Exzellenzcluster
Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

Dynamically Reconfigurable Hardware for Embedded Control Systems

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

M.Sc. Carlos Vladimir Paiz Gatica

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferent:	Prof. Dr.-Ing. Joachim Böcker

Tag der mündlichen Prüfung: 21. Dezember 2011

Paderborn / Januar 2012
DISS KS / 02

Para Araceli, mi compañera de vida, y Ernesto, nuestro fruto y esperanza.

Abstract

This thesis explores the use of dynamically reconfigurable hardware for the realisation of embedded control systems, using the most well-known example of this kind of technology: Field Programmable Gate Array (FPGA). The focus of the first part of the thesis is on assessing the resource utilisation of FPGA- and CPU-based realisations, relating the results to the algorithmic characteristics of the implemented controller, and the properties of both hardware and software architectures. Using a selected set of benchmarks, it is shown that an FPGA-based design achieves a higher computational density ($C_{density}$=throughput/area) and a higher energy efficiency ($E_{efficiency}$=throughput/power) than a CPU-based implementation. Furthermore, it is shown that when the average parallelism of the algorithm to be implemented increases when increasing the problem size (i.e., the amount of computations required for that algorithm), the gap between FPGA- and CPU-based realisations in terms of computational density increases, too.

The use of run-time hardware reconfiguration to achieve a more efficient resource utilisation than a static approach is investigated in the second part of this work. It is shown that control systems requiring structural and parametric adjustments during execution can benefit from run-time hardware reconfiguration. Application examples are presented showing that the proposed concepts are successfully realisable using current technologies, also for control applications having demanding time-constraints.

New design methodologies are required for embedded control systems using dynamically reconfigurable hardware, specially for those targeting run-time hardware reconfiguration. A Hardware-in-the-Loop design framework is presented in the third part of this work, which allows an early cycle-accurate verification of a design under test (DUT), using a simulated environment. In a second stage, the DUT can be monitored in real-time, and design parameters can be adjusted during operation, while using the target environment of the DUT. Several realisation examples show the efficacy of the proposed framework.

This thesis shows that dynamically reconfigurable hardware, particularly FPGA technology, is a suitable platform for demanding embedded control applications. Methods and tools presented in this thesis disclose the advantages of dynamically reconfigurable hardware, and represent a step towards taking full advantage of the possibilities offered by this technology, in the context of embedded control systems.

Contents

Abstract iii

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Thesis Outline . . . 4

2 Realisation of Digital Control 7
- 2.1 Digital Control . . . 7
  - 2.1.1 Software-Based Design . . . 9
  - 2.1.2 ASIC-Based Design . . . 11
- 2.2 Reconfigurable Hardware . . . 13
  - 2.2.1 Field Programmable Gate Array . . . 14
  - 2.2.2 General FPGA-Based Design Flow . . . 18
- 2.3 Utilisation of Reconfigurable Hardware for Digital Control . . . 19
  - 2.3.1 Application Spectrum . . . 20
  - 2.3.2 Factors of the Technology Migration . . . 22
  - 2.3.3 Coupling of Reconfigurable Hardware and Software Architectures . . . 30
  - 2.3.4 Run-Time Hardware Reconfiguration . . . 34
- 2.4 Summary . . . 35

3 Technology Comparison of Reconfigurable Hardware and Software Architectures 37
- 3.1 Algorithmic Characterisation . . . 38
  - 3.1.1 Controller Representation: Cyclic Data Flow Graph . . . 38
  - 3.1.2 Scheduling of a CDFG . . . 39
  - 3.1.3 Basic Operations Set: Selection and Weighting . . . 41
  - 3.1.4 Normalised Operations and Steps . . . 42
  - 3.1.5 Average Parallelism . . . 48
- 3.2 Resource Utilisation Assessment . . . 50
  - 3.2.1 Computational Density . . . 52
  - 3.2.2 Energy Efficiency . . . 55
- 3.3 Computing Architectures . . . 56
  - 3.3.1 PowerPC 750-G Processor . . . 56

3.3.2 FPGA Device . . . 58
3.4 Realisation Flow . . . 60
3.4.1 Hardware Implementation-Flow . . . 60
3.4.2 Software Implementation-Flow . . . 62
3.5 Benchmarks . . . 63
3.5.1 PID Controller . . . 64
3.5.2 State-Feedback Controller . . . 75
3.5.3 State Observer . . . 84
3.6 Summary . . . 93

4 Run-Time Hardware Reconfiguration 97
4.1 Controller Adjustment . . . 98
4.2 Run-Time Hardware Reconfiguration . . . 100
4.2.1 Configuration Granularity . . . 102
4.2.2 Configuration Interface . . . 104
4.2.3 Partial Reconfiguration Process . . . 105
4.2.4 Partition and Placement Approaches . . . 108
4.2.5 Communication Infrastructure . . . 110
4.3 Control Adjustment Through Run-Time Reconfiguration . . . 111
4.3.1 Structure Adaptation . . . 111
4.3.2 Parameter Adaptation . . . 114
4.4 Implementation Examples . . . 116
4.4.1 The RAPTOR System . . . 117
4.4.2 System Architecture . . . 118
4.4.3 Inverted Pendulum System . . . 121
4.4.4 Self-Optimising Motion Controller . . . 128
4.5 Summary . . . 138

5 Design Verification through Hardware-in-the-Loop Simulations 141
5.1 Classification of Test-Systems . . . 142
5.1.1 Model- and Software-in-the-Loop . . . 142
5.1.2 Rapid Prototyping . . . 143
5.1.3 Hardware-in-the-Loop Simulation . . . 143
5.1.4 On-Line Test . . . 143
5.1.5 FPGA-in-the-Loop . . . 144
5.2 HiLDE: HiL Design Environment . . . 146
5.2.1 Hardware Components . . . 147
5.2.2 Software Components . . . 149
5.2.3 Communication and Performance . . . 151
5.2.4 HilDE Tool Flow . . . 155
5.2.5 Implementation Examples . . . 159

5.3 HiLDEGART: HiL Design Environment for Guided Active Real-Time Test . . . 166
5.3.1 Hardware Components . . . 167
5.3.2 Software Components . . . 168
5.3.3 HiLDEGART Tool Flow . . . 170
5.3.4 HiLDEGART Implementation Examples . . . 171
5.4 Summary . . . 176

6 Summary and Outlook 177
6.1 Summary . . . 177
6.2 Outlook . . . 180

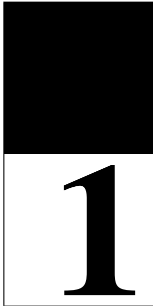
Author's Publications 183

Bibliography 187

List of Figures 205

List of Tables 211

Glossary 213



1 Introduction

Digital technology is constantly evolving in response to the ever-increasing computational requirements of modern society. A good example of this evolution is the so-called Moore's law, which says that the amount of transistors contained in a processor doubles every two years [Moo98]. This trend has been observed since the publication of the original paper in 1965, and will provably continue for some more decades. However, there are physical limitations (e.g., thermal noise [Kis02], transistor scaling limitations [Tho06], or power dissipation issues [Kis04]), which force us to look for other possibilities to increase performance. Furthermore, this exponential growth of the number of transistors in a processor does not necessarily results in exponential growth of CPU performance. One way to increase performance while keeping resource efficiency is to adapt the computer architecture to the application.

Digital Signal Processors (DSP) are a good example: the most common operation of applications in the domain of filtering or video processing is multiplication. Therefore, most of the current DSP architectures have at least one specialised multiply and accumulate (MAC) unit. Architectures for digital control have evolved similarly. Let us consider microcontrollers as an example. They have been used traditionally for motion control applications in both low-performance AC inverter drives and high-performance servo drives. This platform has evolved from very simple architectures to one-chip solutions, incorporating specialised DSP functions (e.g., MACs), Digital to Analog Converters, Analog to Digital Converters, Pulse Width Modulators, along with dedicated hardware for networked communication. This means that not only the level of specialisation, but also the level of architectural parallelism has been increased, by adding specialised processing units in response to application requirements.

In the last decade, the rise of more complex and more computation-intensive control schemes has motivated engineers and researchers to explore new architectures, or even new computing paradigms to reach the required performance. Reconfigurable Hardware (RH), specifically the most well-known architecture of this kind of technology, Field Programmable Gate Array (FPGA), has emerged as an alternative for demanding

applications, because of its high architectural parallelism, and the possibility to change the configuration of the device according to the application.

FPGAs are heterogeneous devices, constituted by programmable functional blocks and embedded application-specific hardware, such as embedded processors, memory, or multipliers, interconnected by a reconfigurable network. This allows for new ways to implement digital controllers, leaving the traditional CPU-based approach for a highly parallel realisation. However, can all kinds of control algorithms benefit from this technology for their realisation? Is an FPGA-based realisation faster, or less energy consuming than a CPU-based realisation? These questions motivate the first part of the present work.

It is well-known, that the feature of reconfigurability of SRAM-based FPGAs comes at the prices of a great amount of silicon resource dedicated to enable reconfiguration of logic elements (more than 80% of silicon resources [Fen06]). Are there ways to take advantage of resources dedicated for configuration when the device is in operation? For which kinds of control applications would such an approach be beneficial? The second part of the present work is inspired by these questions.

Furthermore, aspects of the design flow of digital controllers such as design verification and real-time monitoring, which are already standard for CPU-based design flows, are addressed in the last part of the present work. The main contributions of this thesis are summarised in the following section.

1.1 Contributions

FPGAs offer a different approach to realise computations when compared to centralised architectures, such as general purpose processors. The architectural parallelism offered by FPGAs allows for spatial computation in contrast to the classical approach, in which a problem is first decomposed in single steps that are executed sequentially. The first contribution of this thesis is a quantitative comparison of software- and FPGA-based realisations, from a technological point of view. The main contributions are:

- A set of metrics is proposed to evaluate algorithmic characteristics of controllers. Particularly, the number of average operations per execution step (AOS) is used to measure average parallelism, which together with the size of the algorithm ($SizeAlg$) are used to characterise selected benchmarks. Furthermore, computational density ($C_{density} = throughput/area$), and energy efficiency ($E_{efficiency} = throughput/power$) are used to assess the resource utilisation of a hardware- and software-base realisation. Computational density and energy efficiency are metrics taken from literature and are adapted to the application field of control systems.

- Three representative control algorithm are chosen as benchmarks for this comparison: a PID controller, a state-feedback controller, and a full state observer. Based on the implementation results of these benchmarks, and on their algorithm characteristics, the advantage of realising controllers using FPGA technology are quantitatively demonstrated. It is shown that an FPGA implementation leads to a higher $C_{density}$ and $E_{efficiency}$, which implies a more efficient use of resources. Presented implementation results show that an algorithm having an increasing problem size (i.e., $SizeAlg$) with constant average parallelism (i.e., *AOS*) derives in a reduction of the gap between FPGA- and CPU-based realisations, regarding achievable values of $C_{density}$ and $E_{efficiency}$. On the contrary, when an algorithm has an increasing *AOS* when increasing $SizeAlg$ the gap between FPGA- and CPU-based realisations increases, too.

Achievable performance decreases for controllers requiring more resources than those available in an FPGA device, because configurable logic has to be time-shared. This situation can be caused by having more resources occupied on an FPGA than those required at a given time. This is the case of control approaches requiring some kind of adjustment at run-time, because all possible configurations have to be loaded into the FPGA when the configuration of the device stays constant through the whole operation cycle (i.e., a static approach). To tackle this problem, the utilisation of run-time reconfiguration of FPGAs for control application is presented. The main contributions are summarised in the following paragraphs:

- It is shown that FPGA-based control systems requiring adjustments during operation can benefit from RTR. Two cases of adjustments are distinguished: structural and parametric changes.
- It is shown that the resource utilisation of a dynamic approach depends on the worst-case configuration of the system, whereas for a static implementation the resource utilisation depends on all required configurations. This can lead to a better resource utilisation for systems using run-time-reconfiguration, in contrast to a static approach (i.e., the configuration of the device does not change during operation).
- The problem of having a reaction time (i.e., reconfiguration times plus initialisation time) longer than the required control cycle is analysed and solutions are proposed.
- Implementation examples show that the proposed approach can be realised with current technology, also for demanding control applications.

An important part of the design of digital systems is verification. This thesis contributes in this field, by proposing a Hardware-in-the-Loop framework for FPGA-based designs. The main contributions are:

- A cycle-accurate FPGA-in-the-Loop simulation framework using well-known simulation tools such as Matlab/Simulink or CaMEL-View is presented. Hardware-

in-the-Loop Design Environment (HiLDE) allows the early verification of the design under test (DUT) using a simulated environment. Furthermore, it is shown that simulations are accelerated, shortening required design times. Several examples show the efficacy of HiLDE.

- A real-time test framework using the target environment of the DUT is introduced. The focus of this framework called HiLDEGART (HiLDE for Guided Active Real-Time Test) is on monitoring internal states and I/Os of a DUT while it is in operation, on adjusting design parameters, as well as on verifying timing issues. The advantages of using HiLDEGART are disclosed, using several prototypical implementations.
- A tool-flow is presented, which enables the automatic integration of a DUT to a HiL simulation (HiLDE or HiLDEGART), thus making the verification process easier and less error prone.
- Several realisation examples prove the efficacy of the proposed frameworks, also for applications requiring run-time reconfiguration.

These topics are addressed in the following chapters, as explained in the next section.

1.2 Thesis Outline

Chapter 2 presents relevant background on digital control, showing the main differences between a software-, an ASIC-, and an FPGA-based design flow. The chapter offers a review of the state-of-the-art regarding the utilisation of reconfigurable hardware for control applications. It is shown that FPGAs are leaving their role as prototyping platforms to become the target architecture for demanding control applications. The reasons are examined and the research problems investigated in this work are motivated.

Chapter 3 explores the benefits and challenges of using FPGA-technology in contrast to a CPU-based realisation for embedded control applications. This chapter begins by introducing used metrics for algorithm characterisation and for resource utilisation assessment. Furthermore, the used computing architectures are presented in detail. The chapter continues by showing the design- and tool-flow used for the implementation of benchmarks. Implementation results and the corresponding analysis and discussion follows, and finally the main contributions are summarised.

The use of run-time reconfiguration (RTR) for control applications is presented in chapter 4. The concept of RTR is explained, analysing different aspects of the design of systems using RTR. It is shown that FPGA-based control systems requiring some kind of adjustments during operation can benefit from RTR. Two cases of adjustments are distinguished: structural and parametric changes. For both cases

RTR can be used to achieve a better resource utilisation, depending on the amount of structural variations, or the size of the algorithm requiring parametric changes. Two implementation examples are examined: a two-controller system for an inverted pendulum, and a self-optimizing motion controller. Prototypical realisations of the presented concepts show the advantage of this approach.

Chapter 5 introduces a Hardware-in-the-Loop design environment for FPGA-based controllers, which includes an off-line simulation framework (HiLDE) and an online monitoring tool (HiLDEGART). These frameworks support the design flow of FPGA-based controllers targeting run-time reconfiguration. Hardware and software components of both frameworks are presented, as well as a tool-flow which allows the automatic integration of the design under test. Several application examples are presented, showing the benefit of using HiLDE and HiLDEGART.

Finally, chapter 6 summarises the main results presented in previous chapters and offers conclusions, giving also an outlook based on analysis of collected experiences during this thesis work.

2 Realisation of Digital Control

This chapter introduces background concepts on digital control and reconfigurable hardware. The main contribution of the chapter is a literature review of the use of Field Programmable Gate Array (FPGA) technology for control applications. Based on this review, relevant research problems undertaken in the thesis are motivated. The chapter ends with a brief summary.

2.1 Digital Control

The aim of a controller is to influence the behaviour of a system, usually referred to as plant, by applying control signals in order to achieve a particular control objective. Control signals can be computed with or without direct information of the plant, which is known as open- and close-loop control, respectively. If direct information is gathered from the plant, control signals are usually a function of an error signal ($e(k)$); the difference between a desired value or set point, ($s(k)$) and a measured value ($y(k)$), cf. equation 2.1.

$$e(k) = s(k) - y(k) \tag{2.1}$$

Digital control refers to the utilisation of signals discrete in value and time to control a plant. This process involves the utilisation of digital hardware (e.g., a processor) to compute the control effort. Figure 2.1 depicts a typical digital close-loop controller.

Feedback signals ($\tilde{y}(t)$, cf. figure 2.1) are typically measured by sensors, which convert a given form of energy, such as mechanical movement, heat, or light, into electrical signals. Sensor signals are ideally directly proportional to the measured property. These signals are usually conditioned ($y'(t)$), e.g., filtered and amplified, before they are digitalised ($y(k)$). The digitalisation implies three steps:

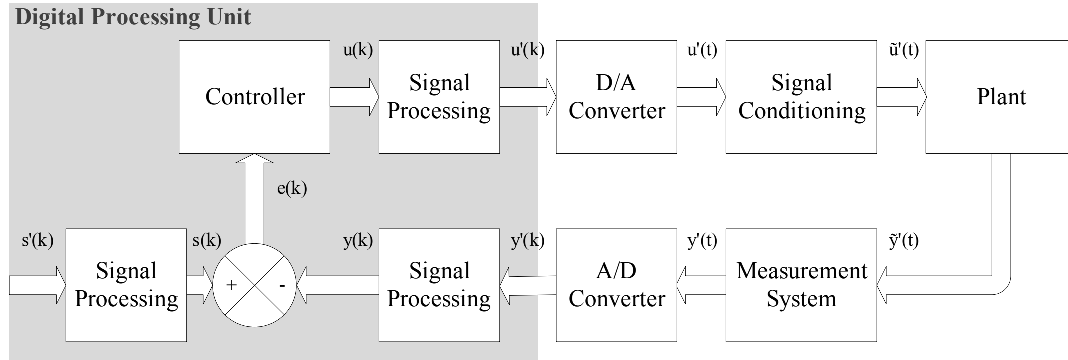


Figure 2.1: Block diagram of a typical digital control system

1. Sampling: involves the conversion of a signal continuous in time and amplitude into a signal discrete in time and continuous in amplitude. This process is done at regular intervals of time known as *sampling period*, which are considered to be constant.
2. Quantisation: is the conversion of the sampled signal into a discrete-time discrete-valued signal. A given signal sample is then represented by a finite binary string, whose length determines the quantization level, given by equation 2.2

$$QL = \frac{FSR}{2^n} \tag{2.2}$$

   where FSR is the Full Signal Range (e.g., the maximum voltage level supported by the ADC), and n is the number of bits used for the quantization.
3. Coding: is the process, in which quantized samples are represented by a specific numbering format (e.g., fixed point, floating point).

Although these steps are presented as separated processes, they are generally done in a single chip, known as analog to digital converter (ADC). After data has been digitalised, it typically needs some kind of adjustment ($y(k)$), e.g., rescaling and filtering, which is a process done by the digital processing unit. Control signals ($u(k)$) might require some processing ($y'(k)$) before they are converted to analog signals ($y(t)$). Correspondingly, analog outputs generally need some kind of conditioning, such as filtering or amplification, ($\tilde{u}(t)$) before they can be applied to the plant.

Embedded digital controllers have quantifiable requirements, such as energy consumption, performance (hard real-time computation), and implementation costs. Therefore, the implementation of a controller based on an embedded device differs from a controller based on a general purpose computing platform, where those factors are not implicit.

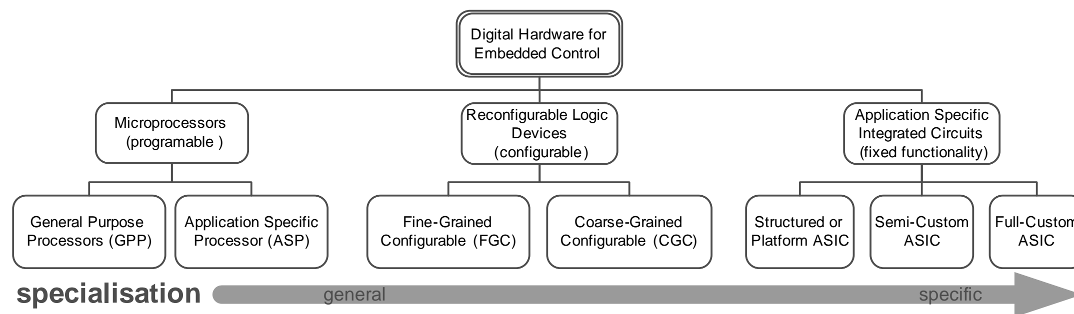


Figure 2.2: Digital hardware platforms to implement digital controllers classified according to the level of specialisation [Des06]

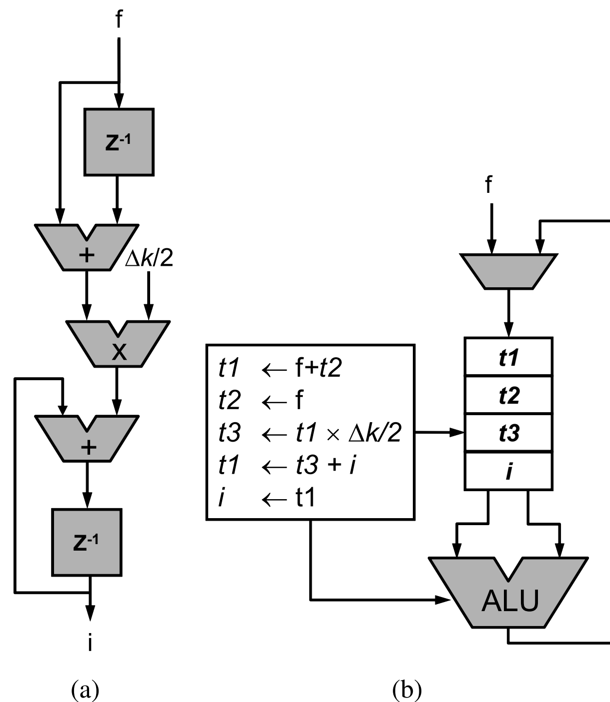
Traditionally, the target technologies for embedded digital controllers have been, on the one hand, software architectures varying in the level of specialisation, and on the other hand, application specific integrated circuit (ASIC). Recently, reconfigurable logic devices, particularly FPGA technology, have gained popularity as target platform. All these architectures differ in the level of specialisation to the application, as depicted in figure 2.2.

Microprocessors are based on a computation scheme, where a central processing element, e.g., an arithmetic logic unit (ALU), is used to sequentially process a set of instructions, which represent a temporally sequenced algorithm. These devices can compute any computable function, by changing their functionality every clock-cycle. On the contrary, when implementing a design in reconfigurable hardware, several processing elements can be used concurrently to compose the desired function, cf. figure 2.3. Reconfigurable hardware devices can compute any computation that fits the available resources (e.g., configurable elements), and are typically configured every operational epoch.

If the controller is implemented as an ASIC, several processing units can also be used concurrently (spatial implementation), but the device does not change its functionality after fabrication. The design flow of an embedded controller differs significantly depending on the target technology. These differences are pointed out in the following sections, where only the implementation using the target technology is considered.

2.1.1 Software-Based Design

Software architectures have a common characteristic: little architectural parallelism (e.g., typically a single Arithmetic Logic Unit - ALU). Therefore, an algorithm to be computed has to be described as a list of instructions, which are executed sequentially. There are several approaches to implement controllers using embedded software



(a) (b)

Figure 2.3: Spatial and temporal implementation of a proportional-integral controller (PI), using trapezoidal integration (sketch based on [DeH00])

architectures. However, typically the design starts by specifying the requirements (e.g., required sampling rate). In this stage a mathematical model of the plant is derived, together with a mathematical description of the controller. The use of high-abstraction level tools, such as Matlab/Simulink, or CaMEL-View is very common at this stage of the design flow. The controller is then simulated using either continuous-time floating- or fixed-point (or a mixture of them) based models. In a second stage, high-level code is derived, either automatically from a model or manually. A second verification stage can be carried on, by using the generated fixed-point code in combination with discrete models. Afterwards, processor specific code (machine code) is derived through a compilation/linking process. At this stage processor-in-the-loop or hardware-in-the-loop simulations are used for verification. Figure 2.4 presents an overview of the design flow.

Because of the nature of software architectures, the designer does not have a direct influence on the underlying platform, but rather, the architecture and the algorithmic characteristics of the controller define the achievable performance.

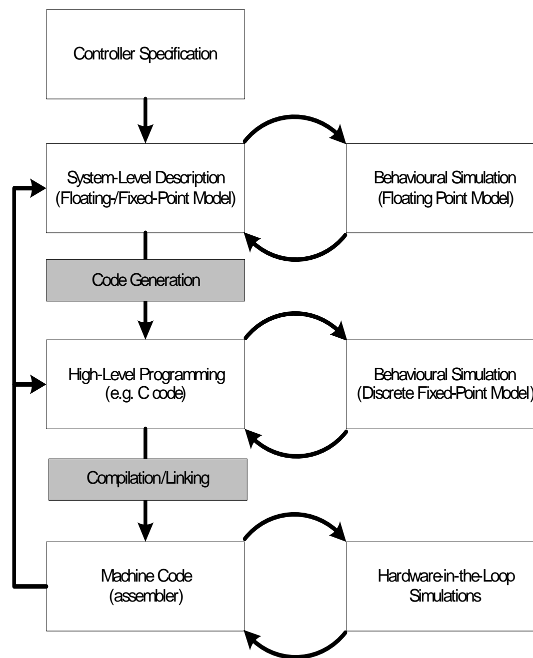


Figure 2.4: Embedded software design flow

2.1.2 ASIC-Based Design

An ASIC is an integrated circuit, which has been designed for a specific application. ASICs can be classified into three main categories: full-custom, semi-custom, and structured, as depicted in figure 2.2.

Full-custom ASICs are entirely fitted to an application, which implies that the designer can freely optimise the device in terms of area, time, and energy, thus reducing recurring component costs. Photo masks for all layers of the device, which are required for the photolithographic process of an ASIC production, have to be specified. This implies a increased production- and design-time. Thus, an increased price per device.

Semi-custom ASICs are designed based on predefined elements. Two kinds can be distinguished: standard-cell design, and gate-array design. Standard-cell ASICs are based on a pre-characterised collection of gates, which are typically provided by the manufacturer. The designer performs a very similar flow to that of full-custom ASICs, defining the placement and interconnection of the design, but instead of defining every gate, a standard library is used. Gate-array ASICs are constituted by pre-placed transistor-arrays. The device is customised by defining the local and global interconnect.

Structured ASICs (also known as Platform ASICs) are built from a sea of tiles, otherwise called modules, and a combination of embedded cores (such as memory

or I/O blocks). Tiles are logical elements, whose granularity varies from transistors to lookup-tables (LUTs). The internal interconnection of tiles is predefined, the designer has to define the interconnection of the tiles and the configuration of the existing embedded block (although much of the local and global interconnect is also predefined). This results in a shorter design- and production-time, in comparison with semi- and full-custom ASICs.

The design flow of ASICs varies depending on the target technology. However, the design can be split up into *behavioural design*, *logic design*, and *physical design*, as depicted in figure 2.5. The flow can be roughly described as follows:

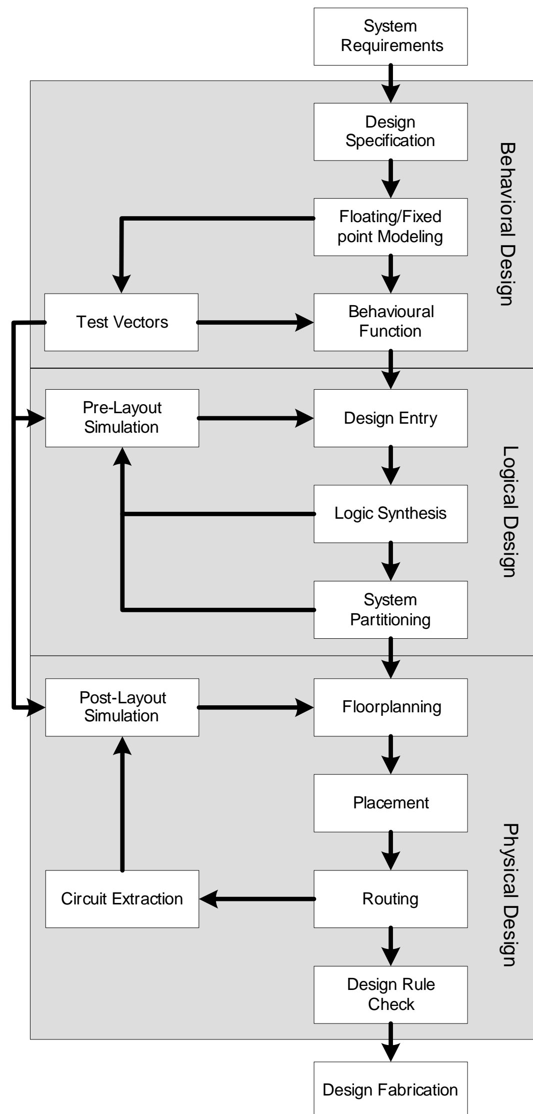


Figure 2.5: Typical ASIC design flow [Nek03]

- System requirements: in this stage, the requirements of the design are sketched.
- Design Specification: formal specifications for the design are derived from the first stage.
- Floating/Fixed Point Modeling: a mathematical model is derived, focusing on the behavioural (mathematical) description of the algorithm, using either floating-point or fixed-point simulations.
- Test Vectors Generation: at this stage test vectors are generated, which are then used in the following verification steps.
- Behavioural Simulation: the mathematical description of the algorithm is verified, using behavioural simulations.
- Design Entry: the design is described by using either hardware description languages (e.g., VHDL, Verilog), or by using an schematic entry.
- Logic Synthesis: form the hardware description, a netlist is extracted, which is a structural model of the design containing references to logic cells.
- System Partitioning: the design, if required, is partitioned into many ASICs.
- Floorplanning: arranges the cells of the circuit and sets space for interconnect.
- Placement: sets the cell location in a block.
- Routing: creates the connections between cells and blocks
- Extraction (Back Annotation): determines the parasitic capacitance and resistance of the interconnect, vias, and contacts.
- Postlayout (Physical) Simulation: verifies the design with the information gained in the previous step.
- Design Rule Check: verifying that the circuit layout complies with the specification of the design rules.

2.2 Reconfigurable Hardware

Reconfigurable hardware, as opposed to CPU-based architectures, is constituted by several processing elements, whose function and interconnection are configurable. These hardware architectures can be categorised by using three main criteria: by the granularity of their building blocks, by the kind of reconfiguration, and by the diversity of their building blocks.

- Granularity: this classification refers to the size and complexity of the most basic computing element of an architecture. Fine-granular architectures allow bit level operations, medium-granular architectures allow operations with different number of bits, and coarse-grade architectures operate at the word level.

- Reconfiguration: two kinds of reconfiguration are distinguished, static reconfiguration, and dynamic reconfiguration, also known as run-time reconfiguration. The later allows a part of the device to be reconfigured while the rest operates, whereas the former requires execution to be stopped. Static configuration can be further classified in SRAM-based configuration, and Flash memory-based configuration. The former losses its configuration after power-off, while the later retains it.
- Block diversity: reconfigurable architectures can be constituted by a variety of computing blocks (e.g., lookup tables, embedded multipliers, etc.), or by a replication of the same basic computing element. The former kind is known as homogenous architecture, while the later is called heterogeneous.

This work focuses on Field Programmable Gate Array (FPGA) technology, which is a fine-granular, dynamically reconfigurable, and heterogeneous architecture. The main components of FPGA technology are introduced in the next section.

2.2.1 Field Programmable Gate Array

Nowadays there are several academic and commercially-available FPGA architectures. Most of them share the same set of basic elements, such as configurable logic blocks, I/O blocks, interconnect memory, embedded cores, clock management blocks, and configuration memory (see figure 2.6). In this section, these basic blocks are briefly introduced.

Configurable Logic Blocks: Configurable logic blocks (CLB) are the basic logic units of FPGAs. Typically, CLBs are composed of lookup tables (LUT), storage elements, multiplexers, and logic gates. The LUTs can be configured to realise any logical function, limited to the number of inputs and outputs of the LUT. Interconnect resources are used if more complex functions are required.

In the Virtex II architecture from Xilinx [Xil07b], a CLB is composed of four slices, which are the basic processing unit of FPGAs from Xilinx, plus interconnect resources. Each slice, shown in figure 2.7 is composed of two four-input one-output LUT, two flip-flops (FF), logic gates, multiplexers, carry chain logic for arithmetic functions, and a horizontal cascaded OR chain for implementing sum of products.

LUTs can be used as functions generators, as shift registers or as a RAM (Random Access Memory). CLBs are attached to interconnect resources, to build more complex functions.

Configuration Memory: FPGA resources are uncommitted, and must be configured to realise a digital design. There are mainly three basic types of configura-

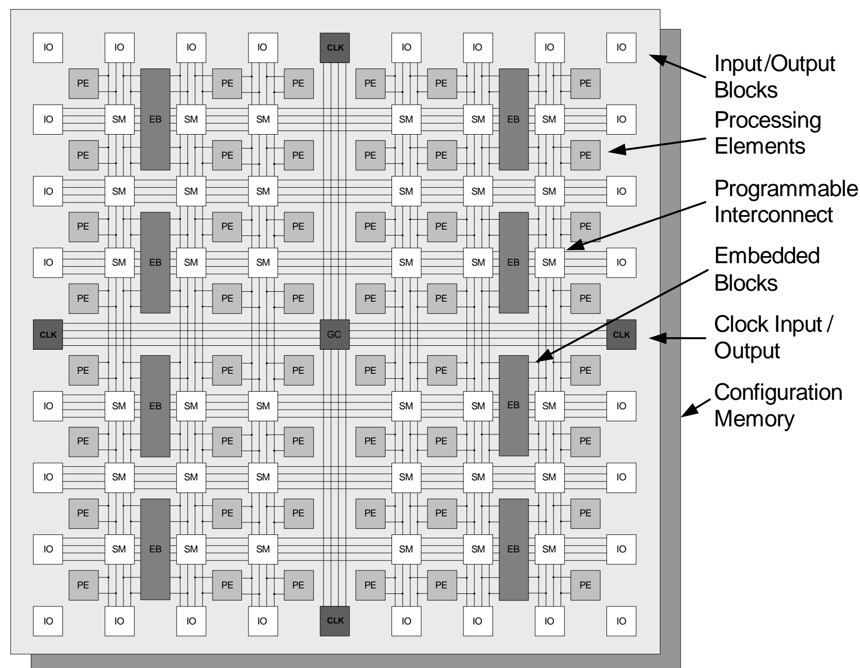


Figure 2.6: Sketch of a typical FPGA architecture

tion approaches: SRAM based configuration, Nonvolatile-based configuration, and antifuse-based configuration.

Antifuse uses one-time programmable connections, whose impedance change on the application of a high voltage signal (the programming voltage). When the device is not configured, connections between blocks have very high impedance values (in order of Giga Ohms), so than the connection is virtually open. When the programming voltage is applied, connections fuse, reducing drastically their impedance to few ohms, thus establishing connections. The functionality of the device can not be changed after configuration.

Non-volatile configuration uses the same principle that EPROM (Erasable Programmable Read-Only Memory), EEPROM (Electrically Erasable Programmable Read Only Memory), or Flash memory use. Floating gate transistors are used to store configuration bits. Thus, configuration remains after power-off. These devices can be reconfigured to support a different functionality.

SRAM configuration uses volatile memory cells to store the configuration of the device. Pass-transistors or lookup tables can be used to configure the device. Since the configuration is volatile, an external memory is required to load the configuration file at power-up.

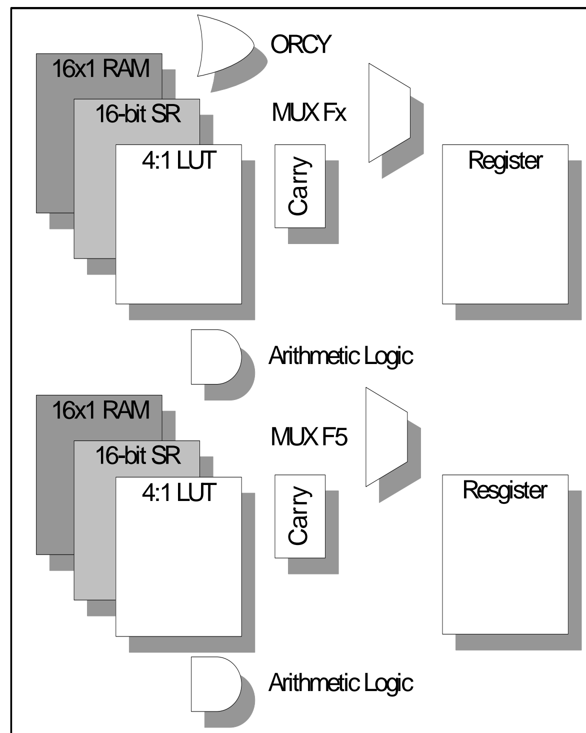


Figure 2.7: Simplified view of a Virtex II slice [Xil07b]

The Virtex FPGA family from Xilinx has an SRAM based configuration, allowing also partial configuration, which is one of the topics being investigated in this thesis. The configuration of Xilinx devices is further examined in chapter 4.

Configurable I/O Blocks: Input/Output blocks provide a bidirectional programmable interface between the FPGA and its peripheral environment. Basically, the I/O blocks can provide three states: input, output, and high-impedance. I/O blocks usually provide registers, in order to reduce the critical path between outside devices and the FPGA.

Programmable Interconnect: Programmable interconnection has a strong influence on the characteristics of the FPGA architecture. Programmable switches are used to realise connection between the different blocks of the FPGA and the routing resources. Typically, FPGAs have their routing resources organised as an island style, where the logic blocks are surrounded by a sea of routing resources, providing a high degree of flexibility (c.f. figure 2.6).

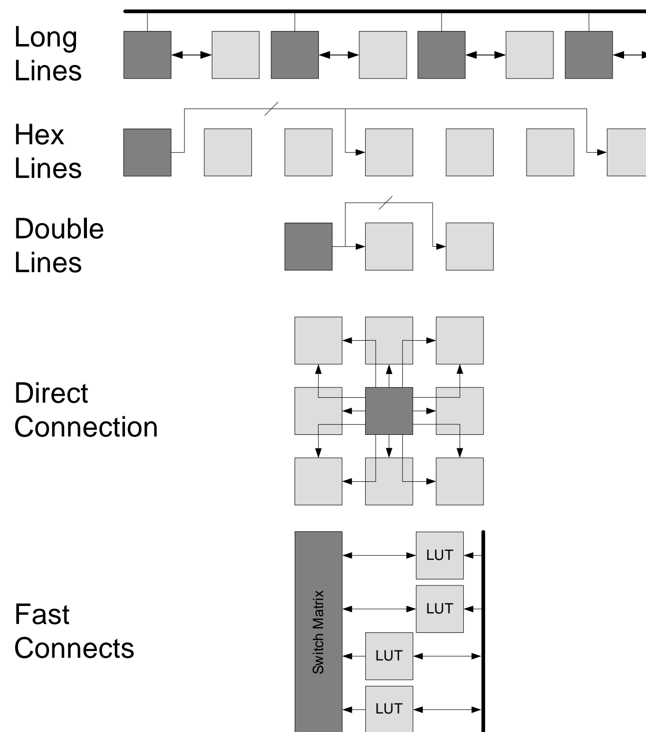


Figure 2.8: Hierarchical routing resources [Xil07b]

Let us consider the Virtex II FPGA as an example, whose hierarchical routing resources are depicted in figure 2.8. It has fast connects for internal CLB I/Os, direct connection to neighbouring blocks, vertical and horizontal double, hex and long lines.

Clock Management: FPGAs have dedicated clock resources, such as clock lines, buffers, multiplexers, and clock managers. Dedicated clock lines provide low-capacitance paths for clock signals. Clock buffers and multiplexers allow to halt or redirect clock signals. Digital Clock Managers (DCMs) provide a flexible control over clock frequency, phase shift and skew. The three most important functions of DCMs are to mitigate clock skew due to different arrival times of the clock signal, to generate a large range of clock frequencies derived from the system clock signal and, to shift the signal of all its output clock signals with respect to the input clock signal.

Embedded ASIC Cores: Modern FPGAs have, besides reconfigurable logic, embedded cores, such as multipliers, memory blocks, and processors. For example, some of the Virtex-II Pro devices have embedded PowerPC 405 RISC processors, high speed transceivers, dual port block-RAMs, Digital Clock Managers, and Multipliers, as depicted in figure 2.9.

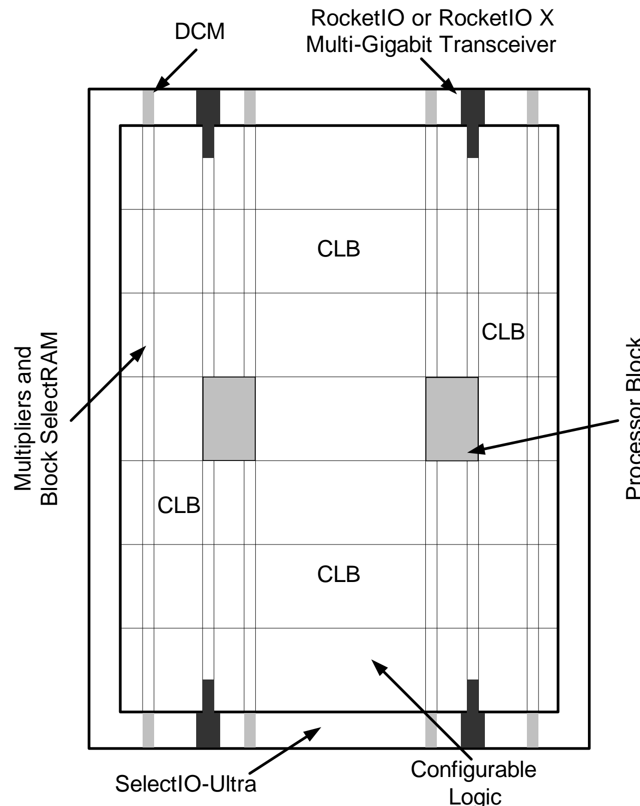


Figure 2.9: Virtex-II Pro Generic Architecture Overview [Xil07c]

2.2.2 General FPGA-Based Design Flow

The design flow begins typically with the specifications of requirements and the definition of the functionality. Typically, a mathematical model of the design and its environment is derived from the previous steps and simulations are done using floating point precision (e.g., using Matlab/Simulink). From this high-level descriptions, hardware description is derived, as shown in figure 2.10. This step can be done using different hardware descriptions, ranging from traditional hardware description languages, such as VHDL or Verilog to C-like languages or schematic entries.

The choice of a hardware description has a great influence on the design development-time and resource-efficiency. A schematic hardware description such as System Generator from Xilinx [Sysb], or Synplify DSP from Synplicity [Synb] can lead to reduce the development time, without compromising resource-efficiency for digital control applications.

At this stage of the design flow, functional simulations are carried out to verify correct logic functionality. The next step is synthesis, where the design entry is translated from a functional description (e.g., VHDL) to a structural description (i.e.,

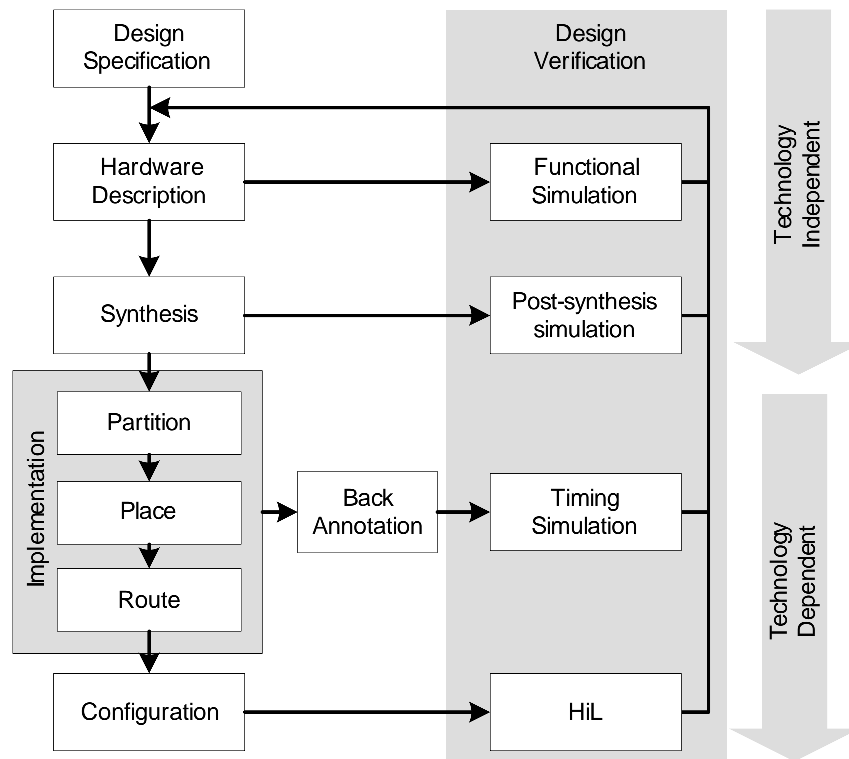


Figure 2.10: Typical FPGA-based design flow [Des06, Sim10]

netlist). Post synthesis simulations are done, to further verify the design. At this stage, technology-dependent steps take place, where the design is further synthesised, partitioned, placed, and routed, taking into account user-defined design constraints. This step is automatically done by vendor-specific synthesis tools. After these steps, post-place-and-route simulations are executed, which include more accurate timing information from the design. Typically, the design is at this stage ready to be loaded on the FPGA. A final verification step includes Hardware-in-the-Loop simulations, analysed in chapter 5.

2.3 Utilisation of Reconfigurable Hardware for Digital Control

The use of reconfigurable hardware for digital control applications, not only as prototyping platform but as final target architecture, has been reported in literature since the early 90's. However, it is only until recently that researchers have started to show a greater interest in this technology, because of higher computational demands of digital control systems, and the fast evolution undergone by FPGAs in the last decades.

Figures 2.11 and 2.12 present two aspects of the evolution of FPGAs; the reduction of the minimum feature size, also known as λ (cf. figure 2.11). Figure 2.12 shows a logical consequence of the reduction of λ, namely the increase of equivalent logic cell per device, which enables the realisation of complex control schemes.

There are also other aspects of FPGA technology that have drastically evolved during the last decades, such as the basic logic cells themselves, the integration of embedded ASICs (e.g., processor cores) into the reconfigurable logic, or improvement of software tools to map digital designs onto this technology, as presented in section 2.2. How this evolution has impact the view on this technology, producing a technology migration, which has taken FPGA from rapid prototyping platforms to target devices [5], is analysed in the following sections, pointing out specific examples from literature and classifying the factors that contributed to adopt FPGA technology for control applications.

2.3.1 Application Spectrum

In literature, the reported application spectrum is very wide. However, three main areas can be distinguished: direct motor control, power electronics and motion control, as depicted in figure 2.13. Motor control refers to the direct manipulation of a DC or AC motor inputs (e.g., voltage or current) to obtain a specific torque, position or speed (for a review on this application domain see [Mon02a]). Power electronics deals with the implementation of control strategies for DC-AC, AC-DC, or DC-DC

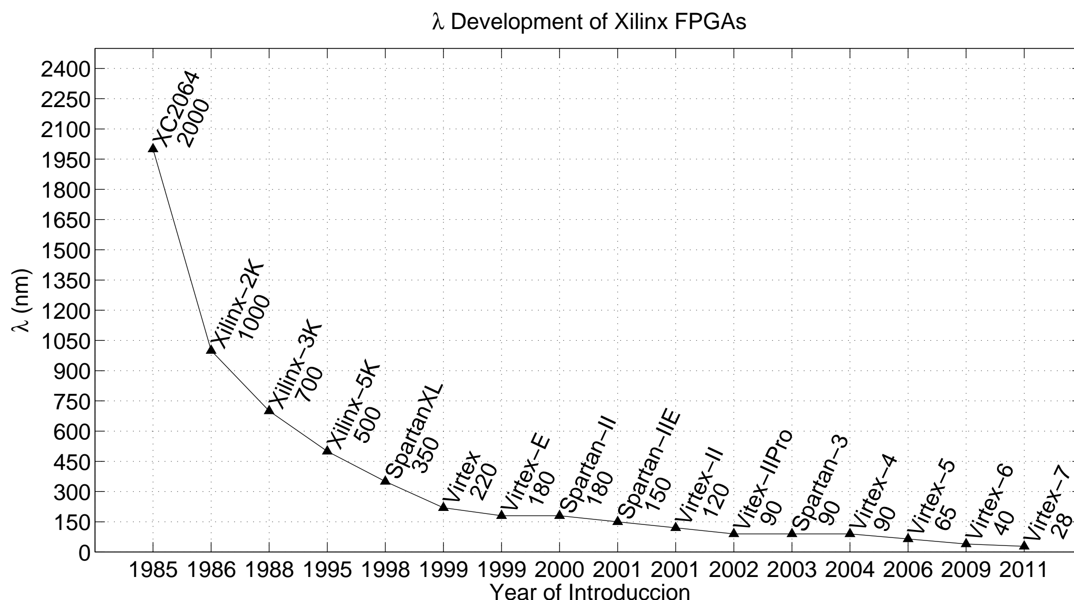


Figure 2.11: Overview of the development of the feature size of FPGAs from Xilinx

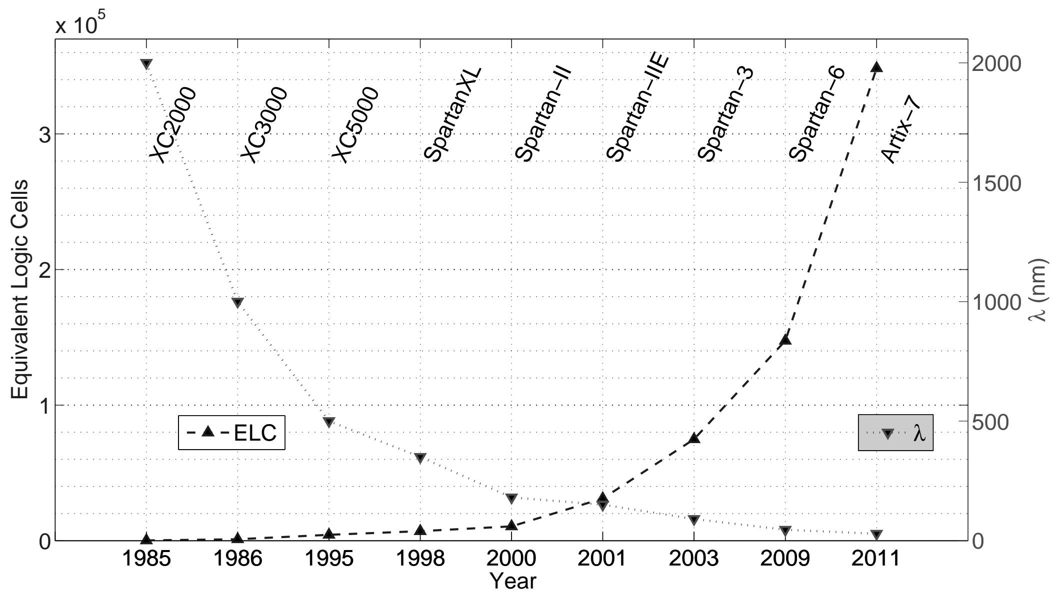


Figure 2.12: Overview of the development of Low-cost FPGAs from Xilinx

converters. Motion control refers to the implementation of algorithms for obstacle avoidance, acceleration profile generation, or route planning, usually related to robotics or Computerised Numerically Control (CNC) machines.

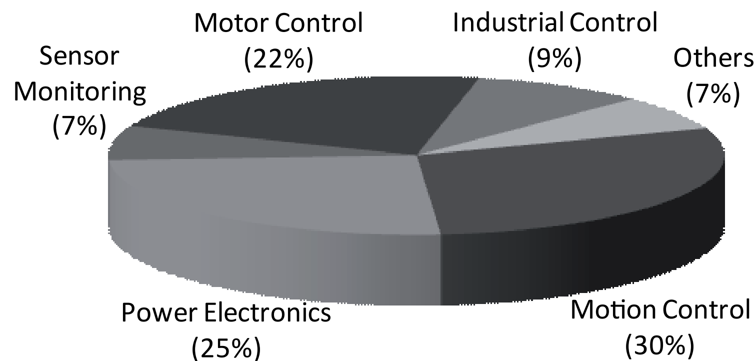


Figure 2.13: Application distribution of the reviewed papers

The utilisation of Reconfigurable Hardware (RH) to implement industrial controllers for production lines as replacement of Programmable Logic Controllers (PLC) is also presented in literature. Furthermore, publications reporting sensor monitoring applications were reviewed, where the main focus is the hardware-based implementation of algorithms to process data from sensors. Other applications include the realisation of fuzzy controllers for temperature control [Jua05], PID-based control of an electro-

static levitation system [Nak02], or embedded controllers for automotive applications [Chu02].

In the reviewed articles (more than 100 papers from many scientific journals and conference proceedings), many authors report the use of FPGAs instead of traditional platforms. The reasons are analysed in the following section.

2.3.2 Factors of the Technology Migration

The utilisation of FPGAs instead of other architectures is mainly based on four factors: the acceleration of the design or parts of it, the flexibility of reconfigurable hardware, the reduction of development costs, and energy consumption. These factors have a different effect on each application area, as depicted in figure 2.14.

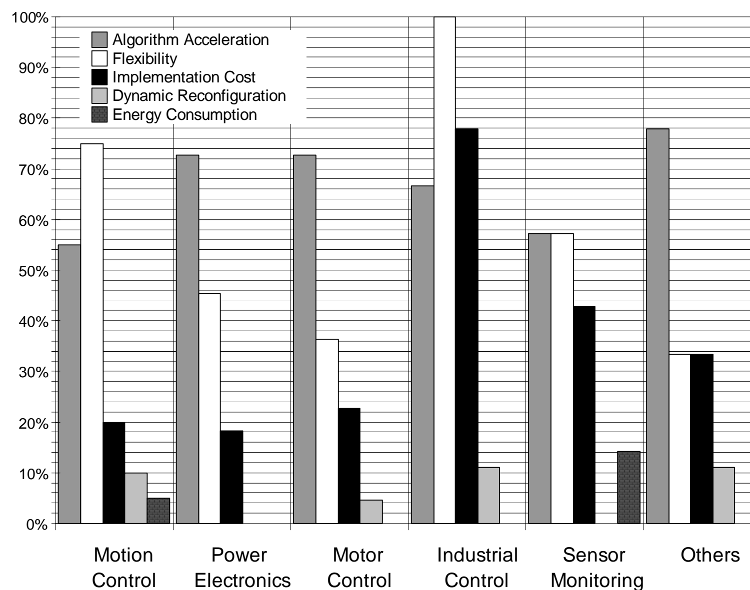


Figure 2.14: Distribution of the contributing factors of FPGAs in the application spectrum

In the review process, a score was given to each factor per publication. Therefore, one publication might report more than one contributing factor. The use of run-time hardware reconfiguration is also included in figure 2.14, and is analysed in section 2.3.4. The reported effect of each factor on the implementation of digital controllers is analysed in the following section, giving specific examples.

Controller Acceleration

Using reconfigurable hardware to accelerate algorithms has been extensively reported (e.g., for digital signal processing [Tes01]). In 58% of the reviewed papers, algorithm acceleration is described as one of the main contributions of RH to the implementation of digital controllers. The term acceleration implies a faster hardware or hardware/-software (Hw/Sw) realisation of a given design in contrast to a software equivalent. This was achieved through different means, such as parallel processing, reduction of the computation overhead or heavily pipelined realisations. For realisations using Hw/Sw partitioning on-chip co-processing was exploited.

Parallel Processing. In contrast to software architectures (e.g., processors), a hardware realisation with various processing elements operating in parallel can achieve a better performance. However, the extent in which this feature can be exploited is highly dependent on the intrinsic parallelism of the algorithm to be realised. Therefore, it is meaningful to detect the amount of concurrency in early stages of the design flow [Nao04, Cha04].

Parallel processing was used in applications such as stepper motor control [Car03]. The utilisation of a Xilinx XC4006 FPGA resulted in an increment of the reachable motor speed due to a faster processing. The required sampling period was 800 *ns* and the clock frequency was set to 40 MHz. In [Zum03] parallel processing was used to accelerate a digital controller of an AC-DC Converter using a Xilinx XC4010 FPGA with a clock frequency of 20 MHz. To realise the same design using DSP technology, a much higher clock frequency would have been required. A similar approach was presented in [Her04] to process information from an ultrasonic ranging sensor. A Xilinx XCV1000E FPGA was used to implement the algorithm, achieving a sampling period of 235 μs at a clock frequency of 50 MHz. The availability of independent processing elements in combination with embedded processors, embedded multipliers units and block RAM made possible the realisation of distributed computation, leading to the reported algorithm acceleration. A similar approach was presented in [Yao10] for speed control of turbines. A PID-Fuzzy controller was implemented, reaching execution times well beyond CPU-based realisations. A direct hardware realisation of a Fuzzy controller was reported in [Che11], for a DC-DC controller of a photovoltaic system.

Reduced Computation Overhead. The utilisation of dedicated hardware reduces the required computation overhead of general purpose architectures. Moreover, many operations, such as bit shifting or multiplication and division by a power of two, are done implicitly when realising them in hardware.

Design specialization was used in [Ada00] to implement industrial controllers as a replacement of Programmable Logic Controllers. In [He04], application specific

hardware accelerated a controller for a robotic hand with multiple motors. By using a combination of dedicated hardware and software it was possible to achieve a sampling period of 200 μs having a clock frequency of 150 MHz. The FPGA implementation of an iterative algorithm for time optimal control (TOC) of AC drives was presented in [Bol04]. The utilisation of an Altera EP20K200EFC484-2X with a sampling rate of 16 μs at a clock frequency of 2.2 MHz allowed the realisation of time-critical parts (e.g., equations including trigonometric operations) of the TOC algorithm in hardware to accelerate the design.

Implementation of specialised processors for control algorithms was presented in [CP01]. A specialised architecture to realise state-space based controllers was reported. This architecture outperformed various commercially available DSPs and required a low gate count for its implementation. A similar approach was presented in [MV10b] for motion controllers of CNC machines. A specialised processor architecture allow a better performance than DSP- and PC-based realisations.

For designs requiring Hw/Sw partitioning the utilisation of soft-core processors (e.g., MicroBlaze from Xilinx) reduced the communication overhead, in comparison to systems with external chip couplings. Although the reported designs used only soft-core processors, modern FPGAs integrate embedded hard-wired processors and DSP units (e.g., the Virtex-II Pro FPGA from Xilinx, with two PPC processors and 192 18x18 hard-wired MAC units), which can achieve higher clock frequencies than their soft-core counterparts. In addition to block RAM, configurable logic can be used as memory. Thus reducing overhead, in comparison to a scheme in which data has to be stored off-chip.

On-chip co-processing was used for motion control of autonomous mini-robots [Rog03] where the soft-core RISC processor NIOS was used to perform various control-flow oriented operations, such as network monitoring and interfacing. In [Pou04] an adaptable thermal compensation system for strain gage sensors was presented. A NIOS processor was used to perform floating point arithmetic operations. Robotic arm manipulation [Kun05] was performed using an Altera Stratix EP1S10 FPGA. A NIOS processor was used to realise those parts of the position control algorithm that required a low sampling frequency. Similarly, in [BO10] the realisation of a PID-controller as hardware accelerator of a multi-processor on chip (MPoC) architecture for motor control was presented. A similar approach was presented in [Kun10] for motion control of a three-axis wafer-handling robot.

Heavily Pipelined Realisations. This factor can augment the throughput of a design at the cost of introducing some time-delay and using more hardware resources. This technique was used in applications such as current vector control of AC machines [Taz99], where a sampling period of 50 μs was obtained. A pipelined realisation was also used for image processing for robotic motion control [Bol01]. 285 images per second, of 9728 pixels each, could be processed during 60 ns as part of a motion

detection algorithm. According to the authors, the achieved performance could be easily improved by adding more stages to the pipeline.

Flexibility

An architecture is said to be flexible if it can be modified to meet new requirements. This feature is usually related to software architectures, where such an adaptation is done by replacing the instructions controlling the central processing unit (e.g., an ALU). This feature is called programmability. Flexibility is then mostly related to the binding time of a given architecture, that is, the time when the functionality of a device is specified. Describing the architecture spectrum as a function of its binding time, one extreme would be occupied by pre-fabrication operation binding devices (e.g., ASICs) and the other extreme by cycle-to-cycle operation binding devices (e.g., processors)[DeH99]. SRAM-based FPGA technology is considered to be flexible because it has a late binding time. (cf. figure 2.15).

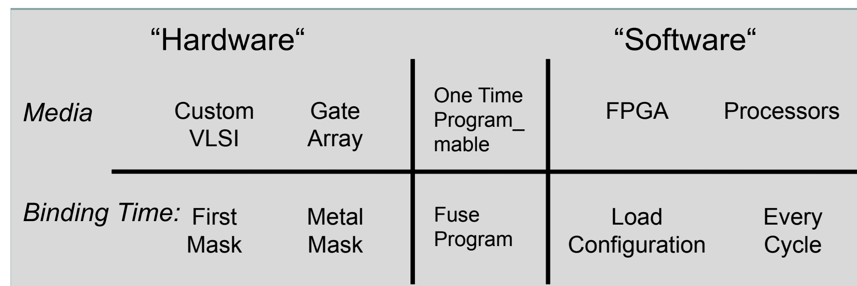


Figure 2.15: Binding time continuum [DeH99]

Flexibility was one of the most reported features of RH; 54% of the publications reported flexibility as a deciding factor to have chosen RH as implementation platform. However, not all authors defined flexibility in the same way. Two factors were considered in this review as contributing to make RH a flexible platform: hardware reconfiguration and the possibility to realise Hw/Sw partitioning on the same platform.

Hardware Reconfiguration. The more specialised an architecture is, the more efficient it performs. However, changes in the application can strongly lower the reachable performance, even for general purpose architectures with a certain level of specialization. DSPs are a good example. They were originally designed to couple with operations in which the same operands are applied to a certain number of values (vector operations). When required to execute operations such as look-up tables, tree searches, or sum of absolute differences, these specialised units (e.g., MAC units) can not be optimally exploited, resulting in a loss of performance. In this sense, the algorithm has to be adapted to meet specific processor architectural characteristics.

Like general purpose processors, RH architectures have a late binding time. This allows the adaptation of reconfigurable devices to the application [Ada00, Sag04, Rue03a, Che02, Li03, Cho01, Che00], usually reaching a better performance than a software equivalent. If the application should change, the new requirements can be handled by accordingly reconfiguring the design [Tho99, Her04, Kou05, Kel97], optimizing it to different possible situations [Bol01, Cho01, Yin04, Ric03]. The precision (e.g., the bit-width) of the design can be changed [Fan05] accordingly to the requirements, making the design scalable [Dep04]. There are spatial limitations (e.g., chip area) that constraint the complexity (e.g., the necessary resources) of a design that can be implemented on a RH platform. However, there are also methods that help to overcome this limitation. Namely, run-time reconfiguration [2, 3, 6, 7, Nas04, He04, Chu02, Dan03a, Dan03b], analysed in section 2.3.4, and in detail in chapter 4. This feature gives a new degree of freedom to the design space of embedded systems, since it is possible to trade speed and area in run-time.

On-Chip Hw/Sw Realisation. FPGAs allow a Hw/Sw realisation on a single chip. This is possible by using configurable logic to realise dedicated hardware in combination to embedded soft- or hard-wired processors. This feature opens new possibilities in the design-space, offering a higher level of flexibility than pure hardware or pure software platforms. Parts of a control algorithm having high diversity of operations and requiring a low sampling frequency are better implemented in software, while other computational intensive parts of the algorithm are better implemented as a dedicated hardware. In [Kun05] Hw/Sw partitioning was used to realise an architecture for robot arm control, or in [Cab04] to implement fuzzy controllers. Similar approaches were presented in [Pou04, Rog03, Pat10b, Pat10a, Kun10]. Furthermore, by using run-time reconfiguration (see section 2.3.4), the Hw/Sw partitioning can be adjusted dynamically (software tasks can be realised as hardware tasks and vice versa), in order to adapt the resource-availability to the requirements of the system, as suggested in [3] for robotic applications (this topic is analysed in chapter 4).

Costs

Having specific requirements for a given application, such as a minimum sampling rate or a desired functionality, the choice of an implementation platform is a matter of finding a suitable compromise of the factors involved in the design process, e.g., total price, performance, or safety. The cost was reported in 31% of the reviewed papers as an important reason to use RH instead of other technologies. In 46% of those papers the comparison was against DSPs, 9% against PLCs, 7% against general purpose processors, and 38% did not report the replaced technology. The key factors for preferring RH are the cost/performance ratio, hardware description, time-to-market and development cost. These factors are analyzed in the following sections.

Cost/Performance Ratio. Considering not only the price of a single device when choosing the implementation platform, but the cost/performance ratio [Nas04, CP01]. It was shown that although a single FPGA chip might be more expensive than a single DSP, the performance that the former can reach is potentially grater [Tom04, Rey04] leading to an overall lower price when considering how many DSP units would be needed to reach the same performance.

Berkeley Design Technology Inc (BDTI) made an analysis based on an orthogonal frequency division multiplexing (OFDM) benchmark [Alt05], in which two Altera FPGAs, the Stratix 1S20-6 and the IS80-6, and a Motorola MSC8101 DSP were compared. The report showed a better cost/performance of the FPGAs, despite the fact that the DSP had a lower cost than both FPGAs.

Cost/performance ratio was reduced by the use of specialised design techniques, which allowed the utilisation of cheaper devices for the implementation. In [MI04] a control algorithm for a switching DC converter was realised using FPGA technology. Because of the specialisation of the design, it was possible to replace a high-resolution ADC converter, required for most DSP calculations, with comparators, leading to an overall price reduction. A similar approach was presented in [Ben99] for high-performance thyristor gate control for line-commutated converters. The design specialization leaded to a low-cost FPGA implementation, avoiding the use of several DSPs. Similar implementations were presented in [Don03]; the use of specialised hardware designs (e.g., a parallel FPGA implementation) allows the utilisation of simpler RH architectures, which are often cheaper than a corresponding software-based solution. Furthermore, the specialisation of a soft-core processor presented in [MV10b], allowed the realisation of a system identification algorithm for a motion controller of a CNC machine.

Hardware Description. The design of controllers for general purpose processors or DSPs usually starts with an abstract, high-level design entry (e.g., C code), which is then translated automatically into an executable format. This tool flow enables the design engineer to focus on the control algorithms without having to deal with the underlying architecture. Expert programmers can optimized critical parts of the design by manually inserting lower level code (e.g., assembly code). Hardware description can also be realised at different abstraction levels; from a register transfer level (RTL) to a behavioural description by using languages such as VHDL or Verilog [Zum03, Aco02, Rei03], providing technology independence [Mat05, Bol04]. Hardware Description Languages (HDL) might not be suited for engineers already used to the design flow of software architectures. For such engineers there are various C-like HDLs [Tom04, Aco02], such as System-C [Sysa], or Handel-C from Celoxica [Han], which integrate the necessary features to describe hardware (e.g., parallel constructs). These HDLs are supported by compilers and synthesis tools, providing a way to generate either VHDL code or netlists from the original script.

In the last decade, several manufacturers of configurable hardware introduced very-high-level hardware descriptions, such as System Generator from Xilinx [Xil08c], DSP Builder from Altera [DSP] or Synplify DSP from Synopsys [Synb], which can be used within Matlab/Simulink. Outcome of these design flows is a structural description of the design, which can be mapped (synthesized) onto an FPGA. Such hardware descriptions provide a higher abstraction level than traditional HDLs and C-like HDLs, thus reducing the design effort [2, Nao04, Cha05b].

The utilisation of Intellectual Property (IP) cores, visual programming languages, and designs reusability methods facilitate the implementation of complex system on chip (SoC)[Don03, Old05, Rey04]. Hardware description has a direct impact on the design effort required to complete a design. This in turn influences the required time-to-market (TTM), which is discussed in the next section.

Time to Market. Introducing a product late into the market could lead to a potentially lower revenue. A simplified model presented in figure 2.16 suggests that the lost can be estimated by the equation 2.3 [Des06].

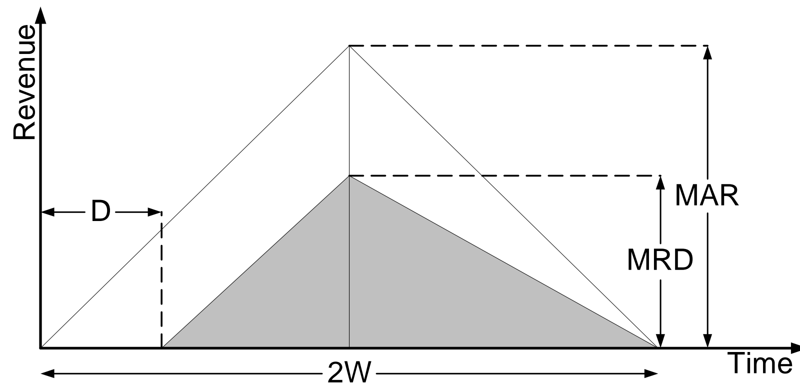


Figure 2.16: Cost of delayed entry into a market [Des06]

$$MRD = \frac{D(3W - D)}{2W^2} MAR \tag{2.3}$$

Where: D is the delay, W is half of the production life-span, MAR is the maximum available revenue, MRD is the maximum revenue from a delayed entry. As an example, if a product A has a lifespan of 3 years (36 months) with an estimated $MAR = 50$MD and has a delay of 3 months, the cost of that delay is a loss of 23.61% of the original revenues estimate, that is a $MRD \cong 38.2$MD. In literature it was claimed that the use of RH can potentially avoid a late product delivery.

It was said that when using RH, TTM can be drastically reduced in comparison to an ASIC lead-time. The exclusion of some design steps inherent to the ASIC fabrication (e.g., mask generation, silicon fab, silicon verify) and the parallelisation of software and hardware development due to early system prototyping enables the reduction of TTM. This allows a quick implementation of complex algorithms [Ho00] resulting in a shorter TTM [Aco02, Kim00]. TTM has been further reduced by the introduction of intellectual property (IP) blocks [Bol04], and high abstraction level HDLs (see discussion in section 2.3.2), which allows the integration of optimized ready-to-use blocks into the design. A modular design strategy [Cha04] also contributes to shorten the design time, and thus reach a shorter TTM.

Design verification is time- and resources-consuming. The total cost of verifying the functionality of a design might by shortened by the introduction of Hardware-in-the-Loop (HIL) simulations [Ise99]. For software based designs, this technique has reduced the gap between controller design and implementation in the final platform. For FPGA-based controllers, the principles of HIL can also be exploited, resulting in a speedup of the simulation process and providing a cycle accurate verification of the design [4]. This topic is further analysed in chapter 5.

For high-performance applications, for which using many CPUs concurrently to reach a desired throughput is required, two situations can arise: the overall development cost increases due to the utilised extra processing units and the software routines that allow multi-tasking and parallel processing become difficult to handle [Mon99]. Depending on the engineer expertise, this situation could lead to a longer design cycle than expected.

Development Costs. When using RH, it is possible to realise most of the required functional blocks on the same chip, which avoids to use many discrete elements [RT04, Car03, OR09, OR08], and reduces the required board size and the energy consumption [Kel97]. The availability of low-cost and large-capacity FPGAs, an increasing number of intellectual property (IP) modules, and powerful CAD tools enables the development of a whole system on programmable chip (SoPC) [Cab04, Pat10b, Pat10a].

This technique was used in [Rue03a] to implement an FPGA-based emulator for series of multi-cell converters. The integration of observers in the design permitted a sensorless implementation, reducing the overall cost. Similarly, in [Kun05] the utilisation of FPGA technology allowed the implementation of all necessary computing elements to control a vertical articulated robot arm. In [Li03] this approach was used to implement an FPGA-based fuzzy behavior control for an autonomous mobile robot.

The development costs of RH-based realisations was also compared to that of ASIC realisations [Aco02, Chu02], which were avoided because of the implicated high costs for low-quantity productions. In [Old05, Old01] an FPGA-based servo control was

presented. The advantage of FPGAs was said to be that custom parallel processing architectures can be embedded on a single device, without incurring the high NRE costs and re-spins associated with ASIC development, for low-quantity productions.

Energy Consumption

The energy consumption of a system may be a critical factor when choosing an implementation architecture, specially for systems running on limited energy supplies (e.g., batteries) or with heat dissipation constraints. When using CMOS technology the total account of energy consumption depends on the static and dynamic power dissipation. The main cause of static power dissipation is leakage, which is largely determined by the device type, operating temperature and process variations. The dynamic power consumption is completely design-dependent, and is determined by factors including resource utilisation, logic partition, mapping, placement and routing. The designer has influence mainly on the dynamic power dissipation. Due to the energy overhead required for routing resources and configuration memory FPGA-based designs have a higher power dissipation than ASICs.

In [Kel97], a methodology was presented to implement state-space based controllers using FPGA technology. The use of this technology was said to reduce the energy consumption by 50% when compared to designs built out of many integrated circuits, because the capacitive loads were lowered. In [Scr02] a comparative study of the energy efficiency of FPGAs and programmable processors for $n \times n$ matrix multiplication was presented. The measurements showed that a Virtex-II Pro FPGA (from Xilinx) achieved the shortest latency and used less energy than a TMS320C6415 (from Texas Instruments) and a PXA250 (from Intel) for this specific task. However, the Virtex-II Pro performed the worst under a different configuration. Energy consumption of FPGA-based designs is further analysed in chapter 3.

2.3.3 Coupling of Reconfigurable Hardware and Software Architectures

For applications with Hw/Sw partitioning, RH can be classified according to its levels of communication as: co-processor, attached processing unit, standalone processing unit, or RH with embedded processor [Tod05] [Com02] as depicted in figure 2.17.

Although the majority of the reported works were implemented as application specific designs (42 %, see figure 2.18), the use of Hw/Sw partitioning was a widespread design approach. Each kind of coupling is described in the following sections.

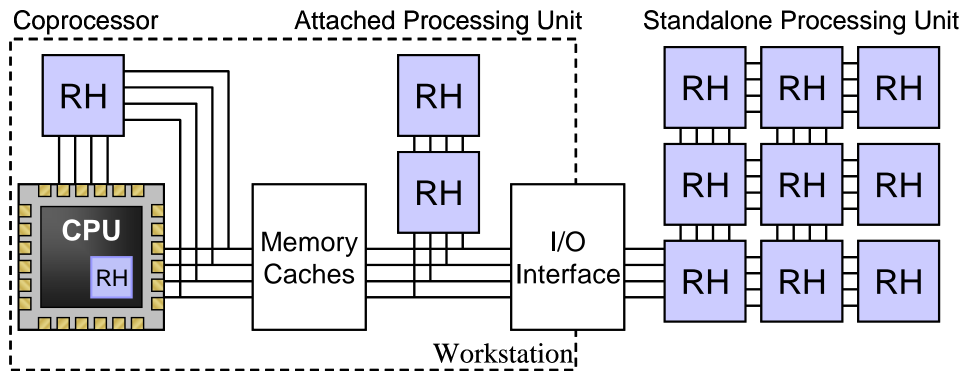


Figure 2.17: Different levels of coupling in a reconfigurable system[Com02]

RH as a Coprocessor

In this kind of coupling, RH is able to perform operations without the constant supervision of a host processor. Usually RH is used to realise computational intensive operation and sends the resulting information to the host processor. This generally allows the processor and RH to run parallelly. Examples of this approach are a position measurement algorithm, reported in [Lyg98], in which an Altera FLEX was coupled with a Texas Instruments (TI) DSP TMS 320C31. In [Jun99], a PWM controller for DC/AC converter was realised in a Xilinx XC4005 and coupled to a TI DSP TMS 320C14. Moreover, an algorithm for adaptive motion control, described in [Gwa02]

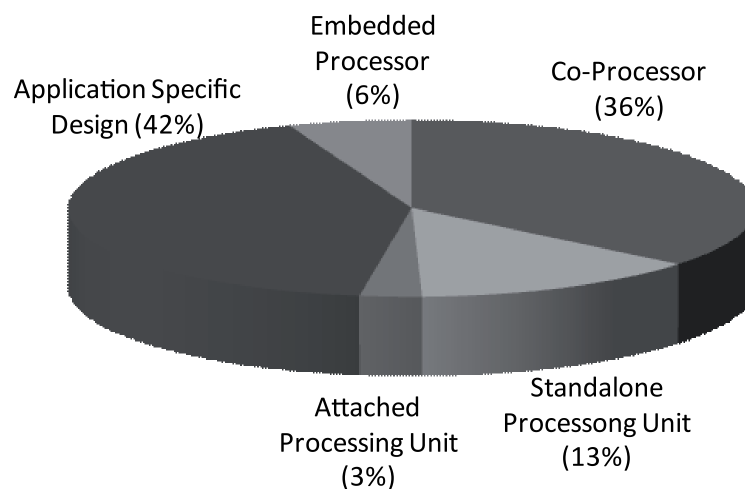


Figure 2.18: Classification of reconfigurable hardware according to the reported coupling

was realised in a Xilinx XCV300 coupled to a DSP. In the aforementioned publications the DSP unit realised control flow oriented task (e.g., monitoring).

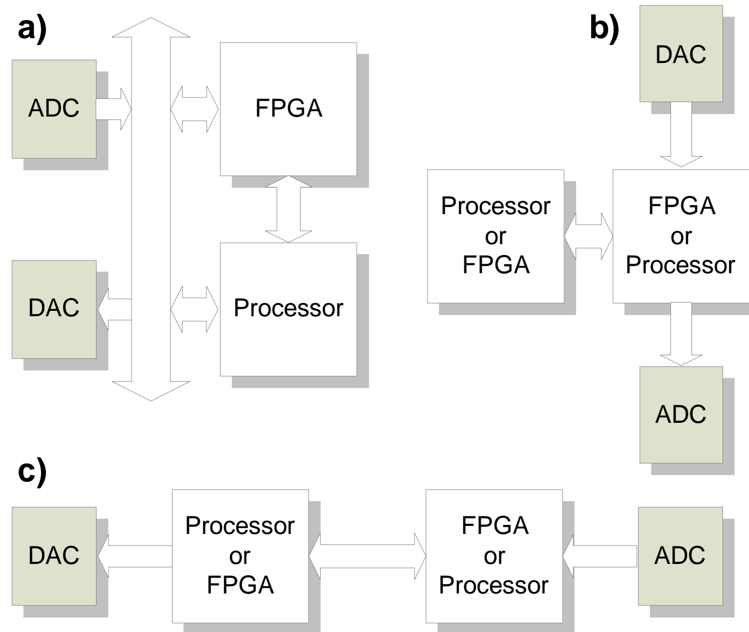


Figure 2.19: Reconfigurable hardware and processor couplings in reference to the I/Os. a) Processor and FPGA have access to the ADC and DAC. b) Either an FPGA or a Processor is directly connected to the ADC and DAC with a tight communication with the other processing element. c) Either a Processor or an FPGA receive information form ADC and the other processing element is connected to the DAC.

There were different co-processor configurations in which RH and software architectures were coupled as depicted in figure 2.19. In configuration (a) both processor and RH have the same level of connectivity to the I/Os. This makes the partitioning of hardware and software tasks more flexible. In configuration (b) the architecture connected to the I/Os (either RH or a software platform) performs most of the control tasks, assigning the other device to supervisory tasks. In most of the reviewed papers, an FPGA was directly connected to the I/Os. In configuration (c) the device connected to the inputs realises data conditioning tasks (e.g., units transformations, data scaling, digital filtering). The device connected to the outputs realises most of the control tasks. In the majority of the reviewed papers a processor was connected to the inputs and an FPGA to the outputs.

Attached Processing Unit

In this kind of coupling the communication between RH and the host processor is less frequent than in the previous kind. Usually RH was used to execute parts of the controller, an then either to send information back or to send a command directly to the plant. FPGAs were used in [Din05] to generate high-frequency PWM signals for a voltage-source-converter-based distribution static compensator. A DSP (TMS320C40 from TI) was coupled with an FPGA (FLEX8000 from Altera) via a bus system. The control system was implemented using the DSP and the multiple-PWM generator was implemented in the FPGA together with a dead-time insertion module. The data exchange was limited to the required duty-cycle. A similar approach was reported in [Faa04]. The utilisation of radial functions neural networks (RFNN) to implement a sensorless motion control algorithm for spindle motors was presented. A synchronous PWM with dead-time compensator was done in a Xilinx FPGA, which was coupled to a Pentium-based PC. Another example was reported in [Jia02]. The implementation of a servo drive for an ultrasonic motor was presented. The controller was implemented in an Altera 10K100 FPGA, which was coupled to a Pentium II-based PC. The communication between PC and the FPGA board was limited to supervisory tasks.

Stand Alone Processing Unit

This is the loosest coupling. The communication between the host processor and RH happens infrequently. It can be compared to the kind of communication among workstations in a network. An example was presented in [Han02], in which the implementation of a genetic algorithm to set up a neural-controller was realised on several Altera FLEX EPF8452 FPGAs, coordinated by a PC. Another multi-FPGAs design was presented in [Bol01]. Several FPGAs were used simultaneously to realize an image processing algorithm for movement detection, used in a mobile robot. A Pentium-based PC coordinated the computations realized on the FPGAs. The use of RH allowed the acceleration of the computational intensive image processing algorithm.

RH with Embedded Processors

Another approach to Hw/Sw design was to include an embedded soft-core processor instead of an external unit. The integration of the processor inside the FPGA reduced communication overhead, costs and board size, as discussed in section 2.3.2.

Hw/Sw partitioning of the implemented design was often reported in literature. The method to realise the partitioning was rarely mentioned. However, it was observed that in general control flow operations, such as variable-length loops or branch control were realised using a software platform, for instance operations whose hardware

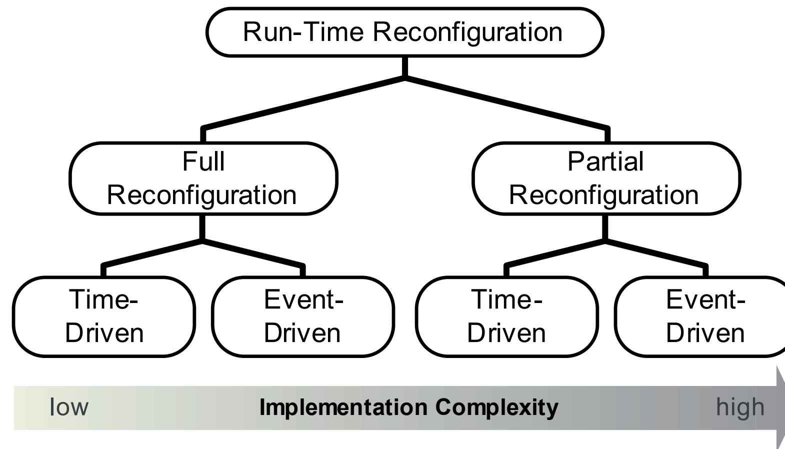


Figure 2.20: Reported reconfiguration types for digital control applications

implementation were too expensive (e.g., floating point arithmetic). Parts of the algorithm, which could be benefited from a hardware implementation, such as fixed path operations or parallel operations, were implemented in reconfigurable logic.

2.3.4 Run-Time Hardware Reconfiguration

Run-time reconfiguration (RTR) is a scheme in which an FPGA is partially or fully reconfigured during the execution of a task, where a task consists of one or more sub-modules. When the FPGA is only partially reconfigured the process is called *partial run-time reconfiguration* or *dynamic reconfiguration* (DR). Both full RTR and DR can be used to enhance the resource-utilisation of an FPGA by time-sharing logical resources among non-periodic designs (*event-driven* reconfiguration) or by periodically time-multiplexing sub-modules of a design (*fixed schedule*) [3].

A classification of RTR schemes used to implement digital controllers is depicted in figure 2.20. The realisation of a RTR architecture becomes more complex depending on the reconfiguration kind (full or partial), and the reconfiguration scheme (*fixed schedule* or *event-driven*)

Fixed Schedule RTR

In this scheme a design is partitioned in sub-modules, which are loaded on the FPGA periodically in a pre-defined sequence. This scheme was used in [Kim00] to implement fuzzy controllers. The authors divided the design into many temporally independent functional modules and performed full RTR sequentially with the bitstreams corresponding to each module, saving intermediate results in an on-board SRAM. Each module contained an SRAM interface and a bus interface. The reconfiguration was

controlled by a host computer. In [Nas04] DR was used to implement a controller for an elevator using the *fixed schedule* scheme. The FPGA was divided into a static and a dynamically reconfigurable area. Intermediate results were stored in a register bank of the static area.

Event-Driven RTR

In this scheme the reconfiguration time is not known during the design phase, requiring the implementation of a more complex configuration manager. In [Vas04] *event-driven* full RTR was suggested to implement a fault tolerant control system for AC Drives fed by tandem converters. Full RTR was chosen because of the technological complications of realizing partial RTR. *Event-driven* partial RTR was used in [Dan03b] to implement a multi-controller approach for linear time-invariant systems. The configuration of the FPGA was realised via a host computer. In [Tos05] the use of this scheme to implement the controller of systems having different functions (e.g., normal operation, test, etc.) was suggested. Another approach was presented in [Chu02], where *event-driven* partial RTR was used to implement a fault tolerant controller for automotive applications. An architecture including a reconfiguration controller, memory to store partial bitstreams, and a module to switch between controllers was presented.

Although the potential benefits of using DR are known, this scheme is still largely unexplored for control applications, and constitutes one of the research points of the present work. In chapter 4, this topic is analysed in detail, addressing topics such as resource utilisation, control disturbances, and showing case-studies.

2.4 Summary

This chapter offers relevant background on digital control implementation, pointing out the key differences between a software-, an ASIC-, and an FPGA-based design flows. Of central interest is to review the state-of-the-art on the utilisation of reconfigurable hardware for embedded control applications. In this chapter it is shown that FPGAs, the most well known architecture of reconfigurable devices, is leaving its role as prototyping platform to become a target computing architecture. This technology migration is mainly based on the increasing complexity of embedded controllers, and the fast evolution undergone by FPGAs in the last decade. This evolution brought mainly four factors: the possibility to accelerate a controller (i.e., achieve a higher throughput than a software based design), a flexible platform (e.g., the architecture can be adapted to the application), reduction of implementation costs, and reduction of energy consumption, all of them analysed in section 2.3.2.

Although many papers report implementation results, the great majority lack of a quantitative comparison to support claimed advantages of FPGA-based realisations. Furthermore, many authors do not clearly define the advantage of using FPGAs (e.g., the feature flexibility was related to the hardware description language in many publications (e.g., [Han03, Zum03]). Furthermore, the reasons that make control algorithms to benefit from a realisation using FPGA technology were not fully analysed. This lack of insight motivates the first research problem undertaken in this thesis: a quantitative technology comparison focused on control applications, discussed in chapter 3.

As exposed in this chapter, FPGA technology offers new possibilities to the implementation of digital controllers. However, there is a special feature of modern FPGAs that has not been fully explored for control applications: the possibility to partially or fully reconfigure an FPGA in run-time, known as run-time reconfiguration (RTR) or dynamic reconfiguration (DR). The use of run-time reconfiguration was presented, showing the growing interest on this scheme but also the lack of methodologies to apply it to the field of embedded control systems. Chapter 4 addresses this topic, showing the advantages of using a run-time reconfiguration in contrast to a static approach.

Design verification is an important part of the design flow of electronic systems. This approach has been largely used in software-based design flows. Chapter 5 explores this aspect of the design flow of FPGA-based controllers, and presents two frameworks to verify a FPGA-based design: HiLDE (Hardware-in-the-Loop Design Environment) and HiLDEGART (HiLDE for Generic Active Real-Time Test).

3 Technology Comparison of Reconfigurable Hardware and Software Architectures

The choice of a computer architecture to realise a given application depends, on the one hand, on the algorithmic properties and requirements of that application. On the other hand, the choice depends on the characteristics of the considered architectures and how they fit to the application. In the previous chapter a literature review was presented, showing not only the increasing use of reconfigurable hardware to realise control applications, but also the reasons that moved researchers and engineers to use this technologie. It is claimed in literature that reconfigurable hardware offers a better choice than general purpose processor or digital signal processors. However, the factors that make the difference for control applications are not quantitatively shown (see section 2.3). Furthermore, the benefits of using RH, are not analysed in relation to the properties of the application area.

This chapter presents a quantitative comparison between software and hardware platforms, taking also algorithmic properties of representative control applications into account. As a first step, two kinds of metrics are defined: metrics to assess the algorithmic characteristics of controllers, and metrics to evaluate how efficiently resources (e.g., time, area, energy) of that architecture are used (see sections 3.1, and 3.2). Furthermore, computing architectures used for the comparison are selected (section 3.3). Then, a set of representative control algorithms are selected to be used as benchmarks (see section 3.5). The algorithmic properties of the set of benchmarks are then evaluated, as well as the implementation results, using the previously defined set of metrics. Finally, results are analysed and conclusions are drawn at the end of the chapter.

3.1 Algorithmic Characterisation

Understanding the algorithmic properties of a controller is of relevance, because it allows to relate those properties to the choice of the target computing architecture. One important property is the amount of parallelism of a given control algorithm. We can intuitively say that an algorithm with a high degree of parallelism can take advantage of a realisation, which uses many processing units. On the contrary, if the algorithm consist mostly of operations, which have to be executed sequentially, using many processing units for its realisation does not bring any advantage.

The properties of an algorithm have to be analysed based on a mathematical description, without any architecture-specific constraint. Furthermore, an abstract representation is required, which enables us to evaluate algorithmic properties of a controller. One such abstract representation are Cyclic Data Flow Graphs (CDFG), explained in the next section.

3.1.1 Controller Representation: Cyclic Data Flow Graph

Cyclic Data Flow Graphs (CDFG) are directed graphs (i.e., the information flow has a defined direction) where nodes represent operations and edges represent precedence relations among operations. A simple example of a CDFG is shown in figure 3.1. There are five operations in this graph, represented by nodes o_1,o_2,o_3, o_4 and o_5. Data dependencies are well established through edges.

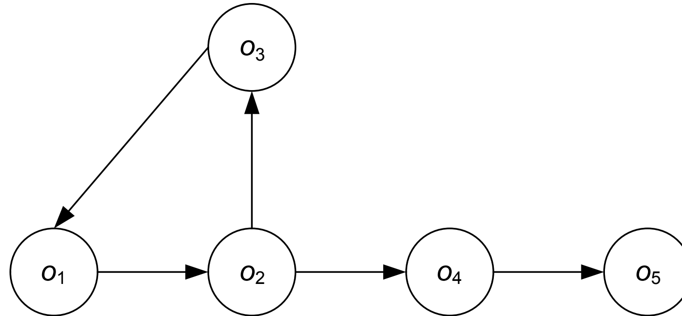


Figure 3.1: Example of Cyclic Data Flow Graph

A CDFG G can thus be described by $G(O,E)$, where $O = \{o_1, o_2, ..., o_n\}$ represents the n nodes of the graph, and E is a set of ordered pairs of nodes, representing the connection between nodes and its direction. For instance, the graph shown in figure 3.1 has $O = \{o_1, o_2, o_3, o_4, o_5\}$, and $E = \{(o_1, o_2), (o_2, o_3), (o_3, o_1), (o_2, o_4), (o_4, o_5)\}$. The same graph can be described by an Adjacency Matrix, which for a graph G with n nodes is an $n \times n$ matrix where the non-diagonal entry e_{ij} is the number of edges from

node i to node j, and the diagonal entry e_{ii} is the number of loops. The adjacency matrix of the graph shown in figure 3.1 is:

$$\text{Adjacency Matrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where the rows and columns represent the nodes of the graph shown in figure 3.1, starting from the top left position.

A CDFG naturally exhibits the parallelism of an algorithm, because it does not impose any constraint other than the data precedence among operations, and therefore maximum concurrency can theoretically be exposed. An important aspect is that the granularity of the operations is independent from this representation enabling a flexible abstraction of the algorithm.

The reason to use this abstract representation is to enable the characterisation of an algorithm. To accomplish this, we need a way to calculate the execution time of a CDFG, and a way to weight node according to the operations they represent. In the next section, a method to calculate the execution time by using a cyclic scheduling is presented.

3.1.2 Scheduling of a CDFG

From a CDFG representation the latency required to complete one execution of the graph can be computed using a cyclic schedule [Šůc09]. A cyclic schedule for the execution of nodes $O = \{o_1, o_2, ..., o_n\}$ assumes that the operations set O is executed a large number of times. One execution of O is labeled with the integer index $k \geq 1$ and is called an iteration and is equivalent to the total latency of the CDFG, which is designated as lat_{alg}. The graph shown in figure 3.1 has to be further extended by assigning processing units and processing times to each node, and by adding communication delays and precedence indexes to the edges [Šůc08]. Edge e_{ij} from the node i to j is labeled by a couple of integer constants l_{ij} and h_{ij}. Length l_{ij} represents the minimal time (i.e., steps) from the start time of the operation o_i to the start time of o_j and it is always greater than zero. Height h_{ij} specifies the dependence distance, i.e. the variation of the iteration index related to the data produced by o_i and read (consumed) by o_j [Šůc09]. The transformation of the graph presented in figure 3.1 is shown in figure 3.2.

In the CDFG shown in figure 3.2 all operations have their own processing element, which is shown together with the processing time in the numbers inside the nodes. For instance, node o_2 has a latency of 1 and is assigned to processor 2. Furthermore,

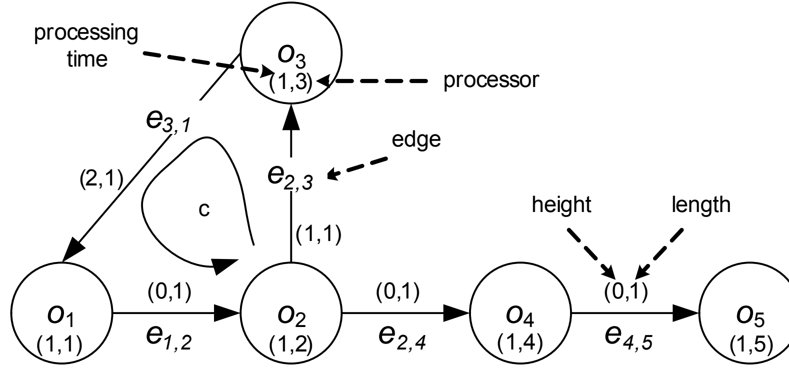


Figure 3.2: Example LH Graph

all edges have a latency of 1 (i.e., $\forall e_{ij} \in G : l_{ij} = 1$), and all edges, except for $e_{2,3}$ and $e_{3,1}$, have an iteration index $h_{ij} = 0$. An iteration index $h_{ij} = n$ means that the dependence distance between node j and node i is equal to n iterations.

Having such a graph, where each node represents an operation and to each operation a processing element is assigned, a cyclic scheduling can be found based on the Basic Cyclic Scheduling (BCS) problem [Han95]. The aim of the BCS is to find a periodic schedule with minimal period w which is also called critical circuit. This is a circuit in G, which maximises the ratio

$$w = \max_{c \in C(G)} \frac{\sum\limits_{e_{ij} \in c} l_{ij}}{\sum\limits_{e_{ij} \in c} h_{ij}} \tag{3.1}$$

where $C(G)$ denotes the set of cycles in G [Šůc08]. In the graph presented in figure 3.2 the critical circuit c is marked with an arrow, and is composed by three nodes $c = o_1, o_2, o_3$, which results in $w = (l_{1,2} + l_{2,3} + l_{3,1})/(h_{1,2} + h_{2,3} + h_{3,1}) = 3/3$. The schedule for this graph is shown in figure 3.3, three iterations are presented. It can be seen that the period of the critical circuit is $w = 1$, whereas the latency of the graph is $lat_{alg} = 4$. The nodes responsible for the latency of the graph (c_{lat}) can be deduced from the schedule presented in figure 3.3, in the example shown in figure 3.2 the latency is the summation of the execution time of the nodes in the critical path, i.e. $c_{lat} = o_1, max(o_2, o_3), o_4, o_5$, where $max(i, j)$ represent the greatest value between nodes i and j. In this examples all nodes have a execution time of 1, therefore $max(o_2, o_3) = 1$.

The scheduling is computed using TORSCHE (Time Optimisation, Resources, SCHEduling), a toolbox of scheduling algorithms for Matlab. The scheduling is based on an integer-linear programming algorithm, with the goal of finding a cyclic

periodic schedule with precedence delays on limited number of dedicated processors (cf. [Šůc06]).

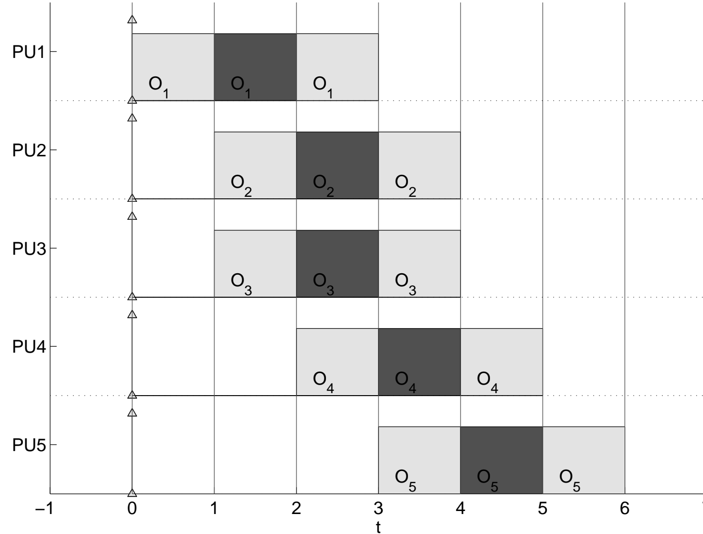


Figure 3.3: Scheduling for the LH graph depicted in figure 3.2. Three execution periods are shown, indicated by the different colors (gray scale)

The cyclic schedule presented in figure 3.3 assumes an equal execution time of all operations in the graph. However, not all operations require the same amount of computational effort for their implementation. In the next section a set of basic operations is presented, and a method to weight them according to their implementation effort is proposed.

3.1.3 Basic Operations Set: Selection and Weighting

The CDFG representation allows a flexible abstraction of an algorithm. An important aspect of this abstraction is the selection of the granularity of the operations used as nodes in a graph. On the one hand, a fine-granular selection (e.g., logic gates) would make the representation of a controller too complex and thus impractical. On the other hand, a coarse-granular selection (e.g., control blocks such as integration or matrix multiplication) would not help exposing parallelism. Therefore, middle-granular operations are used, because it allows to have a small set, it makes the representation of a controller simple, and allows exposing parallelism. A suitable set of basic operations *BO* is presented in table 3.1.

This set of basic operations is still not-homogeneous regarding the computational effort, which is required for their realisation. Therefore, a way to account for these differences has to be found. Using circuit complexity theory [Cor01, Weg87], operations of table 3.1 can be weighted by their time and space complexity value. However,

complexity values tells us only the growth rate of size (e.g., circuit size) and depth (e.g., critical path delay) for a given operation with respect to the used number of bits. This measure is of relevance when dealing with large numbers. However, for embedded control applications, using large numbers is not typical, an therefore, another method to weight operations has to be found.

An empirical method is proposed, where all operations presented in table 3.1 are implemented as an ASIC, varying the number of input bits from 4 to 64 bits. Synthesis results (e.g., circuit area, and critical path delay) are then used to weight nodes of a CDFG according to the operation they represent.

All operations are first described in VHDL, and synthesised using Synopsys Design Compiler (C-2009.06-SP4) [Syna], with a low-power standard-cell 65nm library from ST Microelectronics. All operations are synthesised to achieve the fastest parallel realisation possible, using the design flow for design space explorations described in [Jun10], based on the DesignWare library from Synopsys [Syna]. For arithmetic operations a set of IP cores, listed in table 3.2, are selected, based on the critical path of the resulting realisation, where the fastest IP cores are selected.

Figure 3.4 shows area utilisation of the selected operations. Because the goal is to achieve the fastest realisation possible, area was not constrained during synthesis. As expected, division and multiplication have the greatest area requirements.

Figure 3.5 shows the critical path delay for selected operations in *ns*. The Restoring Carry-Look-Ahead divider has the longest critical path delay as function of the input bits.

3.1.4 Normalised Operations and Steps

Area requirements and critical path delay shown in figure 3.4 and 3.5 can be used to weight operations according to their computational effort. However, these synthesis results have to be normalised to obtain a relative measure of computational effort. The equivalent operation ($NormOp_{h,i}$) measure is defined as the chip area of a given operation $h \in BO$, when using i number of input bits ($SizeOp_{h,i}$), divided by the chip area of a normalising operation $n \in BO$ using the same number of bits ($SizeOp_{i,n}$), cf. equation 3.2.

Kind of Operation	Operations
Arithmetic	Addition, Subtraction, Multiplication, Division
Control Flow	Relational, Switch, Select
Memory Operations	Storage

Table 3.1: Set of basic operations

Operation	Synthesis Model
Addition	Carry-Look-Ahead
Subtraction	Carry-Look-Ahead
Multiplication	Carry-Save Array
Division	Restoring Carry-Look-Ahead

Table 3.2: Used synthesis models from DesignWare library (Synopsys, [SYN11])

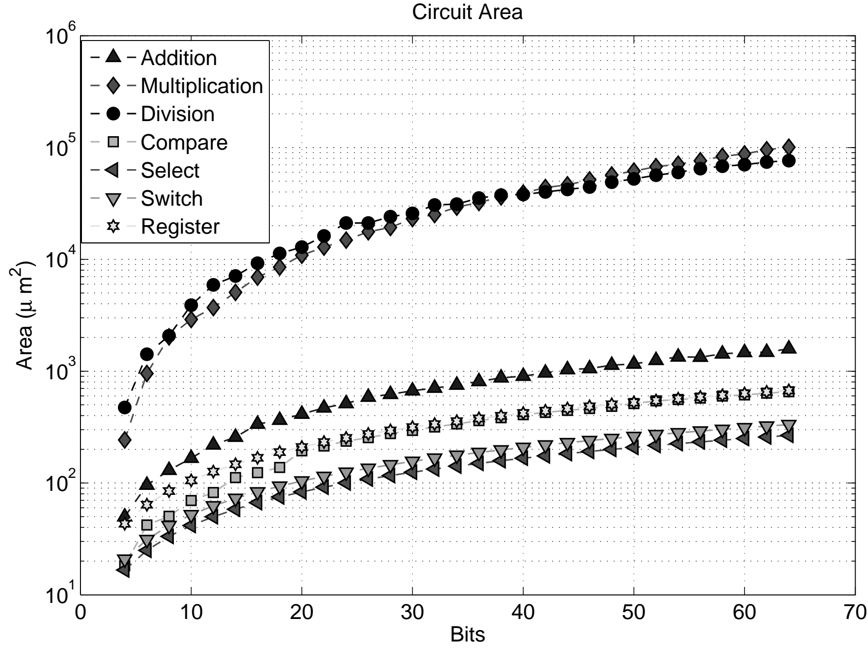


Figure 3.4: Circuit area of operations listed in table 3.1 for various bit-widths

$$NormOp_{h,i} = \frac{SizeOp_{h,i}}{SizeOP_{n,i}} \tag{3.2}$$

The equivalent step ($Steps_{h,i}$) measure is defined as the critical path delay of an operation $h \in BO$ using i number of input bits ($DepthOp_{h,i}$) divided by the critical path delay of a normalising operation $n \in BO$ using the same number of input bits ($DepthOp_{n,i}$), cf. equation 3.3.

$$Steps_{h,i} = \frac{DepthOp_{h,i}}{DepthOp_{n,i}} \tag{3.3}$$

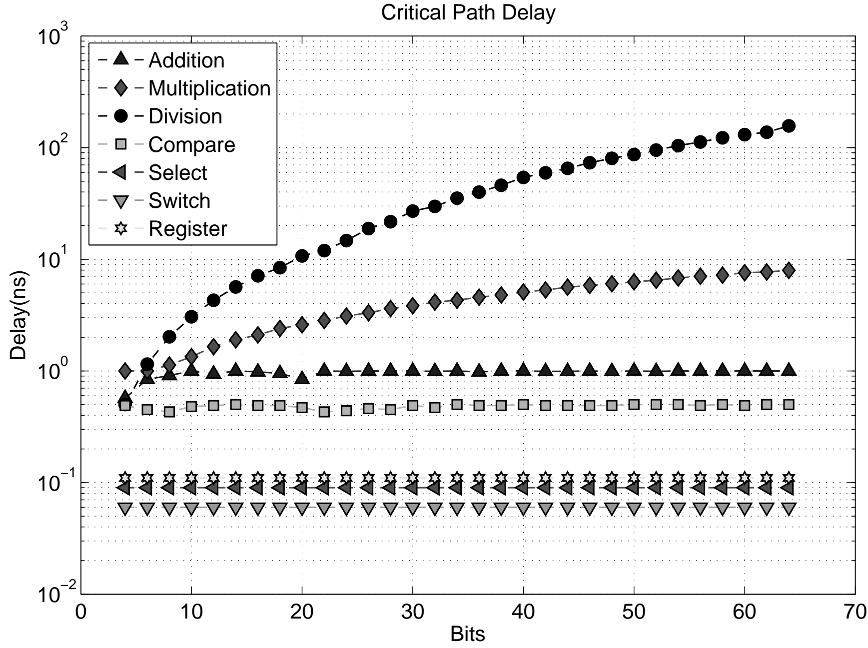


Figure 3.5: Critical path delay operations listed in table 3.1 for various bit-widths

As normalising operation a Carry Look-Ahead adder (see figure 3.4) is used, because it is a common operation to all investigated benchmarks. Furthermore, it scales linearly when increasing the number of input bits.

Using equations 3.2 and 3.3, relative computational effort values can be derived from the critical path delay and circuit area values presented in figures 3.5 and 3.4 respectively, as shown in figure 3.6 and 3.7. These values are used for all investigated algorithm (cf. section 3.5).

Because addition is used as normalising operation, its *NormOp* values is always 1. Similarly, equivalent steps are shown in figure 3.7, where addition has also a constant value of 1 step.

The total number of normalised operations of an algorithm is calculated by adding all individual normalised operations $h \in O$, using i number of bits, cf. equation 3.4. This equation give us a way to compare the size of different algorithm.

$$NormOpAlg_i = \sum_{h \in O, i = bits} NormOp_{h,i} \tag{3.4}$$

Furthermore, the total number of steps required to execute a CDFG representing a specific algorithm is computed by adding the steps of the individual operations in the critical path of the equivalent graph, cf. equation 3.5.

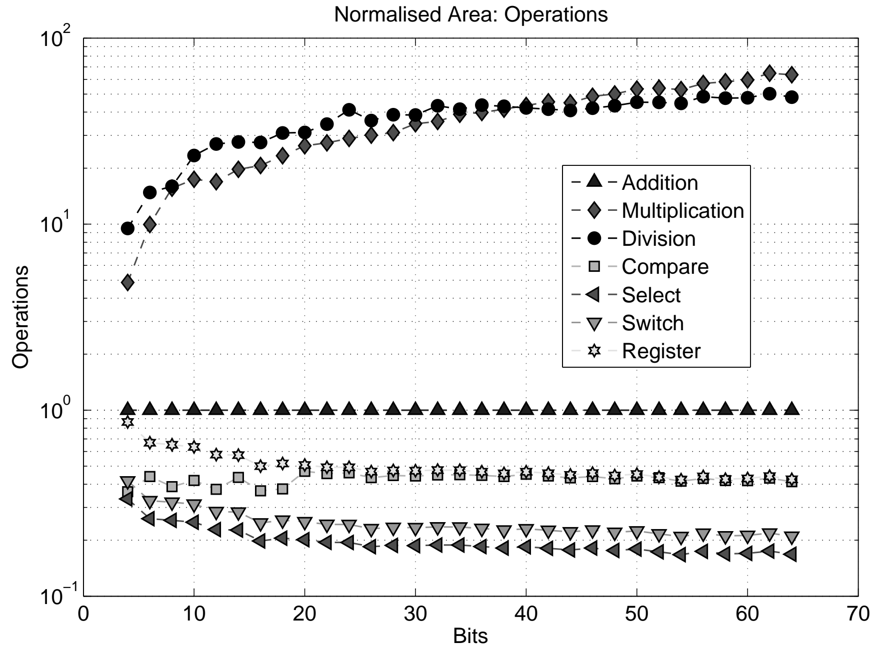


Figure 3.6: Normalised operations ($NormOp_{h,i}$), calculated using equation 3.2

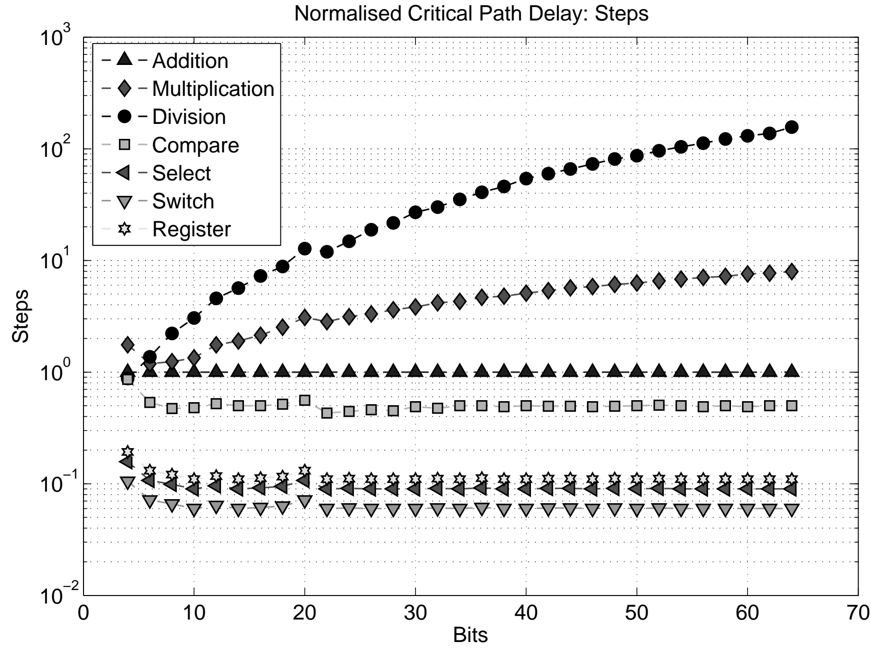


Figure 3.7: Normalised steps ($Steps_{h,i}$), calculated using equation 3.3

$$StepsAlg_i = \sum_{h \in c_{lat}, i=bits} Steps_{h,i} \tag{3.5}$$

Another measure of interest is the equivalent circuit size of an algorithm, which is estimated by adding the size of all individual operations $h \in O$, where O represents the set of nodes of the equivalent graph of the algorithm (see figure 3.4) when using a specific number of bits i, as defined in equation 3.6. This metric provides a very rough estimate of how the circuit area scales, without taking elements into account, which are necessary for an actual ASIC implementation, such as communication resources.

$$SizeAlg_i = \sum_{h \in O, i=bits} SizeOp_{h,i} \tag{3.6}$$

$SizeAlg_i$ is used in this chapter only to validate the proposed metrics with actual implementation values (see section 3.5).

A trapezoidal numerical integration is used as an example to illustrate how the normalised synthesis results ($NormOp_i$ and $Steps_i$) are used to weight operations (i.e., nodes) of a CDFG. The trapezoidal integration algorithm is mathematically expressed by equation 3.7.

$$\int_a^b f(t)dt \approx \frac{a-b}{2}(f(t)+f(t+1)) \tag{3.7}$$

The time-discrete approximation of the integral is given by:

$$i(k) = i(k-1) + \frac{\Delta k}{2}(f(k-1)+f(k)) \tag{3.8}$$

A CDFG of the trapezoidal integration (equation 3.8) is shown in figure 3.8. This CDFG is derived manually from equation 3.8, and shows a suitable representation of that equation. The numbers above the edges represent communication latency and execution index (cf. section 3.1.1), respectively. All operations have a communication latency of 1. The execution index of the operations R_3 and R_1, which represent a discrete time delay (i.e., Z^{-1}), is 1. This execution index is indicated in equation 3.8 by the terms $i(k-1)$, and $f(k-1)$.

This algorithm has 8 operations: 5 storage operations (F, R_1,R_2, R_3, I), 2 additions (A_1, A_2), and one multiplication (M_1). Using a cyclic scheduling algorithm (TORSCHE Toolbox in Matlab [Šůc06]) the resulting latency is lat_{alg} =5 steps, and the execution period $w = 2$ steps. The scheduling is presented in figure 3.9, where only one execution period is shown.

In the scheduling shown in figure 3.9, all operations require the same execution time (1 Step). When using the normalised critical path delay with 32 bits to set the processing time of all operations, the value of the minimal period is $W = 4.17$, and the latency is $lat_{alg} = 6.39$. In this case differences in the execution time of all operations-types can be noticed, cf. figure 3.10. Furthermore, the weight of each

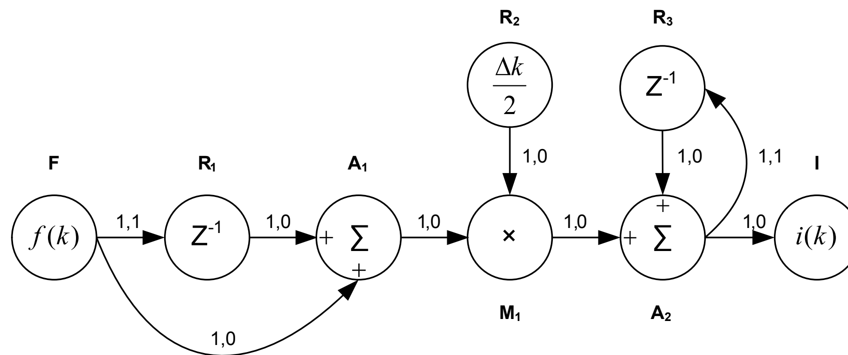


Figure 3.8: Proposed Cyclic Data Flow Graph of a trapezoidal integration algorithm (cf. equation 3.8)

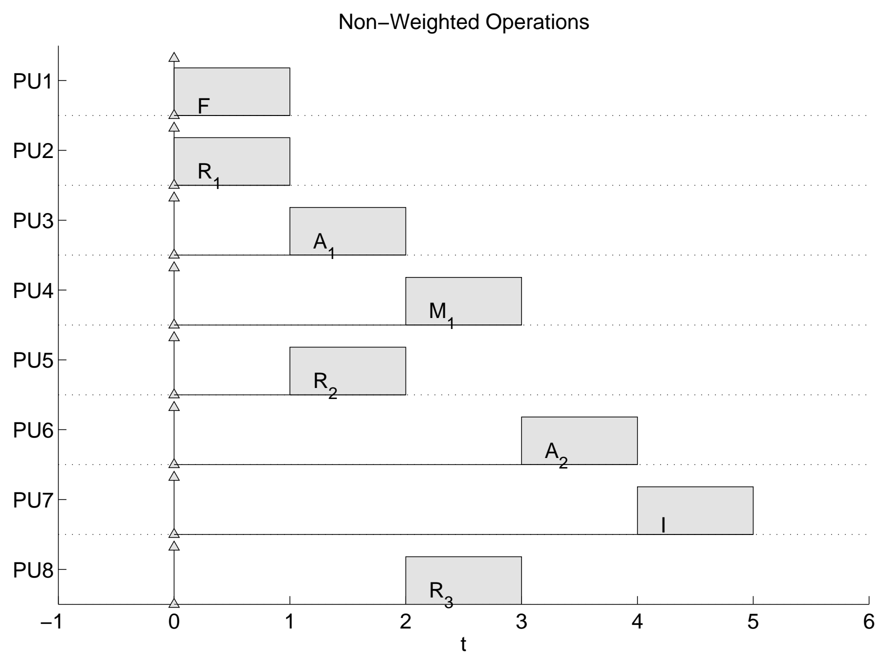


Figure 3.9: Cyclic scheduling for the trapezoidal integration depicted in figure 3.8 without weighting operations

operation according to their realisation effort (*NormOp*, cf. equation 3.2) is also reflected in the total amount of normalised operations (*NormOpAlg* cf. table 3.3).

The scheduling shown in figure 3.10 differs from the one shown in figure 3.9 regarding the execution order of some operations. For instance the storage operation R_3 in figure 3.9 starts at the same time that M_1, and in figure 3.10 the same operation is executed at the very beginning of the schedule. However, in both schedules the data

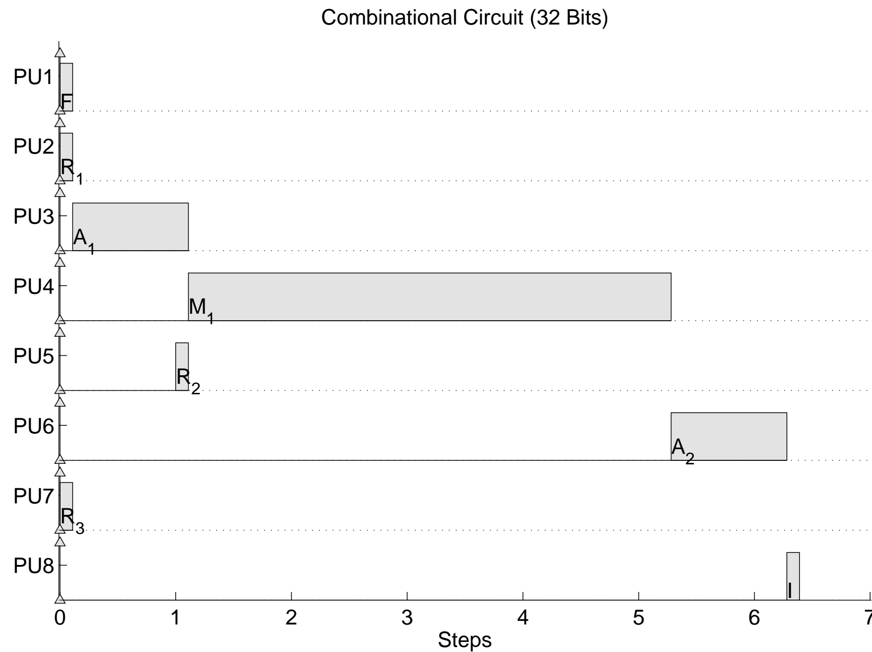


Figure 3.10: Scheduling for the trapezoidal integration depicted in figure 3.8 using normalised synthesis results as time weight

precedence of A_2 are fulfilled (e.g., R_3 has to be executed before A_2), and therefore both schedules are correct.

Using the values presented in figures 3.6 and 3.7, various implementations can be explored using different bit-widths, as presented in table 3.3. As expected, the total number of normalised operations ($NormOpAlg_i$), as well as the total number of normalised steps ($StepsAlg_i$) growth as the number of input bits (i) increases. These two measures are used to approximate the average parallelism of an algorithm, as explained in the next section.

3.1.5 Average Parallelism

Parallelism relates to the amount of operations, which have no data dependency at certain point of time of the execution flow and therefore can be realised concurrently. This measure is of relevance, because it gives us a hint on the amount of architectural parallelism that can be used when implementing an algorithm using a particular device. If the operations conforming the algorithm can be realised concurrently, using many processing units can yield a speedup in contrast to a single-processor approach. There are limits to the amount of speedup gained from a parallel implementation (cf. Amdahl's law [Amd67] and Gustafson's law [Gus88]). However, to know how much

Bit-Width (i)	$SizeAlg_i[\mu m^2]$	$NormOpAlg_i$	$StepsAlg_i$
8	2711.80	20.86	3.48
16	8448.44	25.15	4.37
24	17141.28	33.36	5.35
32	28281.24	40.05	6.39
40	43080.96	47.72	7.31
48	61544.60	54.32	8.31
56	81758.04	61.20	9.26
64	107280.68	67.71	10.18

Table 3.3: Algorithm characterisation of a numeric trapezoidal integration

architectural parallelism can be reasonably used to speedup an implementation, we need a way to estimate parallelism in a very early stage of the design flow.

Most of the existing methods to measure parallelism are proposed in the context of multi-processor systems, where tasks of a program have to be scheduled for execution [Sev89] or to compile C programs into hardware [Ven04]. Furthermore, these methods are based on the assumption that all processing elements are identical [Eag89, Ove00], which in the case of a hardware implementation is not necessarily true (i.e., a multiplier differs from an adder in terms of the computational resources required for its implementation).

For the evaluation presented in this chapter, algorithms will be characterise by estimating their average parallelism, which represents an upper bound to the asymptotic speedup (i.e., the speedup of the application with infinite resources and no synchronization costs) [Eag89, Ove00]. Average parallelism is estimated in this work by the amount of normalised operations done per normalised execution step, (*AOS*), cf. equation 3.9.

$$AOS_i = \frac{NormOpAlg_i}{StepsAlg_i} \tag{3.9}$$

Figure 3.11 shows two LH graphs, where all nodes represent the same operation, a carry look-ahead adder, having the same bit-widths. Furthermore, all nodes are associated with a processing element. For the graph shown in figure 3.11(a), 5 steps are required to execute the graph, i.e., $StepsAlg_i$=5, and the total number of normalised operations is $NormOpAlg_i$=6, therefore $AOS_i = 6/5$. For the graph shown in figure 3.11(b), 3 steps are required to execute the graph, i.e., $StepsAlg_i$=3, and the total number of normalised operations is $NormOpAlg_i$=6, therefore $AOS_i = 6/3$. As can be seen, the graph in figure 3.11(b) benefits more from using many processing elements than the graph in figure 3.11(a).

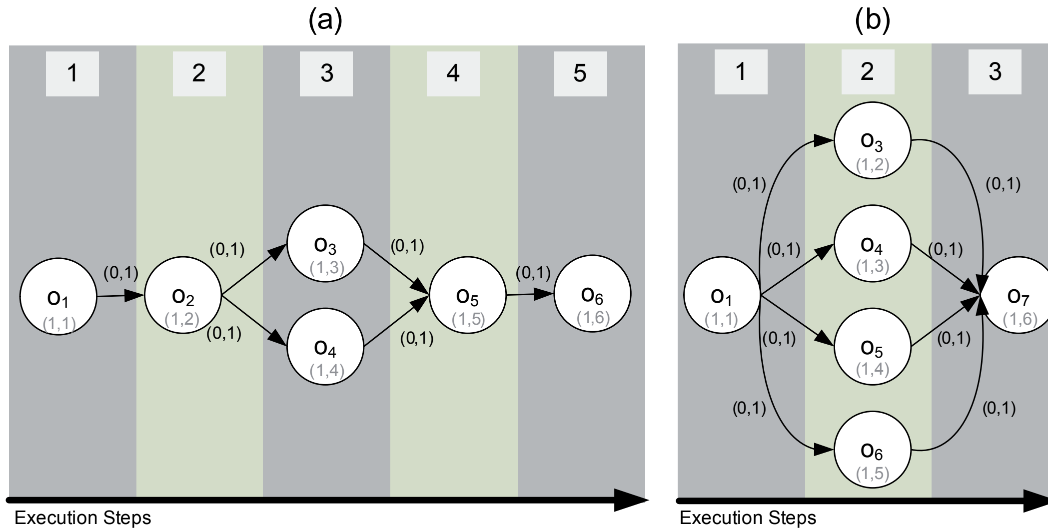


Figure 3.11: Two simple LH Graphs to exemplify the concept of average operations per step (*AOS*). (a) has an *AOS* of 6/5, whereas (b) has an *AOS* of 6/3

Notice that for previous examples, the number of utilised bits does not affect the value of AOS_i, because the same operation, a carry look-ahead adder, is used in every node, which results in every node having the same value regarding the number of normalised operations and steps, i.e., $NormO_i$=1 and $Steps_i$=1, cf. equations 3.2 and 3.3.

To exemplify the effect of having different operation and using different bit-widths, consider the trapezoidal numerical integration shown in figure 3.8. Table 3.4 shows the amount of operations and steps for different input bits, and the corresponding values of AOS_i. With an increasing bit-width increases also the number of normalised operations, and the number of normalised steps, which is reflected in the value of AOS_i.

The content of table 3.4 is also presented in figure 3.12. $NormOpAlg_i$ and $StepsAlg_i$ increase as the bit-width of each operation increases, which results in very small increment of AOS_i. This small increment was expected, because the trapezoidal integration consist mainly on registers and additions.

3.2 Resource Utilisation Assessment

Given the realisation of any algorithm on a specific computing architecture, the question arises as to how efficiently the resources of this computing architecture are being used to realise that algorithm? In this work two kinds of computing architectures

Bit-Width (i)	AOS_i
8	5.99
16	5.76
24	6.23
32	6.26
40	6.53
48	6.53
56	6.61
64	6.65

Table 3.4: Average operations per step (AOS_i) of a numeric trapezoidal integration for various bit-widths

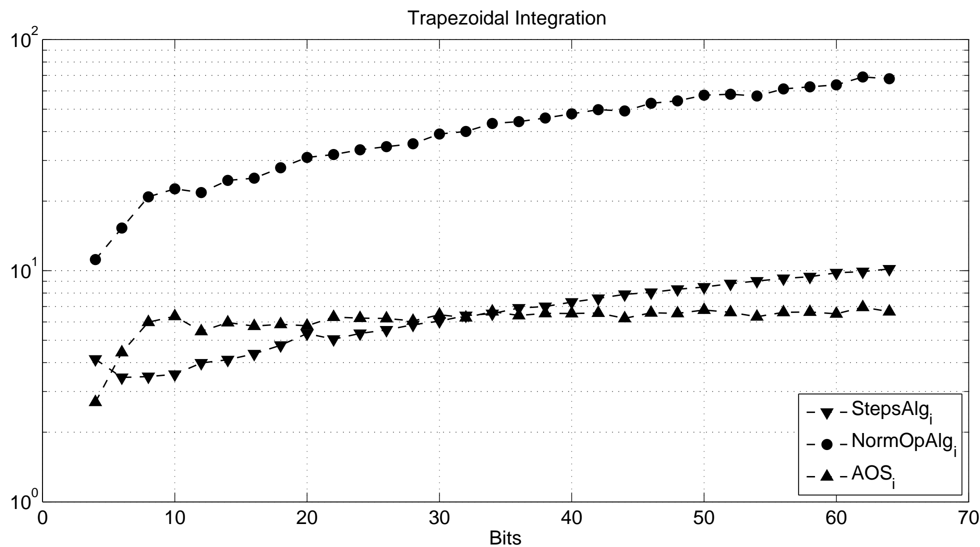


Figure 3.12: Algorithmic characterisation of a trapezoidal numeric integration (see figure 3.8) using normalised synthesis results

are considered: general purpose processors (GPP) and field programmable gate arrays (FPGA).

In contrast to general purpose computing, embedded control systems have well defined requirements and limitations [Hen07]. A control system must work in real-time, which for digital controllers implies a fixed throughput. Less throughput leads to a malfunctioning of the control loop, having more throughput does not necessarily improve the performance of the system. Furthermore, embedded control systems have typically limited spatial resources (e.g., because of implementation costs), and energy limitations (e.g., because the system is battery-driven, or because of heat dissipation).

In this section metrics are defined to measure how efficiently silicon area is being used in terms of computational density (a metric introduced by A. DeHon in [DeH95]), and to assess the energy consumption of those architectures. These metrics are introduced in the next sections.

3.2.1 Computational Density

Computational density is a measure of how much computation is done per area unit for a given computing architecture. This metric was proposed by A. Dehon in [DeH95] (cf. equation 3.10), and has been used by many other researchers in literature (e.g., [Jon10, Gal10, Pak10, Do10]).

$$C_{density,DeHon} = \frac{Throughput}{Area} \tag{3.10}$$

Throughput is generally defined as:

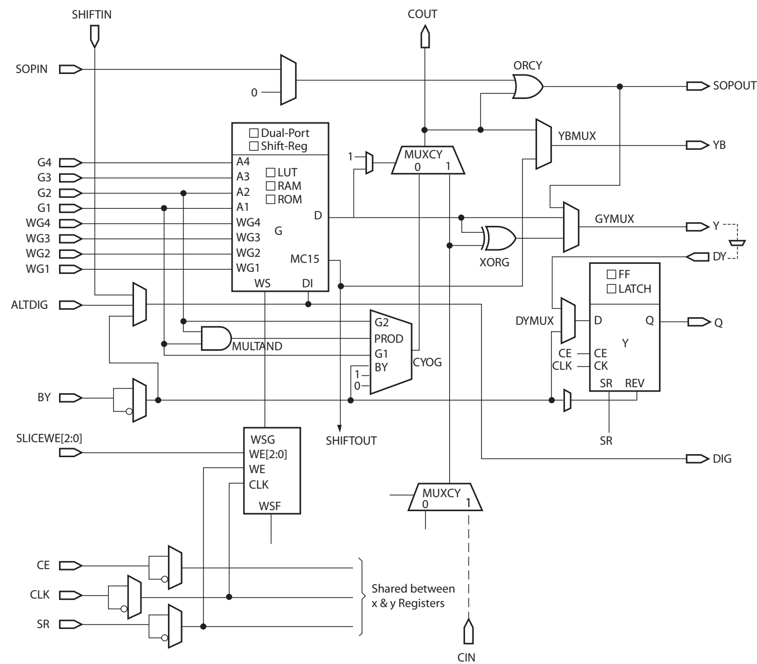
$$Throughput_{general} = \frac{Set\ of\ Operations}{Execution\ Time} \tag{3.11}$$

Throughput is typically measured in millions of operations per second (MOPS). For software-based architectures, $Set\ of\ Operations$ can be based on the instructions that can be executed by an ALU in one clock cycle. Furthermore, in [DeH95] it was proposed to use one-bit operations as base to compare hardware and software architectures. Thus, operations were defined as the required number of ALU operations multiplied by the bit-width of the processing element (cf. equation 3.12).

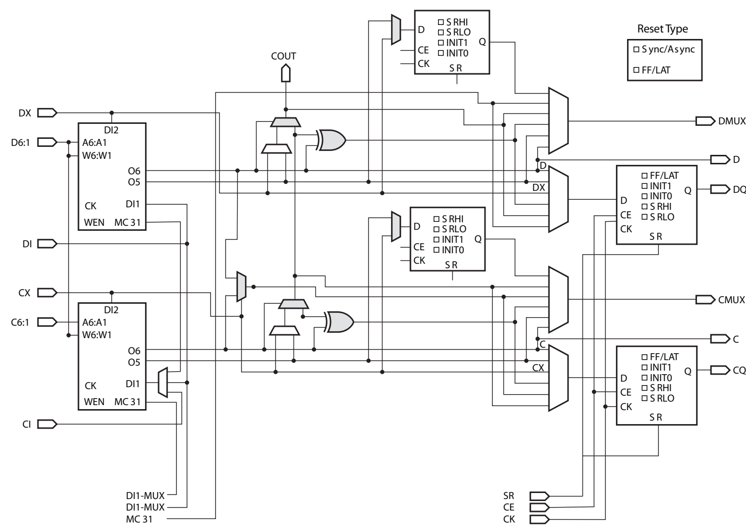
$$Set\ of\ Operations = ALU\ Operations \cdot WL \tag{3.12}$$

where WL is the word length of the ALU. The equivalent of a basic operation in terms of reconfigurable resources is not easily defined. It has been proposed in literature that one basic operation is equivalent to 4 LUTs [DeH95]. However, as presented in section 3.3.2, although processing elements of FPGAs are mostly LUT based, modern FPGA are not homogenous, i.e. they include a variety of embedded specialised processing units such as multipliers and memory, and their basic blocks are constituted by more than LUTs. Furthermore, the size of the LUTs varies depending on the device; e.g., Virtex-6 devices have LUTs with 6 inputs and one output, whereas slices of the Virtex II devices have 4 inputs and one output, cf. figure 3.13(a).

How this resources are used depends on the kind of operations to be implemented, and how they are mapped to the target architecture. In table 3.5, an example of the resource utilisation of some basic operations using four and eight bits is presented. All designs were realised using synthesis tools from Xilinx.



(a)



(b)

Figure 3.13: (a) Virtex-II Slice (Top Half) [Xil07c], and (b)Virtex-6 SliceM (Top Half) [Xil09c]

The definition of a basic computation unit for reconfigurable hardware architectures has to be based on the amount of logic resources used for an operation. However, as can be seen in table 3.5, resource utilisation is different depending on the operation.

Operation	Description	LUTs	MUX	ORCY	FF
Multiplier	4 bits, one constant	8	7	9	7
Multiplier	4 bits, two inputs	23	23	23	8
Full Adder	4 bits	5	4	4	6
Half Adder	4 bits	6	1	0	4
Accumulator	4 bits	6	1	0	8
Logic Function	4 bits XOR	4	0	0	4
Multiplier	8 bits, one constant	22	23	25	15
Multiplier	8 bits, two inputs	83	83	82	16
Full Adder	8 bits	9	8	8	10
Half Adder	8 bits	8	7	7	8
Accumulator	8 bits	8	7	7	8
Logic Function	8 bits XOR	8	0	0	8

Table 3.5: Resource utilisation of various basic operations, synthesised for a Virtex II device (XC2V4000), using synthesis tools from Xilinx

Therefore, using only LUTs and FlipFlops (FF) as indicator would be oversimplifying. In table 3.5 other resources available in each slice are listed: multiplexers (MUX), and OR gates for carry logic (ORCY).

The deterministic nature of digital controllers and the associated hard real-time constraints make it possible to define a block of operations, which are executed repetitively. Therefore, the definition of throughput can be based only on the execution time of a specific implementation of an algorithm:

$$Throughput = \frac{1}{T_{con}} \tag{3.13}$$

Where T_{con} is the execution time of a controller implementation. Therefore, equation 3.13 measures throughput as the amount of generated outputs per time period. Because the chosen architectures are implemented using different technologies normalisation of area, delay, and energy is required. Area is normalised using scaling rules presented in [Bol09, Cha10, Fra01], cf. equation 3.14.

$$S_{area} = \left(\frac{\lambda_{ref}}{\lambda}\right)^2 \tag{3.14}$$

Were λ is half the minimum drawn feature size of a process, and λ_{ref} is that value of a selected architecture, in this case λ from the reconfigurable hardware device will be used. Thus equation 3.10 becomes:

$$A_{norm} = Area \cdot S_{area} \tag{3.15}$$

Delay is normalised by the following factor [Bol09, Cha10, Fra01]:

$$S_{delay} = \frac{\lambda_{ref}}{\lambda} \tag{3.16}$$

This normalisation does not hold as λ goes into sub-micrometer dimensions (i.e., $\lambda \ll 100nm$) [Bol09]. However, for the devices used in this comparison (cf. sections 3.3.1 and 3.3.2), these normalisation gives a valid estimate. Throughput is then normalised as follows:

$$Throughput_{norm} = \frac{1}{T_{con}} \cdot \left(\frac{1}{S_{delay}}\right) \tag{3.17}$$

Finally the equation for computational density is:

$$C_{density} = \frac{Throughput_{norm}}{A_{norm}} \tag{3.18}$$

High values of $C_{density}$ indicate more throughput per silicon area. This metric penalises the silicon area of the device, using it as a cost indicator. Other cost factors such as design costs (e.g., cost of the design tools), NRE, or packaging also influence the production cost. However, using these factor reflects more the current economic situation rather than the actual required resources.

3.2.2 Energy Efficiency

A metric to measure energy efficiency depends on the type of computation to be performed. In [Bur96, Lan05] three different kinds of applications are defined: fixed throughput, maximal throughput, and burst throughput mode. Digital controllers have typically fixed sampling rates, therefore, energy efficiency can be measured by the amount of power required for a fixed throughput mode, as presented in equation 3.19 [Cla99, Fan07, Gal10].

$$E_{efficiency} = \frac{Throughput}{Power} \tag{3.19}$$

Power can be normalised as follows [Bol09, Cha10, Fra01]:

$$Power_{norm} = Power \cdot S_{power} \tag{3.20}$$

where the normalising factor S_{power} is defined as:

$$S_{power} = \left(\frac{V_{dd,ref}}{V_{dd}}\right)^2 \cdot \left(\frac{\lambda_{ref}}{\lambda}\right)^2 \tag{3.21}$$

Where V_{dd} is the voltage input of the architectures core, and $V_{dd,ref}$ is that of the reference architecture.

As technologies shrink and supply voltage decreases (e.g., $V_{dd} < 1V$), this normalisation step is no longer applicable [Cha10]. However, for the selected devices this simple normalisation suffices.

$$E_{efficiency} = \frac{Throughput_{norm}}{Power_{norm}} = \frac{Throughput}{Power} \cdot \left(\frac{1}{S_{power} \cdot S_{delay}}\right) \tag{3.22}$$

The higher $E_{efficiency}$ is, the more computations per power unit are realised. Therefore, the devices with higher values of $E_{efficiency}$ is better suited for the investigated algorithm, with respect to the energy consumption.

3.3 Computing Architectures

Two computing architectures were used for this comparison. A PowerPC 750 and a Spartan 3A FPGA from Xilinx. These devices are introduced in the following section.

3.3.1 PowerPC 750-G Processor

The PowerPC 750 is a reduced instruction set computer (RISC) microprocessors implemented in a 20 μm CMOS technology, it has a silicon area of 40 mm^2. The PowerPC 750 implements the 32-bit portion of the PowerPC architecture, which provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits. Figure 3.14 shows the parallel organization of the execution units (marked in gray). The instruction unit fetches, dispatches, and predicts branch instructions.

The 750 is a superscalar processor that can complete two instructions simultaneously. It incorporates the following six execution units [IBM02]:

- Floating-point unit (FPU)
- Branch Processing Unit (BPU)
- System Register Unit (SRU)
- Load/Store Unit (LSU)
- Two Integer Units (IUs): IU1 executes all integer instructions (multiply, divide, shift, rotate, arithmetic, logical). IU2 executes all integer instructions except multiply and divide instructions.

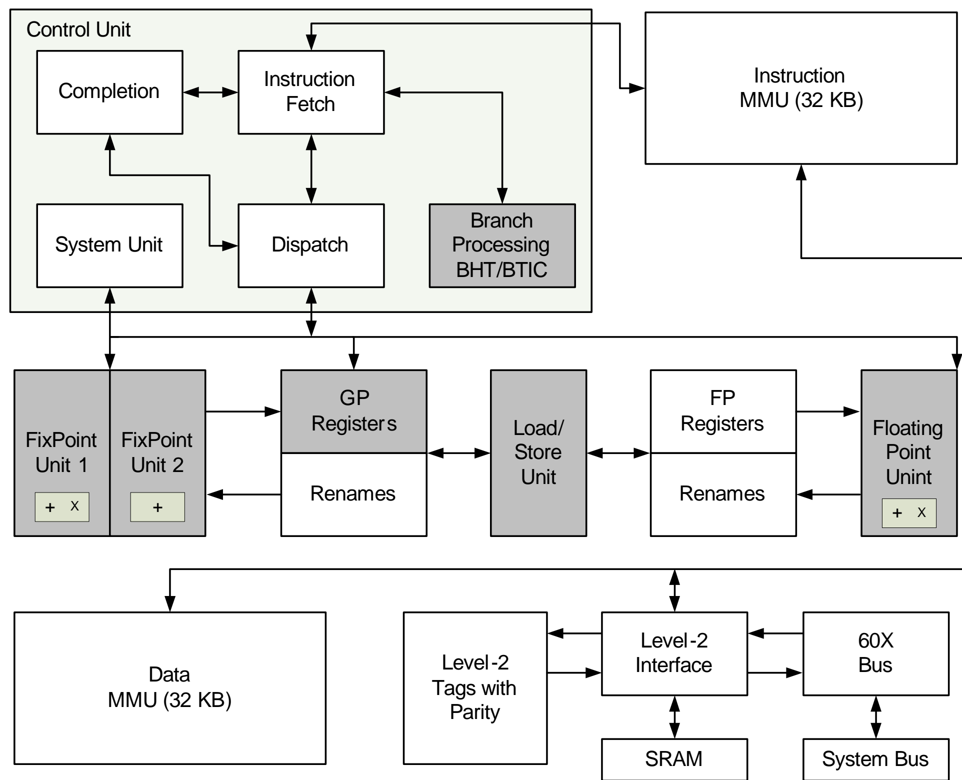


Figure 3.14: Simplified block diagram of the PowerPC 750 RISC Microprocessor

Most integer instructions execute in one clock cycle. However, multiplication and Division are multi-cycle instructions. The FPU is pipelined, three single-precision floating-point instructions can be in the FPU execute stage at a time. Double-precision add instructions have a three-cycle latency; double-precision multiply and multiply-add instructions have a four-cycle latency.

The 750 has four software-controllable power-saving modes. Three static modes, doze, nap, and sleep, progressively reduce power dissipation. When functional units are idle, a dynamic power management mode causes those units to enter a low-power mode automatically without affecting operational performance, software execution, or external hardware. The 750 also provides a thermal assist unit (TAU) and a way to reduce the instruction fetch rate for limiting power dissipation.

The PowerPC is evaluated using a dSPACE card DS1005, depicted in figure 3.15. The DS1005 provides 1 MByte, level 2 external cache memory, 16 MByte Flash memory (1 MB reserved for booting), 128 MByte SDRAM global memory (64 MB for the PowerPC separately arbitrated). A clock frequency of 480 MHz is provided to the PowerPC [dSP05].

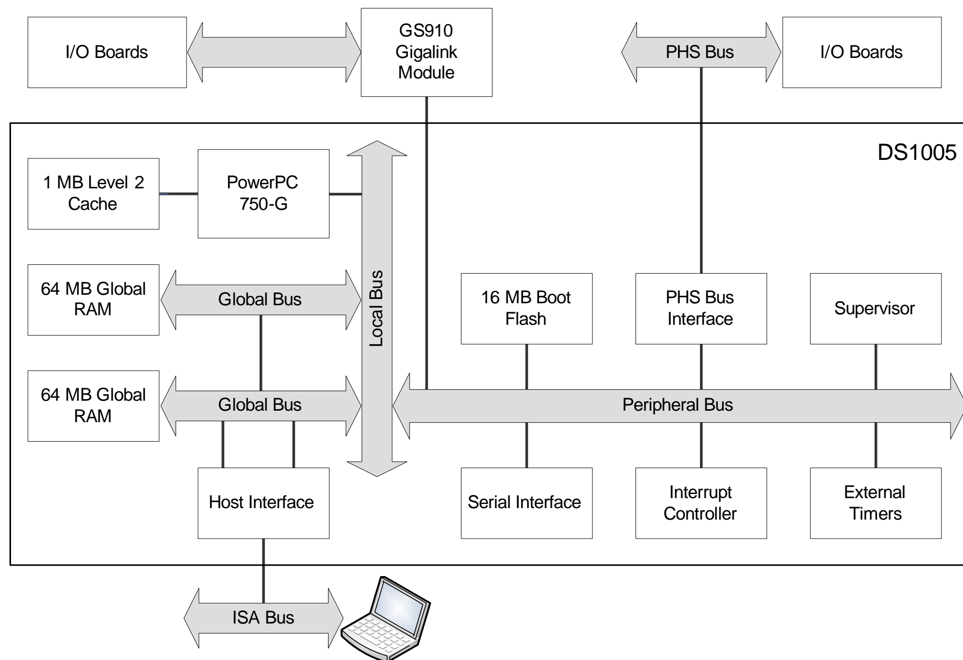


Figure 3.15: Simplified block diagram of the dSPACE DS1005 board

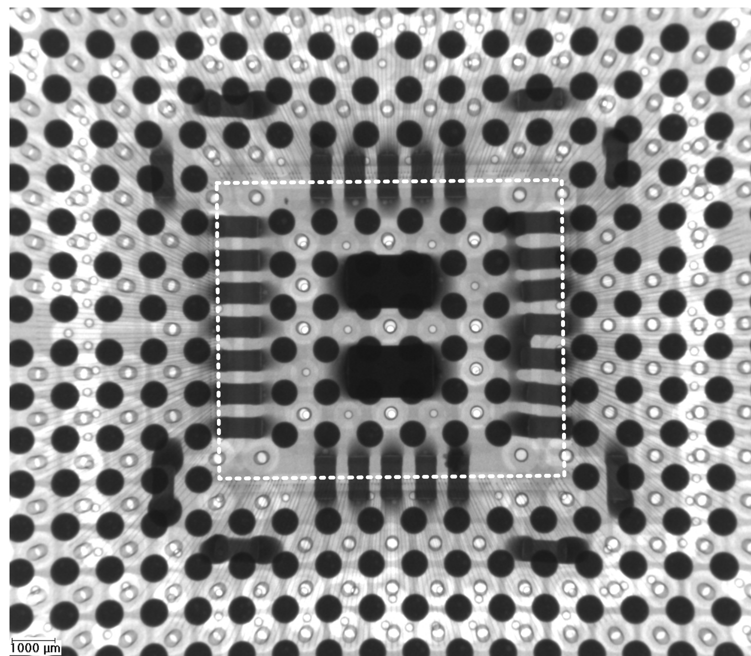
The DS1005 allows communication with a host-PC via the ISA port, or via the PCI through an adapter. The board provides also the PowerPC with three timers, and interrupt handling, among others. Data access and parametrisation of the board is done through software from dSPACE (i.e., ControlDesk). Furthermore, the implementation of controllers using Matlab/Simulink is also supported (cf. [dSP11]).

3.3.2 FPGA Device

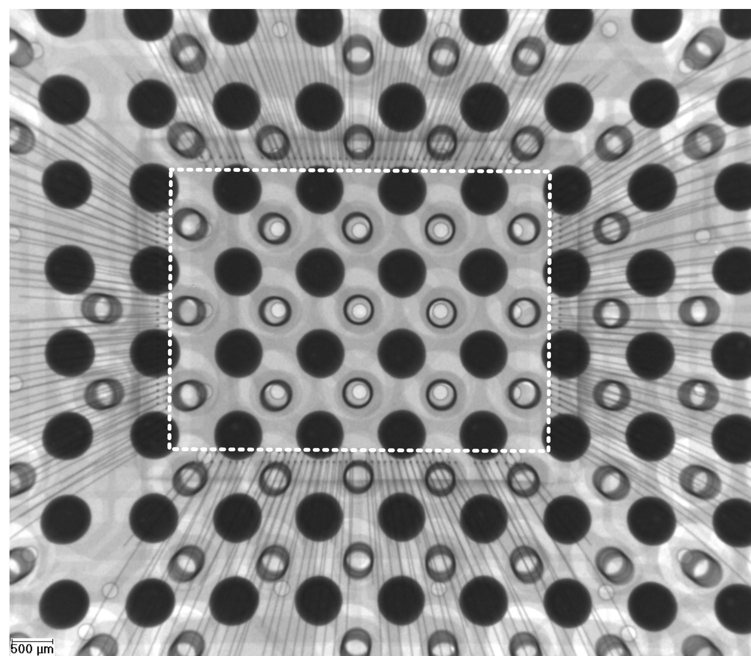
The variety of FPGAs devices in the now-a-days market is very wide. In order to chose an FPGA for the comparison, the chip area has to be taken into account. However, the chip area of Xilinx devices has not been published (except for a few devices, e.g., [Yui02]). Therefore, the area has to be measured. To achieve this, a radiography of several available FPGAs from Xilinx was first obtained, as shown in figure 3.16.

From these images the chip area was measured, as presented in figure 3.16(a). A suitable FPGA from the available set is the Spartan-3 XC3S1500 (package FG456), the area of this device is about $\sim 67mm^2$.

The Spartan-3 FPGA from Xilinx was introduced in April of 2003 in a 90 nm technology, it has 3328 CLBs, 576 Kbits embedded Block-RAM, 32 18×18 embedded multipliers, 4 DCMs, and 333 I/Os.



(a)



(b)

Figure 3.16: Radiagraphy of (a) an Spartan-3 XC3S1500, and (b) a Spartan-3 XC3S200 devices from Xilinx

3.4 Realisation Flow

Each of the benchmarks presented in the next section was realised in hardware and software using the procedure described in the following sections.

3.4.1 Hardware Implementation-Flow

The tools used for hardware implementation of benchmarks are listed in table 3.6.

To implement a benchmark in hardware the following steps were done:

- Hardware Description: A hardware description of the controllers was derived from their CDFG. The Synplify DSP blockset [Synb] was used to generate the hardware description under Simulink. As in the case of the software realisations, the parameters of the design were set to constant values that allow functional testing, without a specific control goal.
- Setting Bit-Width: The bit-width was constrained for each operation of the controller. To make a fair comparison with the processor, 8, 16, 32, and 64-bit

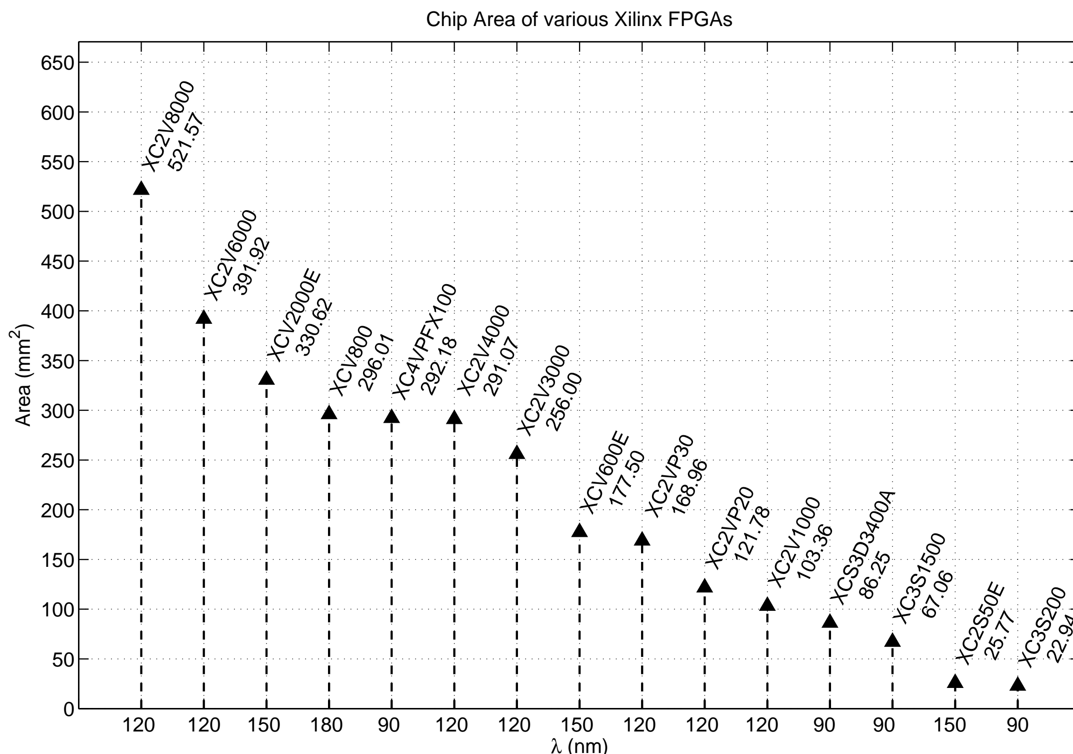


Figure 3.17: Area measurement of various Xilinx FPGA

Software Tool	Version	Company
Matlab	7.3.0.267	Matworks
Simulink	6.5	Matworks
Synplify DSP	3.6	Synopsys
Synplify Premier	3.6	Synopsys
ISE	10.1.03	Xilinx
XPower	10.1.03	Xilinx

Table 3.6: Tools used for the software implementation of the benchmarks

versions of each controller were realised. Controller parameters were set by using equation 3.23.

- Optimization: Synplify DSP allows three optimisation options: retiming, folding, and multichannelisation. Retiming allows automatic rearrangement of registers or placement of new register to reduce the critical path of a design, while sacrificing throughput; for each extra latency inserted the clock speed has to be correspondingly increased to get the desired throughput. Folding is an area optimisation that folds the algorithm by reusing resources over multiple physical clock cycles (e.g., multipliers). This allows the reduction of required resources by increasing the required clock frequency, too. Multichannelisation is an optimisation that minimise hardware by sharing the hardware over multiple channels. Folding and retiming were only used for selected cases (it is indicated), where the resources requirement were too high for the selected FPGA. Multichannelisation was not used for the experiments presented in the next section.
- Synthesis: RTL-level VHDL code was generated using the Synplify DSP toolflow. This code was then synthesised using Synplify Premier with Design Planner [Synb], which generated a netlist and the corresponding user constraint file. In this step IO pads where assigned to the ports of the design. After netlist generation, the design was placed and routed using synthesis tools from Xilinx (part of the ISE foundation ver. 10.1), where optimization goals where set to increase speed.
- Measurements: For each benchmark, a post-place and route advanced timing report was generated, from where the maximum allowable clock-rate was obtained. To estimate power, the XPower Analyzer from xilinx was used, after the design was placed and routed, setting up the estimation parameters to values, which according to [Xil11], represent a worst-case scenario: flipflop toggle rate = 20%, I/Os toggle rate= 20%, I/Os enable rate=100%, Output load=5 pf, BRAM write rate=50%, BRAM enable rate=100%.

3.4.2 Software Implementation-Flow

For the software implementation, a Matlab-Simulink based flow with tools from the company dSPACE was used, cf. table 3.7.

Software Tool	Version	Company
Matlab	7.3.0.267	Matworks
Simulink	6.5	Matworks
RealTime Workshop	6.5	Matworks
ControlDesk	3.3	dSPACE
dSPACE Profiler	1.1.1	dSPACE

Table 3.7: Tools used for the software implementation of the benchmarks

- Generate Simulink Model: The first step was to create a simulink model of the discretised controller equations. The simulink model was based on the CDFG of each benchmark. A first version of each controller was done using floating point format, with the purpose of testing the function of the controller. Because no particular design goal was set, the parameters of the simulink model were first set to a constant that allows testing its function.
- Definition of Data Types: A data type was assigned to each block of the model: int8, int16, int32, or float64. Float64 was used to force the compiler to use the floating point unit of the PPC750. However, for calculations only integers where used. Furthermore, results using 64 bits are only used to discuss software-based realisations, because hardware-based designs used only fix-point arithmetic.
- Definition of Constants: Each parameter was assigned to a random number high enough to avoid an early operation exit when computing a multiplication. The parameters where set using equation 3.23.

$$Data = 2^{Bits-1} + rand(2^{Bits-1}) \tag{3.23}$$

  where *Bits* is the number of bits corresponding to the selected data type. This step was required to avoid that the compiler optimised data type operations. Furthermore, the integer units of the PPC support early operation scape (cf. section 3.3.1) for multi-cycle operations such as multiplication and division. Therefore, all constant values were initialised with a value, which ensures no early-scape optimisations. The PowerPC can process 8, 16, 32, and 64 Bits operations (cf. section 3.3.1). Therefore, all benchmarks were implemented using the corresponding data types.
- Definition of compiler optimization parameters: The dSPACE software uses the Microtec PowerPC C Compiler Version 2.0m for the PowerPC 750. Full inline

expansion was set, which saves the overhead usually associated with function calls, parameter passing, register saving, stack adjustment, and value return. The highest recommended level of optimization was used (i.e., -05), which provides cross-module inter-procedural optimisations. This mode supports whole-program analysis.

- Generation of C code: C code was generated using Real-Time Workshop from Simulink, selecting the dSPACE card as the target platform, and the compiler-optimisation options previously described.
- Insertion of profiling functions: Because the use of a profiler is not automatically supported by the tool-flow, code has to be inserted manually to access this function. Marks are inserted to signalise the beginning and ending of functions corresponding to the controller, thus neglecting the time-overhead of functions such as communication with the host computer.
- Configuration of the dSPACE card: For this purpose software from dSPACE, called ControlDesk was used. This software enables the visualisation of all variables defined in a simulink model, and if required the parametrisation of a model in run-time.
- Measurements: Execution time was measured by retrieving profiling information from the dSPACE card. Absolute time and the time difference between the inserted marks can be retrieved, allowing the exact calculation of the execution time corresponding to a benchmark. Several hundred measurements per benchmark were performed, from which the mean values were calculated.

  To calculate energy consumption, typical power consumption was used. Depending on the operating mode, the power consumption of the PPC750 varies, as shown in the graph presented in figure 3.18. The typical power consumption of the Full-On mode was used for estimations, except for operations using the 64-bits floating point unit, which according to [Nam01] uses almost two times more energy than the integer units to perform the same operation.

In the next section the algorithmic characterisation, and hardware and software realisation of a selected set of benchmarks is presented, and results are discussed.

3.5 Benchmarks

For the following benchmarks, there is neither a specific control goal, nor a particular plant to be controlled. The goal of the experiments is to achieve the maximum throughput possible, assess the resource utilisation, and relate the algorithmic characteristics to the results. Three structures were chosen: a PID controller, a state-feedback controller, and a state observer. These control algorithm are widely used [As08, Oga87, Dor11], and are therefore representative of the application area.

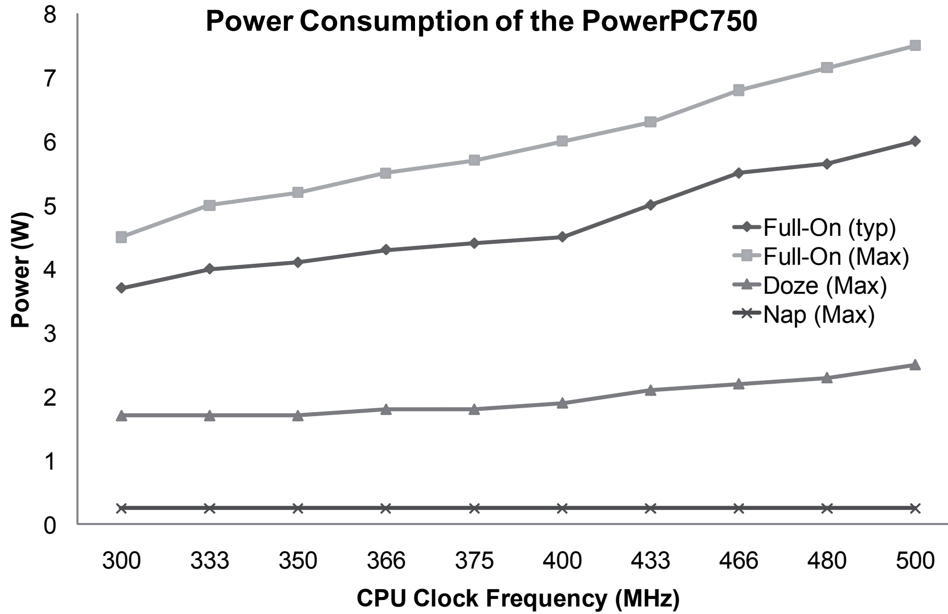


Figure 3.18: Power consumption of the PowerPC 750 CPU for various operation modes [IBM02] (power for 480 MHz was calculated using polynomial interpolation)

3.5.1 PID Controller

PID control is still one of the most widely used algorithms in industry, it is estimated that more than 90 % of the low-level loops of process control are still PID-based [As01]. The standard PID algorithm is described by:

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right) \tag{3.24}$$

where $u(t)$ is the control signal, $e(t)$ is the error signal, defined as $e(t) = y_{sp}(t) - y(t)$, and the control parameters are K_p (proportional gain), T_i (integral time), and T_d (derivative time). A Laplace transform of equation 3.24 yields:

$$U(s) = K_p \left(Y_{sp}(s) - Y(s) + \frac{1}{sT_i} \left(Y_{sp}(s) - Y(s) \right) + sT_d \left(Y_{sp}(s) - Y(s) \right) \right) \tag{3.25}$$

For a real-world implementation of equation 3.25, some modifications are required [Wit03]. The derivative is modified to avoid large amplifications of measurement noise, resulting in the following expression:

$$sT_d \approx \frac{sT_d}{1 + sT_d/N} \tag{3.26}$$

Thus, high-frequency noise is limited to N at high frequencies. Another modification to the algorithm is to avoid the action of the derivative term on the command signal (set-point). Furthermore, a parameter b, which regulates the weight of the command signal on the proportional term, is introduced [As01]. After this modifications, equation 3.25 becomes:

$$U(s) = K_p \left(bY_{sp}(s) - Y(s) + \frac{1}{T_i s}\left(Y_{sp}(s) - Y(s)\right) + \frac{sT_d}{1 + sT_d/N} Y(s) \right) \tag{3.27}$$

To avoid the negative effects of integral action in the presence of saturation (e.g., caused by actuator limitations), a mechanism has to be implemented, which detects such non-linear effects and accordingly avoids them. Such a mechanism is known as anti-windup [As01]. A simple implementation of the anti-windup strategy is:

$$e\prime(k) = \begin{cases} e_{min}, & \text{if } f(k) < Act_{min}; \\ e_{max}, & \text{if } f(k) > Act_{max}; \\ e(k), & \text{otherwise.} \end{cases} \tag{3.28}$$

Where $e\prime(k)$ is the output of the anti-windup operator, $f(k)$ is a feedback signal from the actuator (if not measurable, this signal can be generated by a mathematical model), Act_{min} is the minimum value that the actuator can reach, and Act_{max} is the maximum value, e_{min} and e_{max} are suitable values to avoid integrator windup. The components of the PID controllers are:

$$P(s) = K_p(bY_{sp}(s) - Y(s)) \tag{3.29}$$

$$I(s) = \frac{K_p}{T_i s}\left(Y_{sp}(s) - Y(s)\right) \tag{3.30}$$

$$D(s) = \frac{K_p sT_d}{1 + sT_d/N} Y(s) \tag{3.31}$$

The discretisation of these components is discussed in the following paragraphs.

Proportional Component The proportional action depends only on instantaneous values of the error signal, and is zero only if $e(t) = 0$. A discretisation of the proportional term is given by:

$$p(k) = K_p(by_{sp}(k) - y(k)) \tag{3.32}$$

where k represents the sampling instant.

Integral Component The output of the integral term is proportional to the accumulated error. The inclusion of the integral term allows a zero steady state error.

Numerical integration can be implemented in several ways. First order approximation (e.g., forward and backward Euler approximation), or second order approximation (e.g., Sympson's rule), are the most common methods, because of their low complexity and stability. Here Sympson's rule is used, which is given by equation 3.33.

$$i_s(t+1) = i_s(t) + \frac{\Delta t}{3}(f(t-1) + 4f(t) + f(t+1)) \tag{3.33}$$

where Δt is the difference between two sampling instants (i.e., the duration of the control cycle). When substituting 3.33 in 3.30 it yields:

$$i_s(k+1) = i_s(k) + \frac{K_p \Delta k}{3T_i}(e(k-1) + 4e(k) + e(k+1)) \tag{3.34}$$

where $e(k) = y_{sp}(k) - y(k)$.

Derivative Component The derivative term is proportional to the rate of change of the feedback signal. When the feedback signal stays constant, the derivative term contributes zero to the control action.

The most common method for numerical differentiation is backward differences, given by:

$$f\prime(t) = \frac{f(b) - f(a)}{b - a}$$

when including the modifications made in equation 3.26 the derivative term is then given by:

$$d(k) = \frac{T_d}{T_d + N\Delta k}\left(d(k-1) - K_p N\left(y(k) - y(k-1)\right)\right) \tag{3.35}$$

Thus, the PID controller is given by:

$$u_3(k) = p(k) + i_s(k) + d(k) \tag{3.36}$$

Algorithmic Characterisation

Using the metrics defined in section 3.1, we first look at the algorithmic characteristics of the controller. A suitable CDFG corresponding to equation 3.36 is presented in figure 3.19. This CDFG is manually derived from equation 3.36.

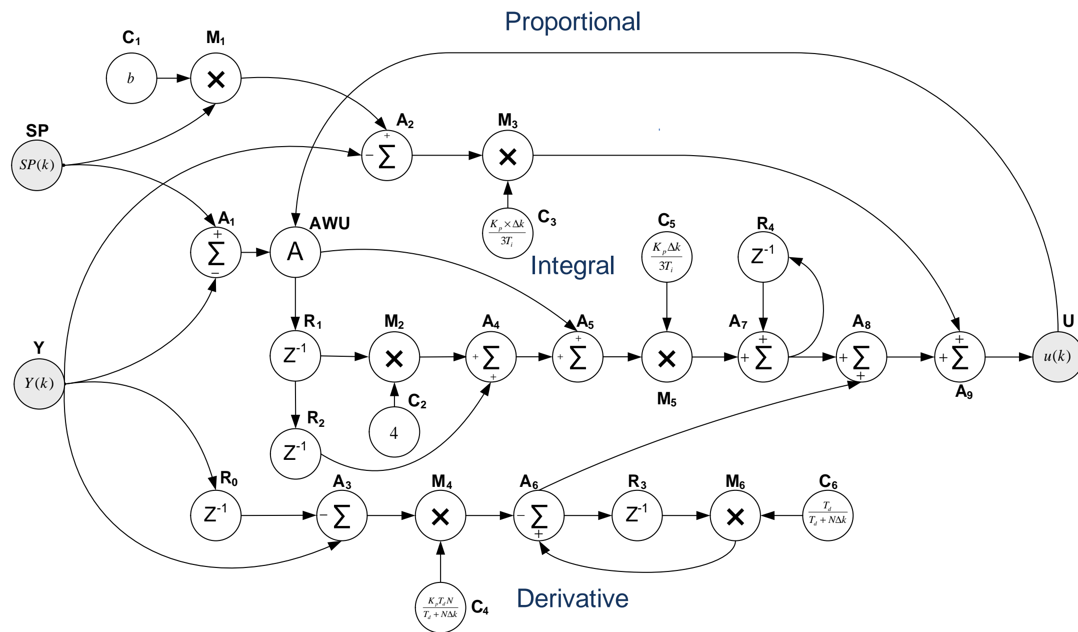


Figure 3.19: CDFG of a PID controller, using Simpson numerical integration. Node *A* represents an anti-windup algorithm, shown in figure 3.20

A corresponding CDFG for the anti-windup algorithm (cf. equation 3.28) is shown in figure 3.20. This CDFG corresponds to the block *A* in figure 3.19. Gray nodes represent inputs and outputs of the graph.

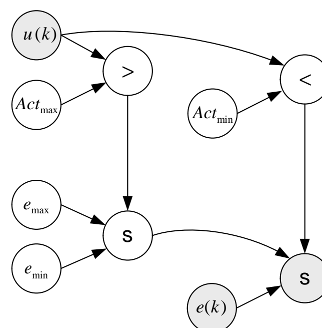


Figure 3.20: Implementation of a simple anti-windup strategy

The PID controller, including a simple anti-windup algorithm has a total of 37 operations: 6 multiplications, 9 additions, 18 memory operations, 2 select operators, 2 comparisons. Computing a schedule for the CDFG presented in figure 3.19 using TORSCHE (cf. [Šůc06]), the latency of the PID is lat_{alg}=10 steps, when nodes of the CDFG are not weighted using the normalised units defined in section 3.1.3), cf. figure 3.21. For the algorithmic characterisation of the PID controller, the only parameter that is changed is the bit-width of the operations. The next examples deal with the case where the number of operations is changed, cf. sections 3.5.2 and section 3.5.3.

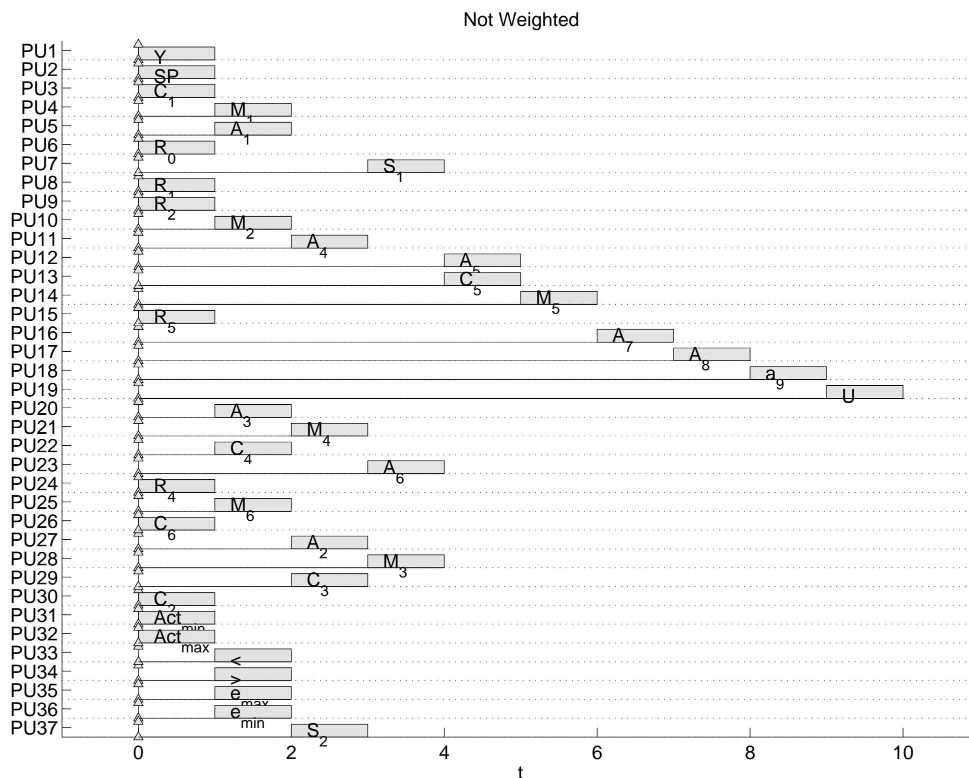


Figure 3.21: Scheduling for the PID controller depicted in figure 3.19 without weighting operations

A summary of the algorithmic characteristics of the PID controller using different bit-widths, and the normalised weights defined in section 3.1.3, is presented in table 3.8. It is shown that, as expected, the modelled circuit size ($SizeAlg_i$) and modelled circuit depth ($DepthAlg_i$) increase when increasing the base bit-width of each operation. Correspondingly, the number of normalised operations ($NormOpAlg_i$) and normalised steps ($StepsAlg_i$) increase.

Results presented in table 3.8 are shown in figures 3.22, using a logarithmic scale for the Y axis. A significant growth of the number of normalised operations ($NormOpAlg_i$), and the number of normalised steps ($StepsAlg_i$) can be observed.

Bit-Width	$SizeAlg_i(\mu m^2)$	$DepthAlg_i(ns)$	$NormOpAlg_i$	$StepsAlg_i$	AOS_i
8	15031.12	7.03	76.96	7.39	10.42
16	48048.00	9.32	142.41	7.36	19.34
24	98951.84	11.37	186.62	7.56	24.70
32	164442.72	13.43	217.58	7.75	28.08
40	251869.28	15.40	231.20	8.06	28.68
48	361180.04	17.23	265.84	8.15	32.61
56	481107.64	19.30	273.04	8.28	32.98
64	632708.44	21.14	307.55	8.45	36.38

Table 3.8: Algorithm characterisation of a parallel PID controller

However, the growth rate of average operations per step AOS_i is rather low. In other words, the algorithmic size of the PID controller has only a small influence on the number of average operations per step. This fact has a direct repercussion on the resource utilisation of hardware and software implementations, because it is expected that using a parallel architecture wont result in a significant speedup increment for an increasing problem-size.

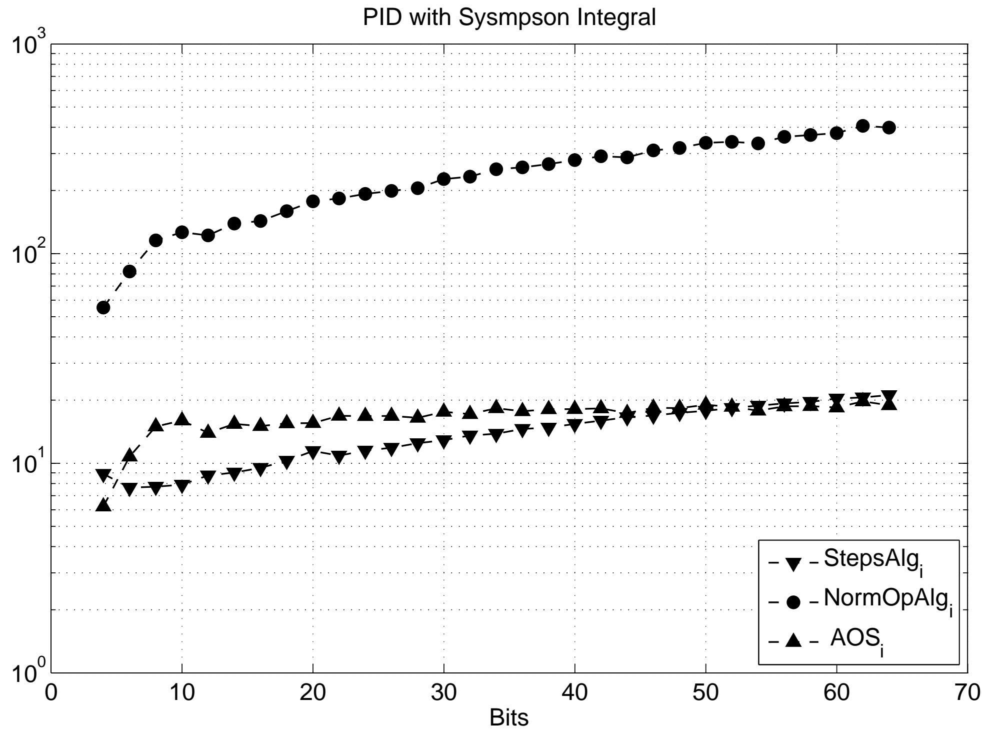


Figure 3.22: Algorithmic characterisation of a PID controller (see figure 3.19)

Reconfigurable Hardware Implementation

The PID controller shown in figure 3.19 was implemented using various bit-widths for each operation. A hardware description of the controller was done using the Synplify DSP toolbox in Matlab/Simulink (cf. figure 3.23) following the tool-flow described in section 3.4.1.

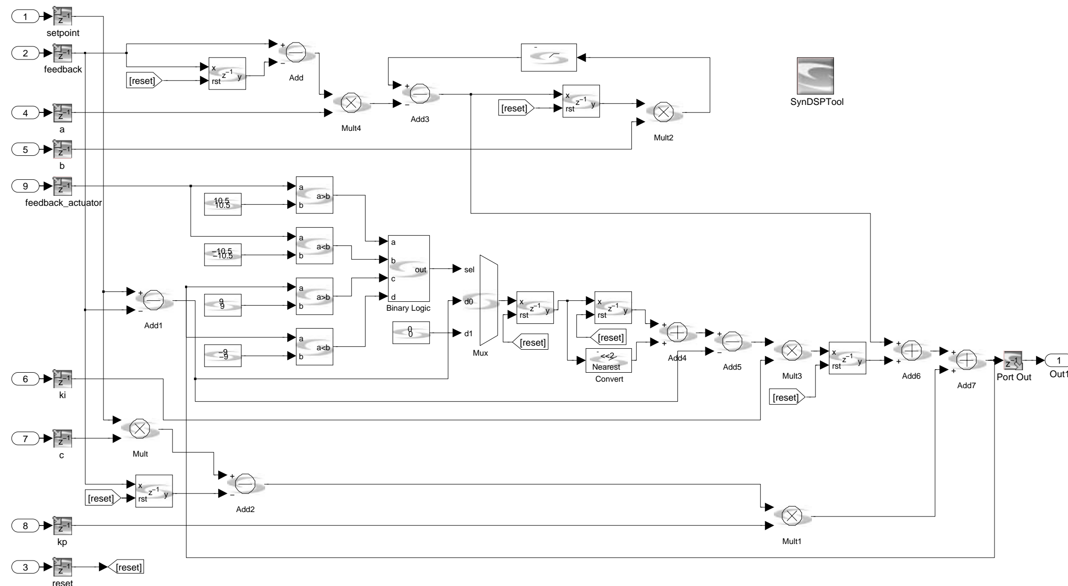


Figure 3.23: Implementation of a PID controller with Synplify-DSP

Implementation results are shown in table 3.9. As the bit-with of the operations grows, the number of LUTs required to realise one single operand increases (LUTs in the Spartan 3 FPGA have 4 inputs and one output), as predicted by the model (cf. figure 3.22). This causes a single operation to be implemented using several LUTs, thus increasing the routing delay, which explains the increment of execution time as the bit-width growths. The percentage change, using the execution time of the 8-bit version of the PID as reference, goes from 11% to 119% for the 16-bit, and the 64-bit versions, respectively.

Furthermore, although the achievable clock frequency decreases as the bit-width increases, power consumption of the different PID controllers raises with the amount of reconfigurable resources being used, as shown in table 3.9, which results from the toggle activity of the extra logic resources and I/Os pads used. The percentage change of power consumption, using the 8-bit realisation as reference, goes from 9% to 54% for the 16- and 64-bit versions, respectively.

Software Implementation

Using the flow described in section 3.4.2 a PID controller was implemented using different data types, corresponding to different bit-widths. The controller was implemented using standard simulink blocks, as shown in figure 3.24. For the 64 bit realisation, the floating point unit of the PPC is used. However, this values were not used to compare hardware and software realisations, because hardware-based designs used exclusively fixed-point arithmetic.

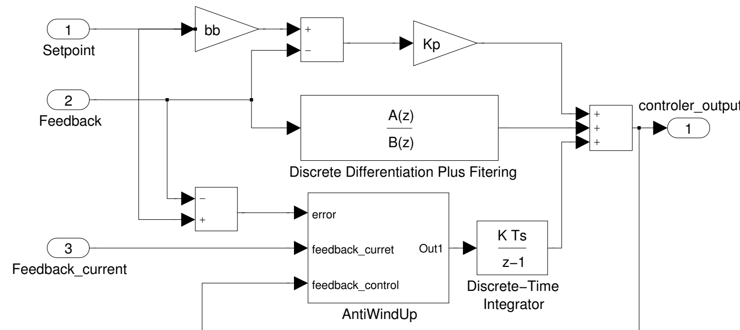


Figure 3.24: Implementation of a PID controller using Matlab/Simulink

Implementation results are shown in table 3.10. Reported execution times represent mean values over several hundred measurement (cf. section 3.4.2).

In table 3.10, it is shown that the execution time of the 8- and 16-bit version of the controllers does not differ much from each other (> 4%). In contrast, the 32-bit version of the PID controller has an execution time more than 42% longer than the 8-bit version. This increment results from the fact that the multiplier of the IU1 arithmetic unit of the PowerPC supports early exit for operations that do not require full 32- x 32-bit multiplication, which is the case of the first two versions of the PID controller. The 64-bit realisation uses the floating point unit, which achieves 4 cycle latency, 2 cycle throughput, for a double-precision multiply-add operation, in contrast to the multi-cycle operation mode of the integer unit IU1. The use of this specialised unit

Bit-Width	**Slices**		**Multipliers**		Time (μs)	Power (W)
8	204	(1,5%)	5	(15.6%)	$1,6622 \times 10^{-2}$	0,17389
16	454	(3,4%)	5	(15.6%)	$1,8364 \times 10^{-2}$	0,19829
32	905	(6,8%)	20	(62.5%)	$2,5893 \times 10^{-2}$	0,31071
64	1596	(12%)	32	(100%)	$3,6444 \times 10^{-2}$	0,65653

Table 3.9: Various FPGA-based implementations of a PID controller

translates in shorter execution times. For the PID controller, the percentage change between the 8- and 64-bit version is a decrease of 24%, cf. table 3.10.

Bit-Width	Execution Time (μs)
8	0,9606
16	0,9924
32	1,4166
64	0,7499

Table 3.10: Varios CPU-based implementations of a PID controller

Validation of Empirical Approximation of Average Parallelism

To validate the approximation of average parallelism (AOS), the modelled growth rate of circuit size ($Size_{alg}$), which is the base to calculate the number of normalised operations (cf. equation 3.4), and the used reconfigurable resources of various PID implementations are compared in table 3.11.

Bit-Width	$SizeAlg_i(\mu m^2)$	Slices
8	15031	407
16	48048	1452
32	164442	4511
64	632708	17345

Table 3.11: Comparison of circuit size ($SizeAlg_i$) and utilised slices of various realisations of a PID controller

The values in table 3.11 for the FPGA-based realisation correspond to implementations of a PID controller without using any embedded multiplier. To compare these two data sets values were scaled to the range [0,1], as shown in figure 3.25. Furthermore, a trend estimation of both modelled an measured data was done, which shows a very similar trend for modelled and measured circuit size, cf. equation 3.37.

$$\begin{aligned} y_{modelled} &= 0,0065e^{1,244SizeAlg}, R^2 = 0,99 \\ y_{measured} &= 0,0068e^{1,239Slices}, R^2 = 0,99 \end{aligned} \tag{3.37}$$

Equation 3.37 was generated using an exponential regression, where equation 3.38 was used to calculate the least squares fit of both data sets presented in table 3.11.

$$y = ce^{bx} \tag{3.38}$$

Where c and b are constants, and e is the base of the natural logarithm. The quality of the exponential regression is calculated with a coefficient of determination (i.e., R^2), also shown in equation 3.37. Values of R^2 close to 1 indicate that the data regression is reliable [Raw01].

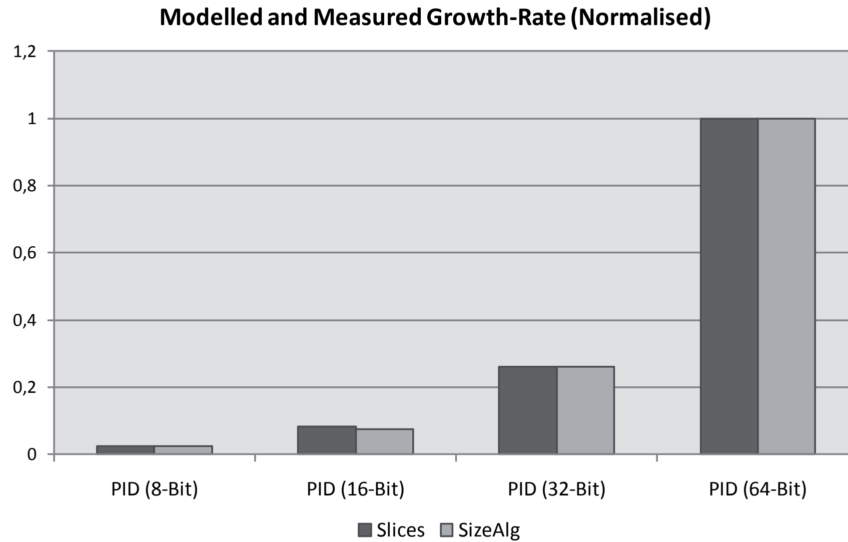


Figure 3.25: Comparison of modelled and measured circuit growth rate

Figure 3.25 shows a very similar trend of both modelled and measured data. This comparison is of relevance, because the approximation of average parallelism is based on modelled values, which are used to compute a relative measure of computational effort to weight operations of an algorithm (cf. section 3.1.3).

Comparison of Hardware/Software Realisations

When comparing the raw realisations results, it can be noticed that hardware implementations achieve a throughput up to two orders of magnitude higher than software realisations, while consuming two order of magnitude less energy. However, these values have to be normalised to the corresponding technologies, and the price to achieve the reported throughput (e.g., silicon resources) has to be taken into account. Therefore, metrics defined in section 3.2 are used to assess resources-utilisation for both architectures.

Computational density and energy efficiency (cf. section 3.2) are calculated for hardware and software implementations of the PID controller. Figure 3.26 shows computational density together with the average operations per step (AOS_i) and the

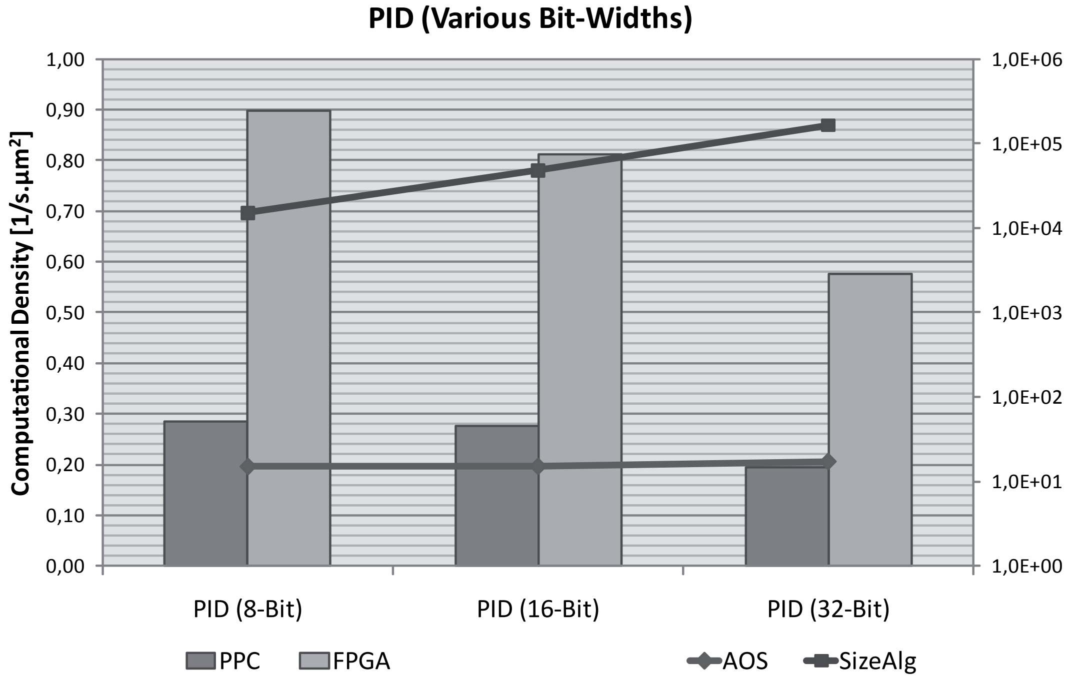


Figure 3.26: Algorithmic characterisation and Computational Density (C.D.) of hardware and software realisations of various versions of a PID controller

modelled circuit size ($SizeAlg_i$) for three versions of the PID controller presented in tables 3.9 and 3.10 (the 64-bit version is excluded from this comparison). As can be seen, hardware-based realisations show a better throughput/area ratio than software-based realisations. The gap between hardware and software realisations does not significantly changes, it goes from 103% (8-bits version) to 98% (32-bits version). The percent difference was calculated using equation 3.39. It can be noticed that the used number of bits for each operation does not strongly affect software implementations as long as the desired bit-width can be supported from the processor without emulation (i.e., required bit-width $\leq$ ALU bit-width). For hardware implementations bit-width of operations has a direct impact on the required reconfigurable resources, routing delays, and energy consumption, cf. figure 3.26.

$$Percent\ Difference = \frac{|Value_{Hardware} - Value_{Software}|}{\frac{1}{2} \cdot (Value_{Hardware} + Value_{Software})} \cdot 100 \qquad (3.39)$$

In this example, average parallelism (AOE_i) does not grow at the same ratio as algorithm size ($SizeAlg_i$) grows. This happens because the number of operations (i.e., nodes of the CDFG corresponding to the PID controller) does not increase, but only the bit-width of each operand, which means that data dependencies among operations stay constant, and only the problem size grows (i.e., the number of normalised operations

NormOpAlg). The effects of this can be seen in the reported results (cf. figures 3.26, and 3.27).

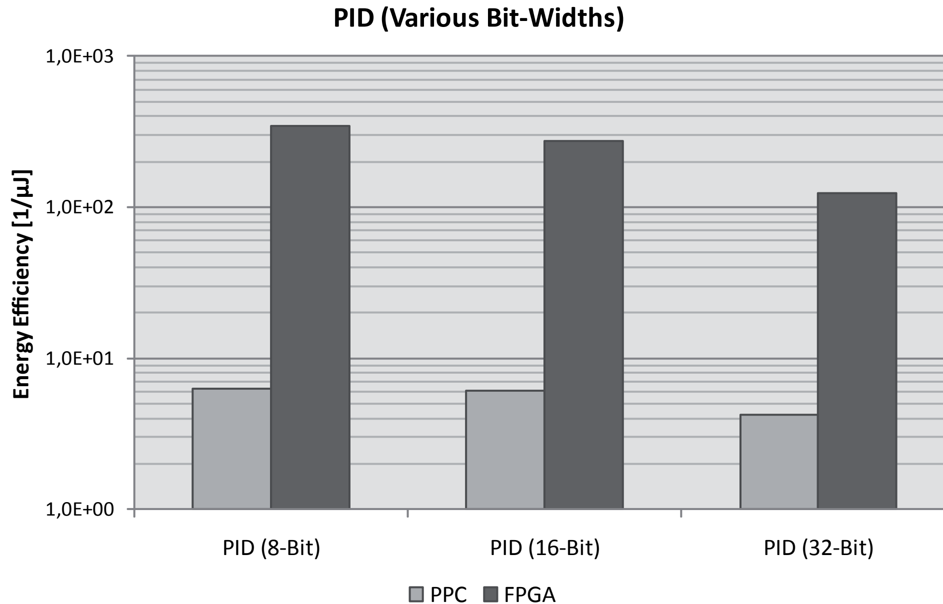


Figure 3.27: Energy Efficiency of hardware and software realisations of various versions of a PID controller (logarithmic scale is used for the Y axis)

Energy efficiency for both hardware and software implementations is presented in figure 3.27. Hardware-based implementations have a better throughput/area ratio than software-based realisations. The use of many processing elements in parallel allows for a higher throughput in spite of lower clock frequencies. However, the gap between hardware- and software-based realisations decreases, as shown in figure 3.27, because for software-based realisations increasing the bit-width does not greatly affect power consumption (i.e., execution time does not vary significantly), as long as the used bit-width is supported by the ALU of the architecture at hand. The gap between hardware and software realisations decreases from 193% to 186% for the 8-bit and 32-bit versions of the PID controller correspondingly.

In the next section a second benchmark is analysed, a state-feedback controller, for which the effects of increasing number of average operations per step on the resource utilisation of both hardware and software architectures is explored.

3.5.2 State-Feedback Controller

State-feedback is a very popular approach to control a linear, time-invariant system. The system to be controlled is described by a set of first order differential equations, as presented in equation 3.40, called state-space representation.

$$\begin{aligned} \dot{\underline{x}} &= \underline{\underline{A}}\,\underline{x} + \underline{\underline{B}}\,\underline{u} \\ \underline{y} &= \underline{\underline{C}}\,\underline{x} + \underline{\underline{D}}\,\underline{u} \end{aligned} \tag{3.40}$$

where matrices $\underline{\underline{A}}, \underline{\underline{B}}, \underline{\underline{C}}, \underline{\underline{D}}$ are constant coefficient matrices. A block diagram of equation 3.40 is presented in figure 3.28.

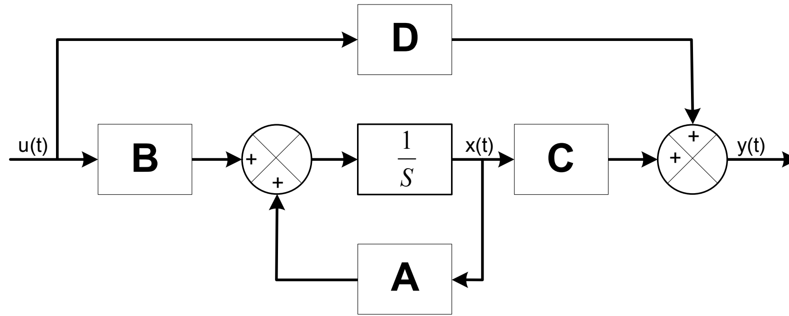


Figure 3.28: Block diagram of a state-space representation of a system

A state-feedback controller for a system described by equation 3.40 can only be designed when the system is controllable and observable. If the system is not observable other techniques have to be used, such as the use of state observers (see section 3.5.3).

Controllability is a property of linear systems, which determines whether a system can or cannot be driven from any initial condition to the origin, via a suitable selected input. In other words, the system is said to be controllable if the values of all entries of the state vector can be made zero after some amount of time via a choice of a suitable input. Observability is a measure for how well internal states of a system can be deduced by knowledge of its external outputs. A system is observable if, for any possible sequence of state and control vectors, the current state of the system can be determined in finite-time using only the outputs of the system [Kal59].

The basic idea of state-feedback is that having a fully observable and controllable system, a controller can be designed based solely on the internal states of the system multiplied by a feedback gain, as shown in figure 3.29.

Where $\underline{\underline{K}}$ is a $r \times n$ matrix, n being the number of states of the system and r the number of inputs. A realistic implementation of a state-feedback controller includes a scale factor to the reference signal $\bar{N}$, as shown in figure 3.29. Thus the equation of the state-feedback controller is given by equation 3.41.

$$\underline{u}(t) = -\underline{\underline{K}} \cdot \underline{x}(t) + \underline{\underline{\bar{N}}} \cdot \underline{r}(t) \tag{3.41}$$

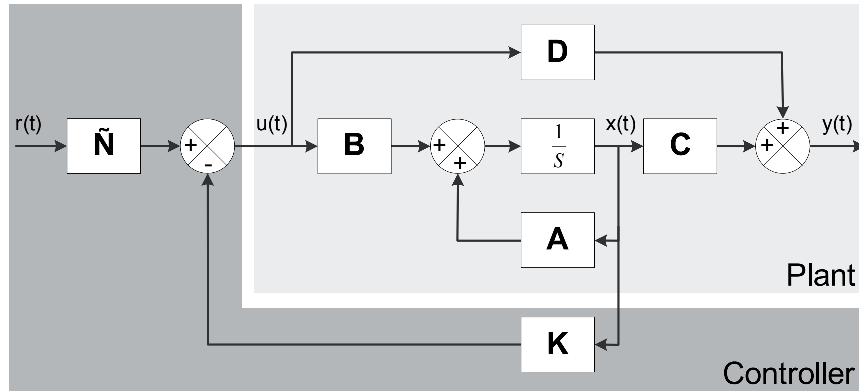


Figure 3.29: Block diagram of a state-feedback controller

When substituting continuous-time by difference equation, feedback-controller is given by:

$$\underline{u}(k+1) = -\underline{\underline{K}} \cdot \underline{x}(k) + \underline{\underline{\bar{N}}} \cdot \underline{r}(k) \tag{3.42}$$

where k is the sampling time of the controller. In the next section, the algorithmic characteristics of equation 3.42 are analysed.

Algorithmic Characterisation

As an example, figure 3.30 shows a suitable CDFG representation of equation 3.42, having two states ($n = 2$) and two reference inputs ($r = 2$), cf. equation 3.43.

$$\begin{bmatrix} u_1(k+1) \\ u_2(k+1) \end{bmatrix} = -\begin{bmatrix} K_{11}K_{12} \\ K_{21}K_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} \bar{N}_{11}\bar{N}_{12} \\ \bar{N}_{21}\bar{N}_{22} \end{bmatrix} \cdot \begin{bmatrix} r_1(k) \\ r_2(k) \end{bmatrix} \tag{3.43}$$

For this specific example the CDFG of a state-feedback controller has a total of 28 operations: 8 multiplications, 6 additions, and 14 storage operations. The number of operations increase when increasing the number of reference inputs (r) or the number of states (n).

The schedule presented in figure 3.31 shows that a two-state, two-intput state-feedback controller has a latency of five steps (i.e., $lat_{alg} = 5$) when the nodes of graph 3.30 are weighted with a unit delay. The latency of the graph varies when using $Steps_{alg}$ to weight the execution time of each node as reflected in figure 3.32.

The number of operations of a state-feedback controller can be expressed as a function of the number of states St and inputs In of the controller as follows:

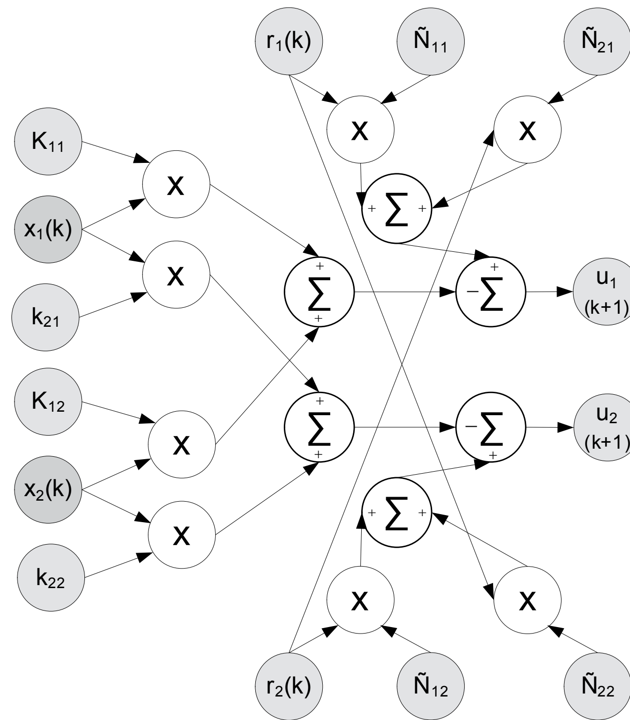


Figure 3.30: Cyclic Data Flow Graph of a feedback controller for a two-state two-input system, cf. equation 3.43

$$\begin{aligned} Multiplications &= (St \cdot In) + In \\ Additions &= St \cdot In \\ Storage &= St \cdot 2 + In \cdot 3 \end{aligned} \tag{3.44}$$

Using equation 3.44 enables to easily explore the algorithmic characteristics of a state-feedback controller, such as the latency of the equivalent CDFG, or the AOS_i. For this purpose, the number of states, the number of inputs, and bit-width, are used as parameters. The results of this characterisation are shown in figure 3.32. Equation 3.44 has been empirically derived, and used for the examples shown in this comparison.

As can be observed in figure 3.32, average operations per step (AOS_i) of the state-feedback grows rapidly when increasing the number of inputs, outputs, states, and bit-width of operations.

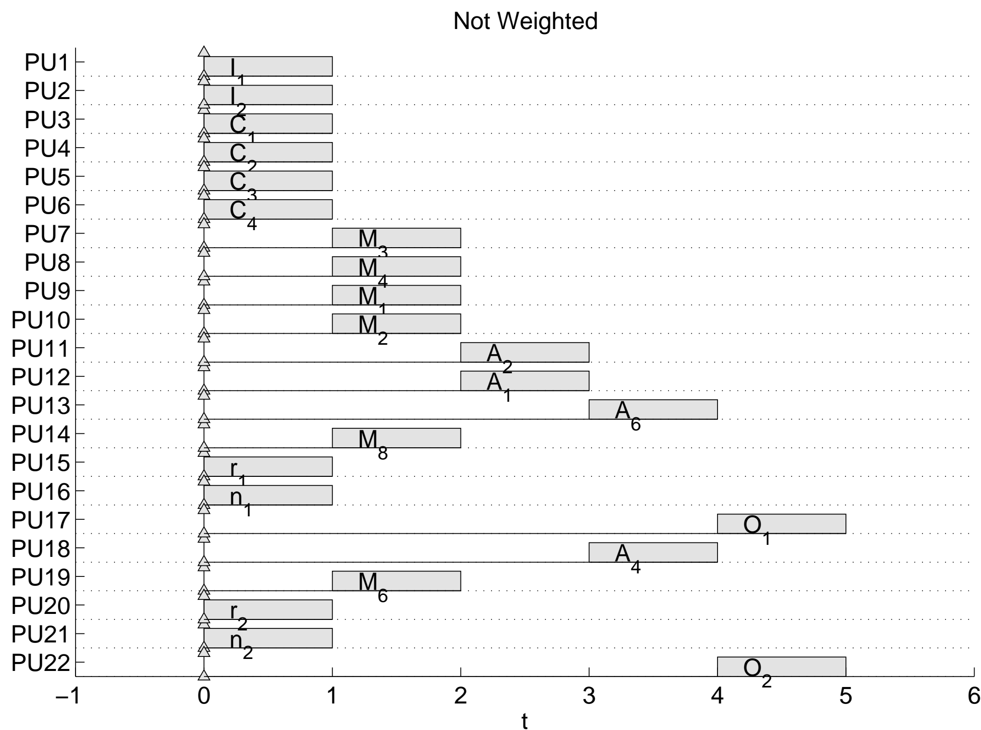


Figure 3.31: Scheduling for the state feedback controller depicted in figure 3.30 with operations not being weighted

Hardware Implementation

Different state-feedback controllers were implemented, using various numbers of inputs, states, and bit-widths. The main limitation was the number of available I/Os of the Spartan-3 (333 I/Os), because pads were assigned to each input and output port, to maximise parallelism. For an actual realisation, another methods are preferred to save IO blocks.

A design was implemented for each combination of number of I/Os and states, using the Synplify DSP toolbox. Because most of the multiplications in the design where constant gains, LUT-based multipliers where used. Table 3.12 shows hardware implementation results of state-feedback controllers using different number of inputs and states.

As it was the case for the PID controller implementation presented in the previous section, increasing the bit-width results in lower achievable clock frequencies, because of the increased routing delay. This can be observed in the $2/1$ version (2 inputs and outputs, and 1 state) of the state feedback controller, where the execution time of the 64-bits version increased 220% in contrast to the 8-bits version. Increasing the number of operations results only on small variations of the execution time. This can

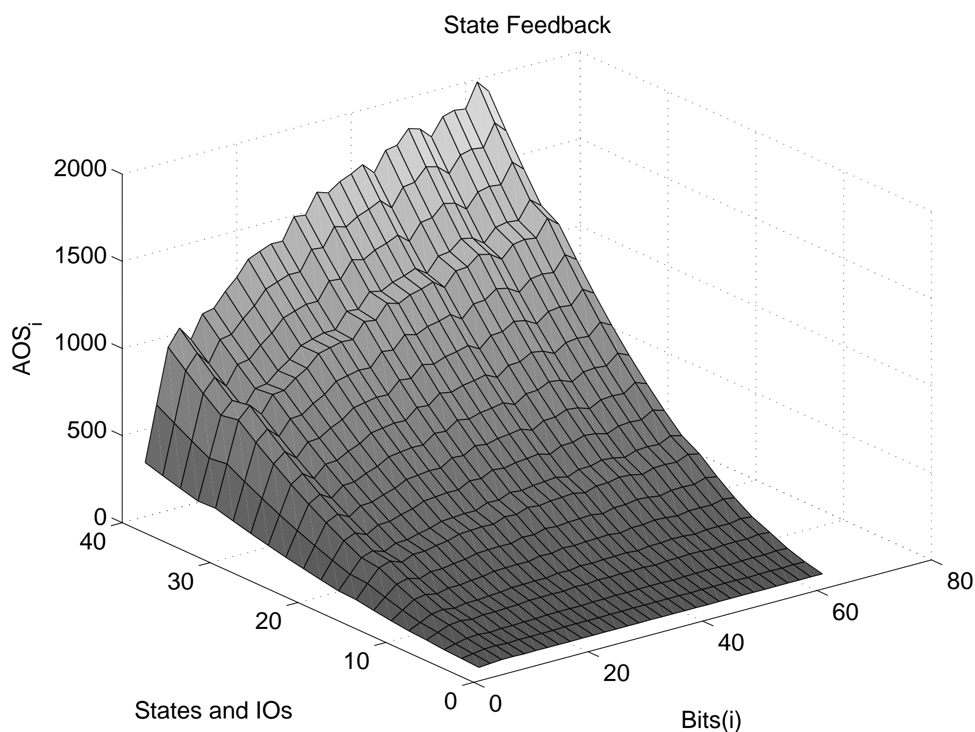


Figure 3.32: Normalised algorithmic characterisation of a state-feedback controller (see figure 3.30)

I / S	Bit-Width	Slices		Time (ns)	Power (W)
2 / 1	8	116	(0,8%)	7,487	0,1819
2 / 1	64	1222	(9,2%)	23,990	0,2567
4 / 2	8	324	(2,4%)	8,111	0,1796
4 / 2	32	2760	(20,7%)	24,059	0,3092
6 / 8	8	1606	(12,6%)	8,144	0,2592
6 / 8	16	4906	(36,9%)	17,502	0,4140
12 / 16	8	4555	(34.2%)	8,486	0,4855

Table 3.12: Various FPGA-based implementations of a state-feedback controller

be observed when comparing all implementations of the state-feedback controller, which use 8 bits for each operation, where the difference in execution time between the fastest and the slowest design is about 13% (from 7,48ns to 8,48ns, cf. table 3.12). In contrast, the power consumption increases more than 180% (from 0,17W to 0,48W), as a result of the increment of switching activity in the device, among other factors.

Software Implementation

Implementation results are shown in table 3.13. These execution times represent mean values over several hundred measurement. Furthermore, the state-feedback controller was implemented using the flow described in section 3.4.2.

I / S	Bit-Width	Execution Time (μs)
2 / 1	8	0,6227
2 / 1	64	0,4696
4 / 2	8	0,7439
4 / 2	32	0,7985
4 / 2	64	0,5848
6 / 8	8	2,3378
6 / 8	16	2,6473
6 / 8	64	1,5499
12 / 16	8	12,7166
12 / 16	64	4,7303

Table 3.13: Varios CPU-based implementations of a state-feedback controller

As it was the case for the PID controller presented previously, varying bit-width does not affect greatly the execution time of an algorithm. This can be noticed when comparing 8-bit realisations with 16- and 32-bit realisations. For instance, the 4/2 version of the state-feedback controller increases its execution time on 6,8% when using 32-bit instead of 8-bit operations (from 0,74 to 0,79 μs). The same example, when implemented in hardware has an increment of more than 196% of the execution time (from 8,11 to 24,05 ns). When using a specialised hardware unit, as in the case of 64-bits realisations, execution times decreased significantly. The FPU of the PPC achieves 4 cycle latency, 2 cycle throughput, for a double-precision multiply-add operation, in contrast to the multi-cycle operation mode of the integer unit IU1 (cf. section 3.3.1). 64-bits realisations are not used for the comparison presented in this chapter, because FPGA-based realisations use only fixed-point arithmetic units.

Validation of Empirical Approximation of Average Parallelism

As in the case of the PID controller, first the modelled and measured circuit growth-rate are compared in table 3.14. A trend estimation is also shown in equation 3.45.

$$\begin{aligned} y_{modelled} &= 0,0050e^{1,3172SizeAlg}, R^2 = 0,99 \\ y_{measured} &= 0,0068e^{1,2612Slices}, R^2 = 0,99 \end{aligned} \tag{3.45}$$

I / S	$SizeAlg_i(\mu m^2)$	Slices
2 / 1	9050	116
4 / 2	26732	324
6 / 8	118633	1606
12 / 16	444435	4555

Table 3.14: Comparison of circuit size ($SizeAlg_i$) and utilised slices of various 8-bit state-feedback implementations

This trend estimation shows the similarities of both data sets, and the accuracy of the regression results. Scaled data is shown in figure 3.33. Both data series show a similar trend, which also validate modelled results.

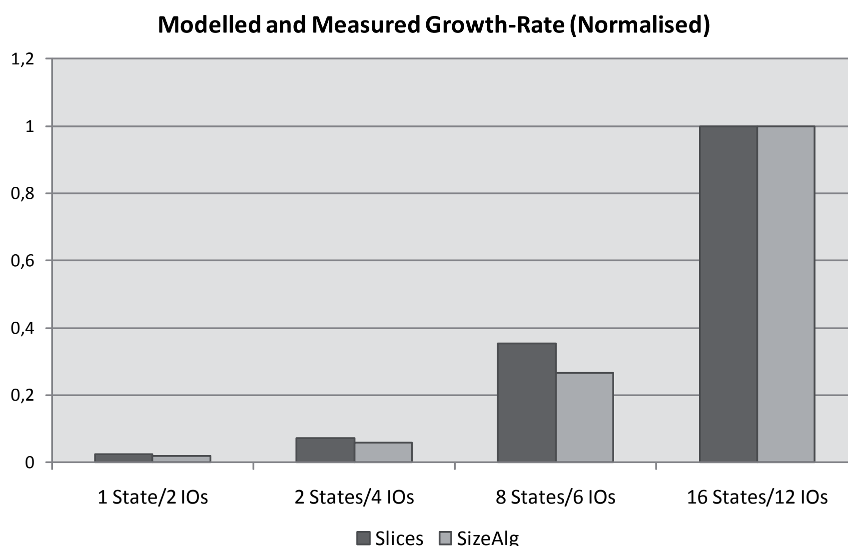


Figure 3.33: Comparison of modelled and measured circuit grow rate

Comparison of Hardware/Software Realisations

To compare hardware and software realisations, only designs using 8 bits as base word-length are used. As in the case of the PID controller, when comparing raw implementation results, it is shown that an FPGA-based realisation results in a speedup of up to three orders of magnitude when compared to a software-based state-feedback, while using up to three orders of magnitude less power consumption (for the 12/16 version). Using the proposed metrics, it is shown that hardware-based implementations of the state observer algorithm result in a better throughput/area ratio than software-based realisations, cf. figure 3.34. For the state-feedback controller the gap between hardware-

and software-based realisations realisations increases as the average operation per step (*AOS*) and circuit size (*SizeAlg*) of the algorithm increase. Cf. figure 3.34, where the left axe has a logarithmic scale.

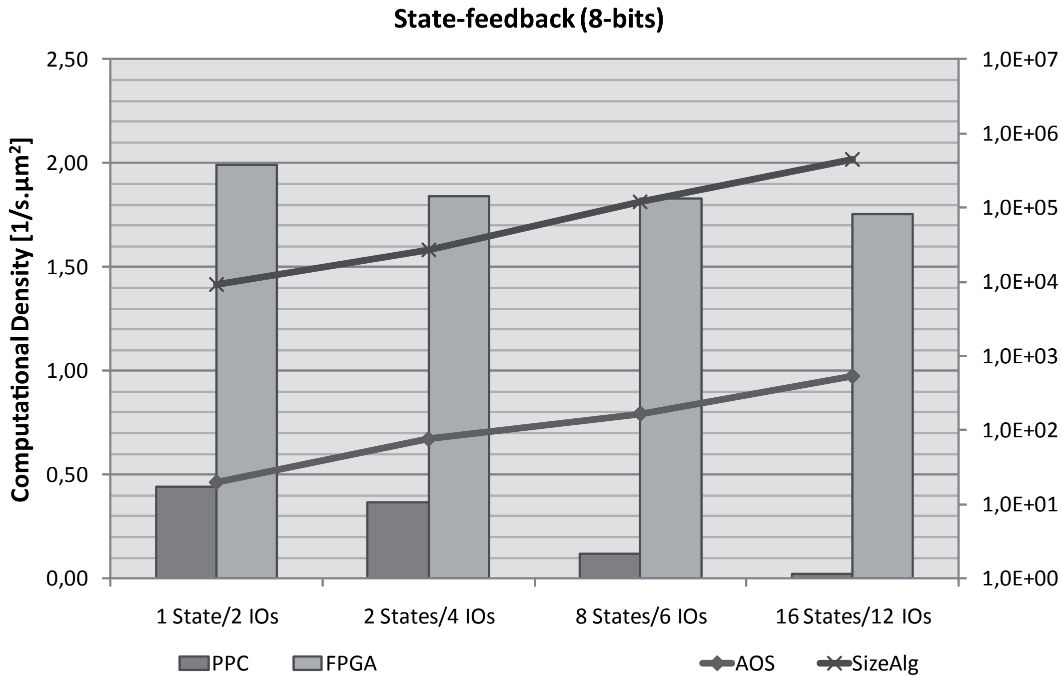


Figure 3.34: Computational Density of hardware and software realisations of a state-feedback controller

In figure 3.34, it is shown that the gap between hardware- and software-based realisations increases as the problem size grows, provided that the *AOS* (i.e., average parallelism) increases, too. This is not the case of the PID controller, presented in the previous section, where an increasing bit-width shortened the gap between hardware- and software-based realisations. For the state-feedback controller the gap between hardware- and software-based implementation changes from a 127% for the $2/1$ version to a 195% for the $12/16$ version. Percentage difference was calculated using equation 3.39.

Energy efficiency of hardware and software realisations show a similar trend as the computational density comparison: the throughput/power ratio decreases, as the number of inputs and output (e.g., the size of the algorithm) increases, cf. figure 3.35. However, the gap between hardware- and software-based realisations does not vary as much as in the case of computational density. Values of percent difference fluctuate between 194% and 199% for the $2/1$ and $12/16$ versions of the state-feedback controller respectively. This results from the fact that power consumption increases significantly for hardware-based realisations, when the problem size grows.

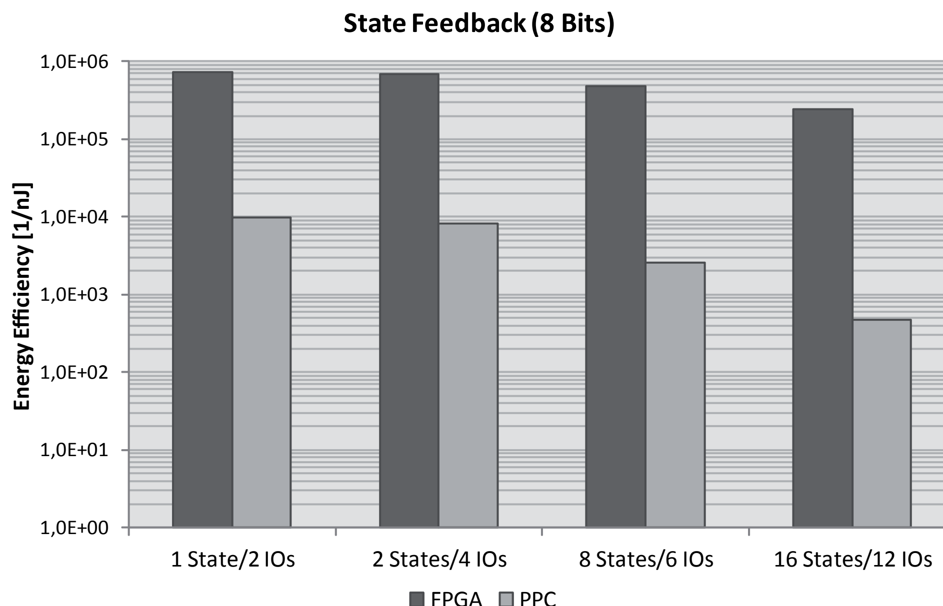


Figure 3.35: Energy Efficiency of hardware and software realisations of a state-feedback controller

In this example, the gap between hardware- and software-based realisations regarding computational density raises, as the problem size grows. However, this trend is bounded by the available resources of reconfigurable architectures and the intrinsic parallelism of the algorithm. As a consequence for hardware-based realisations, if the problem size (i.e., the size of the controller) grows to a point where its realisation requires more resources than available in the selected device, the gap between hardware- and software-based realisations diminishes or it even turns in favour of software-architectures (i.e., software-based realisations would have better throughput/area ratios). In the next section this argument is clearly exemplify, using a state observer algorithm.

3.5.3 State Observer

A state observer, also called state estimator, is a subsystem of a controller, which infers internal states of a linear time-invariant system, based on the measurement of the outputs of the controlled system and the controller itself [Oga87].

State observers are typically used in mechatronic systems when states of a plant can not be directly measured, or the use of sensors to measure states of a system is avoided to reduce implementation costs. Furthermore observers are used to achieve redundancy in order to detect measurement errors [Dor11]. In those cases, the states

of the system have to be estimated based on the available measurable data. A block diagram of a full state observer is presented in figure 3.36.

Figure 3.36: Block diagram of a control system with a state observer

The observer has as an input $\underline{u}(t)$ and $\underline{y}(t)$, giving the estimated state vector $\underline{\tilde{x}}(t)$ as output. The mathematical equations of a full state observer corresponding to the block diagram depicted in figure 3.36 are:

$$\begin{aligned} \dot{\underline{\tilde{x}}}(t) &= \underline{\underline{A}}\,\underline{\tilde{x}}(t) + \underline{\underline{L}}(\underline{y}(t) - \underline{\tilde{y}}(t)) + \underline{\underline{B}}\,\underline{\tilde{u}}(t) \\ \underline{\tilde{y}}(t) &= \underline{\underline{C}}\,\underline{\tilde{x}}(t) + \underline{\underline{D}}\,\underline{u}(t) \end{aligned} \tag{3.46}$$

where matrices $\underline{\underline{A}}, \underline{\underline{B}}, \underline{\underline{C}}, \underline{\underline{D}}$ are constant coefficient matrices. When substituting continuous-time by difference equation, the state observer equation becomes:

$$\begin{aligned} \dot{\underline{\tilde{x}}}(k+1) &= \underline{\underline{A}}\,\underline{\tilde{x}}(k) + \underline{\underline{L}}(\underline{y}(k) - \underline{\tilde{y}}(k)) + \underline{\underline{B}}\,\underline{\tilde{u}}(k) \\ \underline{\tilde{y}}(k) &= \underline{\underline{C}}\,\underline{\tilde{x}}(k) + \underline{\underline{D}}\,\underline{u}(k) \end{aligned} \tag{3.47}$$

Algorithmic Characterisation

As an example, figure 3.37 shows a CDFG of a state observer for a system with two inputs, one state, and two outputs; corresponding to equation 3.48. As in the previous sections, the CDFG was manually derived from the mathematical description of the observer (cf. equation 3.48).

$$
\begin{aligned}
\tilde{x}(k+1) &= A_1 \cdot \tilde{x}(k) + \left[L_1 L_2\right] \cdot \begin{bmatrix} y_1(k) - \tilde{y}_1(k) \\ y_2(k) - \tilde{y}_2(k) \end{bmatrix} + \left[B_1 B_2\right] \cdot \begin{bmatrix} \tilde{u}_1(k) \\ \tilde{u}_2(k) \end{bmatrix} \\
\begin{bmatrix} \tilde{y}_1(k) \\ \tilde{y}_2(k) \end{bmatrix} &= \left[C_1 C_2\right] \cdot \tilde{x}(k) + \begin{bmatrix} D_{11} D_{12} \\ D_{21} D_{22} \end{bmatrix} \cdot \begin{bmatrix} \tilde{u}_1(k) \\ \tilde{u}_2(k) \end{bmatrix}
\end{aligned}
\tag{3.48}
$$

The state observer described by equation 3.48 has a total of 39 operations, from which 11 are multiplications, 18 are memory operations, and 10 are additions. Having the CDFG from equation 3.48, the next step is to find out the minimum execution time, corresponding to the critical path of the graph.

Figure 3.37: Cyclic Data Flow Graph of the state observer corresponding to equation 3.48

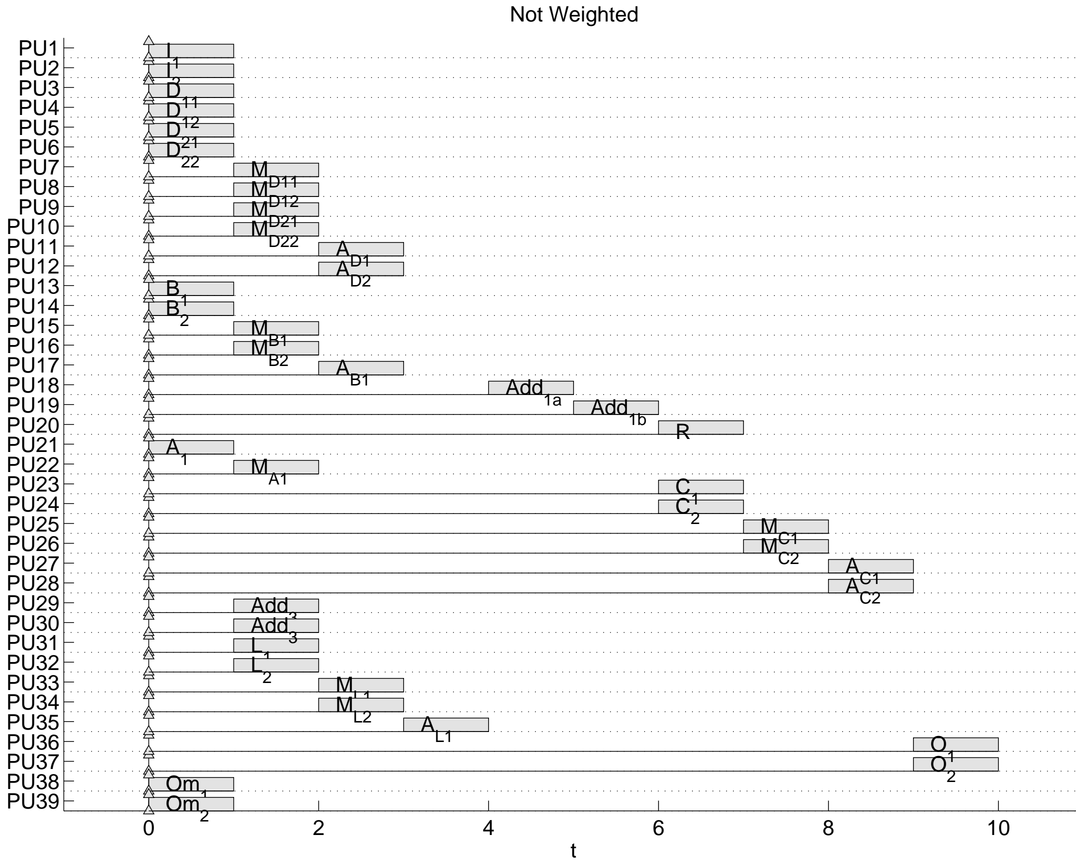


Figure 3.38: Scheduling for a state observer depicted in figure 3.37 without operation weighting. Three cycles are shown in the figure

The cyclic scheduling of a full state observer with 2 inputs, 2 outputs and one state is presented in figure 3.38. The algorithm has a latency of $lat_{alg} = 10$ steps, and a execution period $w = 1$.

As in the previous section, the number of operations can be expressed as a function of the number of inputs In (i.e., $\tilde{u}(k)$), outputs Out (i.e., $\tilde{y}(k)$) and states St (i.e., $\tilde{x}(k)$) of the system, as follows:

$$\begin{aligned} Mult_A &= St \cdot St \\ Add_A &= (St-1) \cdot St \\ Stor_A &= St \cdot St \end{aligned} \tag{3.49}$$

where $Mult_A$ are the multiplications required for the A matrix multiplier (cf. figure 3.36). Add_A are the additions required for the same block, and $Stor_A$ are the number

of storage operations. These equations have been derived empirically based on the examples used in this section. In the same way, the required operations for the rest of matrix multiplications can be calculated with the following equations.

$$\begin{aligned}
Mult_B &= In \cdot St \\
Add_B &= (St-1) \cdot In \\
Stor_B &= In \cdot St \\
Mult_C &= St \cdot Out \\
Add_C &= (Out-1) \cdot St \\
Stor_C &= St \cdot Out \\
Mult_D &= In \cdot Out \\
Add_D &= (Out-1) \cdot In \\
Stor_D &= In \cdot Out \\
Mult_L &= Out \cdot St \\
Add_L &= (St-1) \cdot Out \\
Stor_L &= Out \cdot St
\end{aligned} \tag{3.50}$$

Combining the terms of equations 3.49 and 3.50, the total number of multiplications, additions, and storage operations can be calculated by equation 3.52.

$$\begin{aligned}
Mult_{Observer} &= Mult_A + Mult_A + Mult_A + Mult_A + Mult_L \\
Add_{Observer} &= Add_A + Add_B + Add_C + Add_D + Add_L + \\
&\quad add_{AZ} + add_{CD} + add_{LB} + add_{LO} \\
Stor_{Observer} &= Stor_A + Stor_B + Stor_C + Stor_D + Stor_L + Stor_{IOs} + Stor_{Delay}
\end{aligned} \tag{3.51}$$

where add_{CD} are the additions required to add the output of the matrix multiplier C with the output of the matrix multiplier D, which corresponds to the number of outputs of the state observer, i.e., $add_{CD} = Out$. Correspondingly, $add_{AZ} = St$, $add_{LB} = St$, $add_{LO} = Out$. $Stor_{IOs}$ is the number of storage elements required for the inputs and outputs of the system, i.e., $Stor_{IOs} = 2 \cdot O + I$. Finally, $Stor_{Delay}$ corresponds to the number of storage operations required for the delay block of the state observer, which is equal to the number of states, i.e., $Stor_{Delay} = St$. Based on equations 3.49 to 3.52 the parallelism of a state observer as function of its inputs, outputs and states can be derived, as presented in figure 3.39, where the average normalised operations per

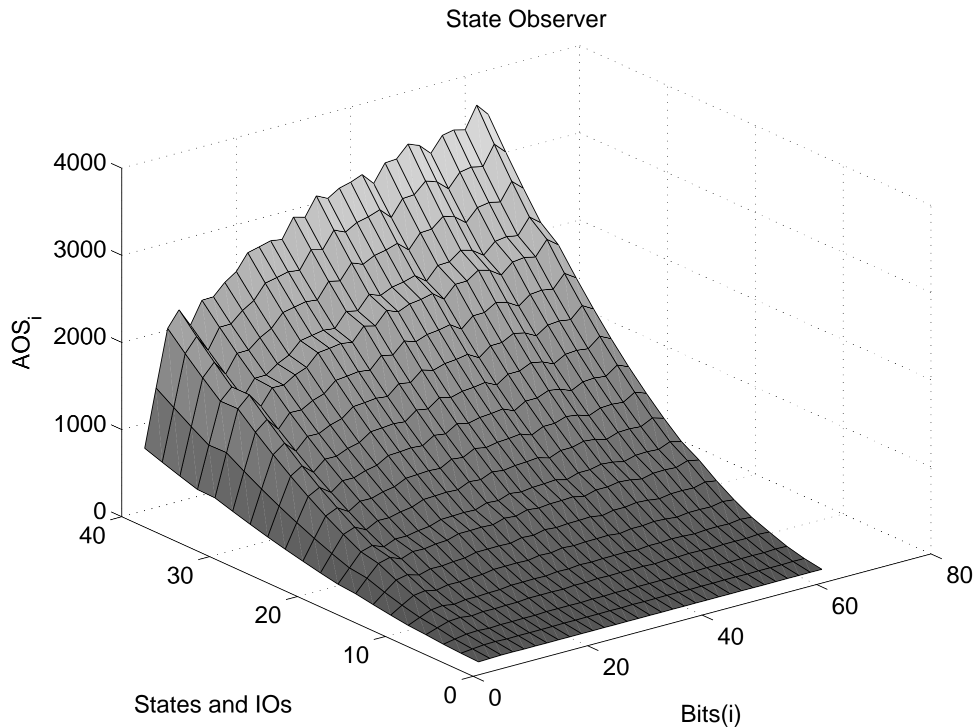


Figure 3.39: Normalised algorithmic characterisation of a state observer

normalised step (AOS_i) is shown, as function of the based bit-width (i.e., i), and the number of inputs, outputs and states.

Although the growth-rate of *AOS* observed in figure 3.39 follows a similar trend that the state-feedback controller shown in the previous section, in the case of the state observer parallelism grows more rapidly.

Hardware Implementation

Table 3.15 shows implementation results of different versions of a state observer, where the number of input/outputs, states and base bit-width are changed.

For the last two designs listed in table 3.15, folding optimisation (resources sharing through time-multiplexing) was used, because the resource utilisation of the design was greater than the available on the selected FPGA. This resulted in a significant decrease in the achievable execution time, in comparison to the previous versions. Taking the 6/8 (6 IOs and 8 states) state observer as an example, and comparing the 8-bit version with the 16-bit version, it can be seen that the execution time increases more than 700%. In contrast to this, when comparing the 3/2 state observer, for which no folding optimisation was used, the execution time of the 32-bit version increased about 100% when compared to the 8-bit version. Longer execution times are partially

I / S	Bit-Width	Slices		Time (ns)	Power (W)
2 / 1	8	256	(1,9%)	11,481	0,1715
2 / 1	64	4685	(35,2%)	28,733	0,3854
3 / 2	8	621	(4,7%)	12,489	0,2012
3 / 2	32	6021	(45,2%)	25,445	0,3435
6 / 8	8	5261	(39,5%)	20,389	0,3789
6 / 8	16	11806	(88,7%)	166,306	0,5523
12 / 16	8	13310	(99%)	959,171	0,6630

Table 3.15: Various FPGA-based implementations of a state observer

caused by longer routing delays (e.g., caused by operations with larger bit-widths), but it is also caused by having to share reconfigurable resources in time, as it is the case for the 16-bits implementation of the 6/8 state observer.

When looking at the effect of size of the algorithm on the execution time, it can be seen that it does not have the same impact as when increasing the bit-width, as long as there are enough resources available, otherwise resource time-multiplexing must by used. Taking the 8-bit versions of the 2/1, 3/2, 6/8, and 12/16 state observers presented in table 3.15 as example, the increase of execution time goes from 8% (3/2 state observer) to 77% (6/8 state observer), when using the 2/1 version of the state observer as reference. The execution time increases about 8254% for the 12/16 version, where folding optimisation was used.

Power consumption does not vary as much as the execution time. Taking the 8-bit versions of the 2/1, 3/2, 6/8, and 12/16 state observers presented in table 3.15 as example, the increase of power consumption goes from 17% (3/2 state observer) to 120% (6/8 state observer), to 286% (12/16 state observer), when using the 2/1 version of the state observer as reference. This trend can be explained by the increasing resource utilisation of the mentioned realisations (cf. table 3.15), and the decrement of the achievable clock frequency.

Software Implementation

Implementation results of various software-based state observer are shown in table 3.16. Reported execution times represent mean values over several hundred measurement (cf. section 3.4.2).

For software-based realisations, the problem size has a direct impact on the execution time. This effect can be observed in the implementation results presented in table 3.16. Taking all 8-bit implementations as example, and using the 2/1 version of the state observer as reference, it can be seen that the execution time increases 41%, 681%, 2470% for the 3/2, 6/8, and 12/16 versions of the state observer, correspondingly. On

I / S	Bit-Width	Execution Time (μs)
2 / 1	8	0,7469
2 / 1	64	0,5075
3 / 2	8	1,0560
3 / 2	32	1,6136
3 / 2	64	1,6545
6 / 8	8	5,8348
6 / 8	16	6,2515
6 / 8	64	5,9999
12 / 16	8	19,1969
12 / 16	64	18,5102

Table 3.16: Varios CPU-based implementations of a state observer

the contrary, the effect of increasing the bit-width is not as significant, which can be observed when comparing the 8-bit with the 32-bit versions of the $3/2$ state observer (cf. table 3.16). The increment of execution time is about 52%. Furthermore, when comparing the 8-bit with the 16-bit versions of the $6/8$ state observer, the execution time increase only 7%.

To compare hardware and software realisations, only the 8-bit version of all implemented designs is used. First, the approximation of average parallelism is validated, by comparing measured and modelled circuit size, as explained in the next section.

Validation of Empirical Approximation of Average Parallelism

Table 3.17 and figure 3.40 show a comparison of modelled and measured circuit growth-rate of various state observer 8-bit implementations. A bigger device (Virtex II, XCV8000) was selected for this comparison, because neither embedded multipliers, nor folding optimisation were used.

I / S	$SizeAlg_i(\mu m^2)$	Slices
2 / 1	14261	232
3 / 2	39764	584
6 / 8	309937	4963
12 / 16	1236765	20775

Table 3.17: Comparison of circuit size ($SizeAlg_i$) and utilised slices of various 8-bit state observer implementations

Equation 3.52 shows the trend for both modelled and measured data. These trend equations show a great similarity, as in the previous examples.

$$
\begin{aligned}
y_{modelled} &= 0,0021e^{1,5442SizeAlg}, R^2 = 0,98 \\
y_{measured} &= 0,0019e^{1,5624Slices}, R^2 = 0,97
\end{aligned}
\tag{3.52}
$$

Scaled data is shown in figure 3.40, where the similarity of modelled and measured data can be observed.

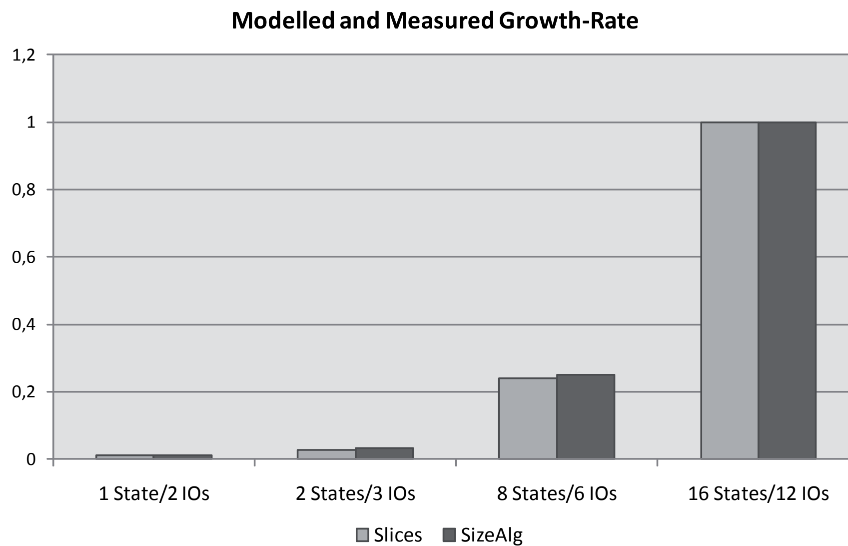


Figure 3.40: Comparison of modelled and measured circuit grow rate

Comparison of Hardware/Software Realisations

In the previous examples, having a higher degree of parallelism increases the gap between hardware- and software-based implementations. However, in the case of the state observer, the device utilisation grew to a point where resources had to be time-shared. The effects of this can be seen (in the case of the last two realisations) in figure 3.41.

The gap between hardware- and software-based realisations goes from a percent difference of 111% (2/1 version), to 175% (6/8 version). For the last design, the gap shrinks to only 8%.

As can be seen in figure 3.42, the low execution time achieved by the 16-States/12-IOs state observer resulted in a low power consumption (although the device utilisation

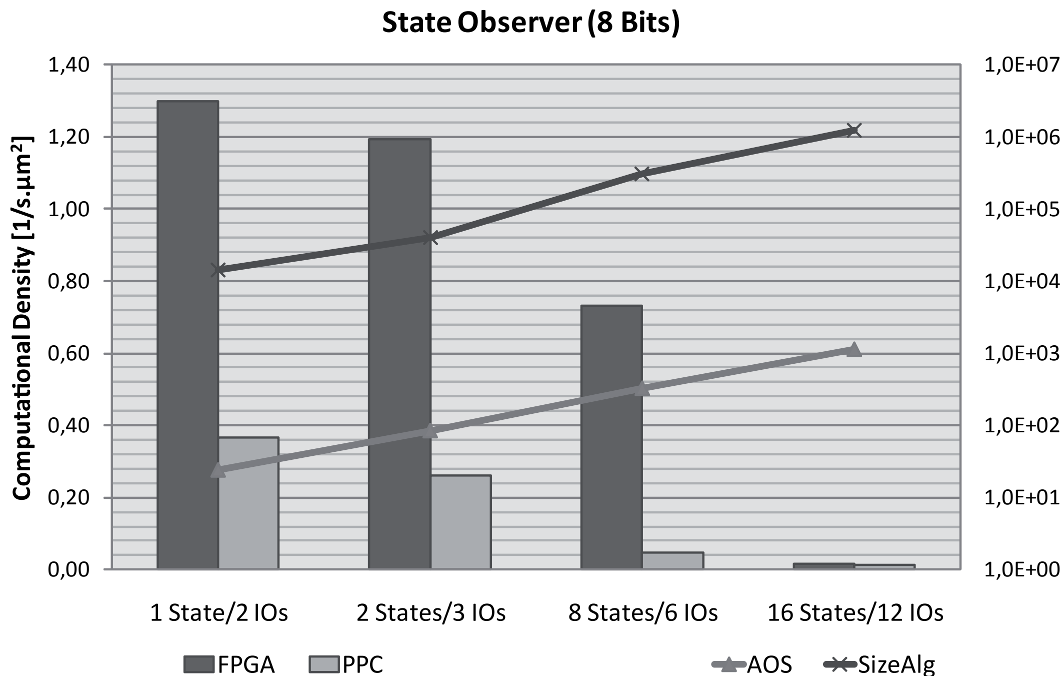


Figure 3.41: Computational Density of hardware and software realisations of an state observer

was very high), because of the reduced maximum allowable clock frequencies achieved for that design. On the contrary, for software implementations an increment of the problem size (i.e., the number of instructions to be executed) implies a longer execution time, which derived in an increment of the power consumption.

For Energy Efficiency the gap between hardware- and software-based realisations goes from a percent difference of 193% (1/2 version) to 194% (6/8 version). For the last design, the gap is reduced to a percent difference of 133%, because of the effects of resource time-sharing.

3.6 Summary

This chapter presents a quantitative comparison between general purpose processors and field programmable gate arrays for embedded control applications. Metrics to characterise algorithmic properties of a controller are defined, and metrics to assess implementation results in both reconfigurable hardware and software devices are taken from literature and adjusted to the application field. Average operations per steps (*AOS*) is used to measure average parallelism, which together with the size of the algorithm (*SizeAlg*) are used to characterise selected benchmarks. Furthermore,

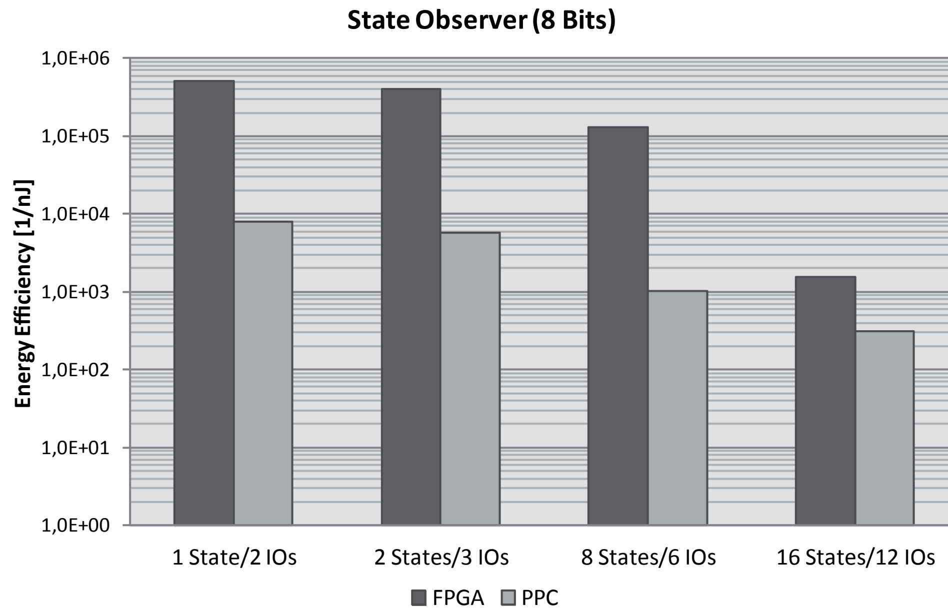


Figure 3.42: Energy Efficiency of hardware and software realisations of an state observer

the metrics computational density ($C_{density}$) and energy efficiency ($E_{efficiency}$) are used to assess resource utilisation of the selected benchmarks. Three representative control algorithm were chosen for this comparison: a PID controller, a state-feedback controller, and a full state observer.

When comparing raw implementation results of presented benchmarks, it is shown that FPGA-based realisations achieve execution times several orders of magnitude above those from software-based realisations, while having less power consumption. Taking the state-feedback controller as an example, it is shown that the speedup grows from 83 to 1498 times for the smallest and the biggest design, respectively (cf. section 3.5.2).

Moreover, it is shown that an FPGA-based implementation leads to a higher $C_{density}$ and $E_{efficiency}$, which implies a more efficient use of resources. Presented implementation results show that increasing the $SizeAlg$ with constant AOS causes a reduction of the gap between hardware- and software-based realisations, as shown for the PID controller, where the percentage difference between FPGA- and CPU-based realisations regarding computational density goes from 103% (8-bits version) to 98% (32-bits version). A similar trend can be observed for energy efficiency values. Furthermore, when the algorithm to be implemented has an increasing AOS when increasing $SizeAlg$ the gap between hardware- and software-based realisations increases, too. This effect was shown with the state-feedback controller, where the percentage difference in terms of computational density goes from 127% to 195% for the smallest and the biggest

designs, respectively. This trend is bounded by the available resources of the FPGA. When the *SizeAlg* grows to a point, where more resources than available in the selected device are required, performance decreases along with the gap between hardware- and software-based realisations, because resource have to be time-shared. This is the case of the state observer example; folding optimisation (resources time-sharing) is used to allow the realisation of the two biggest versions of this design. Thus, the throughput/area ratio decreases significantly, going from 175% to 8% for the biggest design implemented without resources time-sharing and the biggest design, respectively.

Nevertheless, there are design improvements, which do not sacrifice parallelism and improve resource utilisation. One of them is bit-width optimisation. The effect of having long bit-widths was also shown in the realisation results presented in this chapter, where increasing bit-width comes along with increasing execution times. Customisation of the bit-width of arithmetic operands is possible when using an FPGA, which can lead to a significant reduction of the utilisation of reconfigurable resources.

The shortcoming of resources can also be caused by having more resources occupied than those required for the current situation. This is the case of control algorithms requiring adjustments at run-time. In the next chapter this problem is analysed and the utilisation of dynamic hardware reconfiguration is explored to improve the resource utilisation of this kind of control systems.

4 Run-Time Hardware Reconfiguration

In the previous chapter it was shown that FPGA-based realisations lead to a higher computational density (throughput/area) and a higher energy efficiency (throughput/power) than software-based implementations for control algorithms having a high degree of parallelism. The advantage of an FPGA-implementation derives from the possibility of configuring all uncommitted logic resources, in a way that as many processing elements as required can be instantiated concurrently in the device. Given the fine granular nature of the reconfigurable resources of an FPGA, reconfigurability comes at the cost of a high amount of silicon resources for the reconfiguration structure (more than 80% [DeH99], [Fen06]), which is a significant overhead, if configuration resources are not used during the operation cycle of the device. However, some FPGA devices allow the use of configuration resources during operation to change the function of parts of the device. This approach is known as run-time reconfiguration (RTR) or dynamic reconfiguration (DR), and can be used to increase the resource utilisation of FPGAs.

A brief literature survey was presented in section 2.3.4, reviewing applications for RTR, and classifying them according to the configuration scheduling. It was shown that there is a growing interest on using RTR. However, although the potential benefits of RTR are known, the use of this technique has not been fully explored in the domain of mechatronic applications [5].

This chapter explores the use of run-time hardware reconfiguration to improve the resource-utilisation of FPGAs for control applications. First, the kind of control algorithms that can benefit from this approach are analysed in section 4.1. Then, hardware reconfiguration of current FPGAs is presented in section 4.2, focusing on FPGAs from Xilinx, explaining how RTR of controllers is implemented using this technology. The resource utilisation when using run-time reconfiguration is then

analysed in the next section, comparing this to a static approach. Section 4.4 presents two implementation examples. The chapter finishes with a brief summary.

4.1 Controller Adjustment

The increasing complexity of modern mechatronic systems makes necessary that controllers can detect internal and external (to the mechatronic system) variations and react accordingly, initiating some kind of control adjustment, e.g., parameter or structure variations. When control adjustments are motivated by large changes in the nominal model of the system, or faults at sensors or actuators, then adjustments are related to the problem of keeping stability or performance. Control adjustment methods under this condition are part of the field of fault tolerant control. A fault can cause a system to malfunction or to operate below the normal performance level. A reduced quality service is the result of a fault. In contrast, a failure prevent the functioning of the system. Both faults and failures can occur at the component level and at the system level. The task of a fault tolerant controller is to prevent a component fault to become a system failure [Bla03]. Two tasks are required to achieve fault tolerance: fault detection and isolation (FDI) and controller adjustments.

There are in literature a number of methodologies concerned with the problem of fault tolerance. These methodologies can be roughly divided in passive and active methods, as depicted in figure 4.1.

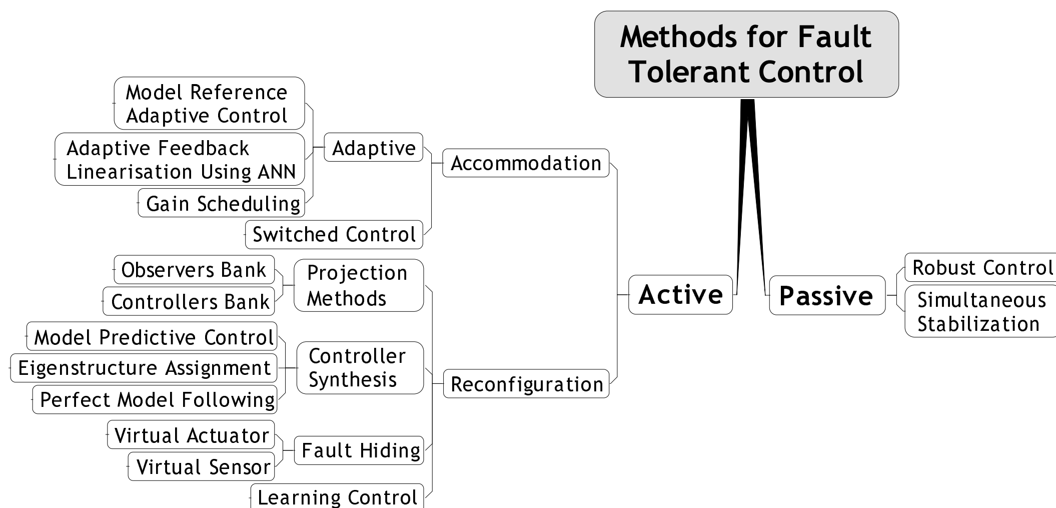


Figure 4.1: Classification of control adjustment methods in presence of faults [Lun06]

Passive methods use robust control theory and related techniques to ensure robust stability of the closed loop in face of changes of the behavior of the plant or faults. A

single controller carries out the control task, limiting the stability region to cases when changes have an incipient effect on the controlled system. Such changes are usually modelled as uncertainty regions around the nominal model [Gev02].

Active methods use direct fault information to perform adjustments and do not assume a static nominal model. Active methods can be further classified in *accommodation* methods and *reconfiguration* methods (cf. figure 4.1).

Adaptive methods perform parametric changes in case of faults, they do not change the original input and output (related to the controller) configuration. Control reconfiguration methods explicitly adjust the structure of the control loop, including I/O configuration, in order to compensate a fault. Figure 4.2 shows a block diagram of a multi-controller system [Mor95]. The measured output y of a process to be controlled drives a bank of controllers, each controller generating a candidate feedback signal u. The control signal applied to the process at each instant of time is then $u \triangleq u_\eta$, where $\eta : [0,\infty) \rightarrow \psi$ is a piecewise-constant switching signal taking values in the family's index set ψ [Lib99]. The switching signal is computed by supervising entity, as presented in figure 4.2.

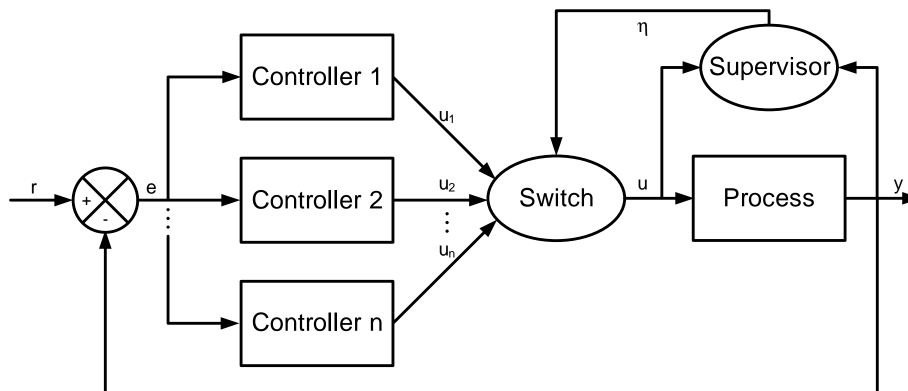


Figure 4.2: Diagram of a supervised multi-control system [Mor95]

If the multi-controller system of figure 4.2 does not require any structural change, then the same system can be achieved by a single controller with adjustable parameters [Mor95], as the one presented in figure 4.3, where the parameters σ are assigned to the controllers structure by a supervising entity.

Controller adjustment can also be motivated by internal and external optimization processes. This scheme is known as self-optimization. Under such varying internal and external optimization goals, a self-optimizing controller is able to optimise its behavior by adapting the structure of used mechanical components, controllers, actuators and/or sensors [Böc06]. The definition of a self-optimising controller is:

"Self-optimization describes the ability of a technical system to endogenously adapt its objective regarding changing influences and thus an autonomous adaption of the

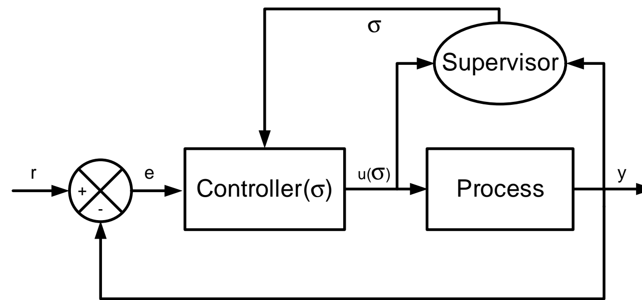


Figure 4.3: Diagram of a supervised adaptive-control system [Mor95]

system behavior in accordance with the objective. The behavior adaption may be implemented by changing the parameters or the structure of the system. Thus self-optimization goes considerably beyond the familiar rule-based and adaptive control strategies; Self-optimization facilitates systems with inherent intelligence that are able to take action and react autonomously and flexibly to changing operating conditions." [SFB11].

Figure 4.4 presents the general structure of Self-Optimizing systems called Operator Controller Module (OCM) [Hes04]. Three well-defined levels can be defined: cognitive operator, reflective operator, and controller. The last level is where control adjustment takes place.

In a static hardware implementation of any of the previously described control schemes, if a structural change is required all possible controllers for a given application have to be mapped into the FPGA concurrently. Accordingly, if parameter adjustments are required, resources have to be allocated to allow new parameters to be loaded and processed. However, FPGAs have a finite amount of reconfigurable resources, which constraints the size and the number of algorithms that can be implemented on a single chip. This implies that the size of the required device does not depend on the amount of configurable logic required for the worst case configuration, but it rather depends on the amount of necessary resources for all possible configurations. A more efficient use of configurable resources can be achieve by using run time reconfiguration. In the next section, run-time reconfiguration and different aspects of it are introduced.

4.2 Run-Time Hardware Reconfiguration

The configuration of the internal logic blocks of an FPGA and their interconnection is defined by a configuration memory. As presented in section 2.2.2, the final phase of the design flow of an FPGA-based system is the definition of a bitstream, which contains the configuration of all uncommitted resources of the device. However, some

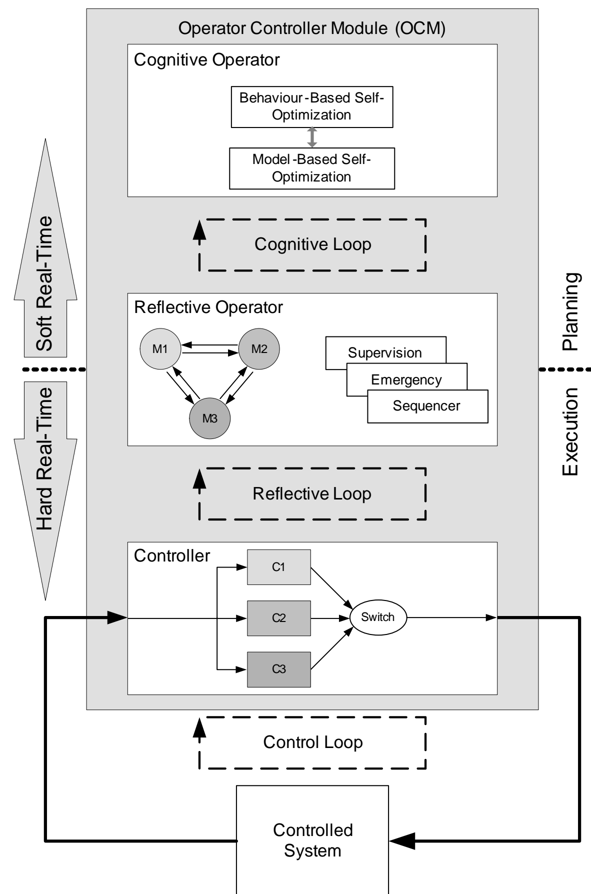


Figure 4.4: Structure of the Operator Controller Module (OCM) [14]

FPGAs allow the configuration of just parts of the device without interrupting the operation of the rest of the device. This approach can become beneficial for some applications, as exemplified in figure 4.5.

A system composed by many sub-tasks (e.g., a multi-controller system) is considered. Figure 4.5 shows different implementation approaches, regarding the temporal use of reconfigurable resources. The system is composed of several tasks, and has different configurations (a), which must be concurrently implemented when using a static implementation approach (b). When using run-time full reconfiguration (c), several full bistreams are produced, with all possible configurations of the design. This leads to the utilisation of a smaller device than in the previous case, but implies a long configuration-time, i.e., full reconfiguration is required each time a new set of modules is to be loaded. When using run-time partial reconfiguration (d) also known as dynamic reconfiguration, the device is divided into Base Region (BR) and Partially Reconfigurable Regions (PRR); the BR (i.e., static area) holds parts of the design that do not change during the operation-time of the system, whereas the reconfigurable

area holds those parts of the design, which are exchanged during the operation of the system. Partial bitstreams are then generated, which are loaded into the reconfigurable area as required. This can lead to the utilisation of a smaller device, in comparison to a static implementation, and to a smaller reconfiguration overhead (i.e., shorter reconfiguration time, and smaller bitstreams to be stored) in comparison to a run-time full reconfiguration.

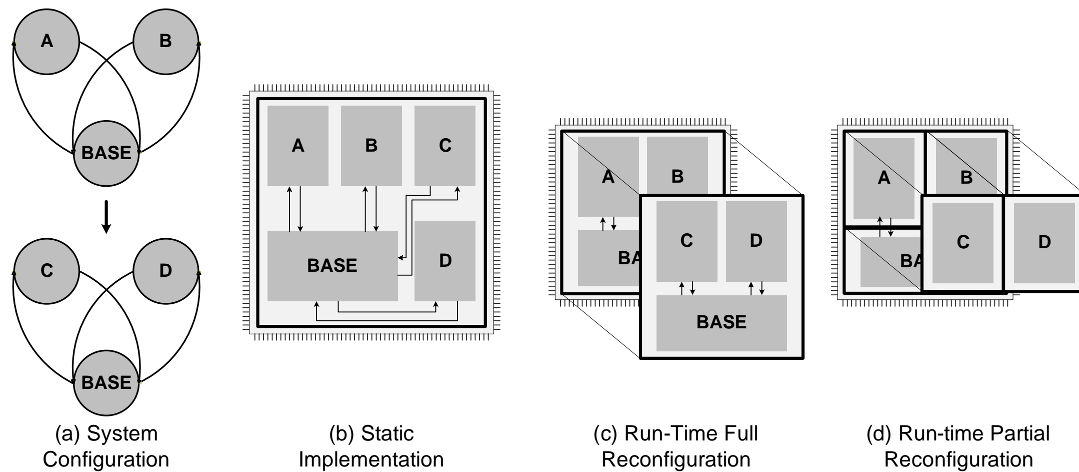


Figure 4.5: Static vs. Run-time full vs. run-time partial hardware reconfiguration

In this chapter we consider only FPGAs from Xilinx, although there are currently other vendors offering run-time partially reconfigurable FPGAs (e.g., the AT40K and AT40KAL series from ATMEL [ATM11], or the ADAPT2000 series from QuickSilver [Qui11]). In the next sections different aspects of the reconfiguration of Xilinx FPGAs are presented.

4.2.1 Configuration Granularity

As presented in section 3.3.2, the function and interconnection of processing elements of an FPGA is determined by the content of its configuration memory. When using partial and dynamic reconfiguration, an important factor is the minimum reconfigurable area of an FPGA, which differs depending on the device. Xilinx supports partial and dynamic reconfiguration for Virtex-II, Virtex-II Pro (only with ISE [1] versions older than 12), Virtex-4, Virtex-5, and Virtex-6 FPGAs. Because these devices have different architectural characteristics, the minimum configuration region changes. In the Virtex-II and Virtex-II Pro the minimum configuration region (configuration frame) corresponds to a full FPGA column. There are different kinds of columns in

[1] Integrated Synthesis Environment [Xil11]

an FPGA, and correspondingly different kinds of configuration frames. For example, the Virtex-II family has the following 6 kinds of configuration frames [Xil07a]:

- IOB Columns: IOB columns configure the voltage standard for the I/Os on the left and right edges of the device.
- IOI Columns: IOI columns configure the IOB registers, multiplexers, and 3-state buffers in the IOBs on the left and right edges of the device
- CLB Columns: The CLB columns program the configurable logic blocks, routine, and most interconnect. IOBs on the top and bottom edges of the device are also programmed by CLB configuration columns.
- BlockRAM Columns: BlockRAM configuration columns program only the BlockRAM user memory space.
- BlockRAM Interconnect Columns: BlockRAM Interconnect columns program all other BlockRAM and multiplier features, including aspect ratios.
- GCLK Column: The global clock column configures most global clock resources, including clock buffers and DCMs.

Each configuration column in an FPGA requires a number of configuration frames to be fully reconfigured, for instance a CLB column of the Virtex-II family requires 22 frames, whereas a BlockRAM column requires 64 frames. Each configuration frame has a fixed width of 32 bits, and a variable length depending on the size of the device, e.g., the length of a configuration frame of a XC2V8000 device is 286, while the corresponding length of a XC2V40 device is 26. This has a direct influence on the achievable reconfiguration time, as explained in the next section.

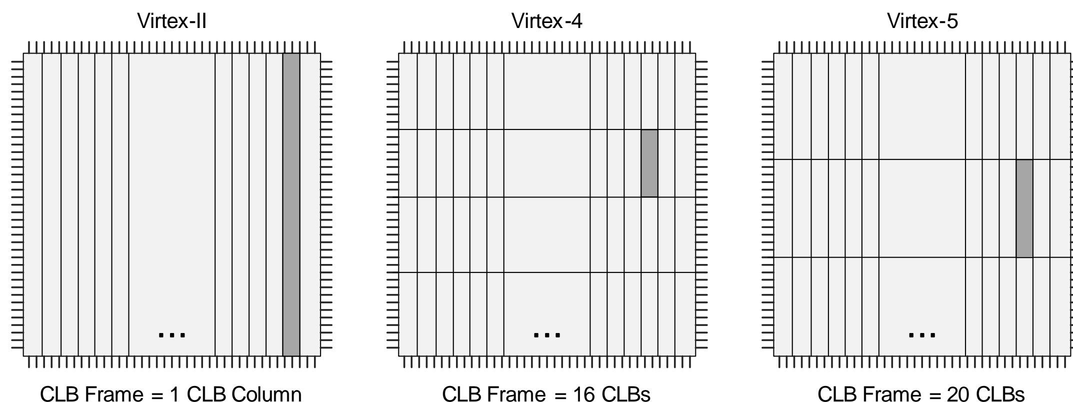


Figure 4.6: Configuration granularity of Xilinx FPGAs

Virtex-4, Virtex-5, and Virtex-6 configuration frames are tiled instead of being arranged vertically in columns as in Virtex-II and Virtex-II Pro FPGAs. The configuration frame of a Virtex-4 is 16 CLBs high and 1 CLB wide, whereas the configuration frame of a Virtex-5 is 20 CLBs high and 1 CLB wide, as shown in figure 4.6. The

width and length of the configuration frames for the Virtex-4, Virtex-5, and Virtex-6 are fixed to 32×41, 32×41, and 32×81 bits correspondingly, which translates for instance to 16, 20, and 40 CLBs correspondingly. Defining the reconfigurable area so that it fits on configuration frame boundaries reduces the number of configuration frames that must be reconfigured, resulting in smaller partial bitstreams and faster reconfiguration times.

4.2.2 Configuration Interface

Arrangement of all configurable components of an FPGA is done by loading a bit-stream on to a configuration memory. To access this configuration memory, most of the Xilinx devices offer parallel and serial configuration interfaces (cf. figure 4.7). Two serial interfaces are available; a JTAG (Joint Test Action Group) interface, and a Master/Slave serial interface. Furthermore, Master/Slave selectMAP and ICAP (Internal Configuration Access Port) offer a 8 to 32 bit-width interface to the configuration memory. ICAP interface (available in most of FPGA devices), is only used for partial reconfiguration. Through these interfaces, external- and self-reconfiguration can be realised.

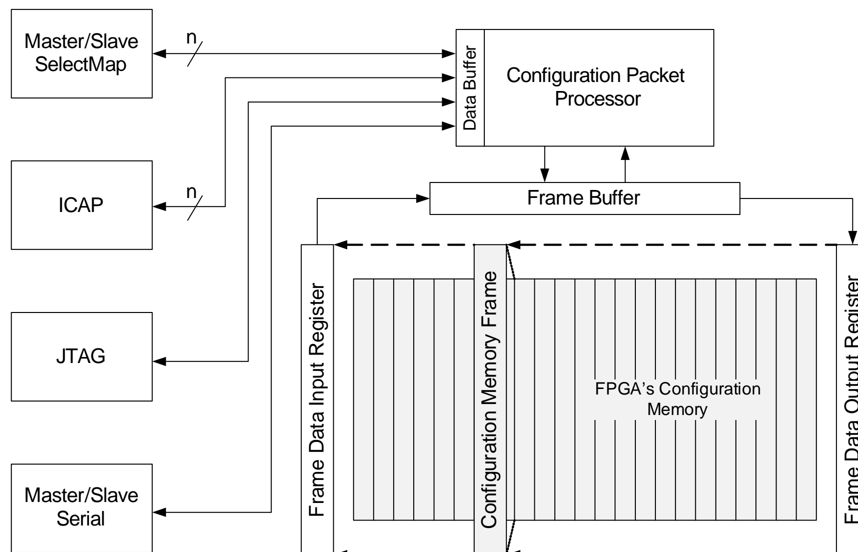


Figure 4.7: Configuration Interface of the Virtex-II family [Xil07a],[Ket08]

All configuration interfaces are connected to the packet processor (cf. figure 4.7), which controls data flow from the configuration interface (i.e., SelectMAP, JTAG, or Serial) to the configuration memory cells.

The speed of the reconfiguration depends on the size of the device, i.e., the amount of required configuration bits (B_{size}), the used configuration interface (parallel or

serial), which determines the number of transferred bit per cycle (*BPC*), and the configuration clock frequency of the packet processor (T_{freq}). The reconfiguration time t_{conf} is given by equation 4.1.

$$t_{conf} = \frac{B_{size}}{BPC \cdot T_{freq}} \tag{4.1}$$

For the Virtex-II and Virtex-II Pro families the maximum transfer frequency is 50 MHz (for continuous transfer), and $W = 8$ (for ICAP and selectMAP interfaces). Newer FPGAs such as Virtex-4, Virtex-5, and Virtex-6 support transfer frequencies up to 100 MHz, and $W = 32$ bits/cycle. Table 4.1 shows the configuration bandwidth for all configuration ports of Virtex devices.

Conf. Mode	Max. T_{freq}	*BPC*	Max. Bandwidth
ICAP	100 MHz[1]/50 MHz[2]	32 bits[1]/8 bits[2]	3.2 Gbps[1] /1.6 Gbps[2]
SelectMAP	100 MHz[1]/50 MHz[2]	32 bits[1]/8 bits[2]	3.2 Gbps[1] /1.6 Gbps[2]
Serial Mode	100 MHz[1]/50 MHz[2]	1 bit	100 Mbps[1] / 50 Mbps[2]
JTAG	66 MHz[1]/33 MHz[2]	1 bit	66 Mbps[1] / 33 Mbps[2]

Table 4.1: Configuration bandwidth for configuration ports in Virtex architectures [Xil10]([1] Virtex-4, Virtex-5 and Virtex-6. [2] Virtex-II and Virtex-II PRO)

4.2.3 Partial Reconfiguration Process

To realise partial reconfiguration, a full configuration of the device is first required. When partial reconfiguration is controlled internally (i.e., self-reconfiguration), a configuration controller has to be instantiated on the static area of the initial full configuration. Later on, partial bitstreams can be loaded from an external memory onto the FPGA, through the configuration controller. The main tasks of the configuration controller, the initial reconfiguration process, and process of loading partial bitstreams are presented in the following sections.

Configuration Controller

Xilinx FPGAs offer different configuration ports (cf. figure 4.7), as well as different configuration modes (i.e., master/slave serial or SelectMAP, and JTAG). In order to perform partial reconfiguration, a configuration controller is required. There are two basic approaches: the configuration controller is instantiated inside the FPGA, or the configuration process is controlled by an external device (cf. figure 4.8). In both cases, external memory is required to store partial and initial full-configuration bitstreams.

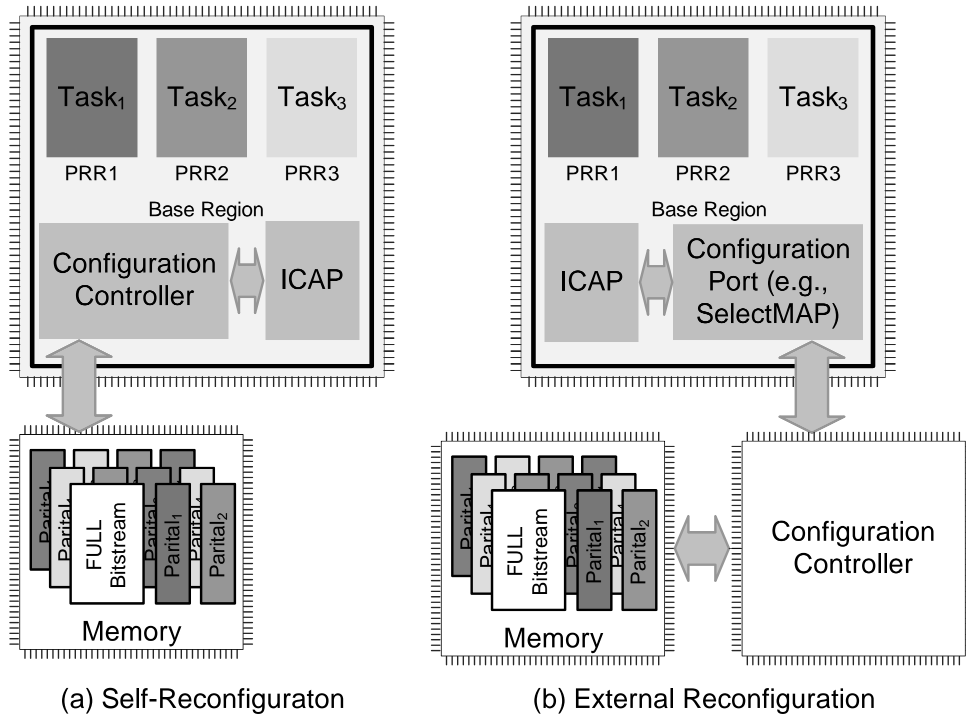


Figure 4.8: Self- and external reconfiguration schemes

Self-reconfiguration is done by directly controlling the ICAP port of the FPGA. After initial full configuration, the configuration controller fetches partial bitstreams from an external memory and feeds them to the ICAP port. The configuration controller has to be placed on a static area, and can be realised as state-machine or an embedded soft-core processor.

External reconfiguration is controlled by an off-chip device. Configuration data is fetched from an external memory and sent to a configuration port (e.g., SelectMAP or JTAG). The advantage of this method is that the same device which performs full configuration can be used to control partial reconfiguration, without using internal FPGA resources.

The realisation examples presented in this work, use a self reconfiguration scheme, enabled by the configuration controller presented in [Hag07b] and [Hag05], which is based on the RAPTOR system (cf. section 4.4.1).

Initial Full Configuration

The steps for the initial configuration of the FPGA are listed in figure 4.9. After the device has been powered up (1), the next step is to serially clear the configuration memory (2), to bring it to a defined state. In order to determine the selected

configuration mode, the device samples configuration mode pins (3), and starts the selected configuration sequence. The bitstream loading process (4-7) is similar for all configuration modes; the primary difference between modes is the selected interface. The next step (until CRC Check) involves the content of a bitstream, which is typically generated using a synthesis software as explained in section 2.2.2.

Before the configuration data frames can be loaded, a special 32-bit synchronization word must be sent to the configuration logic (4). The synchronisation word alerts the FPGA to upcoming configuration data and aligns the configuration data with the internal configuration logic. Any data on the configuration input pins prior to synchronisation is ignored. The synchronisation word is automatically included in the bitstream by synthesis tools.

Once the FPGA is synchronised, a device ID check must pass before the configuration data frames can be loaded (5). This prevents an attempted configuration with a bitstream that is formatted for a different device. Thereafter, configuration data frames are loaded (6). As the configuration data frames are loaded, the device calculates a Cyclic Redundancy Check (CRC) value from the configuration data packets (7), which can be used upon request to compare the CRC of the loaded configuration frames with an expected CRC value. If a CRC error occurs during configuration, the device must be resynchronised and reconfigured. Upon successful CRC verification, the device can go on with a start-up sequence (8), where the device is setup for operation (e.g., enabling I/Os, or Asserting Global Write Enable, which allows RAMs and flip-flops to change state). This steps are the same as for an static implementation.

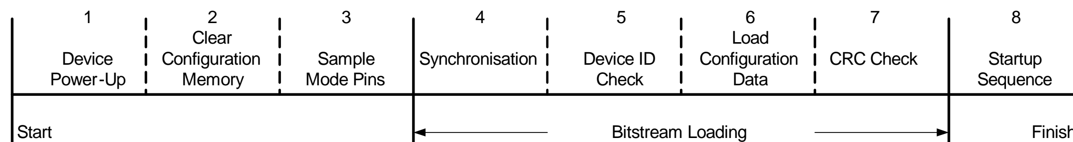


Figure 4.9: Virtex-4 Configuration Process [Xil09b]

Loading Partial Bitstreams

After initial device configuration with a full bitstream, partial bitstreams can be loaded into the FPGA at run-time. Any of the configuration ports can be used to load the partial bitstream: SelectMap, Serial, JTAG, or ICAP. For partial reconfiguration only steps 4 to 7 from the initial configuration (cf. figure 4.7) are executed (except for some commands, such as asserting global reset). Because the partial bit file contains mainly frame address and configuration data, plus a final checksum value, the configuration controller has to monitor the data being sent to detect when configuration has finished. This can be detected by synchronisation word written at the end of each partial bitstream.

4.2.4 Partition and Placement Approaches

As mentioned in section 4.2.1, the configuration granularity of Xilinx FPGAs varies according to the device, e.g., from a whole column to some CLBs of a column (cf. figure 4.6). When designing a dynamically reconfigurable system, the configuration granularity of Xilinx devices allows for different partition and placement strategies. To better explain the concept of partition and placement some definitions are first required [Hag07b].

Base Region (BR): The base region is the area of the FPGA, which is configured at the initialization of the system and does not changed at run-time. All static components of the system are located in the base region.

Partially Reconfigurable Region (PRR): This is the FPGA region used for run-time reconfiguration. All components, which shall be changed during operation are located in a PR Region. Depending on the application requirements, one or several PR Regions can compose a partially reconfigurable system.

Reconfigurable Tile (RT): A reconfigurable tile is the smallest partially reconfigurable unit, which can be larger than a single reconfiguration frame (cf. section 4.2.1). A PR Region can be composed of one or more individually reconfigurable tiles.

Partial Reconfiguration Module (PRM): A PR Module represents the implementation of a dynamic system component (e.g., one controller of a multi-controller system). It can be placed and removed at run-time according to the needs of the application.

A PR Module can be constituted by a single tile that covers the whole area of the PR Region. Consequently, PRMs can only consist of a single tile, which constraints the maximum number of modules that can be placed and executed at the same time to the total number of PR Regions. The disadvantage of this approach is that even small modules occupy a complete PR Region (cf. figure 4.10 (a)). This approach is known as fixed-size slots placement [Hag06]. To avoid the limitations of fixed-slots placement, the PR Region can be partitioned into multiple tiles as shown in figure 4.10 (b). A PR module does not have to composed of a single tile, but of a set of contiguous tiles. This approach can lead to a efficient use of reconfigurable resources, because the size of the area occupied by a module corresponds more to the actual size of the module than when using a fixed-size slot approach. This flexibility comes to the price of a higher communication complexity, and thus resources overhead. This topic is presented in section 4.2.5.

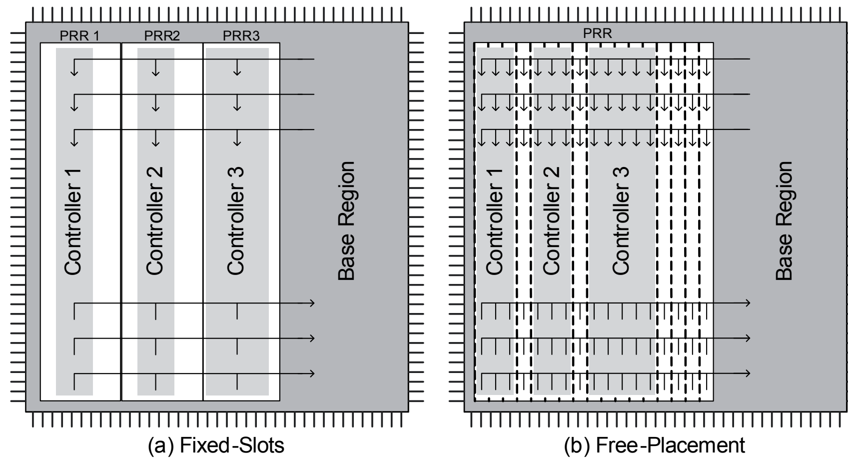


Figure 4.10: Placement methods for 1D approach

Fixed-size slots and free-placement can be realised either using a 1D or a 2D partition approaches, which deal with the size, number and arrangement of tiles within a PR Region, as described in the following paragraphs [Hag07b].

1D-Partitioning: In a 1D-approach the height of the tiles is equal to the height of the PR Region, as shown in figure 4.11(a), using a fixed-size slot approach.

Multi-1D-Partitioning: In a 1D placement, the PR Region can be subdivided in a way that a tile does not have the same height as the PR Region, as shown in figure 4.11(b). This can be beneficial to avoid internal routing problems when implementing a PR Module. The multi-1D-partitioning is especially suitable for FPGAs with column-based partial reconfigurability, such as the Virtex-4 , Virtex-5 and Virtex-6 devices, where the smallest partially reconfigurable unit is a configuration frame, which is just a fraction of a column (cf. section 4.2.1).

2D-Partitioning: In the 2D-partitioning the PR Region is partitioned into horizontally and vertically arranged tiles, as shown in figure 4.11(c). A module is represented by a group of tiles arranged in a rectangular shape. In this partition approach, the height of a module is no longer restricted to the height of the PR Region. When generating a module the area and the aspect ratio can be optimized according to the internal routing of the module.

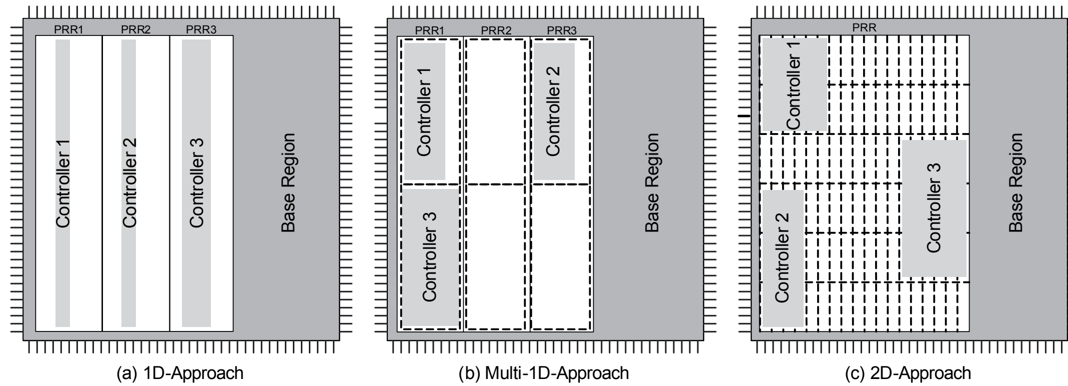


Figure 4.11: 1D vs. 2D placement methods

4.2.5 Communication Infrastructure

An important aspect of designing dynamically reconfigurable systems is the communication between the Base and PR Regions. When partial reconfiguration is done, communication channels have to remain configured to guarantee data flow among modules.

To solve this problem, Xilinx offers Bus Macros, which are a set of locked connection links placed at the border between PR and Base regions, and provide a pseudo-static communication channel between this two regions [Xil10], cf. figure 4.12 (a). Bus macros are well suited for simple communication schemes, where direct point-to-point connections are required for dedicated signals.

For more sophisticated communication schemes, where communication lines are shared among many modules, embedded macros are a better option [Hag07b]. Embedded macros reserve routing resources for inter-modular communication, providing an homogeneous communication infrastructure, cf. figure 4.12(b). Embedded macros can be implemented based on tristate-buffers, or based on slices [Hag07a]. The choice between one of these options depends on the width of the tiles within a PR Region, the required communication bandwidth, and the available resources. A slice-based embedded macro offers a higher bandwidths for tiles expanding less that 20 CLBs [Hag07b].

Figure 4.12 shows a Bus Macro (a) and a Embedded Macro realisations of communication channels for a reconfigurable system with three PR Regions using Fixed-Size Slot placement. In the next section, the resource utilisation of a static implementation approach in comparison to that of a RTR-based system is analysed.

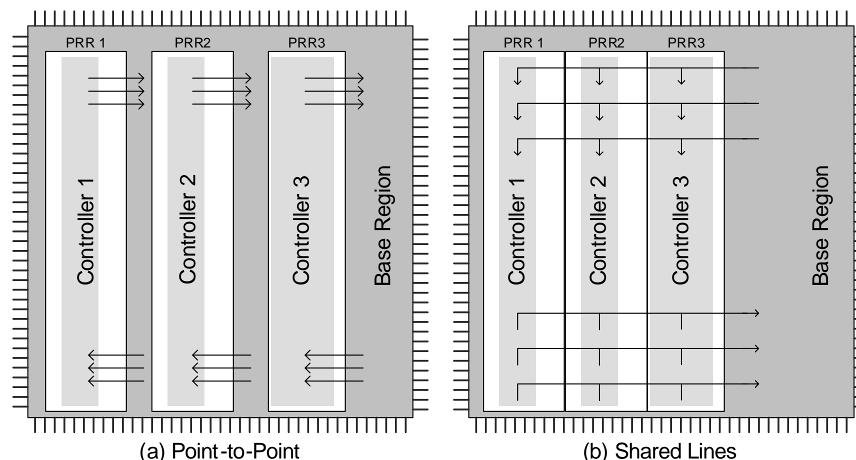


Figure 4.12: Point-to-Point (Bus Macros) vs Shared Lines (Embedded Macros) communication

4.3 Control Adjustment Through Run-Time Reconfiguration

As presented in section 4.1, there are different approaches to control adjustment. The benefit of using run-time reconfiguration to realise structural or parametric control adjustments can be analysed depending on the kind of adjustment. We distinguish two cases:

1. A system is controlled by a set of algorithms with different structures, where only one structure is active at any given time.
2. A system is controlled by a set of algorithms with a single structure, whose parameters are to be changed in run-time

To the first case belong active adjustment approaches such as projection methods, switched control, or fault hiding (cf. section 4.1). To the second case belong adaptive control approaches such as gain scheduling (cf. figure 4.1). When implementing these control-adjustments using reconfigurable hardware, run-time reconfiguration can be used to improve resource utilisation. The resulting resource allocation when using RTR depends on the control period, the reconfiguration time, and the placement approach, as presented in the next section.

4.3.1 Structure Adaptation

Let $C = \{c_1, c_2, ..., c_n\}$ be a set of n controllers with different structures, used to control a given plant. A controller $c_i \in C$ occupies a set of $s_i \in S$ FPGA resources

(e.g., slices, BRAMs, embedded multipliers). Thus the set of resource requirements for all n controllers $\in C$ is given by $S = \{s_1, s_2, ..., s_n\}$. Furthermore, each controller $c_i \in C$ requires initialization resources u_i when it is activated, which is given by $U = \{u_1, u_2, ..., u_n\}$. Besides the controllers, additional hardware resources A_{const} might be required, e.g., for transmitting and processing data from and to the I/Os. Furthermore, each controller $c_i \in C$ has a reaction time tr_i, which is the time required by a controller to become operative. For controllers that are reconfigured in run-time, reaction time is given by equation 4.2.

$$tr_i = tconf_i + init_i \tag{4.2}$$

where $tconf_i$ is the reconfiguration time required for the ith controller, and $init_i$ is its initialization time. For a static implementation tr_i is neglected, because all required controllers are instantiated concurrently. The resource requirements of a static implementation is thus given by equation 4.3.

$$A_{static} = A_{const} + \sum_{i=1}^{n} (s_i + u_i) \tag{4.3}$$

If run-time reconfiguration is used, there are four possible resource-utilisation scenarios, depending on the reaction time of the controller and the placement approach (cf. section 4.2.4). If tr_i is longer than the control period of the ith controller (p_i), the to-be-replaced controller has to stay configured until the new one has been loaded and initialised. However, if $tr_i < p_i$, then the reconfigurable area of the old controller can be used for the new one, or for its initialization routine (u_i). If the reconfigurable system architecture is build using a fixed-size slot approach, the size of all PR Regions (an thus the size of the partial bitstreams of all controllers) is defined by the amount of resources used by the largest controller. In fact, the required area depends on the amount of configuration frames that suffice to realise the largest controller. This area can be larger than the area requirements of the controller itself. Therefore, the worse case resource utilisation is given by equation 4.4.

$$A_{dynamic_FS} = \begin{cases} 2 \cdot s_{max1} + A_{const} + A_{rec}, & \text{if } tr_{max1} < \ p_{max1}; \\ 3 \cdot s_{max1} + A_{const} + A_{rec}, & \text{if } tr_{max1} > \ p_{max1}; \end{cases} \tag{4.4}$$

Where s_{max1} is the area requirements of the largest controller $c_{max1} \in C$. Correspondingly, tr_{max1} is the reaction time of that controller, and p_{max1} is the control period. Furthermore, A_{rec} represents the resources required to realise partial reconfiguration, e.g., for the communication infrastructure, and for the reconfiguration controller. Equation 4.4 is exemplified in figure 4.13.

If a free placement approach is used, then the required reconfigurable area depends on the size of each controller. Here again, the minimal amount of configuration frames

is what defines the size of each controller . Thus, the worse case resource utilisation is given by equation 4.5.

$$A_{dynamic_FP} = \begin{cases} s_{max1} + u_{max1} + A_{const} + A_{rec}, & \text{if } tr_{max1} < p_{max1}; \\ s_{max1} + s_{max2} + u_{max1} + A_{const} + A_{rec}, & \text{if } tr_{max1} > p_{max1}; \end{cases} \quad (4.5)$$

Where s_{max2} is the area requirements of the second largest controller $c_{max2} \in C$. Correspondingly, tr_{max2} is the reaction time of that controller, and p_{max2} is the control period.

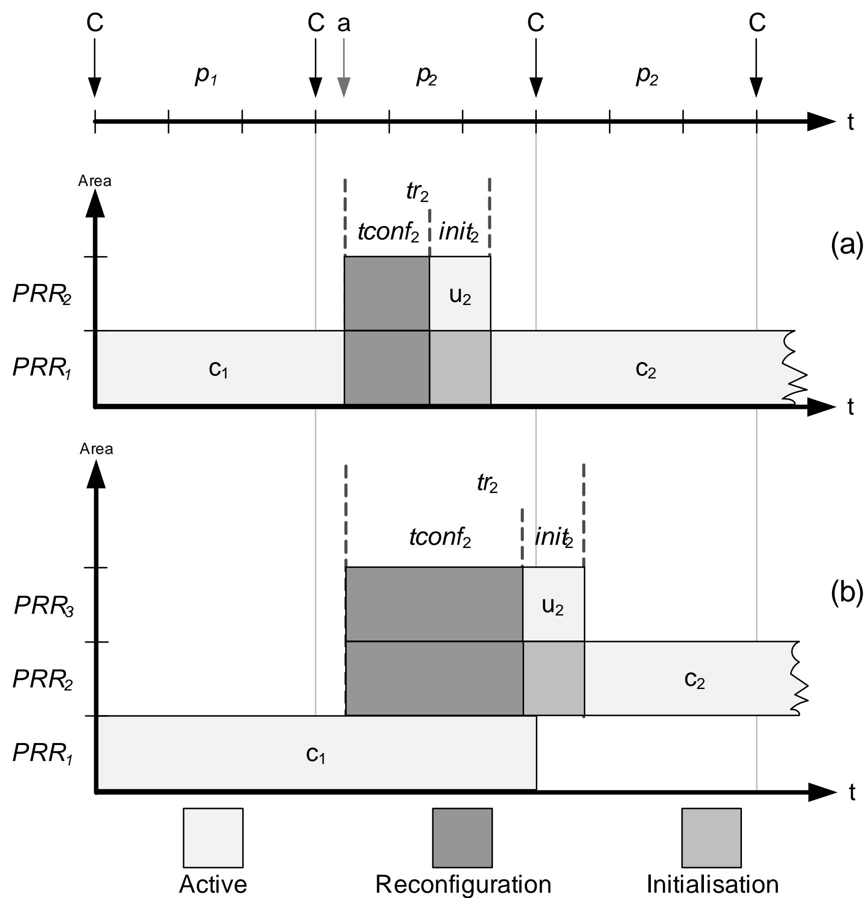


Figure 4.13: Resource utilisation as described in both cases of equation 4.4

Both cases of equation 4.4 are exemplified in figure 4.13. The first axis represents time, showing time events. The second and third axis represent the area utilisation of the PR Regions through time. In the first axis C represents the time where a control output is generated, p_1 and p_2 are the control periods of controllers c_1 and c_2, respectively. Finally, a represents a reconfiguration request. In figure 4.13(a)

the reaction time tr_2 of the new controller is shorter than the control period $p_1 = p_2$. Because a fixed-size slot approach is used, the worst-case resource utilisation is given by $A_{dynamic_FS} = PRR_1 + PRR_2 + A_{const} + A_{rec}$. In figure 4.13(b) the reaction time tr_2 of the new controller is longer than the control period $p_1 = p_2$. Therefore, the worse case resource utilisation is given by $A_{dynamic_FS} = PRR_1 + PRR_2 + PRR_3 + A_{const} + A_{rec}$. From this example, it can be observed that the reaction time of a controller has a great influence on the worst case utilisation scenario when using RTR. A dynamic implementation leads to a better resource utilisation than a static approach (e.g., RTR enables to use a small FPGA device in contrast to a static approach), if the set C is large enough, so that $A_{static} > A_{dynamic_FS}$, or $A_{static} > A_{dynamic_FP}$.

4.3.2 Parameter Adaptation

When one single control structure is required, whose parameters are adapted, two realisations are possible: a general structure, where parameters can be loaded as required, and several specific structures, which can be exchanged using dynamic reconfiguration. A specific implementation requires less resources, because hardware dedicated to load new parameters are no longer used. Furthermore, the difference regarding resource-requirements between a specific and a general implementation increases when considering the case of a structure, whose parameter sets have zero components (e.g., spare matrixes). In those cases the zero components would not be realised in a specific implementation. Moreover, in a specific realisation, the word-length can be optimised to each parameter set.

Let us take a vector multiplication as example. Figure 4.14 shows the implementation of a vector multiplier as a parameterisable structure (a), and as a constant multiplier (b).

The resource utilisation of both structures when increasing the number of inputs, and using 8 bits as a base word-length is presented in figure 4.15. It can be noticed, that a specific implementation requires much less resources than a general structure. Using dynamic reconfiguration, a specific implementations can be exchange in run time according to the requirements, avoiding the use of a general structure.

This approach is suited for control structures such as state-feedback regulators, or state-observers (cf. section 3.5.2 and 3.5.3 respectively), which have typically a large number of parameters to be changed in case of a control adaptation.

To analyse the resource utilisation, let us first define $PAR = \{par_1, par_2, ..., par_n\}$ as the set of n parameter-groups for a control structure. When implementing a general structure, i.e., allowing parameter changes (cf. figure 4.14(a)), the resource requirements, B_{static}, are given by equation 4.6.

$$B_{static} = B_{general} + B_{const} \tag{4.6}$$

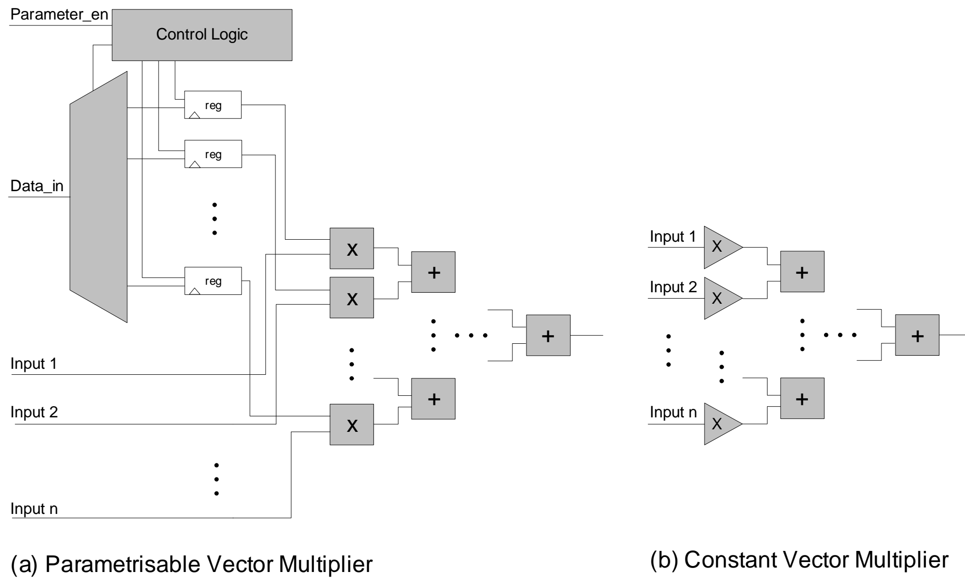


Figure 4.14: Schematic of a vector multiplier: specific implementation vs. general implementation

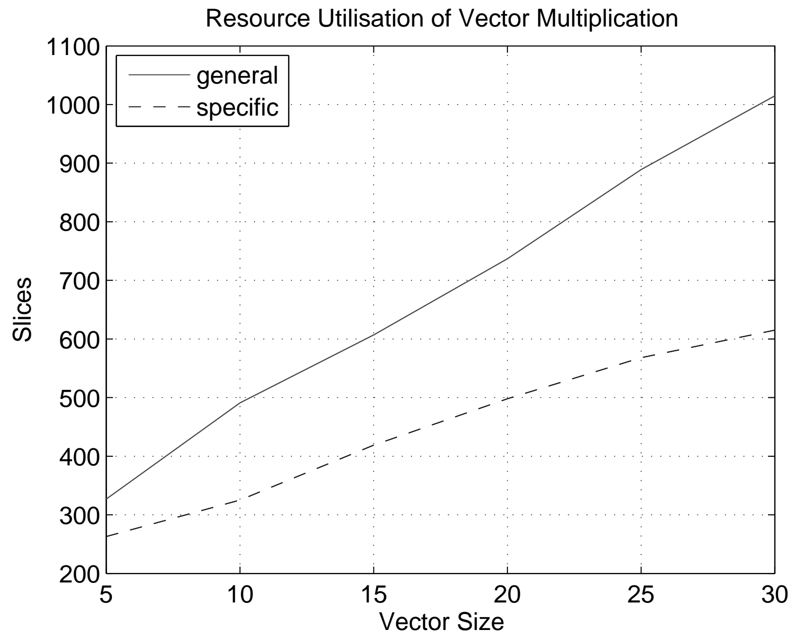


Figure 4.15: Resource utilisation of vector multiplier: specific implementation vs. general implementation based on a Virtex II FPGA (XCV4000)

where $B_{general}$ represents hardware resources required for a parameterisable structure, and B_{const} represents additional hardware resources, required for instance to

process I/O signals. In the case of a constant parameters implementation the resource requirements is given by $A = \{\alpha_1, \alpha_2, ..., \alpha_n\}$, for all $par_i \in PAR$. Furthermore, each specific implementation requires initialization resources γ_i when it is activated, which is given by $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$. To allow dynamic reconfiguration of the specific controllers, dedicated resources are required, symbolised by B_{rec}. The reaction time of the ith specific realisation α_i is given by equation 4.7.

$$tr_s_i = tconf_s_i + init_s_i \tag{4.7}$$

where $tconf_s_i$ is the reconfiguration time of the ith specific realisation, and $init_s_i$ is its initialization time. In this case there are again four possible cases of resource utilisation scenarios. If the reconfigurable system architecture is build using a fixed-size slot approach (cf. section 4.2.4), the size of all fixed slots is defined by the amount of resources used by the largest specific implementation α_{max1}. Thus the worse case resource utilisation is given by equation 4.8.

$$B_{dynamic_FS} = \begin{cases} 2 \cdot \alpha_{max1} + B_{const} + B_{rec}, & \text{if } tr_s_{max1} < p_s_{max1}; \\ 3 \cdot \alpha_{max1} + B_{const} + B_{rec}, & \text{if } tr_s_{max1} > p_s_{max1}; \end{cases} \tag{4.8}$$

where tr_s_{max1} is the reaction time of the largest specific realisation (cf. equation 4.7), and p_s_{max1} is its control period. If a free placement approach is used, the worst-case resource utilisation is given by equation 4.9.

$$B_{dynamic_FP} = \begin{cases} \alpha_{max1} + \gamma_{max1} + B_{const} + B_{rec}, & \text{if } tr_s_{max1} < p_s_{max1}; \\ \alpha_{max1} + \alpha_{max2} + \gamma_{max1} + B_{const} + B_{rec}, & \text{if } tr_s_{max1} > p_s_{max1}; \end{cases} \tag{4.9}$$

Where γ_{max1} is the FPGA area required for the initialization routine of the largest specific controller, and α_{max2} is the area required for the second largest specific controller.

A dynamic implementation leads to a better resource utilisation than a static realisation when the number of parameters, which have to be exchange is sufficiently large (cf. figure 4.15). This means that the resource required to realise a general control structure have to be such that $B_{static} > B_{dynamic_FS}$ or $B_{static} > B_{dynamic_FP}$.

4.4 Implementation Examples

In this section implementation examples are presented. First, the underlying prototyping hardware platform, the RAPTOR system, is introduced. Next, a system on chip

architecture is presented [15], which has been used to realise the application examples. After this, two case studies are shown, and finally a brief summary and discussion are given at the end of the chapter.

4.4.1 The RAPTOR System

The RAPTOR System has been developed in the system and circuit technology group [Por09]. It is a modular PCI-based board, consisting of a base system and a variety of extension modules. The RAPTOR system allows inter-module communication through direct connections, or through two bus systems: the Local Bus for control flow and the Broadcast Bus for data flow (cf. 4.16). The RAPTOR system allows the communication of any module to the host PC through a PCI bus bridge, as shown in figure 4.16.

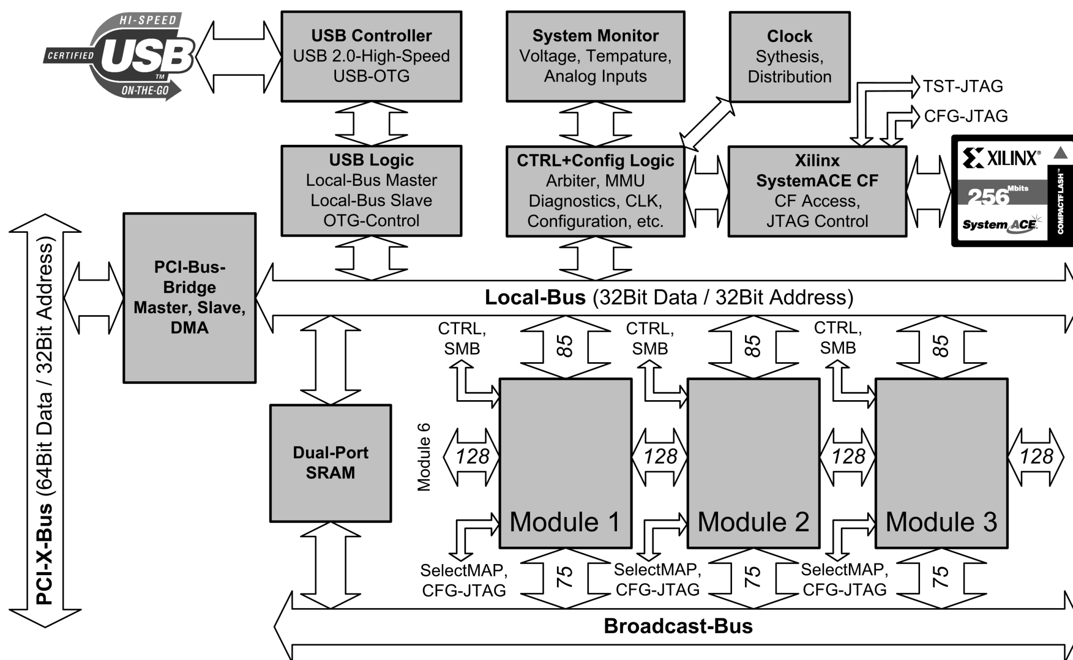


Figure 4.16: Block diagram of the RAPTOR64 system [Por09]

Communication management and configuration control are also provided in the RAPTOR system. Furthermore, this features can be controlled through a software library, called RaptorLIB, which offers a direct interface to the hardware. The RaptorLIB provides the required functions to manage communication to the modules from the host computer, FPGA configuration, clock management, and monitoring, among others. This software library is the base of the HiL frameworks described in chapter 5, but briefly introduced in the next sections.

4.4.2 System Architecture

A system-on-chip architecture based on the RAPTOR prototyping system has been implemented. The architecture is implemented on a daughter board of the RAPTOR system, which is based on a Virtex-II Pro FPGA from Xilinx (XC2VP30). The architecture is composed of an embedded PowerPC processor (PPC) connected to dynamic reconfigurable resources (PR Region 1 to 4), and to further hardware components described in this section (cf. figure 4.17). The PPC runs at a clock frequency of 300 MHz, whereas the PR Regions have a clock of 30 MHz. A processor local bus (PLB) allows communication to the local bus (LB) of the RAPTOR system, and from there to the host PC, through a Peripheral Component Interconnect (PCI) bus, as depicted in figure 4.17.

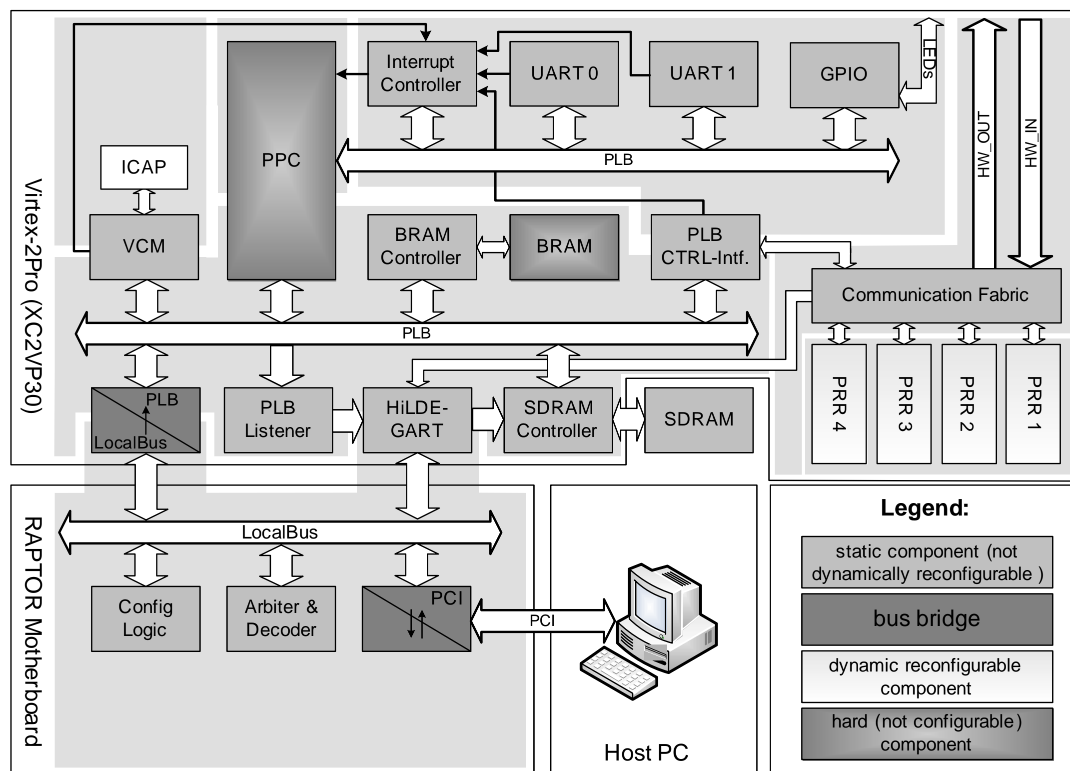


Figure 4.17: Schematic of the system architecture implemented on the RAPTOR system [15]

There are four PR Regions, which are realised as fixed-size slots of equal size. These slots are used to implement controllers or signal conditioning blocks, since these elements are exchanged according to the current state of the plant and the current objective of the system.

The reconfiguration of any of the PR Regions is carried out by the Virtex Configuration Manager (VCM) [Hag07b], which uses a clock frequency of 50 MHz for the reconfiguration (continuous transfer to the ICAP port). The reconfiguration of any of the PR Regions lasts about 4,38 ms. A program running on the PPC can initiate the reconfiguration, indicating the memory space from the external SDRAM where partial bitstreams can be fetched (cf. figure 4.17). The destination PR Region is embedded in the partial bitstream. When a reconfiguration is requested, the VCM initiates DMA (Direct Memory Access) transfers from the SDRAM controller, loads the desired partial bitstream to the target PR Region by accessing the ICAP, and sends an interrupt to the PPC when the configuration is done. A supervising program, running in the PPC, is in charge of monitoring system activity and triggering the reconfiguration of any of the PR Regions.

The architecture incorporates a flexible communication system, enabling data transmission between static and dynamic components (e.g., between PR Regions and the PPC), as well as between internal components and external components (e.g., between controllers and the plant). All PR Regions are connected through a multiplexer to the external components, a select signal controlled by the PPC defines one of the PR Regions as current output, which can be changed as required at run-time. Furthermore, each PR Region has four 16-bit I/O ports (called Crosspoint ports, cf. figure 4.18), and up to 64×16-bit wide I/O ports connected to the Channel Bus. Crosspoint ports are suited for data-streaming between PR Regions (e.g., initialization of a new-loaded controller from another PR Region). Channel Bus communication is slower, and best suited for parameter exchange.

The communication fabric is highly configurable, allowing the PPC, to set the source for the 16-bit Crosspoint ports, Channel Bus and the outputs (HW_OUT, cf. figure 4.18). Any of the PR Regions or the PPC can be the source of any of the Crosspoint ports of the PR Regions; this configuration can be changed at run-time. For system monitoring, the PPC has access to all values and Channel Bus vectors. Furthermore, as the PPC has access to all signals of the system, its also possible to implement a controller completely in software without utilising some of the PR Regions. This feature is specially important, because based on this flexibility a controller running in the PPC can be exchanged by an FPGA-based controller, placed in any of the PR Regions, which can be initialised by a dedicated design placed in any other PR Region or even from the PPC. This feature is used in the example presented in section 4.4.4.

Data traffic is synchronised by the shadowing interface. Data from the Crosspoint ports, from the Channel Bus, and from external I/Os is first copied to the shadowing logic block, which upon a trigger signal releases it to the other components.

Triggering signals are available also to each PR Region. These signals can be generated from any of the PR Regions, from an internal trigger generator, or from an

external signal. The PPC has access to all values in the shadowing interface, after and before it has been triggered.

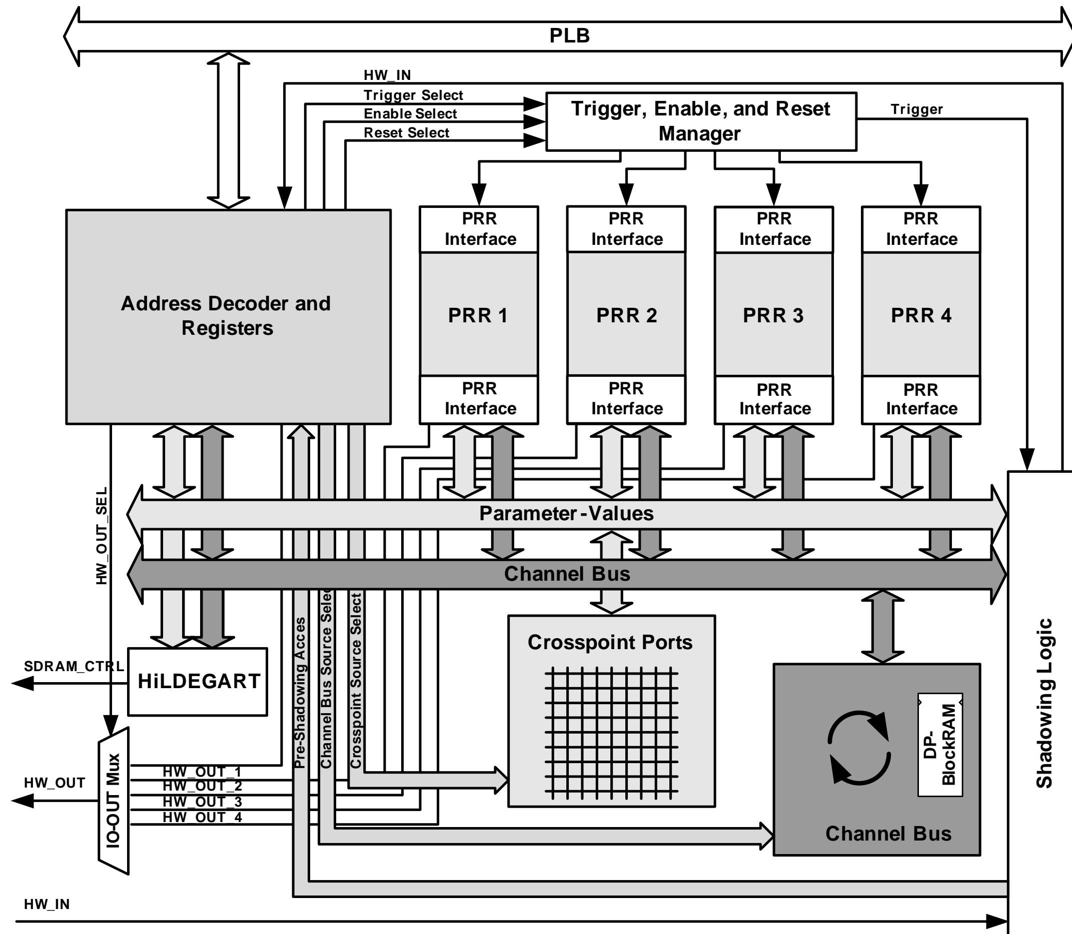


Figure 4.18: Simplified schematic of the communication fabric [15]

The architecture includes a HiLDEGART component for high-speed monitoring. HiLDEGART is part of an on-line monitoring framework introduced in section 5.3. It records all parameter and value ports from the communication fabric (cf. figure 4.18), and sends a low-frequency sampled signal to the GUI (sampling frequency of about 125 Hz) through the Local Bus of the RAPTOR system. High-frequency sampled data is send to the SDRAM. The sampling rate of each I/O port can be set individually. Displaying of I/O values and all configuration tasks are controlled via a GUI from the host PC.

Dynamic reconfiguration is constantly monitored by a PLB-listener [Gra09]. The PLB-listener waits for the VCM master request to the SDRAM. When the request is performed by the VCM, the listener is activated and waits for valid data from the SDRAM controller. All the data sampled by the PLB-listener is transferred to

a external application for parsing and visualisation, allowing visualisation of the reconfiguration using a dedicated GUI.

A UART interface allows serial communication to the host computer from a running application on the PPC. This communication interface is well suited for debugging and low-speed monitoring purposes. Furthermore, an LED matrix can be accessed from the PPC through a General Purpose Input/Output (GPIO) interface, which is used to signalise internal states of the architecture (e.g., PPC initialization completed).

This system architecture serves as the platform for the case-studies presented in the following sections. It is not meant to be an application-specific architecture, but it is rather designed to allow a wide variety of experiments. The utilisation of RTR to implement the control system of an inverted pendulum is presented in the next section.

4.4.3 Inverted Pendulum System

This section presents the implementation of the control system for an inverted pendulum. It is a proof-of-concept example, which shows the principles of using run-time-reconfiguration for control applications [1, 2].

The inverted pendulum is a classical problem in control theory; it has been used in literature as an example of a well-understood yet non-trivial system to test control algorithms [Oga87]. There are many variations of the problem, however we focus on the basic approach, in which the cart can only move in the x coordinate, and the pendulum has only one degree of freedom.

Test Bed Description

A diagram of the inverted pendulum is shown in figure 4.19. There is one actuator and two sensors; one for position detection and another for angle detection. The actuator is an AC servo motor (MSM030C-0300-NN-M0-CG0 from Rexroth), with a maximum torque of 3.8 Nm, a maximum speed of 5000 RPM, and 400 W of peak power [Rex04]. The motor has an attached absolute encoder (e_1) with a resolution of 131072 increments/revolution. The encoder is connected to the power drive trough a serial interface. The other encoder (e_2) is attached to the pendulum, it is an incremental rotary encoder (Heidenhain ROD 420), with a resolution of 5000 inc/rev, and is used to detect the angle of the pendulum. Both sensor signals are sent to the RAPTOR system for further processing (cf. figure 4.20).

The cart of the pendulum is moved by a spindle with a resolution of 5mm/rev. the spindle is part of a Compact Module from Rexroth (CKK 12-90), and is 750 mm long. The spindle is in turn moved by the motor through a timing belt (see tb in figure 4.19) with a gear ratio of 1:1 (i.e., one revolution of the motor produces one revolution of

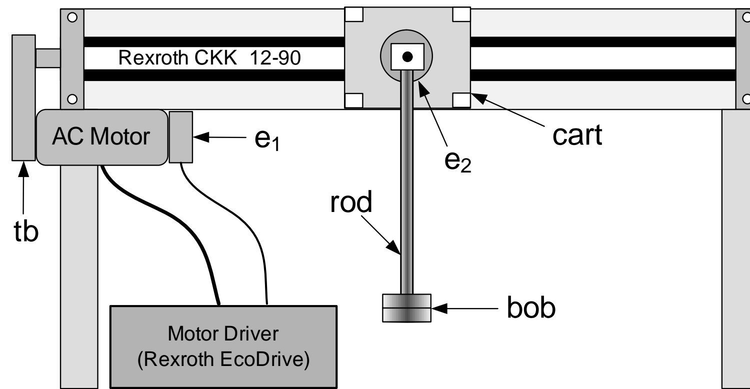


Figure 4.19: Schematic of the inverted pendulum system

the spindle). The pendulum itself is composed by the cart (weighting 0,514 Kg), the rod (measuring 241,5 mm and weighting 254,6 g), and the bob (weighting 326,4 g).

The inverted pendulum test-bed is shown in figure 4.20. The data processing system architecture is mapped onto two daughterboards of the RAPTOR system. The DB-MC module is based on a Xilinx Spartan-II E FPGA (XC2S50E), which controls the interaction with five eight-channel analog to digital converters (ADC), one four-

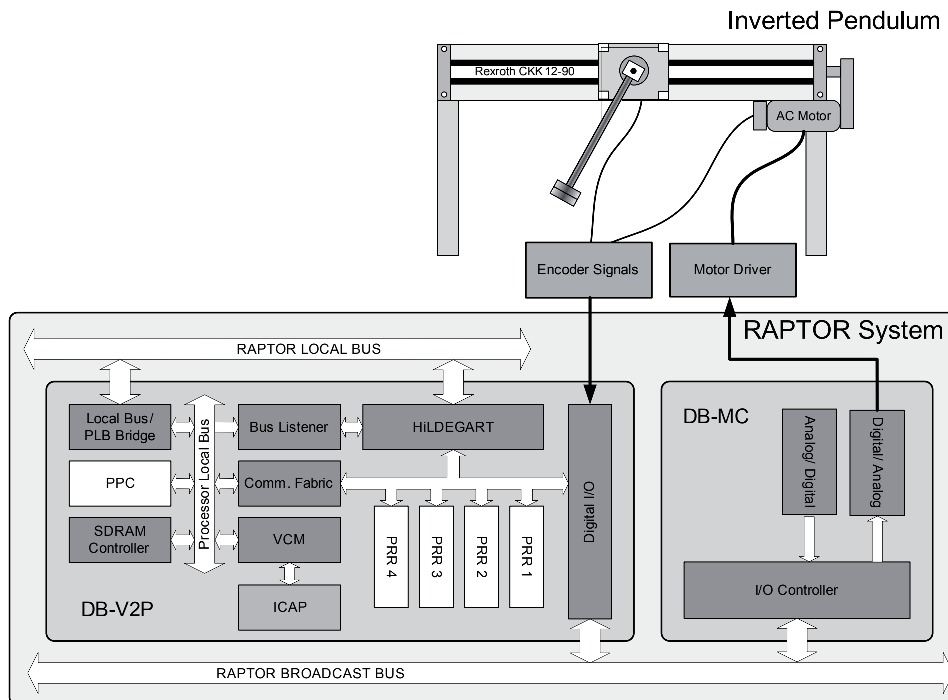


Figure 4.20: Schematic of the test-bed of the inverted pendulum system

channel digital to analog converters (DAC), and one four-channel serial synchronous interface (SSI). Incoming and outgoing data is stored in Block-RAMs located in the FPGA, and can be accessed through the broadcast bus of the RAPTOR system. The rest of the architecture is implemented on a second daughterboard equipped with a Xilinx Virtex II-Pro FPGA (XC2VP20), as described in the previous section.

For the inverted pendulum example, only one DAC is used, to send the control values in a range of 0 to 10 volts (14 bits resolution) to the Motor Drive. Position sensors (encoders) are directly connected to digital I/Os of the Virtex II-Pro module.

Control Structures

A controller for the inverted pendulum has two basic tasks: bring the pendulum from its rest position to a vertical position, (swing up the pendulum), and once the pendulum is at the up-right position the second task is to keep it balanced.

The swing-up controller follows a simple strategy. The controller starts by moving the pendulum to the right, then it waits until the pendulum reaches an angle $\theta \approx 0$, and moves the cart in the opposite direction. In this way, it is assure that the movements of the cart contribute to gradually add energy to the movements of the pendulum until it reaches the upright position, as depicted in figure 4.21.

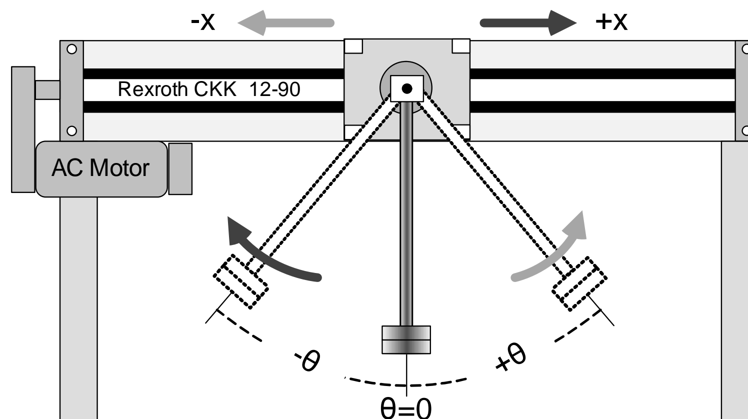


Figure 4.21: A simple strategy to swing-up the inverted pendulum

To implement the swing-up control strategy, a position controller combined with a state machine is used. A simplified block diagram is presented in figure 4.22. The set-point computation block (a) generates two possible maximum positions, whose values depend on the current angle of the pendulum (e.g., for small angles high values, and vice versa). The finite state machine block (b) decides which set-point values to use, also depending on the current angle of the pendulum (e.g., when $d\theta/dt > 0$ and

$\theta \approx 0$, then $x_{set_point} = x_{max_negative}$). Finally, a cascade-position controller computes control signal based on the set-point and the current position of the cart.

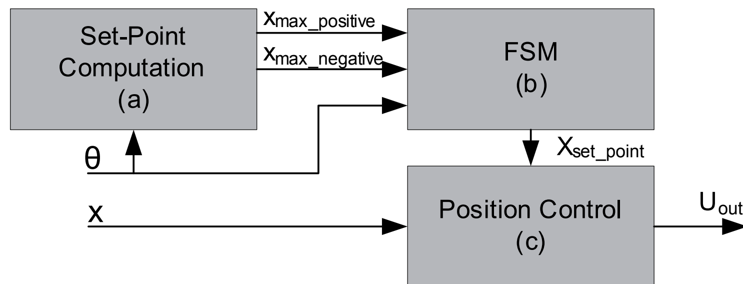


Figure 4.22: Block diagram of the swing-up controller

The balance controller is based on a state-feedback scheme, as depicted in figure 4.23. A linear model of the plant was derived, from which the state-feedback gains are computed using a Linear Quadratic Controller (LQR) approach. The system described in figure 4.23 is then discretised (sampling rate = 1 *ms*) and converted to a state-space representation.

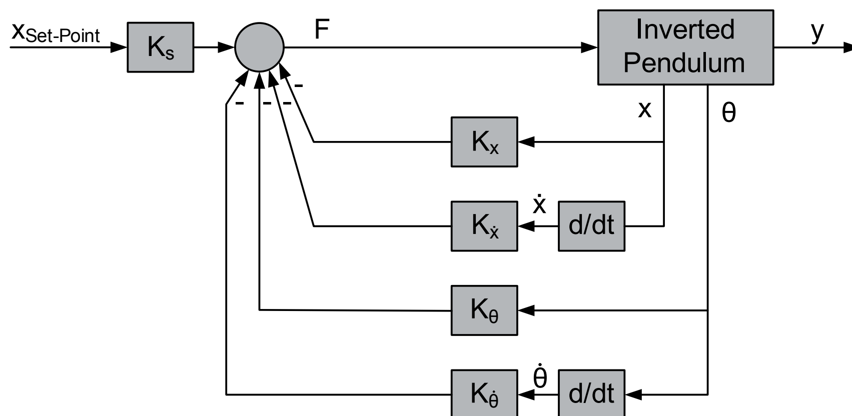


Figure 4.23: Block diagram of the balance controller

Both controllers are implemented using System Generator from Xilinx. Furthermore, a signal processing module is implemented to generate position, angle, speed and angular position from the pulsing signal sent from the incremental encoders to measure horizontal position and angle of the pendulum. Table 4.2 summarises the resource utilisation of both controllers and the signal processing module.

The static implementation corresponds to the synthesis of all modules together, whereas the rest of the values are results of individual synthesis.

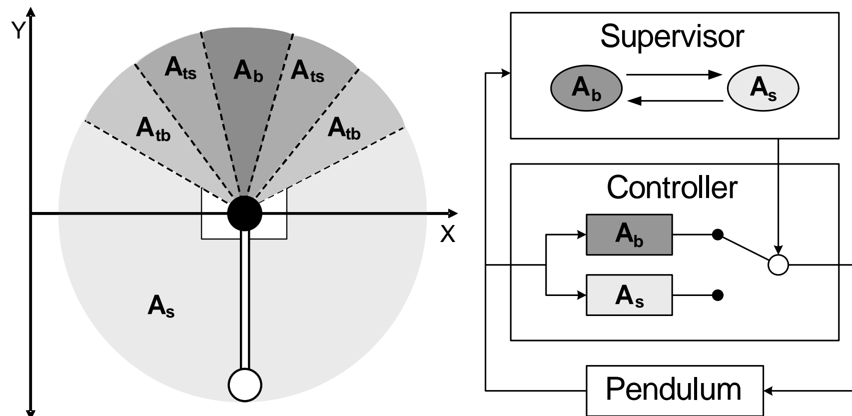


Figure 4.24: Schematic diagram of a two-controllers system for the inverted-pendulum

Control Reconfiguration Scheme

Controller switching is decided upon the current angle of the pendulum, its angular speed, and the position of the cart relative to the spindle. The idea is to activate the balance controller when the pendulum is near the upright position, and when its angular speed is in a suitable range (i.e., $Angular_Speed_{max} \geq Angular_Speed \geq Angular_Speed_{min}$). Following this idea, the area where the pendulum moves can be divided in operative and transition regions, as shown in figure 4.24. A_{tb} are the transition areas from the swinging-up controller to the balancing controller. It should be noticed that a controller exchange from swing-up to balance can also occur in between these two areas (i.e., in areas A_{ts} and area A_b), as long as the angular speed is in the previously-mentioned suitable range. A_{ts} are the transition areas from balancing controller to swinging-up controller. Furthermore, A_b is the operative region for the balancing controller, A_s is the operative region for the swinging-up controller.

The values of $Angular_Speed_{max}$, $Angular_Speed_{min}$, A_{tb}, A_{ts} are determined experimentally. The selection of transition regions determines not only the achievement of control goals, but also the worst case resource utilisation. Control switching re-

Design	Slices	LUTs	FlipFlops	Mults	BRAMs
Swinging-Up	470	615	371	0	0
Balancing	779	1038	1090	2	6
Signal Processing	414	481	660	0	8
Static	1625	2509	1394	2	10

Table 4.2: Resource utilisation of swinging-up and balancing controllers, synthesised for Virtex II-Pro FPGA

garding the used PR Region and reconfiguration time for the inverted pendulum is exemplified in figure 4.25.

It can be noticed that the reconfiguration lasts several control cycles. However, the early switching between swinging-up controller and balance controller allows that the new controller is loaded and ready for operation before the pendulum leaves region A_b. This in turn allows the utilisation of one single slot (in case of using fixed slot placement, cf. section 4.2.4), because the new-loaded controller does not require any initialization.

The reconfiguration time, and the required resources of the control system depend on the selected FPGA device (cf. table 4.1), the partition approach, and the placement method. A static version of the controller requires $A_{static} = 1625$ Slices + 2 Multipliers + 10 BRAMs cf. table 4.2. Supposing a device, which allows a tile size of $s_{max1} = 779$ Slices + 2 Multipliers + 6 BRAMs, the resource requirements of a fixed-slot size approach would be $A_{partial_RTR_FS} = 1558 + 4$ Multipliers + 12 BRAMs + A_{rec}. Neglecting the reconfigurable resources overhead require to realise dynamic reconfiguration (e.g., external reconfiguration is used, c.f. figure 4.8), $A_{partial_RTR_FS} < A_{static}$. However, if a self-reconfiguration approach is used, A_{rec} is not small enough to be neglected, which would lead in this case to $A_{partial_RTR_FS} > A_{static}$. For a situation where more control structures are involve, $A_{static} > A_{partial_RTR_FS}$ even when using a self-reconfiguration scheme.

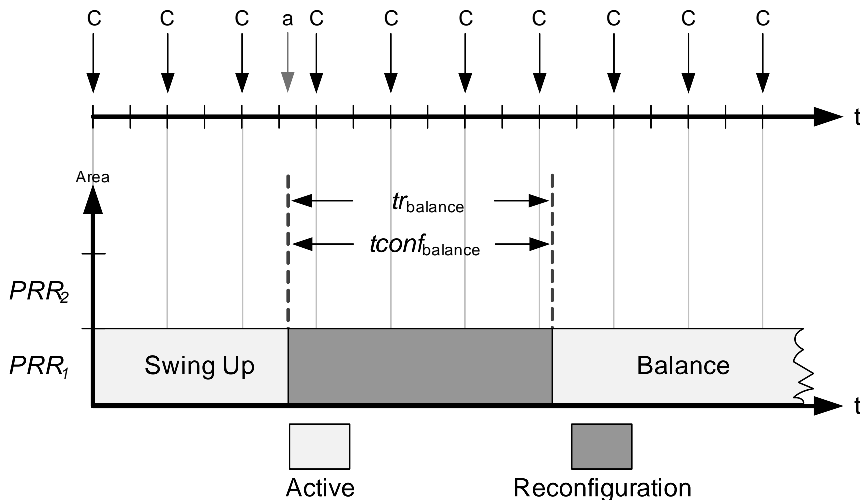


Figure 4.25: Use of PR Regions for the inverted pendulum controller

Control Reconfiguration Validation

Using the test-bed described in figure 4.20 both controllers and the switching strategy were tested. The switching strategy is implemented as software in the embedded PPC. Measurements of the output of swinging-up and balancing controllers, as well as the position of the cart, the angle of the pendulum, and the active controller are shown in figure 4.26.

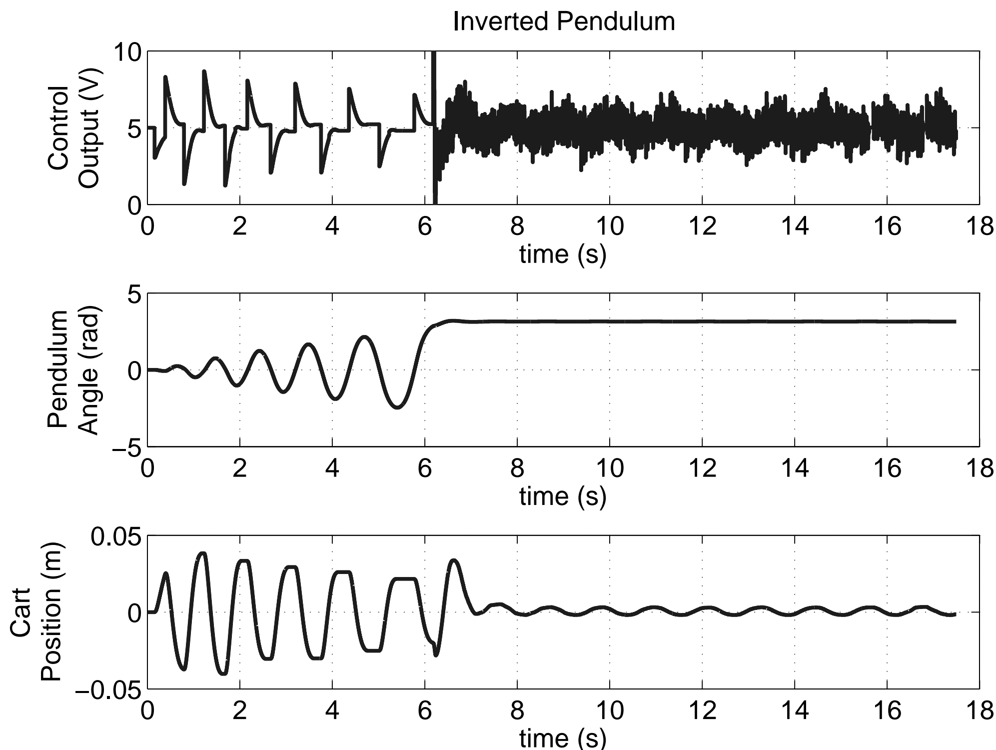


Figure 4.26: Controller exchange for the inverted pendulum system

The Output of the controller has a range of 0 to 10 Volts, where 5 Volts sets the cart to stand still. The angle of the pendulum is presented in radians, where either $\pm\pi$ indicates that the pendulum is at the upright position. The cart position, presented in meters, shows the cart starting in the middle of the Compact Module (cf. figure 4.21).

The system starts in the swinging-up state and stays there until the pendulum enters in any of A_{tb} areas. Then the output of the controller is set to no motion. For this, the supervising program switch the source of the output HW_OUT (cf. figure 4.17) to the PPC, and sends a value to the DAC corresponding to 5 Volts. After this, partial reconfiguration is initiated and the supervisor enters a wait state until the reconfiguration is done. The VCM initiates a bitstream fetching, and transfers

the partial bitstream to the ICAP port. When configuration is done VCM sends an interrupt to the PPC. The reconfiguration process for the balance controller takes about 8,76 ms (two PR Regions are required to map this controller), whereas the Swing-Up controller reconfiguration takes 4,38 ms. Upon leaving the wait-state the output of the controller is enabled again, entering the balancing state. These measurements show that the controllers, the switching strategy, and reconfiguration scheme work satisfactory.

4.4.4 Self-Optimising Motion Controller

This implementation example outlines the implementation of a self-optimizing system composed of several possible hardware and software realisations of controllers for a permanent magnet servo motor. How well a specific controller realisation is suited to the current situation is evaluated based on control quality and realisation effort (i.e., CPU time, reconfigurable area). This example is the result of a collaboration with the Power Electronics and Electrical Drives Group of University of Paderborn. The goal of the example is to show the use of dynamically reconfigurable hardware to realise self-optimizing controllers. The main aspects of this section have been published in [7], [14], and [15].

Self-Optimization Scenario

For this example, a complex mechatronic system composed by many sub-tasks is considered. The computational hardware is shared among all sub-tasks, and must be understood as having limited resources, e.g., memory, CPU-time, or FPGA area. The drive-control sub-task is composed of various controllers, and different realisations of those controllers (e.g., CPU- or FPGA-based), which consequently have different computational requirements, and control characteristics. In a self-optimizing system, a control algorithm may be understood as an optimal solution for the current internal and external objectives of the system. Therefore, to each possible situation of the system, there is a drive controller, and correspondingly an FPGA- or CPU-based implementation of that controller, which represents a solution in that situation.

From the drive-control point of view, the following operative conditions are considered:

- Stationary operation: constant speed and constant load-torque
- Accelerated motion: with constant load-torque
- Load change: constant speed and varying load torque

Furthermore, from the information processing system point of view, the following states are considered:

- Limited CPU-time: the mechatronic system allocates computing time for other sub-task with higher priority
- Limited FPGA-area: higher priority sub-tasks get access to more FPGA-area

These operating conditions are crucial in determining the objectives of the system. Thus, to realise controllers that compete with other sub-tasks of the mechatronic system to access the limited computational resources, the optimal operational condition of each controller and their possible realisations should be known. With respect to the drive application, a concurrent FPGA-based realisation of all required control algorithms would enable control adaptation, as presented in [Mat07, Mat11]. However, the amount of computational resources that have to be allocated to that sub-task from the mechatronic system would be too high. By introducing partial run-time reconfiguration the resource allocation can be improved, and thus the mechatronic system can assign free resources to other sub-tasks.

According to the definition of self-optimization [14], the decision to switch between different control algorithms or different implementations of those algorithms has to be taken in three steps:

1. Analysis of the current situation: By determining the kind and amount of available resources (Memory, CPU-Time and FPGA-area), and the current situation of the controlled system.
2. Determination of objectives: By distinguishing the optimal solution for the total mechatronic system according to the drive application as well as to the cost-benefit ratio for the control switching. The characteristics of the available controllers (cf. table 4.3) and their implementation (cf. table 4.4) are considered in this step.
3. Adaptation of the system behavior: Accomplishing control switching (if required). This step has a direct influence on the available computational resources and the control quality.

The cyclic repetition of these three steps satisfies the self-optimizing framework [Böc06]. This example focuses on the control drive sub-task, without considering a concrete mechatronic system or other sub-tasks. Different realisations of well-known control structures are used to explore controller switching between several kinds of implementations.

Test Bed Description

A simplified schematic of the information processing system and its connection to the EUT (Equipment Under Test) is shown in figure 4.27. Power electronics and computation hardware are connected through a fully isolating digital interface board for sensor and actuator signals. Activation signals for power electronic are created in the FPGA system, and are transmitted as digital signals to the switches.

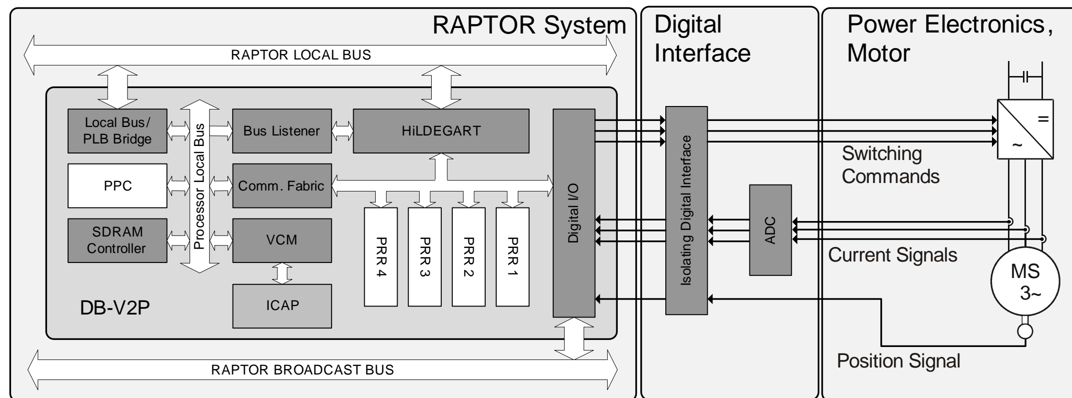


Figure 4.27: Schematic of the test-bed of the self-optimizing motion controller

The ADCs to sense current signals use a delta-sigma-modulator, which allows that the quantization as well as the sampling rate of the current sensor signals are scaled by an optimized decimation filter [Pet09]. The utilisation of sensor signals for current and position in the computation hardware is supported by the power electronic system. This test bed allows emulating many different drive applications, such as speed, position and torque control. The special capability of this test bed is the on-line reconfiguration of the drive controllers. This is not restricted to only FPGA-based controllers; the exchange of CPU- and FPGA-based realisations is also supported. This feature requires a flexible underlying information processing system, not only because different realisations of the controllers are supported, but also because the information flows from sensors and to actuators have to be reconfigurable at run-time (cf. section 4.4.2).

Drive Control Structures

In this implementation example, torque controllers for a permanent magnet synchronous motor drive are considered. All controller structures are based on a Field Oriented Control (FOC) scheme [Bla72]. The FOC scheme consists of controlling the stator currents represented by a vector. This control is based on projections, which transform a three-phase time and speed dependent system into a two coordinate (d- and q- coordinates) time invariant system [Nab80]. These projections lead to a structure similar to that of a DC machine control. Field orientated controlled machines need two constants as input references: the torque component (aligned with the q coordinate) and the flux component (aligned with d co-ordinate).

The differences between the controllers investigated in this case-study are the consideration of the feed-forward parts for the Back-EMF and the decoupling between the currents in d- and q-axes. In the elementary control structure of an FOC-scheme

the output of a PI-controller is directly the output of the controller. The medium scaled structure (FOC-EMF) contains a feed-forward for Back-EMF compensation. As such, the dynamics of the control loop is improved for speed changes. The large scaled control structure (FOC-EMF-DeC) has an additional decoupling of the currents for improving the behavior in the case of load torque change, cf. table 4.3.

The properties of these controllers are well known and have been presented by several authors (e.g.,[Bla72], [Bay72], [Mon02c]). In this case-study the implementation of the controllers using FPGAs as target-architecture, and their run-time exchange by means of partial run-time reconfiguration as a way of providing self-optimization to the system is presented. Moreover, the possibility to switch between software- and hardware-based realisations as a further degree of flexibility is explored. Special attention has been paid to the transition between controllers, since this can lead to undesired effects (e.g., disturbances). The common abilities and some realisation aspects of the considered controllers are shown in table 4.3.

The considered FOC structures can be separated in two parts. The first part contains modules for output (reference voltage in stator-fix α-, β-axes), and input (measured currents in stator-fix α-, β-axes and position) as well as the coordinates transformation. These computations are placed in a Basis Region of the system architecture, because they are common for all realised control schemes. The second part contains the current controllers for d- and q-axis, which is the main part of the controller structure.

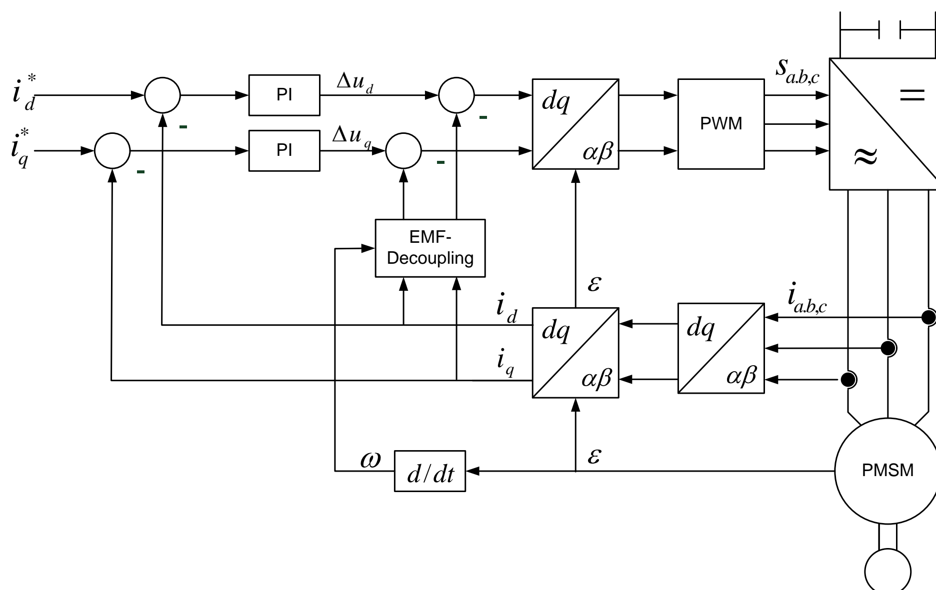


Figure 4.28: Schematic of a Field Oriented Control structure with back-EMF compensation and decoupling of currents [Nab80]

No.	Control Algorithm	Situation revolution speed	load torque behavior	Complexity computing time per cycle	required sample rate
1	FOC (P-Controllers)	low	constant	low	slow
2	FOC (PI-Controllers)	low	constant	low	slow
3	Back EMF compensation	medium	constant	medium	medium
4	current decoupling	high	fluctu-ating	high	high
5	Direct Torque Control	very high	fluctu-ating	low	high

Table 4.3: Abilities and realisation aspects of motor controllers [15]

The presented FPGA-based controllers were realised using the Xilinx System Generator [Sysb]. FPGA resources for the different control structures and the hardware interface used for measurements are given in table 4.4. These implementation results are based on a Xilinx VII-Pro XC2VP30-FPGA.

Structure: FPGA Resources:	Hardware Interface	FOC Ctrl	FOC Init	FOC-Back EMF Ctrl	FOC-Back EMF Init	FOC-EMF-DeC Ctrl	FOC-EMF-DeC Init
Slices	1273	360	194	453	283	671	497
FlipFlops	1052	153	74	172	91	272	187
LUTs	1525	605	340	767	502	1154	889
BRAMs	1	4	0	4	0	4	0
MULTs	1	0	0	0	0	2	2

Table 4.4: Resources of implemented FPGA-based controllers [15]

All controllers presented in table 4.4 are based on a PI-controller. As explained in the next section, for each controller there is an initialization block, which computes the initial state of the integrator of the to-be-loaded controller. This block is loaded concurrently to the controller, and uses input and output signals of the to-be-replaced controller to compute the initialization value.

The realisation of CPU-based controllers is commonplace in industrial drive applications. Furthermore, the theoretical and practical aspects of CPU-based drive control

reconfiguration can also be found in literature [Ho90], [Kha91], and [Mon02c]. Therefore, the realisation of such standard CPU-based controllers is not further analysed. However, the dynamic reconfiguration of FPGA-based controllers, and the switching from an FPGA- to a CPU-based controller is a new step in drive controllers.

A comparison of the execution-time of FPGA- and CPU-realised controllers on our SoC architecture is presented in figure 4.29. The PPC works with a clock frequency of 300 MHz, whereas the reconfigurable PR Regions have a clock frequency of 30 MHz. The PWM-Carrier is depicted to illustrate the timing constraint of the controllers (i.e., control cycle). The used Delta-Sigma-ADC is realised using regular sampling.

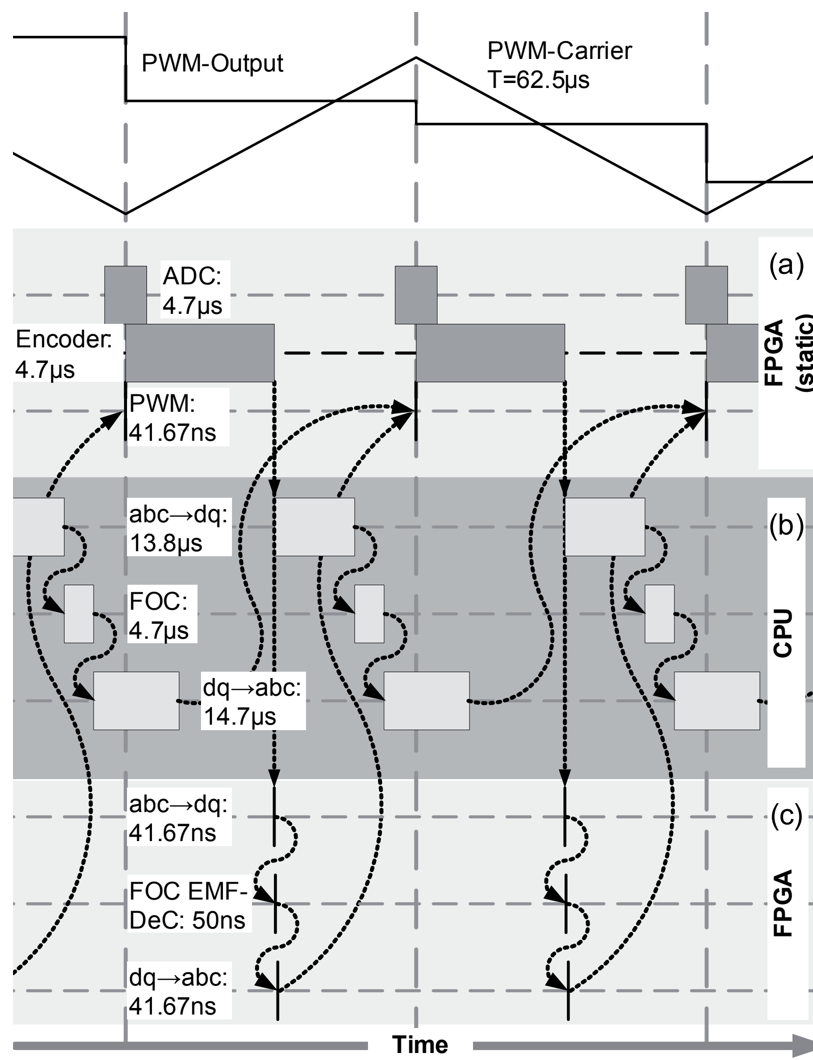


Figure 4.29: Comparison of the execution-time of (b) CPU- and (c) FPGA-based motor controller realisations (FOC and FOC-EMF-DeC correspondingly), including execution times of the (a) static part of the controller [15]

As can be seen, the timing of the encoder is dominated by the serial data transfer and synchronised to the ADC. The outlined timing of static part of the PWM supports the displacement for the zero-voltage vector of the set-voltages (Zero Sequence Signal). Even though the precise timing depends on the actual clock-rate, it can be noticed that the execution time of the CPU realisation is longer than a single control cycle, causing the controller to have a time delay of one sample period. The FPGA realisation is two orders of magnitude faster than the CPU realisation, and has no significant time-delay. This speedup comes from the concurrent utilisation of several processing elements (cf. Table 4.4), in contrast to the serial realisation of the CPU-based controller. The low execution-time of the FPGA-realisation enables the implementation of more complex control schemes (e.g., a speed-adaptive PWM-period can be easily implemented).

Controller Reconfiguration Scheme

Figure 4.30 shows the sequence diagram for the run-time reconfiguration of torque controllers. Controller 1 is the active design at the system start. When the Supervisor detects that a new controller structure (e.g., Controller 2) is required, it starts loading the necessary initialization function and Controller 2 on the PR Regions of the system architecture. This is done by sending a configuration request from the PPC to the VCM, which then starts loading the corresponding partial bitstream from the external SDRAM and places it in a free PR Region of the FPGA. After this operation has been performed, the VCM sends a handshake signal to the PPC. The initialization module calculates the internal states of the new controller for initialization directly after both components are loaded. This initialization has to be done with the input and output signals of the to-be-replaced controller.

The initialization module detects the steady state of the input and output signals to ensure that the initial values are valid. Switching between the controllers is completed within one clock-cycle (i.e., the assignment of HW_OUT, cf. figure 4.17), so that the controller output is continuous (bumpless) from the point of view of the motor. After the controller has been replaced, the components Initialization and Controller 1 are no longer required and the corresponding PR Regions can be used by other sub-systems of the mechatronic system. After the initialization of the new controller, the Supervisor resumes to analyse incoming and outgoing data. The worse case resource utilisation when using 1-D partition, with a fixed-size slot placement approach is exemplified in figure 4.31, where PR region PRR1 holds the to-be-replaced controller (C_1), PRR2 holds the to-be-used controller (C_2), and PRR3 holds the initialisation routine (u_2). Because the reaction time of any of the to-be-reconfigured controllers is $tr_{max} \gg p_{max}$, the resource requirements are $A_{partial_RTR_FS} = 3 \cdot S_{max1} + A_{const}$, as shown in figure 4.31.

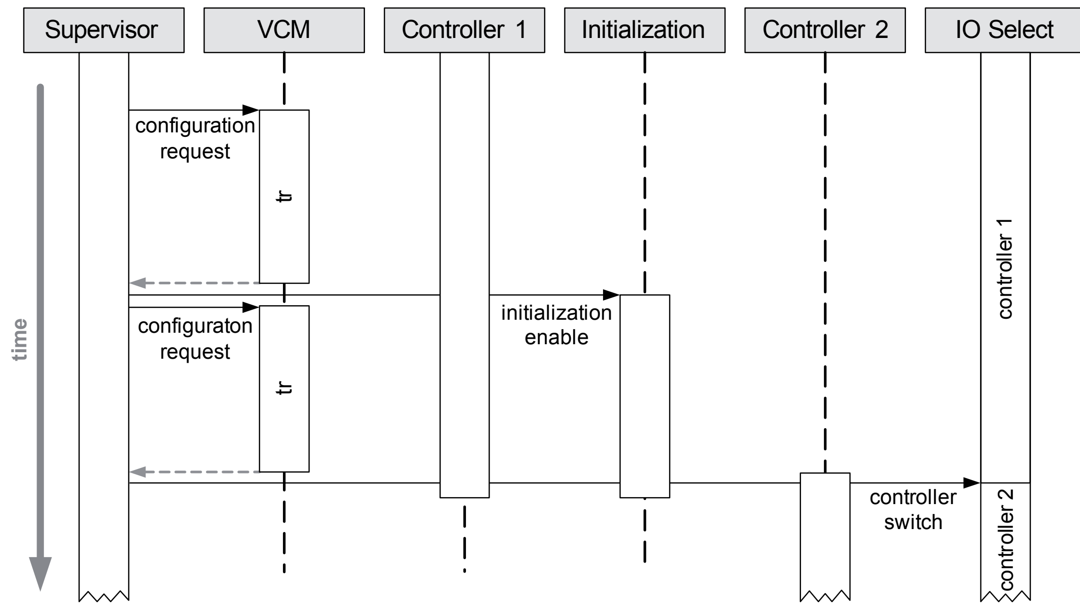


Figure 4.30: Flow of the run-time reconfiguration of controllers [15]

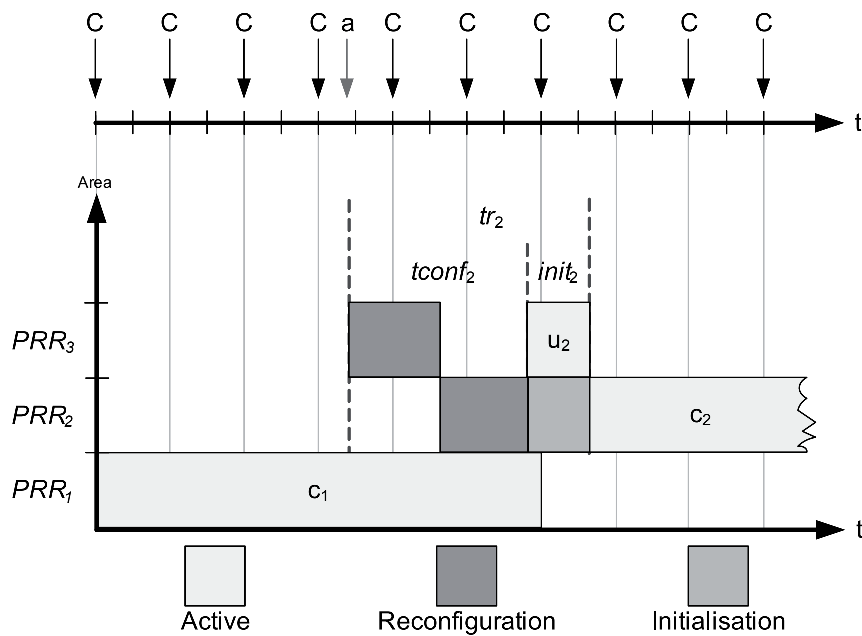


Figure 4.31: Use of PR Regions for the self-optimizing motion control controller

Control Reconfiguration Validation

For validation of a proper switching between controllers, a HiL-Simulation (cf. chapter 5, sections 5.2 and 5.3) of such control exchange is presented in figure 4.32. The motor is first controlled with a FOC (No. 2 in table 4.3), and at time-point zero the control is switched to a FOC-EMF-DeC (No. 4 in table 4.3). As can be seen, switching was done without disturbing the controlled currents. To enable this bump-less control switching, a proper initialization of the internal state of the controller (e.g., integral initial state) is required [7], as discussed in the previous section and presented in figures 4.32 (cf. also figure 5.25, in chapter 5). Without such an initialization the controlled currents show a disturbance at the time of the switching, as can be observed in figure 4.33. In this figure measurements of a control switching with the EUT are shown, using the same controllers as in figure 4.32, but without initialisation.

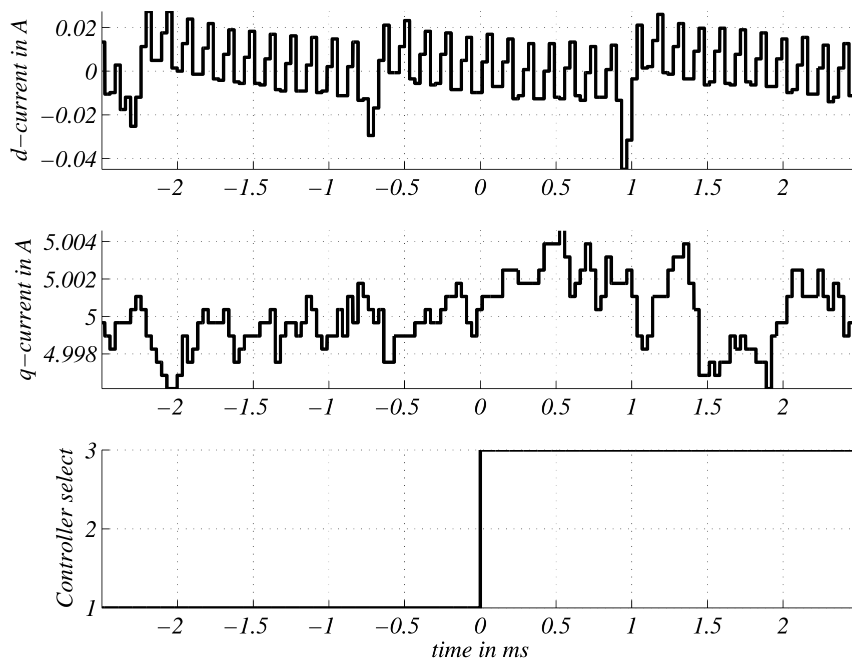


Figure 4.32: Controller switching from FOC to FOC-EMF-Dec at 3000 RPM (HiL: Controller at FPGA, Motor in Simulation) [15]

Figure 4.34 shows measurements of a controller switching between a FOC based on a PI-controller (table 4.3, No. 2) realised on the PPC with a control period of $p_{ppc} = 68,3\mu s$, to an FPGA-based FOC with a P-controller (table 4.3, No. 1) with a control period of $p_{ppc} = 34\mu s$. The amount of noise of the current is defined by the selection of the control algorithm, its realisation and external perturbations. On the one hand, the P-controller used for the measurements shown in figure 4.34 produces low noise, but produces also a steady state error. On the other hand, the PI-controller

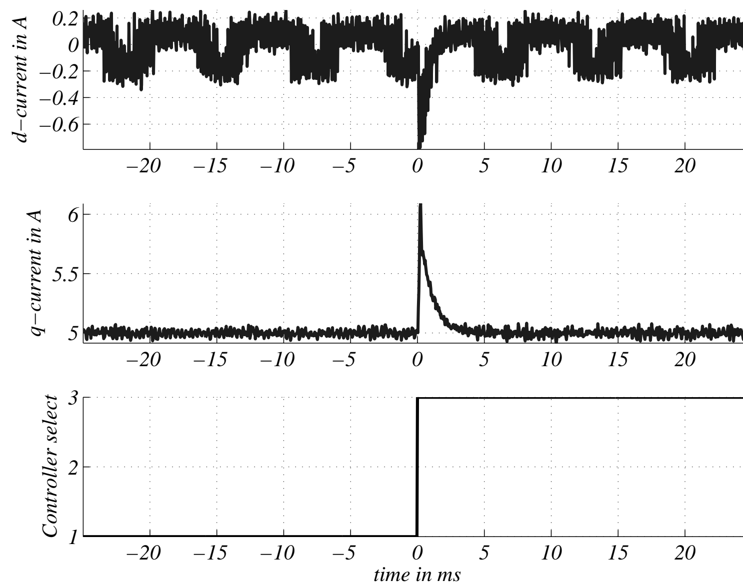


Figure 4.33: Controller switching from FOC to FOC-EMF-Dec at 3000 RPM without using initialisation (Test bed: Controller at FPGA, Motor as EUT) [15].

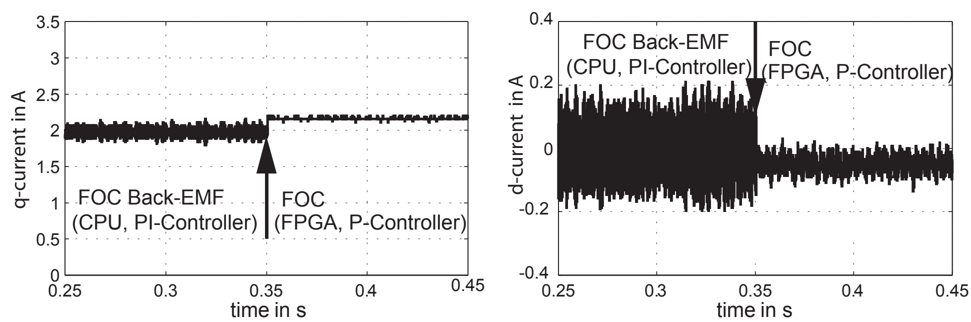


Figure 4.34: Controller switching from a CPU- and an FPGA-based realisations at 3000 RPM (Test bed: Controller at CPU and FPGA, Motor as EUT) [15]

has a better steady state response, but requires more resources for its implementation. These measurements and simulations results show that the presented concept works satisfactory.

4.5 Summary

This chapter presents the use of run-time hardware reconfiguration for control applications. It is shown that FPGA-based control systems requiring adjustments during operation can benefit from RTR. Two cases of adjustments are distinguished: structural and parametric changes. For both cases RTR can be used to achieve a better resource utilisation, depending on the amount of structural variations, or the size of the algorithm requiring parametric changes. It is shown that resource utilisation of a dynamic approach depends on the worst-case configuration of the system, whereas for a static implementation the resource utilisation depends on all required configurations.

The process of dynamic hardware reconfiguration is presented, showing aspects such as system partition approaches (1D, Multi-1D, and 2D), placement strategies (fixed-size slots, and free placement), kinds of communication infrastructures for reconfigurable systems (point-to-point vs. shared lines), and the configuration control (internal or external). The realisation of these aspects depends on the application, and the underlying hardware platform (e.g, the chosen FPGA device). The worst-case resource utilisation for dynamic systems is analysed for a free placement and a fixed-size slot approaches.

Two application examples are presented: an inverted pendulum system, and a self-optimising motion controller. Basis of this realisations is a System-on-Chip (SoC) architecture that enables the use of dynamic hardware reconfiguration, and the run-time adaptation of the communication infrastructure. The architecture is based on a Virtex-II Pro FPGA from Xilinx, which includes two embedded PowerPC processors.

The dynamic reconfiguration of two controllers for the inverted pendulum is shown in the first application example. This proof-of-concept example shows the potentials of using RTR compared to a static approach. The early switching between controllers allows the utilisation of one single slot, despite the fact that the reaction time of both controllers is longer than the control period.

Having too long configuration times in comparison to the duration of control cycles poses a challenging problem, because controller initialisation strategies are required, for which reconfigurable resources have to be allocated . In the case of the inverted pendulum system, a smart switching strategy allows to work around this situation. In general reconfiguration latencies have been improved in the newest Xilinx devices, where the reconfiguration controller works with higher bandwidths (up to 32 bits at 100 MHz), and the minimal reconfiguration areas are smaller (cf. section 4.2.2). However, this is still not sufficient to guarantee that the response time of a controller (i.e., reconfiguration time and initialisation) is always shorter than the required control period. Therefore, a strategy is shown in the second example, where the to-be-configured controller is initialised to achieve a bump-less transition between controllers.

In the second implementation example the realisation of an FPGA-based self-optimizing motion controller is presented. This approach allows the adaptation of parameters and structure of controllers. Furthermore, not only the control algorithm, but also its realisation and the execution platform (FPGA or embedded CPU) can be dynamically changed. It was shown that switching between different FPGA-based realisations and from an FPGA- to a CPU-based realisation (and vice versa) can be done, without perturbation of the controlled system.

Given the demanding requirements of the controlled electric drive, achieving a bump-less switching is a special qualitative feature of this implementation. Furthermore, considering the short execution times of FPGA-based controllers, and the possibility to still use a CPU-based controller, allows the adaptation of the control system, not only regarding the plant, but also regarding the available resources of the underlying computing architecture and how they are used by other sub-systems. This empowers the control system to react to situations far beyond the classic approaches.

Measurement of both realisation examples show that dynamic hardware reconfiguration is a promising approach to realise complex control schemes.

5 Design Verification through Hardware-in-the-Loop Simulations

Verification is a crucial part in the design of digital controllers. Simulations are usually the starting point to verify the design under test (DUT), using mathematical models of both the plant and the controller. Later on, the integration of the target architecture (e.g., a microcontroller or DSP) or parts of the plant in the simulation loop, known as Hardware-in-the-Loop (HiL), makes verification more realistic and thus more effective. Such verification methods have been intensively reported in literature for CPU-based digital controllers, being a standard in current design flows.

In this chapter a HiL design environment for FPGA-based controllers is presented, which includes an off-line simulation framework and an on-line monitoring tool. These frameworks support the design flow of FPGA-based controllers targeting run-time reconfiguration. The proposed frameworks where done in collaboration with Christopher Pohl, and are also described in chapter 4 of [Poh10]. In the present work, the proposed frameworks are explained in the context of mechatronic systems and the utilisation of FPGA technology for the design of digital controllers.

In the next section a brief review of HiL verification approaches is presented, including the state-of-the-art in FPGA-based HiL simulations. Afterwards, the off-line framework, HiLDE (Hardware-in-the-Loop Design Environment) is introduced, showing the underlying hardware and software components, and implementation examples. The on-line framework, HiLDEGART (Hardware-in-the-Loop Design Environment for Guided Active Real-Time Test) is then introduced, exemplifying the framework with two realisation examples. The chapter finishes with a brief summary.

5.1 Classification of Test-Systems

In the design of digital control systems, the design under test (DUT) is tested at different stages of the design flow, using different levels of abstraction and a variety of technologies, as depicted in the V-model in figure 5.1.

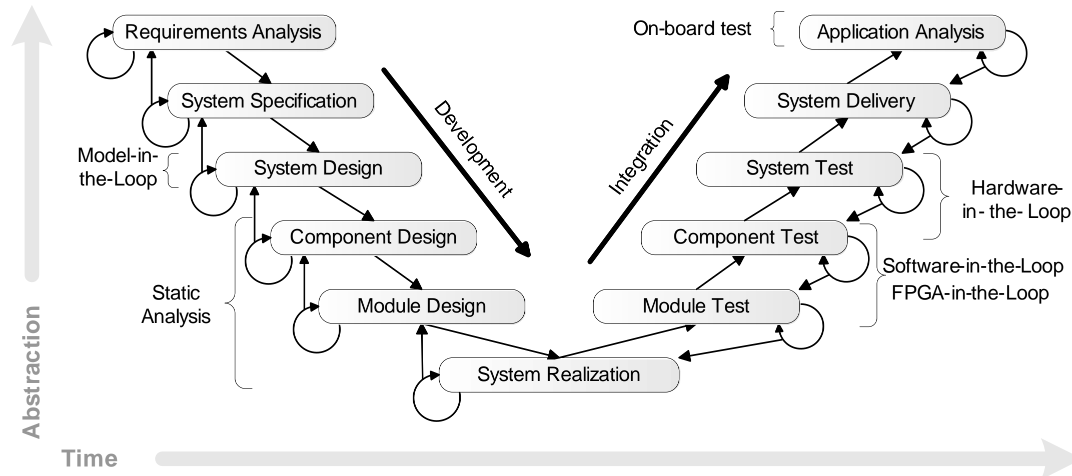


Figure 5.1: Positioning of FPGA-in-the-Loop simulations in the V design model

On the left-side of the V-model, different stages of development of the design are shown in a top-down oriented approach, and on the right side the correspondent test and integration steps are shown in a bottom-up oriented approach. The V-model also represents from top to bottom decreasing abstraction-levels, and from left to right the time-flow of the design process. As can be seen in this model, early in the design flow simulations are usually carried out, using mathematical models (model-in-the-loop) of both the plant and the controller. Later on, at the integration phase, Software-in-the-Loop (SiL) and FPGA-in-the-Loop (FPGA-iL) are used to test components of the system (cf. figure 5.1). System test is done through Hardware-in-the-Loop simulations, and finally on-line tests are done to perform application analysis. The different test approaches are further explained in the following sections.

5.1.1 Model- and Software-in-the-Loop

In the development phase of the design flow, mathematical models of the controller and its working environment are developed, which enables an early verification of the behaviour of the DUT. This approach is known as model-in-the-loop (MiL). Software tools such as Simulink, or CAMEL-View, are typically used in this stage [Tew02]. Software-in-the-Loop (SiL) is done, when the model of the controller is translated to some architecture-specific code, and the same simulation environment is used. This

step typically includes the conversion of the controller to a different numbering format, e.g., from floating point to fixed point.

MiL and SiL simulations allow the monitoring of internal signals of the design, which are not accessible in later phases of the design flow. Therefore, exhaustive tests are possible, at the cost of larger simulation times.

5.1.2 Rapid Prototyping

To improve the test, the DUT can be tested using its real environment or parts of it. This approach is known as rapid prototyping, and improves on MiL and SiL by bringing the design to a more realistic test environment. The goal of rapid prototyping is to develop a prototype in an early stage of the design flow.

5.1.3 Hardware-in-the-Loop Simulation

A HiL simulation is characterized by the operation of real components in connection with real-time simulated components [Ise99]. The simulated components are either the processes being controlled including sensors and actuators, or the controller itself. Typically, the controller runs in the target architecture, while its environment is simulated in real-time. The benefits of simulation (e.g. high flexibility when accessing signals and internal states) often come along with increased simulation time, which increases with the complexity of the system. In contrast to this, fast execution can be achieved by utilisation of rapid prototyping, however debugging is often difficult to perform due to limited access to the internal states. HiL simulations combine the flexibility of SiL and MiL and the execution speed of rapid prototyping.

5.1.4 On-Line Test

As a final verification step, the DUT can be integrated in its real environment, this approach is also known as on-board test [Har01] in the automotive industry. At this stage the monitoring and parametrisation of the DUT is desirable to verify its behaviour. Thus, some components, such as data loggers or monitoring software routines, need to be added to the DUT. With an on-line test, the correct functioning of the DUT can be asserted.

The different combinations of simulated and real elements in the verification process are shown in figure 5.2. When using FPGA-based systems, a new category can be distinguished, known as FPGA-in-the-Loop (FPGA-iL). Unlike a MiL or a SiL approach, an architecture specific description of the DUT is not used, but its actual implementation using the target technology. Furthermore, this test differs from a rapid prototyping test in that the execution of the DUT is not done in real-time. The focus

of an FPGA-iL test is functional verification of the DUT in its real execution platform, as explained in the next section.

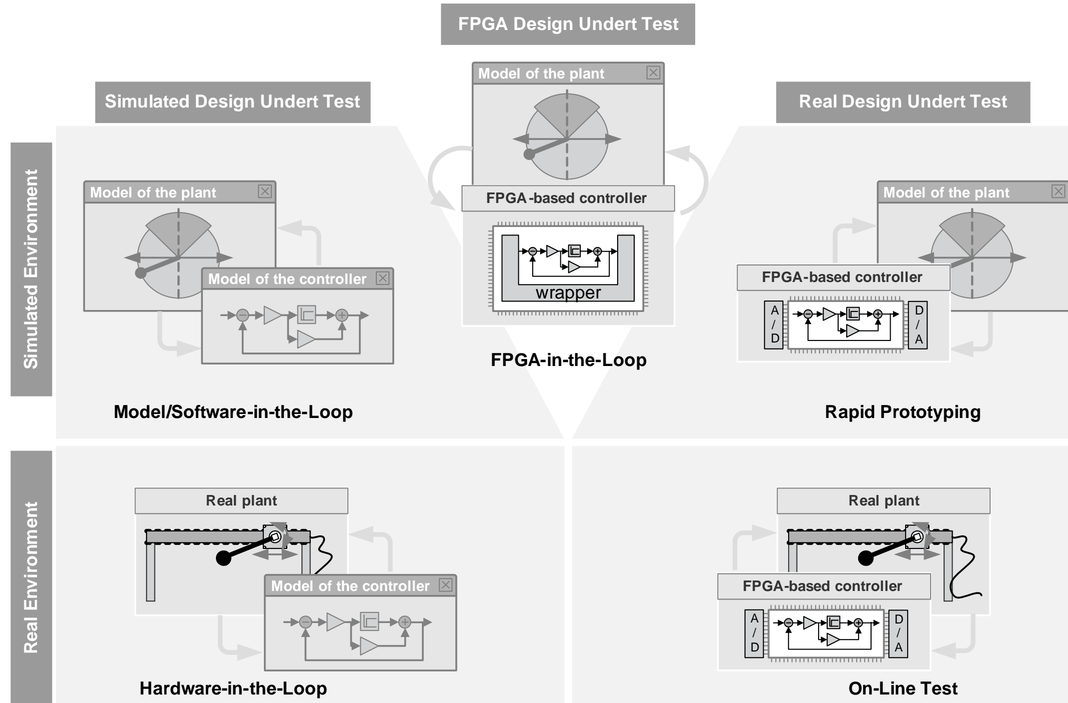


Figure 5.2: Combinations of simulated and real elements in the design process

5.1.5 FPGA-in-the-Loop

Typically, simulation software tools are run using standard desktop computers, which do not support real time execution. However, in order to perform a simulation, where the DUT is executed using an external FPGA boars, a synchronisation strategy between simulation and DUT is required. Therefore, in an FPGA-in-the-Loop simulation the clock of the DUT is controlled in a way that its execution is coordinated with the simulated environment, thus making possible a clock-cycle accurate verification.

As presented in figure 5.2, FPGA-iL represents a new category, where the simulation of the environment controls the speed of the simulation, allowing the functional verification of the DUT in an early step of the design flow. FPGA-in-the-loop can be seen as a step between SiL and HiL, where the focus is on cycle-accurate verification and simulation acceleration.

There are commercially available FPGA-based prototyping boards which can perform FPGA-iL simulations. In [Can02] an FPGA-iL system, DIME from Nallatech, was presented. This board is connected to the host PC via PCI bus. Their approach is

not universal and requires the user to develop on the DIME board. Another example is the *R Series Intelligent DAQ Devices* from National Instruments [Nat11]. The system can only be used with software and development boards from National Instruments (LabView FPGA module). The development boards feature Virtex II and Virtex 5 FPGAs from Xilinx. A further example is the Hardware Accelerator und Cosimulator HAC2 from Gleichmann Electronics Research [Rei05]. In this framework, the DUT is loaded to an PCI-based FPGA board, while the test-bench runs in a Modelsim simulation. A speedup of up to 566 % was reported for open-loop simulations. However, Matlab/Simulink simulations are not supported, an therefore a test-bench has to be implemented in a hardware description language. An interesting feature of this framework is the possibility to include other CPU-architectures in the simulation loop (e.g., ARM processors), for which there is no HDL model available.

HES from ALDEC [ALD11] allows the partitioning of a large designs into several FPGAs, enabling automatic system partitioning. This tool includes its own simulator and supports only some commercially available processor cores (e.g., ARM 720T), which prevent its use for self-designed systems. Another example is the Palladium III system, which can emulate or co-simulate very complex systems (up to 256M ASIC gates and 74GB of memory) [CAD11]. The main drawback of this system is its cost [Ell09].

Academic examples have also been presented. In [Dep04] a framework for the design of control algorithms for mechatronic systems, including HIL simulations, is presented. The design and implementation of linear, time-invariant (LTI) control systems on FPGA technology is described using a self-developed software called Computer-Aided Mechatronics Laboratory (CAMeL), as design environment. However, the use of FPGA-iL was only suggested in this paper. An integration of CaMEL View to the FPGA-iL presented in this thesis is shown in [10]. In [Bra05] an FPGA-iL framework is presented, which enables the realisation of an FPGA design from Simulink blocks, and its verification in an FPGA-iL simulation. The software tool translates simulink blocks into GenericC (a C-like programming language from COSSAP, Synopsys [Syna]) and then to VHDL by using a C to VHDL tool (ARTBuilder, no longer developed). However, it was mentioned in the paper that the resulting hardware description is not optimal, regarding resource utilisation, in comparison with a manually-coded design, being this the main drawback of the proposed approach, because hardware description is completely abstracted from the user. In [Lu05] another framework for FPGA-iL is presented, which is capable of real-time simulations. One of the main components of the framework is a modified Linux kernel for real time applications. A design to be simulated in this framework has to be designed using a different tool-flow (e.g., Matlab/Simulink).

The approach presented in this thesis is platform independent and can be adapted to any existing prototyping environment with a reasonably fast communication link. It allows the integration of multiple FPGAs in the simulation, and support Programmable

Input Output (PIO) and Direct Memory Access (DMA) communication with the host computer. In the following sections the proposed FPGA-in-the-loop is presented in detail.

5.2 HiLDE: HiL Design Environment

HiLDE is an FPGA-in-the-Loop framework, which allows the early verification of a DUT in the design flow [8, 12] (c.f. figure 5.1). HiLDE consists of an FPGA-based rapid prototyping system, RAPTOR (see section 4.4.1), and a set of hardware and software interfaces, which enable the interaction of a DUT running on an FPGA module of the RAPTOR system and a CAMeL-View simulation (cf. figure 5.3), or Matlab/Simulink simulation (cf. figure 5.4).

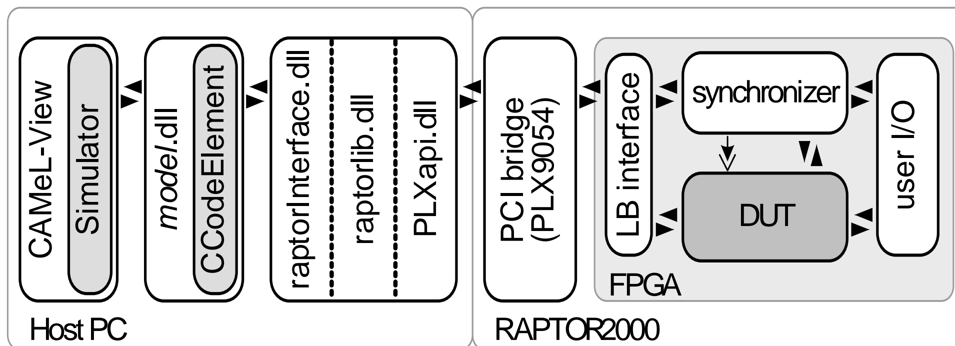


Figure 5.3: Information-flow of the Hardware-in-the-Loop simulation framework with CAMeL-View

The information flow for both simulation tools is shown in the figures 5.3 and 5.4. Parameterisable hardware and software interfaces allows a transparent communication and coordination between DUT and simulation. These interfaces are described in the following sections.

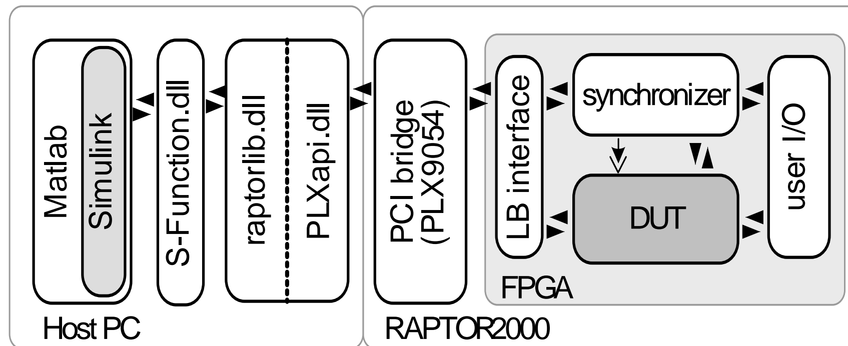


Figure 5.4: Information-flow of the Hardware-in-the-Loop simulation framework with Matlab/Simulink

5.2.1 Hardware Components

Clock management is a critical part of the FPGA-in-the-Loop simulation. In order to generate a coordinated simulation, it is necessary to precisely control the clock of the DUT, because for each simulation step (cf. figure 5.7(b) and figure 5.8) the DUT should run for a specific number of clock cycles, which correspond to the latency of the DUT. This is done by stopping the clock of the DUT in between two simulation steps, an approach known as clock-gating. This is similar to stepping through code in a debugger. Using this approach, the functionality of the design can be verified. However, the timing is not correctly reproduced, and therefore other means need to be employed to verify whether the DUT meets timing requirements or not. The second framework, presented in section 5.3, addresses this topic.

In order to coordinate a simulation with a DUT, an FSM called Synchronizer, has been implemented. Synchronizer enables the DUT clock after a request from either Matlab/Simulink or CaMEWL-View. Furthermore, Synchronizer controls the clock of a given DUT in two possible modes: periodic and aperiodic, as depicted in figure 5.5.

For control applications only the periodic mode is used, since controllers have naturally well defined periods. Synchronizer also detects whether there is a discrepancy between the given sampling period and the time required by the DUT to complete a cycle. This happens if its latency is greater than the sampled period reported by the simulation. In this case the DUT is disabled and a warning signal is sent to the simulation. The simulation can then react to this exception.

Both Synchronizer and the DUT, are embedded in a hardware wrapper, depicted in figure 5.6. The Wrapper provides specialised hardware for interfacing Synchronizer and the DUT with a simulation (using either Matlab/Simulink or CAMeL-View) running on the host computer through the PCI bus (see section 5.2.3). In order to

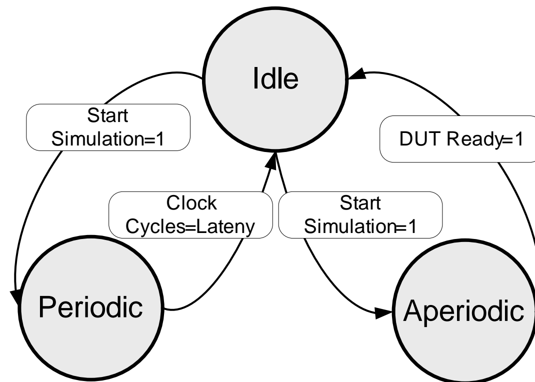


Figure 5.5: Synchronizer state machine

embed the DUT into the Wrapper, the bus interface is adapted to the input and output ports of the DUT. This process is done automatically as described in section 5.2.4.

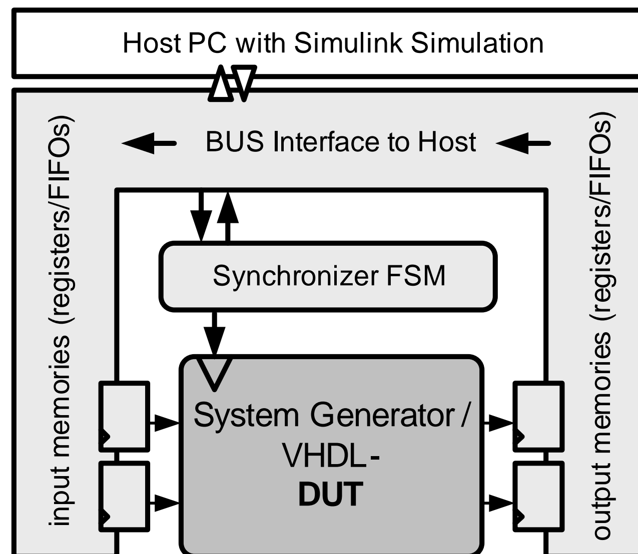


Figure 5.6: Synchronizer embedded in the bus interface

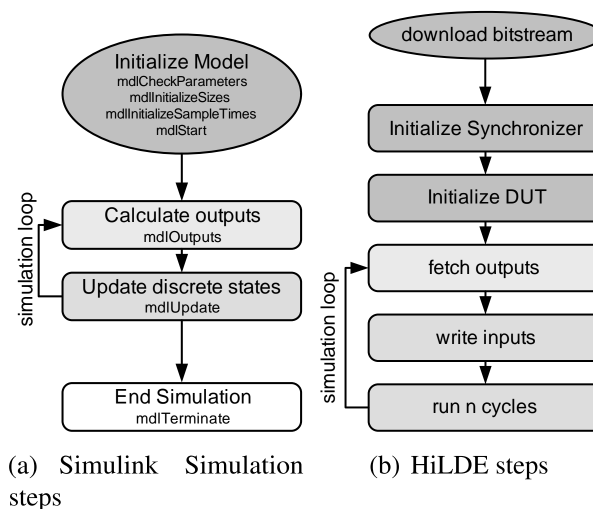
The Wrapper enables reading and writing data from and to the input/output ports of the DUT from the simulation. There are two methods to realise these operations: using a registers bank and using a FIFO memory. For control applications where there is a feedback from the plant, the only suitable read and write method is using registers, since measured signals from the plant have to be fed back to the inputs of the controller, and vice verse, without delay (cf. section 5.2.3). For multi-inputs multi-outputs (MIMO) systems, this process can be accelerated by utilizing DMA transfers (see section 5.2.3).

5.2.2 Software Components

Depending on the simulation software the integration of the RAPTOR system differs. In this section the software component of the HiLDE framework for both Matlab/Simulink and CaMEL-View are described.

Simulink Integration

MATLAB provides a generic interface for integrating user defined software into the Simulink simulation process, the so-called S-Function. The basic simulation steps and their pendants for HiL simulation with RAPTOR are displayed in figure 5.7(a) and 5.7(b). The *mdlStart()* function is used for hardware initialization (i.e., download of the bitstream, configuration of Synchronizer). If *mdlStart()* succeeds, the simulation loop sequentially calls *mdlUpdate()* and *mdlOutputs()*. In *mdlOutputs()* the data in the hardware output registers is read and propagated to the outputs of the Simulink block. In *mdlUpdate()* data from the input ports of the simulink blocks is sent to the hardware input registers, respectively. *mdlUpdate()* also starts the synchronizer to activate the DUT clock for *n* clock cycles, where *n* is the latency of the DUT. In addition to these communication steps, several translation steps from Simulink floating point data types to fix point data types have to be accomplished inside the S-Function. The parameters for this translation as well as information of the hardware configuration are given in a configuration string provided by HiLDE, cf. section 5.2.4.



(a) Simulink Simulation steps

(b) HiLDE steps

Figure 5.7: Simplified simulation flow diagram

CAMeL-View Integration

The integration of CaMEL-View to the HiLDE framework was done in cooperation with the Control Engineering and Mechatronics group of the university of Paderborn, and was published in [10].

The integration of the RAPTOR system into CAMeL-View is realised by a *CCodeElement*, the *RaptorInterfaceClass*. This element provides functions for initializing and communicating with the RAPTOR system. Furthermore it offers the necessary interfaces to the CAMeL-View simulation process. The element contains miscellaneous parameters for configuring the RAPTOR system (e.g., module number), the inputs and outputs (e.g., scaling, offset, bit-width, and address) as well as parameters for defining the sample rate.

The CAMeL-View specific library *raptorInterface.dll* was developed to provide an easy access to the RAPTOR system (cf. figure 5.4). It communicates to the generic *raptorlib.dll* and encapsulates the provided functions for their use with *CCodeElement*. Furthermore the conversion of the inputs and outputs from floating point to fixed point numbers (scaling, offset and bit-width) and vice-versa takes place in this library.

Figure 5.8 shows the simulation loop of CAMeL-View along with the most important simulation functions.

The simulation loop is divided into major and minor time steps. During the major steps the current state of the system is evaluated and all the outputs of the system are calculated. The results of the minor steps have no physical meaning, they merely calculate temporary results to transfer the continuous states from one major time step to another. Discrete states where therefore varied during the major steps. Controllers implemented on an FPGA are generally designed as discrete components. Therefore they are only evaluated during the major time steps (cf. figure5.8).

The equations of a CAMeL-model are divided into the categories of *evalND* for calculating the non-direct feedthrough equations, *evalD* for the direct feedthrough equations and *evalS* for calculating the new discrete states and the time derivatives of the continuous states. The same classification is used for the code of the *CCodeElements*. Thus two alternatives exists for the integration of the RAPTOR system into the simulation progress of CAMeL-View:

Variant 1: The calculation takes place with a delay of one sample. In this implementation the inputs are written to the RAPTOR system at the end of the *evalS* evaluation. The results of these inputs are read at the beginning of the next time step, in the *evalND* evaluation. Thus the calculation of the minor time steps can occur in parallel to the calculation on the FPGA.

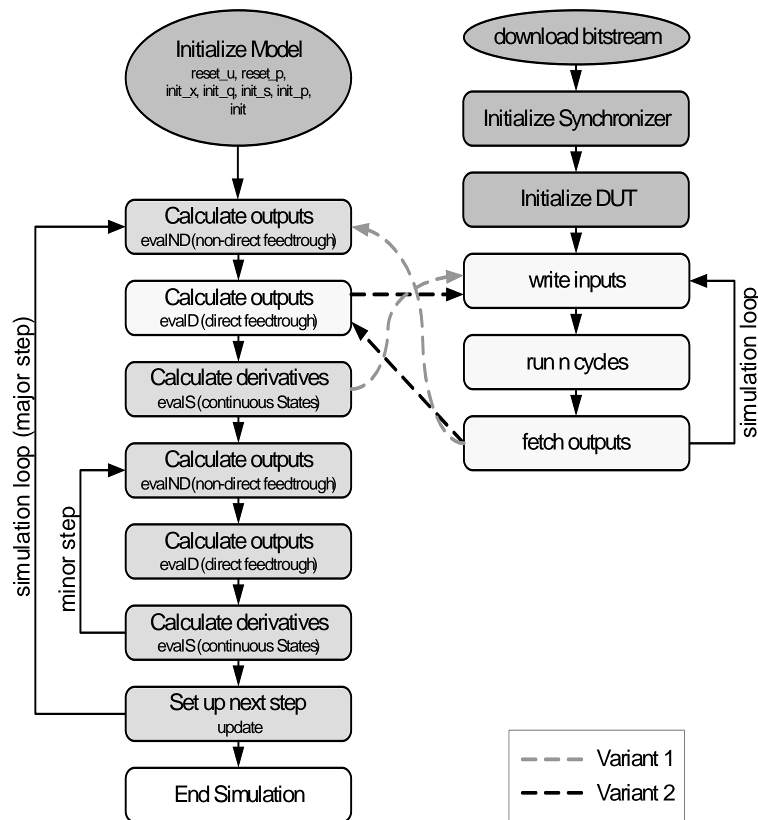


Figure 5.8: Simulation flow

Variant 2: The evaluation is integrated in CAMeL-View as a direct feedthrough function. This means, the inputs and outputs of the RAPTOR system are written/read at the same simulation time. Thus the calculation is done in zero-time. This variant was used in this work and it is depicted in figure 5.8.

5.2.3 Communication and Performance

In order to realise FPGA-in-the-Loop simulations the RAPTOR system has to be connected to a standard PC, as depicted in figure 5.9. The main board of a PC has typically a processor and a set of buses and bridges (i.e., a chipset) to interconnect peripheral components, such as memory, video cards, and external devices. The RAPTOR system uses the PCI-Bus to connect to the PC. In order to exchange data between RAPTOR and a host processor, PIO and DMA transmission methods can be used, both methods are described in the next sections.

For the experiments presented in this section, a Pentium 4 processor from Intel, with 3,0 GHz clock frequency, 1 GByte PC400 Double Data Rate (DDR) RAM are

used. The Mainboard has a 865G-Chipset, whose connection to the RAPTOR system is depicted in figure 5.9. Although the results of the experiments are specific to this setup, they can be generalised to newer computer systems.

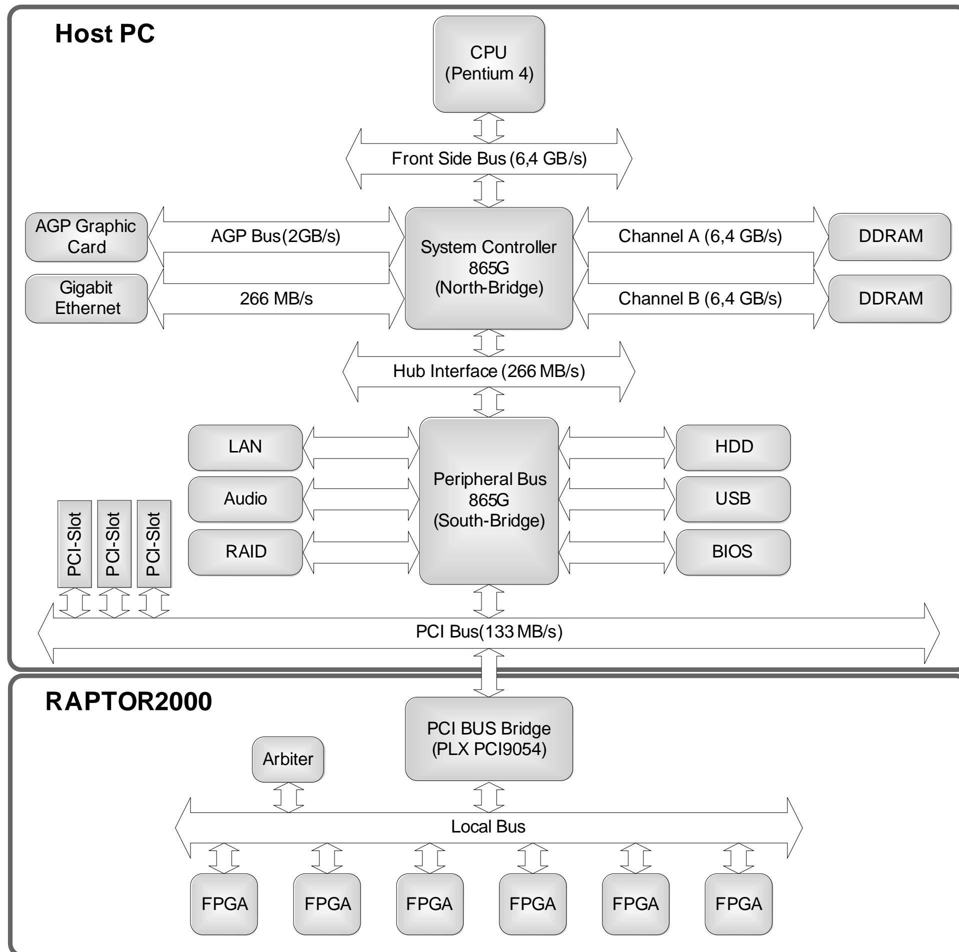


Figure 5.9: Coupling of host computer and RAPTOR. In this example a Pentium 4 with a 865G-Chipset is presented

In the following section, the different kinds of FPGA-iL simulations are presented, relating them to the choice of a transmission method.

Open-Loop vs. Close-Loop Simulations

In an open-loop simulation the DUT does not have an implicit or explicit feedback loop to the simulated environment. A typical example of an open-loop simulation is a digital filter. In contrast to this, in a close-loop simulation the DUT has a close interaction with the simulated environment. Control systems require typically

close-loop simulations, because their outputs are computed based on the state of the controlled system.

The kind of simulation has a great influence on the kind of communication (e.g., PIO or DMA) that is best suited to the FPGA-iL simulation. In a close-loop simulation, data has to be exchanged between DUT and simulation software at every integration step. Therefore, the kind of memory used to store input and outputs of the DUT, and the kind of communication has to be selected accordingly. In an open-loop simulation, the amount of data that can be sent to the DUT depends mainly on the speed of the simulation. Therefore, data can be sent to the DUT in a way that the communication overhead is reduced, e.g., burst of data can be sent at once.

PIO Communication

In PIO transmission mode, the processor loads data to be transferred to one of its registers, before the data is actually sent through the Front-Side-Bus, the PCI-Bus and finally the Local-Bus to a DUT running on the FPGA (cf. figure 5.9). correspondingly, the data generated by the DUT (i.e., control signals) are sent from the registers of the RAPTOR to registers of the processor by a read command of the host processor. This transmission mode blocks the processor during the data transfer.

DMA Communication

Direct memory access (DMA) is a transmission mode where a peripheral device transfers information directly to or from memory, without the processor being required to perform the transaction. This has the advantage that the processor can execute other tasks while the transfer is taking place.

The PCI-Bridge of the RAPTOR system is able to operate as a DMA controller with two independent channels. This Bridge is able to execute DMA-transfers to the PCI-Bus as well as to the Local-Bus. The initialisation of a DMA transfer plus the arbitration of the PCI- and Local-Bus makes DMA worth using instead of PIO only if the amount of data to be transferred is above a certain threshold-value, which is explored in the following section.

Simulation Performance

To estimate the maximum performance of the presented framework, several pre- and post-processing steps need to be considered, which have to be conducted in every simulation cycle. A maximum for the simulation frequency F_{sim} is given by

$$F_{sim} = \frac{1}{T_{sw2hw} + T_{send} + T_{run} + T_{receive} + T_{hw2sw}} \tag{5.1}$$

where T_{sw2hw} and T_{hw2sw} are the conversion-times from a simulator-internal to a hardware-specific number representation and vice versa, T_{send} and $T_{receive}$ are the transfer-times from the main memory of the host to the prototyping system and back, and T_{run} is the latency of the design itself. All values except T_{run} depend on the interface between the simulation environment and the hardware design, while T_{run} depends on the speed of the hardware design only. The delay of the simulator, which may be running a test-bench, or a data logger, or similar, can not be estimated here, because it depends on the complexity of the simulation. As the interface latency is highly dependent on the underlying host architecture, the following measurements are presented as an example for transfer and conversion times.

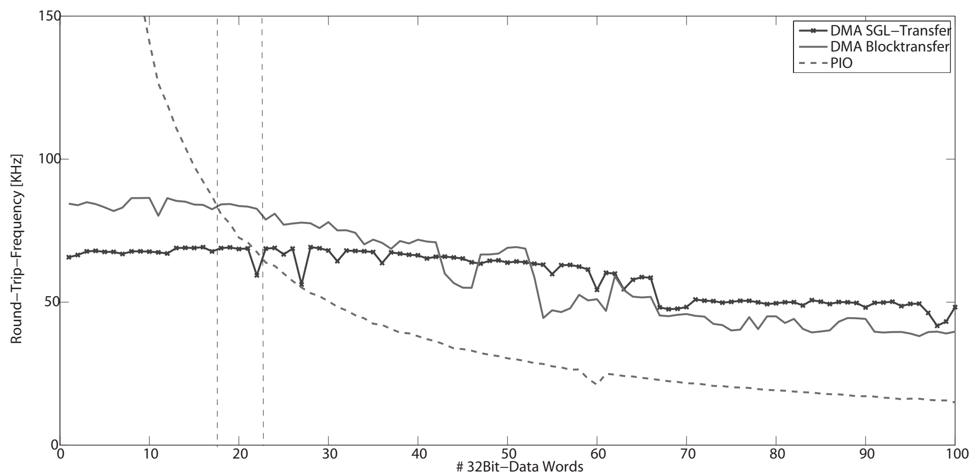


Figure 5.10: Maximum simulation frequency for a given number of input/output pairs

In figure 5.10 the simulation frequencies for different transfer modes against the number of I/O-pairs (i.e., combination of one input and one output) are shown. I/O pairs are used, because the transfer times are different between the write and read transfer, and assuming the same number for inputs and outputs is a good approximation to real scenarios. It can easily be seen that the transfer mode should be selected according to the number of I/Os, since PIO is faster for up to 18 I/O-pairs. As from 22 I/O-pairs, DMA block transfers are faster.

Communication Optimization

In the simulation flow as described above, all I/O data have to be transferred at every clock cycle, resulting in redundant I/O operations when data has not changed. To decrease this overhead, two further concepts were integrated in HiLDE: *Event based communication* and *Transactors*:

- **Event based communication:** to reduce the number of redundant I/O operations, only data that actually changes has to be transferred. While this is straightforward to be implemented in software (Simulink provides appropriate functions), the hardware wrapper has to be extended. The register of every output port is extended with a mechanism to detect changes at the output. For n_o output ports an additional register with n_o bits stores the results of these detectors, and thus indicates which values must be read by the host computer. The number of additional read operations to retrieve this information is dependent on the bit-width of the bus to the host computer, resulting in an overall number of read accesses $\tilde{n}_r$:

$$\tilde{n}_r = \Delta(out) + \left\lceil \frac{n_o}{wordwidth} \right\rceil \quad (5.2)$$

  where $\Delta(out)$ is the number of output ports with a new value. Given that n_r denotes the number of read operations in the standard HiLDE wrapper, the benefit $n_r \tilde{n}_r$ is dependent on the relation of I/Os with regularly changing values to the overall number of I/Os in the DUT. In general DUTs with irregularly changing I/Os will benefit from this technique.
- **Transactors:** whenever the sequence of events (value changes) is predefined, such as in communication protocols, the number of I/O operations can be reduced even further by implementing adaptors for the simulation and for the FPGA. The amount of savings here is dependent on the complexity of the protocol: instead of transferring all control-signals or control-signal changes, the adaptors detect protocol activity and transfer only the necessary data, such as address and data, the actual protocol handling is processed in the adaptors in the simulation environment and in the FPGA. While the functionality of the HiL simulation is not affected by this method, the amount of I/O operations for a protocol as described in [Kal02] can be reduced by over 90%.

5.2.4 HilDE Tool Flow

As presented in sections 5.2.1 and 5.2.2, HiLDE is composed by standardised hardware and software components. Therefore, the integration process of a DUT into the HiLDE framework can be automated. This has the advantage of accelerating the verification process, and reduce the possibility of introducing errors. The automatic integration of a DUT into the HiLDE framework is carried out by vMAGIC, which is presented in the next section, followed by the complete tool flow required for a HiLDE simulation.

vMAGIC

vMAGIC (VHDL Manipulation and Generation Interface) has been developed to provide a basis for all kinds of code generators by implementing three important basic tasks [16, 11, 17]:

- Reading an existing hardware description of a DUT
- Manipulation of existing hardware description
- Writing of manipulated and/or generated hardware description

Using vMAGIC accelerates design processes whenever uniform tasks can be automated and reused many times, which is the case of the task of generating a HiLDE hardware wrapper (cf. section 5.2.1).

The vMAGIC API is a Java library compatible with run-time environments 1.5 and later. Therefore, it is platform independent and usable in command line tools, graphical user interfaces and scripts. The basic functionality is described in the following points:

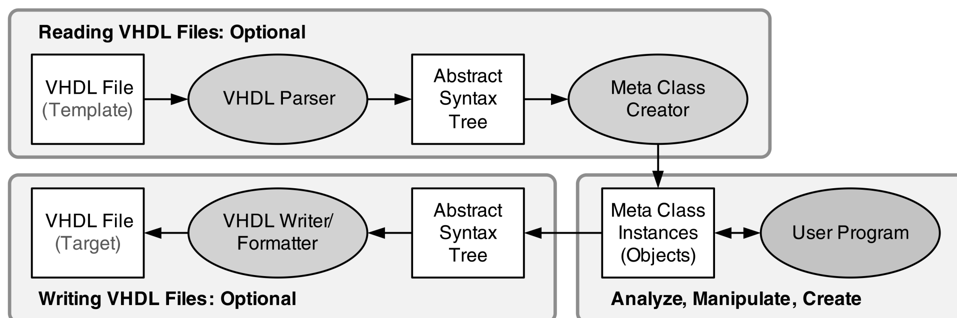


Figure 5.11: vMAGIC Designflow, reading and writing VHDL is optional. As such a vMAGIC application can be a pure VHDL generator or analyzer

- **Parser:** vMAGIC implements a VHDL'93 compliant parser to transform VHDL code into a more convenient internal representation called AST (Abstract Syntax Tree). An AST in general contains all information from the code, while redundancy (parenthesis, semicolons and so on are implicitly included in the tree structure) is removed; this AST however is shaped in a way optimally suited for the manipulations described next.

  The vMAGIC parser was generated using ANTLR 3.1 [Par95], a powerful parser generator; parsers generated with ANTLR support certain error recovery strategies. This implies that the vMAGIC parser can correct certain syntactical errors while parsing VHDL source code.

- **Modification and generation of code:** The AST by itself is a tree structure, which is not well suited for human interaction. To hide this structure behind a simple API, a set of so-called meta-classes was defined. Meta-classes combine the functionality to generate or modify specific VHDL constructs and the

knowledge how to interact with the AST, such that the developer is using a homogenous API with intuitive functions. E.g., `Signal.getIdentifier()` returns the identifier of a signal, `new Process()` creates a new VHDL process.

Objects of meta-classes are created either by the user, defining a VHDL design from scratch, or using a VHDL template. The template is parsed into an AST, which is then parsed by a tree parser generating the meta-objects and discarding the original tree. This approach is, again assuming that the tree grammar is correct, another means of ensuring that the generated code is correct regardless of coding style or context. The meta-objects implicitly define a descendible tree structure, beginning with a `VhdlFile` object with members for `Entity` and `Architecture` objects and so on. User programs work on this meta-tree rather than on the AST, allowing for intuitive software development for hardware generation or analysis purposes.

There are two different levels of abstraction represented by meta-classes: the so-called low-level classes represent basic VHDL constructs such as signal declarations or processes; the high-level meta-classes combine several low-level classes such as to create complex functionality like registers or state machines. The use of high-level classes implies a higher level of abstraction and therefore an improved coding speed.

- **VHDL Writer:** To generate VHDL code from an AST, a VHDL Writer based on ANTLR's StringTemplate system was developed. Again, a tree parser is used to analyse the tree and templates are used to generate VHDL code constructs. These templates are defined in a single text file in a very simple format, such that the developers preference in coding style (e.g., the use of lower case or upper case letters for keywords, or using optional identifiers at the end of a process or entity) are implementable by changing this text file.

The vMAGIC design flow, as depicted in fig. 5.11, follows the three steps as described above. The generation of the HiLDE hardware wrapper is a very uniform procedure, usually varying only in the number and width of I/Os, or in the transfer mode as described above. The following steps are completed by a Java program utilizing vMAGIC:

1. Parse the DUT and a special template file
2. Create an instance of the host communication bus and connect the Synchronizer to the bus
3. Declare and instantiate the DUT in the template
4. Create registers for ever I/O port and connect them to the DUT instance
5. Add all registers to the bus
6. Generate configuration files for different simulators (currently Simulink, and CaMEL-View)

While the manual (error prone) implementation of the wrapper can take hours, the HiLDE Wrapper Generator takes seconds at most. To setup a HiLDE simulation, there are two possible tool flows, depending on weather the target simulation software is Matlab/Simulink or CaMEL-View. Both flows are described in the following sections.

Simulink Tool Flow

The design flow presented in this work supports any hardware VHDL-based description. The tool-flow for a Matlab/Simulink based FPGA-iL simulation is presented in figure 5.12.

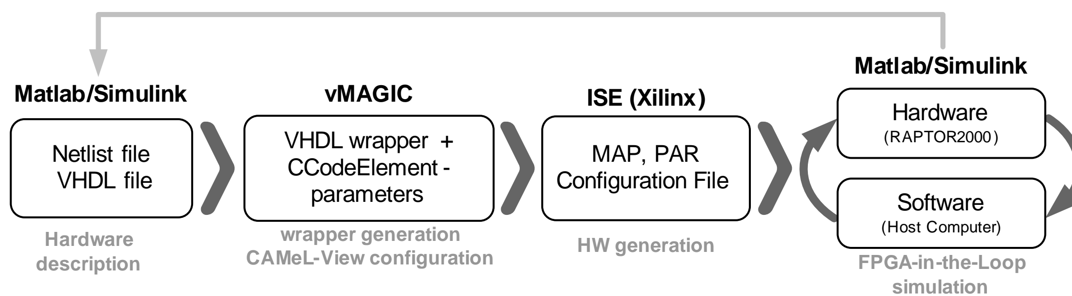


Figure 5.12: Toolflow for HiL simulations with Matlab/Simulink

Typically, a SiL simulation is first done using a hardware description of the DUT. After obtaining a satisfactory simulation, a vMAGIC application is used to integrate the DUT into the Matlab/Simulink framework, as described in the previous section. The generated VHDL file can then be synthesised using vendor specific tools (e.g., ISE from Xilinx) to generate an FPGA configuration file. Configuration files for the Matlab/Simulink simulation interfaces are also generated by vMAGIC. Using these files, an FPGA-iL simulation can be setup in Matlab/Simulink.

The simulations are performed as usual. However, the designer can now realise whether the controller, which is executed on an FPGA module of the RAPTOR system, works as expected. In this stage more intensive tests can be conducted. Since the structure of the controller has already been designed and tested, the next step is an intensive test of its parameters or its response to different operative regions. This process is greatly accelerated by HiL simulations, besides the enhanced reliability provided by this kind of simulations.

CAMeL-View Tool Flow

FPGA-iL simulations with CAMeL-View as described in this thesis includes the use of several software tools (cf. figure 5.13). Starting with a description of the plant (e.g., a multi body system) the control engineering components (e.g., controller, observer)

are developed by the designer using CAMeL-View. After the parametrisation of these components, they are exported to Matlab/Simulink along with the model of the plant. In Matlab/Simulink the DUT can be translated to a hardware description after a digitalisation process. This step is done by using System Generator from Xilinx, which automatically generates structural hardware descriptions (e.g., netlists or VHDL) from a very high-level representation. This VHDL or netlist file, containing the implementation together with the corresponding interface description, serves as an input to HiLDE: a hardware wrapper is generated along with the appropriate CCodeElements and parameters for interfacing CAMeL-View. The design is now ready for hardware synthesis through Xilinx ISE, resulting in an FPGA-configuration bitstream containing the controller implementation as well as the appropriate interface for communication with the simulation environment. CAMeL-View can now start an FPGA-iL Simulation according to the steps in figure 5.8.

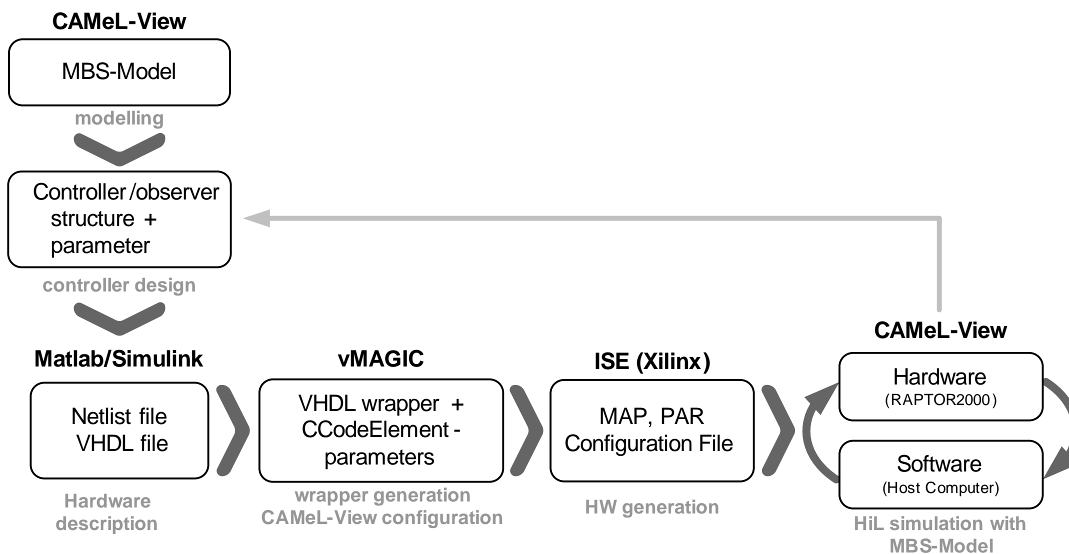


Figure 5.13: Toolflow for FPGA-in-the-Loop simulations with CAMeL-View

5.2.5 Implementation Examples

In this section the use of HiLDE is exemplified with various realisation examples, showing not only a successfully verification process, but also the speedup with respect to a SiL simulation. For all examples presented, SiL and HiLDE simulations used the same test-bench (i.e., simulated environment of the DUT).

PID-based Speed Controller

As a first example, a Proportional-Integral-Derivative (PID) algorithm to control the speed of brushed-DC motors for robotic applications is presented. The control task is to regulate the speed of a DC motor through its input voltage. A classical parallel PID is realised using a Simpson numerical integration rule, as presented in section 3.5.1.

The controller was implemented using System Generator from Xilinx. The design uses one output and nine input ports. All parameters of the design were defined as import ports (cf. figure 3.19), allowing complex tests (e.g., parameter adaptation). The test bed of the controller consists of a model of a DC motor, and some scopes. The PID controller runs on a module of the RAPTOR system, based on a Virtex II FPGA. The hardware implementation requires 1063 LUTs, and 1043 FFs, for a total of 693 Slices ($\approx$ 3%).

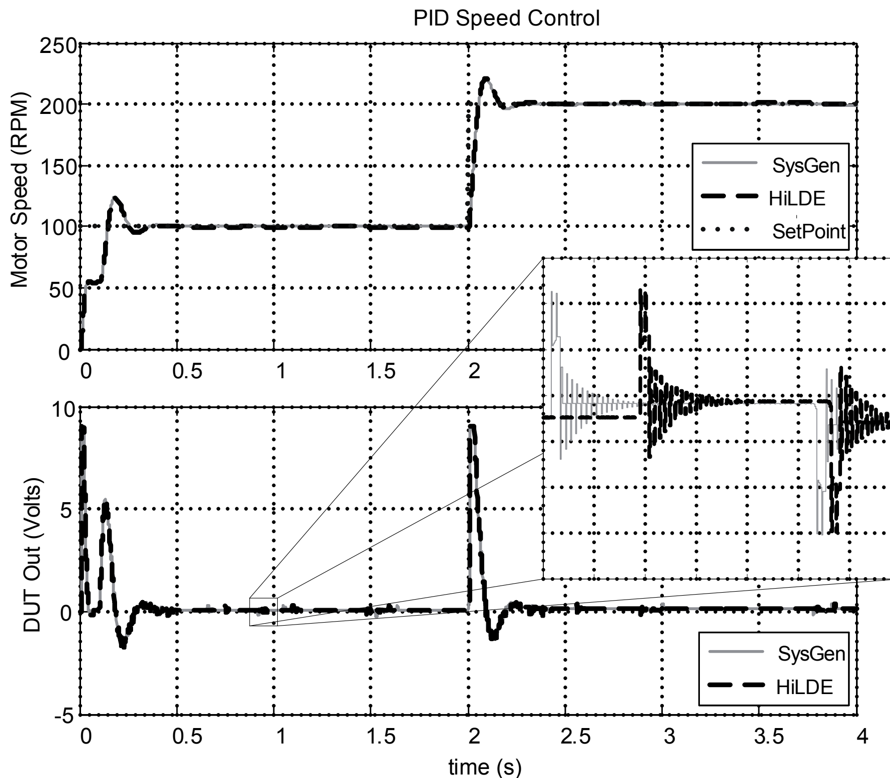


Figure 5.14: Simulation results of a PID-based speed control: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 50 ms

Simulation results form both HiLDE and System Generator are presented in figure 5.14. The sampling frequency was set to 1KHz. The simulated time was 4 seconds. SiL simulation lasted in average 3.77 seconds, whereas using HiLDE the simulation lasted in average 1.49 seconds, resulting in an average speedup of 2.53. This speedup is

relatively low because of the small size of the design and the communication overhead (PIO communication was used). Verification of the DUT was done, showing only small differences compared to the SiL simulation, as can be seen in figure 5.14, where the set point of the controller was changed from 100 to 200 RPMs after two seconds (upper part of the figure). The small differences between HiLDE and SiL simulations do not affect the function of the controller, as can be seen in the speed response of the DC motor.

State-Feedback Controller for an Inverted Pendulum

The controller to balance the inverted pendulum, presented in section 4.4.3 was verified using HiLDE. The test bench of the controller consisted of a state-space model of the pendulum-cart system and blocks for the simulation of the power electronics, as well as scopes. The State-Feedback controller is implemented using a Virtex II module in the RAPTOR System.

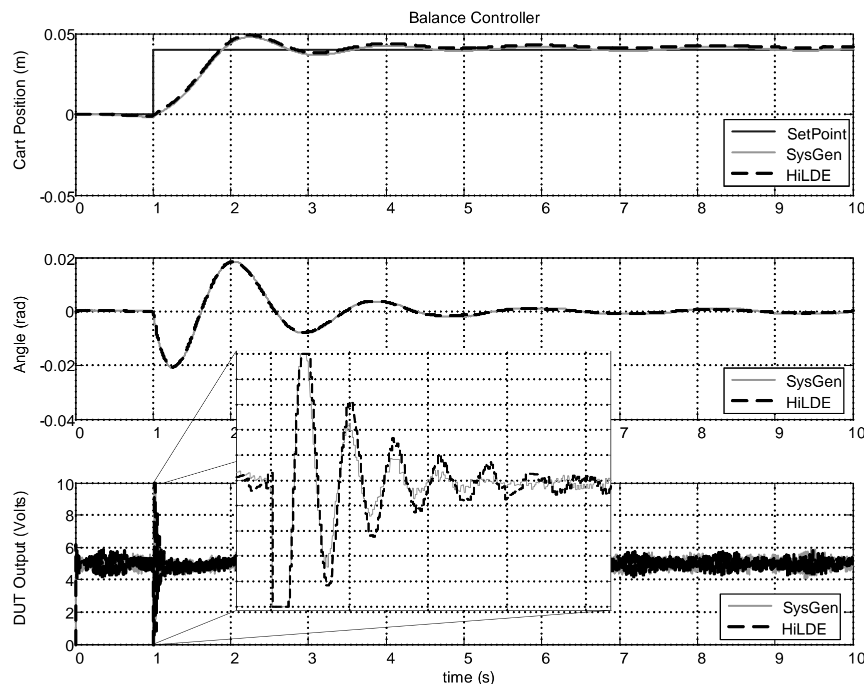


Figure 5.15: Simulation results of a balance controller for an inverted pendulum: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 50 ms

The system has three inputs, the angle of the pendulum, the position of the cart, and the target position. Furthermore, the controller has one output, the new position of the cart. The sampling period of the controller is $10\mu\ s$. System Generator was used to

implement the controller, which required 3 BRAMs (≈ 2%), 1892 LUTs, 285 FFs, for a total of 1033 Slices (≈ 4%).

The model of the pendulum is linearised at the operative point in the upright position. To test the balance controller, the target position of the cart is set 4 cm to the right. The task of the controller is to move the pendulum to the target position while keeping the pendulum balanced. Results of a SiL and HiLDE simulations are shown in figure 5.15. The first plot shows the position of the cart, the angle of the pendulum is presented in the second plot, and the output of the controller is shown in the last plot. As can be seen, the control goal was achieved. The HiLDE controller worked just as well as the simulated design. However, small differences were found, as presented in figure 5.15. These differences do not affect greatly the function of the controller. However, this represents an undesired behaviour, which can be spotted out by performing HiLDE simulations.

Ten seconds of simulation using the System Generator blocks lasted in average 5.59 seconds, while using HiLDE the simulation time was reduced in average to 0.17 seconds. This represents a speedup of 32.15 times. The improved speedup is because of the larger complexity of the DUT in comparison with the PID controller, and the lower communication overhead.

PLL-based Controller for a Piezo-Actuator

This implementation example was presented in [9], and was done in collaboration with the Mechatronik und Dynamik group of the university of Paderborn.

Piezoelectric transducers are often used as ultrasonic actuators. To gain the highest possible vibration amplitude, these actuators are driven in their resonant frequency. For the realisation of this operation mode, if the system is not strongly damped, a closed loop control is necessary. The most common control approach it the Phase-Looked-Loop (PLL) control, which holds the phase between current or velocity and voltage of the piezo actuator on a zero value. A block-diagram of typical PLL-based control configuration is shown in figure 5.16.

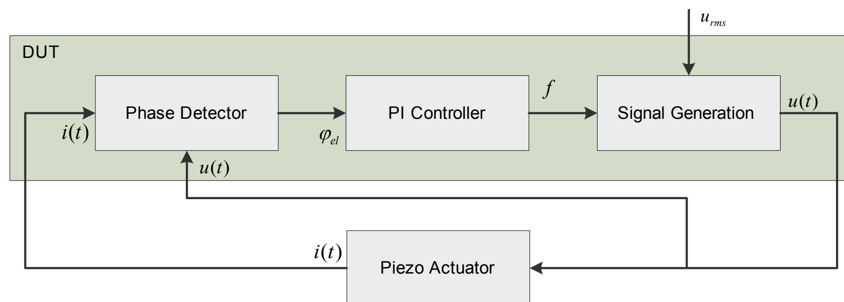


Figure 5.16: Standard configuration of a PLL-based controller for piezo actuators

The PLL has three major components: the phase detection, the filter and the signal generation. The filter can be a PI controller, the signal can be generated by a Direct Digital Synthesis (DDS) algorithm. The most demanding part of this control approach is the phase detection, because of the need of accuracy and speed. For the operation of the transducer, only the spectral component with the driving frequency is of interest for the controller. For the example presented in this section a Fast Fourier Transform (FFT) approach is used to realise phase detection. Simulation results of the realised controller are presented in figure 5.17.

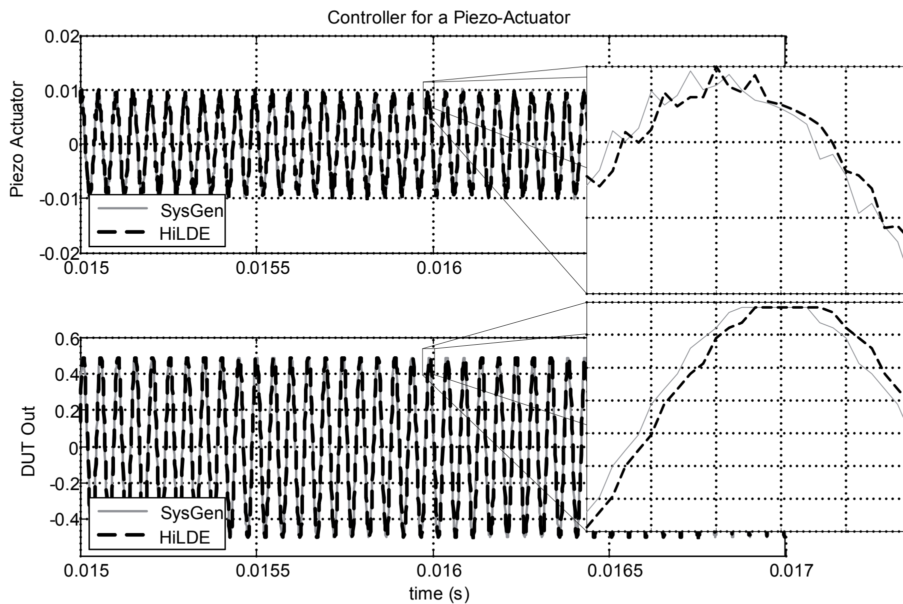


Figure 5.17: Simulation results of a phase controller for a piezo-actuator: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 10 μs

The DUT was realised using System Generator, using five input ports, and one output. The design requires 28 embedded multipliers ($\approx$ 23%), 10 BRAMs ($\approx$ 8%), and 8033 Slices ($\approx$ 34%). A SiL simulation of 1 second requires in average 2.46 seconds, whereas a HiLDE simulation takes in average 0.09 seconds. This represents a speedup of 27.33 when using a HiLDE simulation.

Observer for an Active Suspension Testbed

This implementation example was presented in [10], and was done in collaboration with the Control Engineering and Mechatronics group of the university of Paderborn.

An observer for the half-vehicle testbed for a suspension tilt system of a railway system, the RailCab [NBP], is presented. The active suspension system mainly consists

of the body mass that represents the coach of the vehicle, two actuator modules and the carriage frame. Each actuator module mounted under the body consists of three hydraulic cylinders that are coupled over a lever kinematics to a glass-fiber reinforced plastic spring (GRP spring), so that the bases of the spring can be displaced actively to apply the necessary forces between the body and the carriage. The carriage frame can be excited by three hydraulic cylinders to simulate disturbances in the rail track. The forces necessary for the damping are computed by the control and transferred to the body by displacing the spring bases via hydraulic cylinders [Sch06]. Every hydraulic cylinder includes an integrated position sensor. The relative displacement between the body and the carriage is measured by two displacement sensors. Furthermore, acceleration sensors on the body mass are used to measure the absolute motions of the coach.

A model of the testbed has been developed to test the observer in a virtual environment. The testbed is modelled as a multi-body-system using CAMeL-View, but also includes hydraulic and control engineering components [Gei05]. The model contains 7 masses: the coach-body mass, as well as 3 masses for the kinematics of each actuator module. Each lever kinematic is mounted onto three hydraulic cylinders.

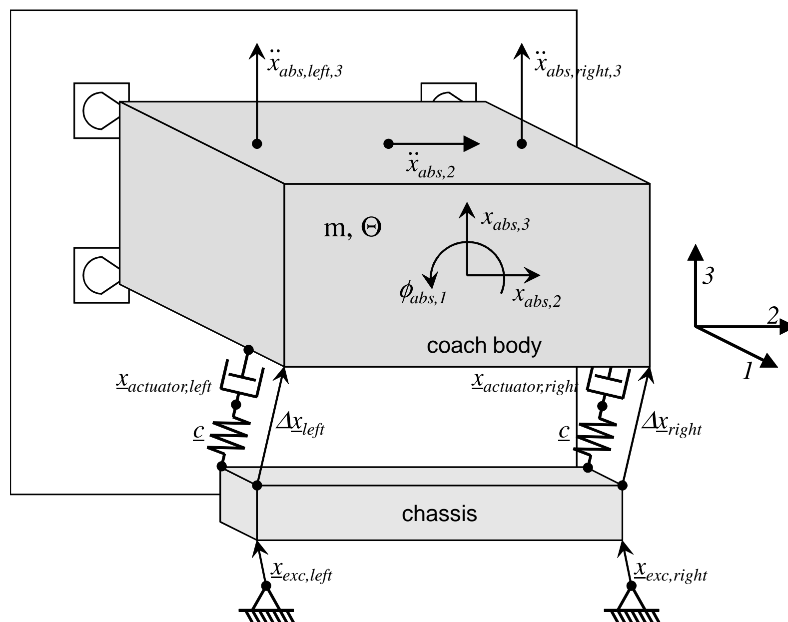


Figure 5.18: Mechanical model of the active suspension testbed [10]

The linear state-space representation was derived by linearisation of the complex testbed model. A subsequent model reduction reduces the linear model to the significant system-states. The resulting state-space model is of order 15. In figure 5.18 a mechanical analogous model of the testbed is depicted. This sketch shows the significant values for the observer-design, which are described in the following.

The absolute position of the coach-body is given by the horizontal and vertical position x_{abs_2} and x_{abs_3} as well as the rolling angle ϕ_{abs_1}. These values form the desired observer output vector $\underline{\hat{y}}_{obs}$. The coach-body acceleration $\ddot{x}_{abs,right_3}$, $\ddot{x}_{abs,left_3}$, $\ddot{x}_{abs_2}$ as well as the relative displacement between the chassis and the coach-body $\Delta\underline{x}_{right}$, $\Delta\underline{x}_{left}$ are measured by sensors. These measurements build up the output vector of the plant model $\underline{y}$.

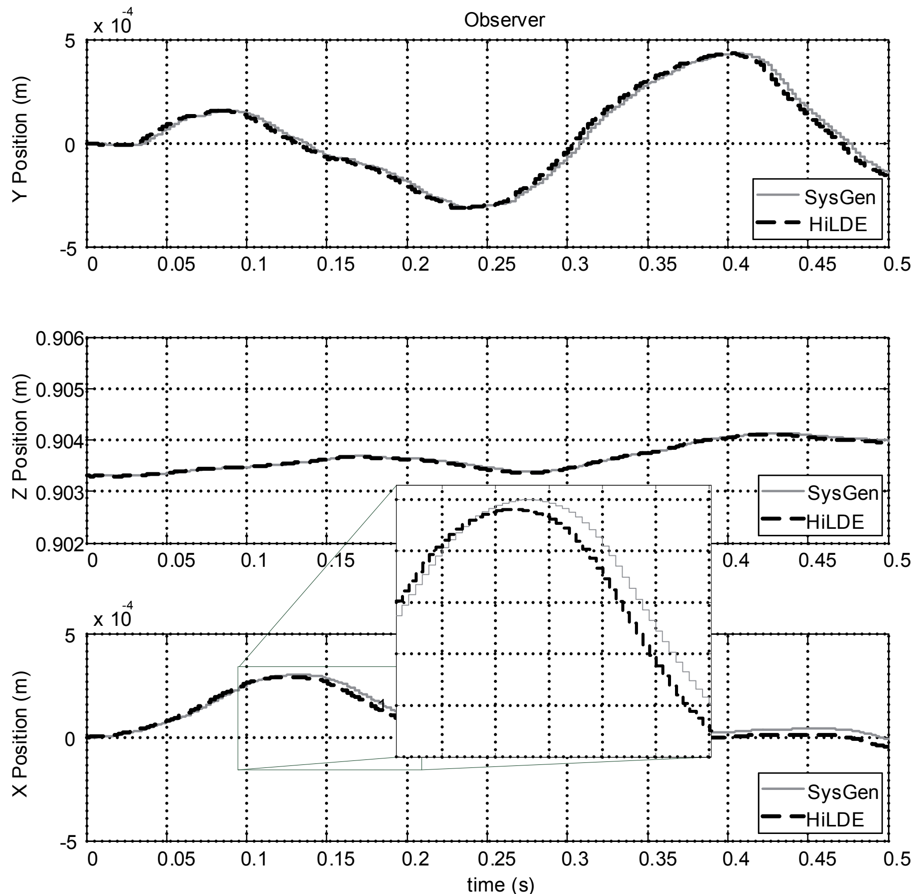


Figure 5.19: Simulation results of a state observer for an Active Suspension Testbed: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 20 ms

After a satisfactory simulation in CAMeL-View, the observer is exported to Matlab/Simulink together with a model of the testbed, where a hardware implementation of the observer is done using System Generator from Xilinx. Simulation results of an improved version of the observer presented in [10] are shown in figure 5.19. For this implementation, matrix multiplications are split into vector multiplications, which are serialised so that only one embedded multiplier is used for a whole matrix multiplication. Consequently, the latency of the design (1224 clock cycles) is much

longer than a fully parallel implementation. This implementation requires only 2 embedded multipliers ($< 1\%$), and 8081 Slices ($\approx 35\%$), in contrast to the 85 embedded BRAMs ($\approx 70\%$), 40 embedded 18x18 multipliers ($\approx 40\%$), and 15,572 Slices ($\approx 67\%$) of the DUT realisation presented in [10]. A SiL simulation of the DUT takes in average 1049 seconds, while a HiLDE simulation takes only 0.59 seconds. This difference comes from the fact, that for each integration step, the HiLDE simulation only requires to send input values to the DUT and read the output values, whereas in a SiL simulation, all elements of the DUT have to be simulated at a faster simulation frequency ($Period_{Observer} = Period_{Simulation}/Latency_{Observer}$). Results presented in figure 5.19 show a satisfactory validation of the design.

Design	Slices	I/Os	S. Rate [μs]	Sim. Time [s]	Duration (Sim) [s]	Duration (HIL) [s]	Speedup
PID Controller	693	6/1	100	4	1.49	3.77	2.53
State-Feedback	1033	3/1	100	10	5.59	0.17	32.15
Torque Control	2129	6/6	0.03	0.01	74.99	25.87	2.89
Piezo Controller	8033	5/1	1	0.006	2.46	0.09	27.33
Observer	8081	10/18	2500	0.5	1049	0.595	1763

Table 5.1: Implementation Examples

Table 5.1, presents a summary of the examples presented in this section. The torque controller is presented in section 4.4.4. All measurements were done with a standard PC equipped with an Intel Core Duo CPU (E6750), running at 2.66 GHz, with 2 GB of RAM at 2.67 GHz. The operative system is Windows XP Professional Version 2002, Service Pack 3. Simulations were done using Matlab 2006b (7.3.0.267), Simulink version 6.5 (R2006b). A Virtex II (2v4000ff1152-4) device from Xilinx served as hardware platform for all presented examples.

5.3 HiLDEGART: HiL Design Environment for Guided Active Real-Time Test

After performing a cycle-accurate functional design verification with a simulated environment, the DUT can be tested as a prototype using the real plant. At this point

of the design flow, timing verification of the DUT can be performed. Furthermore, parameters of the controller can be tested and adjusted, for which mechanism to load and synchronously execute parameter changes is required. Moreover, monitoring inputs and outputs of the DUT is required, to observe the effects of parameter adjustments, or simply for debugging. Additionally, measurements can be performed without the need of external measurement devices (e.g., logic analysers) [13]. To realise this, one approach is to use real-time verification tools such as ChipScope from Xilinx [Xil08b]. However, aside from its limited allowable monitoring time, this kind of tools do not permit an interaction with a DUT. Another approach are logic analysers, but they are expensive and it is very time consuming to set up a test environment. For this purpose, HiLDEGART (HiLDE for Guided Active Real-Time Test) was developed. Our approach can be implemented with a standard PC, and allows the automatic integration of a design to the HiLDEGART framework. In the following section, the concept and realisation of HiLDEGART is presented, focusing on the communication between the DUT and the host computer.

5.3.1 Hardware Components

Figure 5.20 shows the basic concept of the presented Hardware-in-the-Loop (HiL) framework. The design under test (DUT), a controller, is implemented on an FPGA. The testbed consists of a plant to be controlled and an analog/digital interface. There are three main components surrounding a DUT to be tested with HiLDEGART:

Bus-Interface to Host allows the communication between the host PC and the DUT. It works very similar to the interface described in section 5.2.1. The main difference is the use of embedded FIFOs and external SDRAM memory to assure meeting the required sampling rates, as explained below.

Resampling Control the user has the choice to select a specific sampling rate for each port of a DUT using this module. There are two kinds of sampling mechanisms, real-time and off-line sampling. Real-time sampling enables the visualisation of the selected signals at run-time. The amount of signals that can be visualised in real-time is limited by external factors (i.e., I/O-bandwidth). For off-line sampling, an SDRAM memory directly attached to the FPGA is used for buffering the data, allowing for very high sampling rates. The buffered data, as opposed to real-time data, is transferred to the host for visualisation after the simulation has finished.

Event Monitoring allows basic compare operations to generate events. Those events may be combined by boolean operators to form conditions like $(A > \tilde{A}) \wedge \neg(B = \tilde{B})$, where A and B are the ports and $\tilde{A}$ and $\tilde{B}$ are values defined by the user at run time. With the resulting events, either the changing of the

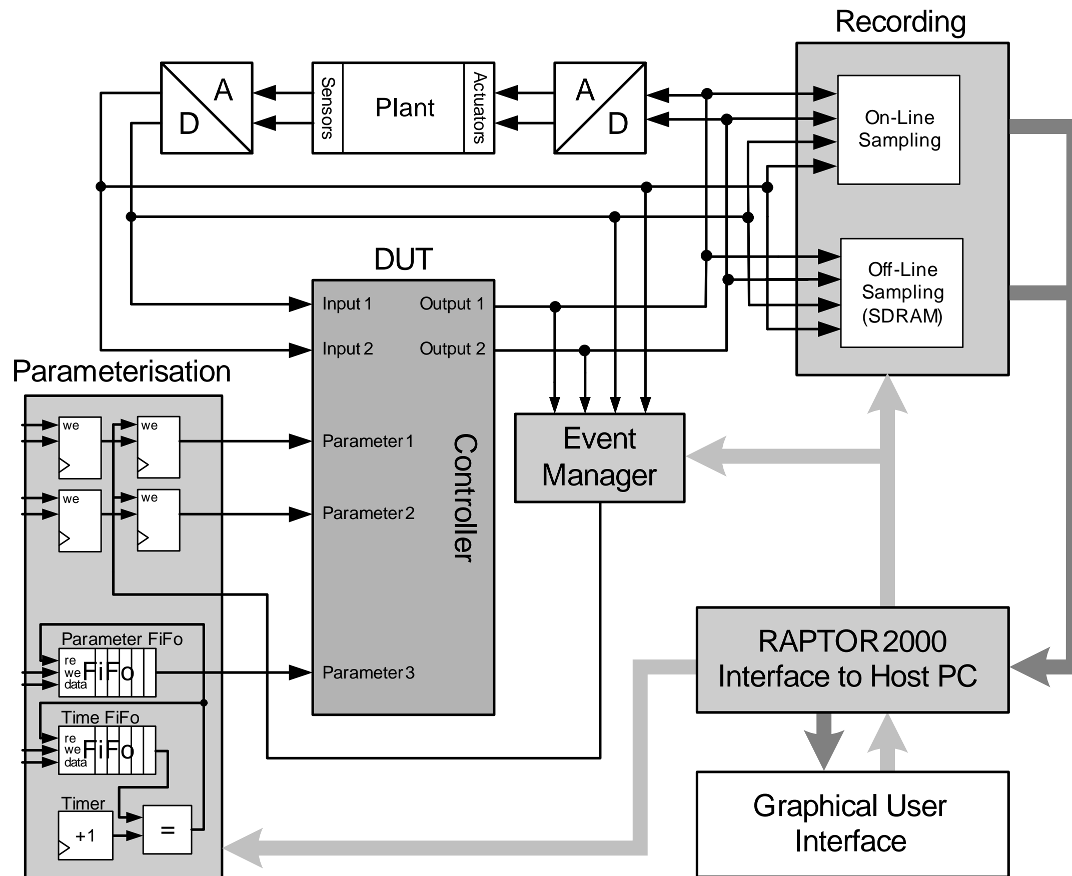


Figure 5.20: Structure of a real-time FPGA-in-the-Loop scenario utilizing HiLDEGART

sampling rates or the changing of the parameters of the design can be triggered in real-time. Additionally, the events can be used to start or stop recording.

5.3.2 Software Components

The presentation of I/O values and all configuration tasks are controlled via a GUI, which has been implemented using Trolltech's platform-independent programming environment Qt [Tro] in combination with QWT [Rat]. The project files describe the hardware interface including addresses and number representations (e.g., fix point/binary configuration). They are generated by a vMAGIC application based on user annotations in the VHDL code. The GUI is automatically generated based on the interface description, including graphs, LCD-like displays for current values, and input boxes for parameters, as can be seen in figure 5.21.

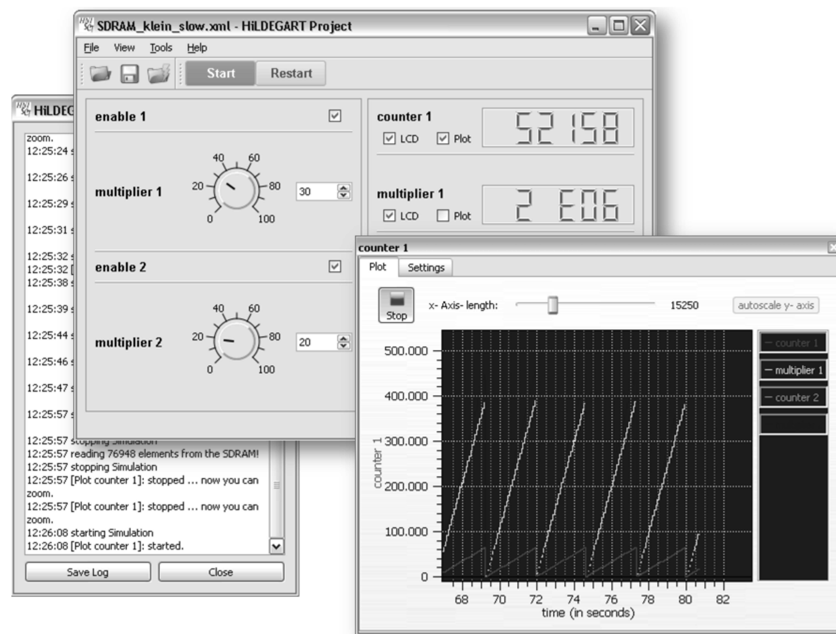


Figure 5.21: Main-, Log-, and Plot-Window of HiLDEGART. The GUI is generated from an XML file generated by a vMAGIC application

While visualisation of the DUT (i.e., internal states, input and output ports) in 2D graphs can be achieved directly within HiLDEGART, more sophisticated visualisations are created by external programs using a standardised interface. E.g., model related data can be visualised using an Augmented Reality (AR) tool, embedding, e.g., physical properties of the model into a video image of the real system [16]. AR is a human-computer-interface, which augments the perception of reality with multi-modal computer-generated information [Azu97]. This information can be, e.g., texts, 3D models or other annotations. They are shown in the field of view of the user with a spatial relationship to a real object. In order to use AR, a special viewing device is necessary. A common viewing device is the so-called head mounted display (HMD). There are several meaningful ways to use these AR techniques in the HiLDEGART environment, such as displaying the state of a controller, the state of a reconfigurable process or visualising torques and forces *directly* where they apply (cf. section 5.3.4). A TCP/IP connection is used for communication between HiLDEGART and the AR software.

A visual impression of an augmented view can be seen in figure 5.22. The AR mark is recognised by the software, which displays corresponding annotations in form of 3D animations. In the example shown in figure 5.22, an inverted pendulum is presented (cf. section 4.4.3 and 5.3.4), showing the state of the SoC used to control the system.

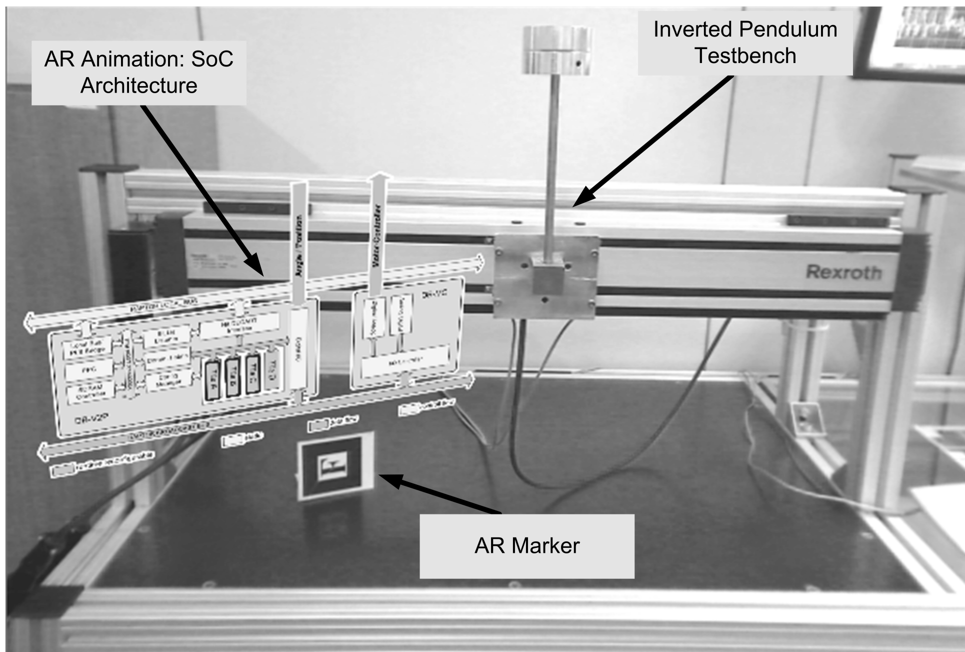


Figure 5.22: View of a HiLDEGART test using AR annotations

The automatic generation of the HiLDEGART hardware wrapper using vMAGIC is similar to the automatic generation of the HiLDE hardware wrapper. This process is described in the next section.

5.3.3 HiLDEGART Tool Flow

Generating the hardware wrappers for HiLDE and HiLDEGART is an application of vMAGIC, as depicted in figure 5.23. The starting point of the flow is a VHDL file containing the DUT's entity definition (if no internal signals should be monitored the DUT itself can be described in any HDL), which is then analysed by vMAGIC. The user program generates DUT-specific wrappers according to the specifications described in section 5.3.1 and generates the configuration files for HiLDEGART. These configuration files contain information regarding hardware addresses, sampling rates and number formats (e.g., fix-point position). As the number formats and sampling rates can not be deduced from the hardware interface, those are supplied via vMAGIC-tags in the source code or directly in the GUI. This completes the vMAGIC specific part; after this, the wrapper and design files have to be synthesised using vendor specific tools. After the FPGA bitstream has been generated, the HiLDEGART project is configured using the configuration files and the monitoring can be started.

An extended design flow for dynamically and partially reconfigurable devices is presented in [6], where Xilinx System Generator is used as design entry tool and then

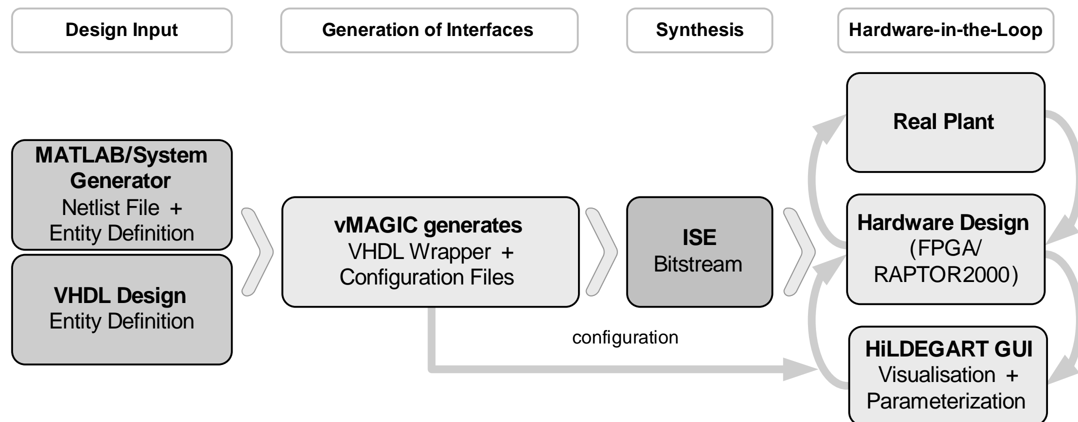


Figure 5.23: Tool flow for a real-time design verification using HiLDEGART

the structural descriptions of the controllers is adapted to our system architecture. A wrapper, containing the required interfaces, is used to integrate the controllers into the target architecture. Netlists and partial bitstreams are generated automatically. The supervisory entities, which are realised in software, are linked to the hardware modules using the Embedded Development Kit from Xilinx. Finally, the design can be tested with the real plant.

5.3.4 HiLDEGART Implementation Examples

Two of the implementation examples presented in the previous chapter are used here to demonstrate the use of the HiLDEGART framework to the design of FPGA-based controllers, including control-systems where dynamic hardware reconfiguration is used (cf. [6, 16].

Torque Control of a Synchronous AC Motor

This example was presented in section 4.4.4. In this section further measurements with HiLDEGART, which allow the verification of controllers, are presented.

When using dynamic reconfiguration to exchange controllers at run time it is relevant to make sure that all controllers work individually. HiLDEGART can be used for this purpose, allowing the monitoring and parametrisation of all controllers. An example of this are the measurement with the real motor presented in figure 5.24. The figure shows the q-current step response of a permanent magnet servo motor using three different controllers:

- FOC: The control output is only driven by the PI-Controller and the dynamic of the control loop is determined by the PI-Controller.

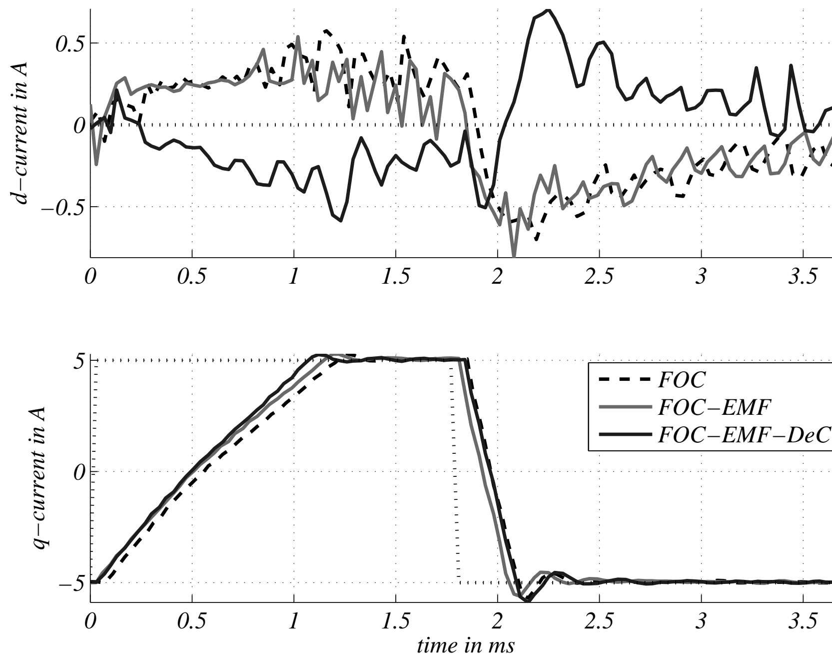


Figure 5.24: q-current step response at 3000 rpm (Test bed: Controller at FPGA, Motor as EUT)

- FOC with Back-EMF compensation: The main part of the control output is generated by the Back-EMF compensation. The dynamics of the control loop improved (compared to FOC).
- FOC with Back-EMF compensation and decoupling: This control structure gives almost the same results as the prior controller. Only in case of transient currents and higher electrical frequencies the additional effort has an effect on the behavior of the control loop.

The results show a proper control behaviour of all three controllers. The plots are created by using HiLDEGART.

Controllers are typically embedded in an environment (e.g., a SoC), and it is equally important to verify the design when working in its final platform, specially when dynamic reconfiguration is used, since a reconfiguration should not affect the behaviour of the system, and therefore, monitoring is crucial to verify the design. An example of this is shown in figure 5.25, where the switching between torque controllers is shown (cf. section 4.4.4).

The time of switching is marked by the signals in the first line. The graph corresponding to the control error shows no effects in the next 8 samples after the switching.

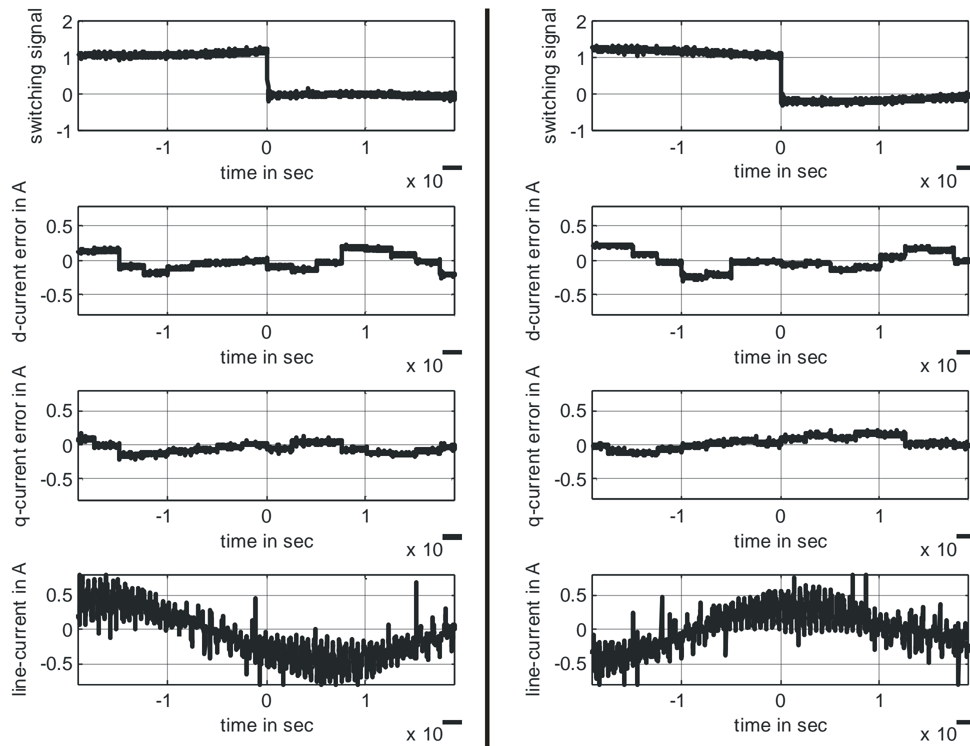


Figure 5.25: Switching between Control Schemes. Left: FOC to FOC-Back EMF-DeC. Right: FOC-Back EMF to FOC [7]

Considering these measurements it is shown that for the considered examples, the switching does not have a negative influence on the control loop, and has no direct disadvantage at the time of switching. The later behavior is determined by the dynamic features of the new control structure.

As explained in section 5.3.2, AR can be used to show more sophisticated annotations (e.g., information about the motor) during operation. The use of the AR extension is presented in the next example.

Inverted pendulum

The inverted pendulum system was presented in section 4.4.3. In this section, the focus is on the support given by the HiLDEGART framework to the design and verification process.

The goal of a HiLDEGART test is to verify the function of a DUT in real-time, using a real environment. In the case of the inverted pendulum system, the controllers are executed in a complex system-on-chip (SoC) architecture, cf. 4.4.2. Therefore, a

monitoring system, which allow real-time debugging on-chip was important to fully test the control system.

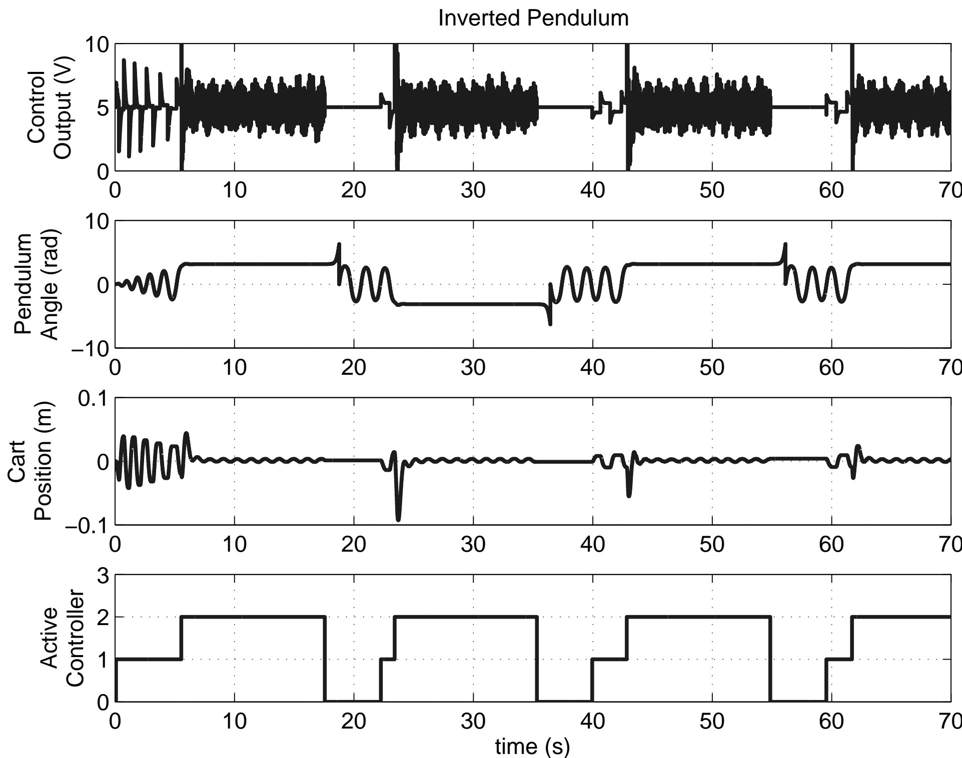


Figure 5.26: HiLDEGART measurements of the inverted pendulum system. Several reconfiguration cycles are shown

The controllers are loaded into the reconfigurable areas (Tiles 1-4) of the SoC architecture based on a Xilinx Virtex-II Pro FPGA (XC2VP30), cf. figure 4.20. This SoC is composed of an embedded PowerPC processor (PPC) connected to dynamically reconfigurable resources (Tile 1 to 4), and to dedicated communication blocks, as well as a HiLDEGART interface. The Broadcast bus of the RAPTOR system is used to communicate values to the DB-MC board (cf. fig. 4.20), which in turn converts these values into voltages for the power electronics part of the pendulum. The angle and the position of the pendulum are gathered from incremental encoders, such that the digital I/Os of the FPGA board can be used to decode these values. The process of dynamic reconfiguration of controllers is supervised by a program on the PPC, which can access the inputs and outputs of the controllers. Once a suitable switching condition has been reached, the PPC can start reconfiguration. This process can be monitored by using HiLDEGART and the AR tool, enabling the validation and further test of the reconfiguration in the target environment.

Control signals, and the inputs and outputs of the plant are shown in figure 5.26. The test presented in this section consisted on initializing the swing-up controller and then switch to the balance controller when the conditions are suitable (cf. section 4.4.3). The balance controller stays enable for some seconds, and then it is deactivated. Therefore, the swing-up controller has to be activated again, completing the cycle, which is repeated several times. This iterations are shown in figure 5.26, where the last plot shows the currently selected controller. This process is also shown in the AR extension of HiLDEGART, as presented in figure 5.27.

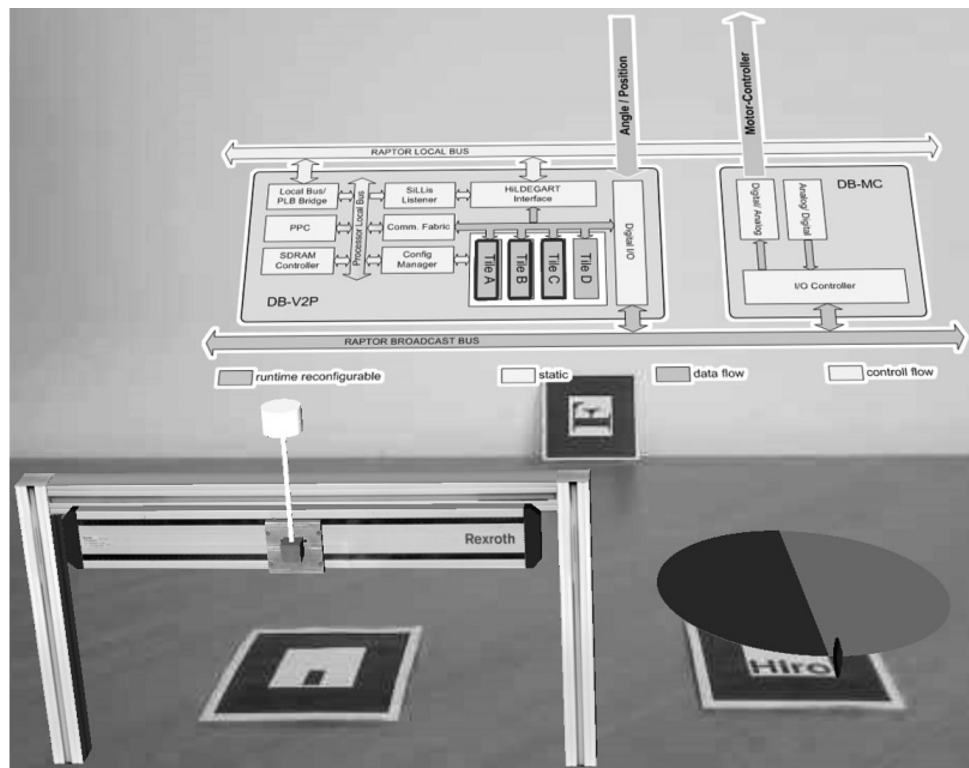


Figure 5.27: HiLDEGART test of the inverted pendulum system using an Augmented Reality extension. Balance controller is active

The AR extension shows, in addition to the pendulum, context-sensitive information regarding the control unit, including a logical structure of all components and bus systems. The visualisation also shows the allocation of a controller to a certain Tile of the SoC architecture. Furthermore, the current angle of the pendulum and the force and direction of the movements of the cart are shown, as presented in figure 5.27. This clarifies the mode of operation of the system, because annotations are shown in a visual context, which is familiar to the viewer (i.e., the physical context), and therefore, the interpretation of the annotations is easier. The AR extension can also be used in combination with a model of the pendulum, which is executed in real time. In

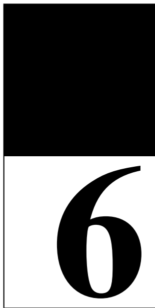
this case, outputs of the model can be visualised, in a way that the user can intuitively understand them [16].

5.4 Summary

This chapter presents a Hardware-in-the-Loop (HiL) design environment for FPGA-based systems. The presented framework supports a two-stage verification process: A cycle-accurate HiL simulation using well-known simulation tools such as Matlab/Simulink or CaMEL-View, and a real-time test using the target environment of the DUT. The first stage allows a cycle accurate early verification of the DUT using a simulated environment, while the focus of the second stage is on monitoring internal states and I/Os of the DUT in operation, and on adjusting design parameters. All hardware and software interfaces required for both stages are generated individually and automatically by the proposed tool-flow.

The results show that HiLDE can be used as a cycle accurate debugger, spotting out differences between a SiL simulation and the actual realisation in an early stage of the design flow. Furthermore, HiLDE is capable of speedup simulations within Matlab/Simulink and CaMEL-View. The acceleration of the simulation depends largely on the complexity of the simulated design and on the number of input and outputs ports. When the number of I/O operations stays constant the speedup grows with the complexity of the design. Presented examples achieved a speedup of up to three orders of magnitude in comparison with SiL simulations. This speedup results in a shorter development and testing time, given the advantage of using a simulated environment to test the system.

The second framework, HiLDEGART, allows the on-line verification and parametrisation of a design in real-time. The presented examples disclose the possibilities of the proposed HiL framework and the use of AR: If the systems become more complex, a context-sensitive visualisation of annotations facilitates the comprehension of the entire situation and its evaluation. For applications using dynamic hardware reconfiguration, visualisation of the internal states of the architecture (e.g., currently used FPGA area) allows for an easier verification process. The presented framework enables a seamless integration of a DUT to a HiL simulation, thus making the verification process easier and less error prone.



6 Summary and Outlook

The increasing complexity of embedded control systems calls for new computing paradigms. In contrast to CPU-based architectures, when using FPGAs the way computations are realised can be adapted to the application. Furthermore, these devices offer a new degree of flexibility, because the configuration of the device or parts of the device can be adjusted at run-time, allowing the adaptation of the design to varying internal or external conditions. This work has explored the use of FPGA technology for embedded control systems, focusing on three research points:

- A technological comparison of FPGA- and CPU-based realisations, where results are analysed and explained taking algorithmic characteristics of implemented controllers into account.
- The use of run-time hardware reconfiguration for FPGA-based controllers to improve resource-utilisation.
- The use of Hardware-in-the-Loop simulations to improve the design flow of FPGA-based control systems.

The main contributions are summarised in the next section.

6.1 Summary

In the first research point, the advantages of using FPGA technology in contrast to classical CPU-based architectures are analysed, using a PID algorithm, a state-feedback controller, and a state observer as benchmarks. These algorithms are selected because, given they wide-spread use in research and industry, they are representative of the application area. Furthermore, a set of metrics is proposed to measure algorithmic properties of controllers, such as average operation per step (*AOS*), which is used to measure average parallelism, or the size of an algorithm (*SizeAlg*). These algorithmic characteristics are then related to realisation results. Moreover, two metrics taken

from literature and adapted to the application area of embedded control systems, are used to assess the resource utilisation of selected benchmarks: computational density ($C_{density}$=throughput/area) and energy efficiency ($E_{efficiency}$=throughput/power). It is shown that an FPGA-based realisation leads to a higher computational density, and a higher energy efficiency than a CPU-based realisation, depending on the problem size (i.e., the amount of operations to be executed, $SizeAlg$), and the average parallelism (*AOS*) of the algorithm to be implemented. Realisation results show that if the average parallelism (*AOS*) grows along with the problem size ($SizeAlg$), the difference of $C_{density}$ values between FPGA- and CPU-based realisations increases. I.e., FPGA-based realisations achieve greater values of $C_{density}$ than CPU-based results, and this difference increases as the size of the design to be implemented increases. On the contrary, when the *AOS* decrease or does not grow at the same rate of problem size, the gap between FPGA and CPU decreases as $SizeAlg$ grows.

FPGAs have limited logical resources, and therefore, the achievable performance decreases for controllers requiring more configurable logic elements than those available in the device, because these resources have to be time-shared. Apart from design optimisations, which help to reduce resource utilisation without sacrificing parallelism (e.g., bit-width optimisation), shortcoming of resources can be avoided for controllers that require some kind of adjustment during operation. For this kind of controllers, all possible configurations (e.g., for a multi-control system, all required control structures) have to be instantiated on the FPGA to allow adaptation of the controller at run-time, if the configuration of the device stays constant during the whole operation cycle (i.e., a static approach). In the second part of this thesis, the possibility of reconfiguring parts of an FPGA at run-time (run-time reconfiguration) is proposed, as a way to improve resource utilisation. Two cases of adjustments are distinguished: structural and parametric changes. For both cases run-time reconfiguration (RTR) can be used to achieve a better resource utilisation, depending on the amount of structural variations, or the size of the algorithm requiring parametric changes. It is shown that the resource utilisation of a dynamic approach depends on the worst-case configuration of the system, whereas for a static implementation the resource utilisation depends on all required configurations. This leads to a better resource utilisation when using RTR when compared to an static approach, depending on the number of structural variations, or the amount of parameters to be adjusted.

The relative long configuration periods of current FPGAs pose a challenging situation, because loading and initialising a controller might take several control cycles, which can lead to disturbances of the controlled system if the new-loaded controller is not properly initialised. A strategy is presented, where the to-be-configured controller is reconfigured and initialised, while the old controller is still in operation to achieve a bump-less transition between controllers. This kind of initialisation is only required for control algorithms with internal memory (e.g., the initial value of the integrator in a PID controller). This allows the use of RTR at the cost of increasing the worst-case of

resource utilisation, which for control systems requiring a large set of configurations still leads to a better resource utilisation than a static approach.

To illustrate the advantage of the proposed concepts and strategies, two implementation examples are presented: a multi-controller system for an inverted pendulum, and a self-optimising motion controller. The run-time exchange of controllers through dynamic reconfiguration for the inverted pendulum is presented as proof-of-concept example, which shows the potentials of using RTR compared to a static approach. The early switching between controllers allows the utilisation of one single slot (i.e., a fixed-slot placement was used), despite the fact that the reaction time (i.e., reconfiguration time plus initialisation time) of both controllers is longer than the control period. In the second implementation example, the realisation of an FPGA-based self-optimizing motion controller is presented. This approach allows the adaptation of parameters and the structure of controllers. Furthermore, not only the control algorithm, but also its realisation and the execution platform (FPGA or embedded CPU) can be dynamically changed. It was shown that switching between different FPGA-based realisations and from an FPGA- to a CPU-based realisation (and vice versa) can be done, without perturbation of the plant. This approach empowers control systems to react to situations far beyond the classic approaches, because not only the parameters, or the structure of the controller can be adapted, but also its kind of implementation. This allows also the optimisation of computational resources of a mechatronic system.

The realisation of such control systems using FPGA technology calls for new design methodologies, to which suitable verification mechanisms belong. A Hardware-in-the-Loop (HiL) design environment for FPGA-based systems is presented in the third part of the presented work. The focus of the framework is, in a first stage (HiLDE), the cycle accurate verification of a design under test (DUT) using a simulated environment. In a second stage (HiLDEGART) the design can be monitored and parametrised in real-time using the target environment. An important aspect of the proposed framework is that all hardware and software interfaces required for both stages are generated individually and automatically.

Many implementation examples were presented, disclosing the efficacy of the proposed frameworks. It is shown that HiLDE can be used as a cycle accurate debugger, detecting discrepancies between a SiL (e.g., using System Generator) and a HiLDE simulation. These discrepancies are mainly caused by badly simulated inter-block quantisation effects, which are not present when a DUT is running in its target platform. Furthermore, it is shown that HiLDE is capable of speeding up simulations when using Matlab/Simulink or CaMEL-View. The acceleration depends largely on the complexity of the simulated design and on the number of input and outputs ports of the DUT. It is shown that when the number of I/O operations stays constant the speed up grows with the complexity of the design. This results in a shorter

development and testing time, given the advantage of using a simulated environment to test the system.

The second framework, HiLDEGART, allows the on-line verification and parametrisation of a design in real-time, while using the target environment. This is specially beneficial for the investigated examples using RTR, because of the complexity of the target environment of the DUT, which consisted of a complex reconfigurable system-on-chip architecture. The presented examples show the capabilities of the proposed HiL framework. Furthermore, the presented framework enables a seamless integration of a DUT to a HiL simulation, thus making the verification process faster and avoiding any introduction of errors in the test process.

The results presented in this thesis show that dynamically reconfigurable hardware is a suitable implementation platform for demanding control applications. It is shown why control algorithms can benefit from this technology, how the resource utilisation can be improved through dynamic hardware reconfiguration for controllers requiring parametric or structural adjustments in run-time, and how this approach gives a new degree of flexibility to the design of mechatronic systems. Furthermore, this thesis presents tools and methods, which allows the verification of an FPGA-based design through different steps, using Hardware-in-the-Loop simulations.

6.2 Outlook

Some issues remain unexplored in this thesis, which can improve the resource utilisation of FPGA-based controllers. One of them is bit-width optimisation. In chapter 3, the consequences of increasing the bit-width of an arithmetic operand are shown (i.e., higher resource utilisation, longer execution times), regarding resource utilisation, and achievable execution time. A point that can be explore is the combination of bit-width optimisation of fix-point arithmetic operands, and run-time hardware reconfiguration. The required bit-width of a fix-point operand strongly depends on the current situation of the plant, i.e., on the numerical range of the input signals of the controller. The required bit-width also depends on the desired control goals, e.g., if the set-point of the controller is changed, it is likely that the numerical range of input signals will change, too.

If computational resources of a mechatronic system have to be shared, a set of realisations of a control structure can be implemented, where the bit-widths of each operation is optimised for a specific operation range of the system. Using run-time-reconfiguration, the optimal version of the controller structure can be selected, in order to optimise used computational resources.

Furthermore, the use of other kind of reconfigurable devices with shorter reconfiguration times can further improve the utilisation of silicon resources of the device. An example of this is the Time-Machine device from Tabula [Hal10], which allows

to store several configurations (up to eight) of the active logic in an internal memory. These configurations can then be switched depending on the operational context, to realise a different function. This approach allows a very rapid reconfiguration of the entire device (up to 1.6 Billion times per second, according to [Hal10]). The idea is not new, already in the late 90s Xilinx proposed a similar concept [Tri97]. However, there has been no relevant practical or commercially available devices until recently. This approach would result in even higher computational density values, not only for high-performance control applications, but also for low-performance systems (i.e., a larger amount of active logic than physically available can be emulated through cyclic run-time reconfiguration).

Finally, the aspect of power consumption of current reconfigurable architectures can be further explored. In this work, it was shown that FPGA technology achieves a higher throughput/power ratio than CPU realisations. Nevertheless, if power consumption has to be limited (e.g., because the system runs on batteries, or heat-dissipation issues), then technological changes of reconfigurable architectures are required. Several trends are currently being follow:

- Improving CAD Tools to achieve optimal power consumption([Bhu10, Has10]).
- Using sub-threshold technologies for FPGA architectures ([Gro11, Rya10, Cal10]).
- Optimising the FPGA architecture itself to minimise energy consumption ([Sir10, Has10]).

A different approach that can be explored is the design of power-aware control systems. As in the case of the previously exposed idea of dynamic bit-width optimisation, run-time hardware reconfiguration can be used to switch among a set of controller implementations, with different throughput/power ratios, in order to react to spare-energy or energy-saving scenarios. In case of an imminent system shut-down (e.g., because of a low-energy battery level), the stability of the controlled plant has to be guaranteed (i.e., a controlled shut-down), while enlarging the operation cycle as much as possible. Under such scenario, a trade-off between controller performance and power consumption has to be made. Another application scenario are self-optimizing mechatronic systems, where the mechatronic system can adjust its optimisation goals depending on internal and external conditions. Run-time optimisation of power consumption for FPGA-based controllers can be realised using dynamic reconfiguration.

In this work, is has been shown that dynamically reconfigurable hardware, particularly FPGA technology, is a suitable platform for demanding control applications. Methods and tools presented in this thesis are an effort to disclose the advantages of FPGAs, and a step towards taking full advantage of the possibilities offered by this technology in the context of embedded control systems.

Author's Publications

[1] B. Kettelhoit, A. Klassen, C. Paiz, M. Porrmann, and U. Rückert, "Rekonfigurierbare Hardware zur Regelung mechatronischer Systeme," in *3. Paderborner Workshop: Intelligente mechatronische Systeme*, (Paderborn, Germany), pp. 195–205, March 2005.

[2] C. Paiz, B. Kettelhoit, A. Klassen, and M. Porrmann, "Dynamically Reconfigurable Hardware for Digital Controllers in Mechatronic Systems," in *IEEE International Conference on Mechatronics (ICM2005)*, (Taipei, Taiwan), pp. 675–680, July 2005.

[3] C. Paiz, T. Chinapirom, U. Witkowski, and M. Porrmann, "Dynamically Reconfigurable Hardware for Autonomous Mini-Robots," in *32nd Annual Conference of the IEEE Industrial Electronics Society IECON 06*, (Paris, France), pp. 3981–3986, November 2006.

[4] C. Paiz, C. Pohl, and M. Porrmann, "Reconfigurable Hardware in-the-Loop Simulations for Digital Control Design," in *3rd International Conference on Informatics in Control, Automation and Robotics (ICINCO 2006)*, (Setubal, Portugal), pp. 39–46, August 2006.

[5] C. Paiz and M. Porrmann, "The Utilization of Reconfigurable Hardware to Implement Digital Controllers: a Review," in *the IEEE International Symposium on Industrial Electronics*, (Vigo, Spain), pp. 2380–2385, June 2007.

[6] C. Paiz, K. Boris, and M. Porrmann, "A Design Framework for FPGA-based Dynamically Reconfigurable Digital Controllers," in *The IEEE International Symposium on Circuits and Systems (ISCAS)*, (New Orleans, USA), pp. 3708–3711, May 2007.

[7] B. Schulz, C. Paiz, J. Hagemeyer, S. Mathapati, M. Porrmann, and J. Boecker, "Run-Time Reconfiguration of FPGA-Based Drive Controllers," in *12th European Conference on Power Electronics and Applications (EPE)*, (Aalborg, Denmark), September 2007.

[8] C. Pohl, C. Paiz, and M. Porrmann, "Hardware-in-the-Loop Entwicklungsumgebung für informationsverarbeitende Komponenten mechatronischer Systeme,"

in *5. Paderborner Workshop Entwurf mechatronischer Systeme*, (Paderborn, Germany), pp. 69–79, March 2007.

[9] J. Twiefel, M. Klubal, C. Paiz, S. Mojrzisch, and H. Krüger, "Digital Signal Processing for an Adaptive Phase-Locked Loop Controller," in *SPIE Smart Structures and Materials and Nondestructive Evaluation and Health Monitoring*, (San Diego, California, USA), SPIE–The International Society for Optical Engineering, January 2008.

[10] E. Münch, A. Gambuzza, C. Paiz, C. Pohl, and M. Porrmann, "FPGA-in-the-Loop Simulations with CAMEL-View," in *In Proceedings of the 7th International Heinz Nixdorf Symposium*, p. 429–445, February 2008.

[11] C. Pohl, C. Paiz, and M. Porrmann, "vMAGIC – VHDL Manipulation and Automation for Reliable System Development," in *3rd International Workshop on Reconfigurable Computing Education*, (Montpellier, France), April 2008.

[12] C. Paiz, C. Pohl, and M. Porrmann, "Hardware-in-the-Loop Simulations for FPGA-Based Digital Control Design," *Informatics in Control, Automation and Robotics*, vol. 3, pp. 355–372, 2008.

[13] C. Pohl, C. Paiz, and M. Porrmann, "A Hardware-in-the-Loop Design Environment for FPGAs," in *Design, Automation and Test in Europe DATE, University Booth*, (Munich, Germany), 2008.

[14] P. Adelt, J. Donoth, J. Gausemeier, J. Geisler, S. Henkler, S. Kahl, B. Klöpper, M. E. Krupp, A., S. Oberthür, C. Paiz, M. Porrmann, R. Radkowski, A. Romaus, C. Schmidt, B. Schulz, H. Vöcking, U. Witkowski, K. Witting, and A. Znamenshchykov, "Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen und Konzepte," *Bd. 234, Heinz-Nixdorf Institut, Universität Paderborn 2009 (HNI-Verlagsschriftenreihe), Deutschland*, 2009.

[15] C. Paiz, J. Hagemeyer, C. Pohl, M. Porrmann, U. Rückert, B. Schulz, W. Peters, and J. Boecker, "FPGA-Based Realization of Self-Optimizing Drive-Controllers," in *the IEEE International Symposium on Industrial Electronics (IECON 2009)*, pp. 2848 –2853, 2009.

[16] C. Paiz, C. Pohl, R. Radkowski, J. Hagemeyer, M. Porrmann, and U. Rückert, "FPGA-in-the-Loop-Simulations for Dynamically Reconfigurable Applications," in *the 2009 International Conference on Field-Programmable Technology (FPT'09)*, (The University of New South Wales, Sydney, Australia), pp. 372–375, 2009.

[17] C. Pohl, C. Paiz, and M. Porrmann, "vMAGIC - Automatic Code Generation for VHDL," *International Journal of Reconfigurable Computing*, 2009. Article ID 205149.

Bibliography

[Aco02] Acosta, N. and Tosini, M. "Custom Architectures for Fuzzy and Neural Networks Controllers." In R. Jordan, ed., *Journal of Computer Science and Technology*, vol. 2. 2002, pp. 9–15.

[Ada00] Adamski, M. "Reprogrammable Application Specific Logic Controllers and SFC based Design." In *45th International Scientific Colloquium Ilmenau Technical University*. 2000.

[ALD11] "ALDEC.", 2011. URL `http://www.aldec.com/products/hes`.

[Alt05] Altera. "FPGAs for High-Performance DSP Applications." White paper, Altera Corporation, San Jose, CA., May 2005.

[Amd67] Amdahl, G. M. "Validity of the single processor approach to achieving large scale computing capabilities." In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, New York, NY, USA, 1967, pp. 483–485.

[As01] Å ström, K. J. and Hägglund. "The future of PID Control." In *Control Engineering Practice*. 2001, pp. 1163–1175.

[As08] Å ström, K. J. and Murray, R. M. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, illustrated edition ed., April 2008. ISBN 0691135762.

[ATM11] "ATMEL Homepage.", 2011. URL `http://www.atmel.com`.

[Azu97] Azuma, R. T. "A Survey of Augmented Reality." In *Presence*, vol. 6, 1997:pp. 355–385.

[Bay72] Bayer, K.-H., Waldmann, H., and Weibelzahl, M. "Field-Oriented Closed-Loop Control of A Synchronous Machine With the New Transvektor Control System." In *Siemens Review*, vol. 39. 1972, pp. 220–223.

[Böc06] Böcker, J., Schulz, B., Knoke, T., and Fröhleke, N. "Self-Optimization as a Framework for Advanced Control Systems." In *Int. Electronics Conference (IECON), November 2006, Paris, Frankreich*, 2006.

[Ben99] Benedetti, G., M.and Uicich. “New High-Performance Thyristor Gate Control Set for Line-Commutated Converters.” In *IEEE Transactions On Industrial Electronics*, vol. 46. October 1999, pp. 972–977.

[Bhu10] Bhunia, S. and Mukhopadhyay, S. *Low-Power Variation-Tolerant Design in Nanometer Silicon*. Springer, 2010. ISBN 9781441974174.

[Bla72] Blaschke, F. “The Principle of Field Orientation as Applied to the New Transvector Closed-Loop Control System for Rotating-Field Machines.” In *Siemens Review*, vol. 34. May 1972, pp. 217–220.

[Bla03] Blanke, M., Kinnaert, M., Lunze, J., and Staroswiecki, M. *Diagnosis and Fault-Tolerant Control*. Springer-Verlag, Berlin, Germany, 2003.

[BO10] Ben Othman, S., Ben Salem, A., and Ben Saoud, S. “Hw acceleration for FPGA-based drive controllers.” In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*. july 2010, pp. 196–201.

[Bol01] Boluda, J. A. and Domingo, J. “On the advantages of combining differential algorithms and log polar vision for detection of self motion from a mobile robot.” In *Robotics and Autonomous Systems*, vol. 37. Elsevier, 2001, pp. 283–296.

[Bol04] Bolognani, S., Ceschia, M., Tomasini, M., Tubiana, L., and Zigliotto, M. “FPGA Implementation of an Iterative Algorithm for Time Optimal Control of AC Drives.” In *11th International Conference on Power Electronics and Motion Control*. 2004.

[Bol09] Bol, D., Ambroise, R., Flandre, D., and Legat, J.-D. “Interests and Limitations of Technology Scaling for Subthreshold Logic.” In *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 10, oct. 2009:pp. 1508 –1519.

[Bra05] Brandmayr, G., Humer, G., and Rupp, M. “Automatic Co-Verification Of FPGA Designs In Simulink.” In *Model-Based Design Conference*. June 2005.

[Bur96] Burd, T. D. and Brodersen, R. W. “Processor design for portable systems.” In *J. VLSI Signal Process. Syst.*, vol. 13, no. 2-3, 1996:pp. 203–221. ISSN 0922-5773.

[Cab04] Cabrera, A., Sanchez-Solano, S., Brox, P., Barriga, A., and Senhadj, R. “Hardware software codesign of configurable fuzzy control systems.” In *Applied Soft Computing*, vol. 4. Elsevier Science, 2004, pp. 271–285.

[CAD11] "Palladium III.", 2011. URL `http://www.cadence.com`.

[Cal10] Calhoun, B., Ryan, J., Khanna, S., Putic, M., and Lach, J. "Flexible Circuits and Architectures for Ultralow Power." In *Proceedings of the IEEE*, vol. 98, no. 2, February 2010:pp. 267–282. ISSN 0018-9219.

[Can02] Cantle, A., Devlin, M., and Lord, R., E.and Chamberlain. "High Frame Rate Low Latency Hardware-in-the-Loop Image Generation." White paper, Nallatech Ltd, 10-14 Market Street, Kilsyth, Glasgow, Scotland, G65 0BD, 2002.

[Car03] Carrica, D., Funes, M., and Gonzalez, S. "Novel Stepper Motor Controller Based on FPGA Hardware Implementation." In *IEEE/ASME Transactions On Mechatronics*, vol. 8. IEEE/ASME, March 2003, pp. 120–124.

[Cha04] Chapuis, Y., Blonde, J., and Braun, F. "FPGA Implementation by Modular Design Reuse Mode to Optimize Hardware Architecture and Performance of AC Motor Controller Algorithm." In *11th International Conference on Power Electronics and Motion Control*. 2004.

[Cha05b] Charaabi, L., Monmasson, E., Nassani, M.-A., and Slama-Belkhodja, I. "FPGA-based implementation of DTSFC and DTRFC algorithms." In *Annual Conference of the IEEE Industrial Electronics Society*. 2005.

[Cha10] Chang, L., Frank, D., Montoye, R., Koester, S., Ji, B., Coteus, P., Dennard, R., and Haensch, W. "Practical Strategies for Power-Efficient Computing Technologies." In *Proceedings of the IEEE*, vol. 98, no. 2, feb. 2010:pp. 215 –236.

[Che00] Chen, R. X., Chen, L. G., and Chen, L. "System Design Consideration for Digital Wheelchair Controller." In *Transactions On Industrial Electronics*, vol. 47. IEEE, August 2000, pp. 898–907.

[Che02] Chen, J. and Lin, I. "Toward the implementation of an ultrasonic motor servo drive using FPGA." In *Mechatronics*, vol. 12. Elsevier Science, 2002, pp. 511–524.

[Che11] Cheng, Z., Yang, H., and Liu, Y. "Self-Adjusting Fuzzy MPPT PV System Control by FPGA Design." In *Power and Energy Engineering Conference (APPEEC), 2011 Asia-Pacific*. march 2011, pp. 1 –4.

[Cho01] Chohra, A. and Schöll, P. "Neural Networks (NN) Based Learning of Elementary Behaviors and their Integration in FPGA Architectures for a Fast Moving Robot Team (RoboCup)." In *Proceedings of the Workshop on Autonomous Artificial Systems Exploring Hostile Environments*. 2001, pp. 65–71.

[Chu02] Chujo, N. "Fail-safe ECU System Using Dynamic Reconfiguration of FPGA." In *R & D Review of Toyota CRDL*, vol. 37. April 2002, pp. 54–60.

[Cla99] Claasen, T. "High speed: not the only way to exploit the intrinsic computational power of silicon." In *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*. 1999, pp. 22 –25.

[Com02] Compton, K. and Hauck, S. "Reconfigurable Computing: A Survey of Systems and Software." In *ACM Computing Surveys*, vol. 34. June 2002, pp. 171–210.

[Cor01] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. ISBN 0070131511.

[CP01] Cumplido-Parra, R., Jones, S. R., Godall, R. M., Mitchell, F., and Bateman, S. "High Performance Control System Processor." In R. Merker and W. Schwarz, eds., *System Design Automation: Fundamentals, Principles, Methods, Examples*. Kluwer Academic Publishers, March 2001, pp. 140–151.

[Dan03a] Danne, K., Bobda, C., and Kalte, H. "Increasing Efficiency by Partial Hardware Reconfiguration: Case Study of a Multi-Controller System." In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, Nevada, USA*. June 2003.

[Dan03b] Danne, K., Bobda, C., and Kalte, H. "Run-time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration." In *International Conference on Field Programmable Logic and Applications*. September 2003.

[DeH95] DeHon, A. "Comparing Computing Machines." In *SPIE. Configurable Computing: Technology and Applications*. 1995, pp. 124–133.

[DeH99] DeHon, A. and Wawrzynek, J. "Reconfigurable computing: what, why, and implications for design automation." In *DAC 99: Proceedings of the 36th ACM/IEEE conference on Design automation*. ACM Press, New York, NY, USA, 1999, pp. 610–615.

[DeH00] DeHon, A. "The density advantage of configurable computing." In *Computer*, vol. 33, no. 4, Apr 2000:pp. 41–49.

[Dep04] Deppe, M., Zanella, M., Robrecht, M., and Hardt, W. "Rapid prototyping of real-time control laws for complex mechatronic systems a case study." In *The Journal of Systems and Software*, vol. 70. 2004, pp. 263–274.

[Des06] Deschamps, J., Bioul, G., and Sutter, G. *Synthesis of Arithmetic Circuits*. Wiley-Intersciences, New Jersey,USA, 2006.

[Din05] Dinavahi, V., Iravani, R., and Bonert, R. “Design of a Real-Time Digital Simulator for a D-STATCOM System.” In *IEEE Transactions On Industrial Electronics*, vol. 51. October 2005, pp. 1001–1008.

[Do10] Do, T. and Le, T. “High throughput area-efficient SoC-based forward/inverse integer transforms for H.264/AVC.” In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. 30 2010-june 2 2010, pp. 4113 –4116.

[Don03] Donecker, S., Lasky, T., and Ravani, B. “A Mechatronic Sensing System for Vehicle Guidance and Control.” In *IEEE/ASME Transactions On Mechatronics*, vol. 8. December 2003, pp. 500–510.

[Dor11] Dorf, R. and Bishop, R. *Modern control systems*. Pearson Prentice Hall, 2011.

[DSP] “DSP Builder.” URL `http://www.altera.com`.

[dSP05] dSPACE GmbH, Paderborn, Germany. *DS1005 PPC Board, Features*, 5 ed., November 2005.

[dSP11] “dSPACE Homepage.”, 2011. URL `http://www.dspace.de`.

[Eag89] Eager, D., Zahorjan, J., and Lazowska, E. “Speedup versus efficiency in parallel systems.” In *IEEE Transactions on Computers*, vol. 38, no. 3, 1989:pp. 408–423.

[Ell09] Ellithorpe, J. D., Tan, Z., and Katz, R. H. “Internet-in-a-Box: emulating datacenter network architectures using FPGAs.” In *DAC ’09: Proceedings of the 46th Annual Design Automation Conference*. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-497-3, pp. 880–883.

[Faa04] Faa-Jeng, L., Dong-Hai, W., and Po-Kai, H. “RFNN controlled sensorless induction spindle motor drive.” In *Electric Power Systems Research*, vol. 70. Elsevier Science, December 2004, pp. 211–222.

[Fan05] Fang, Z., Carletta, J., and Veillette, R. “A Methodology for FPGA-Based Control Implementation.” In *IEEE Transactions On Control Systems Technology*, vol. 13. November 2005, pp. 977–987.

[Fan07] Fang, W. and Spaanenburg, L. “Power-driven FPGA to ASIC conversion.” In *SPIE 6590*. 2007, pp. 22 –25.

[Fen06] Feng, W. and Greene, J. W. "Post-placement interconnect entropy: how many configuration bits does a programmable logic device need?" In *SLIP '06: Proceedings of the 2006 international workshop on System-level interconnect prediction*. ACM, New York, NY, USA, 2006, pp. 41–48.

[Fra01] Frank, D., Dennard, R., Nowak, E., Solomon, P., Taur, Y., and Wong, H.-S. P. "Device scaling limits of Si MOSFETs and their application dependencies." In *Proceedings of the IEEE*, vol. 89, no. 3, Mar 2001:pp. 259–288.

[Gal10] Galal, S. and Horowitz, M. "Energy-Efficient Floating Point Unit Design." In *Computers, IEEE Transactions on*, vol. PP, no. 99, 2010:p. 1.

[Gei05] Geisler, J. *Auslegung und Implementierung der verteilten Aktor- und Aufbauregelung für ein aktiv gefedertes Schienenfahrzeug*. Master's thesis, University of Applied Sciences Osnabrück, 2005.

[Gev02] Gevers, M. "A decade of progress in iterative process control design: from theory to practice." In *Journal of Process Control*, vol. 12, no. 4, 2002:pp. 519 – 531. ISSN 0959-1524.

[Gra09] Grassi, P. R., Santambrogio, M. D., Hagemeyer, J., Pohl, C., and Porrmann, M. "SiLLis: A Simplified Language for Monitoring and Debugging of Reconfigurable Systems." In *Engineering of Reconfigurable Systems and Algorithms*. 2009, pp. 174–180.

[Gro11] Grossmann, P. and Leeser, M. "A prototype FPGA for subthreshold-optimized CMOS." In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0554-9, pp. 279–279. URL `http://doi.acm.org/10.1145/1950413.1950470`.

[Gus88] Gustafson, J. L. "Reevaluating Amdahl's law." In *Commun. ACM*, vol. 31, no. 5, 1988:pp. 532–533. ISSN 0001-0782.

[Gwa02] Gwaltney, D., King, K., and Smith, K. "Implementation of Adaptive Digital Controllers on Programmable Logic Devices." In *5th Military and Aerospace Programmable Logic Devices (MAPLD) International Conference*. NASA Marshall Space Flight Center, September 2002.

[Hag05] Hagemeyer, J. *Partielle dynamische Selbst-Rekonfiguration mit Virtex-II FPGAs*. Master's thesis, Studienarbeit, System and Circuit Group, Universität Paderborn, Paderborn, Germany, 2005.

[Hag06] Hagemeyer, J., Kettelhoit, B., and Porrmann, M. "Dedicated module access in dynamically reconfigurable systems." In *Parallel and Distributed Processing Symposium, International*, vol. 0, 2006:p. 189.

[Hag07a] Hagemeyer, J., Keltelhoit, B., Koester, M., and Pomnann, M. "A Design Methodology for Communication Infrastructures on Partially Reconfigurable FPGAs." In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. 27-29 2007, pp. 331 –338.

[Hag07b] Hagemeyer, J., Kettelhoit, B., Koester, M., and Porrmann, M. "Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs." In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*. Las Vegas, USA, 2007.

[Hal10] Hallfhill, T. R. "Tabula's Time Machine, Rapidly Reconfigurable Chips Will Challenge Conventional FPGAs." In *Microprocessor*, vol. 24, no. 3, March 2010:pp. 17–30.

[Han] "Handel-C." URL `http://www.celoxica.com`.

[Han95] Hanen, C. and Munier, A. "A study of the cyclic scheduling problem on parallel processors." In *Discrete Applied Mathematics*, vol. 57, no. 2-3, 1995:pp. 167–192.

[Han02] Hannan Bin Azhar, M. A. and Dimond, K. R. "Design of an FPGA Based Adaptive Neural Controller for Intelligent Robot Navigation." In *Euromicro Symposium on Digital System Design*, vol. 00. 2002, pp. 283–295.

[Han03] Hannan Bin Azhar, M. A. and Dimond, K. R. "Hardware Implementation of a Genetic Controller and Effects of Training on Evolution." In *ICES*. 2003, pp. 344—-354.

[Har01] Hartmann, N. *Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik*. Ph.D. thesis, Universi
"at Karlsruhe, January 2001.

[Has10] Hassan, H. and Anis, M. *Low-Power Design of Nanometer FPGAs: Architecture and EDA*. Morgan Kaufmann series in systems on silicon. Elsevier Science, 2010. ISBN 9780123744388.

[He04] He, P., Jin, M., Yang, L., Wei, R., Liu, Y., Cai, H., Liu, H., Seitz, N., Butterfass, J., and Hirzinger, G. "High Performance DSP/FPGA Controller for Implementation of HIT/DLR Dexterous Robot Hand." In *IEEE International Conference on Robotics and Automation*. New Orleans, LA, April 2004, pp. 3397–3402.

[Hen07] Hennessy, J. and Patterson, D. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2007.

[Her04] Hernandez, A., Urena, J., Garcia, J., Mazo, M., Hernanz, D., Derutin, J., and Serot, J. "Ultrasonic Ranging Sensor using Simultaneous Emissions from Different Transducers." In *IEEE Transactions On Ultrasonics, Ferroelectrics, And Frequency Control*, vol. 51. December 2004, pp. 1660–1670.

[Hes04] Hestermeyer, T., Oberschelp, O., and Giese, H. "Structured information processing for self-optimizing mechatronic systems." In *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004)*. Setubal, Portugal, 2004.

[Ho90] Ho, E. and Sen, P. "A microcontroller-based induction motor drive system using variable structure strategy with decoupling." In *Industrial Electronics, IEEE Transactions on*, vol. 37, no. 3, Jun 1990:pp. 227–235.

[Ho00] Ho, Y. C., Man, K. F., Tang, K. S., and Kwong, S. "A Codesign Approach to Real-time High Precision Control." In *Real-Time Systems*, vol. 19, no. 1, 2000:pp. 41–60. ISSN 0922-6443.

[IBM02] IBM Microelectronics Division, New York, USA. *PowerPC 740 and PowerPC 750 Microprocessor Datasheet*, 2 ed., June 2002.

[Ise99] Isermann, R., Schaffnit, J., and Sinsel, J. "Hardware-in-the-loop simulation for the design and testing of engine-control systems." In *Control Engineering Practice*, vol. 7. Elsevier, May 1999, pp. 643–653.

[Jia02] Jian-Shiang, C. and In-Dar, L. "Toward the implementation of an ultrasonic motor servo drive using FPGA." In *Mechatronics*, vol. 12. Elsevier Science, 2002, pp. 511–524.

[Jon10] Jones, K. "Algorithm Design for Hardware-Based Computing Technologies." In *The Regularized Fast Hartley Transform*, Signals and Communication Technology. Springer Netherlands, 2010, pp. 65–75.

[Jua05] Juang, C. and Hsu, C. "Temperature Control by Chip-Implemented Adaptive Recurrent Fuzzy Controller Designed by Evolutionary Algorithm." In *IEEE Transactions On Circuits And Systems*, vol. 52. November 2005, pp. 2376–2384.

[Jun99] Jung, S. L., Chang, M. Y., Jyang, J., Yeh, L. C., and Tzou, Y. "Design and Implementation of an FPGA-Based Control IC for AC-Voltage Regulation." In *Transactions On Power Electronics*, vol. 14. IEEE, IEEE Press, May 1999, pp. 522–532.

[Jun10] Jungeblut, T., Luetkemeier, S., Sievers, G., Porrmann, M., Rückert, U., and Kastens, U. "A modular design flow for very large design space explorations."

In *Proceedings of the CDNLive! EMEA 2010, Munich, Germany, 2010*. May 2010.

[Kal59] Kalman, R. "On the general theory of control systems." In *Automatic Control, IRE Transactions on*, vol. 4, no. 3, December 1959:pp. 110–110. ISSN 0096-199X.

[Kal02] Kalte, H., Porrmann, M., and Rückert, U. "A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs." In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*. Hamburg, Germany, 2002.

[Kel97] Kelly, J. S., Rao, V. S., Pottinger, H. J., and Bowman, H. C. "Design and implementation of digital controllers for smart structures using field programmable gate arrays." In *Smart Materials and Structures*, vol. 6. October 1997, pp. 559–572.

[Ket08] Kettelhoit, B. *Architektur und Entwurf dynamisch rekonfigurierbarer FPGA-Systeme*. Dissertation, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, 2008. HNI-Verlagsschriftenreihe, Paderborn.

[Kha91] Khambadkone, A. and Holtz, J. "Vector-controlled induction motor drive with a self-commissioning Scheme." In *Industrial Electronics, IEEE Transactions on*, vol. 38, no. 5, Oct 1991:pp. 322–327.

[Kim00] Kim, D. "An Implementation of Fuzzy Logic Controller on the Reconfigurable FPGA System." In *IEEE Transactions On Industrial Electronics*, vol. 47. IEEE, June 2000, pp. 703–715.

[Kis02] Kish, L. B. "End of Moore's law: thermal (noise) death of integration in micro and nano electronics." In *Physics Letters A*, vol. 305, no. 3-4, 2002:pp. 144 – 149. ISSN 0375-9601. URL `http://www.sciencedirect.com/science/article/pii/S0375960102013658`.

[Kis04] Kish, L. "Moore's law and the energy requirement of computing versus performance." In *Circuits, Devices and Systems, IEE Proceedings -*, vol. 151, no. 2, april 2004:pp. 190 – 194. ISSN 1350-2409.

[Kou05] Koutroulis, E., Dollas, A., and Kalaitzakis, K. "High-frequency pulse width modulation implementation using FPGA and CPLD ICs." In *Journal of Systems Architecture*. 2005.

[Kun05] Kung, Y. and Shu, G. "Design and Implementation of a Control IC for Vertical Articulated Robot Arm using SOPC Technology." In *Proceedings of the IEEE International Conference on Mechatronics*. Taipei, Taiwan, July 2005, pp. 532–536.

[Kun10] Kung, Y.-S., Hsu, C.-T., Chou, H.-H., and Tsui, T.-W. “FPGA-realization of a motion control IC for wafer-handling robot.” In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*. july 2010, pp. 493 –498.

[Lan05] Langen, D. *Abschätzung des Ressourcenbedarfs von hochintegrierten mikroelektronischen Systemen*. Ph.D. thesis, Schaltungstechnik, Heinz Nixdorf Institut, Universität Paderborn, Fürstenallee 11, 33102 Paderborn, Germany, December 2005.

[Li03] Li, T. S., Chang, S., and Chen, Y. “Implementation of Human-Like Driving Skills by Autonomous Fuzzy Behavior Control on an FPGA-Based Car-Like Mobile Robot.” In *IEEE Transactions On Industrial Electronics*, vol. 50. October 2003, pp. 867–880.

[Lib99] Liberzon, D. and Morse, A. “Basic problems in stability and design of switched systems.” In *Control Systems, IEEE*, vol. 19, no. 5, oct 1999:pp. 59 –70.

[Lu05] Lu, B., Wu, X., and Monti, A. “Implementation of a low-cost real-time virtue test bed for Hardware-in-the-Loop testing.” In *Industrial Electronics Society, 2005. IECON 2005. 31st Annual Conference of IEEE*. November 2005, pp. 239–244.

[Lun06] Lunze, J. and Richter, J. H. “Control Reconfiguration: Survey of Methods and Open Problems.” Research Report 4-FB-2006.08, Institute of Automation and Computer Control,Ruhr-Universität Bochum, 44780 Bochum, Germany, February 2006. URL `http://www.atp.ruhr-uni-bochum.de`.

[Lyg98] Lygouras, J. N., Lalakos, K., and Tsalides, P. G. “High-Performance Position Detection and Velocity Adaptive Measurement for Closed-Loop Position Control.” In *Transactions On Instrumentation And Measurement*, vol. 47. IEEE, IEEE Press, August 1998, pp. 978–985.

[Mat05] Mattavelli, P., Spiazzi, G., and Tenti, P. “Predictive Digital Control of Power Factor Preregulators With Input Voltage Estimation Using Disturbance Observers.” In *IEEE Transactions On Power Electronics*, vol. 20. January 2005, pp. 140–147.

[Mat07] Mathapati, S. and Böcker, J. “Implementation of Dynamically Reconfigurable Control Structures on a Single FPGA Platform.” In *12th European Power Electronics and Adjustable Speed Drives Conference, Aalborg, Denmark*, 2007.

[Mat11] Mathapati, S. *FPGA-Based High Performance AC Drive*. Dissertation, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, 2011.

[MI04] Murshidul-Islam, M., Allee, D. R., Konasani, S., and Rodriguez, A. A. "A Low-Cost Digital Controller for a Switching DC Converter With Improved Voltage Regulation." In *POWER ELECTRONICS LETTERS*, vol. 2. IEEE, December 2004, pp. 121–124.

[Mon99] Monmasson, E. and Echelard, J., H. Louis. "Dynamically Reconfigurable Architecture Dedicated To The Test Of PWM Algorithms." In *International Conference on Power Electronics*. 1999.

[Mon02a] Monmasson, E. and Chapuis, Y. "Contributions of FPGAs to the Control of Electrical Systems, a Review." In *IEEE Industrial Electronics Society Newsletter*, vol. 49, no. 4, December 2002:pp. 8–15.

[Mon02c] Monmasson, E., Robyns, B., Mendes, E., and De Fornel, B. "Dynamic reconfiguration of control and estimation algorithms for induction motor drives." In *Symposium on Industrial Electronics, 2002. ISIE 2002. Proceedings of the 2002 IEEE International*, vol. 3. 2002, pp. 828–833.

[Moo98] Moore, G. "Cramming More Components Onto Integrated Circuits." In *Proceedings of the IEEE*, vol. 86, no. 1, jan 1998:pp. 82 –85.

[Mor95] Morse, A. S. "Control Using Logic-Based Switching." In *Trends in Control: A European Perspective*. Springer-Verlag, 1995, pp. 69–113.

[MV10b] Morales-Velazquez, L., de Jesus Romero-Troncoso, R., Osornio-Rios, R. A., Herrera-Ruiz, G., and de Santiago-Perez, J. J. "Special purpose processor for parameter identification of CNC second order servo systems on a low-cost FPGA platform." In *Mechatronics*, vol. 20, no. 2, 2010:pp. 265 – 272.

[Nab80] Nabae, A., Otsuka, K., Uchino, H., and Kurosawa, R. "An Approach to Flux Control of Induction Motors Operated with Variable-Frequency Power Supply." In *Industry Applications, IEEE Transactions on*, vol. IA-16, no. 3, May 1980:pp. 342–350. ISSN 0093-9994.

[Nak02] Nakamura, T., Awa, Y., Shimoji, H., and Karasawa, H. "Control System Of Electrostatic Levitation Furnace." In *Acta Astronautica*, vol. 50. 2002, pp. 609–614.

[Nam01] Namkung, J. *An Event-level Power Measurement and Analysis Methodology*. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2001.

[Nao04] Naouar, M., Charaabi, L., Monmasson, E., and Belkhodja, I. “Realization of a Library of FPGA Reconfigurable IP-Core Functions for the Control of Electrical Systems.” In *11th International Conference on Power Electronics and Motion Control*. 2004.

[Nas04] Nascimento, P. S. B., Pand Maciel, P. R. M., Lima, M. E., Sant’ana, R. E., and Filho, A. G. S. “A partial reconfigurable architecture for controllers based on Petri nets.” In *SBCCI ’04: Proceedings of the 17th symposium on Integrated circuits and system design*. ACM Press, New York, NY, USA, 2004, pp. 16–21.

[Nat11] “R Series Intelligent DAQ Devices.”, 2011. URL `http://www.ni.com`.

[NBP] “Neue Bahntechnik Paderborn.” URL `http://www-nbp.upb.de`.

[Nek03] Nekoogar, F. *From ASICs to SOCs: a practical approach*. Prentice Hall modular series for engineering. Prentice Hall/Professional Technical Reference, 2003. ISBN 9780130338570.

[Oga87] Ogata, K. *Discrete-Time Control Systems*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, USA, first ed., 1987.

[Old01] Oldknow, K. and Yellowley, I. “Design implementation and validation of a system for the dynamic reconfiguration of open architecture machine tool controls.” In *International Journal of Machine Tools and Manufacture*, vol. 41. 2001, pp. 795–808.

[Old05] Oldknow, K. D. and Yellowley, I. “FPGA-Based Servo Control and Three-Dimensional Dynamic Interpolation.” In *IEEE/ASME Transactions On Mechatronics*, vol. 10. February 2005, pp. 98–110.

[OR08] Osornio-Rios, R. A., de Jesus Romero-Troncoso, R., Herrera-Ruiz, G., and Castañeda-Miranda, R. “The application of reconfigurable logic to high speed CNC milling machines controllers.” In *Control Engineering Practice*, vol. 16, no. 6, 2008:pp. 674 – 684. Special Section on Large Scale Systems, 10th IFAC/IFORS/IMACS/IFIP Symposium on Large Scale Systems: Theory and Applications.

[OR09] Osornio-Rios, R., de Jesús Romero-Troncoso, R., Herrera-Ruiz, G., and Castaņeda-Miranda, R. “FPGA implementation of higher degree polynomial acceleration profiles for peak jerk reduction in servomotors.” In *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 2, 2009:pp. 379–392.

[Ove00] Overeinder, B. J. *Distributed Event-driven Simulation - Scheduling Strategies and Resource Management*. Ph.D. thesis, Department of Computer

Science, University of Amsterdam, Amsterdam, The Netherlands, November 2000. Promotor: Prof. Dr. P.M.A. Sloot, Co-promotor: Prof. Dr. M. Livny.

[Pak10] Paker, O., Eckert, S., and Bury, A. "A low cost multi-standard near-optimal soft-output sphere decoder: algorithm and architecture." In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2010, pp. 1402–1407.

[Par95] Parr, T. and Quong, R. "ANTLR: A Predicated- LL(k) Parser Generator." In *Software - Practice and Experience*, vol. 25, no. 7, 1995:pp. 789–810.

[Pat10a] Patel, P. and Moallem, M. "Reconfigurable system for real-time embedded control applications." In *Control Theory Applications, IET*, vol. 4, no. 11, november 2010:pp. 2506 –2515.

[Pat10b] Patel, P. and Moallem, M. "Using FPGA-based platforms for embedded control applications in Mechatronics." In *Advanced Intelligent Mechatronics (AIM), 2010 IEEE/ASME International Conference on*. july 2010, pp. 1356–1361.

[Pet09] Peters, W., Schulz, B., Mathapati, S., and Bocker, J. "Regular-Sampled Current Measurement in AC Drives Using Delta-Sigma-Modulators." In *Power Electronics and Applications, 2009. EPE '09. 13th European Conference on*. 8-10 2009, pp. 1 –9.

[Poh10] Pohl, C. *Konfigurierbare Hardwarebeschleuniger für selbst-organisierende Karten*. Dissertation, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, 2010. HNI-Verlagsschriftenreihe, Paderborn.

[Por09] Porrmann, M., Hagemeyer, J., Romoth, J., and Strugholtz, M. "Rapid Prototyping of Next-Generation Multiprocessor SoCs." In *In Proceedings of Semiconductor Conference Dresden, SCD 2009*. Dresden, Germany,, 2009.

[Pou04] Poussier, S., H., R., and Weber, S. "Adaptable thermal compensation system for strain gage sensors based on programmable chip." In *Sensors And Actuators A: Physical*, vol. 199. Elsevier, April 2004, pp. 412–417.

[Qui11] "QuickSilver technologies Homepage.", 2011. URL `http://www.qstech.com`.

[Rat] Rathmann, U. *Qwt - Qt Widgets for Technical Applications*. Http://qwt.sourceforge.net/.

[Raw01] Rawlings, J. O., Pantula, S. G., and Dickey, D. A. *Applied Regression Analysis: A Research Tool (Springer Texts in Statistics)*. Springer, April 2001.

[Rei03] Reinemann, T. and Kasper, R. "High Speed Implementation of Controllers and Filters for Mechatronic Systems." In *TechOnline*. 2003.

[Rei05] Reichör, S., Zeinzinger, M., and Pfaff, M. "Connecting reality and simulation: Couple high speed FPGAs with your HDL simulation." In *IP-SOC 2005: IP Based SoC Design Conference and Exhibition*. December 2005, pp. 271–275.

[Rex04] Rexroth Bosch Group, D-97816 Lohr a. Main. *Rexroth EcoDrive Cs Drives*, 2nd ed., November 2004. Document Number 120-1000-B344-02.

[Rey04] Reyneri, L. and Renga, F. "Speeding-up the design of HW/SW implementations of neuro-fuzzy systems using the CodeSimulink environment." In *Applied Soft Computing*, vol. 4. 2004, pp. 227–240.

[Ric03] Ricci, F. and Le-Huy, H. "Modeling and simulation of FPGA-based variable-speed drives using Simulink." In *Mathematics and Computers in Simulation*, vol. 63. 2003, pp. 183–195.

[Rog03] Roggen, D., Hofmann, S., Y., T., and Floreano, D. "Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot." In *ACM portal*. 2003.

[RT04] Romero-Troncoso, R., Herrera-Ruiz, G., Terol-Villalobos, I., and C., J.-C. J. "FPGA based on-line tool breakage detection system for CNC milling machines." In *Mechatronics*, vol. 14. 2004, pp. 439–454.

[Rue03a] Ruelland, R., Gateau, G., Meynard, T., and Hapiot, J. "Design of FPGA-Based Emulator for Series Multicell Converters Using Co-Simulation Tools." In *IEEE Transactions On Power Electronics*, vol. 18. January 2003, pp. 244–252.

[Rya10] Ryan, J. and Calhoun, B. "A sub-threshold FPGA with low-swing dual-VDD interconnect in 90nm CMOS." In *Custom Integrated Circuits Conference (CICC), 2010 IEEE*. sept. 2010, pp. 1 –4.

[Sag04] Saggini, S., Ghioni, M., and Geraci, A. "An Innovative Digital Control Architecture for Low-Voltage, High-Current DC-DC Converters With Tight Voltage Regulation." In *IEEE Transactions On Power Electronics*, vol. 19. January 2004, pp. 210–218.

[Sch06] Schlautmann, P. *Entwicklung eines neuartigen dreidimensionalen aktiven Federungssystems für ein Schienenfahrzeug*. Dissertation, University of Paderborn, 2006.

[Scr02] Scrofano, R., Choi, S., and Prasanna, V. "Energy Efficiency of FPGAs and Programmable Processors for Matrix Multiplication." In *The First IEEE International Conference on Field Programmable Technology (FPT)*. December 2002.

[Sev89] Sevcik, K. "Characterizations of parallelism in applications and their use in scheduling." In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 1989, p. 180.

[SFB11] "Homepage of the Collaborative Research Center 614.", 2011. URL `http://www.sfb614.de/en`.

[Sim10] Simpson, P. *FPGA Design: Best Practices for Team-Based Design*. Springer, 2010. ISBN 9781441963383.

[Sir10] Sirigir, V., Alzoubi, K., Saab, D., Kocan, F., and Tabib-Azar, M. "Ultra-low-Power Ultra-fast Hybrid CNEMS-CMOS FPGA." In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. September 2010, pp. 368–373.

[Šůc06] Šůcha, P., Kutil, M., Sojka, M., and Hanzálek, Z. "TORSCHE Scheduling Toolbox for Matlab." In *IEEE Computer Aided Control Systems Design Symposium (CACSD'06)*. Munich, Germany, October 2006, pp. 1181–1186.

[Šůc08] Šůcha, P. and Hanzálek, Z. "Deadline constrained cyclic scheduling on pipelined dedicated processors considering multiprocessor tasks and changeover times." In *Mathematical and Computer Modelling*, vol. 47, no. 9-10, 2008:pp. 925 – 942.

[Šůc09] Šůcha, P. and Hanzálek, Z. "A cyclic scheduling problem with an undetermined number of parallel identical processors." In *Computational Optimization and Applications*, 2009. URL `http://dx.doi.org/10.1007/s10589-009-9239-4`.

[Syna] "Synopsys Design Compiler." URL `http://www.synopsys.com`.

[Synb] "Synplify DSP." URL `http://www.synplicity.com`.

[SYN11] SYNOPSYS. "DesignWare Intellectual Property.", March 2011. URL `http://www.synopsys.com`.

[Sysa] "System-C." URL http://www.systemc.org/.

[Sysb] "System Generator." URL http://www.xilinx.com.

[Taz99] Tazi, K., Monmasson, E., and Louis, J. P. "Description Of An Entirely Reconfigurable Architecture Dedicated To The Current Vector Control Of A Set Of Ac Machines." In *IEEE International Conference on Industrial Electronics, Control, and Instrumentation*, vol. 3. Novembre 1999, pp. 1415–1420.

[Tes01] Tessier, R. and Burleson, W. "Reconfigurable Computing for Digital Signal Processing: A Survey." In *Journal of VLSI Signal Processing*, vol. 28. Elsevier, 2001, pp. 7–27.

[Tew02] Tewari, A. *Modern control design with MATLAB and SIMULINK*. John Wiley, 2002.

[Tho99] Thomas, F., Kishore, J. K., Bharadwaj, K. M., Nayak, M. M., and Agrawal, V. K. "Design and implementation of a wheel speed measurement circuit using field programmable gate arrays in a spacecraft." In *Microprocessors and Microsystems*, vol. 22. Elsevier Science, 1999, pp. 553–560.

[Tho06] Thompson, S. E. and Parthasarathy, S. "Moore's law: the future of Si microelectronics." In *Materials Today*, vol. 9, no. 6, 2006:pp. 20 – 25. ISSN 1369-7021. URL http://www.sciencedirect.com/science/article/pii/S1369702106715395.

[Tod05] Todman, T. J., Constantinides, G. A., Wilton, S. J. E., Mencer, O., Luk, W., and Cheung, P. Y. K. "Reconfigurable Computing: Architectures and Design Methods." In *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, Mar. 2005:pp. 193–207.

[Tom04] Tombs, M., Henry, M., and Peter, C. "From research to product using a common development platform." In *Control Engineering Practice*, vol. 12. Elsevier, 2004, pp. 503–510.

[Tos05] Toscher, S., Kasper, R., and Reinemann, T. "Dynamic Reconfiguration of Mechatronic Real-Time Systems Based on Configuration State Machines." In *19th International Parallel and Distributed Processing Symposium*. 2005.

[Tri97] Trimberger, S., Carberry, D., Johnson, A., and Wong, J. "A time-multiplexed FPGA." In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*. April 1997, pp. 22 –28.

[Tro] Trolltech. *Qt - cross-platform application framework*. Http://trolltech.com/.

[Vas04] Vasarhelyi, J., M., I., Szabo, C., Incze, I., and Adam, T. “FPGA Implementations of the Reconfigurable System for AC Drives.” In *11th International Conference on Power Electronics and Motion Control*. 2004.

[Ven04] Venkataramani, G., Budiu, M., Chelcea, T., and Goldstein, S. “C to asynchronous dataflow circuits: An end-to-end toolflow.” In *International Workshop on Logic Syntheiss*. 2004.

[Weg87] Wegener, I. *The Complexity of Boolean Functions*. B. G. Teubner, and John Wiley & Sons, 1987. URL `citeseer.ist.psu.edu/700371.html`.

[Wit03] Wittenmark, B., Å ström, K. J., and Å rzén, K.-E. “Computer Control: An Overview.” Professional Brief 1, International Federation Of Automatic Control, 2003. Http://www.ifac-control.org.

[Xil07a] Xilinx. “Virtex-II Platform FPGA User Guide.”, November 2007. URL `http://www.xilinx.com/`.

[Xil07b] Xilinx. “Virtex-II Platform FPGAs: Complete Data Sheet.”, November 2007. URL `http://www.xilinx.com/`.

[Xil07c] Xilinx. “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.”, November 2007. URL `http://www.xilinx.com/`.

[Xil08b] Xilinx, Inc., San Jose, USA. *ChipScope Users Guide*, 10.1 ed., 2008.

[Xil08c] Xilinx, Inc., San Jose, USA. *System Generator Users Guide*, 10.1 ed., 2008.

[Xil09b] Xilinx. “Virtex-4 FPGA Configuration User Guide.”, June 2009. URL `http://www.xilinx.com/`.

[Xil09c] Xilinx. “Virtex-6 FPGA Configurable Logic Block.”, September 2009. URL `http://www.xilinx.com/`.

[Xil10] Xilinx. “ug702 Partial Reconfiguration User Guide.”, May 2010. URL `http://www.xilinx.com/`.

[Xil11] “Xilinx Homepage.”, 2011. URL `http://www.xilinx.com`.

[Yao10] Yao, M. “Realization of Fuzzy PID controller used in turbine speed control system with FPGA.” In *Future Information Technology and Management Engineering (FITME), 2010 International Conference on*, vol. 1. October 2010, pp. 261 –264.

[Yin04] Yin, Y. and Zane, R. “Digital Phase Control for Resonant Inverters.” In *IEEE Power Electronics Letters*, vol. 2. June 2004, pp. 51–54.

[Yui02] Yui, C., Swift, G., and Carmichael, C. “Single event upset susceptibility testing of the Xilinx Virtex-II FPGA.” In *Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*. Maryland, USA, September 2002.

[Zum03] Zumel, P., de Castro, A., García, O., Riesgo, T., and Uceda, J. “Concurrent and Simple Digital Controller of an AC/DC Converter with Power Factor Correction.” In *IEEE Transactions on Power Electronics*. 2003, pp. 334–343.

List of Figures

2.1 Block diagram of a typical digital control system 8
2.2 Digital hardware platforms to implement digital controllers classified according to the level of specialisation [Des06] 9
2.3 Spatial and temporal implementation of a proportional-integral controller (PI), using trapezoidal integration (sketch based on [DeH00]) 10
2.4 Embedded software design flow 11
2.5 Typical ASIC design flow [Nek03] 12
2.6 Sketch of a typical FPGA architecture 15
2.7 Simplified view of a Virtex II slice [Xil07b] 16
2.8 Hierarchical routing resources [Xil07b] 17
2.9 Virtex-II Pro Generic Architecture Overview [Xil07c] 18
2.10 Typical FPGA-based design flow [Des06, Sim10] 19
2.11 Overview of the development of the feature size of FPGAs from Xilinx 20
2.12 Overview of the development of Low-cost FPGAs from Xilinx 21
2.13 Application distribution of the reviewed papers 21
2.14 Distribution of the contributing factors of FPGAs in the application spectrum 22
2.15 Binding time continuum [DeH99] 25
2.16 Cost of delayed entry into a market [Des06] 28
2.17 Different levels of coupling in a reconfigurable system[Com02] 31
2.18 Classification of reconfigurable hardware according to the reported coupling 31
2.19 Reconfigurable hardware and processor couplings in reference to the I/Os. a) Processor and FPGA have access to the ADC and DAC. b) Either an FPGA or a Processor is directly connected to the ADC and DAC with a tight communication with the other processing element. c) Either a Processor or an FPGA receive information form ADC and the other processing element is connected to the DAC. 32
2.20 Reported reconfiguration types for digital control applications . . . 34

3.1 Example of Cyclic Data Flow Graph 38
3.2 Example LH Graph 40
3.3 Scheduling for the LH graph depicted in figure 3.2. Three execution periods are shown, indicated by the different colors (gray scale) 41

3.4 Circuit area of operations listed in table 3.1 for various bit-widths . 43
3.5 Critical path delay operations listed in table 3.1 for various bit-widths 44
3.6 Normalised operations ($NormOp_{h,i}$), calculated using equation 3.2 . 45
3.7 Normalised steps ($Steps_{h,i}$), calculated using equation 3.3 45
3.8 Proposed Cyclic Data Flow Graph of a trapezoidal integration algorithm (cf. equation 3.8) 47
3.9 Cyclic scheduling for the trapezoidal integration depicted in figure 3.8 without weighting operations 47
3.10 Scheduling for the trapezoidal integration depicted in figure 3.8 using normalised synthesis results as time weight 48
3.11 Two simple LH Graphs to exemplify the concept of average operations per step (*AOS*). (a) has an *AOS* of 6/5, whereas (b) has an *AOS* of 6/3 50
3.12 Algorithmic characterisation of a trapezoidal numeric integration (see figure 3.8) using normalised synthesis results 51
3.13 (a) Virtex-II Slice (Top Half) [Xil07c], and (b)Virtex-6 SliceM (Top Half) [Xil09c] 53
3.14 Simplified block diagram of the PowerPC 750 RISC Microprocessor . 57
3.15 Simplified block diagram of the dSPACE DS1005 board 58
3.16 Radiagraphy of (a) an Spartan-3 XC3S1500, and (b) a Spartan-3 XC3S200 devices from Xilinx 59
3.17 Area measurement of various Xilinx FPGA 60
3.18 Power consumption of the PowerPC 750 CPU for various operation modes [IBM02] (power for 480 MHz was calculated using polynomial interpolation) 64
3.19 CDFG of a PID controller, using Simpson numerical integration. Node *A* represents an anti-windup algorithm, shown in figure 3.20 67
3.20 Implementation of a simple anti-windup strategy 67
3.21 Scheduling for the PID controller depicted in figure 3.19 without weighting operations 68
3.22 Algorithmic characterisation of a PID controller (see figure 3.19) . . 69
3.23 Implementation of a PID controller with Synplify-DSP 70
3.24 Implementation of a PID controller using Matlab/Simulink 71
3.25 Comparison of modelled and measured circuit growth rate 73
3.26 Algorithmic characterisation and Computational Density (C.D.) of hardware and software realisations of various versions of a PID controller 74
3.27 Energy Efficiency of hardware and software realisations of various versions of a PID controller (logarithmic scale is used for the Y axis) 75
3.28 Block diagram of a state-space representation of a system 76
3.29 Block diagram of a state-feedback controller 77
3.30 Cyclic Data Flow Graph of a feedback controller for a two-state two-input system, cf. equation 3.43 78

3.31 Scheduling for the state feedback controller depicted in figure 3.30 with operations not being weighted 79
3.32 Normalised algorithmic characterisation of a state-feedback controller (see figure 3.30) 80
3.33 Comparison of modelled and measured circuit grow rate 82
3.34 Computational Density of hardware and software realisations of a state-feedback controller 83
3.35 Energy Efficiency of hardware and software realisations of a state-feedback controller 84
3.36 Block diagram of a control system with a state observer 85
3.37 Cyclic Data Flow Graph of the state observer corresponding to equation 3.48 86
3.38 Scheduling for a state observer depicted in figure 3.37 without operation weighting. Three cycles are shown in the figure 87
3.39 Normalised algorithmic characterisation of a state observer 89
3.40 Comparison of modelled and measured circuit grow rate 92
3.41 Computational Density of hardware and software realisations of an state observer 93
3.42 Energy Efficiency of hardware and software realisations of an state observer 94

4.1 Classification of control adjustment methods in presence of faults [Lun06] 98
4.2 Diagram of a supervised multi-control system [Mor95] 99
4.3 Diagram of a supervised adaptive-control system [Mor95] 100
4.4 Structure of the Operator Controller Module (OCM) [14] 101
4.5 Static vs. Run-time full vs. run-time partial hardware reconfiguration 102
4.6 Configuration granularity of Xilinx FPGAs 103
4.7 Configuration Interface of the Virtex-II family [Xil07a],[Ket08] . . 104
4.8 Self- and external reconfiguration schemes 106
4.9 Virtex-4 Configuration Process [Xil09b] 107
4.10 Placement methods for 1D approach 109
4.11 1D vs. 2D placement methods 110
4.12 Point-to-Point (Bus Macros) vs Shared Lines (Embedded Macros) communication 111
4.13 Resource utilisation as described in both cases of equation 4.4 . . . 113
4.14 Schematic of a vector multiplier: specific implementation vs. general implementation 115
4.15 Resource utilisation of vector multiplier: specific implementation vs. general implementation based on a Virtex II FPGA (XCV4000) . . . 115
4.16 Block diagram of the RAPTOR64 system [Por09] 117

4.17 Schematic of the system architecture implemented on the RAPTOR system [15] . . . 118
4.18 Simplified schematic of the communication fabric [15] . . . 120
4.19 Schematic of the inverted pendulum system . . . 122
4.20 Schematic of the test-bed of the inverted pendulum system . . . 122
4.21 A simple strategy to swing-up the inverted pendulum . . . 123
4.22 Block diagram of the swing-up controller . . . 124
4.23 Block diagram of the balance controller . . . 124
4.24 Schematic diagram of a two-controllers system for the inverted-pendulum 125
4.25 Use of PR Regions for the inverted pendulum controller . . . 126
4.26 Controller exchange for the inverted pendulum system . . . 127
4.27 Schematic of the test-bed of the self-optimizing motion controller . 130
4.28 Schematic of a Field Oriented Control structure with back-EMF compensation and decoupling of currents [Nab80] . . . 131
4.29 Comparison of the execution-time of (b) CPU- and (c) FPGA-based motor controller realisations (FOC and FOC-EMF-DeC correspondingly), including execution times of the (a) static part of the controller [15] . . . 133
4.30 Flow of the run-time reconfiguration of controllers [15] . . . 135
4.31 Use of PR Regions for the self-optimizing motion control controller 135
4.32 Controller switching from FOC to FOC-EMF-Dec at 3000 RPM (HiL: Controller at FPGA, Motor in Simulation) [15] . . . 136
4.33 Controller switching from FOC to FOC-EMF-Dec at 3000 RPM without using initialisation (Test bed: Controller at FPGA, Motor as EUT) [15]. . . . 137
4.34 Controller switching from a CPU- and an FPGA-based realisations at 3000 RPM (Test bed: Controller at CPU and FPGA, Motor as EUT) [15] . . . 137

5.1 Positioning of FPGA-in-the-Loop simulations in the V design model 142
5.2 Combinations of simulated and real elements in the design process . 144
5.3 Information-flow of the Hardware-in-the-Loop simulation framework with CAMeL-View . . . 146
5.4 Information-flow of the Hardware-in-the-Loop simulation framework with Matlab/Simulink . . . 147
5.5 Synchronizer state machine . . . 148
5.6 Synchronizer embedded in the bus interface . . . 148
5.7 Simplified simulation flow diagram . . . 149
5.8 Simulation flow . . . 151
5.9 Coupling of host computer and RAPTOR. In this example a Pentium 4 with a 865G-Chipset is presented . . . 152
5.10 Maximum simulation frequency for a given number of input/output pairs 154

5.11 vMAGIC Designflow, reading and writing VHDL is optional. As such a vMAGIC application can be a pure VHDL generator or analyzer . 156
5.12 Toolflow for HiL simulations with Matlab/Simulink 158
5.13 Toolflow for FPGA-in-the-Loop simulations with CAMeL-View . . 159
5.14 Simulation results of a PID-based speed control: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 50 ms 160
5.15 Simulation results of a balance controller for an inverted pendulum: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 50 ms 161
5.16 Standard configuration of a PLL-based controller for piezo actuators 162
5.17 Simulation results of a phase controller for a piezo-actuator: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 10 μs 163
5.18 Mechanical model of the active suspension testbed [10] 164
5.19 Simulation results of a state observer for an Active Suspension Testbed: System Generator vs. HiLDE. The time-intervals of the enlarged figure correspond to 20 ms 165
5.20 Structure of a real-time FPGA-in-the-Loop scenario utilizing HiLDEGART 168
5.21 Main-, Log-, and Plot-Window of HiLDEGART. The GUI is generated from an XML file generated by a vMAGIC application 169
5.22 View of a HiLDEGART test using AR annotations 170
5.23 Tool flow for a real-time design verification using HiLDEGART . . . 171
5.24 q-current step response at 3000 rpm (Test bed: Controller at FPGA, Motor as EUT) 172
5.25 test 173
5.26 HiLDEGART measurements of the inverted pendulum system. Several reconfiguration cycles are shown 174
5.27 HiLDEGART test of the inverted pendulum system using an Augmented Reality extension. Balance controller is active 175

List of Tables

3.1 Set of basic operations 42
3.2 Used synthesis models from DesignWare library (Synopsys, [SYN11]) 43
3.3 Algorithm characterisation of a numeric trapezoidal integration . . . 49
3.4 Average operations per step (AOS_i) of a numeric trapezoidal integration for various bit-widths 51
3.5 Resource utilisation of various basic operations, synthesised for a Virtex II device (XC2V4000), using synthesis tools from Xilinx . . 54
3.6 Tools used for the software implementation of the benchmarks 61
3.7 Tools used for the software implementation of the benchmarks . . . 62
3.8 Algorithm characterisation of a parallel PID controller 69
3.9 Various FPGA-based implementations of a PID controller 71
3.10 Varios CPU-based implementations of a PID controller 72
3.11 Comparison of circuit size ($SizeAlg_i$) and utilised slices of various realisations of a PID controller 72
3.12 Various FPGA-based implementations of a state-feedback controller 80
3.13 Varios CPU-based implementations of a state-feedback controller . . 81
3.14 Comparison of circuit size ($SizeAlg_i$) and utilised slices of various 8-bit state-feedback implementations 82
3.15 Various FPGA-based implementations of a state observer 90
3.16 Varios CPU-based implementations of a state observer 91
3.17 Comparison of circuit size ($SizeAlg_i$) and utilised slices of various 8-bit state observer implementations 91

4.1 Configuration bandwidth for configuration ports in Virtex architectures [Xil10]([1] Virtex-4, Virtex-5 and Virtex-6. [2] Virtex-II and Virtex-II PRO) 105
4.2 Resource utilisation of swinging-up and balancing controllers, synthesised for Virtex II-Pro FPGA 125
4.3 Abilities and realisation aspects of motor controllers [15] 132
4.4 Resources of implemented FPGA-based controllers [15] 132

5.1 Implementation Examples 166

Glossary

Latin Symbols

lat_{alg} Latency of a CDFG, defined as the number of steps required to complete one iteration.

$NormOp_{h,i}$ Normalised operation.

A_{norm} Area normalised to a specific technology.

A_b Operative Area of the Balance controller.

A_{const} Hardware resources required for all configurations of a controller system (e.g., sensors signal processing).

$A_{dynamic_FS}$ Worst-case resource utilisation of a controller-system implemented using run-time reconfiguration, with a free placement partition approach.

$A_{dynamic_FS}$ Worst-case resource utilisation of a controller-system implemented using run-time reconfiguration, with a fixed-size slot partition approach.

A_{rec} Resources required to realise partial reconfiguration (e.g., for the communication infrastructure, and for the reconfiguration controller).

A_{static} Resource-requirements of a static implementation.

A_s Operative Area of the Swing-up controller.

A_{tb} Transition areas from the swinging-up controller to the balancing controller.

Adjaceny Matrix for a graph G with n edges is an $n \times n$ matrix where the non-diagonal entry a_{ij} is the number of edges from node i to node j, and the diagonal entry a_{ii} is the number of loops.

$Angular_Speed_{max}$ Maximum angular speed limit for the transition of controllers of the inverted pendulum system.

$Angular_Speed_{min}$ Minimum angular speed limit for the transition of controllers of the inverted pendulum system.

$Area$ Chip area of a computing device.

B_{const} Additional hardware resources, required for instance to process I/O signals.

$B_{dynamic_FP}$ Worst-case resource utilisation of a parametrisable controller when using run-time reconfiguration with a free-placement slot approach.

$B_{dynamic_FS}$ Worse case resource utilisation of a parametrisable controller when using run-time reconfiguration with a fixed-size slot approach.

$B_{general}$ The resource requirements of a parameterisable structure.

B_{size} Amount of required configuration bits.

B_{static} The resource requirements of a static structure.

BPC Transferred bit per cycle.

C Set of n controllers with different structures.

$C_{density}$ Computational Density, using normalised values.

$C_{density,DeHon}$ Computational Density, defined as a throughput/area ratio.

c critical circuit in a graph G.

$Depth_{n,i}$ Critical path delay of a normalising operation $n \in BO$ using i number of input bits.

$DepthOp_{h,i}$ Critical path delay of an operation $h \in BO$ using i number of input bits.

$E_{efficiency}$ Energy Efficiency, defined as a throughput/power ratio.

F_{sim} Maximum simulation frequency.

$init_s_i$ initialization time of the ith specific realisation.

$init_i$ Initialisation time of a controller.

$Instructions$ is the number of repetitive operations required for a controller.

n_o output ports.

$\tilde{n}_r$ Overall number of read accesses using event based communication.

n_r Number of read operations in the standard HiLDE wrapper.

p_s_{max1}	Control period of the largest specific realisation.
p_{max1}	Control period of the largest controller $c_{max1} \in C$.
PAR	The set of n parameter-groups for a control structure.
AOS_i	Average operations per execution step. It is an approximation of average parallelism..
$Power$	Power consumption of a computing device when executing a specific controller.
$Power_{norm}$	Normalised power consumption.
S	Set of n resource-requirements of all controllers in C.
S_{area}	Area normalisation factor.
S_{delay}	Delay normalisation factor.
s_i	Resource-requirements of the ith controller $\in C$.
S_{power}	Normalisation factor for power consumption.
s_{max1}	Resource-requirements of the largest controller $c_{max1} \in C$.
s_{max2}	Resource-requirements of the second largest controller $c_{max2} \in C$.
$SizeAlg_i$	Size of an algorithm, calculated by adding the size of all individual operations for a specific number i of input bits.
$SizeOp_{n,i}$	Chip area of a normalising operation $n \in BO$ using i number of input bits.
$SizeOp_{h,i}$	Chip area of a given operation $h \in BO$ when using i number of input bits.
$Steps_{h,i}$	Number of equivalent steps required to compute an operation.
t_{conf}	Reconfiguration time.
T_{con}	execution time of a controller, when realised as software or hardware.
T_{freq}	clock frequency of the packet processor.
$tconf_s_i$	Is the reconfiguration time of the ith specific realisation.
$tconf_i$	Reconfiguration time of a controller.
$Throughput$	Control cycles per second.

$Throughput_{norm}$ Normalised throughput.

$Throughput_{general}$ Operations per execution time.

T_{hw2sw} conversion-times from a hardware-specific to a simulator-internal number representation.

tr_s_i Reaction time of the ith specific realisation α_i.

tr_s_{max1} Reaction time of the largest specific realisation of *PAR*.

tr_i Reaction time of a controller.

tr_{max1} reaction time of the largest controller $c_{max1} \in C$.

$T_{receive}$ transfer-times from the prototyping system to the main memory of the host.

T_{run} latency of the design under test.

T_{send} transfer-times from the main memory of the host to the prototyping system.

T_{sw2hw} conversion-times from a simulator-internal to a hardware-specific number representation.

U Set of n resource-requirements of initialisation routines of all controllers in C.

u_i Resource-requirements of the initialisation routines of the ith controller $\in C$.

u_{max1} Resource-requirements of the initialisation routine of the largest controller $c_{max2} \in C$.

u_{max2} Resource-requirements of the initialisation routine of the second largest controller $c_{max2} \in C$.

$V_{dd,ref}$ voltage input of the reference architectures' core.

V_{dd} voltage input of the architectures' core.

WL represents the word length of an ALU.

$wordwidth$ Word width of the port in a HiLDE simulation.

w Period of a graph with periodic scheduling, which is given by the critical circuit of that graph.

Greek Symbols

α_{max1}	Largest specific implementation of a multi-controller system with one structure and a set of n parameter groups.
α_{max2}	Area required for the second largest specific controller.
γ_{max1}	Area required for the initialization routine of the largest specific controller.
$\Delta(out)$	Number of output ports with a new value.
λ	Half the minimum drawn feature size on a process.
λ_{ref}	λ of the reference technology.

Abbreviations

ADC	Anolog to Digital Converter.
ALU	Arithmetic Logic Unit.
API	Application Programming Interface.
AR	Augmented Reality.
ASIC	Application Specific Integrated Circuit.
AST	Abstract Syntax Tree.
BCS	Basic Cyclic Scheduling.
BO	Basic Operations.
BPU	Branch Processing Unit.
BR	Base Region.
CDFG	Cyclic Data Flow Graphs.
CLB	Configurable logic block.
CMOS	Complementary Metal Oxide Semiconductor.
CNC	Computerised Numerically Control.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
DAC	Digital to Analog Converter.
DC	Direct Current.
DCM	Digital Clock Manager.

DDR	Double Data Rate.
DDS	Direct Digital Synthesis.
DEC	Decoupling.
DMA	Direct Memory Access.
DR	Dynamic Reconfiguration.
DSP	Digital Signal Processor.
DUT	Design Under Test.
EEPROM	Electrically Erasable Programmable Read-Only Memory.
EMF	Electromagnetic Field.
EPROM	Erasable Programmable Read-Only Memory.
EUT	Equipment Under Test.
FDI	Fault Detection and Isolation.
FF	Flip-Flop.
FFT	Fast Fourier Transform.
FIFO	First In – First Out (Memory).
FOC-EMF-DeC	Field Oriented Control with back Electromagnetic Field compensation and Current Decoupling.
FOC-EMF	Field Oriented Control with back Electromagnetic Field compensation.
FOC	Field Oriented Control.
FPGA-iL	FPGA-in-the-Loop.
FPGA	Field Programmable Gate Array.
FPU	Floating Point Unit.
FSR	Full Signal Range.
GCLK	Global Clock.
GPIO	General Purpose Input/Output.
GUI	Graphical User Interface.
HiLDEGART	HiLDE for Guided Active Real-Time Test.

HiLDE	Hardware-in-the-Loop Design Environment.
HiL	Hardware in the Loop.
Hw/Sw	Hardware and Software.
I/Os	Inputs/Outputs.
ICAP	Internal Configuration Access Port.
IOB	Input Output Block.
IOI	Input Output Interconnections.
IP	Intellectual Property.
IU	Integer Unit.
JTAG	Joint Test Action Group.
LB	Local Bus (RAPTOR System).
LQR	Linear Quadratic Controller.
LSU	Load/Store Unit.
LUT	Lookup table.
MAC	Multiply-ACcumulate.
MiL	Model-in-the-loop.
MIMO	Multi-Inputs Multi-Outputs.
NRE	Non-recurring engineering.
OCM	Operator Control Module.
OFDM	Orthogonal Frequency Division Multiplexing.
PCI	Peripheral Component Interconnect.
PC	Personal Computer.
PID	Proportional-Integral-Derivative (Controller).
PIO	Programmable Input Output.
PLB	Processor Local Bus.
PLC	Programmable Logic Controllers.
PLL	Phase-Looked-Loop.

PPC	PowerPC.
PRR	Partially Reconfigurable Region.
PWM	Pulse-Width Modulation.
QL	Quantisation Level.
RAM	Random Access Memory.
RFNN	Radial Functions Neural Networks.
RH	Reconfigurable Hardware.
RISC	Reduced Instruction Set Computer.
RTL	Register Transfer Level.
RTR	Run-Time Reconfiguration.
RT	Reconfigurable Tile.
SiL	Software-in-the-Loop.
SoC	System on Chip.
SRAM	Static Random Access Memory.
SRU	System register unit.
SSI	Serial Synchronous Interface.
TTM	Time-to-Market.
UART	Universal Asynchronous Receiver Transmitter.
VCM	Virtex Configuration Manager.
VHDL	Very-high-speed integrated circuit Hardware Description Language.
vMAGIC	VHDL Manipulation and Generation Interface.