Test Yield	93%
Sub Total	**$42.71**
Logistics & Royalty Overhead	$1.00
Total Cost	**$43.71**
Margin	45%
Estimated ASIC Price	**$67.75**

NOTE: Packaging costs $0.0075/pin using FF1148 regarded to be the same cost as BGA package. Testing costs $0.06 per 10 I/Os per second.

However the list price of XC4VLX160 from the distributor – Avnet[17] is $3,125 which is the 2.1% of the equivalent ASIC price –98% discount seems impossible to be negotiable. The given ASIC cost resembles the logic capacity of XC4VLX160 with a fraction of the cost, $67.75. If the both systems carry exactly the same functionality, there

[17] https://emwcs.avnet.com/webapp/wcs/stores/servlet/RemoteAdvancedSearchView?langId=-1&storeId=500201&catalogId=500201&manufacturerPartNum=XC4VLX160-10FF1148C

is no argument that everybody who needs the system will purchase the ASIC instead of the FPGA. However, there are other factors that may make one reconsider the choice.

3.5.1.2 Software Tools Costs

First is the developing cost. As it was mentioned before, the ASIC is an application specific device. Each time there is a new application (or change in the application) the device needs to be implemented (or modified), verified and tested. The design flow of ASIC is more complicated compared to FPGA due to the verifications and tests required in the hardware level. The calculation of software tool cost includes the price of each tool and per seat usage taken from [**78**]. The overall software tools cost for the ASIC design is $299,218 when the cost for the FPGA design is $53,450. The software tools costs are converted in terms of expense/man month (i.e., $1,485 for FPGA and $5829 for ASIC) to calculate the expense(s) applied in each period of the development stage.

3.5.1.3 Development Costs

The development costs are estimated mainly with human resources involved in designing the system in association with implementing the target application (i.e., the number of gates). Table **3-3** shows the summary of these estimations.

Table **3-3**: Development Cost

	ASIC		FPGA	
	personal cost	supply cost	personal cost	supply cost
RTL Development	$4,516,050	$1,914,477	$4,516,050	$487,642
RTL Verification	$9,032,100	$3,828,954	$3,010,700	$325,095
Mask Set and Prototype Wafers		$250,000		$0
Re-Spins, Mask Sets and Wafers		$250,000		$0
Hardware Simulation Tools		$88,200		$0
ASIC Support and Services		$150,000		$0
Bug Fix and Code Verification	$220,000		$55,000	
Subtotal	**$13,768,150**	**$6,481,630**	**$7,581,750**	**$812,737**
Total Development Cost	**$20,249,780**		**$8,394,487**	

First of all, there are several aspects of design: RTL development, RTL verification, Bug fix and code verification, mask and wafer costs. For human resource aspect, an individual engineer salary is settled with nominal $100,000 considering the average overhead – 65%, associated with the ASIC engineer. One engineer for the ASIC design would cost $13,750 per month. The other assumption is how many gates an ASIC engineer can design per month. One puts a rough approximation of 50 K gates per month that is capable of being designed or verified by an ASIC engineer. When these assumptions are combined with the software tool costs previously calculated, the developing cost can be estimated for a particular device (e.g., XC4VLX160). For hardware manufacturing aspect, there is an initial manufacturing cost and an additional cost for re-spin(s). When all aspects are amalgamated for the estimation of the development costs, one can tell why the ASIC is not the only choice despite the difference in the unit cost. The ASIC design for the targeted application would cost $20.25 millions while it costs $8.4 millions for the FPGA design. Figure **3-9** shows the breakeven points where the number of devices equalizes the development costs between ASIC and FPGA using Eq. **3-8** .

$$Number\_of\_devices = \frac{ASIC\_development\_\cos ts - FPGA\_development\_\cos ts}{FPGA\_discounted\_\cos t - ASIC\_unit\_\cos t} \qquad \textbf{3-8}$$

The list price of the target FPGA is used assuming that there is discounted price for a higher volume.

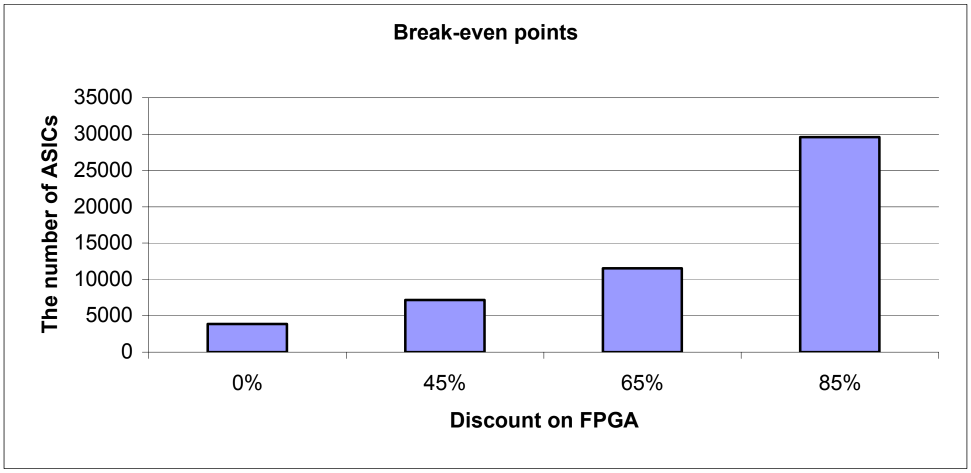


Figure **3-9**: Breakeven points – the number of ASICs vs. FPGA unit cost in volume

With further discounted price, 85% for FPGA, the ASIC should sell at least 527,308 pieces to amount equally with the development costs of FPGAs. The trend seems to show an exponential growth of required sales to amortize the ASIC development costs compared with the FPGAs. Hence, if the application is targeted for a specific market region (e.g., < 20,000 sales with < $1000 FPGA price and XC4VLX160 complexity), the FPGA system can offer the range of competitive price compared with the ASIC – when both the technology and the application are mature. Figure **3-10** illustrates and summarizes the findings of the previous sections.

Since the market prices are normally historically/politically/regionally settled upon various factors: marketing such as name recognition, reliability, customer service etc., the findings of this section are very arbitrary. However, they are intended to emphasize the obvious differences between the cost of FPGA and ASIC designs.

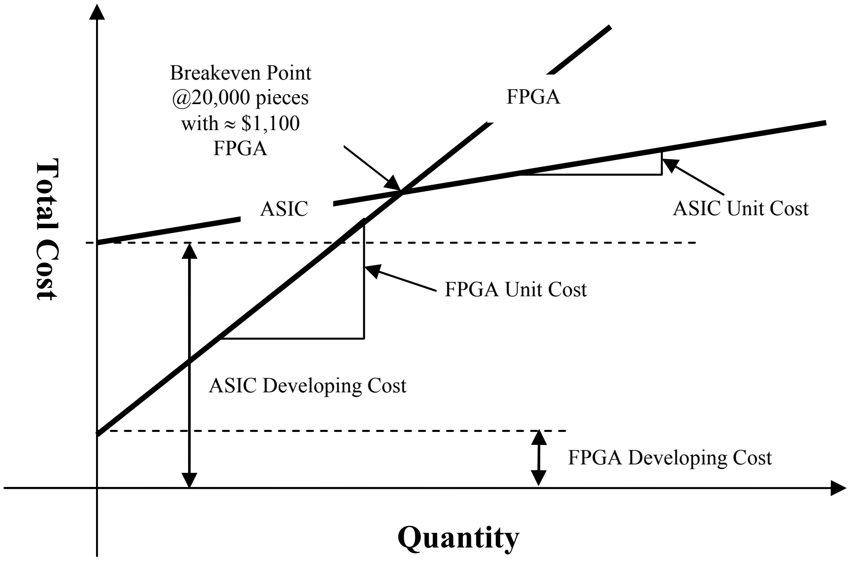


Figure **3-10**: Cost vs. Quantity

More importantly when these estimations are combined with new market trends:

Trend I: Shortened time-to-market and critical time-in-market;

Trend II: Integration of multiple functionalities,

the cost of the FPGA system becomes more effective. One will analyze how the above changes can affect the total system cost and cost-effectiveness of the system collaterally.

3.6 Collateral System Costs: Trend I

Shortening time-to-market radically changes the business model of integrated devices because their profits can only be harvested in a very short period of time with higher risk. The traditional ASIC design methodology is incapable of handling short time-to-market due to its long design, verification and manufacturing lead time. Conversely, an FPGA system can easily adapt its business model upon the short life time of product life cycle. Since manufacturing FPGA hardware is independent of design and implementation process, there is no lead time spent on hardware manufacturing or equivalence verification. Then, how much benefit (i.e., reduction in system costs) does a new market trend bring into the business model? To outlay the benefits accounted by introducing Trend I, one calculates the total system costs differently such that the total system cost is expressed as the cumulative revenue-expenditure (RE) of the life time of the product, $RE_{total} = C_{total}$. Since the success of electronic devices based business model is evaluated depending on the surplus of revenue minus expenditure for a short duration of time, the total system cost should be also regarded as a continuous act of balance for the duration of product life time. The total system cost (RE_{total}) is expressed in Eq. **3-9** accordingly.

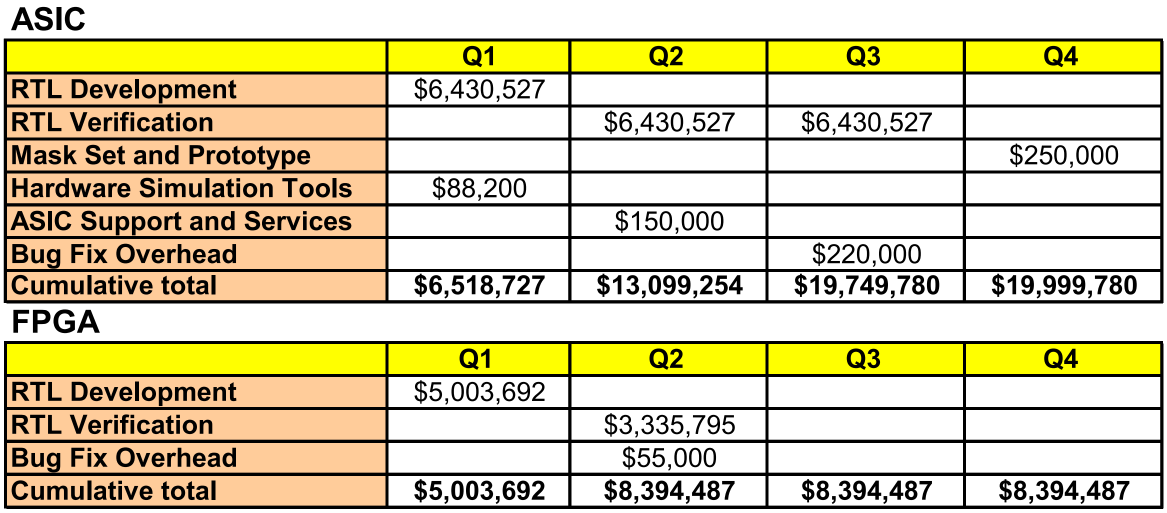
$$RE_{total} = \sum_{time} \left(\alpha C_{unit_{time}} - C_{development_{time}} \right) \qquad \textbf{3-9}$$

where $\alpha C_{unit_{time}}$ is the revenue that is the number of unit sales, α with the price of each unit, C_{unit} at $time$ and $C_{development_{time}}$ is the expenditure, $C_{development}$ at $time$.

To analyze the RE_{total} of ASIC and FPGA approaches quantitatively, one applies the conventional and new product life cycle into our cost projection showing, $\sum_{time} C_{development_{time}}$.

Table **3-4** shows an example expenditure pattern associated with development costs of the ASIC and FPGA design[18].

Table **3-4**: Development Cost Projection ASIC vs. FPGA



ASIC

	Q1	Q2	Q3	Q4
RTL Development	$6,430,527			
RTL Verification		$6,430,527	$6,430,527	
Mask Set and Prototype				$250,000
Hardware Simulation Tools	$88,200			
ASIC Support and Services		$150,000		
Bug Fix Overhead			$220,000	
Cumulative total	**$6,518,727**	**$13,099,254**	**$19,749,780**	**$19,999,780**

FPGA

	Q1	Q2	Q3	Q4
RTL Development	$5,003,692			
RTL Verification		$3,335,795		
Bug Fix Overhead		$55,000		
Cumulative total	**$5,003,692**	**$8,394,487**	**$8,394,487**	**$8,394,487**

The costs of system are logically laid out such as development, verification and prototyping costs. However due to the differences in ASIC and FPGA approaches, not only is the amount of development costs much less in FPGA, but also the time invested in designing. In this example 4 quarters for ASIC and 2 quarters for FPGA. Figure **3-11** shows the estimated volume projection over 3 years based on the conventional product life cycle that enables us to calculate $\sum_{time} \alpha C_{unit_{time}}$.

[18] These calculations are based on the results obtained from the example of unilateral system cost in the section **3.5.1.**

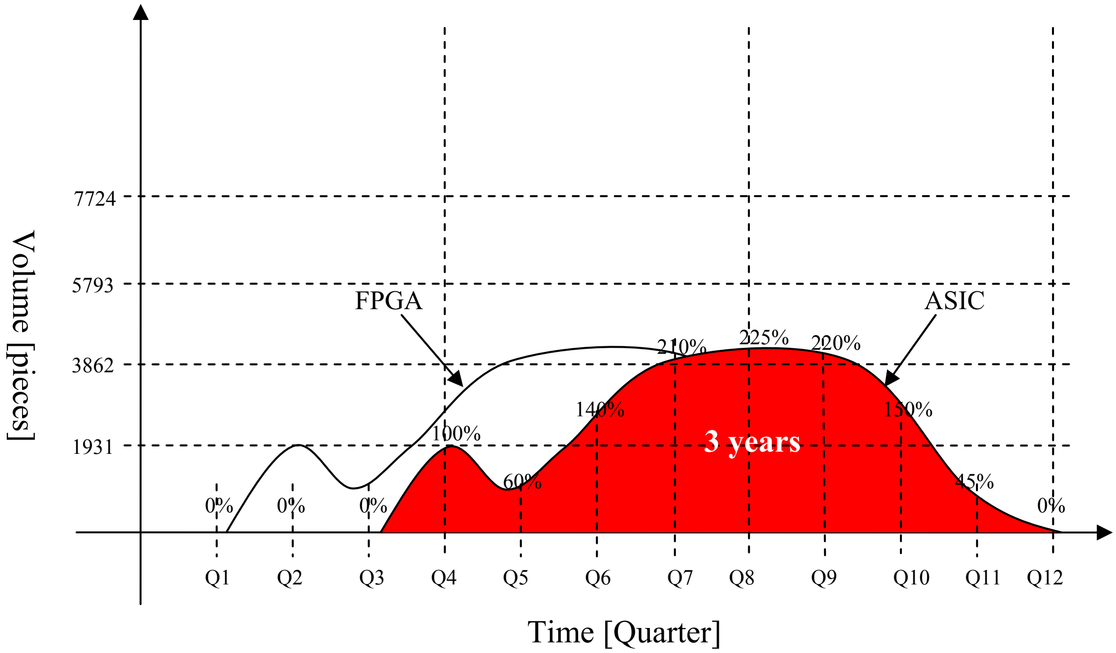


Figure **3-11**: Sale volume estimation (conventional product life cycle)

The volume of the devices[19] is quoted from the breakeven point in Figure **3-10**. If one assumes that the volume sale at the introductory phase is about 1000 pieces which assumes to be 100% at the introductory stage because of early lead time, the ASIC sits comfortably with the conventional product life cycle, the ASIC device sale should be able to recover the costs and the profit margin within 3 years. The price of the ASIC would be estimated to be $1,420.32. In order for the FPGA design to be compatible, one assumes the price of the FPGA system also costs the same as the ASIC design. Figure **3-12** shows the gross margins of FPGA and ASIC over 3 years using conventional product life cycle,

$$RE_{total} = \sum_{time}\left(\alpha C_{unit_{time}} - C_{development_{time}}\right).$$

[19] The profit margin of 45% compared to the development costs is assumed.

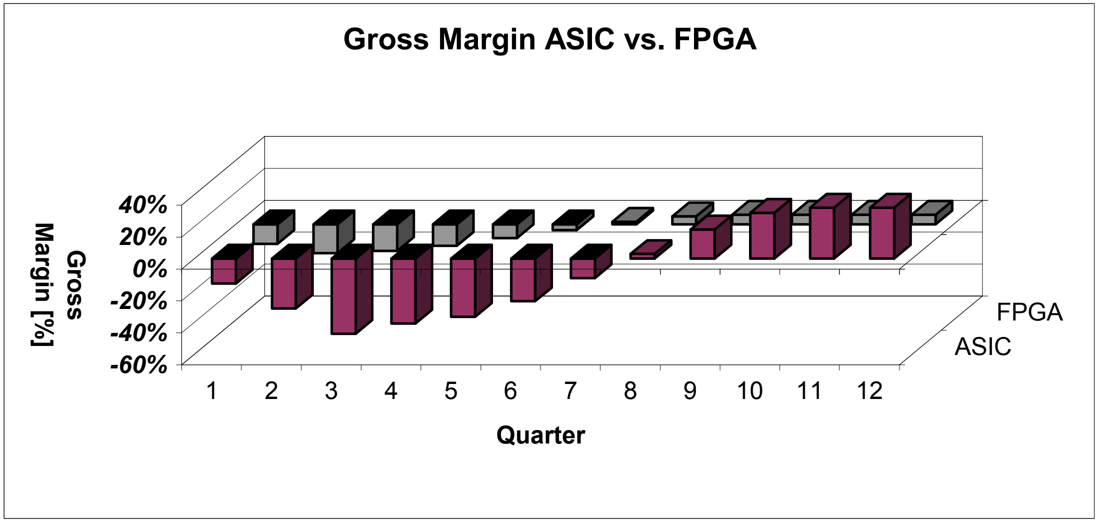


Figure **3-12**: Gross Margin of ASIC vs. FPGA (discount of FPGA, 65%)

The FPGA shows much slower recovery of investments and gradually increasing profits than the ASIC does. Yet there are much more development costs associated with the ASIC. The FPGA starts to gain profits earlier due to short development time (i.e., Quarter 7 instead of Quarter 9). The higher margin of the ASIC due to the low unit cost, allows a faster recovery trend. However, FPGA design does not seem to be cost-effective at the breakeven point of the development costs with the conventional product life cycle. It only achieves 6% margin compared to 32% of the ASIC. The main reason is because of smaller margin caused by high unit cost of the FPGAs. Unless the unit price of the FPGA is reduced to be $718.75, which is 77% discounted price from the list price of XC4VLX160 as shown in Figure **3-13**, the profit of the FPGA will not reach the level of the ASIC. A key to improve the competitiveness of FPGA system is *lowering* unit cost of FPGA.

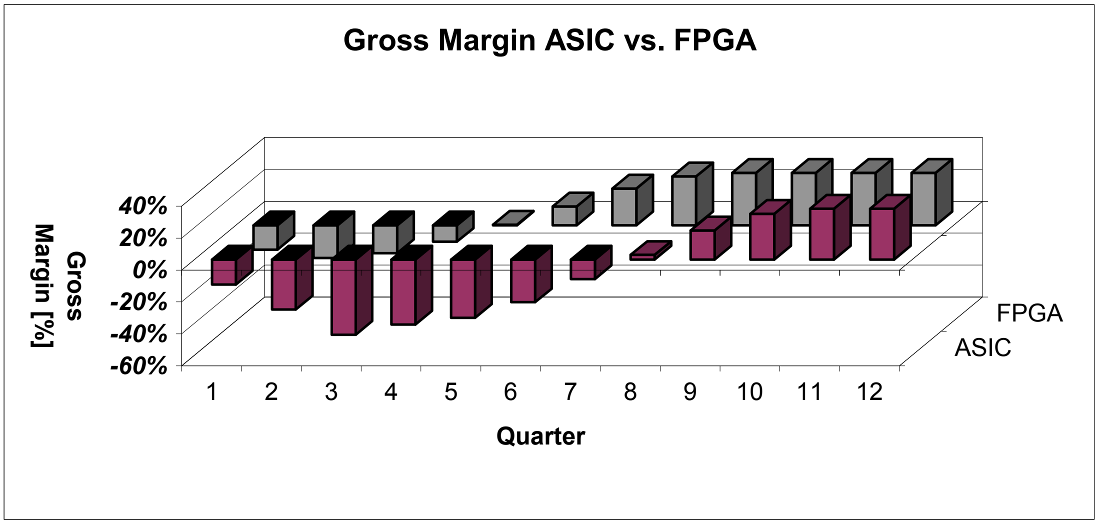


Figure **3-13**: Gross Margin of ASIC vs. FPGA (discount of FPGA, 77%)

So, one assumes that the unit price of FPGA should be about 23% of the listed price to achieve the same gross profits of the ASIC. To explore further collateral impacts of Trend I, one looks at time-to-market and time-in-market issue. Figure **3-14** illustrates the expected volume sale based on one-year time-to-market, $\sum_{time} \alpha C_{unit_{time}}$.

First short time-to-market makes the development of the ASIC very difficult because of its lead time. The only way to offer a continuous flow of revenue is to pipeline a series of products that come out in time before the previous product loses its sale. However, adapting this model, the ASIC design can not be flexible to swift market changes because the next product should be in the development phase before knowing the result of the current sale, thus, predicting the next product. There is no plan time or reaction time.

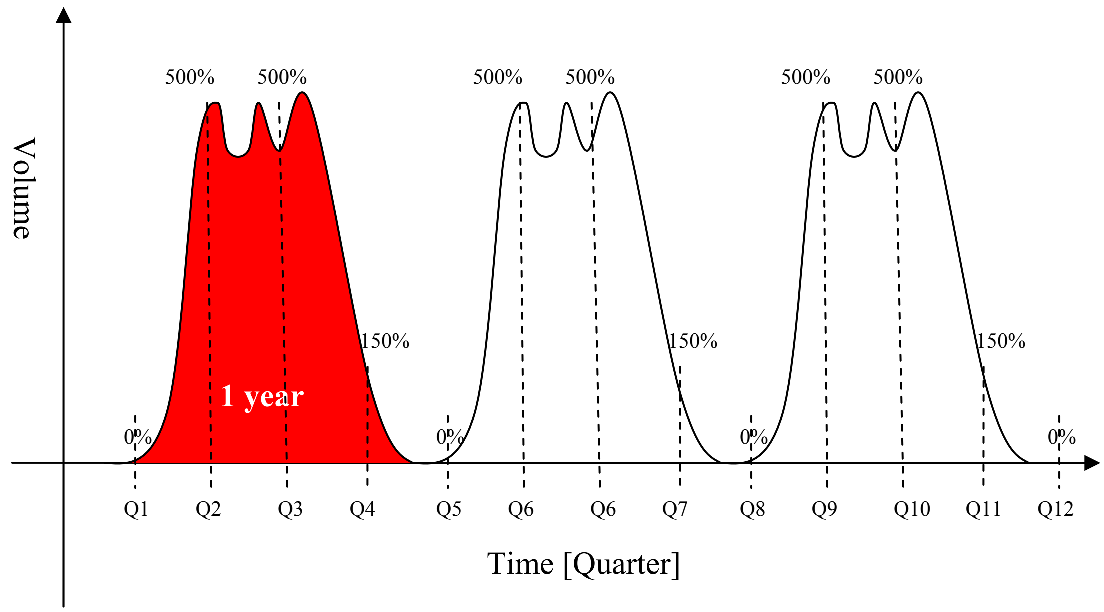


Figure **3-14**: Sale volume estimation (new product life cycle)[20]

While the short time-to-market itself creates more risk than three-year one, the time-in-market generates 200% more risk due to no reaction time for the ASIC design, where all these risk should be considered as a part of business model. With the assumptions built from the previous analysis of costs and revenue, one reconstructed the graph that represents the gross margin, $RE_{total} = \sum_{time} \left(\alpha C_{unit_{time}} - C_{development_{time}} \right)$ with one-year time-to-market in Figure **3-15**.

One first observes that the revenue of the FPGA design is comparably increased while the ASIC suffers with a low margin. Recurring heavy ASIC development costs for continuous capital flow is responsible for a low profit. If one considers the competitive

[20] Total sale volume in a cycle is matched with the one in the conventional product life cycle

ASIC market with risks involved, the FPGA certainly becomes more appealing for the choice of many applications.

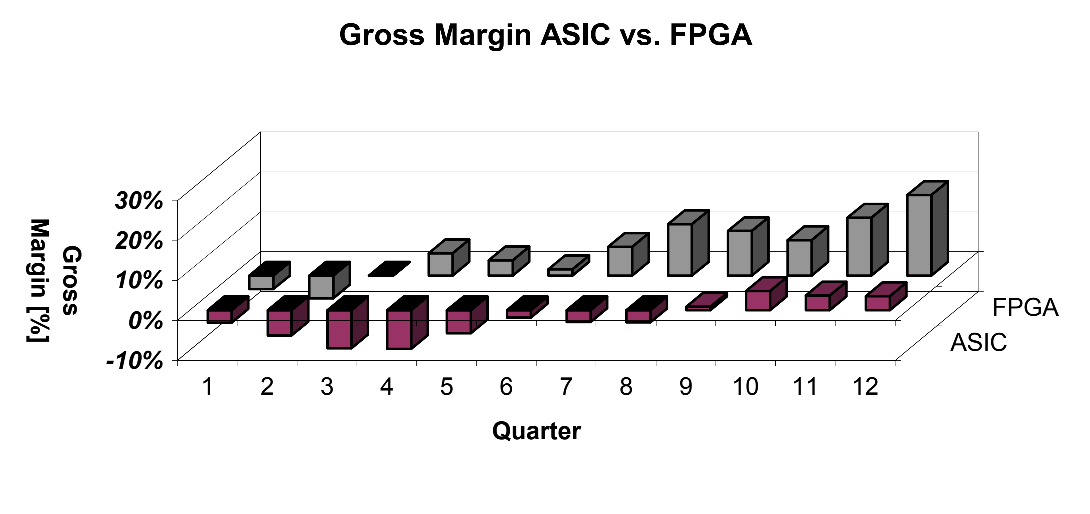


Figure **3-15**: Gross Margin of ASIC vs. FPGA (discount of FPGA, 77%, one-year time-to-market)

Furthermore, the reconfigurability of some FPGAs provides further improvement in the functional cost per transistor because they can be reconfigured to carry different functionalities in temporal domain. Figure **3-16** illustrates the unit price related with the number of logic gates available.

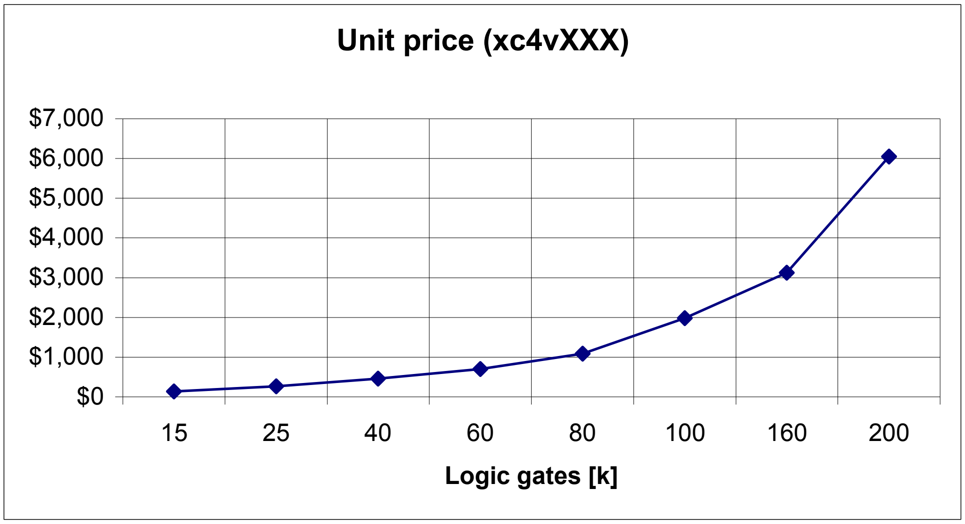


Figure **3-16**: Unit price for Xilinx Virtex-4 FPGA LX family[21]

As one sees in Figure **3-16** the unit price of the FPGA is not linearly related and increases much faster than the number of gates increases. Hence, the functional cost per transistor becomes cheaper as one decreases the number of gates needed for the functionality in time domain.

However, there is service hardware needed and temporal overhead executed if one wants to implement the reconfigurability as to improve the cost of the system. The success of the FPGA system depends on how effectively the system can hide these overheads.

For this purpose the popular stream applications serve as a mechanism to discover regularity of data processing and to exploit the spatial redundancy of the FPGA device where the speedup of execution through architectural manipulation is mandated.

[21] The price of Xilinx FPGA Virtex-4 devices are based on the unit cost from the website of http://www.avnet.com/

3.7 Cost-effectiveness: Trend II

One looked how the trend I: shortened time-to-market and critical time-in-market boosts the competitiveness of FPGA systems in term of its revenue-expenditure balance. One also observed how a higher unit cost of FPGA system influences:

- Unilateral total system cost in Figure **3-10**;
- Collateral total system cost (as gross margin) as the difference between Figure **3-12** and Figure **3-13**.

It is obvious that FPGA devices can not be offered with 77% discount as shown in Figure **3-13**. However, with the trend II: Integration of multiple functionalities, the FPGA devices can become more cost-effective. According to Eq. **3-1**, the cost-effectiveness of the computing system depends on C_{total} as well as $S_{workload}$.

In ASIC approach, increasing $S_{workload}$ normally results in increasing C_{total} due to increase in processing resources. However in FPGA approach increasing $S_{workload}$ does not always need to result in increase in hardware. Because of run-time partial reconfigurability, the $S_{workload}$ can be virtualized (i.e., stored in memory). The virtualization of $S_{workload}$ enables savings of processing resources and encourages the developments of reusable components.

3.7.1 Unilateral Cost-effectiveness

Referring to Figure **3-8** that expresses the breakpoint according to the example given in section **3.5.1**, one can appreciate how much virtualization needs to be employed to be unilaterally meaningful. Figure **3-17** shows the prediction on such assumptions.

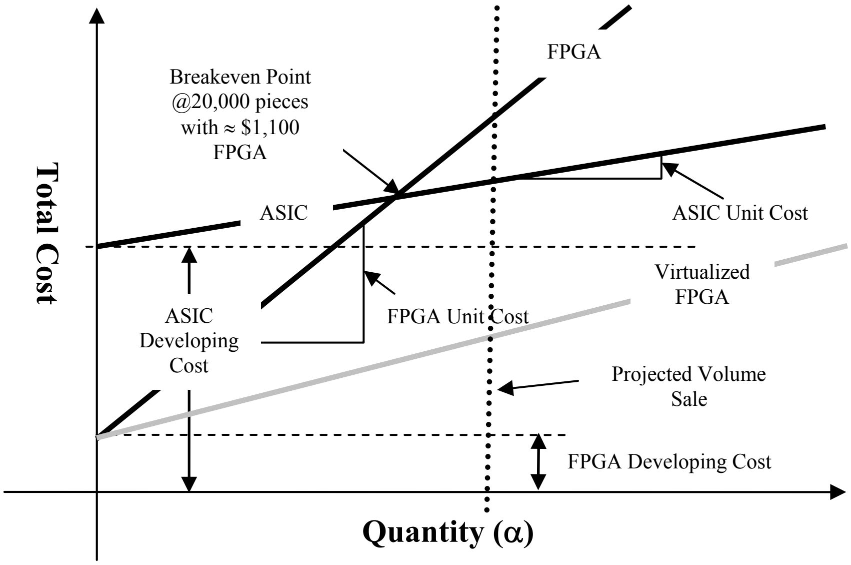


Figure **3-17**: Quantity vs. Costs

The increase of $S_{workload}$ generally causes the costs of computing systems to be increased. However, the unit cost of FPGA system can stay the same by virtualizing processing resources that implement more functionality in the same space. Thus, it appears that the unit cost of FPGA system decreases shown as the grey line in Figure **3-17**.

Each line that represents the total system cost can be expressed with a linear equation $C_{total} = C_{development} + \alpha C_{unit}$ which is equivalent with Eq. **3-1** where C_{total} representing y-axis and α representing x-axis. If the FPGA unit cost, C_{unit}^{FPGA} is the same as the ASIC unit cost, C_{unit}^{ASIC} by virtualization of processing resources, the total system cost of FPGA would never exceed ASICs, C_{unit}^{ASIC} (because the ASIC development cost, $C_{development}^{ASIC}$ is always higher than FPGAs, $C_{development}^{FPGA}$). However, when the FPGA unit cost is larger than the unit cost of ASIC, the number of devices sell in the market -- as represented as a vertical line (i.e., the projected volume sale) in Figure **3-17**, should be taken into the consideration of system cost.

Hence, one can predict the unilateral cost-effectiveness of FPGA system in comparison with ASIC approaches by identifying all – the projected volume sale, unit cost (especially with virtualized processing resources) and development cost of the systems.

3.8 Conclusion

This chapter explores new market trends and various components that affect the cost-effectiveness of computing systems. It first identifies some of historical movements towards reconfigurable computing (i.e., Makimoto's wave) and elaboration of design abstract (Kreutzer's s-curves). Changing economics pushes the systems to be more sensitive to increase of manufacturing costs in integrated circuits. By analyzing the components (e.g., supply costs, software costs, personnel costs, etc.) that are associated with the cost-effectiveness of the system, one could tie the market trends such as shortened time-to-market, critical time-in-market and integration of multiple functionalities into the cost-effectiveness of emerging reconfigurable devices. As one has seen in the qualitative analysis, the market trends demand the reconfigurable devices to be more cost-effective. However to realize the benefits from the unique feature (i.e., dynamic reconfigurability) of reconfigurable devices, *virtualization processing resources* for a multi-task and multi-mode workload should be systematically achieved. The elaboration of design abstract combined with reconfigurable capability discussed in the next chapters should be able to increase the cost-effectiveness of reconfigurable systems as shown in this chapter.

Chapter 4

Exploration of Architectural Spaces for Virtual Hardware Components

To realize our goal in maximizing the cost-effectiveness of a reconfigurable computing system, the system design is divided into three layers where the hardware operating system should be able to assemble the system on a chip out of available components upon requests. Thus, the necessary components should be available where it is readily accessible (i.e., component library).

Synthesizing, placing and routing components are not the tasks of online system as the system becomes too complex and the granularity of component is becoming larger (≈500K logic gates [**13**]). They should be designated to be offline tasks. The benefits of offline tasks are that they can be optimized for many different aspects of designs and can create the components associated strongly (or weakly) to the evaluating parameters. At the same time, the constrained granularity prevents the focus of design shifting from logic to interconnect.

In the process of finding a method to create and optimize the components, one will adapt the method suggested as multi-parametric optimization of the modular computer architecture in [**79**] and apply the constraints that are applicable for the components rather than for the systems. The major benefit of the approach is *the reduction of architectural variants to a large extend*, thus resulting much more compact library establishment. Component library is a key concept to the successful implementation and support of the reconfigurable system activities at all levels of the design process.

4.1 Background

Most of today's complex systems consist of both architectural and behavioural constructs because some parts of the system can not be described by only algorithms. The systems that are increasingly implementing computation intensive operations can only achieve higher performance by the expense of parallelized processing resources.

Upon the arrival of parallel processors and their algorithms, the graphical notations have become popular to express the mathematical operations and their relationships. The sequencing graph (e.g., DFG) is one of abstraction models that represent by procedural HDLs at which graphing can easily articulate required operations and their dependencies.

While the precedence relationships among the individual operations reveal the ultimate performance of the algorithm, the combinations of resource restrictions, performance requirements or I/O specifications can demand to explore different implementations. There are many factors that determine the performance of implementations. Selecting the right algorithm is the foremost efficient way to save resource and increase performance. However, our focus is neither the invention of a new algorithm nor the improvement of existing algorithms. Other factors of performance are mainly concerned with:

- Allocation of operations, storage and interconnects;

- Scheduling of operations;
- Binding.

In other words, the steps involved in behavioural synthesis are the factors that decide the performance of implementation. However these factors need to be synthesized within specified constraints that are given as the parameters for components.

In order to use the architecture synthesis method as multi-parametric optimization of the modular components [**79**], one needs to take a few assumptions. First, one is given with a sequencing graph assembled by the vertices and their dependencies. The vertices are specified with what it does but not limited by how it is implemented. Because of fixed I/O specifications given for the component, the interfaces of resources are assumed to remain static. These assumptions allow various implementations of a node possible, which leads to different scheduling/binding result.

Figure **4-1** shows a typical example of sequencing graph $G_s(V, E)$ that is a hierarchy of directed graphs[22].

The directed graph inherently uses the variables, whose values save the data required and created by the operations. The interval between a variable's birth and death is referred as its lifetime. The birth is the time at which the value is generated by an operation and the death is the latest time at which the variable is consumed as an input of the operation(s). One intends to use the directed graphs such as sequencing graphs for synchronous systems where the storage of variables and its lifetime are well coupled with underlying hardware components.

[22] The facts and terms regarding the sequencing graphs are referred from [**80**]

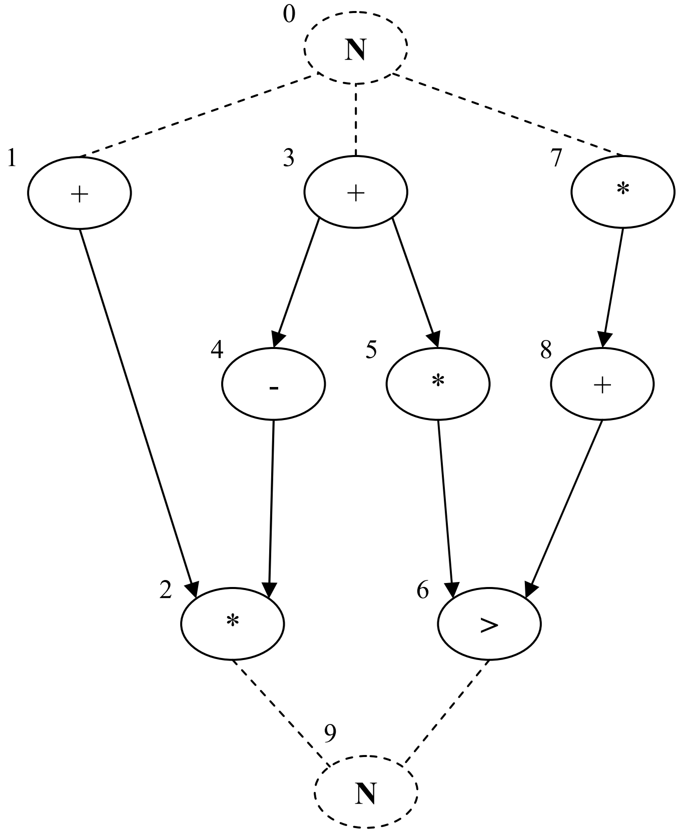


Figure **4-1**: Example of a sequencing Graph

A sequencing graph has source and sink vertices labeled as v_0 and v_n respectively. Hence, the sequencing graph that has the *n* number of operations, n_{ops} would have the n_{ops} + 2 vertices. The vertex set V={v_i; $i = 0, 1, \dots , n$} and the edge set E={(v_i, v_j); $i, j = 0, 1, \dots , n$} where e indicates v_i generates the value of a variable for v_j.

To facilitate the multi-parametric optimization of the modular component architecture the designer explores various implementations of V to achieve its functionality. With fixed E, each variant of resources become an architectural variable to consider.

4.2 Architecture Configuration Graph

Creating components is the optimization process that requires contributions from available resources and feedback from physical constraints. When a component is described by a sequencing graph, all variants of resources can be fully evaluated under the given parameters. A set of constraints such as I/O bandwidth and data type must also be provided. If one assumes that there are $\left\{R_i \mid i=1,2,...,n\right\}$ resources where *n* is the number of types of different resources and each resource has $\left\{R_{i,j} \mid i=1,2,...,n \textit{ and } j=1,2,...,m_i\right\}$ variants where m_i is the number of possible variants that are associated with resources.

If the components are built by assembling system resources, the variances that are available for each resource can make the optimization process quite complex. Figure **4-2** shows an example of the design exploration using these variances.

As it is shown in Figure **4-2** the architectures of the component that requires R_l, R_j and R_k resources can be represented in the form of a tree using $G=\left(V,E\right)$ where the vertices V_i associates with system resources and the edges E_{ij} assigns a variant R_j to the vertex R_i. How complex the tree should be mainly depends on the number of necessary resources to carry a required functionality.

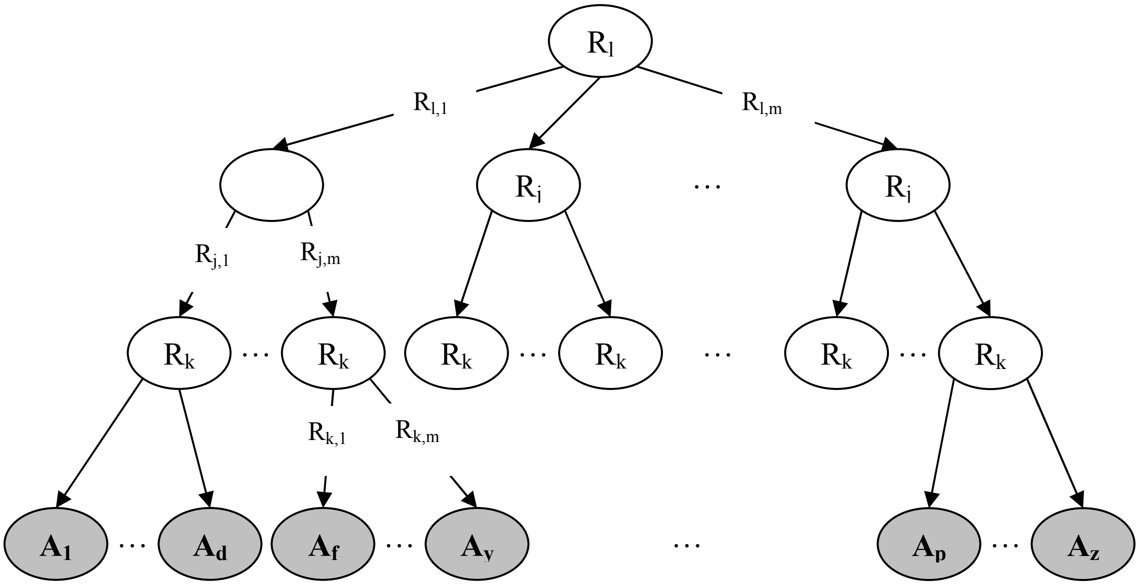


Figure **4-2**: Example of component design space exploration represented by a tree

To obtain a tree graph such as Figure **4-2** one assumes that the followings are given:

- a set of fully characterized resources, $\left\{R_{i,j} \mid i = 1,2,...,n \textit{ and } j = 1,2,...,m_i\right\}$;
- a sequencing graph of the function where it indicates the required resources and the dependencies of the data via resources;
- a set of constraints which limits the ideal implementation of the functionality.

A set of fully characterized resources are classified as *functional, memory* and *interface* resources [**80**]. Due to the complications that rise using wire models in high speed computation the representation of interface resources is becoming a larger problem (e.g., DSM effects [**13**]). Hence one limits the granularity of architecture to stay in the component level where the intercommunication is established within the local routings –

wire delays stay below the gate delays. When the DSM effects are not a part of design concern, the architecture configuration graph (ACG) such as in Figure **4-2** can be accomplished without much difficulty.

Finding an optimal architecture on the ACG is a very repetitive process that demands a large amount of computations for each constraint parameter considered. Especially when there exists a multi-parametric set of constraints, a complex multi-dimensional design evaluation becomes necessary. In order to minimize required computations, it is important to identify Pareto points[23] at the early design stage where they limit the boundaries of multi-parametric spaces by asserting *efficient* or *non-inferior* points.

The goal of this chapter is to establish the method to effectively obtain optimal component architecture by identifying Pareto points and reducing architecture variants for multi parametric design variants. To achieve the goal, one will use the methodology suggested in [**79**].

First, a complex multi-parametric design space is decomposed into two-parametric design spaces that resemble a simple parent-child tree structure. Each of the two parameters is supposed to be a variant of the architecture that contradicts the order of their values. Once decomposed, the trees are ordered depending on the evaluation of the given parameter. Because of the contradictive nature of the given set, the trees are automatically ordered to have one increasing and the other decreasing value of the parameters. When all trees are ordered, one applies the associated constraints for each

[23] Pareto points indicate efficient, non-inferior or local optimal points which contribute to reducing the amount of design space.

tree. Combining the constrained architectural variants results in pareto-optimum design space with the ordered and constrained ACG, the optimal variant of architecture can be searched. The following sections will discuss each step of the method in details.

4.3 Decomposition of design space

Supposedly there are many variants of resources that can be described by the fundamental building blocks of the reconfigurable devices. These variances are the implementations of the description that results in different characteristics. Once these variances are available, one orders them using the contradictive parameters. Because of their opposing nature, the resource variances would be organized to increase from one end and decrease from the other end according to these parameters.

If there are S parameters, p_i, each parameter would have the associated constraint, p_i^{lim}. The parameters $\{p_i$ and $p_j'\}$ are paired to create the contradictory performance depending on the pending parameters.

4.4 Arrangement of the ACG

The arrangement of the ACG is achieved by:

- Horizontal level arrangement of variants according to the decomposed design space(s);
- Hierarchical tree structure construction using required resources.

4.4.1 Horizontal arrangement of resource variances

Horizontal level arrangement is only concerned with one type of resource and its variances. The variances of the resource would be associated with the number of used CLB logics, maximum operating frequency, etc when implemented in an FPGA. For each pair of evaluating parameters, any resource can be organized to order the value of parameters of the given resource, *R*, $p_s\left(R_{i,j}\right) \leq p_s\left(R_{i,j+1}\right)$ or $p_s^{'}\left(R_{i,j}\right) \geq p_s^{'}\left(R_{i,j+1}\right)$ where $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$. Since there is no constraint given for a resource in the component, it is difficult to dissect the branch with the given constraint p_i^{lim}. Yet, the sequencing graph comes with the fixed interfaces (v_i, v_j) between resources when the implementation of a resource is undetermined. This condition of interface between resources is referred as environmental characteristics in [**79**]. Figure **4-3** illustrates how environmental characteristics influence the resource selection. $R_{i,j}$.

If p_s is a measure of throughput for $R_{i,j}$, it is observed that there is a steady increase of p_s until $R_{i,k}$. For example, if the I/O specification of the design focuses on providing the solution for camera-related applications that requires processing 10-bit data, the variances of resource that can deal with 16-bit or 32-bit data would not provide better performance, while the variance that deals with 4-bit or 8-bit can accomplish the required output with degraded performance.

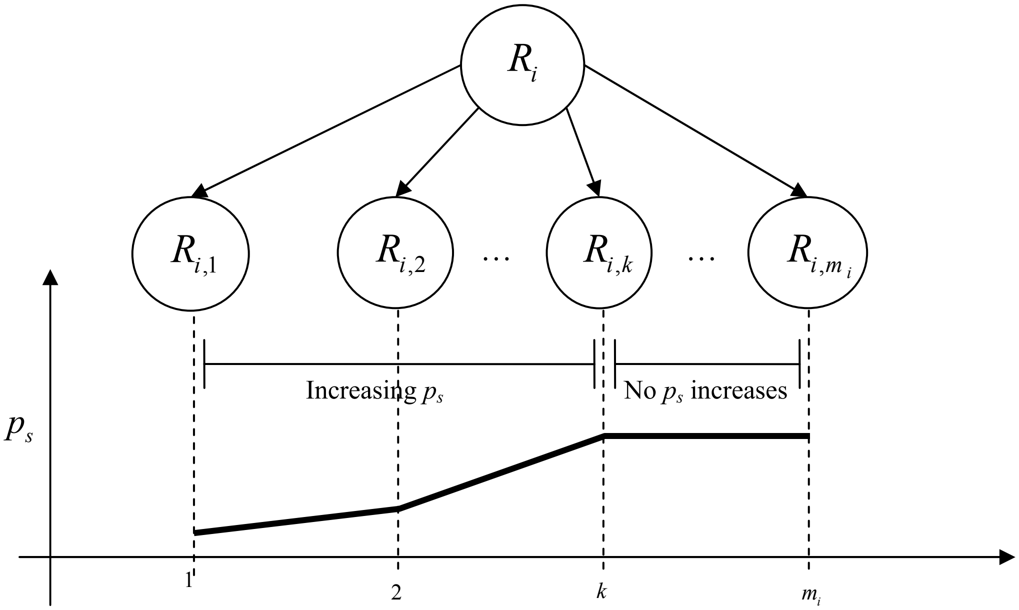


Figure **4-3**: Horizontal level Arrangement

The horizontal level arrangement, $p_s\left(R_{i,j}\right) \leq p_s\left(R_{i,j+1}\right)$ shows that there is no merit to implement $R_{i,j}$ when *j = k+1, k+2, … , m_i*.. The horizontal level arrangement prunes off some branches of resource variances by determining the optimal point (saturation point), $R_{i,k}$ of hardware investment where $p_s\left(R_{i,j}\right) = p_s\left(R_{i,j+1}\right)$ for *j = k+1, k+2, … , m_i.* It should be noted that the variances with lower hardware commitment still have the implementation possibility. The horizontal level arrangement applies to each level of ACG graph where each resource and its choice are concerned. To calculate the savings resulted from the arrangement, one can assume that each variance of resources is bounded by a logic function $R_{i,j} \Rightarrow f_{i,j}\left(C_1 \cap C_2 ... C_l\right)$ that creates the variance depending on the given conditions *C_j = 1, 2, … , l.*

With this assumption the creation of the variances for all resources in a component would cost as shown in Eq. **4-1**.

$$\sum_{i=1}^{n} m_i \times f_{i,j}\left(C_1 \cap C_2 ... C_l\right) \qquad \textbf{4-1}$$

The number of architectural variances for the component can also be expressed by the number of possible paths as shown in Eq. **4-2**

$$\prod_{i=1}^{n} m_i \times f_{i,j}\left(C_1 \cap C_2 ... C_l\right) \qquad \textbf{4-2}$$

The architectural variances are more susceptible to the penalty associated with the number of resource variances. Therefore, it is important to reduce the number of variances at the horizontal level. In a nutshell, the horizontal level arrangement reduces a considerable amount of architectures to be evaluated because m_i is reduced to m_{pi}.

4.4.2 Vertical arrangement of resources

The vertical arrangement of resources starts with calculating the average deviation of an associated resource. Beginning from the root of the ACG to lower subtrees, one obtains the maximum performance value, $P_s = P_{s_{\max}}\left(R_i\right)$ and the minimum performance value $P_s = P_{s_{\min}}\left(R_i\right)$. The average deviation between variances is calculated by Eq. **4-3**

$$D_{ave} = \frac{P_{s_{\max}}\left(R_i\right) - P_{s_{\min}}\left(R_i\right)}{m_i - 1} \qquad \textbf{4-3}$$

where m_i is the number of variances for R_i.

Once D_{ave} of all resources are obtained the resources can be moved to higher level than R_j based on the condition, $D_{ave}\left(R_i\right) > D_{ave}\left(R_j\right)$ where $i \geq 1$ and $j \leq n$. Therefore, the vertical arrangement of resources requires computing D_{ave} of the number of resources, n+2 that are evaluated with the number of parameters, *S* resulting $S \times \left(n+2\right)$ architectural variations.

4.5 Selecting a right architecture

Arranging the ACGs also requires P_s to be prioritized. Since some applications require much tighter P_s than other application, it is advised to prioritize the parameters, P_s depending on the application specific constraints

4.5.1 Determination of architecture validity for a constraint

Determining the subset of valid architectures becomes an easy task when the ACGs are horizontal and vertically arranged. Because the value of performance parameters is monotonically arranged, the binary search provides fast conversion, requiring $\log_2 \prod_{i=1}^{n} m_p\left(i\right)$ calculations of the P_s cost function where $m_p\left(i\right)$ is the number of variances for the resource, R_i. Finding the valid subset of the architecture is determined by selecting $P_s \leq P_s^{\lim}$ or $P'_s \geq P'^{\lim}_s$ through the binary search as shown in Figure **4-4**

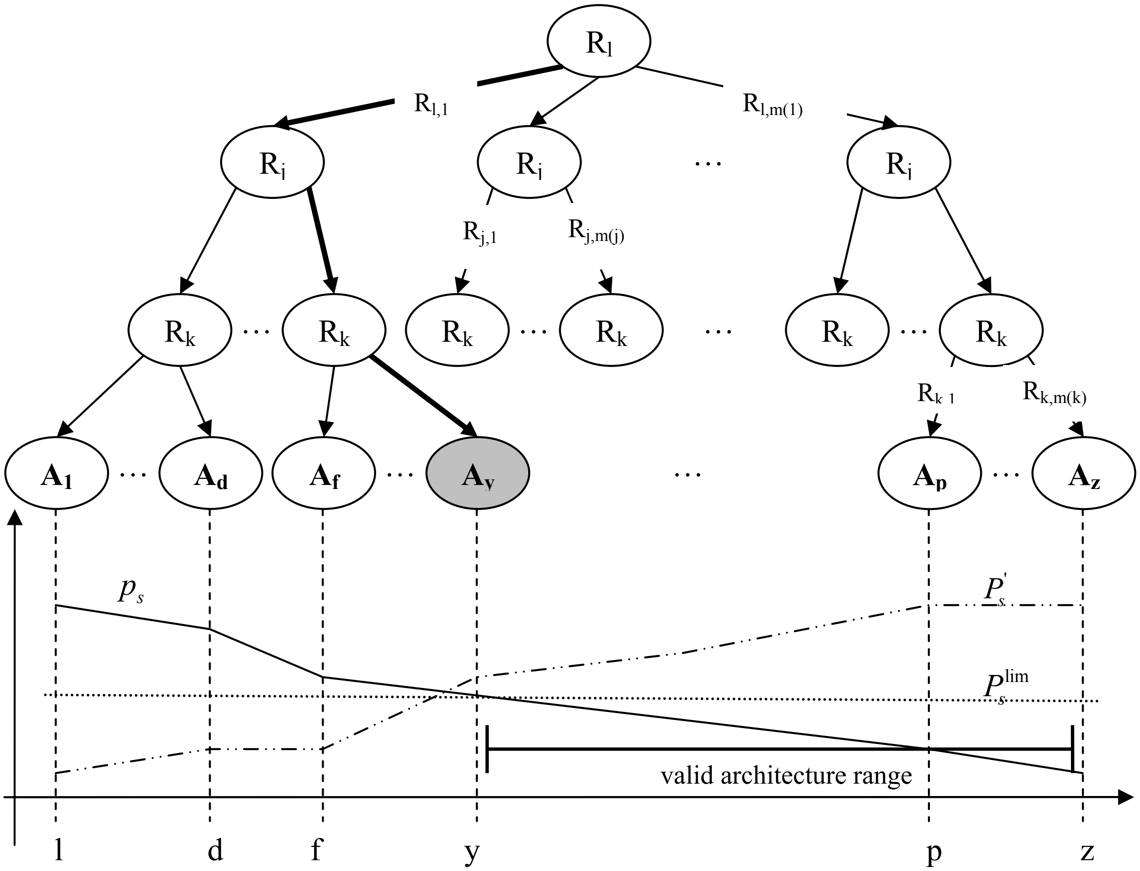
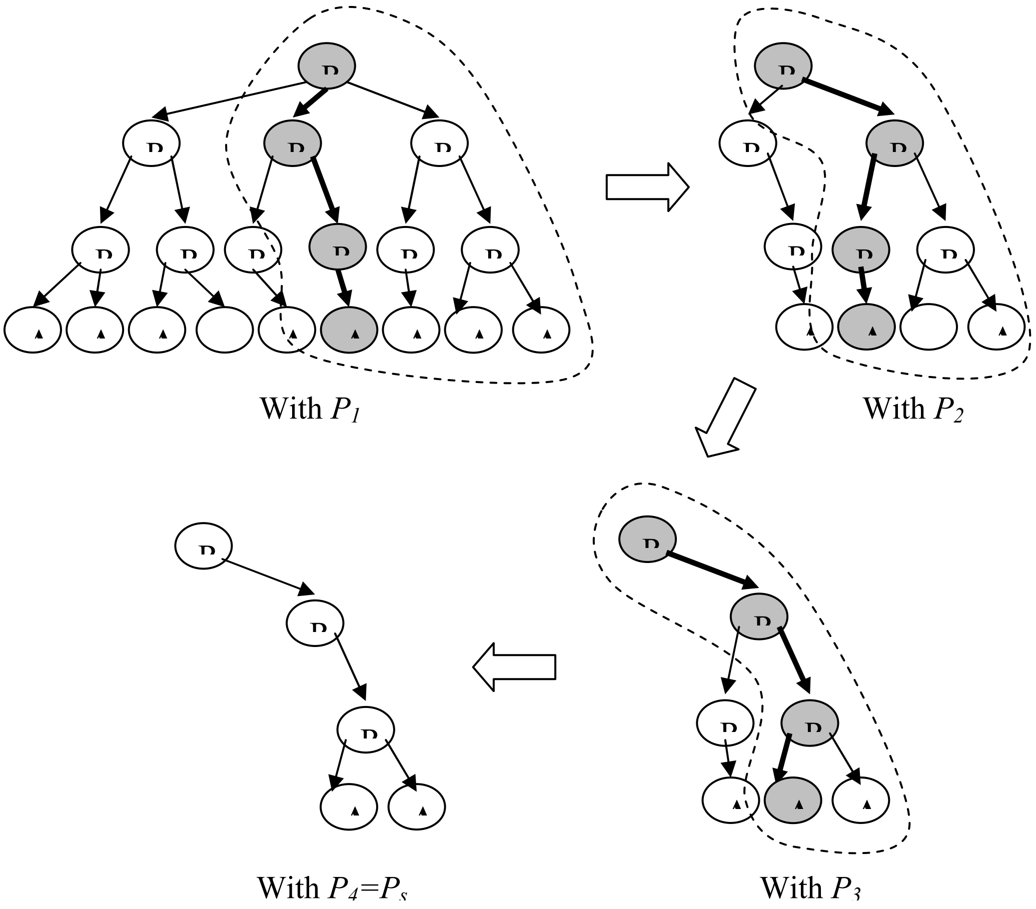


Figure **4-4**: Selecting valid architecture range with P_s

4.5.2 Determination of Pareto-optimal architecture subset

With the prioritization of parametric constraints, one searches the valid ranges for all constraints. However, the search space decreases as the previous search results in smaller subset of architectures as shown in Figure **4-5**



With P_1

With P_2

With P_4=P_s

With P_3

Figure **4-5**: Determination of Pareto-optimal architecture subset

First the conditions need to be prioritized. Then, each subsequent condition inherits a smaller range to search until the last constraint, P_s is applied.

4.5.3 Estimating the number of architecture variances

The goal of the above procedure is to seek the optimal architecture for the given constraints. From the given methodology one can easily extract the complexity of the computation based on representation of computational structure tree:

- For the input tree graph one assumes that one has *n* variances of resources. Thus, it is necessary to evaluate (2+n) variances for each p_i parameter;
- To determine the subset of architectures for a single parameter, it requires evaluating $\log_2 \prod_{i=1}^{n} m_p(i)$ calculations;
- To obtain the Pareto-optimal sub-space of component architectures, the *S* number of parameters should be applied.

The above steps result in the total number of calculations to be as shown in Eq. **4-4**.

$$N_{total} = S \times \left[(n+2) + \log_2 \prod_{i=1}^{n} m_p(i) \right] \qquad \textbf{4-4}$$

When the number of calculations involved with the methodology is compared with exhaustive search, $S \times \prod_{i=1}^{n} m_p(i)$, the computation complexity generally is reduced from $O(n^2)$ to $O(n)$.

4.6 Virtual Hardware Components Constraints

Components that one intends to synthesize and assemble are *virtual* because they instruct how hardware is configured but stored in a memory as bits of information. At the same time, they are *hardware* components because they describe the hardware of pre-manufactured programmable devices. So, one calls them Virtual Hardware Components (VHCs). With these VHCs, one can create many different types of systems and change their functionality as long as the resources in programmable devices are available.

While the granularity of VHC can be system, component or gate levels, it is advised to synthesize the VHC in the component level where it consists of the multiples of configurable units that require typically less than ≈ 10s micro-second time to reconfigure at the current technological level. However, the allowable reconfiguration time should be evaluated based on each individual case of system components associated with application.

The reduction of the number of variances in searching for the optimal component architecture demonstrates the feasibility of constructing a component library consisting of many different flavors of components. However there are different types of constraints that are applied for component where the global resources constraints are not yet clearly sought.

Those constraints that are yet defined to be applicable as the contributions at the component level remains as the flexible parameters to play with at the system level. However, the constraints such as the occupying area and operating frequency are the immediate constraints that choose one VHC out of selections, where Pareto-optimal set is conceivably used.

4.7 Outcome

The idea of VHCs and their library brings us the conceptual bridge to narrow the gap between hardware specific architectures (e.g., ISA) and application specific architecture (e.g., ASIC). The former tries to save costs by reusing hardware, hence minimizing the hardware development costs and the latter attempts to do the same thing by minimizing the hardware cost, hence obtaining better performance. To successfully construct such a system, one needs to construct component library that has a wide spectrum of VHCs applied with various constraints at the disposal of operating system. Allocation of operations to functional units and variables to storage elements for the component are presumed to be given as the behavioural description. Depending on the scheduling and binding of these resources, the characteristics of the component will be determined and various components that result in different constraints would be available.

Chapter 5

Reconfigurable System Design Methodology for multi-task and multi-mode applications

The goal of our reconfigurable system design is to implement the multiple functionalities in a reconfigurable device by reusing processing resources. To achieve a cost-effective reconfigurable system design solution, one needs to find a suitable reconfiguration granularity that results in an optimal static architecture. This chapter describes the architecture synthesis methodology used to obtain an appropriate reconfiguration granularity and to recognize a static architecture for the on-chip self-assembly of a reconfigurable multi-task and multi-mode workload

5.1 Introduction

The prior system design methodologies [**81**] have mainly focused on how to fit the individual application on available hardware system (i.e., Instruction Set Architectures – ISA) or how to construct the system for a range of specific applications (i.e., DSP, network processors and image processors) [**82**] [**30**] [**83**]. The proposed reconfigurable system design methodology distinctively searches for the optimal micro-architecture for a particular application, which is similar to the intention of an Application Specific Integrated Circuit (ASIC). However the reconfigurable system is different than ASICs in the following aspects. The new reconfigurable system:

- Uses dynamically reconfigurable devices to implement applications in hardware;
- Accommodates multiple functionalities as a multi-task and multi-mode workload;
- Re-uses reconfigurable hardware for a multi-task and multi-mode workload to reduce the system costs.

To realize the above results, the new architecture synthesis methodology should identify a multi-task and multi-mode workload, define an optimal reconfiguration granularity and create a static architecture that can accommodate the full realization of the workload.

5.2 Motivation

Today's computation-intensive computing systems more often deal with real-time and stream processing applications. Furthermore these applications embed a multi-task and multi-mode workload reflecting the market trend such as integration of multiple functionalities. As a result of increases in the number of logics available on an integrated circuit [**14**], complex applications composed of many tasks and modes can be implemented and executed simultaneously. But as the ASICs progressively become more expensive, the computing industry needs to find an economically viable solution that is not possible with the frequency scaling of ISAs along.

With the advent of process technology, the microprocessor systems have been able to provide processing solutions for many applications. Because of fixed hardware architecture, there has been no need to explore structural characteristics of applications.

However, one faces new challenges as the process technology no longer provides the promise of temporal advancements [**84**]. For example, the dual or quad core technology based microprocessors do not offer faster processor speed compared with Pentium family processors. When there is no trick to play in the temporal domain, the designers have looked naturally in spatial domain to find solutions.

The attempts to explore spatial domain of computing resulted in many different directions (e.g., ILP, VLIW, vector processor, cell processor, etc.) However, available performance improvements are often limited by:

- Low degree of intrinsic parallelism in the instruction/data stream;
- High complexity and time cost of dispatcher and associated control logic.

The instruction-based processors, including the parallel processing architecture, have been having a hard time keeping up with increasing and specific performance requirements of rapidly evolving applications and have become very cost inefficient for many other applications that do not have the data structure tuned for available parallel processing architecture. It becomes necessary to have a system architecture optimized for a specific application.

Additionally, as the Non Recurring Engineering (NRE) costs of ASICs progressively being more expensive[24], the computing industry needs to find an alternative solution. Reconfigurable systems may offer a cost-effective solution by replacing functional-level tasks that can reuse the same hardware areas. The integrated circuit

[24] Refer to section **3.3**.

becomes a system that is capable of hosting multiple functionalities that interact with the environment at the system-level.

Nonetheless, reconfigurable systems also have many disadvantages. The main drawback of a reconfigurable system (i.e., FPGAs) is a high unit cost[25]. Because of shorter product life cycle, FPGA designs were able to overcome the disadvantages of unit cost and compete with ASIC designs in terms of cost. To improve the competitive position of the FPGA further, lowering a unit cost would be a logical step. However, it is impossible to decrease the unit cost of FPGA as low as ASIC as shown in Figure **3-10**. The only way to amend the situation is by utilizing an FPGA to perform multiple tasks in the same hardware using run-time reconfiguration as shown in Figure **3-17**. One attempts to exploit reconfigurability to effectively reduce the unit cost of FPGA systems. Especially when the applications employ multi-task and multi-mode operations that are executed sparsely in time, the considerable amount of hardware savings can be achieved via reconfiguration. We propose a new design flow for partial run-time reconfigurable FPGAs to enhance its cost-effectiveness on the field operations by employing a reconfigurable system design methodology.

The fundamental principle of Electronic System Level (ESL) design is managing abstraction refinement and complexity while preserving design intent. The design intent of the reconfigurable system must live within the abstraction of the reconfigurable system. To preserve the abstraction of the design intent to reflect the structure of a specific workload, a design flow of the reconfigurable system is established. Throughout the

[25] Refer to the analysis used in Chapter 3.

design flow, architecture synthesis methodology comprises of several distinctive steps that lead to determination of a reconfiguration strategy at the system-level[26] as the responses of configuration events that reflects the structure of a multi-task and multi-mode workload. In the next section, one will look into each step of reconfigurable system design flow that explains what types of information are incorporated and how the reconfiguration should be performed to increase the overall value of the system (or to effectively decrease the unit cost of the reconfigurable system).

5.3 ASIC System Design Flow

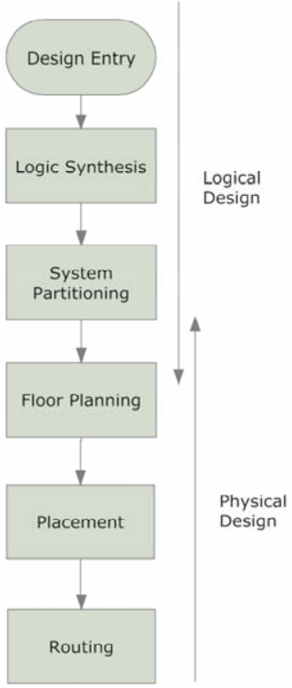


Figure **5-1**: ASIC design flow

[26] Refer to section **3.2.1.**

The first step to develop any electronic computing system is by understanding the problem. Normally design specifications are described in HDL which presents the behaviour description of the system to minimize the ambiguities that the normal language confronts.

The description of the datapath is generally expressed by a Data Flow Graph (DFG). Because of the granularity of HDL description, the nodes of the CDFG are generic operators and the arcs between them represent data dependencies and associated control values. The operators are scheduled into time slots. This scheduling determines the number of hardware required because all hardware in the same time slot should operate concurrently. The values that cross time slot boundaries must be stored in registers. Once the functionality of the system is expressed, the system can be divided into smaller subsystems. The process is called *system partitioning*. System partitioning is a system level design that determines the number of chips (or components) to be used for a design and the subset of the behaviour that needs to be implemented on each chip (or component). The intent of partitioning is to discover the structure implicit in the behaviour.

One starts to employ the idea of system partitioning into the reconfigurable system not only from the hardware point of view but also from the algorithms. Because the structure of behaviour is strongly present in many of today's internal or external data , one believes that one can discover the multi-task and multi-mode system specifications from events in data—so-called, configuration events and apply coarse system changes by monitoring the contents of data.

5.4 Reconfigurable System Design Flow

Because the system cost is directly related with reusability of components and the utilization of processing resources, the components need to be designed at other times rather than at the design time and the same processing resources should be reconfigurable to reuse for multiple functionalities. To utilize the reconfiguration effectively, the system can be constructed by assembling the components. Thus, the design time of the system can be shortened and the unit cost of the system can be reduced.

For instance, in the ASIC designs there are full custom, Cell-based (CB), Masked Gate Array (MGA) ICs, the full custom ICs are designed from scratch. The CBICs are built from the predefined selections of cells. The MGA ICs are constructed based on the pre-masked components. Even though the fixed cost of MGA ICs is higher than other ICs, the MGA ICs can be designed faster than other counterparts [**85**]. When the time-to-market increasingly becomes a bigger portion of system's costs, the pre-manufactured components that are applicable across many application areas started to make sense.

To cash in some of the benefits mentioned above, our design flow separates component construction (i.e., component level design) with system assembly (i.e., system level design) as shown in Figure **5-2**.

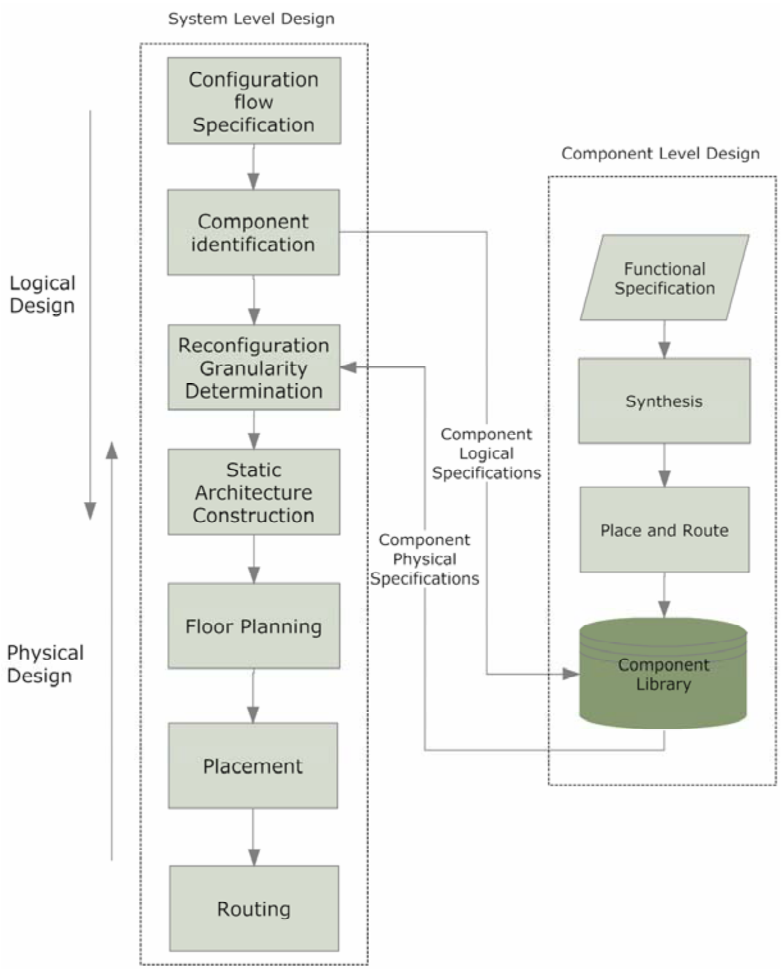


Figure **5-2**: RC Design Flow

One assumes that the structure of system events as well as their contribution for system's functionality is well known. Thus, upon availability of a system event specification, the system can be associated with components that exist in the component library. Due to the structure implicit in the system events, it is possible to assemble the system based on the hierarchical relationship (i.e., the structure of multi-task and multi-task workload) by the content of the events. Once the structure of the workload is obtained and the links are recognized, the reconfiguration granularity for the system can be determined. The reconfiguration granularity decides what part of the workload can be implemented as reconfigurable modules. The snapshots of the system can be expressed as a group of reconfigurable blocks and their links as shown in Figure **5-3**

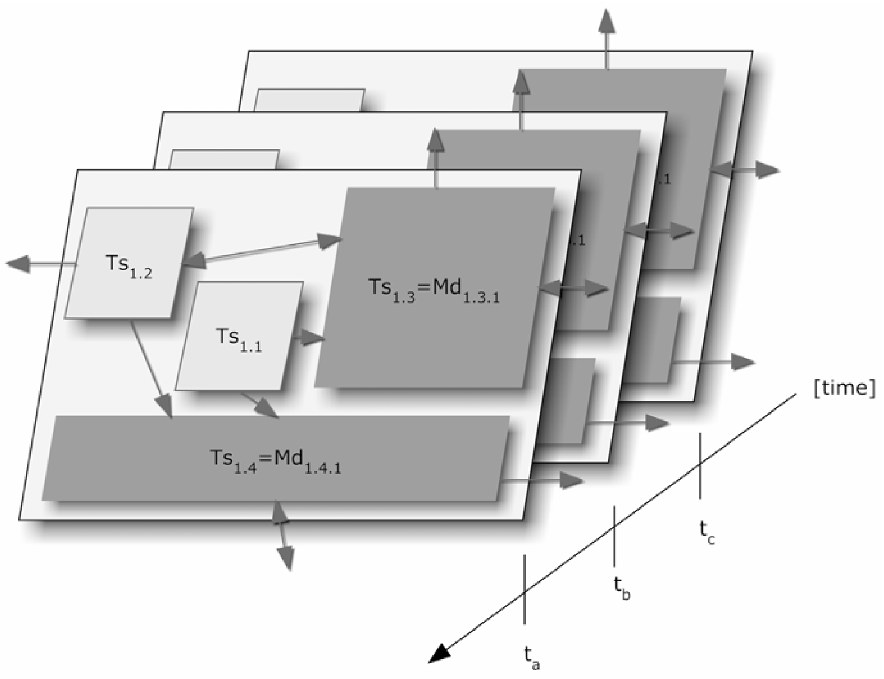


Figure **5-3**: An example of workload representation

The different grey scaling applied on the modules indicates functionality change(s) required by the values of corresponding system event(s) in time. As the reconfiguration strategy is established, the extraction of a static architecture can be initiated as illustrated in Figure **5-4**

The common area(s) is designated to accommodate the multiple functionalities over time. The area keeps the same interface(s) to communicate with other functionalities while being reconfigured. The rest of the system is extracted (i.e., subtracted) as the static architecture of the workload with a particular condition (i.e., values of system events). Depending on the condition, the static architecture can be reduced or expanded.

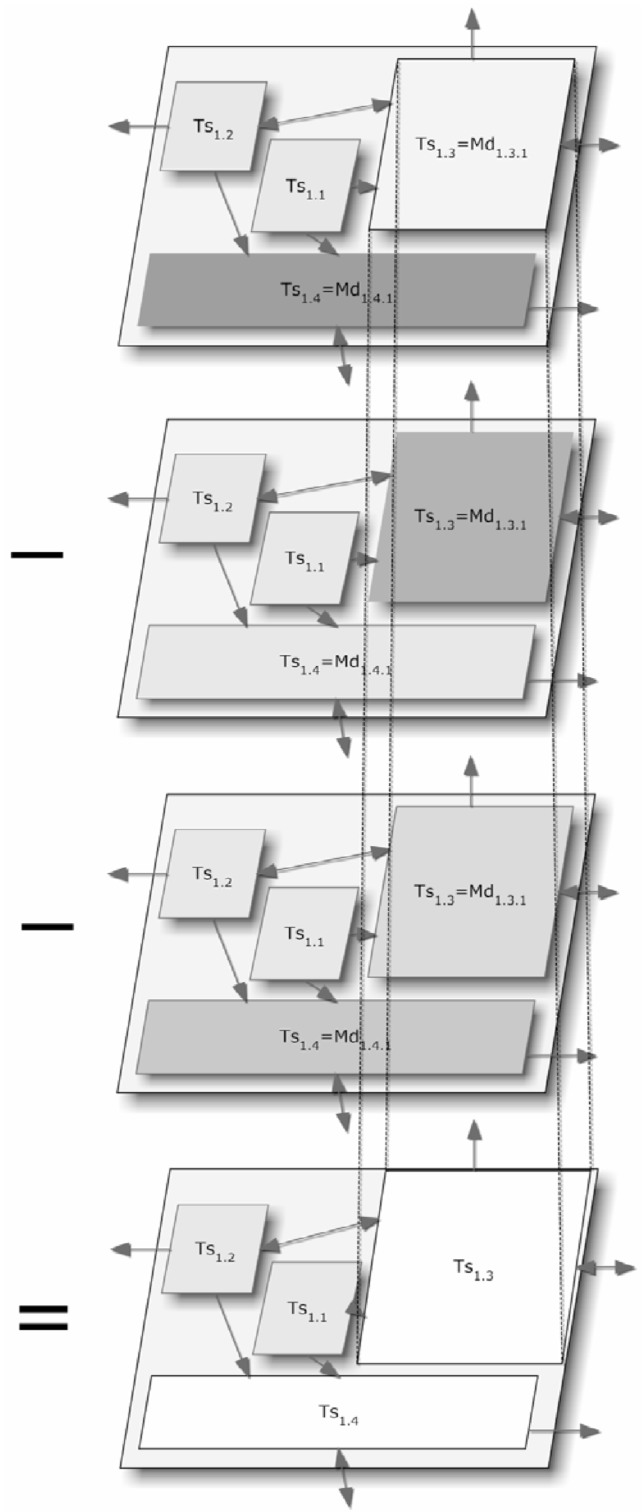


Figure **5-4**: An illustration of static architecture extraction procedure

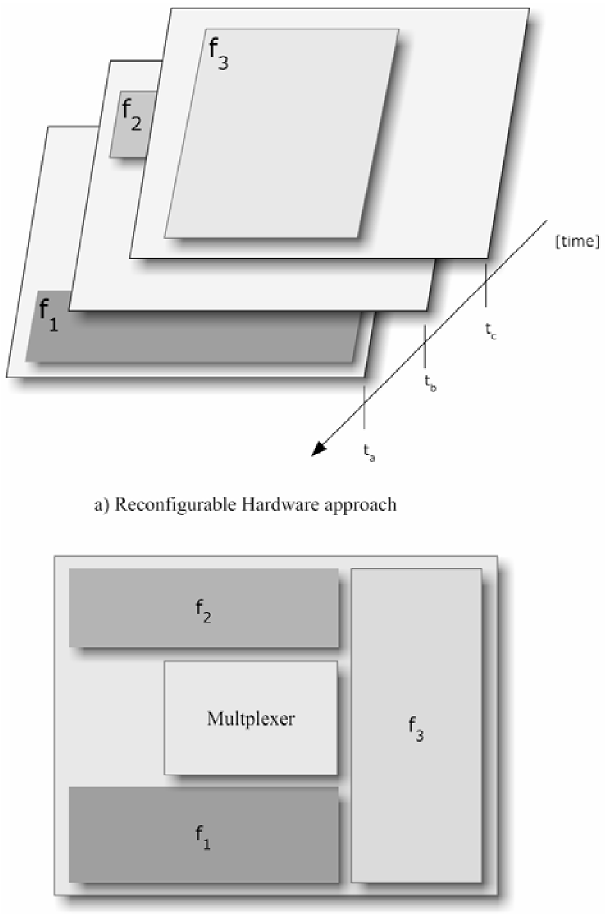
To sum up, the careful analysis of system events that reflect workload's functionality provides us the framework of the static architecture. Each variation of system events can potentially create different system functionalities. However, not all variations of system events should be implemented as reconfigurable module due to the timing overhead associated with reconfiguration process. The configuration granularity determination step reveals an appropriate size of reconfiguration blocks with respect to timing allowance available from the associated system event(s). Once the reconfiguration granularity is determined the design flow can extract a static architecture to implement the system at the beginning of system's operation.

5.5 Silicon Cost

Multiplexing multiple functionalities in a fixed hardware can be substituted by reconfiguring the same hardware for different functionalities in a reconfigurable hardware as shown Figure **5-5** .

However, it is difficult to compare fixed hardware and reconfigurable hardware due to their fundamental differences. Hence, the silicon cost regardless of their forms of implementation is considered to look at the benefits of the reconfigurable system as the extension of the cost analysis provided in Chapter 3.

It is obvious that not all applications implemented in a reconfigurable device can benefit from the reconfigurable approach. Thus, it is important to analyze what conditions of the applications are associated with the silicon cost for reconfiguration. In order to compare the silicon costs, one needs to set up the equations for both fixed and reconfigurable hardware approaches. Let us assume that a reconfigurable system can fit any function at all times when each function is a configuration of the reconfigurable system. The size of the reconfigurable system is decided by the size of the biggest function that needs to be implemented.



a) Reconfigurable Hardware approach

b) Fixed Hardware approach

Figure **5-5**: Fixed hardware vs. Reconfigurable hardware approaches

Since all functions can be implemented in a reconfigurable system, the silicon cost of the reconfigurable system does not increase by the number of functions that the system needs to deploy over time. However, as the number of functions increases the fixed hardware needs to employ bigger silicon as depicted in Figure **5-6**.

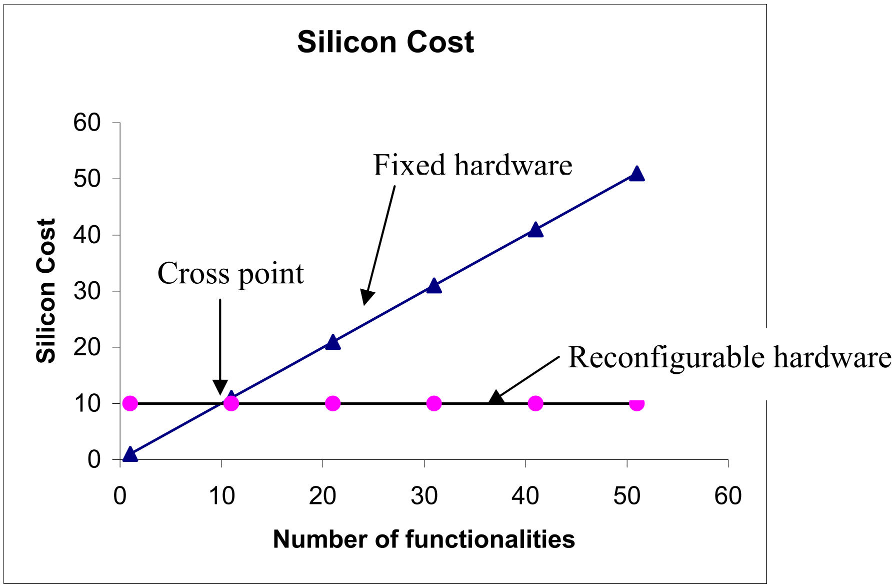


Figure **5-6**: Silicon Costs

The silicon cost of functions is normalized based on the size of the biggest function that requires an entire reconfigurable hardware. Hence, one can assume that all functions cost one configuration of a reconfigurable hardware which is equivalent to the silicon cost of implementing the function in the fixed hardware. The entire reconfigurable hardware can implement the functions that cost the maximum of one unit cost.

Let us assume that all functions cost one unit. Then fixed hardware can add the areas required for the functions. However, in fixed hardware they occupy a fraction of silicon area compared to reconfigurable hardware (e.g., 10%). Thus, with our assumption is all the functions in the reconfigurable hardware should cost ten times (e.g., 10 units) more than the cost of the function in the fixed hardware. The silicon cost for the reconfigurable hardware, $sc_{reconfigurable}$ is expressed by Eq. **5-1**

$$sc_{reconfigurable} = \frac{1}{s_{utilization}} \qquad \textbf{5-1}$$

With the given equations, the silicon cost of reconfigurable hardware starts to become more attractive when the number of functions exceeds 10 (i.e., $N > 10$) as shown in Figure **5-6**.

However, due to the areas demanded by multiplexing logics among multiple functions, the area (i.e., $\cos t_{mult} = \sum_{i=1}^{N-1} A_{mux}$) tends to grow faster than linear increase. If one assumes that it increases by quadratic equation (i.e., $\approx \frac{N^2}{4}$)., then, the silicon cost of the fixed hardware represents the quadratic increases.

At the same time, the fixed hardware approach can greatly utilize optimization techniques across multiple functionalities to extract the common area used by many functions. One can assume that the common area is shared by all functions equally. Therefore, A_{common} is defined as the ratio of common area that is shared across all functions.

The silicon cost for the fixed hardware, sc_{fixed} is Eq. **5-2**

$$sc_{fixed} = N(1 - A_{common}) + \frac{N^2}{4} ov_{multiplexer} \qquad \textbf{5-2}$$

where N is the number of functions that needs to be implemented, A_{common} is the ratio of common area that is shared among functions and $ov_{multiplexer}$ is the ratio of the overhead for multiplexing multiple functionalities. The silicon cost for the fixed hardware based on Eq. **5-2** is depicted in Figure **5-7**.

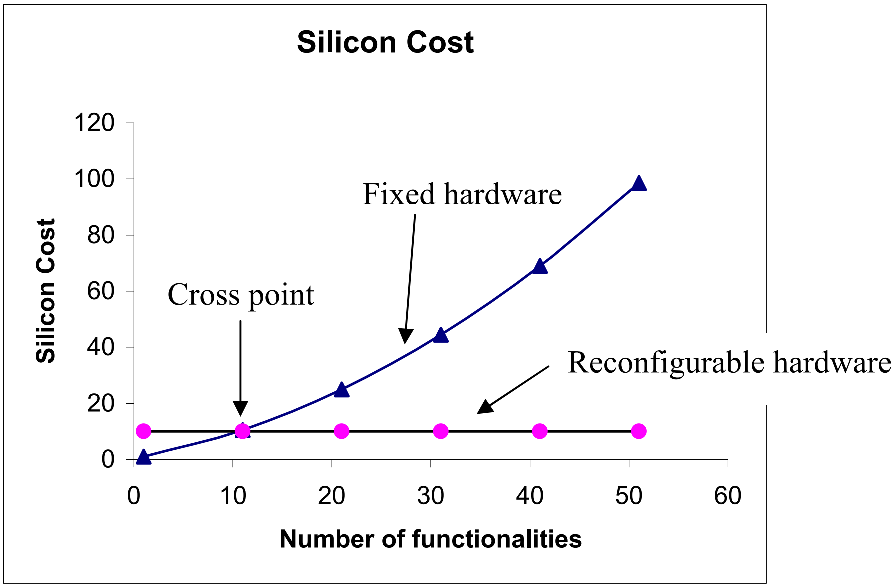


Figure **5-7**: Silicon Cost with quadratic increase in multiplexing area

For the reconfigurable hardware, not all functions are reconfigurable. Therefore, if there is a portion of non-reconfigurable functions, they should be implemented as the functions in the fixed hardware by multiplexing them. Yet they would cost more to implement in reconfigurable hardware than in fixed hardware. They should not include the multiplexing overhead because they are amortized under the silicon utilization of reconfigurable hardware.

The silicon cost for reconfigurable hardware, $sc_{reconfigured}$ should be first divided by the silicon cost for the functions that are reconfigurable, $sc_{reconfigurable}$ and the silicon cost for the functions that are not reconfigurable, $sc_{non-reconfigurable}$ as shown in Eq. **5-3**.

$$sc_{reconfigured} = sc_{reconfigurable} + sc_{non-reconfigurable} \qquad \textbf{5-3}$$

The ratio between reconfigurable and non-reconfigurable functions is determined by the given $r_{reconfigurable}$. First $sc_{non-reconfigurable}$ requires the functions to be implemented as in the fixed hardware. The silicon cost of non-reconfigurable functions needs to be calculated as the same way in the fixed hardware except that the multiplexing overhead is amortized by the abundant routing resources that are available in the reconfigurable hardware expressed by the low silicon utilization, $s_{utilization}$ as shown in Eq. **5-4**

$$sc_{non-reconfigurable} = \frac{\left\{N\left(1-r_{reconfigurable}\right)\left(1-A_{common}\right)\right\}}{s_{utilization}} \qquad \textbf{5-4}$$

where $s_{utilization}$ is the ratio of a typical silicon utilization for implementing functionalities in the reconfigurable hardware – routings and configuration control logics are also considered to calculate the silicon cost and $r_{reconfigurable}$ is the ratio of the functionalities that can be reconfigured using the same area.

As the multiplexing overhead exists for the fixed hardware approach, there is the reconfiguration overhead that exists in the reconfigurable hardware approach. However, unlike the multiplexing overhead, the reconfiguration overhead stays to be constant as shown in Eq. **5-5**.

$$sc_{reconfigurable} = r_{reconfigurable}\frac{1}{s_{utilization}} + ov_{reconfiguration} \qquad \textbf{5-5}$$

where ov_{reconf} is the overhead needed for reconfiguring multiple functionalities. ov_{reconf} stays pretty much the same with even increasing N because of virtualization of the hardware (i.e., configuration files) in the form of memory. When all of the above conditions are considered, the fixed and reconfigurable silicon cost shows the following characteristics shown in Figure **5-8**

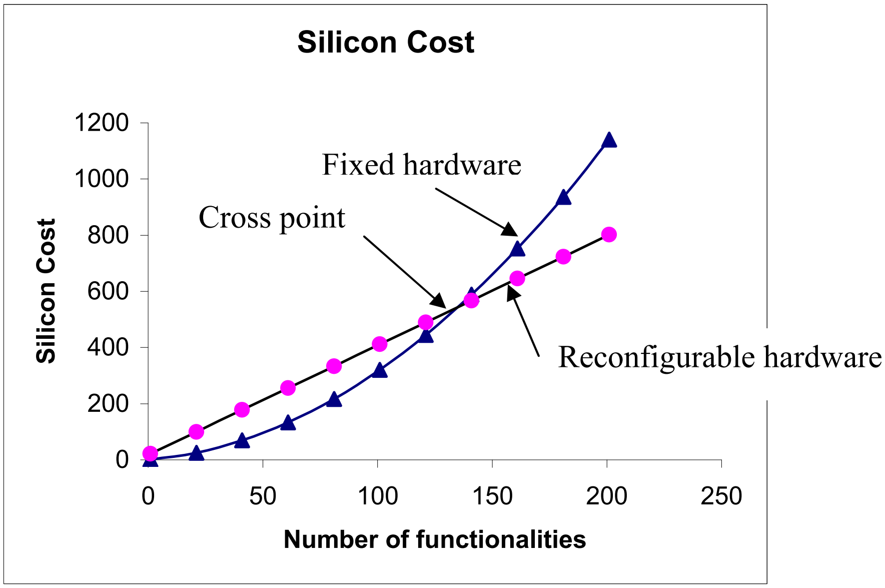


Figure **5-8**: Silicon costs: fixed hardware vs. reconfigurable hardware

As it is shown, the cross point of the silicon costs is moved further out. It states that the application requires being *"very"* multi-functional (e.g., 40% of the functions which means 54 out of 139 functions to be reconfigurable).

5.5.1 Analysis of silicon costs

In the Chapter 3, one analyzed how changing electronic *market trends* as well as fast shifting *fabrication technology* and the rising of *new applications* that demand integration of multiple functionalities makes the reconfigurable computing a preferable choice compare to ASIC approach. However, one did not explore what extent of virtualizing hardware or what conditions of the system makes the reconfigurable system to be more cost-effective. The view of silicon cost adds another way to explain the cost-effectiveness of the reconfigurable systems.

To begin the analysis, let us take a look at what parameters can affect the outcome of the silicon cost. The reconfiguration overhead used in Eq. **5-5** does not change the cost much except that it adds constant values onto the reconfigurable system's silicon cost. The more crucial parameters are the ones which change the rate of increase. One assumes that $A_{common} = 0.35$, $ov_{multiplexer} = 0.1$, $r_{reconfigurable} = 0.4$, $ov_{reconfiguration} = 10$ and $s_{utilization} = 0.25$ are kept as constant while one of these parameters are changed from 0.1 to 0.9. Figure **5-9** plots the cross points of silicon costs when one of values is increased.

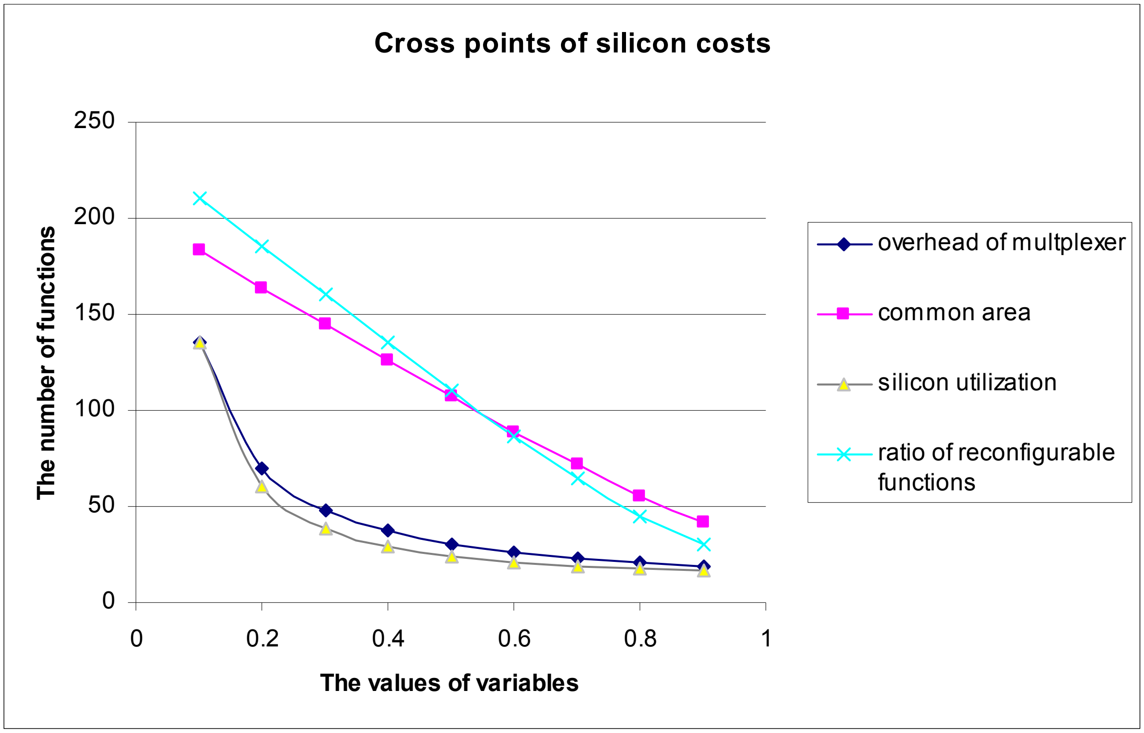


Figure **5-9**: Cross points graph

The overall trend of Figure **5-9** is that the number of functions needed to equalize the silicon cost between fixed hardware and reconfigurable hardware decreases as the values of the parameters increases. In other words, the reconfigurable hardware becomes more competitive when the values of parameters improve.

The conditions of the system that affect the silicon cost are the silicon utilization for reconfigurable hardware and the overhead of multiplexing for fixed hardware. The rate of reconfigurable functions for reconfigurable hardware and the common area of the functions can be considered as the trends in today's applications. It is not hard to imagine that the silicon cost becomes more competitive as the silicon utilization or the ratio of reconfigurable functions in reconfigurable hardware increases. Both parameters allow the reconfigurable system to put more functions in a given hardware. Conversely, as the overhead for multiplexing increases the fixed hardware approach becomes less effective. The only parameter that turned out unexpectedly is the common area. Since the common area affects the silicon cost for both systems, it becomes a bigger factor in the reconfigurable devices as shown in Figure **5-10**.

From the traits of running application's point of view, the more reconfigurable functions (and the more common area) the applications that have, the cheaper the reconfigurable system should cost. Other parameters do not contribute to the traits of running applications but to the characteristics of hardware systems. As the ratio of reconfigurable functions increases, the reconfigurable computing system becomes the more attractive solution. In order to increase the ratio of reconfigurable functions, the amount to reconfigure needs to be decreased to accommodate more functions. Because the static architecture reduces the amount of hardware to be occupied by multiple units, the exploitation of the static architecture among configurations is recommended.

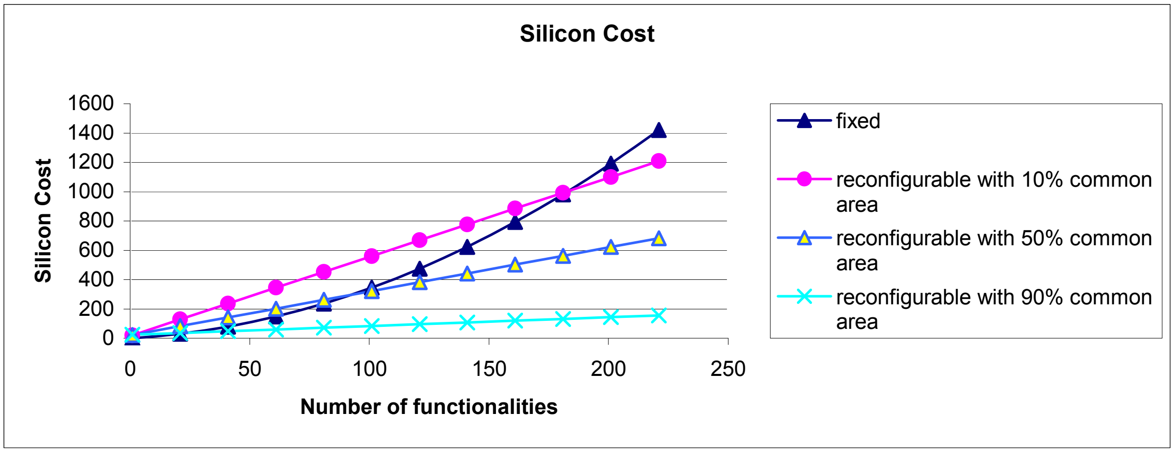


Figure **5-10**: Silicon cost with varying common area

The architecture synthesis methodology analyzes a multi-task and multi-mode workload in a reconfigurable system and suggests a static architecture that can work beyond the granularity of reconfigurable system that the workload demands.

In the following sections one will take a look at how these multiple functionalities of the workload can be realized and be incorporated into the reconfigurable system systematically.

5.6 Configuration Flow Specification

With increasing development of communication medium (e.g., optical fiber) and expanding functionality of consumer devices, it is predicted that stream communication would be a natural representation to transport and process all types of data from textural to visual. For instance, a MPEG-4 scene consists of many objects that require different processing elements, thus resulting different data flow graph. Since the scenes can be dynamically composed with a large variety of objects, the full design of decoding a scene might be too complex to be directly implemented in hardware. Figure **5-11** demonstrates a simplified scene description of MPEG-4 shown in a hierarchical graph.

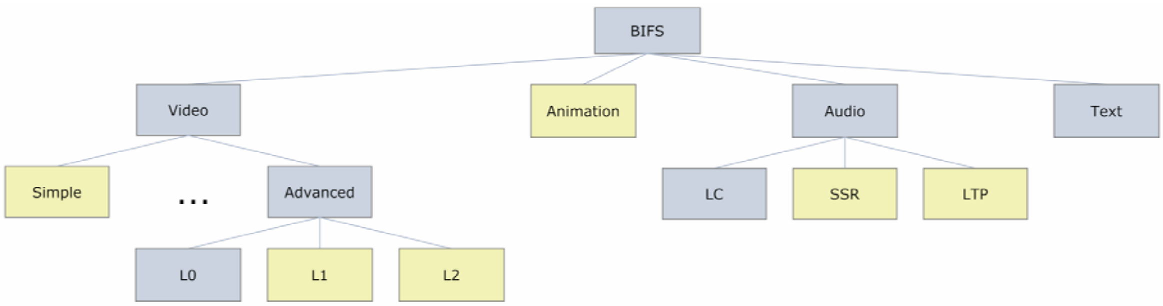


Figure **5-11**: Simplified structure of MPEG-4 scene description

Yet if each scene of MPEG-4 requires combining a few paths of configuration flow shown as the grey areas in Figure **5-11**, the system can save a considerable amount of hardware space compared to the full implementation. If the scene's dynamic changes requires small modification that can fit reconfiguration between scenes, the cost-effectiveness of the system can increase over time.

Similarly today's computing intensive applications such as Digital Signal Processing, multimedia standard and communication protocols have the headers that entail the sequence of the operations for the impending data or the prefixed operations depending on the algorithms that the system needs to deploy. Based on the assumption that system events carry implicit structure of system's behaviour, one can analyze the system according to the system events. Specifically the structure of the multi-task and multi-mode operations should be mapped onto the stream of system events.

Creating a hierarchical design that identifies a multi-task and multi-mode workload, starts with the configuration flow specification as shown in Figure **5-2**. In computer science, the control flow specification is known as the order in which the individual instructions (or functional calls) of a program are executed or evaluated. We define configuration flow specification as:

Definition 23: configuration flow refers to the order in which the groups of particular operations in a reconfigurable system are executed.

Since the reconfigurable system is the mere assembly of the operations, the order indicates to the configuration controller the set of moments when the alternation of operation(s) becomes necessary. In our reconfigurable system, the individual operations are grouped as a mode, task, thread and application. As a result, the configuration flow specification identifies the order based on the granularity of the given group(s).

Since the configuration flow specification should be tailored to each application, it is difficult to generalize the steps involved in the process. However, there are some steps that are common for all system designs [**86**] and provide aids to identify configuration flow in early design stage. One needs to:

- Recognize the environment which the system is modeled in;
- Create a context diagram that expresses overall purpose and represents input(s)/output(s) of the system;
- Identify all events that serve the purpose of application;
- Outline the responses to the associated events.

One starts by creating a preliminary system model consisting of a context diagram and an event list. The context diagram is a special case of the dataflow diagram, where a single node represents the entire system. The event list is a set of external stimulus to which the system must respond. The event list can consist of system-level events that occur without any external provision.

First to note, the estimation of timing can not be carried out with events that are naturally aperiodic. Hence, one only considers the systems with real-time stream applications where input data is structured with exact timing information to determine computational needs, whose assumptions are similar to the ones made in Synchronous Data Flow (SDF) [**87**].

5.6.1 Defining the environment

The first part of system modeling involves outlining the interfaces between the system and the environment that resides outside the system. Outlining the interfaces involves knowing what information comes into the system from the environment and what information goes out from the system to the environment. The context diagram draws the boundary of the system and illustrates interfaces that are inputs and outputs of the system. To construct the environment model, one starts to define *context diagram* and *event list* for each system.

5.6.2 Event list

The event list is generally accepted as a narrative list that invokes the response of the system. The events can be categorized into three different classes: flow-oriented events, temporal events and control events.

Definition 24: A flow-oriented event is one that is initiated by the necessary condition(s) of a dataflow.

The dataflow alerts the system that the event has occurred when a piece of data has arrived. This will result in a dataflow on the context diagram. Yet each dataflow can be invoked by necessary data required by the system to process an event or an event itself. There are also temporal events.

Definition 25: A temporal event is the one that is triggered by the arrival of data at a point in time.

The temporal events are normally initiated by internal clocks which keep track of time for the system. The temporal events may request inputs from terminator(s). In this case, temporal events associated with dataflow(s) are not depicted as the events. The last of three categories are control events.

Definition 26: A control event is considered to be a special case of temporal events, which are not bounded by the regular passage of time.

However in the context of real-time and stream applications, even configuration flows are associated with timing characteristics of the real-time stream application. Thus, the appearance of control events is predictable with known characteristics.

Because our interest is to extract the *coarse* configuration flow that exploits reconfigurability of the system, one pays attention to the timing information related with the control event(s)[27] for the system. In order to know how the control events influence the system and their timing information, one needs to learn how a context diagram incorporates the control events to express a multi-task and multi-mode workload.

5.6.3 Context diagram

The context diagram is a particular case of the dataflow diagram in which a single bubble represents the entire system. The context diagram can consist of:

- Terminators – external entities with which the system communicate;
- Input data – data which the system receives and which needs to be processed by the system;
- Output data – data that is produced by the system;

[27] The term, *control event* is used interchangeably with *system event*.

- Stores – data storage medium that are shared between the system and the terminators. The store can be created by either the system or the terminators.

We start to construct the system model by events and bubbles that show event responses.

Let's view the system as a data transformation, $Y = f\left(X\right)$ producing an output, Y as it accepts an input, X as shown in Figure **5-12**

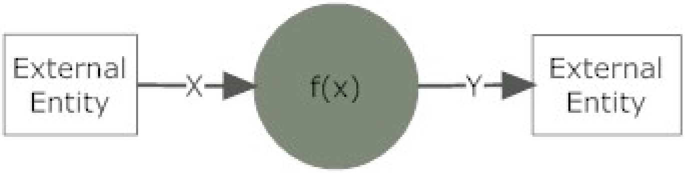


Figure **5-12**: An essential Context Diagram

If the system requires doing one task all the time, then the context diagram in Figure **5-12** represents a correct description of the system and designers should try to implement the given transformation with minimum resources. However, *system* means much more complex and versatile functionalities in today's standards. The representation of these functionalities can be interpreted systematically when one uses the control events. To distinguish the control events in a reconfigurable system, one refers them as *configuration* events.

Definition 26: A configuration event is considered to be a special case of control events that requires reconfiguration of system's functionalities in a reconfigurable device.

Configuration events are considered to be as *prompts*. They do not influence the functionality of the system directly but rather switch functionality of the reconfigurable system on and off. Thus, it controls the behaviour of the reconfigurable system over time. The difference between prompts and input data is equivalent to the difference between a call to a subroutine and the parameters of a subroutine. One needs to identify the configuration events that invoke the subroutines directly knowing the other subroutines are not served at the same time. A simple illustration of a dual-state system is shown in Figure **5-13**

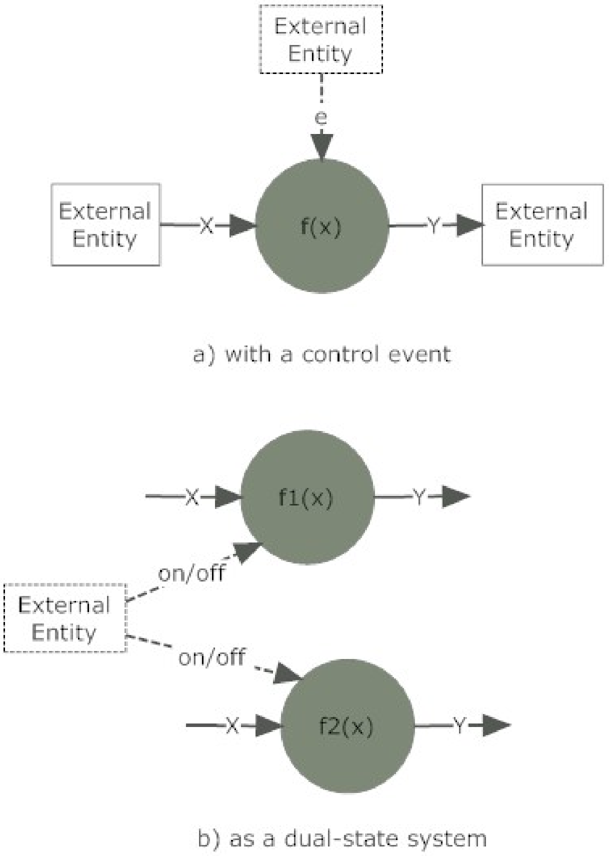


Figure **5-13**: Examples of Context Diagram

While the ASIC approach explores hardware overlap between data transformations (or states) and creates an optimal solution for implementing a full system, $M = \{s_1, s_2, ..., s_n\}$, our approach focuses on implementing a slice of the system, $s_i = f_i$ and exploits the timing redundancy where the system and application allow reconfiguring the system for a different functionality.

If you generalize Figure **5-13**, a reconfigurable system can be modeled as a set of configuration, $M = (\sum, C, \delta)$ where Σ is the configuration events (a finite non-empty set of symbols); C is a finite non-empty set of configurations; δ is the configuration transition function: $\delta : \Sigma \to C$ and where each configuration represents the data transformation at a time, $c_i = f_i$ associated with $E_i \subseteq \sum$. Each configuration $c_i = f_i$ can be portrayed as an application in the workload. Since the reconfigurable system deals with stream data, $(C,\vartriangle)$ the event, E_i should be available for each stream unit, c_i, where $\vartriangle_{i+1} - \vartriangle_i \leq d$. The event E_i is a set of all configuration events $\{e_1, ..., e_n\}$ that are committed to the configuration, c_i.

The difference of the above model with FSM is that δ does not consider the current configuration of the reconfigurable system in order to identify the next, but it indicates the changes necessary, $c_{i-1} - c_i$ to be configured for the next.

The model of leveled context diagram with event lists are derived from [**88**] which is an extension of the state machine and state diagram. Thus, the configuration flow specifications are established by constructing a leveled context diagram and identifying the tools associated with events by applying the structural analysis [**89**] and adapting state diagram [**88**] models.

5.6.4 Leveled context diagram

Depending on complexity, the system can be partitioned further (i.e., Leveled Context Diagram). Since the configuration events are mostly associated with hierarchical data structure of the application (e.g., multi-dimensional stream data), the configuration events can also be grouped into the levels of a hierarchy. One partitions the system until all identifiable configuration events are consumed. The context diagram that represents the hierarchical organization of functional nodes is called the leveled context diagram.

The leveled context diagram serves two main purposes in the configuration flow specification. One is to identify the functional nodes as applications, threads, tasks or modes. The other is to identify the necessary micro network to enable communication between nodes. The definitions of the terminology used for reconfigurable system are restated from section **2.5.1**.

Definition 1 A *task,* ***Ts*** is a group of arithmetic (and/or logical) operations that are necessarily interconnected to perform the described computation.

Definition 2 A *mode,* ***Md*** is one of ways executing a task that is bounded by constraints, and specifications.

Definition 3 An *application, **Ap*** is a group of *task*(s) that are interconnected (or disjointed) to carry out computational requirements for a system given at time(s).

Definition 5 A *thread, **Th*** is a connected set of task(s) given in an application.

Definition 6 A *workload, **W*** represents all application(s) that a system needs to implement at different times.

Any node can be an *application* when there is an event(s) with which it associates. The *application*s in the same level should contribute to the same outputs and only one of the *applications* should be active at any point of time. The title of *thread* is given by identifying the group of *tasks* that are divided by a particular set of input and outputs in the *application*. The *threads* should not share the same outputs and there should be no communication between *threads*. The difference between *task* and *application* is that there is no more configuration event for *modes* of a *task*. Reconfiguration of the system can happen in *application*, *thread* and *task* levels where:

- *Application* level utilizes temporal redundancy of the system where the sub-modules can use either spatial and/or temporal redundancy of the reconfigurable system;
- *Thread* level uses spatial redundancies of the reconfigurable system;
- *Task* level uses temporal redundancies of the reconfigurable system where there is no further sub-module available.

After identification, each node is relabeled to be one of types (e.g., *application*, *thread*, *task,* and *modes*).

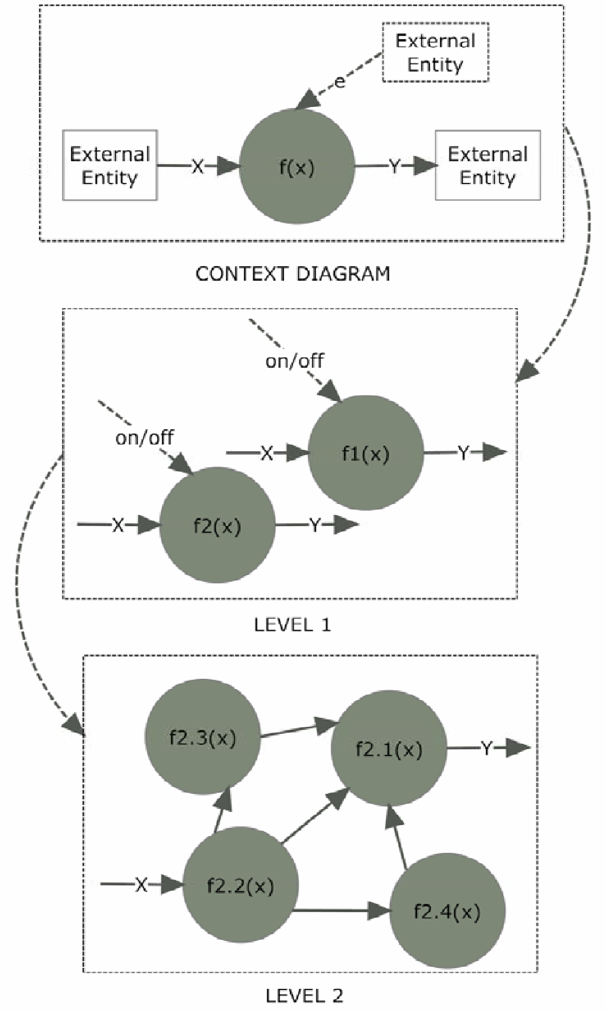


Figure **5-14**: An example of leveled context diagram

The second identification regards the framework of micro network. One assumes that the micro network can be implemented separated from functional nodes. The micro network is not concerned with the functionality of the system. However, it deals with the intercommunication between nodes. The micro network can only be identified by the leaves of the leveled context diagram. These nodes do not have any more configuration event to consider. The identification of the micro network is considered in the level of task where internal operations are driven by common set of input(s) and the operations produce results for the common output(s). The design also should consider that the bandwidth requirements between tasks do not exceed the capacity of micro network in the system.

Figure **5-14** exemplifies a simple case where configuration events attribute to the characteristics of a dual-state system by turning on and off the associated application. In this example, the f1 and f2 are the applications of the system. Either of f1 or f2 can be on, but not both at the same time due to the output conflict (e.g., both shares Y as the output). While the f1 and f2 alternate their insertion to the system based on the value of the configuration event, its duration is fixed by the characteristics of stream input data with which each task is tied. Thus, f1 and f2 are called as applications of the system. Each bubble can be described further by illustrating the subsequent levels. However there would be no hardware changes necessary because of no configuration events were assigned beyond the LEVEL 1.

Figure **5-15** illustrates the case of creating threads. Contrary to Figure **5-14** the configuration event in Figure **5-15** does not initiate (or terminate) the bubbles directly as the outputs of the system is sufficiently fulfilled by the bubbles in the LEVEL 1. Note that the applications in all levels should be able to produce all outputs that belong to the context diagram. As it is observed, the unionized outputs of both bubbles equal to be one in the context diagram.

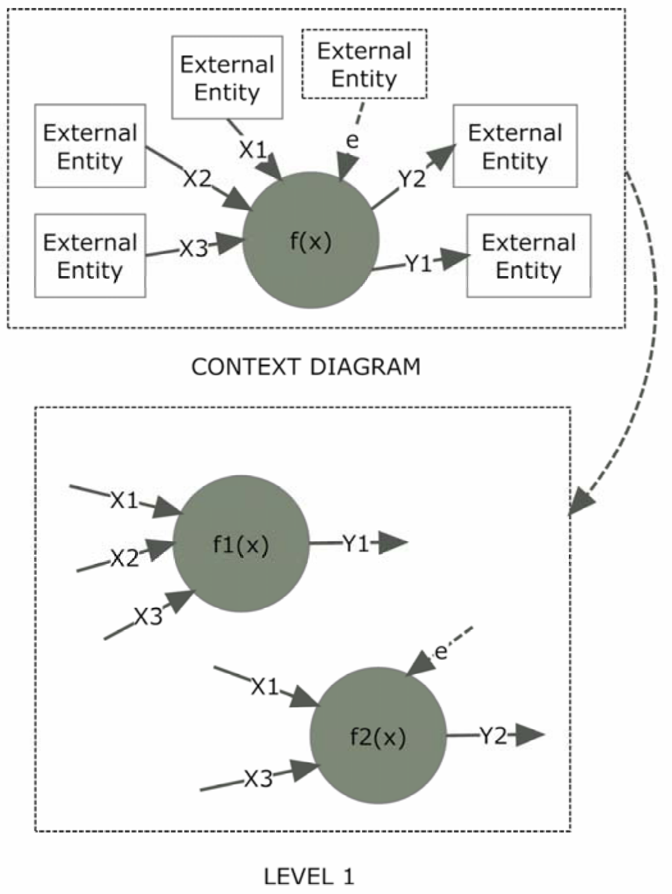


Figure **5-15**: An example of threads in dataflow diagram

Yet the configuration event influences subsequent bubbles in the lower level. There is no configuration event that differentiates the functionality of the system at the LEVEL 1, since f1 and f2 do not use each other's intermediate values and their outputs do not overlap they are classified as threads in the LEVEL 1. Depending on the complexity of the system, there can be applications and threads in the subsequent levels. The applications are created by associating the configuration events. Within an application, if the outputs are indivisible, for the functionality to be carried, there should be no more thread available.

Figure **5-16** shows an example of tasks and modes.

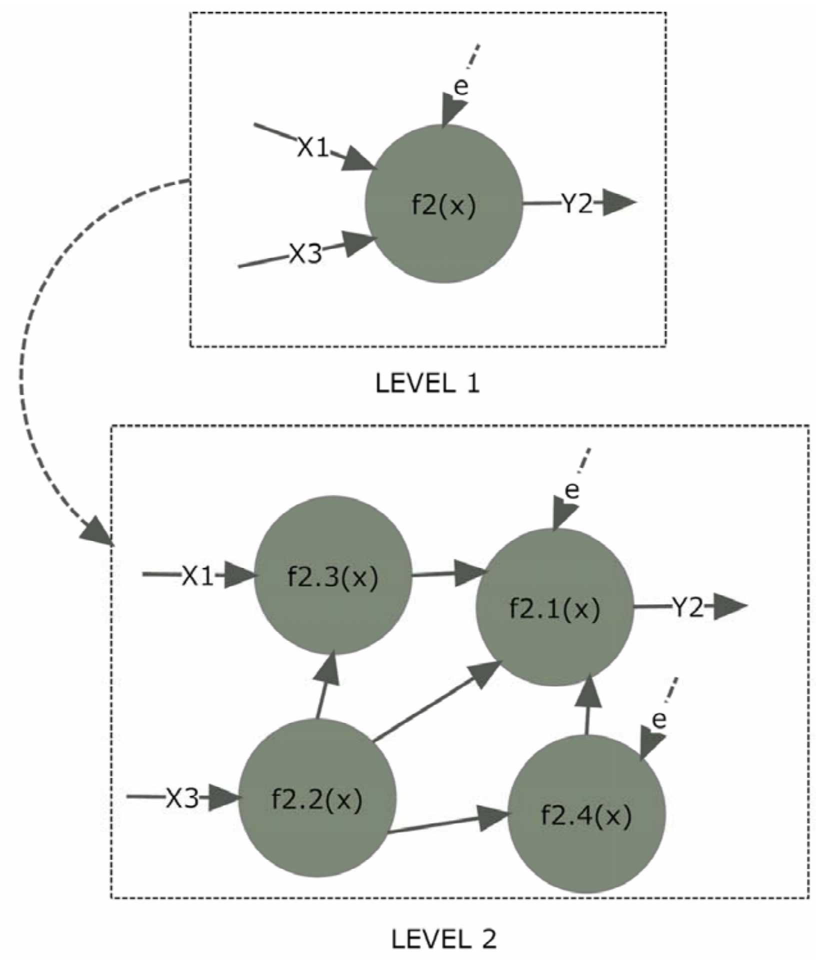


Figure **5-16**: An example of tasks and modes in context diagram

When there is no subsequent thread (or application) available – in other words, when all tasks are interconnected, the assigned configuration event(s) is used to allocate the modes for the tasks. The level which designates the implementation of tasks and modes is one that contributes to constructing the static architecture of the system.

So far one has studied the typical examples that show how applications, threads and modes are created in the leveled dataflow diagrams. The general feature of the leveled context diagram is that it allows the users to look at the system from the various levels of details. More importantly for the reconfigurable system, it provides the interconnection details of tasks and input/output relationship of tasks that are used to construct the static architecture of reconfigurable systems.

In a nutshell the reconfigurable system design methodology recognizes and labels the nodes as *applications*, *threads*, *tasks* and *modes*. Then, it enlists the necessary communication interfaces for the micro network of the system. In the next section one takes a look at the steps of our methodology to design a cost-effective reconfigurable system using dynamic reconfiguration.

5.7 Identifying a multi-task and multi-mode workload

The identification of a multi-task and multi-mode workload first requires obtaining the precise relationships between the configuration events and tasks in the leveled context diagram. There is the relationship reached at each level of the context diagram. Hence the system is divided into the two levels that consist of a parent node and children nodes. By applying the decision diagram given in Figure **5-17** for each pair of parent and children, one finds that:

- If there are multiple independent bubbles in a level, then the bubbles can be either threads or applications;
- If the bubbles in the level share the same set of outputs, they must be applications. The event(s) associated with these applications should allocate a control value that initiates one application at a time. The applications in a level are employed by the same event. They represent temporal partition of the system;

- If the output(s) of bubbles are mutually exclusive, threads are created for each group. The threads divide the application into smaller versions that do not interfere with each other. They represent spatial partition of the system;
- If all tasks in a level are interrelated and there is associated configuration event(s), then the task(s) that does not associate with the configuration events is regarded as static task(s). Otherwise the task(s) possesses the modes of operations according to the configuration event(s);
- If there is either further divisible outputs or more configuration events, then each mode of task(s) can issue threads or applications respectively in the subsequent level(s).

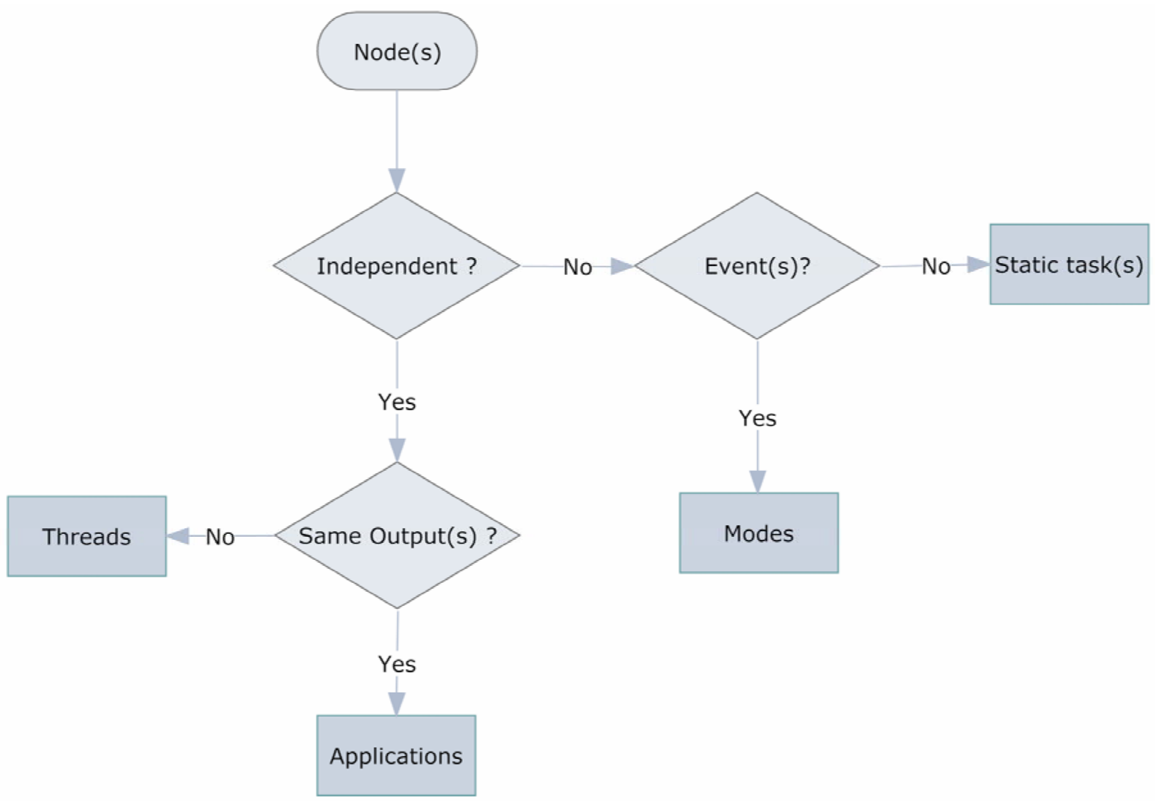


Figure **5-17**: The decision tree for bubbles in a multi-task and multi-mode workload

By applying the above findings, one constructs the hierarchical tree diagram resulting from the leveled context diagram. The tree diagram is called the *leveled tool diagram* that visualizes how the context diagram arrives at the level of tasks by assigning the configuration events. Figure **5-18** illustrates how the configuration events are assigned based on the example of Figure **5-14** . To identify the role of nodes, the bubbles are named after the components in the system model (*Ap* for application, *Th* for thread, *Ts* for task and *Md* for mode).

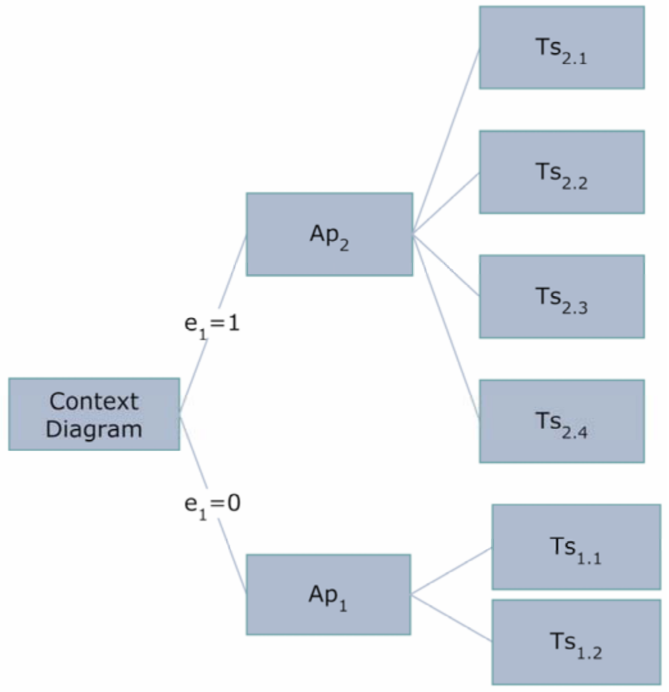


Figure **5-18**: An example of leveled tool diagram with applications

In this example, there are two applications that are associated with the configuration event, e_1 . The applications are assigned by the value of configuration event, where there is no subsequent application or thread level available.

On the other hand, Figure **5-19** illustrates how the configuration events are assigned to threads and modes of tasks based on the example of Figure **5-15**.

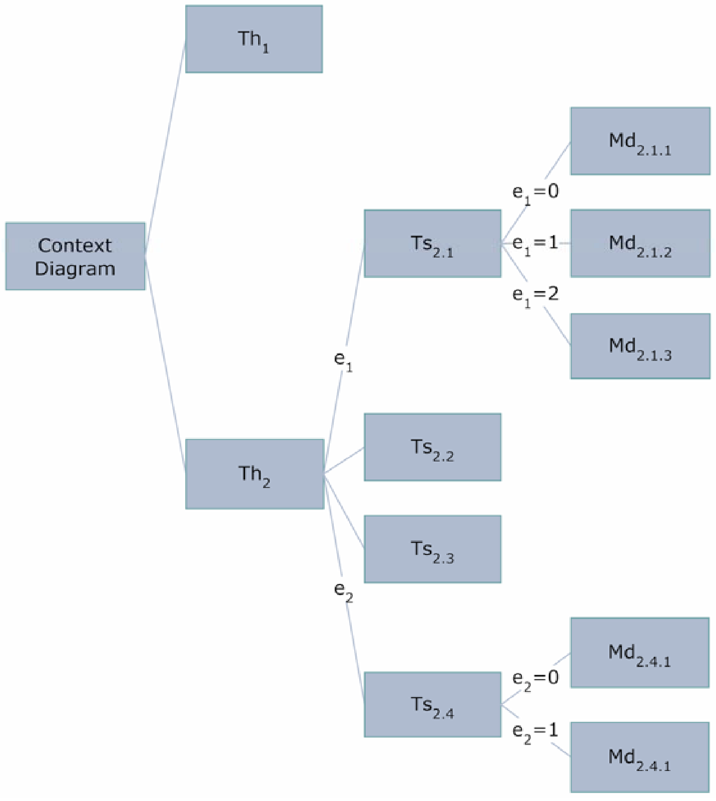


Figure **5-19**: An example of leveled tool diagram with threads

Because tools in the LEVEL 1 are not assigned with any event and there are subsequent levels available, the tools are referred to be threads. Especially for Th_2 there is a pending configuration event which initiates the tools where they depend on the value of the previous event. While Th_1 does not have any configuration event, the subsequent tools referred as tasks use configuration event to invoke the modes for the task, $Ts_{2.1}$ and $Ts_{2.4}$. Hence, Th_1 is considered as a static set of task(s) according to the decision diagram of Figure **5-17**.

As one has seen from the examples, the leveled tool diagrams clearly illustrate the relationships of a parent node and children nodes and assign a value of the configuration event to a particular application (or a mode of the task). The main purpose of identifying a multi-task and multi-mode workload is to recognize the tasks that are reconfigurable. By associating the value of the configuration events to a particular task the methodology provides a systematic way to recognize the reconfiguration of the tasks. For example, if one assumes that the initial configuration of the reconfigurable system is given as $c_0 = \{Md_{2.1.1}, Md_{2.4.1}\}$ using Figure **5-19** and the configuration event for c_1 is given as $E_1 = \{e_1, e_2\} = \{2,1\}$, then the system configuration needs to be changed as $c_1 = \{Md_{2.1.3}, Md_{2.4.2}\}$.

Unlike the leveled context diagram, the leveled tool diagram exclusively describes the relationship of the system events with functional nodes. Each value of the configuration event is assigned with either an application or a mode of the task. One assumes that each node selects a range of possible components from the library that is compliant with its functional description. While the event list traverses down the leveled tool diagram, it establishes a set of tools that belongs to the path which the values of the events possess. The inclusion of tools for the traversed path is decided by sufficient conditions that are given by the values of the events. If the events are the external signals that attribute to the characteristics of a very large function, there should also be independent tools that are not influenced by system events beyond its level, so-called *isolated tools*. The isolated tool(s) are the subfunction(s) that do not produce any other variation of functionality depending on the configuration events that exist below the level of the tool(s). All isolated tools should be identified with the associating configuration event. The upper level events should also be assigned to initiate the isolated tools in the lower level. Figure **5-20**.

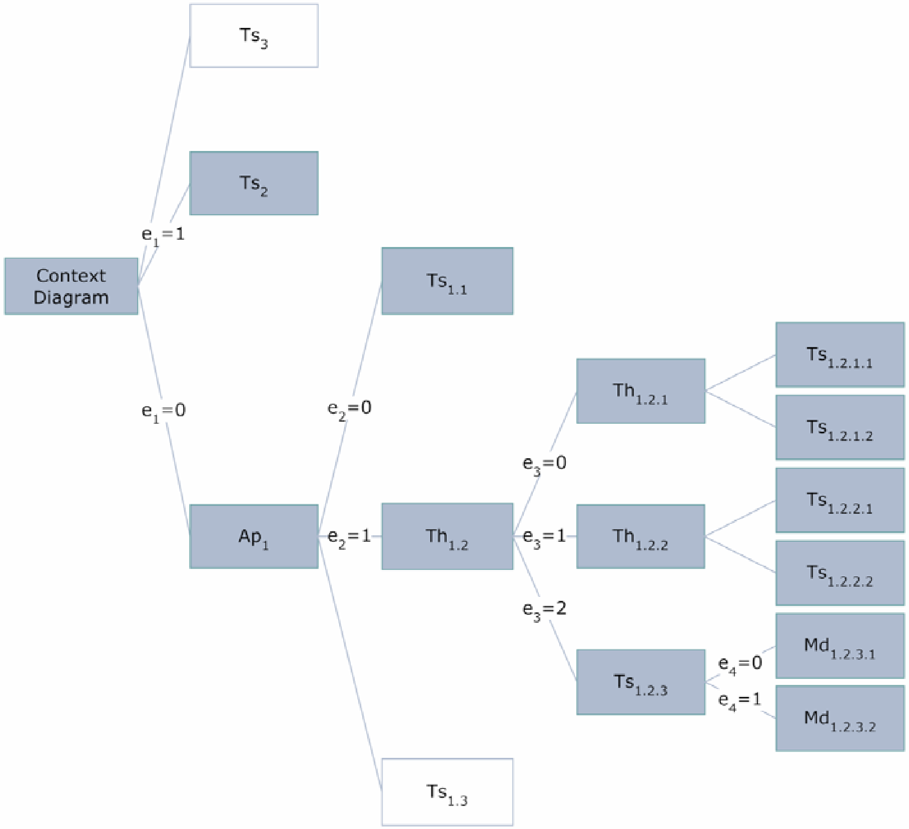


Figure **5-20**: An example of leveled tool diagram with isolated tools

In Figure **5-20** the event associated tools are indicated by grey boxes and the isolated tools by white boxes. The isolated tools as shown become the basis of the static architecture. The system $M = \{\Sigma, C, \delta\}$ holds the event list, $\Sigma = \{e_1, e_2, e_3, e_4\}$. The values of e_1, e_2, e_3 and e_4 are conditioned to select the tools. At the LEVEL 0, depending on the value of the events, e_1 only one of the functions (*Ap_1* or *Ts_2*) can be initiated. Since *Ts_3* does not have any associated configuration event, *Ts_3* is an isolated tool that is present at all times. While the functionality of Ts_2 does not change depending on the system event, Ap_1 can have $Ts_{1.1}$ or $Th_{1.2}$ depending on the value of e_2. In the same sense, Ts_3 is on all the time while $Ts_{.1.3}$ does not get turned on unless the value of system event, e_1 satisfies Ap_1. The leveled tool diagram shows a convenient way to associate the event list with the nodes in the system. The outcome of the leveled tool diagram is the functional node that appears as the leaves of the leveled tool diagram. They must be either tasks or modes. Each leaf of the leveled tool diagram associates with the system event(s) and their list tends to increase when the leaves are in the lower level. For example, $Ts_2 \to \{e_1 \mid e_1 = 1\}$ is in the LEVEL 1 and $Md_{1.2.3.1} \to \{e_1, e_2, e_3, e_4 \mid e_1 = 0, e_2 = 1, e_3 = 2, e_4 = 0\}$ is in the LEVEL 4. The *defining a multi-task and multi-mode workload* step results in the labeled functional nodes that are associated with the values of configuration events.

The reconfigurable system is expressed by $M = (\Sigma, C, \delta)$, where Σ are all configuration events, c_i are the configurations of the system and δ are the transitions required between configurations. The leveled tool diagram shows that the system can always be decomposed into task(s), $T = \{Ts_1, Ts_2, ..., Ts_i\}$ and/or the mode(s) of tasks, $M = \{Md_1, Md_2, ..., Md_j\}$. On these tasks and modes there are links, $L = \{Lk_1, Lk_2, ..., Lk_k\}$ that need to be populated in order to ensure the communication between them and with the external systems. To simplify the expression the task is assumed to be the mode of its own. Then a task can be normalized as the mode $Ts_r = Md_r$ simplifying the expression of the system as the group of the configurations $c_i = (M', L')$ which consists of the subset of the modes and links, $M' \subseteq M$ and

$L' \subseteq L$ where M' invoked by the given values of the configuration events, is a part of the leveled tool diagram.

5.8 Constructing a Static Architecture

The leaves of the leveled context diagram show the interfaces that each node needs in order to carry the required functionality. Analyzing the workload from tasks up to the level of applications grants a *conceptual assembly* of the reconfigurable system. The *conceptual assembly* shown in Figure **5-21** and Figure **5-22** can hypothetically outline the area that the workload occupies. The *conceptual assembly* of the static architecture provides a better way to reflect the constraints that might lie within the geometrical placements of components and their interfaces. In order to establish the conceptual assembly, one suggests the topological arrangement used in [**90**]

The topological arrangement provides a way to estimate the physical outcome of the workload mapped onto a chip. The topological arrangement treats all modules of the static architecture as topological points without considering their real physical dimensions. Instead it establishes a relative topological relations that is desired by various constraints (e.g., timing, routability, etc.) among the modules in the static architecture

There is a particular set of hardware characteristics that is unique in our design methodology. One of them is the coarse-grain size of our blocks. Because of the top-down approach of structural analysis [**89**] and multi-task and multi-mode oriented workloads, the system tends to deploy large blocks, so-called macro blocks. The other is how they are represented inside the floorplan process. Since the reconfigurable devices are thought to possess homogenous structure of configurable logics, the blocks can be expressed as a 2-D coordinate in a FPGA.

Based on the above characteristics, one assumes that the reconfigurable device floorplan problem can be given as[28]:

- A set of *n* rectangular blocks $B = \{b_1, b_2, ..., b_i, ..., b_n\}$ that resembles the state of a multi-task and multi-mode workload where $b_i \in B$ represent coarse-grain block. Each block should have;
  i. ω_i, h_i: width and height of b_i, which represent the number of configurable units (e.g., CLBs of FPGA) involved in the macro block;
  ii. a_i: area of b_i (i.e.,, $a_i = \omega_i \times h_i$), a_i is a constant representing total area of a macro block;
- A set of nets $N = \{n_1, n_2, ..., n_k\}$ that resembles the links listed for the state;
- Timing constraints and size of a chip;
- Each block is assigned to a location, xy-coordinate on the chip;
- Overlapping is guided between multiple configurations that have the same functional blocks.

The topological arrangement is obtained in a greedy fashion by adding one block at a time to the partial floorplan (point placements). One assumes that by using topological arrangement, a *conceptual assembly* of the reconfigurable system can be obtained.

[28] Adapted from [**91**]

The multi-task and multi-mode workload is reflected directly into the architecture of the reconfigurable system as the results of the configuration events which invoke different implementations of the system. In order for the reconfigurable system to be cost-effective, the system needs to associate the given values of the configuration events with a particular implementation. In other words, the system should be capable of recognizing the difference between implementations of c_i and c_{i+1} by looking at the given configuration events E_i and E_{i+1}.

The difference can be built around the static architecture. The static architecture is the skeleton of multi-task and multi-mode workload that does not change during the course of reconfiguration. One of purposes of the static architecture is to maintain the links among the non-reconfigured nodes for seamless computation. For instance, if the mode of a task needs to be changed, the reconfiguration should not touch any other part of the system except the area designated for the task. Because the static architecture keeps the essential interfaces between components, the rest of the system can operate without disruption by reconfiguration. Constructing a static architecture begins with assuming a reconfiguration granularity that is determined by the given configuration events.

If the event, e_1 in Figure **5-20** associates with the timing allowance that is smaller than maximum area that needs to be reconfigured by the nodes in the LEVEL 1, the static architecture should look like Figure **5-21**.

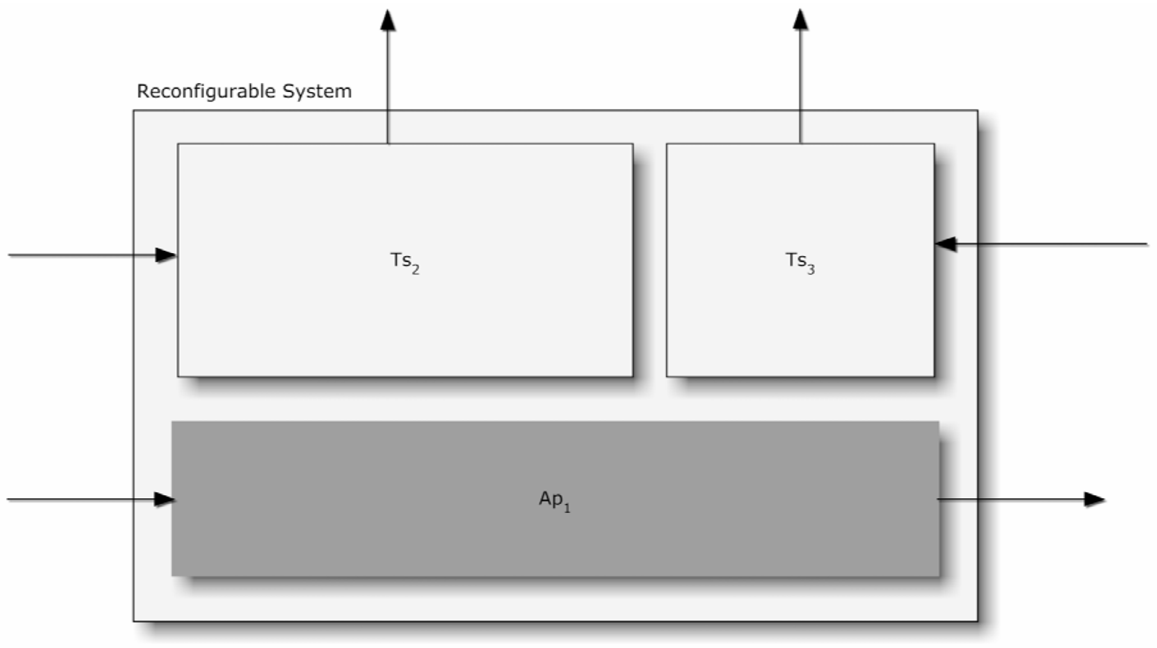


Figure **5-21**: An example of the static architecture

Because the reconfiguration system does not allow the workload to implement *Ap_1*, *Ts_2* and *Ts_3* as the reconfigurable modules of the system – they are too big to be handled by reconfiguration, they allocate the hardware space to be implemented in parallel and indicate the Input/Output connections that each module requires. One of these modules needs to be turned on by the value of e_1 when others are off.

Within the application, *Ap_1* there are many modes of operations requiring only small amount of functional changes. Because it is desired to keep other functions operational the static architecture that maintains the links between components needs to be identified. If there is a pre-existing condition (e.g., e_1=0), the static architecture can be defined further by outlining the links within *Ap_1* in addition to the existing architecture given in Figure **5-21**.

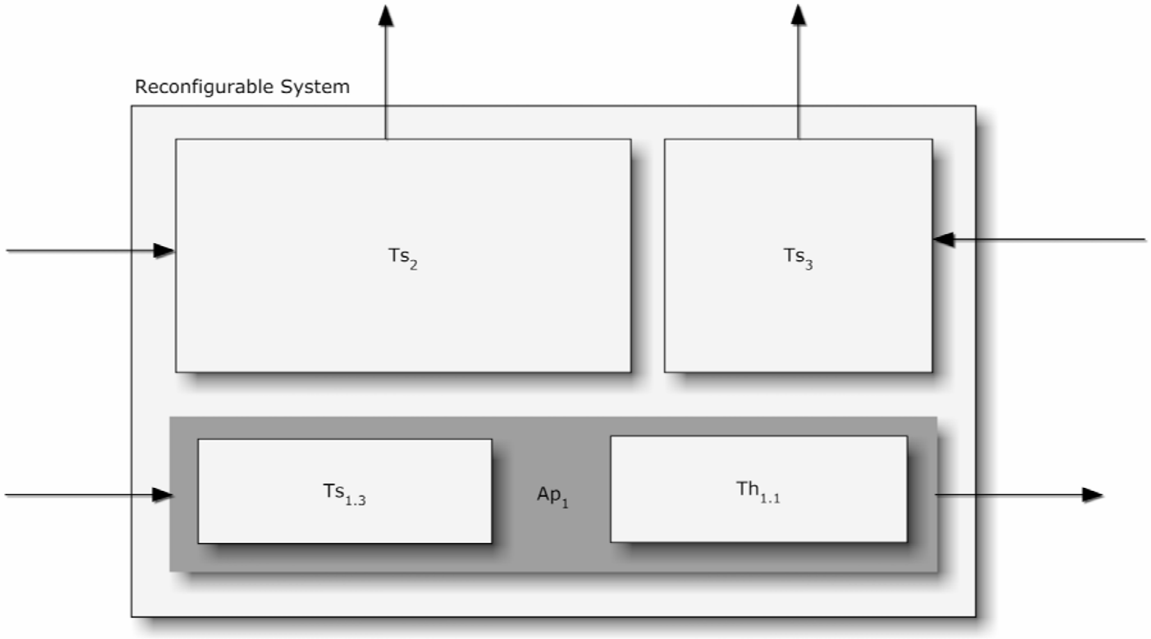


Figure **5-22**: An example of the static architecture with the pre-condition of e_1=0

As long as the configuration events with which the previously loaded architecture associates contains the pre-conditioned value(s), the same static architecture can be utilized again. In other words, the static architecture needs to be loaded in at the beginning of each of the big changes that is identified by *pre-condition(s)*. However, the pre-conditions exclude the system from obtaining other functionalities that the different values of pre-conditioned events can provide. Hence, to gain access to other functionalities the system needs to load a different static architecture.

The pre-conditioned static architecture can be extracted by finding the intersection of all configurations where the configurations represent various functionalities with the pre-conditioned values of the configuration event(s). The extraction of the pre-conditioned static architecture starts with finding all variations of the architecture, where each variation is expressed as a group of a mode for the tasks, $c_i = (M', L')$ with the pre-condition. The static architecture of the reconfigurable system is $A_{static} = \bigcap c_i$ where all c_i are obtained with the same pre-condition. If there exists no pre-condition, the static architecture considers all configurations, $A_{static} = \bigcap c_i$ for all c_i. Then the static architecture reconfigures all functions except the isolated tools in the LEVEL 0. The isolated tool(s) in each LEVEL becomes the skeleton of the static architecture which each configuration employs. With the same pre-condition subsequent configurations can utilize the same static architecture. For instance the following configurations given in Figure **5-23** can exist for three different occasions.

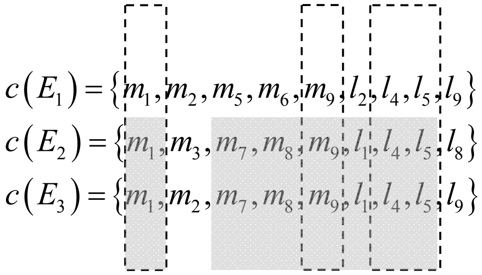

$c(E_1) = \{m_1, m_2, m_5, m_6, m_9, l_2, l_4, l_5, l_9\}$
$c(E_2) = \{m_1, m_3, m_7, m_8, m_9, l_1, l_4, l_5, l_8\}$
$c(E_3) = \{m_1, m_2, m_7, m_8, m_9, l_1, l_4, l_5, l_9\}$

Figure **5-23**: An example of static architecture

With these configurations, the dotted area indicates the static architecture for all configurations. However, if the pre-condition exists between $c(E_2)$ and $c(E_3)$, the grey area becomes what the system reloads as the static architecture. The larger those static architecture become, the smaller reconfiguration the configurations require. However, it is not possible to anticipate what configuration of the system would be required in near future where such anticipations result in minimizing the reconfiguration time and maximizing the static architecture between configurations. Importance of static architecture is the integrity of links that the static architecture guarantees to be active while reconfiguring. The static architecture should be bounded by the reconfiguration granularity that configuration events and the size reconfigurable area predict.

Nonetheless, the static architecture provides the steady links that remain active while the system is reconfigured. Because of the pre-conditions in the static architecture, the system can accommodate the links and keep reconfigurable functions that are smaller than the reconfiguration granularity allows.

5.9 Defining the Reconfiguration Granularity

The reconfiguration granularity is where the system draws the line for the reconfiguration. To figure where the limitation of reconfiguration starts, one studies the leveled tool diagram where every scenario of reconfiguration appears. The leveled tool diagram shows all possible modes of the reconfigurable system depending on the values of the configuration events. Since the reconfiguration only affects the components that are associated with configuration events, one starts from the lowest level of the leveled tool diagram that associates with a configuration event shown as the STEP 1 of Figure **5-24**.

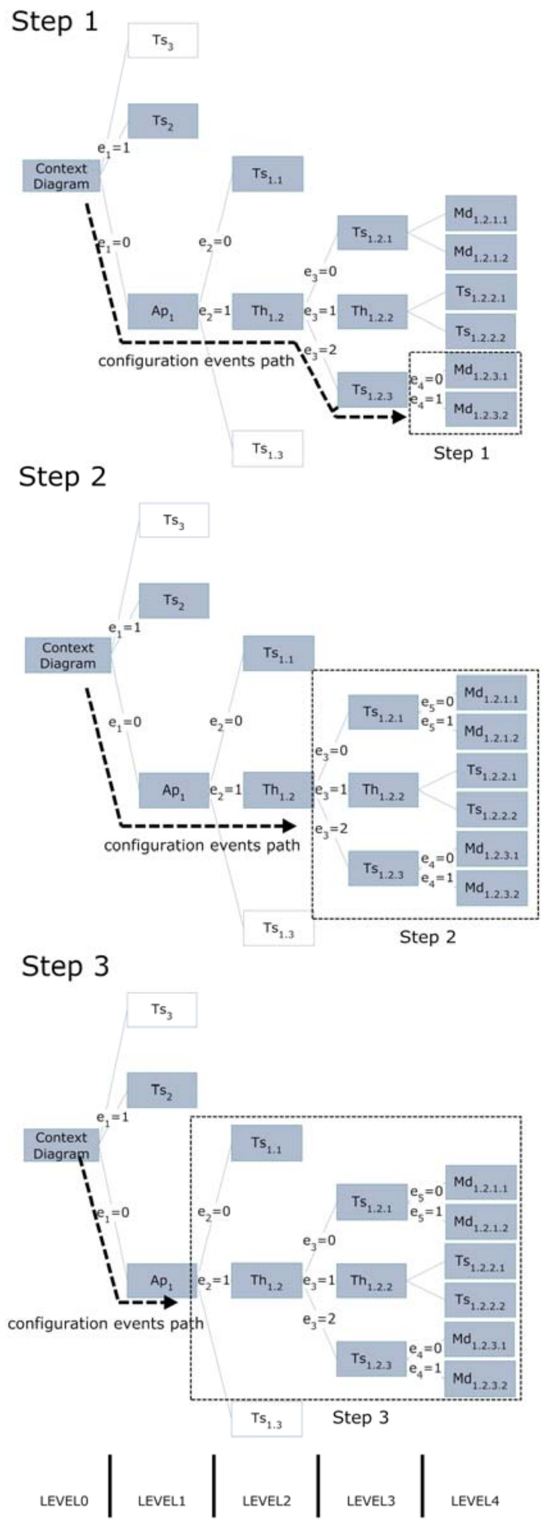


Figure **5-24**: The steps to achieve reconfiguration granularity

The lowest level of the leveled tool diagram is always populated with modes of tasks. If the maximum area that each mode of the task occupies $Area_{Ts} = \lfloor area(Md_i) \rfloor + \lfloor area(Lk_j) \rfloor$ does not exceed the reconfiguration time that is imposed by the shortest deadline, $\lceil d(e_k) \rceil$ where k = 1…n, of the *stream* event given as in Eq. **5-6**,

$$f_{T_{cf}}(Area_{Ts}) \leq \lceil d(e_k) \rceil \qquad \textbf{5-6}$$

where $f_{T_{cf}}(Area_{Ts})$ indicate the reconfiguration time that takes to reconfigure $Area_{Ts}$.

Then the modes can be designated to be *reconfigurable.* For example, in the STEP 1 of Figure **5-24** the maximum area occupied by $Md_{1.2.3.1}$ and $Md_{1.2.3.2}$ does not exceed the area that e_4 allows to reconfigure.

As the design progresses into the lower level, the event list starts to grow. The STEP 2 will consider the additional event e_3 onto e_4. If there are other branches that did not calculate the maximum area $\lfloor area(Md_i) \rfloor + \lfloor area(Lk_j) \rfloor$, then repeat the same evaluation as given in Eq. **5-6**. For example, $Ts_{1.2.1}$ needs to account for the maximum area occupied by $Md_{1.2.1.1}$ and $Md_{1.2.1.2}$ with e_5 before proceeding to the STEP 2.

In general, application, *Ap* can inherit the maximum area from its children without associating to further events. But thread, *Th* needs to add the areas as given in Eq. **5-7** .

$$Area_{Th} = \sum_{i=1}^{l} area(Md_i) + \sum_{j=1}^{m} area(Lk_j) \qquad \textbf{5-7}$$

Overall, the area occupied by the lower level components tends to get larger because they consider more mode(s) to find the maximum area. At the same time because the number of associated configuration events decreases (e.g., k = 1, 2,..., n-2 for the STEP 2 in Figure **5-24**), there is a greater chance to have a longer deadline that can accommodate greater reconfiguration area.

If $f_{T_{cf}}\left(Area_{Ts}\right) > \left\lceil d\left(e_k\right)\right\rceil$ as shown, at the stage 3 of Figure **5-24**, the reconfigurable system can perform the reconfiguration for the LEVEL 3 and beyond. The rest of the functionality should be implemented in parallel.

To estimate the areas for the tasks and the links between them, there are two assumptions taking place. One is that implementation of links is physically independent with functional nodes. The reconfiguration of links does not require change(s) in the functional nodes. Thus, reconfiguration of links does not hinder the functionality of nodes. The second assumption is the availability of $area\left(Md_i\right)$ and $area\left(Lk_i\right)$. While $area\left(Md_i\right)$ is readily available from the given parametric search in the component library, $area\left(Lk_i\right)$ can be tricky to estimate. One assumes that the estimation of $area\left(Lk_i\right)$ can be obtained by the *conceptual assembly* diagram that provides the sketch of a full system with a particular static architecture. However, because the *conceptual assembly* diagram is only estimation of the system, it might take several iterations to get the precise values.

Figure **5-25** illustrates the flowchart diagram to determine the reconfiguration granularity.

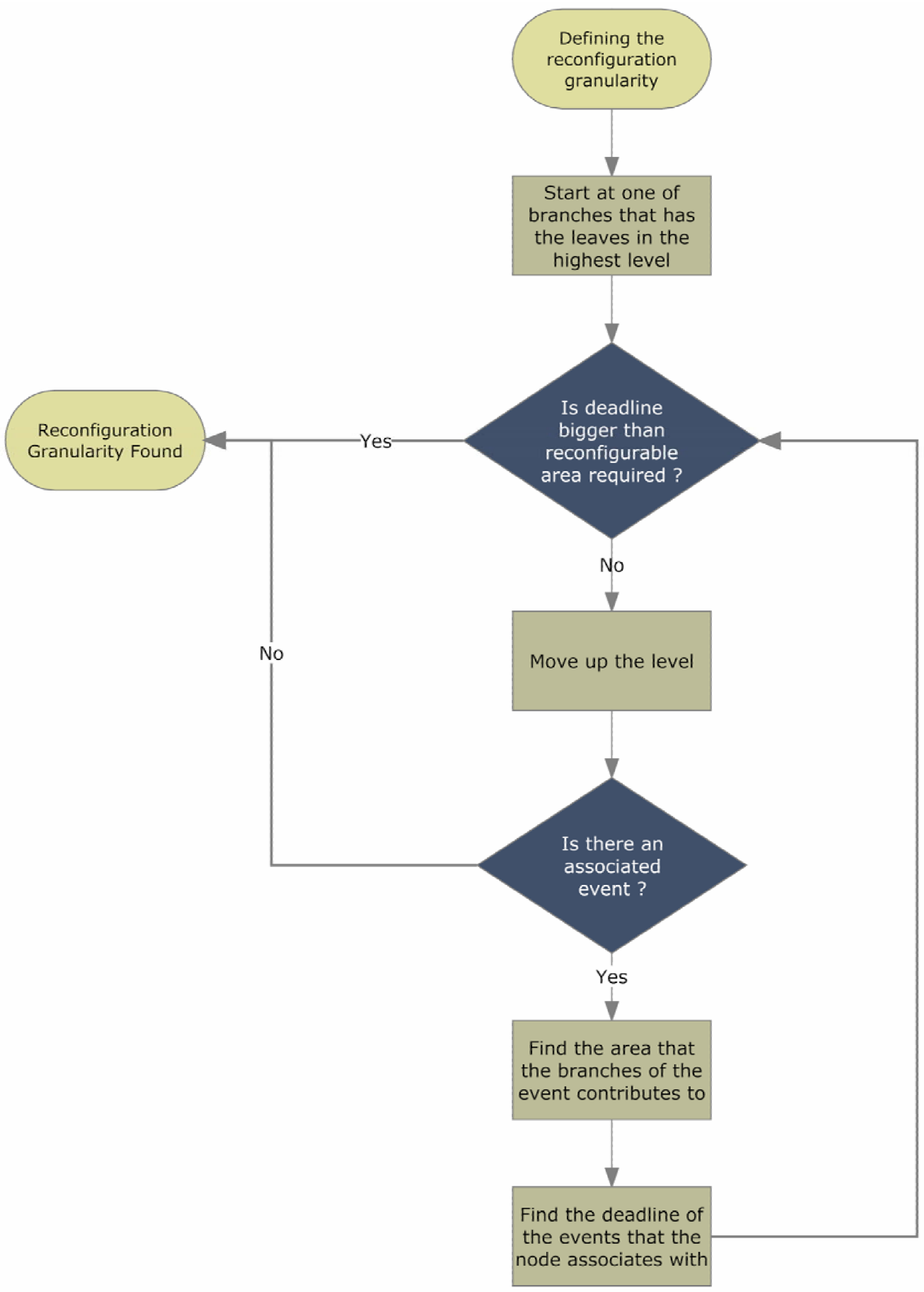

Defining the reconfiguration granularity
Start at one of branches that has the leaves in the highest level
Is deadline bigger than reconfigurable area required ?
Yes
Reconfiguration Granularity Found
No
Move up the level
No
Is there an associated event ?
Yes
Find the area that the branches of the event contributes to
Find the deadline of the events that the node associates with

Figure **5-25**: The flowchart to determine the reconfiguration granularity

In short, the configuration events that attribute the characteristics of the system below the reconfiguration granularity (no matter what value it assigns with) can take place by means of reconfiguration because its demand of reconfiguration is smaller than its timing allowance. On the other hand, the configuration events that attribute the characteristics of the system beyond the reconfiguration granularity should be implemented in hardware.

5.10 Summary

Our goal to achieve a cost-effective reconfigurable design solution can be realized by implementing the multiple functionalities in a reconfigurable device by reusing processing resources. In order to do so, one needs to analyze the multi-task and multi-mode workload associated with the configuration events and find the static architecture for each level of the context diagram and decide a suitable reconfiguration granularity that results in an optimal static architecture. If the system reconfigures a part of the system within the timing allowance of the configuration events and other parts of the system can still maintain their operations without halting, the system can lower the cost of silicon spaces used by multiple functionalities

Chapter 6

Implementation

Our goal to achieve cost-effective reconfigurable system architecture can be realized by reusing processing resources to implement multiple functionalities in a reconfigurable device. The theoretical investigation of how to recognize and organize multiple functionalities in reconfigurable systems was conducted in Chapter 4 and 5. During the investigation, the detailed steps of architecture synthesis methodology for identifying a multi-task and multi-mode workload, constructing a static architecture and defining the reconfiguration granularity were proposed and developed. To demonstrate the feasibility of the proposed methodology an experimental implementation is created to show the benefits of the methodologies and is used to estimate the cost effectiveness of the reconfigurable system. In the implementation:

- The feasibility of on-chip self-assembly via run-time reconfiguration is demonstrated;
- The procedures for on-chip assembly is developed and implemented;
- The framework of run-time reconfigurable system based on static architecture is created;
- The dynamic reconfiguration of various tasks (i.e., VHCs) driven by configuration events is developed;
- Run-time partial reconfigurable implementation of on-chip multi-visual stream processors is demonstrated.

Because none of the commercially available platforms is capable of handling a multi-task and multi-mode workload effectively, a reconfigurable platform based on a programmable device (i.e., Multi-task Adaptive Reconfigurable System Platform – MARS platform) is constructed. The general functionalities of the platform are to support run-time reconfiguration, incorporate Virtual Hardware Component (VHC) library and employ an operating system that analyzes the system events and loads configurations. Furthermore the MARS platform is equipped with the stereo camera module to demonstrate the effectiveness of the multi-task and multi-mode reconfigurable system design methodologies for real-time and stream applications.

6.1 System Organization

The system organization must reflect the general functionalities of the reconfigurable system where the system needs to:

- Detect configuration events;
- Assemble on-chip tasks;
- Enable run-time reconfiguration.

To fulfill the above system's objectives, there needs the consistent system level provision while the functions of a multi-task and multi-mode workload can dynamically change their performance and requirements.

The reconfigurable system is organized to have a set of hierarchical functions. The hierarchical organization of the system relies on the next lower layer to perform more primitive functions and to conceal the details of those functions. It also provides services to the next higher layer. The organization of layer hierarchy establishes the relationship between the layers so that the changes necessary in one layer function does not affect other layer functions. In the same sense, the partition of the system into layers brings much more manageable reconfigurable solution. The architecture of the reconfigurable system is divided into three levels of hierarchy:

- System level: the general functionalities of the reconfigurable system are handled;
- Micro level: the network of multiple functionalities are created and maintained depending on the values of configuration events;
- Component level: the functionalities of the tasks are conformed and details of functions are implemented;

The three-level hierarchy exists because of the independent purpose(s) that each layer serves. The system level provides the means of run-time reconfiguration. It should oversee the configuration events and needs to provide the necessary VHCs in time to re-organize the architecture of reconfigurable hardware as well as the functionality of individual components. The micro level is where the I/O of the reconfigurable hardware is distributed among various components and their inter-communication is implemented. Because of the unique requirements of communication on a chip based on a specific workload (e.g., frequency, delay, throughputs, placements of I/O ports and etc.), the design maintains its structure and verifies its performance as the dynamic requests of the system alters. The component level is where the functionality (or algorithm) of the VHCs is determined. Without being concerned about the complex functionality of the workload, it is where the smallest or fastest components are developed based on a parametric combination of the functionality (e.g., area, frequency, communication interface, protocol, algorithm, required resources and etc.). The following sections describe the details of each level that exits in the system organization.

6.1.1 System Level

The high level system architecture for the reconfigurable computing platform provides the structure of the reconfiguration service hardware for the architecture-to-workload adaptation (i.e., full/partial or static/run-time configuration). In order to take advantage of the architecture-to-workload adaptation and run-time reconfiguration, it needs to implement the following major components shown in Figure **6-1**.

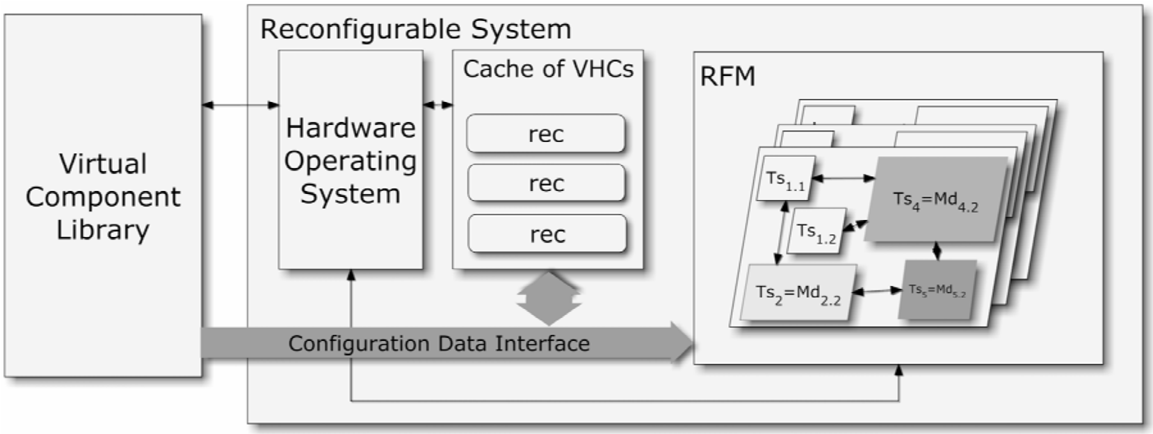


Figure **6-1**: System level architecture

Because run-time reconfiguration of a multi-task and multi-mode workload requires the system level provision, an operating system needs to be implemented to directly control configuration ports of the programmable device and to access the VHCs in memory. Because the library should consist of all components that the reconfigurable system can have, the volume of VHCs will always exceed the capacity of on-board (or on-chip) memory that enables fast reconfiguration (e.g., 32-bit @ 100MHz). Thus, a hierarchical memory structure is deployed to swiftly access the configuration files and load the VHCs that are only necessary for the currently workload:

- Reconfigurable Functional Module (RFM): a run-time and partially reconfigurable device(s) (e.g., Xilinx Virtex FPGAs) that serves the functionality of the system as requested by the workload of the applications. This module can be a device(s) or a part of device. The size of the RFM decides the maximum allowable workload of the system;
- Cache for re-configurable components: temporary memory space (e.g., external flash, SRAM, SDRAM or internal BRAM[29]) for configuration bit streams for the different modes of reconfigurable components. Each component in cache must associate with a reconfigurable area in the RFM;
- Virtual Component Library (VCL): static memory space for all tasks and all modes of tasks. This memory space consists of all configuration bit streams of all task modes and all static architectures;
- Hardware Operating System (HOS): is a part of reconfigurable system that decides or performs the following kernels:
    a. Configuration bit stream upload(s) from VCL to cache;
    b. Monitoring the configuration events;
    c. (re)configuring the static architecture and virtual components of the RFM as necessary;

[29] Internal RAM available for Xilinx FPGAs.

There is normally a short time window (e.g., from hundreds of nanoseconds to tens of milliseconds) available that all reconfigurations should fit in. Depending on the situation and requirements, the HOS needs to look for appropriate components, manipulate configuration stream and re-organize the cache structure. The HOS using Instruction Set Architecture (ISA) would normally not be able to execute the required operations in time. The HOS needs to be realized in “hardware” to accomplish these tasks in the given time window (e.g., 10s μs to 10s ms). The operation system is referred as an Hardware Operation System (HOS) to reflect the aspects of the hardware realization.

6.1.1.1 Reconfigurable Functional Module

Reconfigurable Functional Module (RFM) is where the platform dependant parameters for the whole system can be specified. Therefore, the parametric selection for each component of the workload can be performed by system architect (or possibly by operating system). Due to the consistency of the internal structure of FPGA devices all virtual components can remain to be compliant for all FPGAs in the same family. The processing unit of the Reconfigurable Functional Module (RFM) is a partially reconfigurable FPGA. Hence, the configuration control signals need to be designated to access the internal configuration circuitry at run-time. The proper control of the designated signals grants the access to write or read on-chip configuration SRAM. The explanation of the Virtex-4 FPGA system architecture is given in the following section and the overview of configuration interface is laid out in section **6.1.4.1**.

6.43.36.2 Run-time Reconfigurable Platform (Xilinx Virtex FPGA)

There are several companies who manufacture Field Programmable Gate Arrays (FPGA): Altera Corporation [**56**] Lattice semi-conductor Corporation [**54**], Cypress Incorporated [**92**] Xilinx Inc . [**55**], Atmel Corporation [**93**], Actel Corporation [**53**] and QuickLogic Corporation [**94**]. While Xilinx and Altera still own the biggest share of FPGA market, other players are renowned in a niche market of FPGAs. For example, Cypress is famous for mixed-signal FPGAs. Actel is famous for One Time Programmable (OTP) FPGAs that are popular in space applications. They can be also divided by configuration technology. Xilinx, Altera, Atmel, Lattice semi-conductor and Cypress offer volatile SRAM-based FPGAs while Actel and QuickLogic mainly focus on antifuse and non-volatile flash-based FPGAs. The distinctions between companies become thinner as they explore different processing and configuration technologies to benefit wider customer attractions.

Among them, there are a few companies who provide run-time partial reconfigurable FPGAs: Xilinx and Atmel provide SRAM-based reconfigurable FPGAs for reusable hardware and flexible operations. The FPGAs from both companies aim for providing the architecture and the internal organization of the resources can reflect the current requirements of data processing. However, due to the differences in the logic density and system speed, Atmel can not provide the possibility of implementing a complex multi-task and multi-mode workload that Xilinx FPGAs can offer. Additionally Xilinx put many efforts to develop the support tools for partial reconfiguration.

Depending on how the run-time and partially reconfigurable FPGAs is adapted, the system can reach a nearly perfect architectural point whenever computational requirement changes. In the following sections one will explore how the internal architecture of the device helps the system to dynamically adapt for various computational requirements from the system organization point of view.

6.43.37 Micro Level

Computer architecture is the structure of interactions between various components to produce intended outputs. Since the outcome of micro-level architecture provides the organization of interconnected components, properly arranged micro-level components can create optimal computer architecture. The different interactions among the different components decide the overall system performance (i.e., speed, throughput, latency and hardware usage). Therefore, it is our intention to provide the architecture synthesis methodology for creating static interconnections and for choosing adaptable micro-level components. The unique combination of micro-level components and partial reconfiguration provides adaptable computer architecture that is dynamic for a specific workload when it is necessary.

6.43.37.1 Abstraction of micro-level implementation

The modular design is a methodology of system design that combines physical area constraints into hardware description level to allow the partition of large system into manageable sub-functions that are logically and physically separable. This enables the creation of the reusable components that can reside in memory. In the context of the modular design, the Virtual Hardware Component (VHC) is a uniformly shaped reusable component that can fit in any part of homogeneous programmable devices. The only limitation of the VHC is that it must have the module size that is the multiples of a certain number (e.g., an island of 16-row columns for Xilinx Virtex-4 and 5 families). This limitation originates from the organization of routing architecture in the Xilinx Virtex-4 FPGA devices that GLOBAL lines (i.e., vertical HCLK routing resources) are not allocated with homogeneous architecture but they are grouped by 16-row columns of CLBs.

The entire structure should be defined in a top level design with all reconfigurable modules instantiated as a “black box”. As in any system design using HDL tools, the VHC needs to be synthesized, mapped and placed & routed. The implementations specified by VHCs are described in HDL which enables flexible implementation. However the design process for each VHC does not employ the locking of pin locations as a normal HDL tool would do. But rather it uses the pseudo port(s) to terminate the connections. The description of hardware is translated into netlist that is passed into physical implementation stage with the area constraints. When the both ends of connections are joined through the interconnection called the Virtual Bus (VB), the computer architecture that is assembled by micro-level components reveals data processor(s) for a specific application.

6.43.37.2 Architecture of Xilinx Virtex-4 FPGA

Each Virtex FPGA device consists of Configurable Logic Blocks (CLB), Input/Output Blocks (IOB), Block RAMs (BRAM), First-In First-Out buffers (FIFO), clock resources, local/global routing, configuration SRAM, configuration controller and a number of DSP blocks and a few to several PowerPC cores depending on the family of FPGAs. As it is shown in Figure **6-2** the Virtex FPGA contains the resources such as Arrays of CLBs, Arrays of IOBs, SRAM memory blocks – so-called BRAM, DSP blocks, Clock resources (DCMs, etc.) and Routing resources (global, long and local routing). Figure **6-2** illustrates the organization of these resources in a device (i.e.,Virtex-4 LX Family). While the BRAM, FIFO, DSP and clock resources have its own routing to interconnect with the rest of the device, the CLBs need to communicate via a general routing matrix. The switch matrix comprises an array of routing switches located at the intersections of horizontal and vertical routing channels.

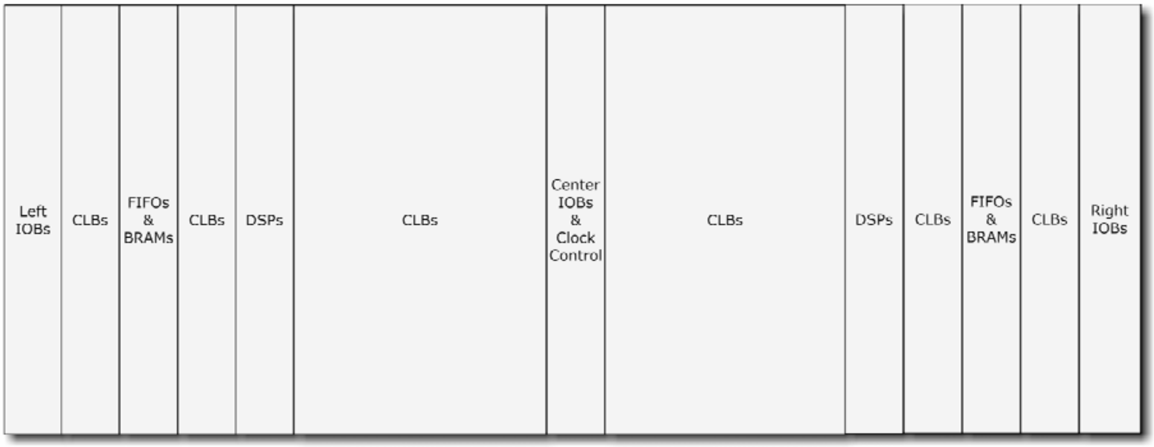


Figure **6-2**: Virtex-4 LX Architecture Overview

Due to the homogeneous structure of the CLBs and its routings, it is possible to implement swappable components. However, there is a need to have a hierarchical routing structure to interconnect components across the physical boundaries. One of them is called long lines in Xilinx Virtex-4 FPGAs as shown in Figure **6-5**. As noted in Figure **6-2** there are the interleaved FIFO, BRAM and DSP blocks along with CLBs. These specialized hardware cores prevent the relocation of hardware components in the FPGA. If hardware components only consist of CLBs, it can be relocated anywhere in the FPGA where CLBs are present. However, if hardware components consist of specialized cores other than CLBs, it can only be moved along the vertical lines of the same hardware structure that hardware component is developed with.

The routing architecture of Xilinx Virtex-4 provides a segmented, hierarchical routing structure that connects to the heterogeneous fabric of elements through a General Switch Matrix (GSM) block. The General Switch Matrix (GSM) is the switching fabric that allows connection to input(s)/output(s) of CLBs as well as to other GSMs. The routing resources that are programmable are physically located in horizontal and vertical routing channels between each GSM. The routing resources available from GSM are shown in Figure **6-3**.

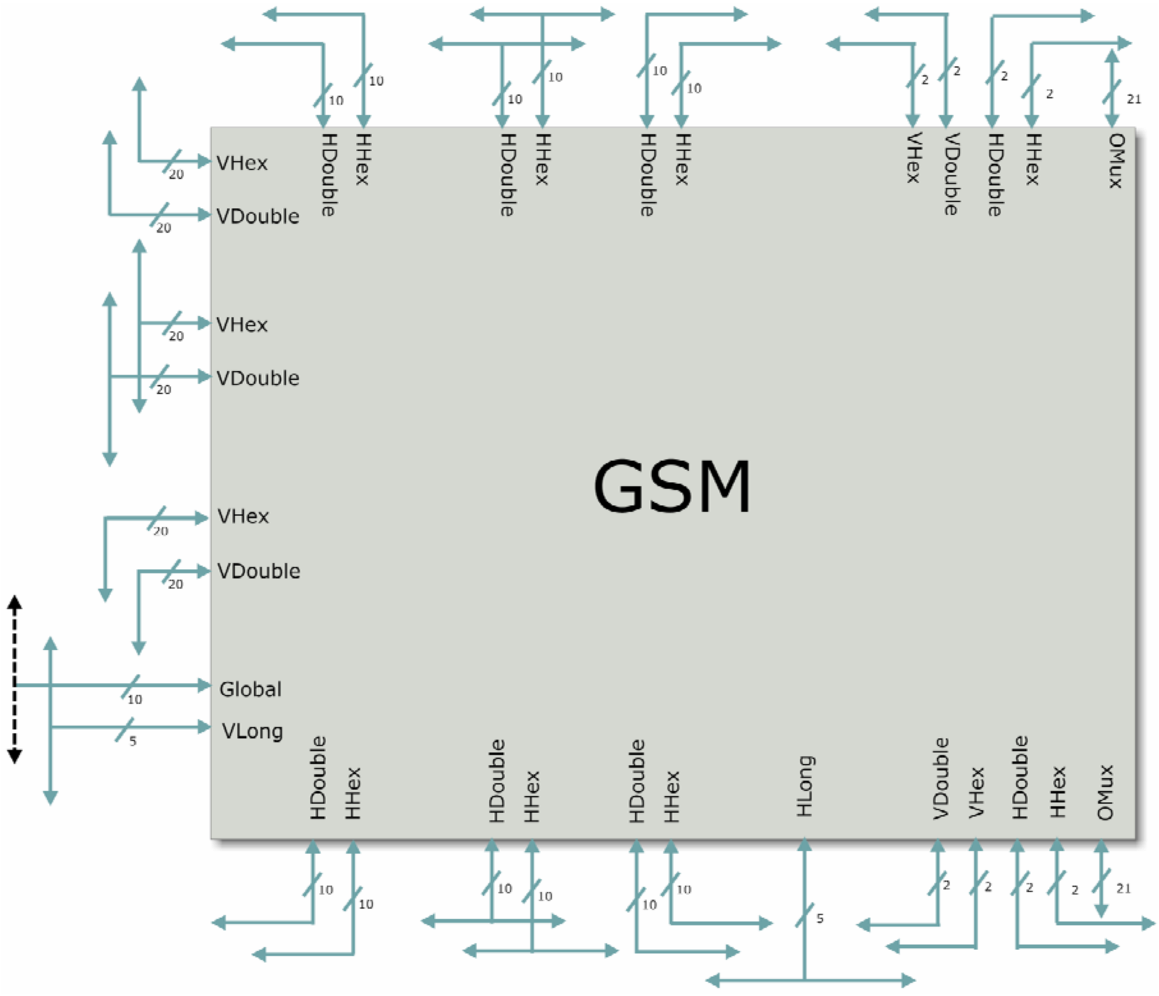


Figure **6-3**: The overview of GSM interconnects

The majority of lines that the GSM has are hex and double lines. Hex lines connect 3 GSMs that are 4 GSMs apart. Double lines connect 3 GSMS that are adjacent to one another. Depending on the affix (e.g., H or V) given for the lines, it indicates whether the lines spans vertical by V or horizontally by H. Additionally if the lines have the arrow pointing upward, it indicates that the GSM is the last one at the bottom in the group. The same can be applied for left and right as shown in Figure **6-4**

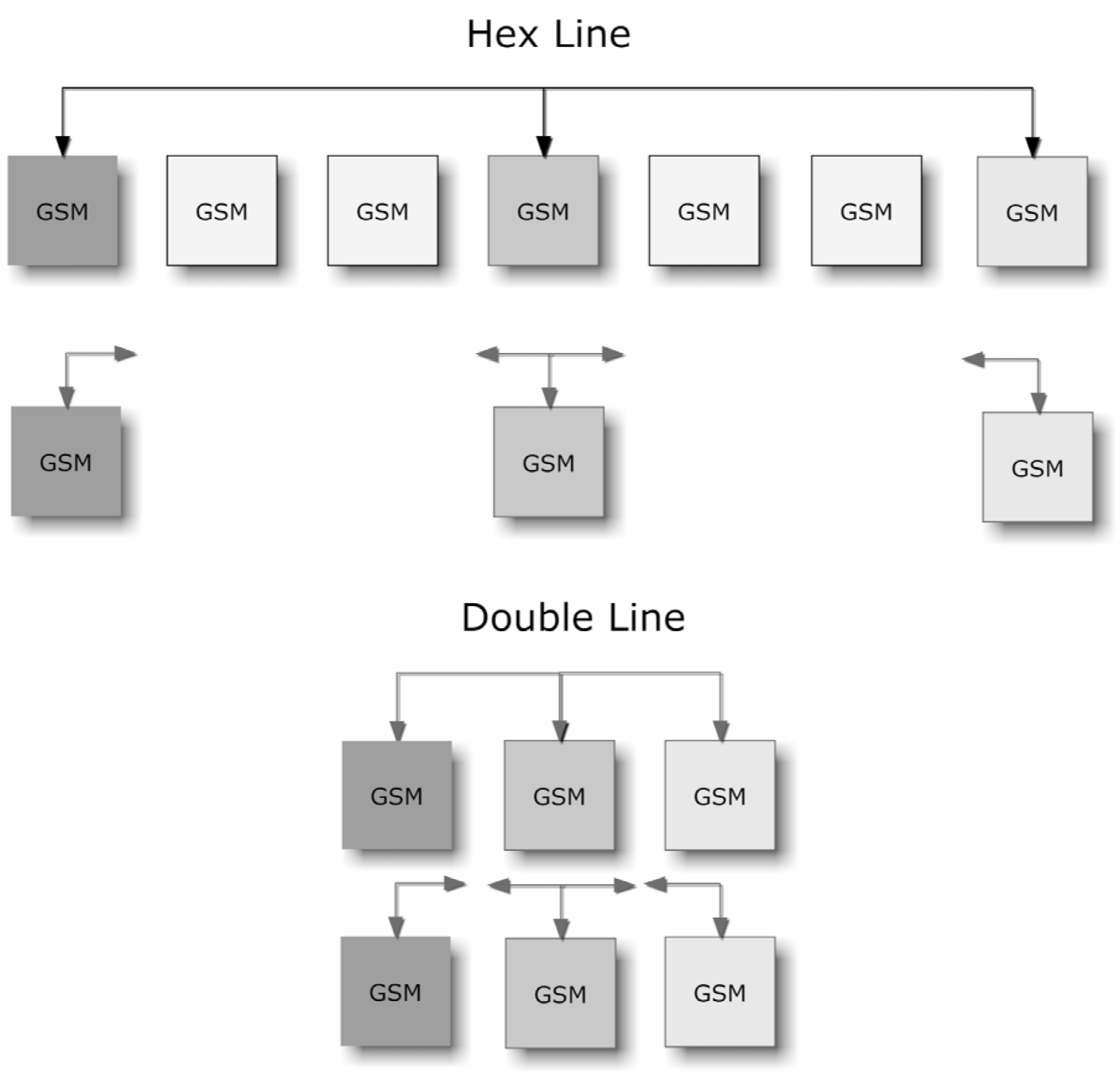
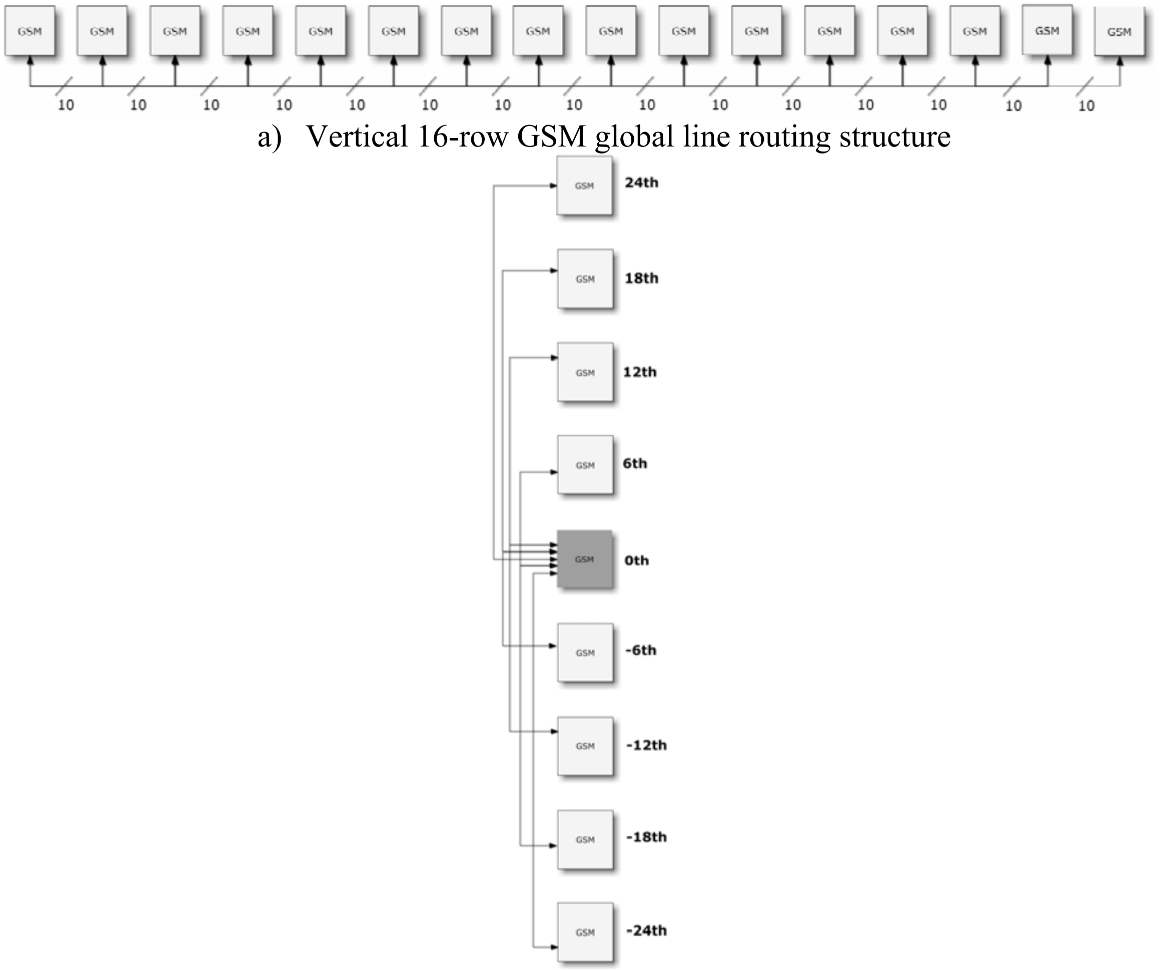


Figure **6-4**: Hex and Double line routing structure

The GSM also consists of all lines shown in Figure **6-5**. Due to the segmented routing structure of global lines, the global lines can go either or both direction(s) depending on where in the row organization of FPGA the GSM belongs to as shown in a) of Figure **6-5**.



a) Vertical 16-row GSM global line routing structure

b) Vertical and horizontal 5-port and 6-hopping long lines

Figure **6-5**: Long & global line routing structure

To summarize the routing resources in GSM, they are listed as:

- 10 long lines that spans 48 height (rows) and width (columns) of the device;
- 10 global lines that connect groups of 16 GSMs;
- 128 hex lines that route to every fourth or eighth block away in all four directions;

- 128 double lines that route to every first or second block away in all four directions;
- 42 direct connect routes that route to all immediate neighbors.

Among these routings, the global lines are used as clock signals and double, hex and direct lines are considered to be local lines. However, the long line routings can be used as the framework of static architecture that interconnects far-distanced components and I/Os ports. In order to assist this purpose, CLBs are allocated to form the ports of the single/multi-port line(s). The slice-based BUS macro is a term used to refer the construction of multi-port bus using the long or local line routings. An example of BUS macro is given in Figure **6-6**.

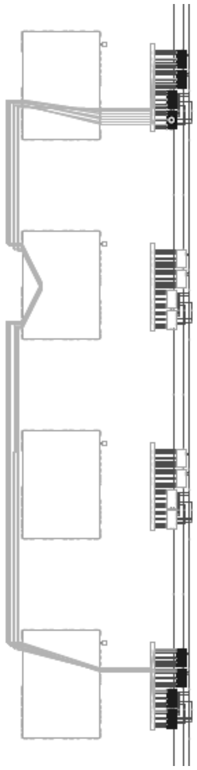


Figure **6-6**: An example of sliced BUS macro

The sliced BUS macro given in Figure **6-6** is deployed to enable communication between adjacent components that requires 8-bit of communication from bottom to top with enable signal using local double lines. Yet, for non-adjacent components or I/O ports, the ad-hoc long lines need to be used to create a specific micro-network.

Overall, the analysis of routing structure is important to realize how static architecture can be realized. Static architecture is a part of micro-network that is assumed to be separated from functional implementation of the system as stated in the previous chapters. But, there is no commercially available chip that allows this separation. The routing structure of FPGA chips is interleaved within the logical ones. Moreover, the routings do not have any meaningful role in CAD tools as they are expressed as ideal wires. As it is emphasized before, the DSM effects should be accounted to ensure successful construction of modular systems. The proposed solution is to use the long lines for communication between components that are not adjacent and I/O ports that are hard to reach by local routings. Because Xilinx FPGA allows "glitchless" reconfiguration[30], the separation of micro network and components can be achieved. When the reconfiguration of components is requested, the pre-existing micro-network is inserted on to the reconfiguring component(s). Hence, there is no change(s) written for micro-network.

[30] "glitchless" configuration means that if the same configuration information is written over the region then, the region carries on its operation without knowing that it is reconfigured. Particularly, this type of operations can be applicable for static routing structure.

6.43.38 Component Level

The configuration memory of FPGAs can be considered to be a rectangular array of bits organized in 16-row columns. The bits are grouped into a vertical line that is one-bit wide and is called a “frame”. Here are the characteristics of frames:

- Frames provide the configuration array (bit-wise) of 16-row columns;
- The size of the frames is fixed to be 41 words that is same as 1312 bits;
- The frames are the smallest unit of configuration that can be written into or read from.

The frames are grouped together into a larger unit. In the Virtex FPGAs, there can be several different units. However, in the context of the swappable (or re-locatable) components, the 16-row columns that consist of CLB units can only be considered. The swappable components that consist of the number of these columns create functional units that are reconfigurable and are interconnected via the bus lines.

FPGA vendors produce the chips that guarantee the maximal operations of the core elements limited by connectivity of clock resources. In the Xilinx FPGA chips, this limitation is apparent by 16-row wide global lines. The Xilinx FPGA chips divide the clock regions imposed by the limitation of 16-row global lines. Therefore, the 16-row is a physical limitation of reconfiguration granularity that is constrained by the routing structure as well as clock reach-ability. One assumes that usage of local routings within 16-row granularity guarantees its timing closure.

6.43.38.1 Virtual Hardware Component

The Virtual Hardware Components (VHCs) are the reconfigurable hardware modules that are assigned to use multiple sections of homogenous structure in the FPGA. VHCs are flexible in terms of their sizes, I/O width and algorithms to be implemented.

The reconfiguration of FPGAs is achieved through loading the configuration data into the appropriately addressed configuration RAM. The configuration data controls logic functions of Look-Up-Table (LUT) and interconnections among logic, I/O, clock and memory resources. The organization of uniformly distributed columns allows the insertion of hardware components independent with placement within the units of 16-row column. The multiple unit(s) of the configuration data file that defines the units of homogenous structure in the FPGA can be represented as a Virtual Hardware Component (VHC). Each VHC consists of a processing element that embeds required functionality and Virtual I/O that connects with physical BUS macro that is pre-routed. The width of communication link is determined at the initial stage of the design. Thus, unless the whole architecture with the different link width is reloaded, the width of communication link remains a static part of the system. However, the processing element of the VHC can be reconfigured at run-time by utilizing the addressable uniform hardware resources.

The implementation of partial reconfiguration in the Xilinx Virtex FPGA has the certain limitation that the addressable unit of configuration is given by the 16-row column-organized structure.

6.43.38.2 Virtual Bus

While the VHCs are independently developed system modules, the virtual bus is an assembling medium to bridge these components together. Since the virtual bus is not bounded by area constraints, they can make signals to cross over partial areas that are occupied by VHCs. The Virtual Bus (VB) utilizes long (or local) lines so that internal communication among VHCs can be constructed. Unlike the VHCs that are created from the descriptions of hardware components the VBs are created in technology dependant hardware level in order to keep static hardware structure over the entire design process. The VBs are instantiated in the same way as the VHCs in the top level. The VBs are used as fixed data paths for inter-module communication as well as external IO communication.

A design method with direct hardware resource bounding is the best way to construct the VBs because they should guarantee fixed performance. One of the suitable tools that are available is Xilinx FPGA editor. The FPGA editor is a graphical application for displaying and configuring FPGAs. Since the structure of the VBs overlaps between different instances, one can use a fundamental design of a VB to create more complicated VB structures. A fundamental design of VB is called Macro BUS. Macro BUS has properties that combine HDL level instantiation with physical level hardware. Input/Output ports that are declared in HDL level are assigned to the hardware that has matched parameters for netlist integration.

6.43.39 Configuration

Because the reconfiguration system spares a considerable amount of the system for controlling and executing (re)configuration, it is important to look at the mechanical and timing aspects of configuration to evaluate its overhead.

6.1.4.1 Configuration interface

Systematic configuration process provides a versatile relationship to system controllability. It also makes the system configure as quickly as possible when each tick of clock relates with the system cost in a real-life situation. The parallel mode configuration for the FPGA is designed so that it satisfies the goal of minimizing cost and timing issues and is compatible with the interfaces of the available hardware – cache or VCL. The design aspect of parallel configuration is similar to the memory access control.

The Virtex family of Xilinx FPGAs uses the term SelectMAP for 8-bit, 16-bit and 32-bit parallel configuration modes. The SelectMAP utilizes a bi-directional access port for reading or writing configuration data of Virtex FPGAs. Several designated pins in the Xilinx Virtex FPGA are used for configuration. Among them a few pins are the dedicated control pins that need to be monitored or driven and are listed in Table **6-1**

Table **6-1**: Configuration Ports

Name	Direction	Description
CCLK	IN/OUT	Configuration Clock
PROGRAM	IN	Asynchronous Reset
DONE	IN/OUT	Configration Statue
CS	IN	Chip Select
WRITE	IN	Low(write)/High(read)
BUSY	OUT	Busy/Ready statue
INIT	IN/OUT	Configuration Error Indication
CDATA	IN/OUT	Configuration Data

The system level architecture using these configuration ports allows configuration of the RFM. The architecture attempts to minimize the reconfiguration overhead by increasing configuration speed. The system level components ensure that the bandwidth allocated by hierarchical structure of configuration memory always provides the maximum speed of (re)configuration.

6.43.39.2 Configuration chart

The external configuration process needs to follow the configuration chart illustrated in Figure **6-7**.

After successful completion of the configuration's initial sequences the configuration data is loaded into the device. Only at this stage the determination of partial or full configuration is achieved. At the end of loading, the pre-calculated CRC value is compared with internally generated CRC. Once the CRC values are matched, the startup sequence is initiated to invoke the operational state.

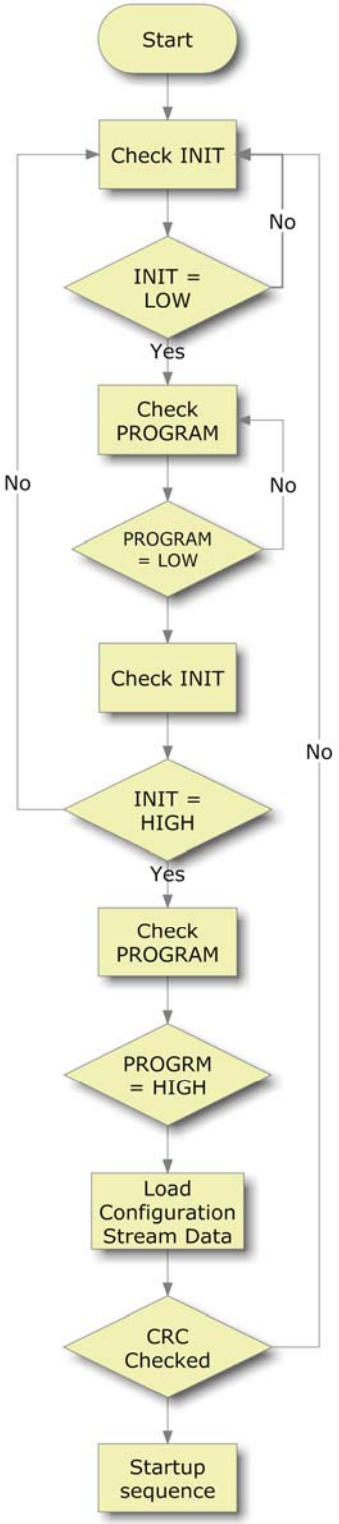


Figure **6-7**: Configuration Flow Chart

6.43.39.3 Configuration steps

The biggest difference between partial configuration and full configuration is in the bit file structure. The full configuration follows the flow chart that requires a specific bit file structure shown in Figure **6-8**.

First, the initialization is used to prepare the internal configuration circuitry for loading the configuration frames. Then, it loads the synchronization word to recognize the word boundaries of 32-bit. From this point, the CRC calculation circuitry is initialized by reset process. The frame length is loaded followed by asserting the command that holds the outputs of all CLBs to one (i.e., GHIGH_B command). Then the configuration options are loaded. After loading all data words, the configuration file sends out the last frame command. If the calculated CRC matches the given CRC, the device can run into operational mode.

The differences that partial reconfigurations make from full configurations are: it has no need for synchronization stage; it does not place GHIGH_B command to halt CLB operations; the starting address of the frame does not start from the first column, but starts from the address where the VHC resides. However, both files share the same overhead that requires for (re)configuration because it follows the same flow in Figure **6-8**. The size of overhead is 1312 bytes. The size of configuration stream depends on the size of the region that needs to be reconfigured and the overhead that needs to be passed on the configuration circuitry of FPGAs. Therefore, the smaller VHCs, the faster reconfiguration.

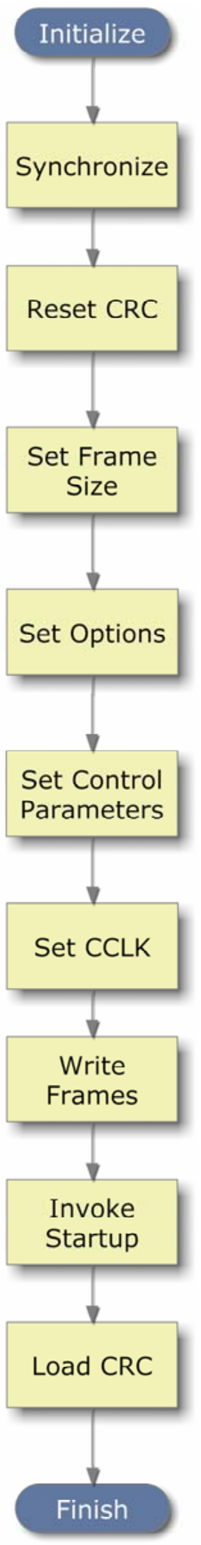


Figure **6-8**: Configuration Processing Flow

Each step of configuration flow can be considered as either the delay or the overhead of reconfiguration. When the delay of reconfiguration is assumed to be negligible, then the (re)configuration time can be estimated as Eq. **6-1**

$$T_{cf} = cf_{size} \times cf_{speed} \qquad \textbf{6-1}$$

where T_{cf} indicates the time that takes to (re)configure, cf_{size} represents the size of (re)configuration stream in byte, 2-byte or 4-byte and cf_{speed} indicates the configuration speed that the interface can achieve. The cf_{size} counts the overheads that are necessary to carry the information for configuration steps mentioned in section **6.43.39.3**. Depending on the interface setups – refer to section **6.1.4.1**, the byte size of cf_{size} should be changed.

Moreover, the time that is taken to reconfigure the FPGA also depends on the characteristics of reconfigurable units that Xilinx FPGA can handle. The Xilinx FPGA uses the physical reference points such as rows and columns to illustrate its reconfigurable units.

6.43.39.4 Reconfigurable Units

Figure **6-9** illustrates the column and row address allocation inside XC4VLX160. One row consists of 16 CLBs vertically. The row of 16-CLB is the smallest unit of reconfiguration. Since each row can be sized depending on the number of column(s) attached, the unit of reconfiguration (i.e., reconfiguration granularity) can be modified.

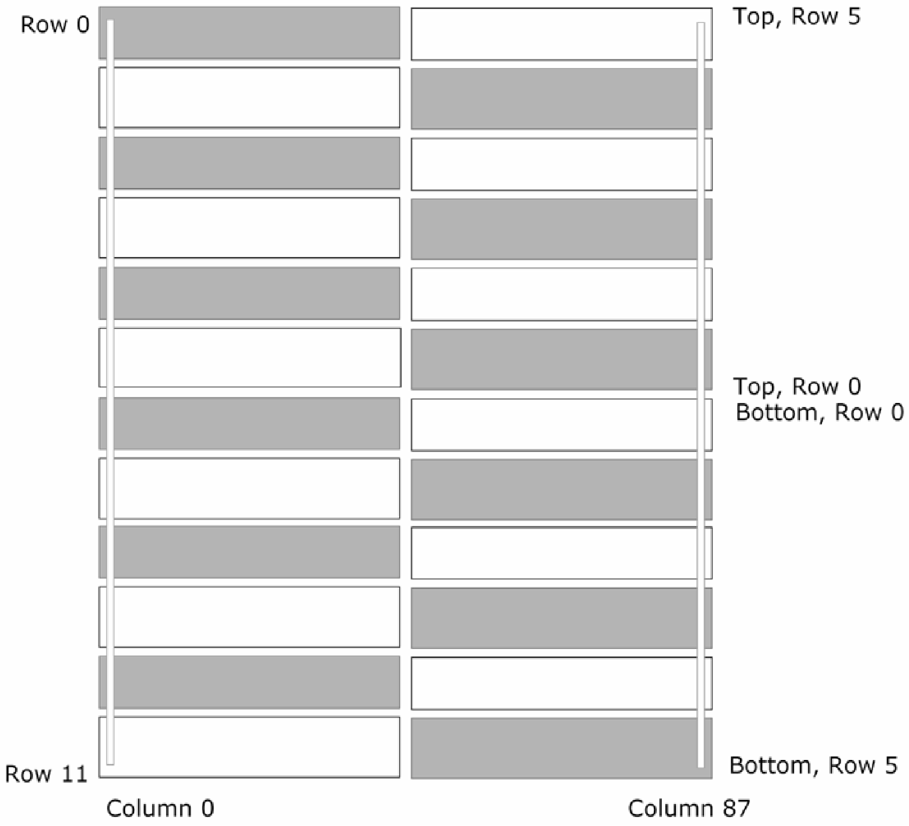


Figure **6-9**: Virtex-4 Family: Allocation of Frames

6.43.39.5 On-chip assembling of Virtual Hardware Components

There is a hierarchy in the system design that implements the VHCs as the parts of the system. From the HDL point of view, the system is the layered hierarchical functions. In the top level of the hierarchy, the VHCs are defined as the instantiated HDL entities. The defined VHCs declare the port directions and the size of ports. The top level design also attaches the specific placements that are associated with each module. The top level design creates a template of the interconnections between components and external I/Os as well as the placement for each module to finalize the "sketching" of system structure. Once the inclusion of module declaration (i.e., port mapping in HDL) is finished in the top level, the design can go on implementing details of applications for each module with inherited placements constraints. As long as the implementation is compliant with the system limitation and assigned area, active partial reconfiguration procedure provides placed and routed modules as well as a module-specific bit files. Once connections to the bus are established and there are no other unrouted lines within the placement, the design can be used to reconfigure the module. The example of the run-time partial reconfiguration is given in the following sections.

6.44 Run-time Reconfiguration Example

A simple example of partial data files is generated to give an easy look at the configuration file structure. The example shown in Figure **6-10** is constructed to use the partial reconfiguration procedure to reconfigure the different counters for the LED controller in the FPGA.

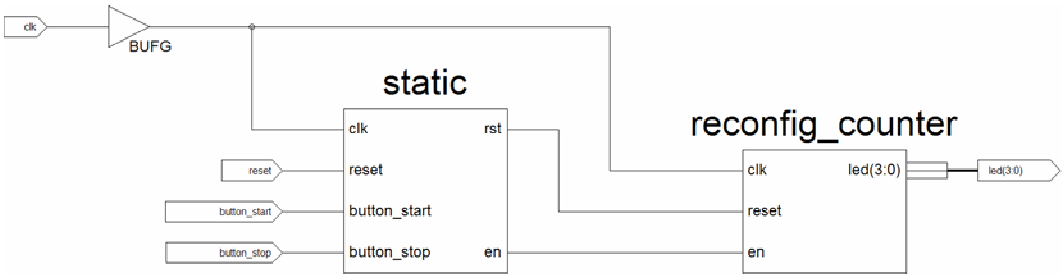


Figure **6-10**: Schematic of “LED counter” example

The system contains two major components in the circuit. The first component is a static component that does not get changed. The second component is a reconfigurable component that behaves as one of counters (e.g., upcounter, downcounter, right shifter and left shifter). The communication links between the static and reconfig_counter are inserted as the sliced BUS macros that create virtual ports around the boundary of the reconfigurable region. The details of Hardware Description Language (HDL) files involving these modules are introduced in Appendix B.

According to the structure of the system, the reconfig_counter should reside in the constrained area in the FPGA. In this example, the area constraint of reconfig_counter is given as the white space (e.g., from X112Y31 to X143Y0) in Figure **6-11** to accommodate the input pins (e.g., clk, button_start, button_stop, reset) that are located in the centre I/O block and output pins (e.g., LEDs) that are in the right.

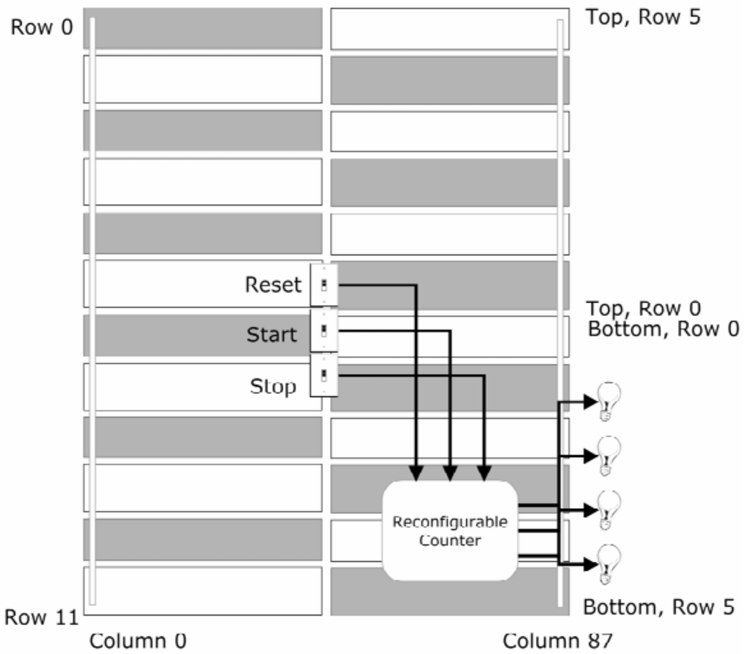


Figure **6-11**: Device XC4VLX160 Floorplan

Because the design is based on modular approach, the partial stream files for the reconfig_counter should consist of the same amount of configuration information. All the files contain the same command set that initializes the configuration circuitry without holding CLB outputs and the data contents that are necessary for parts to be modified.

In order to identify the constraint area of (re)configuration, one should study the configuration stream files, especially the content of Frame Address Register (FAR) in the file. The file consists of the FRAME address assignments (0x3000 2001) and frame address (0x0041 4F00) according to [**95**]. Table **6-2** shows the bit map of FAR that assigns the top/bottom bit, Block type, row, column and minor addresses for the data to be stored. For instance, the 0x0041 52D3 means that it belongs with CLB type of data and has the row and column address of 0b0010 = 5, 0b00111110 = 62 respectively and minor address of ob010011 = 19. One CLB column of Virtex-4 FPGA contains 22 frames and the size of each frame for XC4VLX160 is 1392 bits (41 words) – each word is 32 bits for Virtex FPGA family.

Table **6-2**: Frame Address Fields Register (FAR)

									Top/Bottom	Block Type			Row Address					Column Address								Minor Address					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Block Type	Codes
CLB/IO/CLK	000
RAM Interconnect	001
block RAM content	010
CFG_CLB	011
CFG_BRAM	100

The given frame address, 0x00414F00 in the configuration stream file for the example indicates that it is the first frame of the 5th row that is in the bottom half and 60th column from the left side of the chip according to Table **6-2**. It bears the data size of 0x3889. It indicates that the following data covers the area of 353 frames which is equivalent of 16 columns. The above findings conform with the area constraint inserted for the design from HDL assembling level referring to section **6.43.39.5**, where two increments of index indicate one CLB column or row.

Therefore, including the overhead (e.g., 500 bytes) cf_{size} becomes 14598. If the maximum configuration speed is achieved (e.g., cf_{speed} = 100MHz in Virtex-4 configuration architecture), it takes 145.98 μs to reconfigure the "reconfig_counter" that is one order smaller than 1.2 ms to configure a full system. Moreover, the stream files can be compressed further to reduce bits. For example, the "reconfig_counter" file can be compressed[31] to cf_{size} as 2558 that takes 25.58 μs to reconfigure. Furthermore, the reconfigurable system can reload (or reconfigure) difference between the currently loaded component with newly deployed one. For example, the system only needs to reload 34 frames to change the functionality of "reconfig_counter" from upcounter to left shift, which takes 13.94 μs to reconfigure compared to 145.98 μs.

The above mentioned techniques are the efforts to reduce the size of reconfigurable files (i.e., VHCs), therefore increasing the opportunity to reconfigure more components that fits into the timing allowance that each configuration event offers in the reconfigurable system. The time that takes to reconfigure, T_{cf} strictly depends on the size of column(s) involved, where T_{cf} determines the reconfiguration granularity of the VHCs.

[31] The compression ratio depends on the common data words that can be written into multiple slots of configuration RAM. If the data are different, the ratio of compression would not be high as shown in the example.

The above example illustrates the different ways to create cost-effective configuration files, how to obtain parametric information regarding VHCs and hardware details associated with Xilinx Virtex FPGAs for partial reconfiguration. In summary, the micro-level architecture provides a framework of on-chip assembly which is called static architecture. The static architecture provides the parametric information such as size, location and available resources.

6.45 Run-time partial reconfigurable implementation of on-chip multi-stream processors

The cores of real-time multi-stream video processors are implemented into the reconfigurable system to give a visual demonstration of partially reconfigurable processors in real-time using image capture and display devices. This implementation shows the effectiveness of the proposed reconfigurable system methodologies when it is deployed for stream data applications.

6.45.1 System Organization

The Reconfigurable Functional Module (RFM) of the reconfigurable system consists of two groups of cores: static and dynamic. To demonstrate the benefits of the stream processing system, the real-time video processors were implemented on the RFM. The static cores are the controllers that interface four major hardware components. The first component is a VGA interface board utilizing a 4-channel Digital-to-Analog Converters (DACs). The second component is a camera interface board that controls the camera setting and acquires the video data. The third component is a PC interface that enables the communication among task memory of PC, configuration cache and internal configuration of the RFM. The fourth component is the RFM where the dynamic and static cores reside.

The general organization of hardware in Figure **6-12** illustrates the roles of each component to fulfill the functions of the platform. The PC interface provides an access point to bridge between large pool of virtual components (i.e., VHC Library) and immediate configuration cache and run-time reconfigurable functional hardware (i.e., RFM). The Hardware Operating System (HOS) encapsulates the control of a few devices. It also works as an arbitration mechanism to deal with reconfiguration of the RFM. Ultimately the HOS needs to include hardware mechanism to access task memory or cache and analyze the situation to invoke reconfiguration of the RFM autonomously. The data processing of our implementations on the RFM is uniquely designed to give a highlight on partial reconfigurability of the platform based on the methodologies provided in Chapter 5.

The RFM transfers the data with the stream data devices (e.g., stereo camera). The camera is the input device to the RFM. Conversely, the 4-channel RGB 24-bit DACs are the output devices that display processed images to the multiple video display devices. The controllers of these devices are embedded as a part of static core. While the data acquisition processor (i.e., camera related cores) actively participates to store the video data onto the data memory, the VGA display processor takes the data from the memory to portrait the images into the display devices. The steps of the video processing takes:

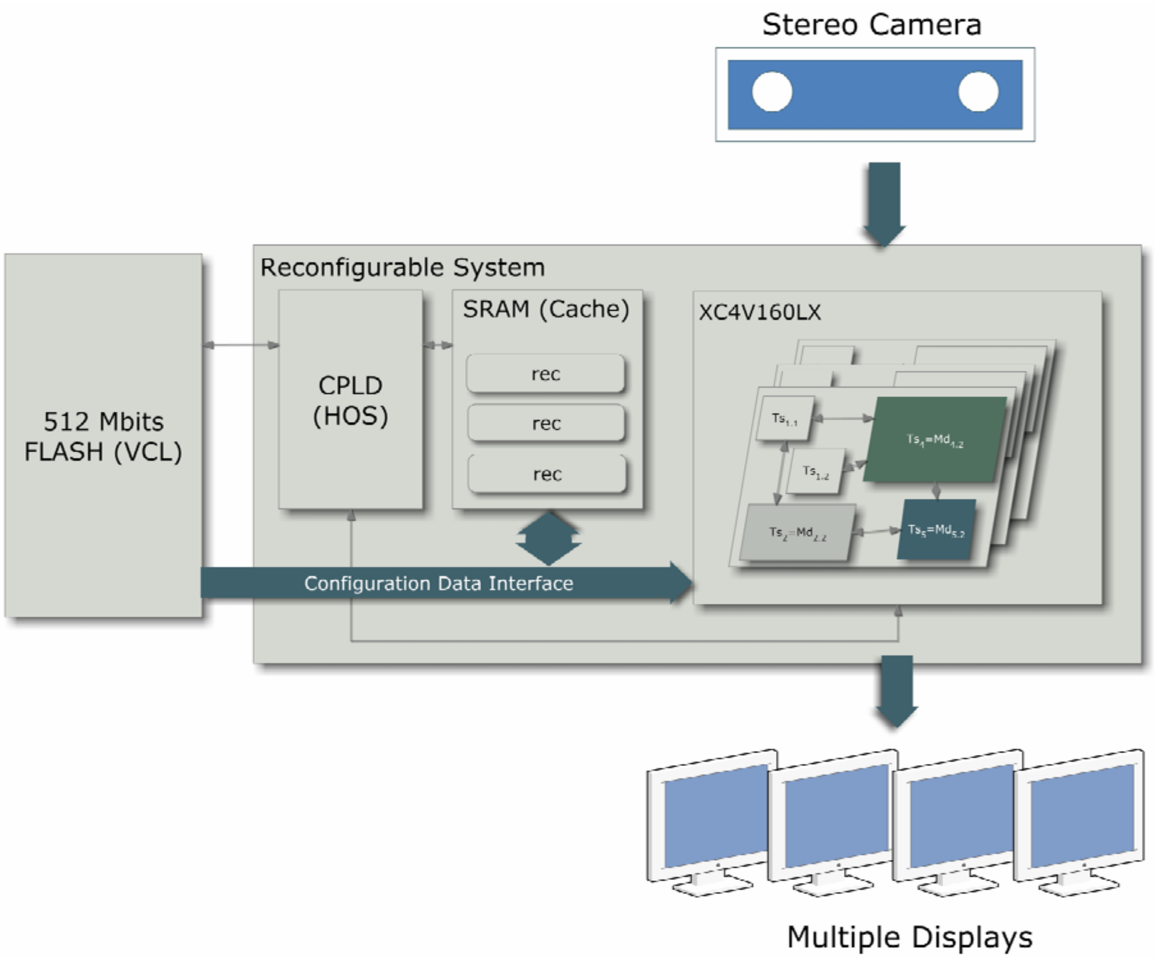


Figure **6-12**: Hardware Organization

1. The raw data is captured from the camera and then stored into the memory;
2. The raw video data is fetched out from the memory and fed into multiple video processing units;
3. Processed outputs of each unit are transferred to the VGA devices for displaying different processing results.

6.45.2 Multi-task Adaptive Reconfigurable System Platform

The Multi-task Adaptive Reconfigurable System (MARS) platform allows designers to explore various embedded reconfigurable system architectures where massive parallel computation is inevitable and is equipped with various configuration capabilities to determine their benefits depending on computation requirements. The platform consists of the followings:

- Xilinx Virtex-4 LX 160 FPGA with 4-lookup table 150K logic cells;
- Xilinx CPLD (i.e., Configuration Stream Loader) with 4 banks of 8-Mbyte FLASH memory with 64-bit parallel architecture for supporting fast FPGA configuration;
- Two banks of 72 Mega bits of Flow Through Synchronous Random Access Memory at 200MHz with 16-bit data interface;
- Two banks of 512 Mega Bytes Double Data Rate Synchronous Dynamic Random Access Memory at 166MHz with 32-bit data interface;
- 4 SVGA DB-15 connectors supported by Digital to Analog Converters (DAC) with 80 MHz conversion rate;
- 2 Gigabit serial transceivers;
- 4-bit 4-bank LVDS transceivers at 200 MHz data rate;
- 64-bit compliant VME BUS connection at 66MHz data rate;
- Microchip microcontroller to create USB/serial to CPLD/FPGA communication.

A block diagram of the MARS platform is shown in Figure **6-13**.

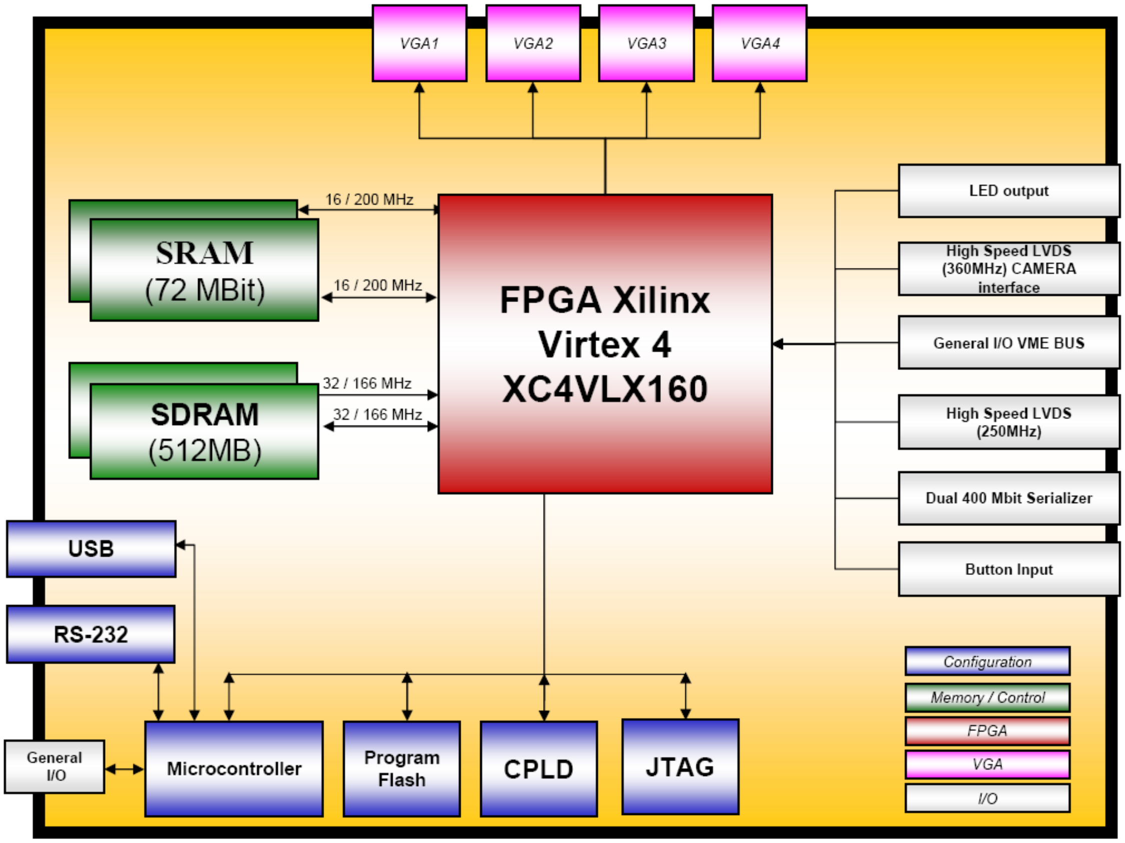


Figure **6-13**: Block diagram of MARS platform

The Xilinx Virtex-4 LX FPGA used in the platform is the result of Xilinx's Advanced Silicon Modular Block (ASMBL) initiatives where the families of FPGAs are manufactured with the target applications in mind. Because LX family provides more homogeneous logics than other families of FPGAs, the largest LX FPGA, XC4V160LX is chosen to be used at the time of manufacturing of the platform. The XC4V160LX contains 152064 logic cells, 67584 slices, 1056 kb distributed RAM, 96 XtremeDSP slices, 5184 Kb Block RAMs and 12 Digital Clock Managers. The picture of the MARS platform is given in Figure **6-14**.

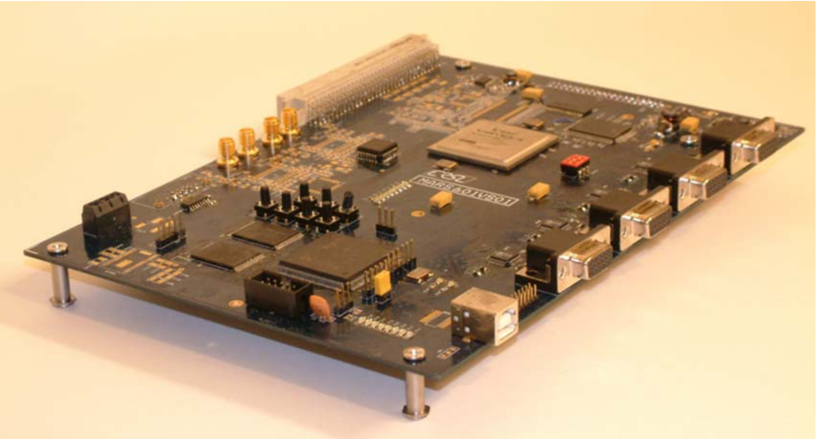


Figure **6-14**: MARS platform

Because of the VME BUS connector that is available in the platform, the FPGAs can be aggregated to process much more complex functionality than one FPGA can handle as shown in Figure **6-15**.

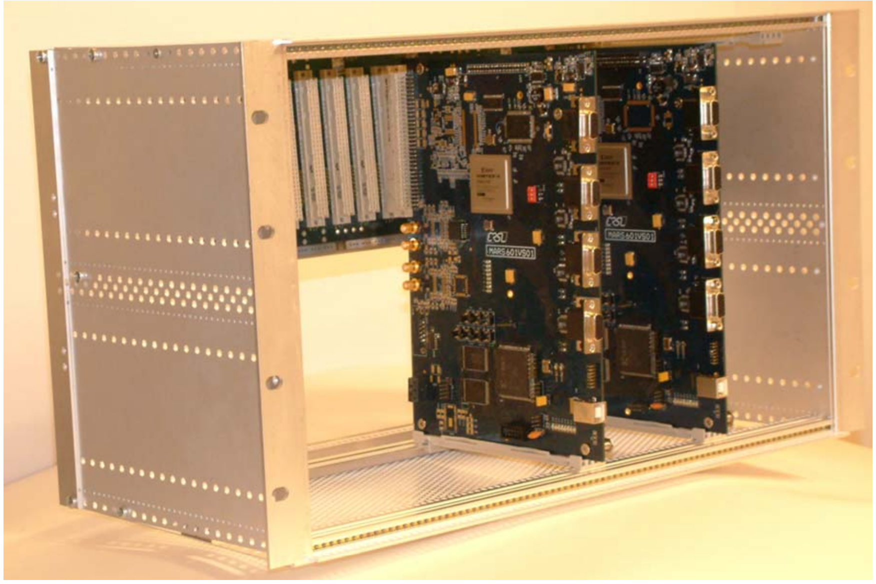


Figure **6-15**: An illustration of aggregated operation of MARS platforms

6.45.2.1 Stereo camera board

The environment of real-time and stream applications is well described by the data sequence of the visual capture module as shown in Figure **6-16**.

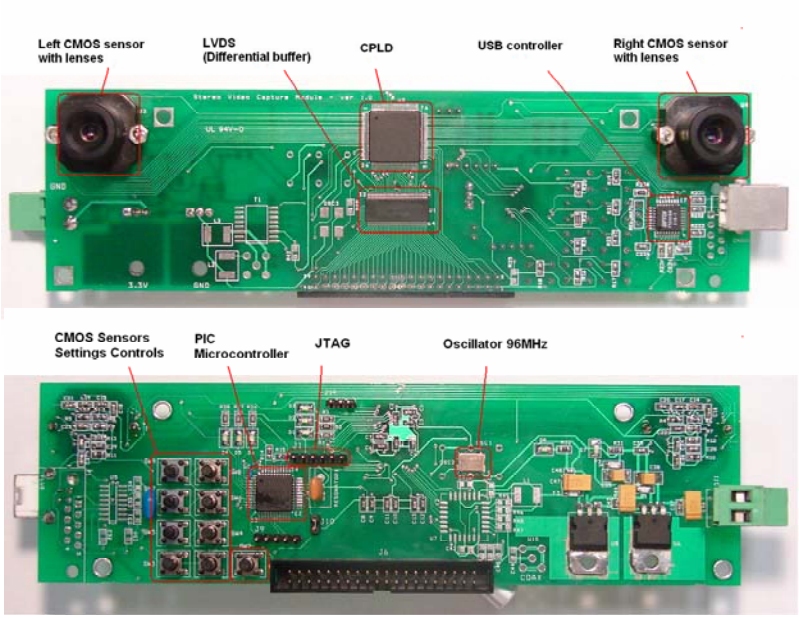

Left CMOS sensor with lenses
LVDS (Differential buffer)
CPLD
USB controller
Right CMOS sensor with lenses
CMOS Sensors Settings Controls
PIC Microcontroller
JTAG
Oscillator 96MHz

Figure **6-16**: Stereo Camera Module

Inspired by recent interests in stereo processing for distance calculation and object tracking of autonomous satellite tracking system, the visual capture module is equipped by two cameras that produce an effective stereo vision by processing the pixels. The Stereo Camera Module (SCM) generally needs to have a glue logic (or processor) that combines (or interleaves) the data. The micro-controller provides the camera setting control via USB interface connecting to PC executed GUI software. The SCM consists of:

- Two CMOS color cameras;

- Xilinx CPLD as data interleave and glue logic for microcontroller, LVDS and camera modules;
- LVDS transceiver at running at 96MHz;
- USB controller and microcontroller for camera setting control.

While the camera data processing with 30fps can be dealt with general processors, the processing of camera data exceeding 100fps becomes challenging depending on what functionality that image processing requires. Moreover, the nature of stereo vision processing requires two cameras work synchronously. When the cameras are in operation by its internal control, then the system can not be synchronized. Therefore, there is a need to control the cameras from the processor's point of view. To reserve the space for the camera controller, an FPGA is deployed instead of CPLD as used in Figure **6-16**. To fulfill the above requirements, a new version of the SCM is constructed as shown in Figure **6-17**.

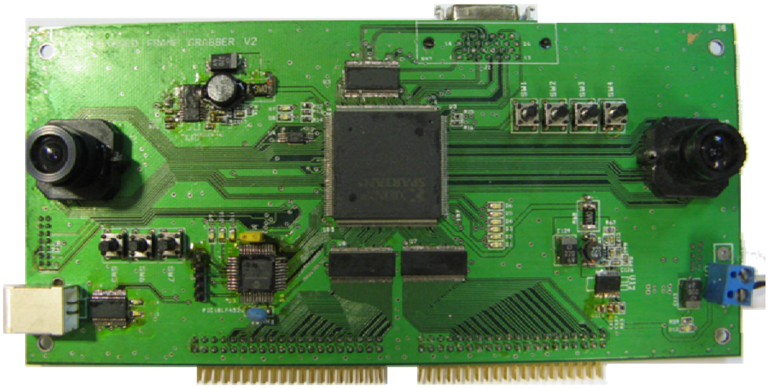


Figure **6-17**: New Stereo Capture Module

The data lines are expanded to accommodate the increased speed of camera modules which demands two LVDS implementation. The evolution of the camera modules reveals where the focus of the reconfigurable system should be.

6.45.3 Identifying a multi-thread and multi-mode workload

The system requires outlining the interfaces between the system and the environment. The system involves taking visual data from two cameras, processing them and producing processed data as outputs. As long as the environment is concerned, there are several terminators (e.g., clock, stereo-camera and multiple display devices) as shown in Figure **6-18**.

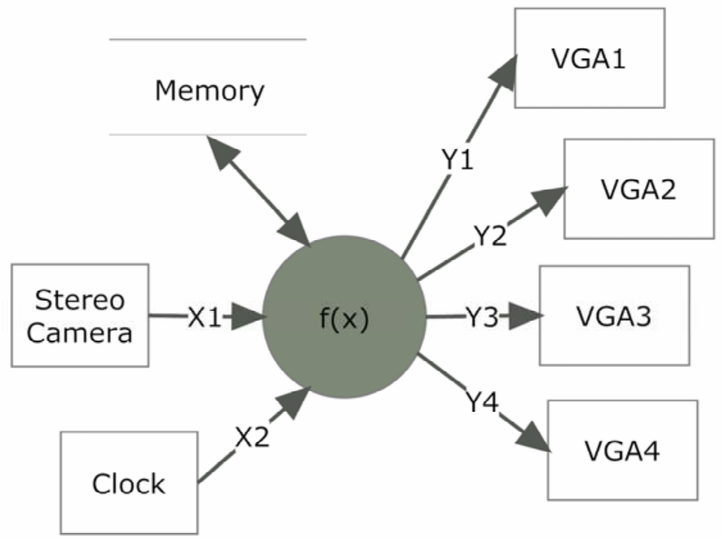


Figure **6-18**: Context diagram: Level 0

There can be three *applications* that require different micro-network as shown in Figure **6-19** .

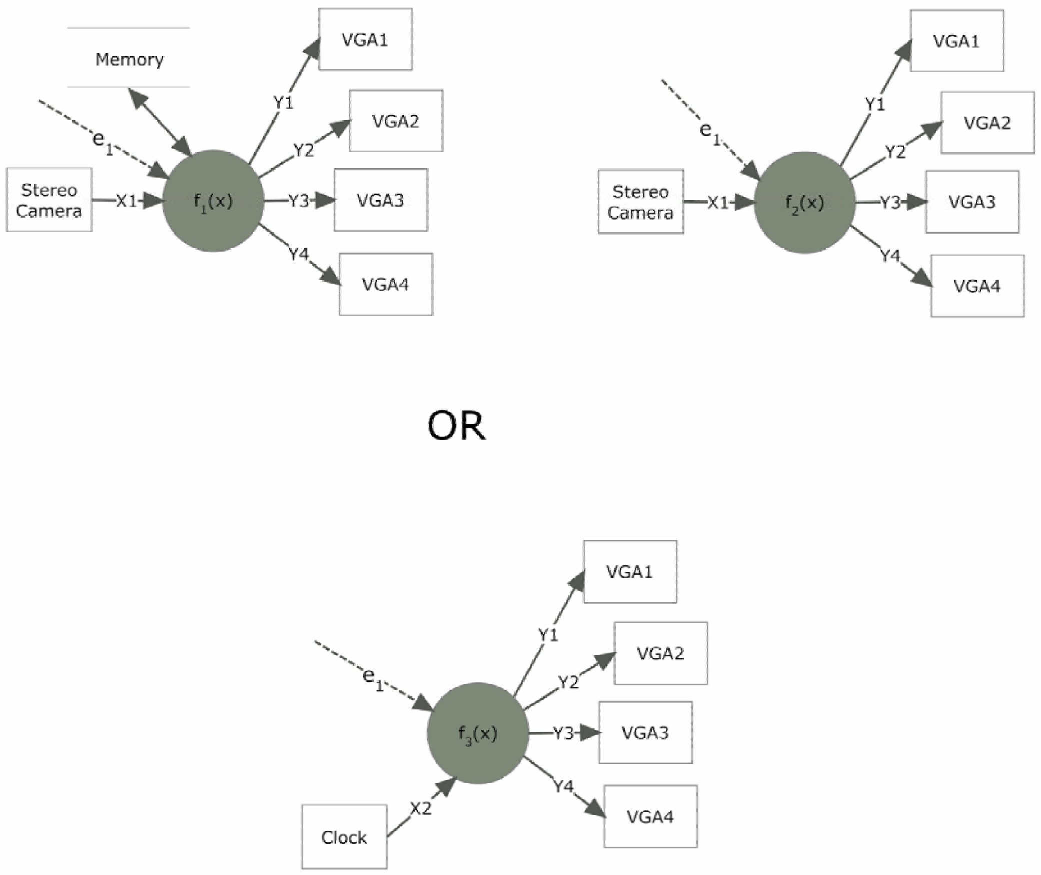


Figure **6-19**: Context diagram: Level 1

One application (i.e., f_2) is the minimal approach, "live video" that acquires the data set (e.g., controlled slave camera with 60 fps) that can be only seen by display device (VGA 60 fps). But because of lack of temporal data, complex processing can not be achieved.

The second application (i.e., f_1) provides more flexibility in term of processing. Due to the bandwidth difference between input device and output device the store works as the frame buffer. The third application (i.e., f_3) is where artificial data is created to be displayed on the display devices based on the clock signal.

Associating memory and its controller as shown in Figure **6-20** result static tasks (or isolated tools in the level 2) from the leveled tool diagram view[32]. The isolated tools in the level 2 are only associated with a particular value of configuration event, e_1 that f_1 associates with. The bubbles in the level 2 reveal not only the relationship with the configuration events but also hardware arrangement that the system needs to accommodate as shown in Figure **6-20**. Some of hardware arrangements are discussed in the following.

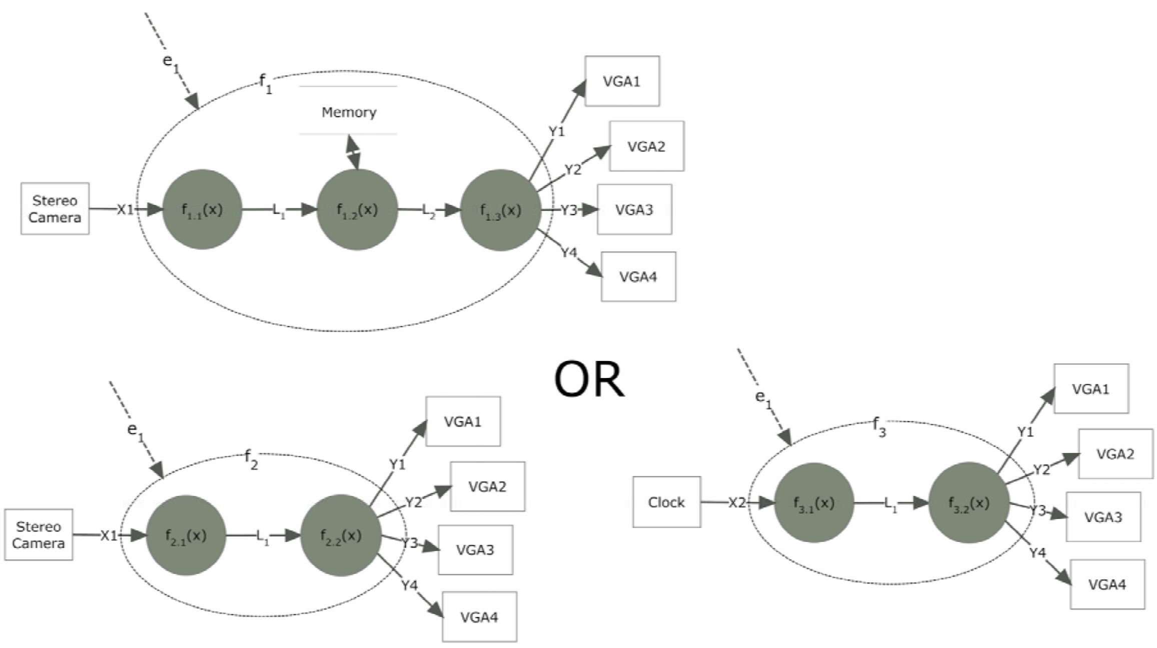


Figure **6-20**: Context diagram: Level 2

$f_{1.1}$ or $f_{2.1}$ is the camera controller that acquires the data from the stereo camera. In case of $f_{1.1}$ the camera controller can accommodate either master or slave mode camera because the speed of camera just needs to be faster than one of display device, where the

[32] Refer to section **5.7**.

camera controller, $f_{2.1}$ can only work in slave mode to synchronize with display devices. The memory controller, $f_{1.2}$ should take care of three way communication among camera data, memory and image processors. Because the memory device is not dual ported, $f_{1.2}$ needs to sit in the center of communication to arbitrate its access. However, the speed of $f_{2.1}$ should match exactly with one of threee display devices. Due to the particularity of its role that oversees other components, $f_{1.2}$ provides auxiliary outputs (e.g., vertical and horizontal control signals) to synchronize the display devices.

The workload can be decomposed further as shown in Figure **6-21**.

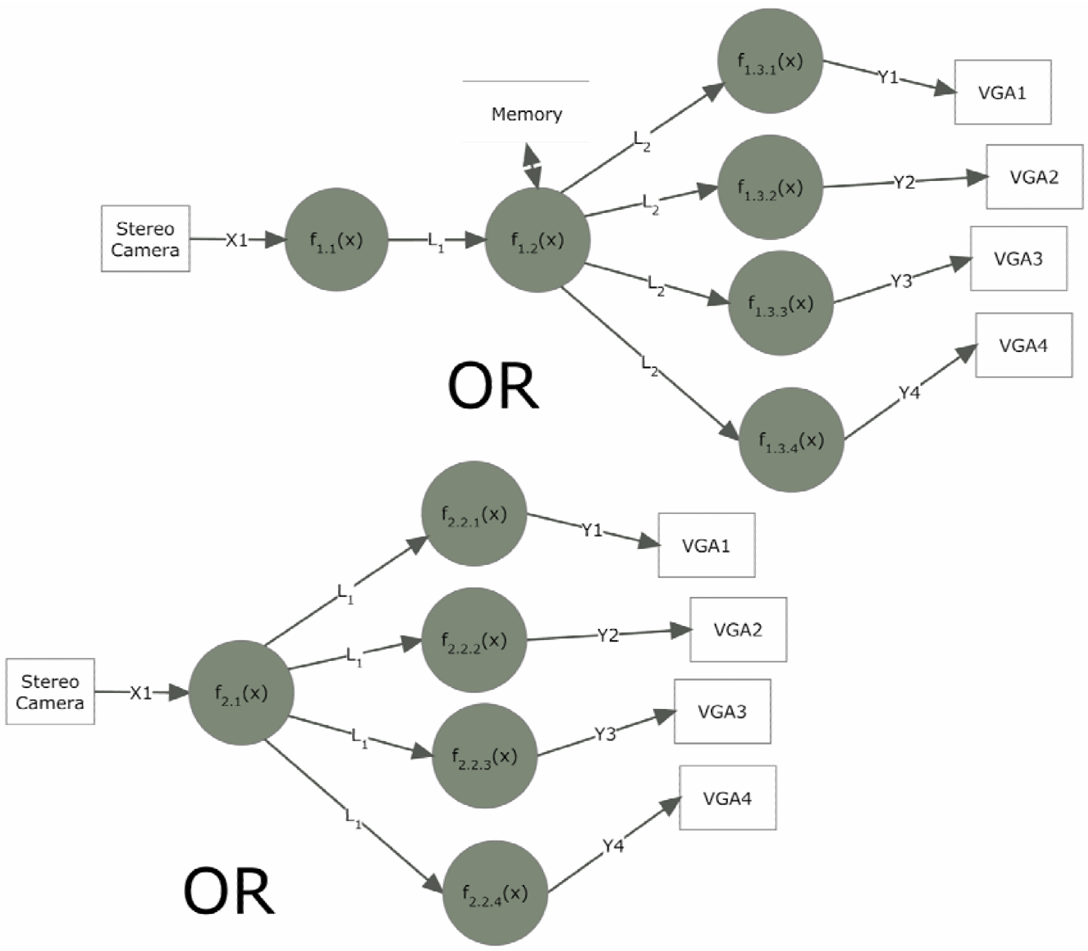


Figure **6-21**: Context diagram: Level 3

f_3 is not displayed in the context diagram for level 3. Because its outputs are indivisible[33] from its functionality point of view, the functionality of f_3 can not be decomposed further. Each bubble of the context diagram can have further applications, tasks and modes depending on the functionality that each node assigns with.

However, because of difficulties associated with establishing the micro-network interwoven with functional cores of the hardware components, one stops decomposing the context diagram further and focuses on finding a reconfiguration granularity.

6.45.4 Constructing a static architecture

Many systems are expected to deliver the performance that is required to deal with fixed input/output(s) of the system. It is the situation of many embedded systems that are developed for a specific application. However, because of the flexibility of the reconfigurable system, the implementation can look at the problem from a different point of view. The reconfigurable system can provide a scalable functionality to adapt to the changing environment or control the performance of terminal devices in order to provide an adaptable solution.

Extending with the previous implementation, one looks at how performance can be adapted. According to the decision tree given in Figure **5-17**, f_1, f_2 and f_3 are *applications* that result in various functionalities implemented in different temporal domain as shown in Figure **6-22**.

[33] Refer to section **5.7**

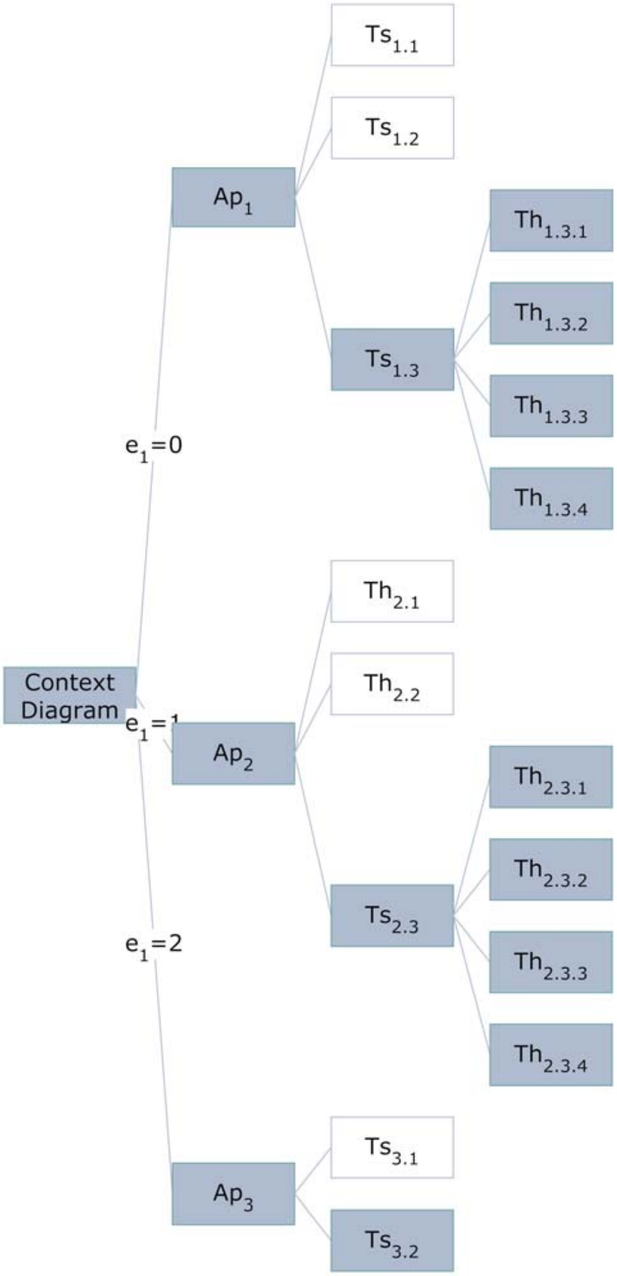


Figure **6-22**: Leveled tool diagram

Some tasks in the level 2 are found to be isolated tools as they represent the specific hardware arrangement that is the device controller or driver. In the level 3, each VGA output requires a particular image processing. Hence, it assigns the designated areas for each VGA output. Each thread of VGA output can be composed of many complex applications, tasks and modes. Before exploring what functionality can be implemented inside the threads of each VGA driver, let us examine what criteria affects the area that these functionalities can be implemented with.

According to the Eq. **5-7**, the area of *thread* can be calculated. Then, the *task* that consists of multiple *threads* would occupy the sum of these areas. The area should be equal to the area engaged by image processors shown in Figure **6-24**. Because the area designated for $Ts_{1.3}$, $Ts_{2.3}$ and $Ts_{3.2}$ covers 6-row and 80-column, the time to reconfigure the area is equivalent to 4.33ms[34]. Any functionality changes happen within the area can be programmed by partial reconfiguration. The input terminal devices can increase data acquisition, which in turn increases the system performance, as long as its timing allowance lies within 4.33ms that is very close to the frame rate of 200fps camera (i.e., 5ms). However, due to the practical reasons such as the contention of latchups, short circuits during dynamic reconfiguration, the empty configuration stream file needs to be loaded first. This takes, in the worst case, two times longer to configure. An empty configuration file resets the states of all CLBs in the region to a default value. Therefore, the region that is occupied by image processors requires 8.66ms to reconfigure its functionality as a whole. The frame rate of camera can only go up to 115 frames per second (fps).

Considering that there are four independent *threads* that exist in the region, the area needs to be divided further, where each *thread* still deals with the same data

[34] It is assumed that the configuration is performed at 100MHz speed with 32-bit SelectMAP interface.

structure of a frame. There is no difference for the time that it takes to process a frame. Therefore the same timing allowance (i.e., configuration time) is imposed on reconfiguring the functionality of each *thread*. As the area that reconfiguration deals with is decreased by four, $Area_{Ts}$, the deadline of configuration events, $\lceil d(e_k) \rceil$ stays the same based on Eq. **5-6**.

Moreover, there is processing of data that deals with a smaller unit than a frame. For instance, the MPEG-4 encoding schemes utilizes the functional module that is based on 8 × 8 pixel matrix that is called macro block. In a MPEG-4 standard that accommodates the frame rate of 30 fps with 640 × 480 resolution of visual data, the macro block requires 8 pixels of data rate, 0.87μs. Considering that an island style of a reconfigurable tile (i.e., a 16-row of column) takes 9.02μs[35] the MPEG-4 encoding can not be dynamically reconfigured using the data structure of 8 pixels. The encoding scheme needs to be modified at the system level to accommodate a bigger data structure that gives a bigger area to implement the functionality of MPEG-4 encoding.

Processing of images such as edge detections, transformation, image enhancing techniques requires smaller area of changes (e.g., a number of frames) between its modes of operations, where timing allowance is relatively large. More analysis of reconfiguration time and data structure is presented in Chapter 7.

In the implementation of *threads*, one considers the implementation of image processors. Because of dependant operations occurring in the image processor, it becomes redundant to separate their operations into smaller tasks. Any variation that

[35] MPEG-4 encoding blocks would require more than one column of 16-row CLBs.

results due to the changes of image processing core is classified to be modes of the *tasks* that are equivalent with *threads*. In the subsequent levels, the tasks, $Ts_{1.3}$ and $Ts_{2.3}$ are represented as shown in Figure **6-23**.

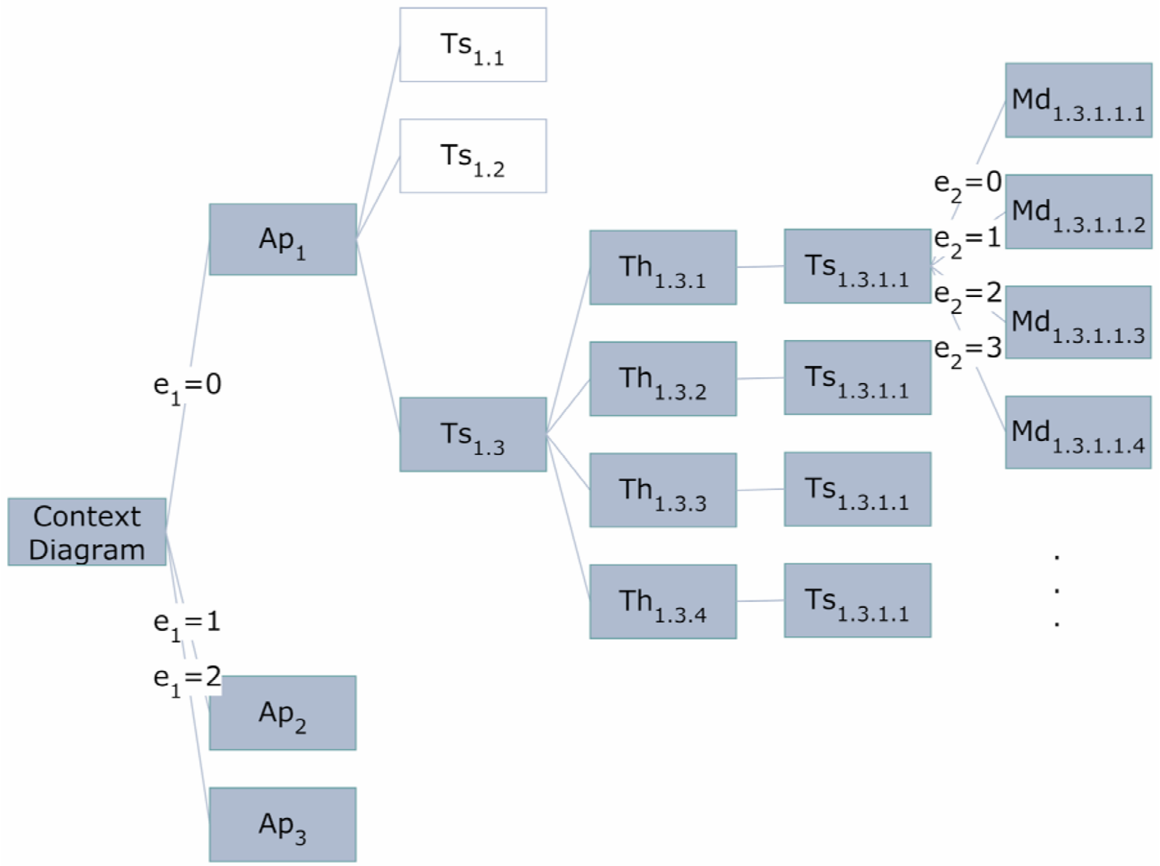


Figure **6-23**: A portion of the leveled tool diagram for $Ts_{1.3}$

Incorporating I/O constraints of the system, the initial conceptual assembly diagram can be achieved as Figure **6-24**.

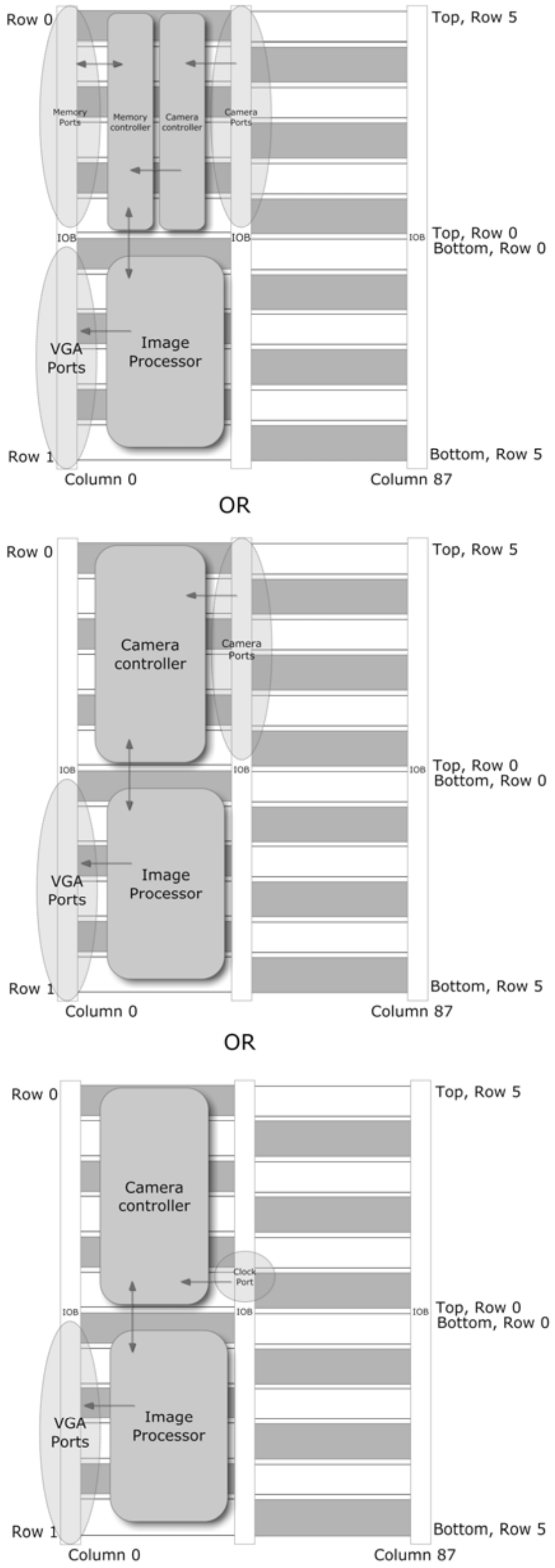


Figure **6-24**: Conceptual assembly: Level 2

The first conceptual assembly in Figure **6-24** illustrates the organization of $f_{1.1}$, $f_{1.2}$, and $f_{1.3}$ in the level 2 of the context diagram, namely camera controller, memory controller and image processors. Then the second one represents $f_{2.1}$ and $f_{2.2}$. Because the camera controller does not have any requirement of interface with memory, it can be placed left or right side of the chip. The third one shows image generator controlled by VGA driver. The placement of VGA driver is confined by I/O ports that connect to the RGB ports of video display devices.

As it may been seen, there are many possible ways to assemble the components. However, considering I/O constraints and possible signal routing complications that rise by placing components (e.g., controllers) far from its I/O ports, the conceptual assemblies shown in Figure **6-24** seem reasonable. In other words, in order to minimize the long routing lines between components, the components should be located right beside each other where it is possible.

Referring to the information available from Figure **6-21** and Figure **6-23**, a static architecture can be identified.

6.45.5 The procedure of on-chip assembly

In this implementation, each image processor has the direct connection to a Digital to Analog Converter (DAC) that outputs red, green and blue 8-bit pixel data. In return, each module requires VSYNC and HSYNC signal to synchronize with VGA display devices and 32-bit signal to import camera data. Figure 6-25 illustrates the connections that are populated by module called “inst_video_processor_v1”. They are two synch signals, R, G and B 8-bit data and 16-bit wide camera data port available for each of image processors. The placement of signal ports, width, protocol and available resources that are used for communication are all subject to be parameters for the selection of VHCs.

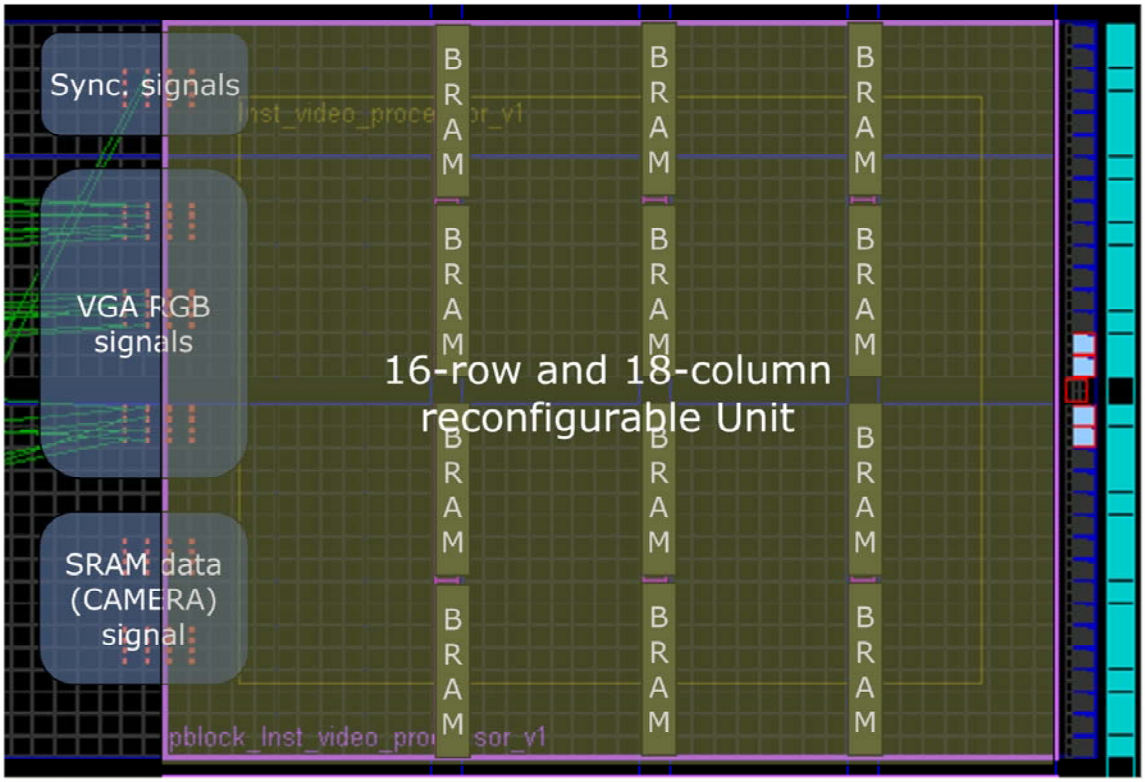


Figure **6-25**: An illustration of inst_video_processor_v1

The area occupied by image processor coincides with 16-row island style reconfigurable unit. Because there is a need for 2 units of Block RAM (BRAM) for each video processor, the modules can be placed either at the right or at the left side of the chip where the BRAMs are located in a column-wise manner. The full implementation shown in Figure **6-26** shows the placements of $Ts_{1.3.1.1}$, $Ts_{1.3.1.2}$, $Ts_{1.3.1.3}$, $Ts_{1.3.1.4}$ with respect to Xilinx Virtex-4 LX 160 device.

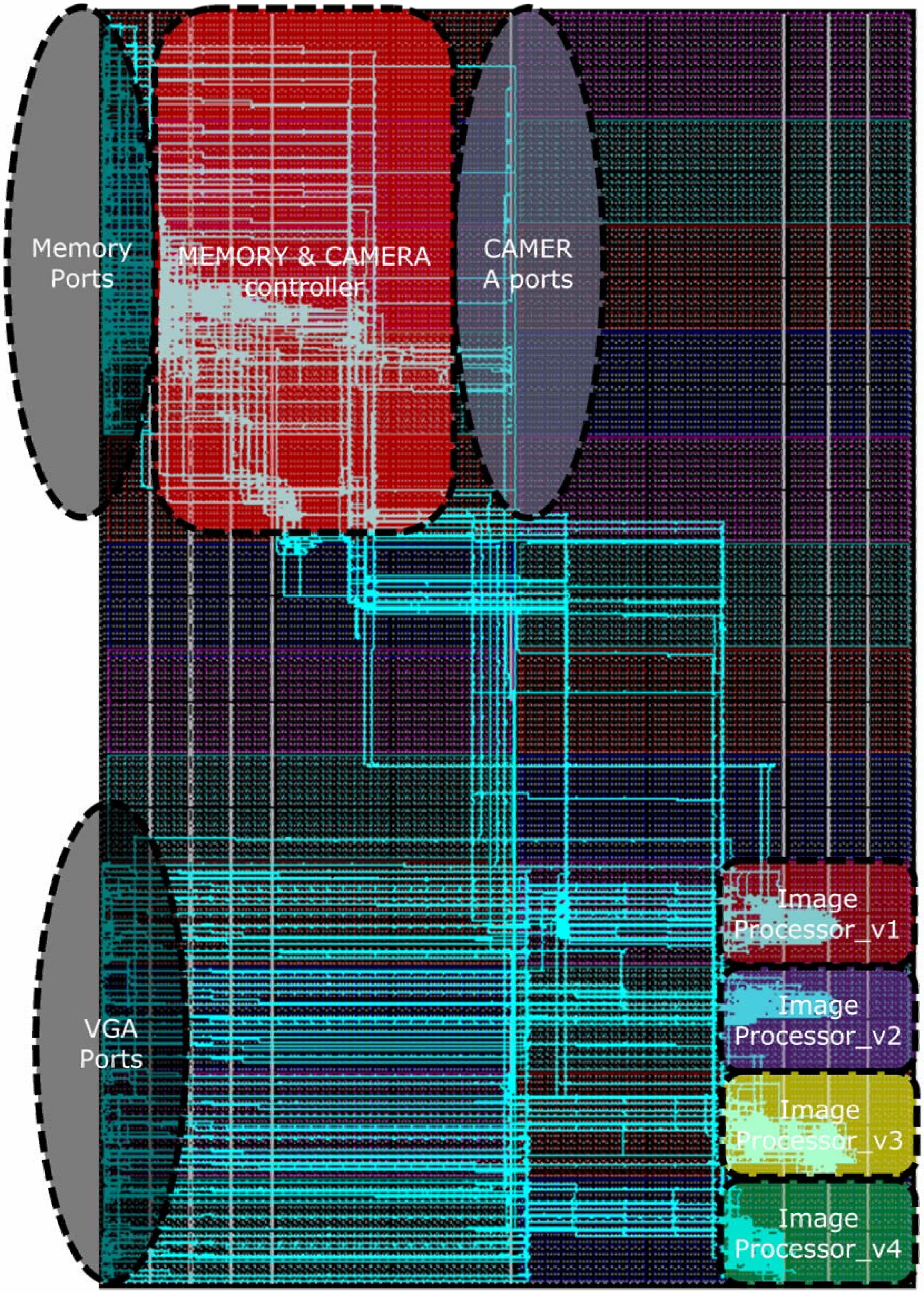


Figure **6-26:** An implementation with e_1=1, e_2=1, e_3=1, e_4=1, e_5=1

The precondition of the design is e_1=1 which allows reconfiguration of the tasks, $Ts_{1.3.1.X}$[36]. The size of tasks varies depending on how many logics are required by its functionalities. However, the allocated area should be the same as the biggest area needed by the functions. Because there is no further thread(s) or task(s) exists under the "image processor" modules, there is no need to create the static architecture for the area that are designated under those functions. The static architecture in this application becomes everything but "image processor" modules. The shape or configuration of static architecture greatly depends on the location and the way how interconnects are established one another. The modes of the tasks (e.g., image processors) should provide the same interface and protocol that the static architecture employs where the complexity of the functionality can be increased or reduced within the area given for the module.

The on-chip assembly of a reconfiguration system is the result of system organization that divides micro and component level architecture while the system level supports the (re)configuration procedures. The procedure of on-chip assembly is:

- Acknowledge configuration events and reconfigure the static architecture if needed;
- Identify the given parameters for reconfiguration and search for the functionality to be loaded in;
- Modify the header of configuration stream file(s);
- Load an empty configuration file for the area(s) if the initial state of registers can lead to misrepresentation of functionality;

[36] Subscribed X indicates that there are the multiple values of possible numbers. For example, $Ts_{1.3.1.1}$, $Ts_{2.3.1.1}$, $Ts_{3.3.1.1}$, $Ts_{4.3.1.1}$ for $Ts_{X.3.1.1}$

- Load a specific VHC(s).

Overall, the HOS that acknowledges the incoming configuration events can create a specific VHC as the detailed parametric information is discovered going through the design steps. The detailed procedure of the configuration as well as the information about hardware structure provides a way to partially reconfigure the programmable platform.

6.46 Summary

Because the intention of the experimental implementation is to show the feasibility of on-chip system assembly based on the commercially available FPGA architecture (e.g., Xilinx Virtex-4 architecture), this chapter outlines the system organization of the implementation platform called Multi-task Adaptive Reconfigurable System (MARS) from its system level to component level architecture.

This chapter also shows the procedural steps needed for run-time reconfiguration to be achieved. The general run-time reconfiguration example, “led counter”, illustrates the details of configuration streams that are important to construct the specific reconfigurable VHCs for on-chip assembly.

The Stereo camera board and SVGA display devices that produce and require real-time and stream data is attached to the MARS platform to create the environment that the reconfigurable system can become more cost-effective. The proposed reconfigurable system design methodologies are applied for the workload to figure out what components are needed to support run-time reconfiguration and what information is needed to create Virtual Hardware Components (VHCs) so that a suitable reconfiguration granularity that results in an optimal static architecture can be employed.

The following chapter will analyze the results of the implementation in details to figure out what structure of applications, architectures and granularity can make the reconfigurable system more cost-effective.

Chapter 7

Analysis of Results

This chapter investigates the results of the multiple image processing cores of stereo camera data on the MARS platform based on a set of metrics which include resource utilization, reconfiguration granularity, scalability and power consumption. All of the metrics affect the cost-effectiveness of the reconfigurable system either by accommodating more powerful application or by reducing system costs when they are adjusted correctly.

7.1 Analysis of Cost-effectiveness

The main motivation of the proposed architecture synthesis methodology is the cost-effectiveness of the reconfigurable system. The easiest way to measure the cost-effectiveness is the amount of silicon used to implement the intended functions. Using the assumptions given for the analysis in section **5.5**, the estimation of the silicon costs can be obtained. However, to make the cost analysis clearer, the cost comparison between *fixed* and *reconfigurable* hardware is substituted with implementation of the intended workload in a presumably *non-reconfigurable* FPGA and *run-time reconfigurable* FPGA respectively. According to the cost analysis given in section **5.5**, the multitude of tasks needs to be in the range of 10s to 100s[37]. The cost analysis of the implementation in section **6.3** (with 14 modes) is used as the basis of explaining the cost benefits of many complex systems (with 10s to 100s reconfigurable functions), namely multi-task and multi-mode reconfigurable system.

The silicon cost for a *non-reconfigurable* implementation is the silicon usage that is required by multiplexing all the functions that are presented in section **6.3**. It consists of three *applications* that exist in different times. Depending on the *applications*, there are changes to be made. Due to non-reconfigurability, the changes require the system to reconfigure as a whole. Yet, when they are implemented as multiplexed functionalities on a chip, only a portion of hardware contributes to the functionality of the system at a time. Note that the results analyzed in this chapter can be greatly different depending on how the functionalities are described in HDL and synthesized into RTL and assigned with technology-level components.

Nonetheless, the question for silicon cost comes down to whether each application in a *run-time reconfigurable* FPGA requires bigger or smaller silicon area than the multiplexed applications in a *non-reconfigurable* FPGA. In order to do the analysis, we need to find the leaves of Figure **6-22**. Each leaf (i.e., task) should be big enough to accommodate all of its modes. For the implementation on a *run-time reconfigurable* FPGA, the required number of 4-input Look Up Tables (LUTs) for each mode is obtained as shown in Table **7-1**.

[37] Referring to the silicon analysis obtained in section **5.5.1**.

Table **7-1**: The list of video processing modes for a camera

	Required Resource		**Resource Utilization**	
Mode	**4-input LUTs**	**RAMB16**	**4-input LUTs**	**RAMB16**
Camera Display	172	2	33.6%	50.0%
Normal Edge	177	2	34.6%	50.0%
Inverse Edge	174	2	34.0%	50.0%
PingPong Edge	341	2	66.6%	50.0%
Color Intensity (Red)	102	2	19.9%	50.0%
Color Intensity (Green)	102	2	19.9%	50.0%
Color Intensity (Blue)	102	2	19.9%	50.0%

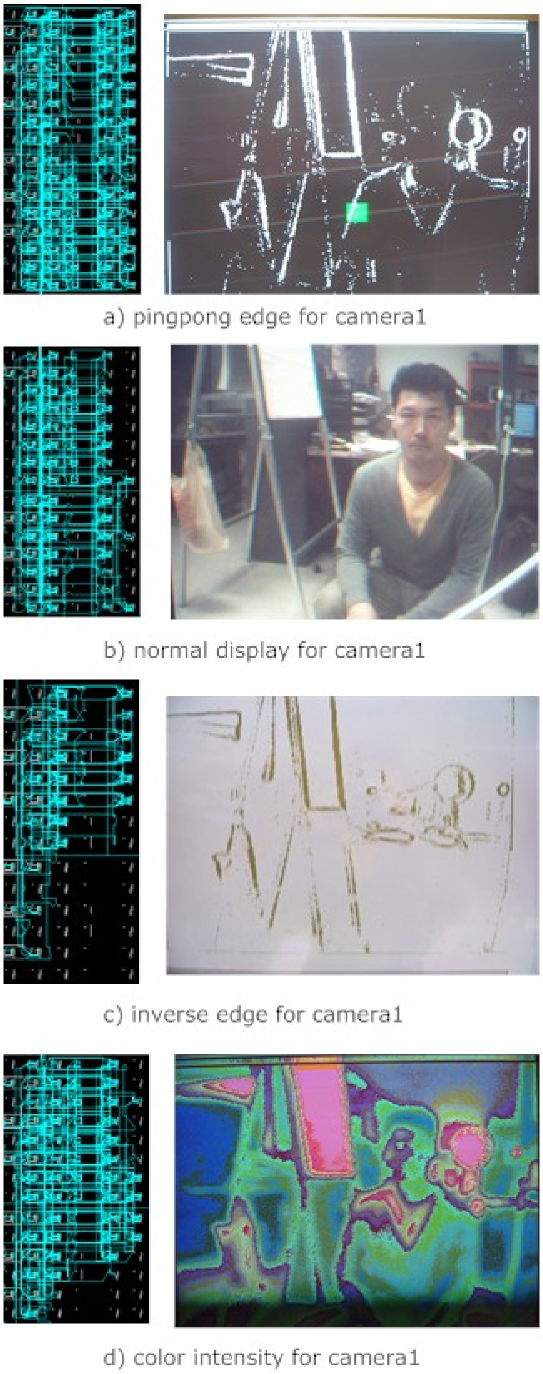
Each mode can be potentially applied to any video processor designated for the outputs of VGA1, VGA2, VGA3 and VGA4. If one accounts for left and right of the stereo camera, the number of modes that each video processor can have is 14.

In order to implement these modes in a *run-time reconfigurable* FPGA, a fixed area needs to be designated. Considering many aspects of reconfiguration such as:

- the number of available LUTs;
- the number of RAM, FIFO or DSP (heterogeneous) blocks involved;
- the area required by local routings including ones by heterogeneous blocks;
- the minimum reconfigurable units (i.e., 16-row columns);
- the boarder area that is sufficient enough to plug in all required interface signals – Macro Blocks (e.g., sync, camera data and VGA Red, Blue and Green signals),

the designated silicon space accommodates all the modes and is chosen to be 8 columns of a 16-row reconfigurable unit because of the maximum number of logics needed and the local routings which are associated with them. The number of LUTs available within the unit is 512 which produce the results of the resource utilization given in Table **7-1**. Figure **7-1** illustrates the examples of these modes that occupy the same designated area of Virtex-4 LX 160 FPGA as shown in Figure **6-25**. Figure **7-1** shows the images of physical FPGA logic and routing structure along with visual output of these functionalities. The variation in logic utilization depending on the mode of the tasks can be observed by placed logics and routed wires on the left hand side of Figure **7-1**. The overall resource utilization (i.e., $s_{utilization}$) of modes is 32.6% for the reconfigurable design.

The common area (i.e., A_{common}) that is shared by the modes of functions is obtained by taking the difference between the area needed for combining all functionalities in a *non-reconfigurable* FPGA and the area needed for all functions in a *reconfigurable* FPGA. The combined functionality can be created by putting all functions in hardware. The total number of logics required to combine all modes for the tasks (e.g., $Ts_{X.2.3.X,}$) in hardware is found to be 7866 4-input LUTs. The total number of logics, 9360 is calculated by adding all the areas needed for each mode of the functions. A_{common} can be extracted by calculating the difference between these two values. Thus, A_{common} is found to be 16%.



a) pingpong edge for camera1

b) normal display for camera1

c) inverse edge for camera1

d) color intensity for camera1

Figure **7-1**: Examples of modes in FPGA implementation vs. visual display

For the implementation in a *non-reconfigurable* FPGA, the controller is developed to multiplex the modes of the functions based on the configuration events that are triggered by the environment. The number of logics required by the controller is referred as the multiplexer overhead, $ov_{multiplexer}$. The $ov_{multiplexer}$ is found to be 17% based on the 4-input LUTs used for the controller which sum up to be 1336 when there are 14 modes possible.

In conclusion, the silicon cost of a *non-reconfigurable* FPGA for the multi-task and multi-mode video processor implementation, sc_{fixed} is found to be *9202* 4-input LUTs, while the silicon cost of the same implementation, $sc_{reconfigured}$ in a *run-time reconfigurable* FPGA is given as *2480* 4-input LUTs. The visual comparison of the silicon costs between the reconfigurable design and the fixed design that implements exactly the same functionalities is shown in Figure **7-2**.

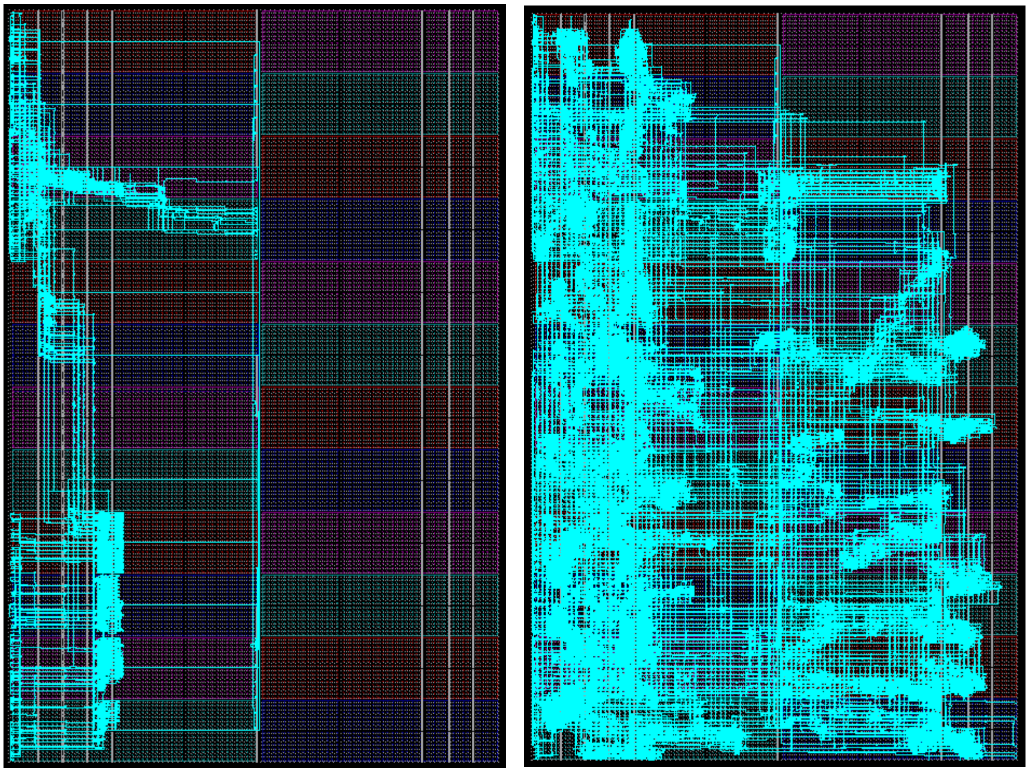


Figure **7-2:** A visual comparison of the fixed design and the reconfigurable design

The silicon costs required by the fixed design is represented by the number of logics required by the implementing all modes of the functions in hardware as shown in b) of Figure **7-2.** Without even counting the routings that are occupied by multiplexing, the logic difference accounts for 73% of hardware compared to the fixed approach (i.e., ASIC).

Today[38] Xilinx Virtex-4 LX160 FPGA costs $4,286.6700 where the FPGA with 75% less logic in the same family (i.e., XC4VLX40) costs $609.3300. The cost of service hardware, $69.96 that enables run-time reconfiguration, is two magnitudes smaller than

[38] The prices are available from http://avnetexpress.avnet.com on January, 20, 2009

the cost difference, \$3677.34, that can be made by reconfigurable design as shown in Table **7-2**. Let us assume that the reconfiguration overhead, $ov_{reconfiguration}$ is 10% of the reconfigurable hardware.

Table **7-2**: Costs for run-time reconfiguration service hardware

Function	Description	Part number	Unit Price
Program Loader	Xilinx Spartan-3 FPGA 1 milion gates	XC3S1000	\$62.10
VHC memory	Micron Flash memory 2Gbit	MT29F2G16	\$7.86

It seems that as the number of modes or reconfigurable functions increases, the more savings can be made. Based on the real values found for the implementation of video processors, let us evaluate how cost-effective the implementation becomes depending on the number of modes of the functions. The equations in section **5.5.1** employ the ratio of reconfigurable functions to express the application's multi-functionality. Yet in this chapter the equations are reformulated to use the number of modes for the multi-functionality. First the silicon cost for the fixed hardware, sc_{fixed} is given in Eq. **7.1**.

$$sc_{fixed} = \left(N \times n\right)(1 - A_{common})c_{LUT} + ov_{multiplexer}\left(N \times n\right) \quad [LUT] \qquad \textbf{7.1}$$

where N is the number of reconfigurable functions and n is the average of the number of modes available for each function. The function, $ov_{multiplexer}\left(x\right)$ represents the amount of overhead required by the number of functions and is formulated by curve fitting with the real data that are extracted from the implementation. The function appears to be a quadratic polynomial with $ov_{multiplexer}\left(x\right) = 4.188x^2 + 15.5x + 173$ where $x > 2$ in term of

4-input LUTs. In order to get the output of sc_{fixed} in terms of 4-input LUTs, the conversion constant for LUT, c_{LUT} = 157.3 is used. [39].

Conversely, the silicon cost for reconfigurable hardware, $sc_{reconfigured}$ is articulated as Eq. **7.2**.

$$sc_{reconfigured} = N \times c_{LUT} + ov_{reconfiguration} \quad [LUT] \qquad \mathbf{7.2}$$

where $ov_{reconfiguration}$ is assumed to be 10% owing to the observations in Table **7-2**. For Eq. **7.1** and Eq. **7.2**, the static functions of the system are excluded from the silicon cost calculation. Figure **7-3** plots the silicon saving versus the silicon cost that can be made on the multi-task and multi-mode video processor implementation.

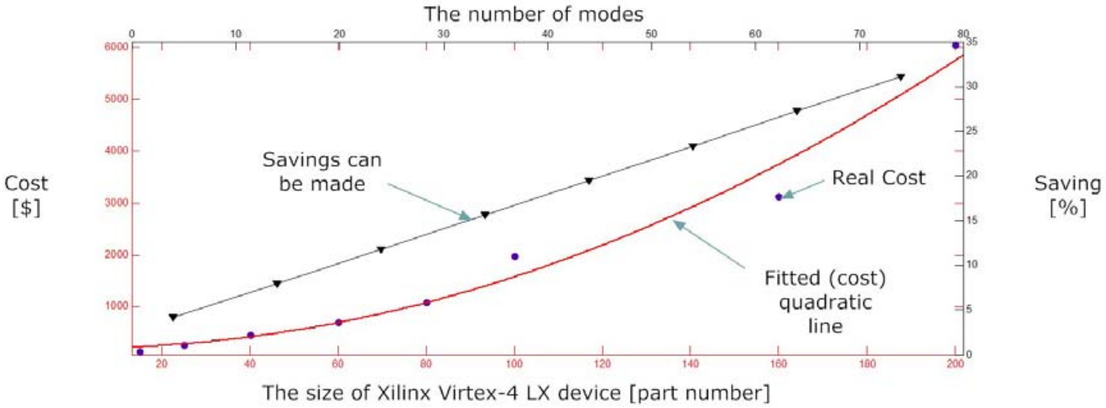


Figure **7-3**: Silicon saving vs. Silicon cost

As the number of modes increases, the reconfigurable design utilizes the smaller area of a device. If you combine both trends shown in Figure **7-3**, it indicates that the linear increase of silicon savings results in quadratic savings where in real life the

[39] Refer to the Appendix B.

designers spend less money to purchase small devices. In other words, the designers are expected to spend more money than the rate that the number of modes increases.

7.2 Estimation of Configuration Granularity

The full reconfiguration requires 4926Kbytes to be loaded in. With the 32-bit SelectMAP at 100MHz, it will take 12.31ms. If each *application* is the function that needs to be changed within the unit of the output device (e.g., one VGA frame – 30 frames per second), it should be able to accommodate the reconfiguration of all the applications.

In short, the full configuration can be conducted to produce the different functionality, if the loss of a frame that is less than 81fps can be tolerated. One of 56 modes can be loaded in to reconfigure the functionality of all functions. One of these functionalities is shown in Figure **7-4**.

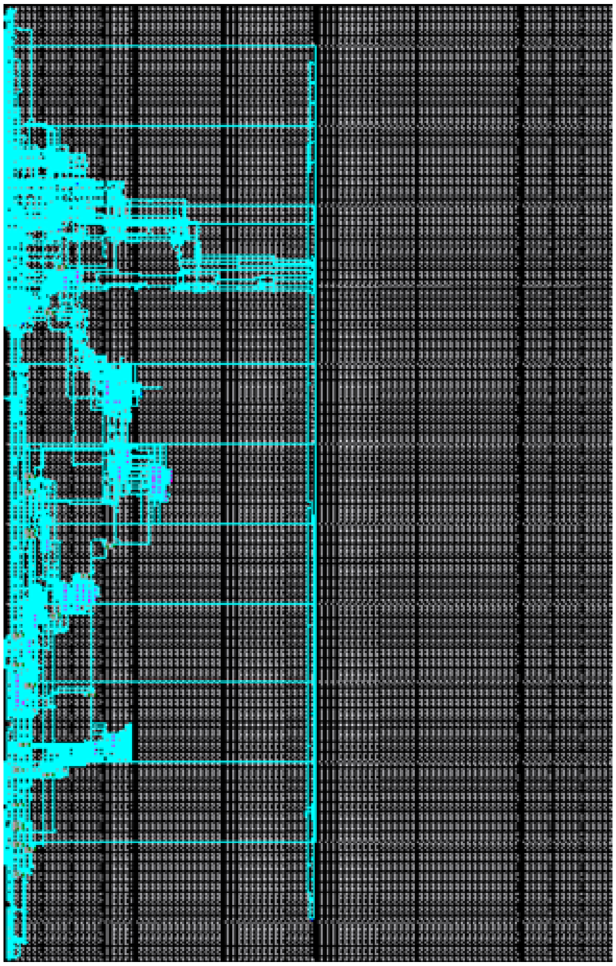


Figure **7-4**: An example of full configuration for a mode of functions

However, full reconfiguration requires all parts of the system to be halted. As a result, there will be no output in any part of the system. All VGA outputs should be halted even though change in only one functionality may be required.

Therefore, if the system desires to continue its operations while there is change(s) of functionality, then a full configuration is not applicable. From the synthesis of functionalities, one finds that there is a need for 512 4-input LUTs. It is equivalent to a 16-row 8-column of reconfigurable logics. The size of the configuration file contains

232,224 bits of information. Hence, it takes 72.57 μs to reconfigure a function at 100MHz with 32-bit SelectMAP.

If the system's goal is to produce the visual output of stereo camera for continuous display, the output data should not be disturbed by any change required for functionalities. Timing allowance (or deadline) must be smaller than the duration of data unit that the function deals with. The following table, Table **7-3** summarizes the different data units.

Table **7-3**: Different data units at 30 fps with 640×480 resolution

Data Unit	Duration	Example
Pixel	22.6ns	color intensity
Lines	multiples of 14.4us	Stereo rectification
Macro block	multiples of 42.4us	MPEG-2 and 4 JPEG
Frame	33.3ms	Object identification

Additionally there are periods that do not produce any output in VGA standards. They are called:

- Vertical Blank (e.g., 64μs)
- Vertical Front Guard (e.g., 1.02ms)
- Vertical Rear Guard (e.g., 0.35ms)
- Horizontal Blank (e.g., 3.77μs)
- Horizontal Front Guard (e.g., 1.89 μs)
- Horizontal Rear Guard (e.g., 0.94 μs)

Because the data should be continuously valid, the reconfiguration of functionality for the multi-task and multi-mode video processor can not be interleaved in between the data units. Interleaving the reconfiguration between the outputs of data can be successfully achieved in other applications where the valid outputs need to be available at times but not all times.

The reconfiguration of a *task* takes 72.57 μs. That is longer than any horizontal periods available (e.g., 72.57 μs > 3.77+1.89+0.94 μs). Thus, the reconfiguration of a task can only be realized at the data unit of a frame where the periods (e.g., vertical blank, front guard and rear guard) are larger than the time that takes to reconfigure the task(s). At the same time, it allows the possibility of implementing much more complex functionality. For instance, the allowance time (e.g., 1,414 μs) can reconfigure the area up to 32-row 78 columns (or 48-row 52 columns) that is 19.5 times larger in functionality than the currently allocated silicon space, 16-row 8 columns. The equation states that the reconfiguration time that is required by the reconfigurable components, $\lfloor area(Md_i)\rfloor + \lfloor area(Lk_j)\rfloor$ should not exceed the shortest deadline, $\lceil d(e_k)\rceil$ where the configuration events are invoked between data units.

7.3 Scalability

Another advantage of the reconfigurable design is its ability to adapt. In many cases, the deadline of the application can be scaled up or down depending on the performance of input/output devices that the system interfaces with. In other words, there can be dynamic requests from input/output terminal(s) to change its performance, therefore changing its interface, not physically, but temporally. The temporal processing, as long as it utilizes the same datapaths, can be achieved within the same reconfigurable area.

In our implementation, the scalability of input/output devices is demonstrated by adapting a new Stereo Capture Module (SCM) shown in Figure **6-17**. The new SCM is capable of providing the frame rate up to 200 fps. Since all of the Virtual Hardware Components (VHCs) implemented for the tasks of "video processor", can work with the frequency that supports the frame rate up to $\approx$ 500 fps, there is virtually no change required for the VHCs to support a faster or slower input/output performance.

Since the reconfiguration time required for a reconfigurable video processor is known as 72.57 μs, the frame rate of a camera (or VGA display) device can theoretically scale up to 1169 fps. However, considering the changes required for the static task(s) (e.g., camera controller) and the possibilities of reconfiguring all 4 tasks at the same time, the maximum frame rate that the implementation of reconfigurable design can support is 146.125 fps. If the desired video processing is faster than the frame rate of 146.125 fps, the video processing tasks should be scaled down in order to be fitted in the area for which reconfiguration is still possible.

7.4 Analysis of Power Consumption

In order to tell the power benefits that the reconfiguration brings, it is necessary to obtain the power consumption of individual video processors that are located in a specific location in the FPGA (i.e., XC4V160LX). One way to find the power consumption of these components is by direct measurement. Table **7-4** shows the measurements based on the given description.

Table **7-4**: Power measurement for reconfigurable multi-video processor implementation

Description				Current [mA]	Supply Voltage [V]	Power [mW]
VP1	VP2	VP3	VP4			
blank	blank	blank	blank	964	5	**4820**
Normal edge + pingpong (c1)	blank	blank	blank	966	5	**4830**
Normal edge (c1)	blank	blank	blank	967	5	**4835**
Inverse edge (c1)	blank	blank	blank	967	5	**4835**
blank	blank	blank	Normal edge + pingpong (c2)	965	5	**4825**
blank	blank	blank	Normal edge (c2)	967	5	**4835**
blank	blank	blank	Inverse edge (c2)	966	5	**4830**
Normal edge (c1)	Normal display (c1)	Normal display (c2)	Color intensity (c2)	972	5	**4860**
Inverse edge (c1)	Normal display (c1)	Normal display (c2)	Color intensity (c2)	972	5	**4860**
Fixed application with all the modes				1109	5	**5545**

Due to the integration of power supply for all hardware components in the MARS platform, it is not possible to find out what percentage of power is consumed by the reconfigurable device (i.e., FPGA) from the measurements. However, it is possible to extract the power consumption that is caused by each individual function from Table **7-4**. Comparing each mode of functionality with the power measurements of all blank functionality, the following information can be generated as shown in Table **7-5**.

Table **7-5**: Power consumption of VHCs in operation.

Description	Power [mW]
VP1 (normal edge + pingpong)	**10**
VP1 (normal edge)	**15**
VP1 (inverse edge)	**15**
VP4 (normal edge + pingpong)	**5**
VP4 (normal edge)	**15**
VP4 (inverse edge)	**10**
Average power difference between fixed and reconfigurable	**685**

Real-life systems would deal with the applications of higher logic utilization and faster frequency which demands higher power than the experimental results obtained in Table **7-5**. Nonetheless, the power difference obtained between fixed and reconfigurable designs shows the fundamental difference due to the logical utilization.

In order to find out how much power is saved by reconfigurable design, the total power consumption of a reconfigurable design needs to be obtained. The power consumed by FPGAs can be broken down into two components, dynamic and static (quiescent) power. The dynamic power consumption of a digital design is a function related to the supply voltage, capacitance and switching activity [**96**]. For FPGAs, the power consumption is also dependent on the amount of time the resource is being utilized. Eq. **7.3** shows the FPGA power can be modeled as a function of capacitance C_i, voltage swing, V_i, and operating frequency activity f for resource i according to [**97**].

$$P = \sum_{i} C_i V_i^2 f_i \qquad \textbf{7.3}$$

Eq. **7.3** makes the designer pay attention to three factors: effective capacitance; resource utilization and switching activity. The power consumed by video processor modules was estimated using XPower, which is a utility available from Xilinx to estimate power using Eq. **7.3**. XPower uses gate level simulations of the design to estimate the average switching activity of the resources and power consumption over time. The simulation allows XPower to estimate the dynamic power consumption associated with utilized resources and their effective capacitance. However, XPower does not model routings as the resources that consume power. The method adapted by [**97**] would model the power consumption more accurately because of deep submicron effects and multi-functionality of the target applications. The analysis using XPower can not measure precise difference. However, it would still show the difference in power consumption.

The average power consumption for a mode of the tasks is shown in Table **7-6**.

Table **7-6**: Reconfigurable design power consumption

Component Name	Value [W]	Used	Utilizatoin [%]
BRAMs		6	2.1
Clocks	0.08066	1	
DCMs	0.02891	1	8.3
IOs	0.21188	209	27.1
Logic		1572	1.2
Signals	0.00225	2235	
Total Quiescent Power	1.26697		
Total Dynamic Power	0.29625		
Total Power	1.56323		

Unused elements found on the FPGA can have a base rate of power consumption identified by the static power as shown in Table **7-5**. Thus, the power consumption of the fixed design with multiplexed functions would cost more power to keep them in hardware even when they are not in use. The fixed system only utilizes one mode at a time that is the same as the functions of reconfigurable system given in Table **7-6**. The power consumption of the fixed design is given in Table **7-7**

Table **7-7**: Fixed design power consumption

Component Name	Value [W]	Used	Utilizatoin [%]
BRAMs	0.40229	88	30.6
Clocks	0.33895	1	
DCMs	0.06037	1	8.3
IOs	0.02192	216	28
Logic	0.00015	10042	7.4
Signals	0.00222	14983	
Total Quiescent Power	1.33895		
Total Dynamic Power	0.6913		
Total Power	2.03025		

The cost benefits (e.g., 29.8% for multi-vide processor implementation) of power consumption of the reconfigurable system (e.g., 1563 mW) compared to the fixed system (e.g., 2030 mW) are due to mainly its lower resource utilization in the temporal domain. All results related-documents are listed in Appendix C.

7.5 Summary

The multi-task and multi-mode reconfigurable system has many advantages of savings compared to the fixed systems. However, the quantitative analysis of these cost matrixes is a difficult task because there are many factors to be considered. In this chapter, the results were obtained through quantitative analysis of multi video processor implementation in a fixed and reconfigurable system. Such result is that the reconfigurable system can quadratically reduce the system cost as you reduce the area utilized by modes of the tasks (e.g., 75% area saving results in $3677.34 when the cost of the reconfigurable device is $609.33). Another is that the power used by the multiplexers and multiplexed functions in a fixed design costs 29.8% more power than the reconfigurable designs. The reconfigurable granularity and scalability is part of a design process that requires the designer’s attention to make the reconfigurable system more optimized in terms of their performance and flexibility.

Chapter 8

Conclusions

The fast changing *market trends*, the ever increasing *logic density* and the recent surge of *multi-functional applications* have been challenging the effectiveness of conventional computing approaches. Using the run-time reconfigurability, the reconfigurable system can adapt its internal structure and behaviour in response to a dynamic environment. A new architecture synthesis methodology for reconfigurable systems is necessary to construct cost-effective reconfigurable architecture that is more competitive than conventional computing approaches. In this thesis one accomplishes:

- A new architecture synthesis methodology that reflects the specifications of a multi-task and multi-mode workload into the physical structure of a programmable device;
- A novel system-level architecture that dynamically changes its functionality while keeping the structure of essential communication;
- A novel reconfiguration granularity that results in optimal static architecture;
- A new design procedure that enables on-chip self-assembly of a reconfigurable system.

In order to demonstrate the benefits of the above features, the Multi-task Adaptive Reconfigurable System (MARS) platform is manufactured with a reconfigurable functional module, a cache for reconfigurable components, a virtual component library and a hardware operating system. The cost analysis of the MARS platform reveals the advantages of the proposed architecture synthesis methodology compared to fixed design approach (e.g., 73% area saving, 29.8% power saving and 19.5 times functional scalability). This thesis shows that reconfigurable system increases its cost-effectiveness when the proposed design methodology is applied to a multi-task and multi-mode workload.

The cost-effective implementation of reconfigurable computing is enabled by an interesting concept – *virtualization of processing resources*. However, in our implementation, the experiments are implemented on the fixed size of processing resources depending on the reconfiguration granularity. The proposed architecture synthesis methodology does not optimize the area of micro-level components (i.e., VHCs). To fully realize the potential of run-time reconfigurable systems, the virtualization of processing resources should be decoupled from its physical constraints. Hence, the system should be able to take any shape or form to represent a lively aspect of current requirements. In order to achieve the above potential, the future of the reconfigurable system requires the implementation of a complex and intelligent Hardware Operating System (HOS) that is capable of creating, relocating and (re)loading arbitrarily shaped VHCs autonomously. The full implementation of the HOS also should enable self-restorable reconfigurable system methodologies that can relocate VHCs and adapt static and network level architecture due to error(s).

With the support of fast configuration mechanism and full implementation of a virtual hardware library and a cache, the reconfigurable system can take the advantages of cost savings that the run-time reconfiguration provides and can offer the services that computation-intensive and high performance applications require.

Bibliography

1. Young-Oak Kim, “Oullim Syncacophony”, In *ICOGRADA Millennium Congress*, Keynote Speech, Seoul, Korea, 2000.

2. Henry Petroski, *The Evolution of Useful Things: How Everyday Artifacts-From Forks and Pins to Paper Clips and Zippers-Came to be as They are*, New York: Random House of Canada, 1992.

3. Laurence A. Moran, University of Toronto, “The Modern Synthesis of Genetics and Evolution”, updated by Sep., 11, 2008, http://bioinfo.med.utoronto.ca/Evolution_by_Accident/Modern_Synthesis.html.

4. Petri Kukkala, Marko Hamnikainen, and Timo D. Hamalainen, “Implementing a WLAN video terminal using UML and fully automated design flow”, In *EUROSIP Journal on Embedded Systems*, Vol. 2007, Issue 1 (January 2007), pp. 20-34.

5. Touradj Ebrahimi, and Fernando Pereira, *The MPEG-4 Book*, Prentice Hall IMSC multimedia series, New Jersey: Prentice Hall RTR, Chapter 1, 2002.

6. T. Weyrich, H. Pfister, and M. Gross, “Rendering Deformable Surface Relectance Fields”, In *IEEE Transactions on Visualization and Computer Graphics*, Volume 11, Issue 1, January 2005, pp. 48-58.

7. A. Gersht, and S. Kheradpir, ”Real-time bandwidth allocation and path restorations in SONET-based self-healing mesh networks”, In *Proceedings of Communications*, 1993. ICC 93, Volume 1, 1993, pp. 250-255.

8. D. Kim, J. Young, S. Milton, H. J. Kim, and Y. Kim, ”A real-time MPEG encoder using a programmable processor”, In *IEEE Transactions on Consumer Electronics*, Volume 40, Issue 2, May 1994, pp. 161-170.

9. J. Dunlop, J. Pons, J. Gozalvez, and P. Atherton, ”A real-time GSM link adaptation hardware demonstrator”, In *Proceedings of Vehicular Technology Conference 2000,* VTC 2000-Spring Tokyo. IEEE 51st Volume 1, May 2000, pp. 590-594.

10. Hubert Zimmermann, “OSI reference model—the ISO Model of Architecture for Open Systems Interconnection”, In *IEEE transactions on Communication*, Volume COM-28, Number 4, April 1980, pp. 425-432.

11. M. Sgroi, "Addressing the system-on-a-chip interconnect woes through communication-based design", In *Proceedings of the 38th conference on design automation*, Las Vegas, Nevada, 2001, pp. 667-672.

12. Dennis Sylvester, and Kurt Keutzer, "Rethinking deep-submicron circuit design", *IEEE Transactions on Computer*, Volume 32, Number 11, November, 1999, pp 25-33.

13. D. Sylvester, and K. Keutzer, "Impact of small process geometries on microarchitectures in systems on a chip", In *Proceedings of the IEEE*, Volume 89, Issue 4, April 2001, pp. 467-489.

14. D. Edenfeld, A.B. Kahng, M. Rodgers, and Y. Zorian, "2003 Technology Roadmap for Semiconductors", *IEEE Transactions on Computer*, Volume 37, Number 1, Jan. 2004, pp. 47-56.

15. Michael D'Amour, "Reconfigurable Systems Craft a New Breed of Soft Appliances that Deliver Topnotch Performance", SOCcentral, August, 1, 2007, http://www.soccentral.com/results.asp?CatID=488&EntryID=21328.

16. J.F. Miller, D. Job, and V.K. Vassilev, "Principles in the Evolutionary Design of Digital Circuits Part I", In *Proceedings of Genetic Programming and Evolvable Machines*, Volume 1, Number 3, 2000, pp. 7-35.

17. J.F. Miller, D. Job, and V.K. Vassilev, "Principles in the Evolutionary Design of Digital Circuits Part II", In *Proceedings of Genetic Programming and Evolvable Machines*, Volume 1, Number 3, 2000, pp. 259-288.

18. V.K. Vassilev, D. Job, and J.F. Miller, "Towards the Automatic Design of More Efficient Digital Circuits", In *Proceedings of Evolvable Hardware 2000*, Palo Alto, CA, USA, 2000, pp. 151-160.

19. H. de Gari, "Evolvable Hardware", In *Proceedings of Artificial Neural Nets and Genetic Algorithms*, Innsbuck, Austria, April 1993, pp. 441-449.

20. Daniel Mange, Maxime Goeke, Dominik Madon, André Stauffer, Gianluca Tempesti, and Serge Durand, "Embryonics: A new family of coarse-grained field-programmable gate array with self-repair and self-reproducing properties", Lecture Notes in Computer Science, Volume 1062, Springer Berlin/Heidelberg, pp. 192-220.

21. C. Ortega-Sanchez, and A. Tyrrell, "Fault-Tolerant Systems: The way Biology Does it", In *Proceedings of 23rd Euromicro conference: New Frontiers of Information Technology-Short Contributions*, Budapest, Hungary, 1997, pp. 146-151.

22. John A. Stankovic, *Deadline scheduling for real-time systems: EDF and related algorithms*, Kluwer Academic publishers, 1998.

23. USENET, Comp.realtime: Frequently asked questions, Version 3.6 (May 2002). http://www.faqs.org/faqs/realtime-computing/faq/.

24. S.D. Bruda, and S.G. Akl, ”Real-time Computation: a Formal Definition and its Applications”, In *Proceedings of 15th International Parallel and Distributed Processing Symposium*, San Francisco, CA, April 2001, pp. 1377-1384.

25. J. A. Watlington, and V. Micheal Bove Jr., “Stream-Based Computing and Future Television”, In *Proceedings of 137th SMPTE Technical conference*, New Orleans , USA, September 1995, pp. 69-79.

26. Dennis, J. B., “Data Flow Supercomputers”, *IEEE Transactions of Computer* Volume 13, Issue 11, November 1980, pp. 48-56.

27. Ray Weiss, “CoreConnect: The on-chip bus system”, Electronic Design, Volume 49, Number 7, April, 2001, p. 110.

28. http://www.amba.com/.

29. ITRS. 2005. International Technology Roadmap for Semiconductors – Interconnect. International Technology Roadmap for Semiconductors (ITRS), http://www.itrs.net/Links/2005ITRS/Interconnect2005.pdf.

30. T. Bjerregaard, and S. Mahadevan, “A survey of research and practices of Network-on-chip”, *ACM Computing. Survey*, Volume 38, Issue 1, June, 2006, pp. 1-54.

31. http://www.ocpip.org/.

32. http://www.vsi.org/.

33. OCP-IP, “Open Core Protocol Specification”, Release 2.0, Revision 1.1.1, 2003.

34. R. Hartenstein,” A decade of reconfigurable computing: a visionary retrospective”, In *Proceedings of Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001,* Munich, Germany, March, 2001, pp. 642-649.

35. J. Becker et al., “Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems”, In *Proceedings of FCCM’00*, Napa, CA, USA, April 17-19, 2000, pp. 205-214.

36. X.Tang, et al., “A Compiler Directed Approach to Hiding Configuration Loading Latency in Chameleon Reconfigurable Chips”, In *Proceedings of FPL2000*, August, 2000, Springer-Verlag, Villach, Austria, 2000, pp. 29-38.

37. H. Singh, et al., "MorphoSys: An Integrated Re-configurable Architecture", In *Proceedings of the NATO RTO Symp. on System Concepts and Integration*, Monterey, CA, USA, April 20-22, 1998, pp. 465-481.

38. K. Syano, and T. Shirakawa, "Pleiades: a prototype of inter-processor network generation system", In *Proceedings of I-SPAN* '97 *Third International Symposium on Parallel Architectures*, *Algorithms, and Networks*, Taipei, Taiwan, Dec. 18-20, 1997, pp. 202-206.

39. D. Chen, and J. Rabaey, "PADDI: Programmable arithmetic devices for digital signal processing", In Proceedings of IEEE Workshop on VLSI Signal Processing, November 1990, pp. 240-249.

40. A. K. W. Yeung, J.M. Rabaey, "A Reconfigurable Data-driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms", In *Proceedings of HICSS-26*, Kauai, Hawaii, Jan. 1993, pp. 169-178.

41. J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", In *Proceedings of IEEE FCCM'97*, Napa, April 16-18, 1997, pp. 12-21.

42. R. Kress et al., "A Datapath Synthesis System for the Reconfigurable Datapath Architecture" In *Proceedings of ASP-DAC'95*, Chiba, Japan, Aug. 29 - Sept. 1, 1995, pp. 479-484.

43. U. Nageldinger et al.,"KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures", In *Proceedings of ASP-DAC*, Yokohama, Japan, Jan. 25-28, 2000, pp. 163-168.

44. E. Mirsky, and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", In *Proceedings of IEEE FCCM'96*, Napa, CA, USA, April 17-19, 1996, pp. 157-166.

45. E. Waingold et al., "Baring it all to Software: RAW Machines", *IEEE Transaction of Computer*, September 1997, pp. 86-93.

46. H. Hinkelmann, A. Gunberg, P. Zipf, L. S. Indrusiak, M. Glesner, "Multitasking Support for Dynamically Reconfigurable Systems" In *Proceedings of 2006 International Conference on Field Programmable Logic and Applications* (FPL), Spain, Madrid, 2006, pp. 219-224.

47. H. Kalte, and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems" In *Proceedings of 2005 International Conference on Field Programmable Logic and Applications*, Tampere, Finland, 2005, pp. 223-228.

48. T. Marescaux et al., ”Run-time Support for Heterogeneous Multitasking on Reconfigurable SoCs”, *Integration, The VLSI Journal*, Volume 38, Number 1, Oct. 2004, pp. 107-130.

49. S. Jovanovic, C. Tanougast, S. Weber, ”A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems”, *2007 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, Edinburgh, Scotland, UK, Aug. 2007, pp. 328-334.

50. Huang, I.-Hsuan, Wang, Chih-Chun, Chu, Shih-Min, and Yang, Cheng-Zen, “Function-level Multitasking Interface Design in an Embedded Operating System with Reconfigurable Hardware”, Lecture Notes in Computer Science, Volume 4808 LNCS, Embedded and Ubiquitous Computing - International Conference, EUC 2007, 2007, pp. 45-54.

51. G. Wigley, and D. Kearney, “Research Issues in Operating Systems for Reconfigurable Computing”, In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture* 2002, Las Vegas, USA, June 2002, pp. 10-16.

52. B. Bailey, G. Martin, T. Anderson, *Taxonomies for the Development and Verification of Digital Systems*, New York: Springer Science+Business Media, 2005.

53. Actel Corporation, “IGLOO family FPGAs”, http://www.actel.com/

54. Lattice Semiconductor Corporation, “EC family FPGAs”, http://www.latticesemi.com/

55. Xilinx Incorporated, “Virtex family FPGAs”, http://www.xilinx.com/

56. Altera Corporation, “Stratix family FPGAs”, http://www.altera.com/

57. Xilinx, Datasheet (DS302), “Vitex-4 FPGA Data Sheet”, June 8, 2008, Version 3.3.

58. Xilinx, Datasheet, “XC6200 Field Programmable Gate Arrays”, April 24, 1997, Version 1.10.

59. J. Heron and R. F. Woods, “Architecture Strategies for Implementing Image Processing Algorithms on an XC6200 FPGA”, In I*nternational conference on Field Programmable Logic and Applications*, Glasgow, UK, 1996, pp. 174-184.

60. Xilinx, Application note, XAPP081, “High performance, low area, interpolator design for the XC6200”, 1997, Version 1.0.

61. R.F. Woods, D.W. Trainor, and J. Heron, “Applying an XC6200 to Real-time Image Processing”, *IEEE Design and Test of Computers*, Volume 15 Issue 1, 1998, pp. 30-38.

62. Xilinx, Datasheet (DS003-2), “Virtex 2.5V Field Programmable Gate Arrays”, December 9, 2002, v2.8.1.

63. Xilinx, Datasheet (DS031-2), “Virtex-II Platform FPGAs”, November 5, 2007, v3.5.

64. http://www.xilinx.com/support/documentation/virtex-5.htm/

65. Xilinx, Datasheet, “XC4000 XLA/XV Field programmable Gate Arrays”, 1997, Version 1.10.

66. Usama Malik, “Configuration Encoding Techniques for Fast FPGA Reconfiguration”, Ph.D. Dissertation, The university of New South Wales, June 2006.

67. S. Hauck, T.W. Fry, M.M. Hosler and J.P. Kao, “The Chimaera Reconfigurable Functional Unit”, In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, 1997, pp. 87-96.

68. H. Schmit, “Incremental recognization for pipelined applications”, In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 1997, pp. 47-55.

69. A.B. Ray and P.M. Athanas, “Wormhole run-time reconfiguration”, In *International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, 1997, pp. 79-85.

70. Paul Beckett, Andrew Jennings, “Towards Nanocomputer Architecture”, In *Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, Volume 6, Australian Computer Society Incorporated, NTU, Taiwan, 2002, pp. 141-150.

71. David Manners, Electronics Weekly, January, 1991.

72. Tsugio Makimoto, “Towards the Second Digital Wave”, CX-News, Volume 33, Sony, http://www.sony.net/Products/SC-HP/cx_news/vol33/sideview.html.

73. R. Hartenstein, “The Microprocessor is No Longer General Purpose: Why Reconfigurable Platforms Will Win” In *Proceedings of Second Annual IEEE International Conference on Innovative in Silicon*, Austin, Texas, USA, 1997, pp. 2-12.

74. Kurt Keutzer, et al., *Building ASIPs: The Mescal Methodology*, Springer US, 2005.

75. R. Goodall, D. Fandel, A. Allan,P.Landler, H. R. Huff, “Long-term Productivity Mechanisms of the Semiconductor Industry”, *American electrochemical society: Semiconductor Silicon 2002*, 9th edition, May 2002, pp. 125-144.

76. International Roadmap Committee, *The International Technology Roadmap for Semiconductors,* 2003 Edition.

77. Pil Woo Chun, “Dynamically Reconfigurable Parallel Stream Processor”, Master Thesis, Ryerson University, Toronto, 2004.

78. “FPGA vs. ASIC project cost calculator”, http://www.altera.com/products/devices/cost/cst-cost_step1.jsp

79. Lev Kirischian, Vadim Gurkov, Valeri Kirischian, Irina Terterian, “Multi-parametric optimization of the modular computer architecture”, *International Journal of Technology Policy and Management*, 2006, Vol. 6, No. 3, pp. 327 – 346.

80. Giovanni De Micheli, *Synthesis and Optimization of digital circuits*, Electronics and VLSI Circuits, McGraw-Hill Incorporated, 1994.

81. D. Andrews, D. Niehaus, R. Jidin, “Implementing the thread programming model on hybrid FPGA/CPU computational components”, *Workshop on Embedded Processor Architectures of the international Symposium on Computer Architecture*, Madrid, Spain, 2004, pp. 23-48.

82. A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu, and J. Rabaey, *The Pleiades architecture*, Application of Programmble DSPs in Mobile Communications, Wiley, 2002.

83. M. H. Sunwoo, Ong Soowan, Ahn Byungdug, Lee Kyoungwoo, “Design and implementation of a parallel image processor chip for a SIMD array processor”, *In proceedings of Application Specific Array Processors*, Strasbourg, France, July, 24-26, 1995, pp. 66-75.

84. Laurie J. Flynn, “Intel Halts Development of 2 New Microprocessors”, New York Times, Published in May, 8, 2004.

85. Michael J. S. Smith, *Application-specific Integrated Circuits (ASICs … the book)*, Addison-Wesley Publishing Company, VLSI Design Series, June, 1997.

86. Edward Yourdon, *Modern Structured Analysis*, Yourdon Press Computing Series, New Jersey: Prentice hall, 1989.

87. Edward A. Lee, and David G. Messerchmitt, “Synchronous Data Flow”, In *Proceedings of the IEEE*, volume. 75, number. 9, September 1987, pp. 1235-1245.

88. David Harel, “Statecharts: a visual formalism for complex systems”, Science of Computer Programming, Volume 8, Issues 3, pp. 231 – 274, June, 1987.

89. Paul T. Ward, Stephen J. Mellor, *Structured Development for Real-time Systems*, New York: Yourdon Press, 1985.

90. J. Shi, A.Randhar, and D. Bhatia, “Macro Block Based FPGA Floorplanning”, In *Proceedings of tenth international conference on VLSI Design*, Hyderabad, India, January, 4-7, 1997, pp. 21-26.

91. Habib Youssef, Khalid Al-Farra, Sadiq M. Sait, "Timing influenced force directed floorplanning," *European Design Automation Conference with EURO-VHDL*, volume 0, Number 0, European Design Automation Conference with EURO-VHDL '95, Brighton, Great Britain, 1995, pp. 156-161.

92. Cypress Semiconductor Corporation, “PLD devices”, http://www.cypress.com/

93. Atmel Corporation, “AT40KAL family FPGAs”, http://www.atmel.com/

94. Quicklogic Corporation, “PolarPro devices”, http://www.quicklogic.com/

95. Xilinx, User Guide (UG071), “Virtex-4 FPGA Configuration User Guide”, v1.10,April 8, 2008.

96. Jamin Islam, Pil Woo Chun, W. James MacLean and Lev Kirischian, “Lowering Power Consumption Using Run-time Reconfiguration for Stereo Rectification”, *In 21st Canadian Conference on Electrical and Computer Engineering*, Niagara fall, Canada, 2008, pp. 1693-1698.

97. Gary Yeap, *Practical Low Power Digital VLSI Design*, Kluwer Academic Publishers, 1998.

98. Xilinx Incorporated, “Application Note 059: Gate Count Capacity Metrics for FPGAs”, February, 1997.

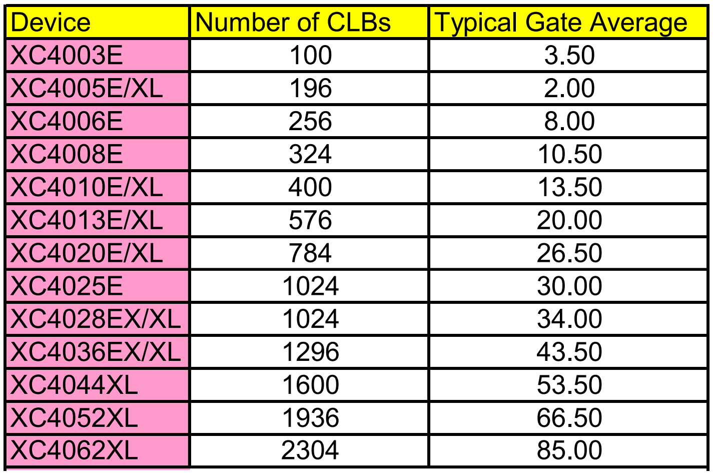
Appendix A

Cost Estimation

A.1 Estimation of Unit Cost

The following data shown in Table **A-1** is taken from Xilinx application note [**98**]. From this data set it is possible to extract what is the typical gate average for a given device when the number of CLBs is known. Table **A-1** shows the typical gate average for XC4000 FPGA – internal memory is also included in the gate number estimation.

Table **A-1**: XC4000 FPGA typical gate average (internal memory is included)



Device	Number of CLBs	Typical Gate Average
XC4003E	100	3.50
XC4005E/XL	196	2.00
XC4006E	256	8.00
XC4008E	324	10.50
XC4010E/XL	400	13.50
XC4013E/XL	576	20.00
XC4020E/XL	784	26.50
XC4025E	1024	30.00
XC4028EX/XL	1024	34.00
XC4036EX/XL	1296	43.50
XC4044XL	1600	53.50
XC4052XL	1936	66.50
XC4062XL	2304	85.00

Furthermore to estimate the gate average for larger devices such as Xilinx Virtex-4 devices, the graph is plotted to find the relationship between the number of CLBs and the gate average. Figure **A-1** illustrates the relationship that found to be linear with the slope of 36 [gate/CLB]. Typically the given assumption that the Xilinx FPGAs have 36 gate used per a CLB given, is assumed – including unpopulated areas of logic and memory.

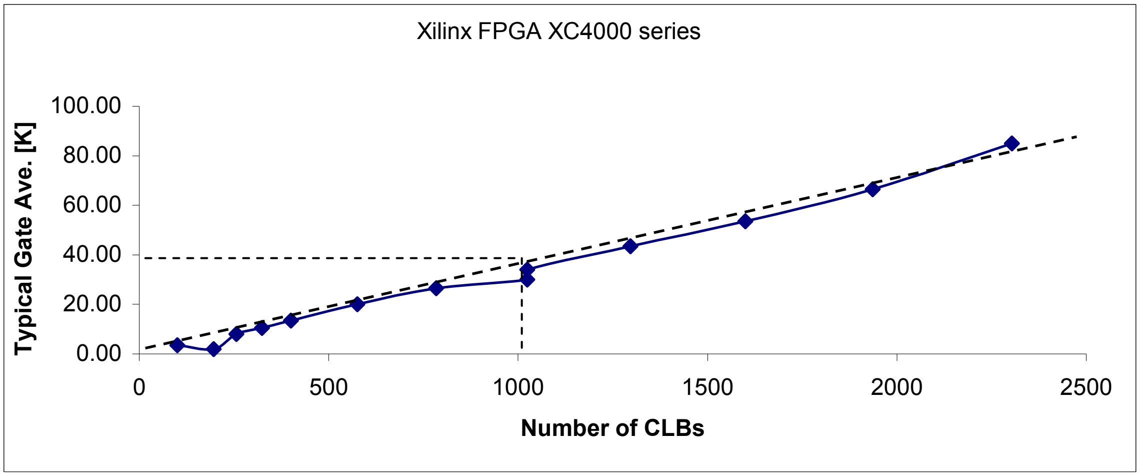


Figure **A-1**: The relationship between the number of CLBs and the gate average

Particularly to extract the average gate usage of the target device, Xilinx Virtex-4 XC4VLX160, the number of CLBs (i.e., 152064 for the target device) is applied with the above rule (×36). Then, the typical gate for XC4VLX160 is calculated to be 5474 [Kgate].

A.2 Estimation of Software Tools Cost

The software tool cost between FPGA and ASIC is quite different due to the verifications and tests steps involved in the ASIC design process. This section tries to discover the actual costs by exploring the details of design processes. Table **A-2** shows the related software tools and associated hardware platform to carry their executions.

Table **A-2**: Software Tool Usage

Vendor/Tool	Per Seat Usage	Program Usage
Leading ASIC Synthesis Tool	0.4	
Optimization	0.4	
Cores with Test Benches	0.4	
ASIC Schematic Debugging Tool	0.4	
ASIC Static Timing Analysis	0.25	
ATPG Test Pattern Generator		1
RTL Simulator	2	
Testbench Automation	0.5	
Memory Tools		
Memory Design Software		1
Memory Simulation Software		1
LINT, Code, Coverage, Revision Control, ...	0	
Low-end Workstation	1	
High-end Server	0.2	

Depending on their involvements in the process the ‘per seat usage’ is determined. If it is one time purchase then, the ‘program usage’ is quoted as 1 instead.

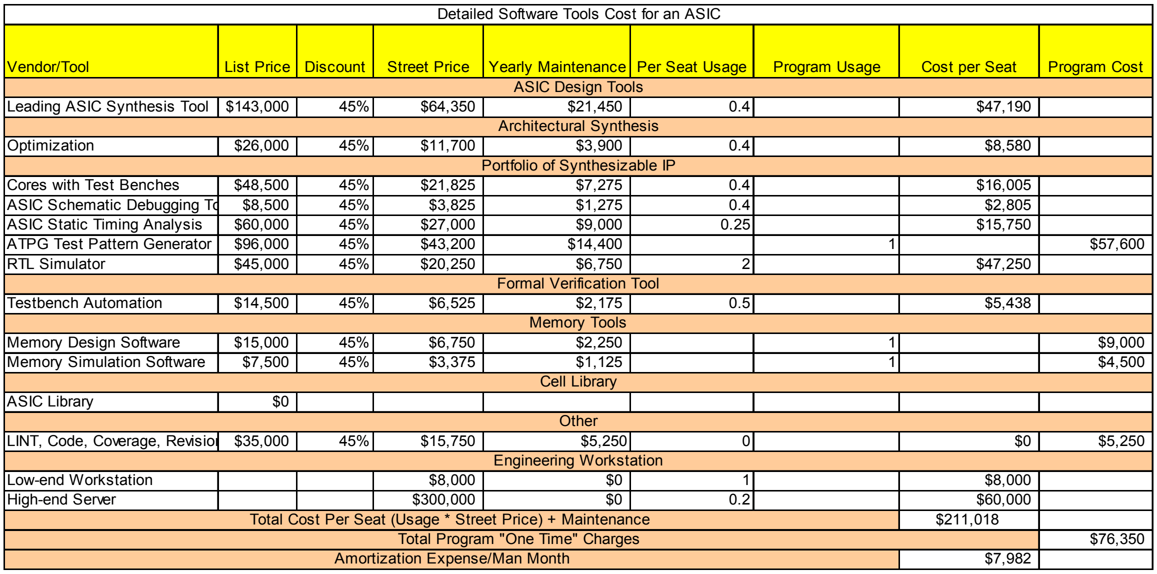
Table **A-3** shows the detailed software tools cost for an FPGA.

Table **A-3**: Detailed Software Tools Cost for an FPGA

Detailed Software Tools Cost for an FPGA						
Vendor/Tool	List Price	Discount	Street Price	Yearly Maintenance	Per Seat Usage	Cost Per Seat
High-end FPGA Synthesis Tool						
	$25,000	45%	$11,250	$2,625	0.4	$7,125
High-end Verilog or VHDL Tool						
VHDL or Verilog Simulator	$15,000	45%	$6,750	$1,575	1	$8,325
Engineering Workstation						
Low-end Workstation				$8,000	1	$8,000
High-end Server				$300,000	0.1	$30,000
Total Cost Per Seat (Usage * Street Price) + Maintenance						$53,450
Amortization Expense/Man Month						$1,485

Table **A-4** shows the detailed software tools cost for an ASIC.

Table **A-4**: Detailed Software Tools Cost for an ASIC

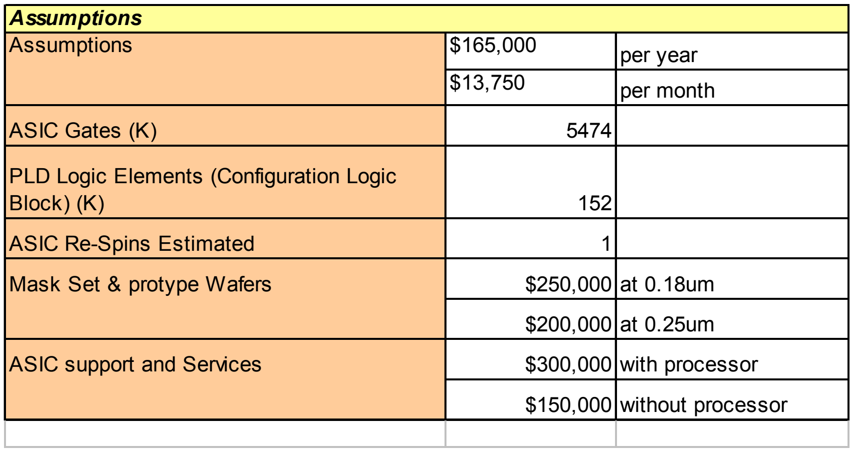


Detailed Software Tools Cost for an ASIC								
Vendor/Tool	List Price	Discount	Street Price	Yearly Maintenance	Per Seat Usage	Program Usage	Cost per Seat	Program Cost
ASIC Design Tools								
Leading ASIC Synthesis Tool	$143,000	45%	$64,350	$21,450	0.4		$47,190	
Architectural Synthesis								
Optimization	$26,000	45%	$11,700	$3,900	0.4		$8,580	
Portfolio of Synthesizable IP								
Cores with Test Benches	$48,500	45%	$21,825	$7,275	0.4		$16,005	
ASIC Schematic Debugging To	$8,500	45%	$3,825	$1,275	0.4		$2,805	
ASIC Static Timing Analysis	$60,000	45%	$27,000	$9,000	0.25		$15,750	
ATPG Test Pattern Generator	$96,000	45%	$43,200	$14,400		1		$57,600
RTL Simulator	$45,000	45%	$20,250	$6,750	2		$47,250	
Formal Verification Tool								
Testbench Automation	$14,500	45%	$6,525	$2,175	0.5		$5,438	
Memory Tools								
Memory Design Software	$15,000	45%	$6,750	$2,250		1		$9,000
Memory Simulation Software	$7,500	45%	$3,375	$1,125		1		$4,500
Cell Library								
ASIC Library	$0							
Other								
LINT, Code, Coverage, Revisio	$35,000	45%	$15,750	$5,250	0		$0	$5,250
Engineering Workstation								
Low-end Workstation			$8,000	$0	1		$8,000	
High-end Server			$300,000	$0	0.2		$60,000	
Total Cost Per Seat (Usage * Street Price) + Maintenance							$211,018	
Total Program "One Time" Charges								$76,350
Amortization Expense/Man Month							$7,982	

A.3 Development Costs

Table **A-5** numerates the assumptions that are made to estimate the development costs of FPGAs and ASICs.

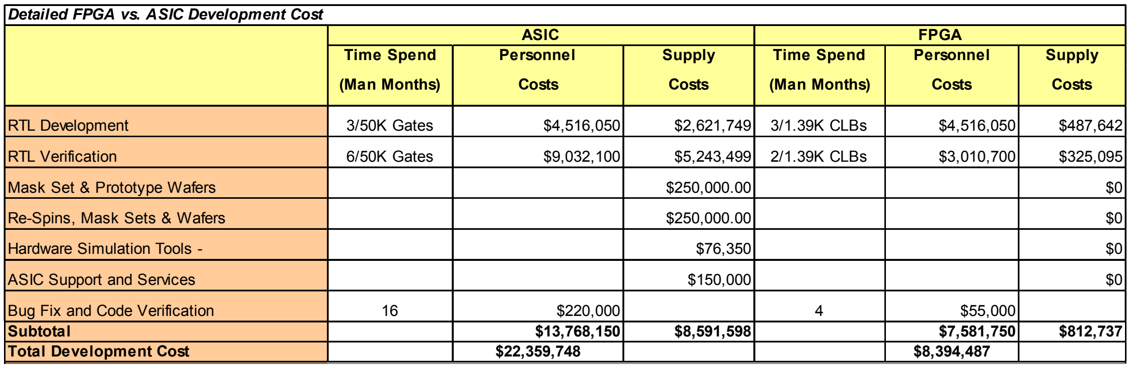
Table **A-5**: Assumptions given for the development costs



Assumptions		
Assumptions	$165,000	per year
	$13,750	per month
ASIC Gates (K)	5474	
PLD Logic Elements (Configuration Logic Block) (K)	152	
ASIC Re-Spins Estimated	1	
Mask Set & protype Wafers	$250,000	at 0.18um
	$200,000	at 0.25um
ASIC support and Services	$300,000	with processor
	$150,000	without processor

Table **A-6** summaries the detailed development costs of FPGA and ASIC

Table **A-6**: Detailed FPGA and ASIC Development Costs



Detailed FPGA vs. ASIC Development Cost

	ASIC			**FPGA**		
	Time Spend **(Man Months)**	**Personnel** **Costs**	**Supply** **Costs**	**Time Spend** **(Man Months)**	**Personnel** **Costs**	**Supply** **Costs**
RTL Development	3/50K Gates	$4,516,050	$2,621,749	3/1.39K CLBs	$4,516,050	$487,642
RTL Verification	6/50K Gates	$9,032,100	$5,243,499	2/1.39K CLBs	$3,010,700	$325,095
Mask Set & Prototype Wafers			$250,000.00			$0
Re-Spins, Mask Sets & Wafers			$250,000.00			$0
Hardware Simulation Tools -			$76,350			$0
ASIC Support and Services			$150,000			$0
Bug Fix and Code Verification	16	$220,000		4	$55,000	
Subtotal		**$13,768,150**	**$8,591,598**		**$7,581,750**	**$812,737**
Total Development Cost		**$22,359,748**			**$8,394,487**	

Appendix B

Experimental Details

B.1 Estimation of Silicon Costs

Because the service hardware that accounts for the reconfiguration overhead, $ov_{reconfiguration}$ is constructed outside of a *run-time reconfigurable* FPGA and can be reused numerous times, one assumes that the $ov_{reconfiguration}$ is amortized over the life time of the device[40]. The ratio of reconfigurable functions, $r_{reconfigurable}$ is the ratio between static or reconfigurable tasks in the workload. It seems that all the functions given in the workload are reconfigurable tasks. However, if you take a close look at Figure **6-26**, the area designated for the static tasks inevitably occupies a quarter of the chip that is located on the upper left side of the chip due to their I/O placements where the VGA ports associated with reconfigurable functions are located at the bottom half of the chip. This arrangement of I/O ports allows the reconfigurable functions to be located anywhere in the bottom half of the device according to Figure **6-24**. Because the area needed for reconfigurable modules is about a third of static area, the ratio of reconfigurable functions can be estimated as 33%.

[40] The $ov_{reconfiguration}$ is a part of the system costs that can be easily compared at the system level in terms of dollar value (e.g., $) rather than silicon costs of the device.

The parametric values that determine the silicon costs are given as: A_{common} = 0.16, $ov_{multiplexer}$ = 0.17, $r_{reconfigurable}$ = 0.33, $ov_{reconfiguration}$ = 0 and $s_{utilization}$ = 0.32. Using Eq. **5-4**, the silicon cost of a *non-reconfigurable* FPGA, sc_{fixed} can be obtained as 56(1-0.16) + 56^2*0.17/4 = 180.32. $sc_{reconfigured}$ is found to be 100. The multiplexing overhead needed to combine the functions of the systems,

B.2 How to calculate c_{LUT}

The constant to convert the unit cost of the reconfiguration functions into 4-input LUTs, c_{LUT} is obtained by averaging the values found in the implementation. These values are shown in Table **B-1**.

Table **B-1**: C_{LUT} values for the multi-task and multi-mode video processors

The number of modes	The logic required by ov_multiplexer [4-input LUT]	Total logic required by the fixed design	The logic occupied by static functions	C_LUT [LUT/unit cost]
4	302	2904	358	140.25
8	565	7014	358	190.34375
12	962	8103	358	141.3125
			Average	157.3020833

B.3 Examples of LED counter

pr_test_led.vhd Wed Feb 18 02:24:55 2009

```
----------------------------------------------------------------------
-------------
```

```
-- Company:
-- Engineer:
--
-- Create Date: 15:04:08 12/08/2008
-- Design Name:
-- Module Name: pr_test_led - Behavioural
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pr_test_led is
Port ( clk : in STD_LOGIC;
reset : in STD_LOGIC;
led_right : out STD_LOGIC_VECTOR (1 downto 0));
end pr_test_led;

architecture Behavioural of pr_test_led is

component BUFGP
port (
I : in std_logic;
O : out std_logic);
end component;

component static is
port(
clk : in std_logic;
reset : in std_logic;
rst : out std_logic);
end component;

component reconfig_led_right is
port(
clk : in std_logic;
reset : in std_logic;
O : out std_logic_vector(1 downto 0));
end component;

component busmacro_xc4v_l2r_async_narrow is
port(
input0 : in std_logic;
```

```
input1 : in std_logic;
input2 : in std_logic;
input3 : in std_logic;
input4 : in std_logic;
input5 : in std_logic;
input6 : in std_logic;
input7 : in std_logic;

output0 : out std_logic;
output1 : out std_logic;
output2 : out std_logic;
output3 : out std_logic;
output4 : out std_logic;
output5 : out std_logic;
output6 : out std_logic;
output7 : out std_logic);
end component;

component busmacro_xc4v_r2l_async_narrow is
port(
input0 : in std_logic;
input1 : in std_logic;
input2 : in std_logic;
input3 : in std_logic;
input4 : in std_logic;
input5 : in std_logic;
input6 : in std_logic;
input7 : in std_logic;

output0 : out std_logic;
output1 : out std_logic;
output2 : out std_logic;
output3 : out std_logic;
output4 : out std_logic;
output5 : out std_logic;
output6 : out std_logic;
output7 : out std_logic);
end component;


signal reset_temp : std_logic;
signal clk_int : std_logic;

signal rst : std_logic;
signal rst_imod : std_logic;
signal en_imod : std_logic;
signal en : std_logic;
signal led_right_omod : std_logic_vector (1 downto 0);

begin

bufgp_0 : BUFGP
port map (
I => clk,
O => clk_int
```

```
);

static_inst: static
port map (
clk => clk_int,
reset => reset,
rst => rst
);

reconfig_led_right_inst: reconfig_led_right
port map (
clk => clk_int,
reset => rst_imod,
O => led_right_omod
);

macro_1: busmacro_xc4v_l2r_async_narrow
port map(
input0 => rst,
input1 => '1',
input2 => '1',
input3 => '1',
input4 => '1',
input5 => '1',
input6 => '1',
input7 => '1',


output0 => rst_imod,
output1 => open,
output2 => open,
output3 => open,
output4 => open,
output5 => open,
output6 => open,
output7 => open
);

macro_2: busmacro_xc4v_l2r_async_narrow
port map(
input0 => led_right_omod(0),
input1 => led_right_omod(1),
input2 => '1',
input3 => '1',
input4 => '1',
input5 => '1',
input6 => '1',
input7 => '1',


output0 => led_right(0),
output1 => led_right(1),
output2 => open,
output3 => open,
output4 => open,
output5 => open,
output6 => open,
output7 => open
);


end Behavioural;
```

static.vhd Wed Feb 18 02:26:37 2009

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 19:43:15 12/08/2008
-- Design Name:
-- Module Name: static - Behavioural
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity static is
Port ( clk : in std_logic;
reset : in STD_LOGIC;
rst : out STD_LOGIC);
end static;

architecture Behavioural of static is

begin

process(clk)
begin
if(clk = '1' and clk'event) then
rst <= reset;
end if;
end process;
end Behavioural;


```

reconfig_led_right.vhd Wed Feb 18 02:36:06 2009

```vhdl
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 18:35:52 12/08/2008
-- Design Name:
-- Module Name: reconfig_led_right - Behavioural
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity reconfig_led_right is
Port (
clk : in std_logic;
reset : in std_logic;
O : out STD_LOGIC_VECTOR (1 downto 0));
end reconfig_led_right;

architecture Behavioural of reconfig_led_right is

signal GND : std_logic_vector(1 downto 0) := "00";
signal VCC : std_logic_vector(1 downto 0) := "11";

begin

process(clk, reset)
begin
if(clk = '1' and clk'event) then
if(reset = '0') then
O <= "00";
else
O <= VCC;
end if;
end if;
end process;

end Behavioural;
```

Appendix C

Power Estimation Details

Even though the obtained results show sufficient evidences that the proposed architecture synthesis methodology produces a cost-effective design solution, there are a few areas that the analysis of reconfigurable system can be improved. Since the architecture synthesis methodology assumes that the network level architecture takes care of the communication interfaces for VHCs, the overhead caused by interfaces is not counted as the reconfiguration overhead. The currently available programmable devices (e.g., Xilinx Virtex-4 FPGAs) embody all the routings including the global and local routing as a part of reconfigurable architecture (i.e., micro architecture). Additionally the full estimation of power consumption for multiplexed functionalities could not be calculated correctly. Since a big portion of power consumption in the design of multiplexed functionalities is caused by complex routing resources, it is crucial to include the power consumption of these resources for power estimation. However, due to the lack of appropriate design support from the programmable device vendor (e.g., Xilinx Design Language) the routing resources could not be accounted for power estimation. Thus, the power consumption for multiplexed functionalities appears smaller than the real case.

C.1 Example of Power Estimation Testbench

testbench.vhd Wed Feb 18 03:06:05 2009

```vhdl

--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 15:36:01 01/25/2009
-- Design Name: top
-- Module Name: C:/users/peter/PhD_demo_stereo_cam_4vgas_v2/pr/Flat/testbench.vhd
-- Project Name: stereo_cam
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: top
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test. Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY testbench_vhd IS
END testbench_vhd;

ARCHITECTURE behaviour OF testbench_vhd IS

-- Component Declaration for the Unit Under Test (UUT)
COMPONENT top
PORT(
cam_data : IN std_logic_vector(7 downto 0);
clk : IN std_logic;
cam_hsync : IN std_logic;
cam_vsync : IN std_logic;
pix_sync : IN std_logic;
SRAM_B_D : INOUT std_logic_vector(15 downto 0);
SRAM_A_D : INOUT std_logic_vector(15 downto 0);
txrx : OUT std_logic_vector(3 downto 0);
dac_clk : OUT std_logic;
SRAM_B_BW : OUT std_logic_vector(1 downto 0);
SRAM_A_ADSC_N : OUT std_logic;
SRAM_B_ADSC_N : OUT std_logic;
SRAM_A_MODE : OUT std_logic;
SRAM_A_ADV_N : OUT std_logic;
SRAM_B_ADV_N : OUT std_logic;
SRAM_A_ZZ : OUT std_logic;
SRAM_B_ZZ : OUT std_logic;
SRAM_B_OE_N : OUT std_logic;
SRAM_A_CE : OUT std_logic;
SRAM_A_ADDR : OUT std_logic_vector(19 downto 0);
SRAM_B_CE : OUT std_logic;
SRAM_B_ADDR : OUT std_logic_vector(19 downto 0);
SRAM_A_OE_N : OUT std_logic;
SRAM_A_BW : OUT std_logic_vector(1 downto 0);
SRAM_B_MODE : OUT std_logic;
SRAM_B_GW_N : OUT std_logic;
SRAM_A_GW_N : OUT std_logic;
SRAM_B_ADSP_N : OUT std_logic;
SRAM_A_ADSP_N : OUT std_logic;
sram_b_clk : OUT std_logic;
sram_a_clk : OUT std_logic;
vsync : OUT std_logic;
hsync : OUT std_logic;
vga1_r : OUT std_logic_vector(7 downto 0);
vga2_r : OUT std_logic_vector(7 downto 0);
vga3_r : OUT std_logic_vector(7 downto 0);
vga4_r : OUT std_logic_vector(7 downto 0);
vga1_g : OUT std_logic_vector(7 downto 0);
vga2_g : OUT std_logic_vector(7 downto 0);
vga3_g : OUT std_logic_vector(7 downto 0);
vga4_g : OUT std_logic_vector(7 downto 0);
vga1_b : OUT std_logic_vector(7 downto 0);
vga2_b : OUT std_logic_vector(7 downto 0);
vga3_b : OUT std_logic_vector(7 downto 0);
vga4_b : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

--Inputs
SIGNAL clk : std_logic := '0';
SIGNAL cam_hsync : std_logic := '0';
SIGNAL cam_vsync : std_logic := '0';
SIGNAL pix_sync : std_logic := '0';
SIGNAL cam_data : std_logic_vector(7 downto 0) := (others=>'0');

--BiDirs
SIGNAL SRAM_B_D : std_logic_vector(15 downto 0);
SIGNAL SRAM_A_D : std_logic_vector(15 downto 0);

--Outputs
SIGNAL txrx : std_logic_vector(3 downto 0);
 SIGNAL dac_clk : std_logic;
 SIGNAL SRAM_B_BW : std_logic_vector(1 downto 0);
 SIGNAL SRAM_A_ADSC_N : std_logic;
 SIGNAL SRAM_B_ADSC_N : std_logic;
 SIGNAL SRAM_A_MODE : std_logic;
 SIGNAL SRAM_A_ADV_N : std_logic;
 SIGNAL SRAM_B_ADV_N : std_logic;
 SIGNAL SRAM_A_ZZ : std_logic;
 SIGNAL SRAM_B_ZZ : std_logic;
 SIGNAL SRAM_B_OE_N : std_logic;
 SIGNAL SRAM_A_CE : std_logic;
 SIGNAL SRAM_A_ADDR : std_logic_vector(19 downto 0);
 SIGNAL SRAM_B_CE : std_logic;
 SIGNAL SRAM_B_ADDR : std_logic_vector(19 downto 0);
 SIGNAL SRAM_A_OE_N : std_logic;
 SIGNAL SRAM_A_BW : std_logic_vector(1 downto 0);
 SIGNAL SRAM_B_MODE : std_logic;
 SIGNAL SRAM_B_GW_N : std_logic;
 SIGNAL SRAM_A_GW_N : std_logic;
 SIGNAL SRAM_B_ADSP_N : std_logic;
 SIGNAL SRAM_A_ADSP_N : std_logic;
 SIGNAL sram_b_clk : std_logic;
 SIGNAL sram_a_clk : std_logic;
 SIGNAL vsync : std_logic;
 SIGNAL hsync : std_logic;
 SIGNAL vga1_r : std_logic_vector(7 downto 0);
 SIGNAL vga2_r : std_logic_vector(7 downto 0);
 SIGNAL vga3_r : std_logic_vector(7 downto 0);
 SIGNAL vga4_r : std_logic_vector(7 downto 0);
 SIGNAL vga1_g : std_logic_vector(7 downto 0);
 SIGNAL vga2_g : std_logic_vector(7 downto 0);
 SIGNAL vga3_g : std_logic_vector(7 downto 0);
 SIGNAL vga4_g : std_logic_vector(7 downto 0);
 SIGNAL vga1_b : std_logic_vector(7 downto 0);
 SIGNAL vga2_b : std_logic_vector(7 downto 0);
 SIGNAL vga3_b : std_logic_vector(7 downto 0);
 SIGNAL vga4_b : std_logic_vector(7 downto 0);

 BEGIN

 -- Instantiate the Unit Under Test (UUT)
 uut: top PORT MAP(
 txrx => txrx,
 dac_clk => dac_clk,
 cam_data => cam_data,
 SRAM_B_D => SRAM_B_D,
 SRAM_B_BW => SRAM_B_BW,
 SRAM_A_D => SRAM_A_D,
 SRAM_A_ADSC_N => SRAM_A_ADSC_N,
 SRAM_B_ADSC_N => SRAM_B_ADSC_N,
 SRAM_A_MODE => SRAM_A_MODE,
 SRAM_A_ADV_N => SRAM_A_ADV_N,
 SRAM_B_ADV_N => SRAM_B_ADV_N,
 SRAM_A_ZZ => SRAM_A_ZZ,
 SRAM_B_ZZ => SRAM_B_ZZ,
 SRAM_B_OE_N => SRAM_B_OE_N,
 SRAM_A_CE => SRAM_A_CE,
 SRAM_A_ADDR => SRAM_A_ADDR,
 SRAM_B_CE => SRAM_B_CE,
 SRAM_B_ADDR => SRAM_B_ADDR,
 SRAM_A_OE_N => SRAM_A_OE_N,
 SRAM_A_BW => SRAM_A_BW,
 SRAM_B_MODE => SRAM_B_MODE,
 SRAM_B_GW_N => SRAM_B_GW_N,
 SRAM_A_GW_N => SRAM_A_GW_N,
 SRAM_B_ADSP_N => SRAM_B_ADSP_N,
 SRAM_A_ADSP_N => SRAM_A_ADSP_N,
 sram_b_clk => sram_b_clk,
 sram_a_clk => sram_a_clk,
 clk => clk,
 cam_hsync => cam_hsync,
 cam_vsync => cam_vsync,
 pix_sync => pix_sync,
 vsync => vsync,
 hsync => hsync,
 vga1_r => vga1_r,
 vga2_r => vga2_r,
 vga3_r => vga3_r,
 vga4_r => vga4_r,
 vga1_g => vga1_g,
 vga2_g => vga2_g,
 vga3_g => vga3_g,
 vga4_g => vga4_g,
 vga1_b => vga1_b,
 vga2_b => vga2_b,
 vga3_b => vga3_b,
 vga4_b => vga4_b
 );

 -- 100MHz main clock simulation
 contant main_clk : time := 5 ns;
 clock_gen : process is
 begin
 clk <= '0' after main_clk, '1' after 2 * main_clk;
 wait for 2*main_clk;
 end process clock_gen;

 -- 12MHz pix_sync clock generation
 pix_sync_gen : PROCESS(clk)
 variable pix_sync_flag : std_logic_vector(3 downto 0);
 BEGIN
 if(clk'event and clk = '1') then
 pix_sync_flag := pix_sync_flag + 1;
 if(pix_sync_flag = "1011") then
 pix_sync_flag := (others =>'0');
 pix_sync <= '0';
 elsif(pix_sync_flag = "0101") then
 pix_sync <= '1';
 end if;
 end process pix_sync_gen;
```

```vhdl

-- pix_hsync strobe generation
pix_hsync_gen : process(pix_sync)
variable pix_hsync_cnt : std_logic_vector(10 downto 0);
variable pix_hsync_flag : std_logic;
begin
if(pix_sync'event and pix_sync = '1') then
pix_hsync_cnt := pix_hsync_cnt + 1;
if(pix_hsync_cnt = "00000000000" and pix_hsync_flag = '1') then
pix_hsync <= '1';
pix_hsync_flag <= '0';
elsif(pix_hsync_cnt = "01010010011") then
pix_hsync_cnt := (others => '0');
pix_hsync <= '1';
pix_hsync_flag <= '1';
end if;
end if;
end process pix_hsync_gen;

-- pix_vsync strobe generation
pix_vsync_gen : process(pix_hsync)
variable pix_vsync_cnt : std_logic_vector(9 downto 0);
variable pix_vsync_flag : std_logic;
begin
if(pix_hsync'event and pix_hsync = '1') then
pix_vsync_cnt := pix_vsync_cnt + 1;
if(pix_vsync_cnt = "0000000000" and pix_vsync_flag = '1') then
pix_vsync <= '1';
pix_vsync_flag <= '0';
elsif(pix_vsync_cnt = "1000001100") then
pix_vsync_cnt := (others => '0');
pix_vsync <= '1';
pix_vsync_flag <= '1';
end if;
end if;
end process pix_vsync_gen;

--cam_data generator PRBS (Pseudo Random Binary Sequence)
cam_data_gen : process(pix_sync)
variable reg : std_logic_vector(7 downto 0);
begin
if(pix_sync'event and pix_sync = '1') then
reg(7) := reg(6);
reg(6) := reg(5);
reg(5) := reg(4);
reg(4) := reg(3);
reg(3) := reg(2);
reg(2) := reg(1);
reg(1) := reg(0);
reg(0) := (reg(7) xor reg(6)) xor reg(5);
end if;
cam_data <= reg;
end process cam_data_gen;
END;
```

C.2 Examples of Video Processors

video_processor_v1.vhd Wed Feb 18 03:16:50 2009

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity video_processor_v1 is
Port ( CLOCK : in std_logic;
vsync, hsync : in std_logic;
DATA_RD1 : in std_logic_vector(15 downto 0);
DATA_VGA1_R : out std_logic_vector(7 downto 0);
DATA_VGA1_G : out std_logic_vector(7 downto 0);
DATA_VGA1_B : out std_logic_vector(7 downto 0)
);
end video_processor_v1;

architecture Behavioural of video_processor_v1 is

component ramb512_32
port (
clka: IN std_logic;
dina: IN std_logic_VECTOR(31 downto 0);
addra: IN std_logic_VECTOR(8 downto 0);
wea: IN std_logic_VECTOR(0 downto 0);
douta: OUT std_logic_VECTOR(31 downto 0));
end component;

component ramb512_16
port (
clka: IN std_logic;
dina: IN std_logic_VECTOR(15 downto 0);
addra: IN std_logic_VECTOR(8 downto 0);
wea: IN std_logic_VECTOR(0 downto 0);
douta: OUT std_logic_VECTOR(15 downto 0));
end component;

subtype counter is std_logic_vector(11 downto 0);
-- it is based on 100MHz clock
constant B : natural := 377; -- horizontal blank: 3.77 us
constant C : natural := 189; -- front guard: 1.89 us
constant D : natural := 2517; -- horizontal columns: 25.17 us
constant E : natural := 94; -- rear guard: 0.94 us
constant A : natural := B + C + D + E; -- one horizontal sync cycle: 31.77 us

constant P : natural := 2; -- vertical blank: 64 us
constant Q : natural := 32; -- front guard: 1.02 ms
constant R : natural := 480; -- vertical rows: 15.25 ms
constant S : natural := 11; -- rear guard: 0.35 ms
constant O : natural := P + Q + R + S; -- one vertical sync cycle: 16.6 ms

constant threshold : std_logic_vector(7 downto 0) := "00011111";
-- Signals latching pixel data read from SRAM (current line) and SRAMB (previous line)
signal button : std_logic;
signal data16c_1 : std_logic_vector(15 downto 0);
signal data16p_1 : std_logic_vector(15 downto 0);
signal data16c_2 : std_logic_vector(15 downto 0);
signal data16p_2 : std_logic_vector(15 downto 0);
-----------------------------------------------------------------------------------------
-- FOR VGA
---+-----+-----+-----+
-- | p11 | p12 | p13 |
-- +-----+-----+-----+
-- | p21 | p22 | p23 |
-- +-----+-----+-----+
-----Signals for VGA#1-------------------------------------------------------------------
signal c1_p11, c1_p12, c1_p13 : std_logic_vector(7 downto 0); -- these registers hold 8-
-- previous line pixel values
signal c1_p21, c1_p22, c1_p23 : std_logic_vector(7 downto 0); -- these registers hold 8-
-- current line pixel values
-----Signals for VGA#2-------------------------------------------------------------------
signal c2_p11, c2_p12, c2_p13 : std_logic_vector(7 downto 0); -- these registers hold 8-
-- previous line pixel values
signal c2_p21, c2_p22, c2_p23 : std_logic_vector(7 downto 0); -- these registers hold 8-
-- current line pixel values
-----------------------------------------------------------------------------------------
signal c1_red22, c1_green22, c1_blue22, c1_green22_pre : std_logic_vector(7 downto 0);
signal c1_red23, c1_green23, c1_blue23, c1_green23_pre : std_logic_vector(7 downto 0);
-----------------------------------------------------------------------------------------
signal c2_red22, c2_green22, c2_blue22 : std_logic_vector(7 downto 0);
signal c2_red23, c2_green23, c2_blue23 : std_logic_vector(7 downto 0);
```

```
-----------------------------------------------------------------------------------------

signal y : std_logic_vector(2 downto 0);
signal edge : std_logic_vector(7 downto 0);


signal pix_flag : std_logic_vector(3 downto 0);
signal din1, din2: std_logic_vector(31 downto 0);
signal din3, din4: std_logic_vector(15 downto 0);
signal addr1, addr2, addr3, addr4 : std_logic_vector(8 downto 0);
signal dout1, dout2 : std_logic_vector(31 downto 0);
signal dout3, dout4 : std_logic_vector(15 downto 0);
signal we1, we2, we3, we4 : std_logic_vector(0 downto 0);
signal pix_flag2 : std_logic;
signal pix_addr : std_logic_vector(18 downto 0);
signal hsync_flag : std_logic;
begin


ram512_32_inst2 : component ramb512_16
port map(
clka => CLOCK,
dina => din3,
addra => addr3,
wea => we3,
douta => dout3);


ram512_32_inst3 : component ramb512_16
port map(
clka => CLOCK,
dina => din4,
addra => addr4,
wea => we4,
douta => dout4);

process(CLOCK)
variable vertical, horizontal : counter; -- define counters
begin
if (CLOCK'event and CLOCK = '1') then

if(vsync = '0') then
vertical := (others => '0');
hsync_flag <= '0';
else
if(hsync = '0') then
horizontal := (others => '0');
if(hsync_flag = '0') then
vertical := vertical + 1;
hsync_flag <= '1';
end if;
else
horizontal := horizontal + 1;
hsync_flag <= '0';
```

```
end if;
end if;
end if;
pix_addr <= vertical(8 downto 0) & horizontal(11 downto 2);
end process;

process(CLOCK)
begin
if (CLOCK'event and CLOCK = '0') then
if(pix_addr(0) = '0') then
pix_flag2 <= '0';
pix_flag <= pix_flag + 1;
if(pix_flag = "0000") then
addr3 <= pix_addr(9 downto 1);
addr4 <= pix_addr(9 downto 1);
elsif(pix_flag = "0011") then
if(pix_addr(18 downto 10) >"000000000" and pix_addr(18 downto 10) <
"111100000" and
pix_addr(9 downto 0) >"0000000000" and pix_addr(9 downto 0)
<"101000000
then
data16c_1 <= data_rd1;
else
data16c_1 <= (others =>'0');
end if;
end if;
else
pix_flag <= "0000";
if(pix_flag = "0000") then
pix_flag2 <= '0';
else
pix_flag2 <= '1';
end if;
end if;
end if;
end process;

process(CLOCK)
begin
if (CLOCK'event and CLOCK = '0') then
if pix_flag2 = '1' then
-----Latching data for VGA#1---------------
c1_p22 <= data16c_1(7 downto 0);
c1_p23 <= data16c_1(15 downto 8);
c1_p21 <= c1_p23;
c1_p12 <= data16p_1(7 downto 0);
c1_p13 <= data16p_1(15 downto 8);
c1_p11 <= c1_p13;
------------------------------------------
elsif pix_flag = "0001" then
------------COLOUR MATCHING ------------------------
if pix_addr(10) = '0' then


-- G R G
-- B G B
---+-----+-----+-----+
-- | p11 | p12 | p13 |
-- +-----+-----+-----+
-- | p21 | p22 | p23 |
-- +-----+-----+-----+
------for VGA#1--------------------------LEFT
c1_red22 <= c1_p12;
c1_green22 <= c1_p22;
c1_blue22 <= c1_p21;

c1_red23 <= c1_p12;
c1_green23 <= c1_p22;
c1_blue23 <= c1_p23;

-----------------------------------------
else

-- B G B
-- G R G
---+-----+-----+-----+
-- | p11 | p12 | p13 |
-- +-----+-----+-----+
-- | p21 | p22 | p23 |
-- +-----+-----+-----+
------for VGA#1--------------------------LEFT
c1_red22 <= c1_p22;
c1_green22 <= c1_p21;
c1_blue22 <= c1_p11;

c1_red23 <= c1_p22;
c1_green23 <= c1_p23;
c1_blue23 <= c1_p13;

-----------------------------------------
end if;
end if;
end if;
end process;

process(CLOCK)
begin
if (CLOCK'event and CLOCK = '0') then
we3 <= "0";
we4 <= "0";
if(pix_flag = "0011") then
if(pix_addr(10) = '0') then
din3 <= c1_green23 & c1_green22; -- write to bank3
c1_green22_pre <= dout4(7 downto 0); -- read from bank4
c1_green23_pre <= dout4(15 downto 8);
else
din4 <= c1_green23 & c1_green22; -- write to bank4
c1_green22_pre <= dout3(7 downto 0); -- read from bank3
c1_green23_pre <= dout3(15 downto 8);
end if;
```

```
elsif(pix_flag2 = '1') then
if(pix_addr(10) = '0') then
we3 <= "1"; -- write to bank3
else
we4 <= "1"; -- write to bank4
end if;
end if;
end if;
end process;



process(clock)
variable a_c1, b_c1 : std_logic_vector(7 downto 0);
begin
if(clock'event and clock = '0') then
if(pix_flag = "0001") then
if(c1_green22_pre > c1_green23) then
if(c1_green23_pre > c1_green22) then
a_c1 := c1_green22_pre - c1_green23;
b_c1 := c1_green23_pre - c1_green22;
else
a_c1 := c1_green22_pre - c1_green22;
b_c1 := c1_green22 - c1_green23_pre;
end if;
else
if(c1_green23_pre > c1_green22) then
a_c1 := c1_green23 - c1_green22_pre;
b_c1 := c1_green23_pre - c1_green22;
else
a_c1 := c1_green23 - c1_green22_pre;
b_c1 := c1_green22 - c1_green23_pre;
end if;
end if;
edge <= a_c1 + b_c1;
elsif(pix_flag = "0000") then
if( threshold < edge ) then
DATA_VGA1_G <= (others => '1');
DATA_VGA1_B <= (others => '1');
DATA_VGA1_R <= (others => '1');
else
DATA_VGA1_G <= (others => '0');
DATA_VGA1_B <= (others => '0');
DATA_VGA1_R <= (others => '0');
end if;
elsif pix_flag2 = '1' then
if( threshold < edge ) then
DATA_VGA1_G <= (others => '1');
DATA_VGA1_B <= (others => '1');
DATA_VGA1_R <= (others => '1');
else
DATA_VGA1_G <= (others => '0');
DATA_VGA1_B <= (others => '0');
DATA_VGA1_R <= (others => '0');
end if;
end if;
end if;
end process;
end Behavioural;
```

VITA

Pil Woo (Peter) Chun

Objective

Dedicated to be an integral part of a progressive group that enables me to demonstrate and exercise my expertises

Selected Qualification

- Familiarity with peripheral devices: memory – flash, sram and sdram, image sensor – high-speed (200fps) and high-resolution (3M), transceiver – LVDS (200MHz), CAMERA Link (2Gbit), USB and RS232, video standard – VGA and XVGA
- Excellence in dealing with engineering related software: Microchip, Atmel and Motorola ASM, MATLAB, SIMULINK, LABVIEW, PTDS (Photonic Transmission Design Suite), PSPICE, Xilinx ISE, Cadence ORCAD, Allegro (capture and layout) and ModelSim
- Fluent usage of computer and HDL languages: C, Java, C# and VHDL
- Professional Engineer of Ontario

Projects

2006-2008 FAST TRACK: High Frame Rate Stereovision Tracking System

Funded by: MDA space missions, Ontario Centers of Excellence (OCE) and CITO

- Investigated space-borne computer stereovision system for automated satellite grasping and automated satellite docking for recent space robotic systems developed at MDA Space Missions
- developed application specific (200 frames per second) stereo video sensors with on-board preprocessing FPGA-based reconfigurable multi-stream platform
- Implemented stereovision algorithms on the FPGA based computing platform
- Embedded CAMERA setting control on microchip microcontroller via USB communication — i.e., FTDI

2005-2006 FPGA based Stream Processing Platform with Partial reconfiguration Mechanism

Funded by: Unique Broadband Systems (UBS Limited), Ontario Centers of Excellence (OCE) and CITO

- Developed the prototype of the platform for the new generation of Multimedia, DVB (Digital Video Broadcasting) and DAB (Digital Audio Broadcasting) systems
- Designed to explore the concept of a run-time reconfigurable FPGA by utilizing on-chip resources and by allocating stream processing tasks operators efficiently in the FPGA based multi-mode and multi-task applications.
- Implemented image sensor, memory and display controller on Xilinx Virtex-4 FPGA
- Embedded Flash controller on CPLD and USB communication via microchip microcontroller

2002-2004 Adaptive Group Organized Reconfigurable Architecture (AGORA) parallel Platform

Funded by: National Science Engineering Research Counsel (NSERC)

- Developed the partial reconfigurable mechanism for Xilinx Virtex-E FPGA device
- Implemented SRAM based configuration stream loader to enable partial reconfiguration via RS232 serial com. and Java programming
- Designed Hardware oriented operating system deployed in Xilinx Virtex device
- Demonstrated the feasibility of partial reconfiguration through Live visual acquisition and real-time display of the conveyed information

Employment

2004-present Ryerson University Toronto, Canada

Part-time Faculty member for ELE804 (Winter07) – course coordinator

- Enrollment 26 students
- Topics: monolithic integrated circuit analysis and designs including analog multipliers, low noise op-amps, log amplifiers, oscillators, voltage regulators, timer and waveform generators
- Taught 4th year electrical engineering students with emphasis on monolithic integrated circuit analysis and design of practical Phase-Lock-Loops and designed the project involving hardware construction and emulation of high-speed communication channels using PLL components

Part-time Faculty member for EES512 (Spring/Summer06, Fall07, Spring/Summer07 and Spring/Summer08)

- Enrollment: 70 to 90 students
- Topics: covers the basic concepts of charge, current, voltage and power and static and transient analysis of R-L-C components
- Taught 3rd and 4th year engineering students with the topic that enables them to be familiar with general electric R, L and C components and their applications and aided students to encourage their participation by running virtual classrooms and online tutorials

Part-time Faculty member for EES612 (Winter06)

- Enrollment: 138 students
- Topics: covers the various electronics devices and their applications such as op-amps, transistor, diode and principle operations of electric machines such as generators and motors
- Taught 3rd and 4th year engineering students with the goal of leading them into design and operation principles of electronic devices and electric machines and prepared the DUMMY user's guides for the laboratory experiments

Part-time Faculty member for EES/COE538 (Fall04 and Fall05)

- Enrollment: 70 to 85 students
- Topics: Microprocessor architecture and structure, I/O serial/parallel communication with/without handshaking, Timing, Interrupts and Exceptions, Internal structure and design of peripheral devices, Memory system design analysis
- Instructed 3rd year students with emphasis on software and hardware interfacing, provided the detail implementation examples of M68HC11 architecture and structure and assisted students with the usage of lab firmware development tools and guided them to understand hardware and software aspects of the embedded system

May 2006 MDA Space mission CorporationBrampton, Canada

Hardware Engineer – contractor

- Reported to Piotr Jasiobedzki

- Evaluated the Alpha-Data card to restore its previous operational status and developed a test suite for its associated memory modules with Xilinx-II device

1999-2001 Nortel (Advance Design Tech.) Kanata, Canada

Researcher

- Investigated on new advance technology such as SONET FEC using RS en/decoder, Electro-absorption (EA) modular, , EDFA amplifier, optical MEM switches and Thermo Electric Cooler (TEC) laser
- Involved in link budget engineering process that requires communication with vendors and testing various electrical/optical components
- Characterized optical tests for 10G Ethernet model involving Optera Metro 8600, HP BERT system and Oki TPG generator with GEthernet framing capability

Education

2004-2009 Ryerson University Toronto, Canada

- Doctor of Philosophy candidate in Electrical and Computer Eng.

2002-2004 Ryerson University Toronto, Canada

- Master of Science in Electrical and Computer Engineering

1997-2002 Ryerson University Toronto, Canada

- Bachelor of Electrical and Computer Engineering with honor

Scholarships and Awards

Ryerson Entrance (1997), Jack Roy Longstaffe Memorial (1998), NSERCUSRA (2002), Graduate school scholarship (2003-2007), OGSA (2003), NSERC (2005-2008), OGS (2005), Winners of OCE professional Outreach Award (08), three times 1st prize winner on SVAR 05, 07 and 08 conferences

Published papers

Journal

- Macro-Programmable Reconfigurable Stream Processor for Collaborative Manufacturing Systems”, - *accepted for publication in the Journal of Intelligent Manufacturing (JIM),* accepted in August 30, 2007
- Laser thermal therapy: utility of interstitial fluence monitoring for locating optical sensors, Phys. Med. Biol. 46 (2001) pp.91-96

Conferences

- Lowering Power Consumption using Run-Time Reconfiguration for Stereo Rectification, *in Proc. of Canadian Conference on Electrical and Computer Engineering* – CCECE 2008, Niagara Falls, Canada, May 4-7, 2008
- Implementing a Cost-effective Run-time Reconfigurable System for Stream Applications, - *in Proc. of 2-nd International Conference on Electrical Engineering* – ICEE2008, Lahore, Pakistan, March 25-26, 2008

- Improving Cost-Effectiveness using a Micro-Level Static Architecture on Stream Processors, - *in Proc. of 4-th International Symposium on Electronic Design, Test & Application* – DELTA2008, Hong-Kong, Jan. 23-25, 2008
- Reconfigurable Macro-Processor – Cost-Efficient Platform for Rapid Prototyping, *FAIM2007 – Proc. of the 17-th International Conference on Flexible Automation and Intelligent Manufacturing*, Philadelphia, USA, June 2007,Vol. 2, pp. 781-788
- A Cost-Efficient Reconfigurable Video-Processing Platform for Machine Vision, *AMT2007- The 7-th International Workshop on Advanced Manufacturing Technologies,* London, Canada, June 2007, p.44
- A Framework for a Partially Reconfigurable System in a Parallel Multi-tasking Environment, *FPL 2006 - 16th International Conference on Field Programmable Logic and Applications,* Madrid, Spain, August 28-30, 2006
- Uniform Reconfigurable Processing Module for Design and Manufacturing Integration, - *in Proceedings of the Fifth International Workshop on Advanced Manufacturing Technologies – AMT2005*, London, Canada, May 2005, pp. 77-82
- Reconfigurable Multiprocessor with Self-optimizing Self-assembling and Self-restoring Micro-architecture, WARFP05 – *Workshop on Architecture Research using FPGA Platforms –in conjunction with* HPCA-11, San Francisco, Feb. 13, 2005
- Laser thermal therapy: Utility of interstitial fluence monitoring for locating optical sensors, *CDROM Proc. World Congress on Medical Physics and Biomedical Engineering* (also 22nd Annual International Conference of the IEEE Eng. Med. Biol. Soc.)

Workshops

- Closed loop high speed design seminar, mentor graphics, Mississauga, Canada, November, 2007
- High-speed interconnect design with Xilinx Virtex-5: XpressTrack series, December, 2007, nu horizons electronics, Brampton, Canada
- Business Development and Entrepreneurship Core Course, OCE Value Added Personnel (VAP) initiative, Mchill Business School, August, 2007
- Multi-stream Adaptable Reconfigurable System, *SVAR 2007 (Space Vision and Advanced Robotics),* MDA corporation
- Project Management Core Course, OCE Value Added Personnel (VAP) initiative, Queen Business School, March, 2007
- Dynamically Reconfigurable Network-On-Chip, Ryerson ELEC colloquium, March, 2007
- Real-time Multi-video Stream Processing System with Dynamic Partial Reconfiguration, *TEXPO2006*
- Fast Track: A High Frame Rate Stereovision Tracking System, *SVAR 2006 (Space Vision and Advanced Robotics),* MDA

- Multi-task Parallel Video-stream Processor with Self-Assembling Micro-architecture, *SVAR 2004 (Space Vision and Advanced Robotics),* MD Robotics
- Implementation of Partially Reconfigurable Computing Platform for Multi-task Video-Processing Applications, *SVAR 2003 (Space Vision and Advanced Robotics)*, MD Robotics