Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Roman Plessl

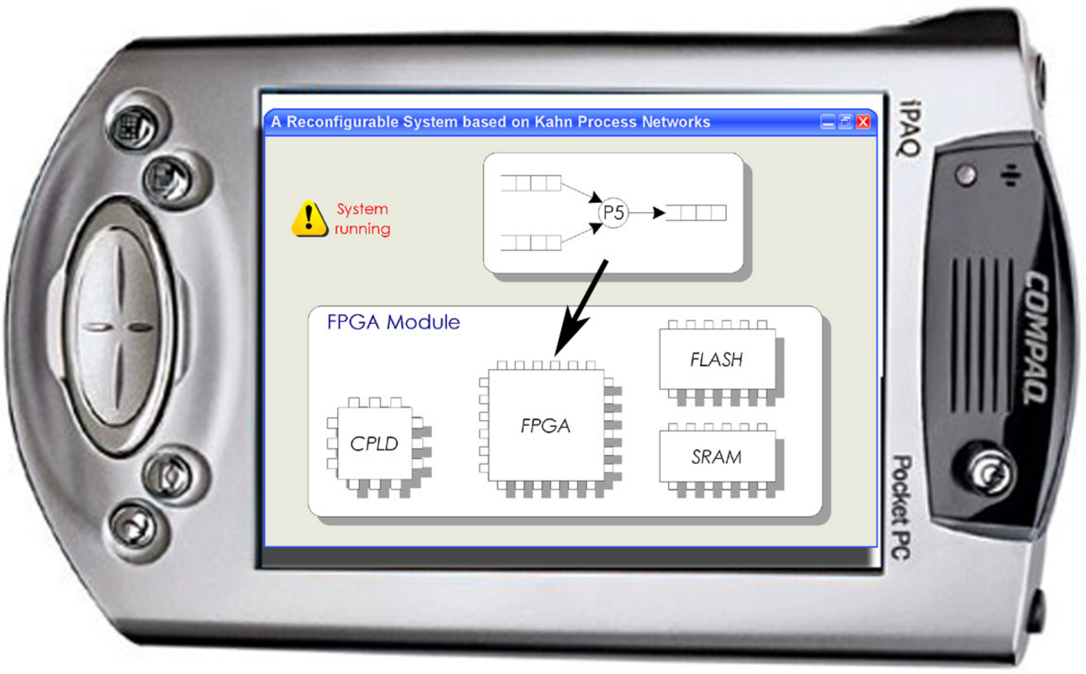
Embedded Machine on FPGA

Masters Thesis MA-2004-3
Winter Term 2003/2004

Tutor: Matthias Dyer

Supervisor:
Prof. Dr. Lothar Thiele

30.4.2004



Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Abstract

Embedded systems like an IPAQ handheld keep getting more complex from release to release. In the last updates they are augmented with improved video and audio playback capabilities. As a consequence the computation demand on the device is massively increased and the main CPU needs to be supplemented by a video-decoder ASIC.

Alternatively to such an ASIC a dynamically reconfigurable Field-Programmable Gate Array (FPGA) could be used. An FPGA is reconfigurable while the rest of a running system is unaffected. The device provides an flexible and powerful architecture for implementing computation intensive applications. Including an FPGA it is even possible to update an handheld with a forthcoming video-codec at a later date.

In this theses I show an implementation approach of an heterogeneous systems consisting of an IPAQ and an FPGA-based hardware extension. As a mathematical foundation I have used the Kahn Process Network which is very suitable for parallelism and reusability in signal processing and media applications.

I explain in the text the steps it tooks to design and implement the current execution unit for Kahn hardware tasks, to upload new configurations to the extension board, to reconfigure the system triggered by an software event from the IPAQ and finally to implement an application and run it in the framework.

Preamble

This text is the report of my master thesis which I have done at the Computer Engineering and Networks Laboratory of the ETH Zürich. In this memorandum the theoretical and practical work during the last 6 month is documented. The background idea when writing this report was a reusability of my work in further thesis.

Chapter 1 of this report offers an insight into the motivation and a brief overview of the thesis topics. The goals which should be finally achieved are mentioned at the end of this chapter.

A comparison of our approach using a Kahn Process Network for modelling hardware task with related works and working systems is done in **Chapter 2**.

The Kahn Process Network is the used model of computation which is introduced in **chapter 3**. A short introduction to other possible models is done in this chapter too. Finally there is a brief comparison between different dataflow languages at the end of this chapter.

A «Big Picture» of the architecture, i.e. the implemented system and the used components is given in **chapter 4**.

Chapter 5 explains the detailed implementation of the environment and running system. Most of the concepts and implementation know-how are documented here.

Some sample cores for the Kahn Task Execution Framework have been implementated and their performance is elaborated in **Chapter 6**.

The status of this thesis and possible future works are described in **Chapter 7**.

Finally the appendix gives several additional background information to the system development.

Acknowledgements

First of all, I want to thank **Matthias Dyer**. He was my advisor during the last six months and guides me through my master thesis. He gave me a great support with ideas for the Kahn Process Network and a possible mapping of this model to an FPGA execution framework. His experience with the IPAQ, the BTNodeFPGA and reconfigurable devices have been very helpful. He was also right at hand for questions concerning the presentation and the documentation.

I thank **Prof. Dr. Lothar Thiele** that I could do this thesis in his research group. The provided knowledge, the infrastructure and the debugging equipment were important key points for this successful work.

As a next point, I want to thank the other students who are working in the same room during my thesis. We have cultured a *brain-pool* for sharing the knowledge and experience for handling the hardware development environment.

Special thanx to the members of the *Badger-Mushroom-Snake-Gang* — the students who are working concurrently to my thesis on a Hardware OS for the XFBOARD: **Samuel Nobs** (he is *the* «7-guy» par excellence, without his help the CPLD design wouldn't run till now. . .), **Simon Steinegger**, **Kristofer Jonsson** and **Silvan Wegmann** (he has discovered the badger-badger-badger song).

A thank you is aimed to my *DDoSVax* friends **Stéphane Racine** and **Lukas Hämmerle**. We had a lot of interesting discussions during the coffee breaks and at lunch time.

Finally, a huge thank to my parents **Esther Plessl** and **Andreas Plessl** for facilitating and supporting me during my studies. Also a big thanx to **Jeannette Burkhardt** for supporting me with food during my implementation and write-attacks and for correcting parts of this documentation `:-)`.

Zürich, 30th April 2004

Roman Plessl

Contents

Preamble *iii*

Abstract *iii*

Contents *viii*

List of Figures xi

List of Tables xiv

List of Listings xv

1 Introduction *1*

1.1 Introduction and Motivation 1

1.1.1 Kahn Process Networks 2

1.1.2 Runtime Environment 3

1.1.3 FPGA internals 3

1.2 Thesis Assignment 4

1.2.1 General Motivation 4

2 Related Work *7*

2.1 Overview – Approaches to Virtualize Hardware 7

2.2 Coordination Languages 8

2.3 Comparable Approaches to our Design 8

3 Model of Computation *11*

3.1 Dataflow Languages 11

3.2 Kahn Process Networks - KPN 11

3.2.1 Determinism 12

3.2.2 Execution 12

3.2.3 The features of a Kahn Process Network 14

3.3 Alternative Models of Computation 14
3.3.1 Synchronous Dataflow - SDF 15
3.3.2 Binary Data Flow - BDF 15
3.3.3 Dynamic Data Flow - DDF 15
3.4 Differences between KPN and SDF 16
3.5 Why we use KPN for FPGA designs 17

4 Architecture *19*
4.1 Overview 19
4.2 Hardware 19
4.2.1 BTNodeFPGA 19
4.2.2 IPAQ 21
4.2.3 Behaviour of the BTNodeFPGA board 23
4.3 Slotmanager 25
4.4 Upload of new bitstreams 27
4.5 Reconfiguration of the FPGA 28
4.5.1 Reconfiguration Toolflow 28

5 Detailed Implementation *31*
5.1 Overview 31
5.2 Concepts 31
5.2.1 Implementation of the FIFOs in SRAM 31
5.2.2 FLASH Configuration 32
5.2.3 Internal Clock Generation 32
5.3 Slotmanager 36
5.3.1 slotmanager_state 38
5.3.2 lut_address_fsm 41
5.3.3 address_register 42
5.3.4 cpucommunication 45
5.3.5 task_wrapper 51
5.3.6 Synthesis & Routing Data 53
5.4 Loader on FPGA 55
5.4.1 System Overview 55
5.4.2 Commands (Serial and CPU Protocol) 57
5.4.3 Loader - 8 Bit Transfer Implementation 58
5.4.4 Loader - 16 Bit Transfer Implementation 62

5.5 CPLD Design 66
5.5.1 Application of the CPLD 66
5.5.2 Final States Machine 68
5.5.3 Synthesis & Routing Data 70
5.6 Scan Chain 71
5.6.1 Introduction 71
5.6.2 Scan Chain in VLSI designs 72
5.6.3 Scan Chain in FPGAs and Related Work 73
5.6.4 Scan Chain Implementation 74
5.6.5 Tool Flow 74
5.6.6 Conclusions 76
5.7 Acquisition of Asynchronous Data - Handshaking 76
5.7.1 Acquisition of a single input signal 76
5.7.2 Acquisition of vectored input signals 77
5.7.3 Metastability at Clock Boundaries 77
5.8 Kahn Task Template 80
5.9 IPAQ Program 80
5.10 Toolflow 81
5.10.1 Bitstream Generation 81
5.10.2 Modifying of Bitstream 82

6 Application and Performance *83*

6.1 Possible Applications 83
6.2 Example Cores inclusive Performance 86
6.2.1 AES Core - encoding and decoding 86
6.2.2 DES Core - only encoding 92
6.2.3 CRC32 Core 96
6.3 Performances 102
6.3.1 Reconfiguration-Performance 102
6.3.2 Slotmanagager-Performance 102
6.3.3 ScanChain-Performance 102
6.3.4 Cores-Performance 102
6.4 Synthesis Tutorial for the Demo Tasks 103

7 Status and Future Work *105*
7.1 Status of the Project Tasks . . . 105
7.1.1 Survey: Actual State of Project Tasks . . . 107
7.2 Future Work . . . 108
7.2.1 Improved FLASH Programming Procedure . . . 108
7.2.2 Scan-Chain Script Improvements . . . 110

A Appendix *111*
A.1 Development Environment . . . 111
A.2 VHDL Coding Guidelines . . . 112
A.3 VHDL Error Hotlist . . . 113
A.4 Scan Chain . . . 113
A.5 Entities and UCF Files . . . 117
A.5.1 FPGA: *Xilinx SPARTAN^{TM} II* XC2S200 . . . 117
A.5.2 CPLD: *Xilinx Coolrunner^{TM} XPLA3* XCR3384XL . . . 119

B IPAQ Driver and Data Streamer *123*

C Tools *147*
C.1 bitstreambuilder.pl . . . 148
C.2 streamer.pl . . . 152
C.3 listener.pl . . . 154
C.4 EdifToXML.pl . . . 156
C.5 XMLToEdif.pl . . . 159

D Contents of the CD *161*

E Glossary *163*

F Bibliography *167*

List of Figures

Preamble Introduction

1-1 An example of a KPN 2
1-2 Envisioned Runtime System 3

Related Work *(no figures)*

Model of Computation

3-1 A system that fails under data-driven scheduling 13
3-2 A system that fails under demand-driven scheduling 13
3-3 Overview of the different dataflow languages 14

Architecture

4-1 Big Picture about the System 20
4-2 BTNODEFPGA: Extension Board 20
4-3 BTNODEFPGA: electronic parts 22
4-4 BTNODEFPGA - Bridge: Connection to IPAQ 22
4-5 BTNodeFPGA: Additional Documentation and Execution Framework Specifics 24
4-6 Slotmanager: Overview 25
4-7 Upload of new bitstreams - Configuration flow 27
4-8 Slotmanager: Reconfiguration of the FPGA 28

Detailed Implementation

5-1 SRAM: Implementation of the FIFO 32
5-2 Flash: Storing scheme bitstreams 33
5-3 Waveforms of the IPAQ extension interface 34
5-4 IPAQ extension port waveforms and timings 35
5-5 Internal Clock Generation in the FPGA 36
5-6 FPGA - Waveforms of fetching an IPAQ data word 37
5-7 Slotmanager - Overview of the components 37
5-8 Slotmanager - FSM 39

5-9 Slotmanager - Configuration FSM 40
5-10 Address Generation Unit 44
5-11 Address Generation Unit - FSM 45
5-12 PortID to FifoID Translation Table 48
5-13 Slotmanager Design : Project View in the ISE 53
5-14 Loader on FPGA - DataFlow 56
5-15 Loader on FPGA - ControlFlow 56
5-16 Loader on FPGA: FSM for the CPU Protocol (8 Bit) 60
5-17 Data Transfer Protocol 61
5-18 Loader on FPGA: FSM for the CPU Protocol (16 Bit) 63
5-19 Loader on FPGA: Mealy automata for data sender 64
5-20 Transfer Protocol 64
5-21 Loader on FPGA: FSM for the SERIAL Protocol 65
5-22 CPLD: Command & Data Transfer 67
5-23 RS-232 data transfer through the CPLD 67
5-24 CPLD - FSM as used in the 8-bit mode 69
5-25 Explicit state saving 71
5-26 Modifying a FlipFlop for Scan 74
5-27 Concept: Scan chain 75
5-28 Concept: Scan Mode 75
5-29 Realisation scan chain 75
5-30 Tool-flow ScanChain 76
5-31 Handshaking over Clock Boundaries 78
5-32 Handshaking over Clock Boundaries: Protocol 78
5-33 Handshaking over Clock Boundaries: Synchronisation Howto 79
5-34 Metastability Wave in CMOS Technology 79
5-35 Task Template: Structural Overview 80

Application and Performance

6-1 Application: Cipher encoding in HW 84
6-2 Application: Pattern recognition in Mobile NetFlow Data 84
6-3 Application: Pattern recognition on scanned or photographed images . 85

Status and Future Work

7-1 Waveforms for writing to the FLASH 109

Appendix

A-1 Waveform of a scan chain in action 115

A-2 Example of a scan chain 116
A-3 Entity: FPGA for the BTNODEFPGA board 117
A-4 Entity: CPLD for the BTNODEFPGA board 120

IPAQ Driver and Data Streamer *(no figures)*

Tools *(no figures)*

Contents of the CD *(no figures)*

Glossary *(no figures)*

Bibliography

List of Tables

Preamble Introduction

1-1 Two Levels of Abstraction for an Application 4

Related Work *(no tables)*

Model of Computation

3-1 Comparison between the KPN and SDF 16

Architecture *(no tables)*

Detailed Implementation

5-1 Bitstream storing scheme for Flash memory 33

5-2 Communication IPAQ to FPGA KPN: Command Words 50

5-3 Communication To FPGA:Synchronisation words 57

5-4 Communication to FPGA and CPLD - Command Words 57

5-5 Communication to CPLD and FPGA - Select Sectors 58

Application and Performance

6-1 AES Core Datas 86

6-2 Results after synthesis with available synthesis tools and after place & route 91

6-3 DES Core Datas 92

6-4 CRC32 Core Datas 96

6-5 Performance of the Loader and for Reconfiguration 102

6-6 Performance of Slotmanger on BTNODEFPGA-Board 102

6-7 Performance of the Scan Chain 102

6-8 Performance of Example Cores for Slotmanger 103

Status and Future Work

7-1 Survey: Actual state of Project Tasks 107

7-2 «scan-able» flip-flop equivalents 110

Appendix

A-1 Coding Style VHDL 112

A-2 LUT Elements ScanChain 114
IPAQ Driver and Data Streamer *(no tables)*
Tools *(no tables)*
Contents of the CD *(no tables)*
Glossary *(no tables)*
Bibliography

List of Listings

Preamble Introduction *(no listings)*
Related Work *(no listings)*
Model of Computation *(no listings)*
Architecture *(no listings)*
Detailed Implementation
5-1 Slotmanager State Entity 38
5-2 Entity for the LUT Address FSM 41
5-3 Generation of LUT Address for the FIFOs 41
5-4 Entity – Address Register 43
5-5 CPU Communication Entitiy 46
5-6 CPU Output Data 47
5-7 CPU Communication: Port to FIFO 48
5-8 CPU Communication: StateBRAM 49
5-9 CPU Communication: BRAM 49
5-10 Task Wrapper 51
5-11 Task Chainer 52
5-12 Task Entity of the VHDL Task Template 80
5-13 Modified Bitstreams 82
Application and Performance
Status and Future Work
7-14 VHDL for writing to the FLASH 108
Appendix
A-15 UCF File for the FPGA and ISE 6.1 119
A-16 UCF File for the CPLD and ISE 6.1 121
IPAQ Driver and Data Streamer
Tools
Contents of the CD
Glossary
Bibliography

1

Introduction

1.1 Introduction and Motivation

Communication *embedded systems*, which occasionally have a high computation demand (e.g. for audio-/video processing, cryptography or DSP), are heterogeneous devices. They are traditionally equipped with a instruction set processor (ISP) and a set of hard-wired ASICs such as an mpeg2-decoder. Alternatively these ASICs could be replaced by a single *Field-Programmable Gate Array (FPGA)*.

FPGA-based designs become popular because of their *reconfigurable* capability and short design-time which the old design style, like ASICs cannot offer. Instead of using FPGAs simply as ASICs replacements, treating these reconfigurable devices as a dynamic resource provides an even more *flexible* and *powerful* approach for implementing computation intensive applications. Furthermore, some of the newer FPGAs provide partial reconfiguration where only one part of the FPGA is reconfigured while the rest remains unaffected and operating. Such a system is comparable to a multi-processor architecture, which executes different processes truly in parallel.

It is now quite common for the design of embedded applications to use a *model of computation* to provide the programmer with the possibility to check the correctness of the application in terms of functionality and timing. *Kahn process networks* (KPNs) are a popular modeling technique for media- and signal-processing applications. A KPN makes *parallelism* and *communication* in an application explicit and thus are very suitable for architectures which can exploit parallelism such as FPGAs. Furthermore, KPNs have a straightforward, unambiguous semantics and they are fully compositional. Therefor, KPNs facilitate the reuse of application models, which is becoming increasingly important for reducing the design time of new products and services.

In a FPGA-based system with partial and dynamic reconfiguration it is possible to execute some arbitrarily large applications on a smaller device by *splitting them up into smaller pieces and using time-multiplexing*. This idea has recently been addressed by [28],[21] and [30]. But these implementations use either a static scenario, where the specified design is compiled by an off-line tool-flow, or they do not rely on a specific model of computation.

This work targets an implementation of a reconfigurable system with KPN as the underlying model of computation. The envisioned system should be able, in contrast to existing implementation, to

- parse the abstract KPN description of the application at runtime,
- execute the pre-compiled processes of the application using dynamic reconfiguration and to
- remain the properties of the KPNs.

1.1.1 Kahn Process Networks

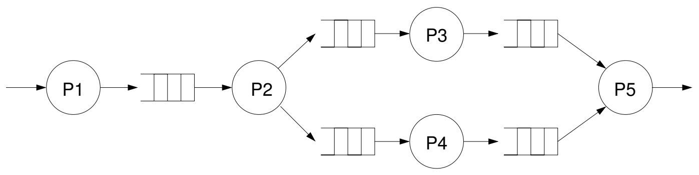


Figure 1-1
An example of a KPN

A KPN is represented as a collection of cooperating processes that communicate via streams of data (see figure 1-1). The processes can be executed in parallel but internally they are assumed to be sequential. The communication to the fifos is done by blocking reads and non-blocking writes. The model therefor implies unbounded fifos. An important restriction of a Kahn process is that it can not test an input queue on the availability of data without blocking. With this restriction, the output sequence generated by the KPN is deterministic and independent of the execution order (schedule) of the processes.

KPNs assume unbounded fifos. However, there exist classes of KPNs, which can be executed with a finite amount of memory. One idea, described in [3] and [13], is to use blocking writes to full fifos and a data-driven scheduling. A new problem arises with this approach. One has to find the proper initial sizes for the fifos for an efficient execution of the KPN. If some fifos are chosen too small, the execution of the KPN will end in an artificial deadlock. Since the amount of memory required for fifos cannot be decided analytically, a scheduler must provide the means to dynamically increase channel capacity where needed.

1.1.2 Runtime Environment

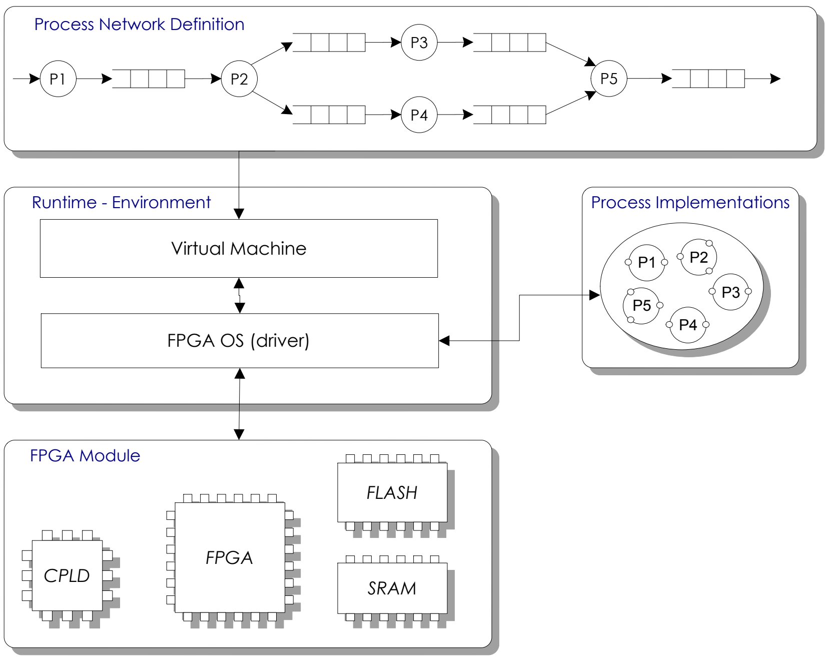


Figure 1-2
Envisioned Runtime System

Figure 1-2 shows an overview of the runtime environment. It consists of two parts:

Virtual Machine: The *Virtual Machine* is an interpreter for the description of the process network. It can load and unload applications. Furthermore it controls the underlying FPGA operating system.

FPGA OS: The *FPGA Operating System* has mainly three tasks:

- Allocation of memory for the channels.
- Scheduling of the KPN.
- Loading of task binaries.
- (Re-)configuration of the FPGA.

1.1.3 FPGA internals

To execute a Kahn Process Network on a FPGA, there is needed a well partitioned framework, which:

- provides a task slot with a defined interface for the Kahn Processes,
- maps the in- and output ports of the processes to the right memory locations,
- handles the communication with the scheduler and
- provides a mechanism to save and load the context of the processes.

1.2 Thesis Assignment

The follow section includes the assignments for this master thesis:

1.2.1 General Motivation

Embedded systems – like «handhelds» and «mobile phones» – are usually hetrogenous systems. In the last years the demand for high computation capabilities are still rising (e.g. for audio- or video processing, image compression or cryptography). Traditionally these requirement are fullfilled with an integrated ASIC. Over the past years, many custom computing machines have been presented that achieve high performance by coupling general-purpose CPUs with field-programmable logic. In the area of portable or smart reconfigurable devices performance, power efficiency and flexibility are the key requirements. Flexibility is required in order to distribute new applications and services on multiple heterogeneous platforms. As a consequence we need to support reconfigurability and to reduce the device dependence. There is a trade-off between flexibility and performance, since they are conflicting criteria.

Flexibility is achieved by virtualizing hard- and software components which could be done on different levels.

Following a model-based design approach, an application is explicitly specified by a directed graph where the nodes are tasks, which represent computations and the arcs represent communication. The tasks are arbitrary subprograms and are specified in a conventional programming language such as C or VHDL, but the interaction between tasks is defined by a precise semantics. We call the language defining this interaction the *coordination language*.

Examples of coordination languages are *process networks* or *synchronous dataflow (SDF)* graphs. In model-based design models of computation are frequently used to specify applications.

Abstraction Level	**Languages**	**Elements**
Coordination & Communication	Process Networks, SDF	Network Graphs, Tasks
Function of Tasks	VHDL, C	Instructions, Gates

Table 1-1: Two Levels of Abstraction for an Application

A language can be either interpreted or compiled. Our approach is to *virtualize and interpret only the coordination language* while *the tasks function is compiled*. Therefore, an application is specified in two parts.

The first one describes the coordination, i.e. the communication and the synchronization between tasks. The second part is the set of pre-compiled tasks, e.g. in the form of complete configuration bitstreams for FPGAs.

Our target architecture is a reconfigurable embedded node, including a CPU (IPAQ) and reconfigurable logic (BTNodeFPGA). We use the Kahn Process Network as a coordination language and the task a implemented in VHDL and Verilog.

Problem Task (german)

1. Erstellen Sie in den ersten zwei Wochen zusammen mit Ihrem Betreuer einen realistischen Zeitplan, welcher Meilensteine festlegt.
2. Arbeiten Sie sich in die Grundlagen von FPGAs und VHDL ein und machen Sie sich mit der Entwicklungsumgebung vertraut. Implementieren Sie als Übung eine einfache Testschaltung.
3. Lesen Sie sich in die Thematik von KPNs und Scheduling von KPNs ein.
4. Erstellen Sie ein Konzept für ein FPGA-KPN-Framework. Definieren und dokumentieren Sie genau die Schnittstellen zwischen den einzelnen Komponenten.
5. Erstellen Sie einen *Proof of Concept* mit dem zur Verfügung stehendem FPGA Modul. Als erste Implementierung sollte ein einfaches Design angestrebt werden, mit nur einem Taskslot und ohne partielles Rekoniguieren.
6. Implementieren Sie eine kleine Demoanwendung, mit welcher Sie das Funktionieren des Frameworks vorzeigen können.
7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Diplomarbeit

Allgemeines

- Der Verlauf des Projektes Diplomarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Diplomarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern.

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *30. April 2004* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen

usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

2

Related Work

2.1 Overview – Approaches to Virtualize Hardware

During the implementation of the Kahn Process Network execution framework an «introduction and survey of hardware virtualization» was written by C. Plessl [25] – a Ph.D. student of the Institute of Computer Engineering and Networks Laboratory. The paper covers many aspects of this «related work» chapter for completeness we like to resume the most important considerations.

The paper presents an introduction to the different methods of hardware virtualization on reconfigurable devices. The approaches can be classified in the following groups: *temporal partitioning*, *virtualized execution* and *virtual machine*.

A *temporal partitioning* splits an application into smaller parts, so that they can be mapped on a reconfigurable device with insufficient hardware capacity. The splitted parts are scheduled and executed sequentially.

Virtual execution allows a certain level of device-independence. An application is mapped to a programming model. This model defines some atomic units of computation — the tasks, which are «compiled» for a determined device family. An application is described by a collection of such tasks and their interactions (communication channels). Several tasks may run concurrently on the same device and the execution is handled by a scheduling algorithm in most of the cases.

The abstraction of a *virtual machine* achieves an even higher level of device-independence as the approach of virtual execution. Instead of mapping an application to a specific architecture, the application is mapped to an abstract computing architecture. For execution this abstract architecture has to be remapped or interpreted in analogy to the Java virtual machine.

The disadvantage of totally virtualizing an FPGA is the overhead and performance loss, arisen from the virtualization on the level of gates and interconnections. The Kahn Process Network execution framework is a member of the *temporal partitioning* and *virtual execution* group and so we like to compare our approach and implementation with papers and implementations of these groups.

2.2 Coordination Languages

Application models in embedded systems often use a *coordination language* which are semantically well-defined and usually more restricted than a high level language. An advantage of such a coordination language is, that they often allow a formal analysis of system properties (absence of deadlocks, maximum throughput or optimal scheduling).

In the majority of the cases coordination languages are used in combination with *implementation languages*. Large applications are splitted in smaller tasks – the computation objects. The interaction between these tasks is modelled using the coordination language. The functionality of the tasks is defined in an implementation language like C or VHDL.

The idea of this approach is to decompose an application into communication channels and tasks which may run concurrently – as an example – on a reconfigurable architecture. The tasks are implemented by conventional synthesis and design implementation tools. The communication channels are implemented by the framework using buffers, I/O ports and so on.

2.3 Comparable Approaches to our Design

Compaan & Laura Streaming applications are often written as a *sequential program* for example in C or MATLAB. They are quickly written and well understood but difficult to map to hardware devices or multi-chip systems. In the approach of the Leiden University [18] and [37] a two-step compilation flow will be executed. This flow transforms sequential programs to a parallel system which is easier to map on hardware.

The *Campaan* tool-chain is a compiler which transforms a sequential MATLAB code automatically to a parallel application model – to a Kahn Process Network as presented in chapter 3.2.

Laura transforms the platform independent KPN to a network of virtual processors. This network is mapped with a library of IP cores to a network of synthesizable processors written in a HDL (like VHDL, Verilog or SystemC).

A transformation of three applications (e.g. a 2D-DCT) to an FPGA implementation (without reconfiguration) was done in [37]. Both the coordination language and the function of the tasks are compiled into one piece of program. Further the program is mapped to a relative huge FPGA, a *Xilinx VIRTEX*^{TM} *II* -6000.

SCORE The *SCORE* [9] model developed by Caspi et al. provides both a specification model and a virtualized execution model for streaming applications. SCORE bases on a formally defined compute model.

Lee's Synchronous Dataflow (SDF) [20] has had a heavy influence to SCORE. Lee's SDFis restricted to static rates and static flow graphs, suitable for systems modelled completely at compile-time. Later work on the SDF-model such as Buck's Boolean-Controlled Dataflow [8], supports some dynamic programming constructs, but still with a compile-time focus. SCORE expands on these models to handle more dynamic characteristics such as a dynamic flow rates and graph evolution, as well as variable hardware resource availabilities.

An application in SCORE is defined as a graph of computation nodes that are connected by unbounded FIFOs. The function of the tasks is defined in a RTL language with a C-like notation.

So far there is no physical device which directly implements a SCORE application model, thus SCORE has been evaluated by simulation only.

The Embedded Machine The use of pre-compiled tasks and an interpreted coordination language has been presented for control-flow applications on a CPU based embedded machine by Henzinger et al. in [14]. For reconfigurable systems on FPGAs this is a new field of research.

Reconfigurable HW operating systems Reconfigurable hardware operating systems (OS) use reconfigurable devices as dynamic resources which are managed at runtime. These approaches introduce tasks or threads as basic units of computation – similar as a software operation system – and provides various communication and synchronisation mechanisms.

An user interaction or the runtime system itself places and schedules tasks on the reconfigurable device in so called task slots (multiple slots allowing a multitasking system). Such an operating system provides a «minimal» programming model although being less restrictive than coordination languages.

The Problems to be solved in such a OS are: (a) dynamic creation of multiple tasks in reconfigurable hardware, (b) run-time re-locatable tasks, (c) dynamic partial reconfiguration of individual task slots without affecting other running HW tasks, (d) broken interconnection networks in multi-tasking FPGAs and finally (e) needed reconfiguration time for task switches.

Functional prototypes that demonstrate multitasking on todays FPGA technologies were described in IMECs T-ReCS [21] [23] and XFBOARD of TIK, ETH Zürich [28] [27].

The mentioned related work has contributed to our goal of developing and implementing a dynamic reconfigurable run-time system. While one group uses a specific coordination language as a model of computation, but only in a static scenario without reconfiguration, another group is working towards a general operation system, which does not investigate a specific model for the optimised execution of streaming based applications.

Our proposed system investigates both the dynamic reconfigurability and a specific coordination language for streaming application.

3

Model of Computation

3.1 Dataflow Languages

Stephen Edwards gives in his book [11] an introduction to dataflow languages:

> Dataflow languages describe systems of procedural processes that run concurrently and communicate through queues. Although clumsy for general applications, dataflow languages are a perfect fit for signal-processing algorithms, which use vast quantities of arithmetic derived form linear system theory to decode, compress, of filter data streams that represent periodic systems of continuously-changing values such as sound or video.
>
> Dataflow semantics are natural for expressing the block diagrams typically used to describe signal-processing algorithms, and their regularity makes dataflow implementations very efficient because otherwise costly runtime scheduling decisions can be made at compile time, even in systems containing multiple sampling rates.

3.2 Kahn Process Networks - KPN

Gilles Kahn has developed in 1974 a simple language for parallel processing with a theoretical basis for dataflow computation.

A *system* in Kahns language is a set of *sequential processes* running concurrently and communicate through *FIFO queues* which are single sender and single receiver communication channels. A process that tries to read from an *empty queue* waits until data is available and cannot ask whether data is available before reading.

Kahn showed these restrictions make these systems *deterministic*. That is, the sequence of messages that pass through each queue does *not depend* on the *speed* of the processes or the *order* in which they execute.

Kahn Process Network are important for dataflow computation theory because of their deterministic concurrency, but their scheduling overhead makes them impractical. Because it can do nearly anything when it is running, a Kahn network demands a flexible compile-time analysis and permit schedulers with less overhead.

3.2.1 Determinism

All Kahn systems have the same result regardless of the decisions made by a scheduler because they are deterministic. This is because in KPN the interprocess communication is restricted to single direction FIFO queues which have a *blocking read* semantics.

The proof of determinism will be explained in the following: The sequence of data values which each process writes to its output ports is *only a function* of the sequence of data values arriving on its input ports. Mainly, the state of each process is only affected by the sequence of values it reads, and not on their arrival time.

As a thought a Kahn task executes in two phases:

1. In the first phase local variables are observed and changed and the values are written out to the output queues. During this phase the other processes don't influence the running process. As a consequence of this no shared variables exist, no other process can read or write local variables of the running task. The data output of a process is completely determined by the task state at the beginning of the phase and the state of the process at the end of the mentioned phase.
2. New data items are read in the second phase. The items are or aren't available when the process asserts a read operation, but in the Kahn model the tasks aren't allowed to communicate if they like to read new data or if the like to wait until a new data are available.

In the Kahn Process Network a FIFO – each FIFO – is exactly read and written by two processes.

3.2.2 Execution

There is always a correct schedule for the KPN. A challenge for the execution is to limit the needed memory consumption. Because non compile-time schedule can be done – the processes are allowed to communicate freely at runtime – a Kahn Process Network needs a dynamic scheduler. Generally, it is difficult to control the produced and received data tokens. There are two traditional approaches to dynamic dataflow scheduling. As a drawback they cause an unnecessary increase of data tokens.

A *data-driven scheduling* runs all processes that have enough data available. Unfortunately, this policy can produce tokens faster than they are consumed.

The data-driven system in figure 3-1 has the following problem: The processes in both loops run constantly under a data-driven policy, but nothing regulates the relative rates of the two loops. The output tokens are consumed by the «merge» node at

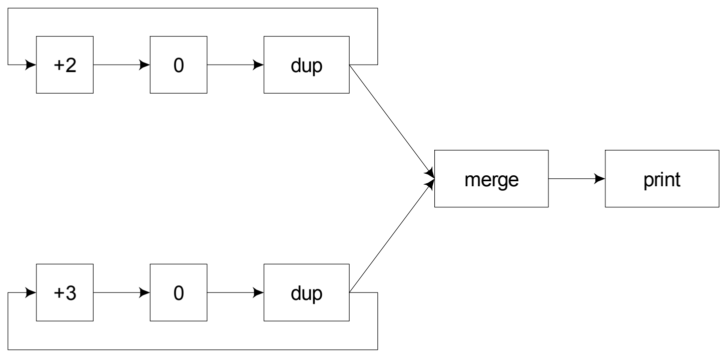


Figure 3-1
A system that fails under data-driven scheduling (after Parks [24, p.36]). The system generates an increasing sequence of integers which could be divided by two or three (i.e. 0, 2, 3, 4, 6, 8, 9, 10, ...). The series is build by merging two streams that are multiple of two and three. The merge operation is done at the «merge node» which orders to two sequences. Any duplicates are rejected.

different rates. Thus, tokens will accumulate indefinitely on one of the two queues leading into the merge node.

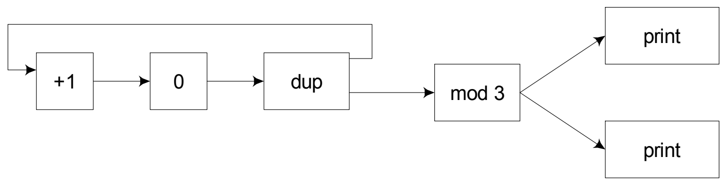


Figure 3-2
A system that fails under demand-driven scheduling (after Parks [24, p.42]) The system generates two sequences: One print statement shows all integers which could be divided by three, the other statement all other numbers. The loop on the left hand side generates an increasing sequence starting at zero. «mod 3» is the modulo operator.

Demand-driven scheduling takes an opposite approach but suffers from the same problems. A process waiting for a token on one of its inputs prompts the scheduler to run the system in figure 3-2. This latter system fails because both print statements demand tokens at the same rate, yet the «mod 3» block produces them at different rates, causing tokens to accumulate without bound on one of the queues leading to a printing process.

Tom Parks solved in his Ph.D. thesis [24] the bounded memory scheduling problem for a KPN by providing a scheduler that executes a system with bounded buffers if possible. He proposes to start the execution of the network with a small capacity size of each FIFO buffer. If during the execution phase the program has a deadlock because of a buffer overflow, the capacity of the smallest full buffer will be increased and the system continues. Often the dynamic scheduler contains a adaptable estimation function for the initial FIFO sizes.

3.2.3 The features of a Kahn Process Network

As a recapitulation we list the main features of a KPN:

- deterministic generation of data output
- token production is independent of the schedule
- blocking reads
- blocking writes (extension of Tom Parks)
- only one process has access to the FIFO (reading or writing)
- unidirectional FIFOs between processes
- no shared variables at all
- parallelism / communication
- the communication of a processes is internally sequential
  - process does exclusively a reading or writing operation (mutex)
- 3 states of processes:
  - the process can be *enabled* (also *running* or *activated* named)
  - blocked by a blocking read
  - blocked by a blocking write
- Kahn processes never terminate (endlessly execution)
- focus of KPN:
  - streaming application (i.e. audio, video)
  - dataflow oriented problems

3.3 Alternative Models of Computation

In this section we like to give an overview of alternative models of computation, more precisely we limit this overview to alternative dataflow languages.

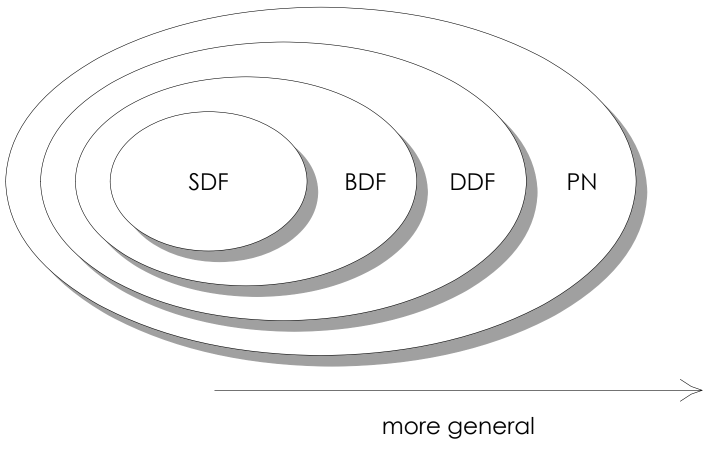


Figure 3-3
Overview of the different dataflow languages

3.3.1 Synchronous Dataflow - SDF

Synchronous Data Flow (SDF) is a dataflow language in which each process (called *actor*) *produces* and *consumes* a *fixed number of tokens per firing*. This behaviour makes an SDF communication patterns independent of the data values and allows to analyse the systems completely at compile-time. As a drawback this behaviour limits what the language can describe, SDF is capable describing most signal-processing algorithms, even those containing multiple sampling rates.

For practical reasons, SDF's compile-time scheduling and expressiveness makes this language a choice for many signal-processing algorithms and DSP implementations.

Like a Kahn process, an SDF actor has a fixed collection of input and output ports. Each port on an SDF actor is marked with the number of tokens it produces and consumes when the actor is fired.

Lee and Messerschmitt developed SDF at the University of California, Berkeley [20] starting in the late 1980s. For solving schedules with loops in the SDF Graph solution are shown in [4]. More information is also available at the webpage of the ptolemy project [26].

3.3.2 Binary Data Flow - BDF

Token flow model, that extends Synchronous Dataflow Languagegraphs by extending actors with token flow that is not known at compile time. Regular actors are simply a special case of a more general actor, which is called a *Boolean-controlled dataflow* (BDF) actor. The conditions for graphs consisting of such BDF actors to possess well-defined cycles, a bounded-length periodic schedule, and a schedule that requires bounded memory.

A regular dataflow actor has the property that the number of tokens produced on, or consumed from each arc is fixed and known at "compile time". Boolean-controlled dataflow (BDF) actors contain the regular dataflow actors as a subset, but in addition, the number of tokens produced or consumed on an arc is permitted to be a two-valued function of the value of a control token.

The behaviour of a conditional input for an actor is determined by a second input for the same actor; this second input always consumes exactly one token, the control token, on each execution. The behaviour of a conditional output for an actor may be determined either by an input (as for conditional inputs) or by an output.

Given this definition for actors, the Kahn condition [17] is satisfied, so that all data streams produced by the execution of BDF actors are determinate, regardless of the order in which the actors are executed

3.3.3 Dynamic Data Flow - DDF

The predictable control flow of SDF allows for efficient scheduling, but limits the range of applications. To support broader applications, the DDF domain uses dynamic (run-time) scheduling. For long runs, involving many iterations, this is more expensive than the static scheduling that is possible with SDF. But in exchange for this additional cost, we get a model of computation that is multi-functional as conventional programming languages. Supported are conditionals, data-dependent iteration, and true recursion.

Although the DDF is, in principle, a fully general programming environment, it is nonetheless better suited to some applications than others. Some signal processing applications with a limited amount of run-time control are a good match. Examples include systems with multiple modes of operation, such as modems (often implement multiple standards), signal coding algorithms (range of compression schemes), and asynchronous signal processing applications like sample-rate conversion.

3.4 Differences between KPN and SDF

As short comparison between the KPN and the SDF is done in the tabular 3-1.

KPN		SDF
processes	$\longleftrightarrow$	nodes
model contains explicitly internal states	$\longleftrightarrow$	sometimes a SDF is modelled without internal states
variable rates of tokens	$\longleftrightarrow$	fixed token rate at Input & Output
flexible model	$\longleftrightarrow$	limited model
branches in general	$\longleftrightarrow$	no branches at all
data-driven branches	$\longleftrightarrow$	no branches
dynamic scheduling	$\longleftrightarrow$	static scheduling scheduling is done at compile-time ideal scheduling: • *memory*, *runtime* and *switches* • *answer: exist a schedule at all?*
not definable: • if a deadlock occurs • memory requirements	$\longleftrightarrow$	predictable with analysis methods (from schedule algorithms)
not known internals of a task (not known how much data has to be handled)	$\longleftrightarrow$	"known" internals abstract view read and write known how often which amount of data

Table 3-1: Comparison between the two models of computation: KPN and SDF

3.5 Why we use KPN for FPGA designs

reconfigurable computation Our systems is a reconfigurable system – the functionality of the FPGA could be changed at runtime. To not loose the connectivity with other connected devices in the system (CPLD, FLASH, IPAQ and SRAM) we need an – always present – communication framework.

One possibility is to do a partial reconfiguration of the FPGA, another is to integrate the mentioned framework in each FPGA configuration.

The reconfiguration is triggered by a OS (Hardware OS, Software OS).

The FPGA could be integrated as a dynamical usable computation node.

- partial reconfiguration of FPGA
- OS
- dynamical useable

programming of embedded system Programming of an embedded system is complex. A good programming model and style reduce the time-to-market and the error rate of the implementation.

Such a model should offer the following features:

- rapid prototyping
- analysis of the embedded system
- compositional
- component IP based design
- «drag'n'drop» of already written or fabricated processes, easy takeover, only communication channels to define
- already established model

KPN on cpu already done The Kahn Process Network is already elaborated for software designs. The following items are the motivation to translate this concept to an FPGA design:

- execution of complex functions
- segmentation of tasks to run the set on much smaller execution unit
- parallel execution (true parallelism)
- partial reconfiguration

4

Architecture

4.1 Overview

Section 4.2 covers a summary over the system hardware. A first view on the Kahn Process Networkexecution framework – the «Slotmanager» is given in sec. 4.3. The tasks need to be loaded into the system by a loading process before the system execution starts. This loading procedure with the aid of a IPAQ is illustrated in section 4.4. The reconfiguration process 4.5 and a brief tool-flow for reconfiguration will close this chapter.

4.2 Hardware

4.2.1 BTNodeFPGA

At the Institute of Computer Engineering and Networks Laboratory (TIK) at ETH Zürich a small mobile sensornode has been developed - the BTNODE [7]. This node contains a microcontroller, a flash and a *Bluetooth*® module. The main application of the microcontoller is to handle the *Bluetooth* module and the flash memory with their operation modes and timings. Additionally, a part of the *Bluetooth* protocol stack and some service functions are implemented on the microcontroller.

For many applications it is a fact that the BTNODE lacks of computing power.

The goal of the diploma thesis of Peter Fercher [12] was to develop an FPGA extension board for the BTNODE – The BTNODEFPGA extension. It provides more computing power to the BTNODE for digital signal processing applications or streaming algorithms with the aid of an FPGA.

The BTNODEFPGA board contains the following elements (and their behaviour):

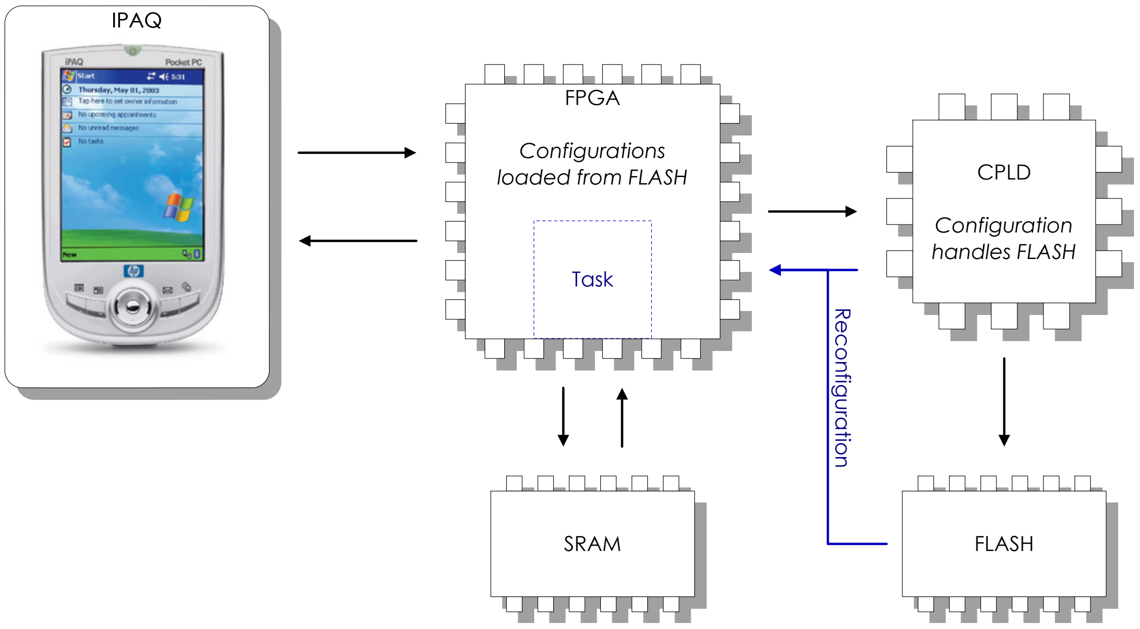


Figure 4-1
For the execution of a Kahn Process Network in reality we use an environment consisting of an IPAQ and an FPGA system:
The IPAQ contains the graph dependencies and the scheduling algorithms for the KPN. The tasks are executed either on the IPAQ (Software Task) or on the FPGA (Hardware Task). The communication between the IPAQ and the FPGA is done by a memory mapped region in the IPAQ using the FPGA as a «SRAM Extension». In the current design a task can only be replaced with a complete slotmanager environment containing the new task.
The reconfiguration bitstreams must be stored in FLASH memory via the FPGA and the CPLD. The reconfiguration process is initiated finally at the CPLD which transfers the new bitstream from FLASH to the FPGA. The SRAM is used for task communication (FIFOs) because this memory keeps the values during reconfiguration.

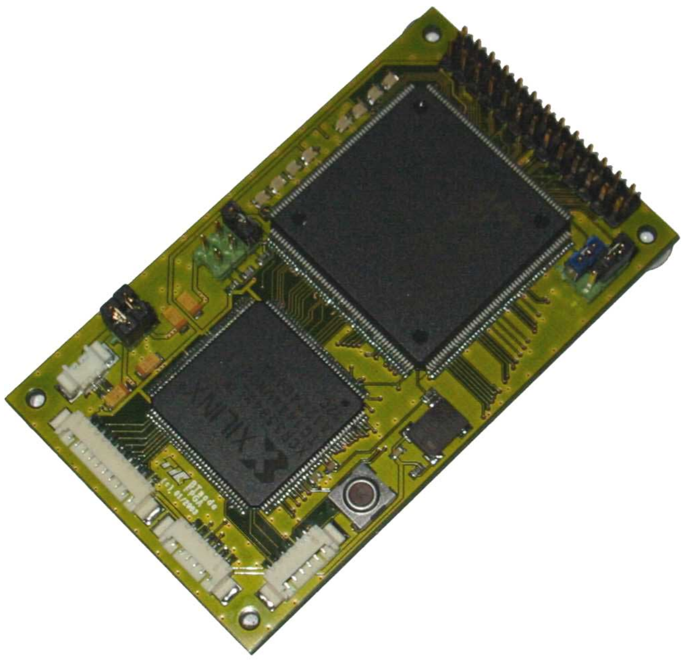


Figure 4-2
BTNODEFPGA*: Overview of the computation extension board for the* BTNODE*.*
On the top the Spartan II FPGA, on the bottom the Coolrunner CPLD. The additional memories - flash and SRAM - are non-visible at the backside of the PCB.

- **FPGA:** *Xilinx SPARTAN^{TM} II* XC2S200, 200k gates, Package PQ208, Speed Grade -5, the FPGA contains SRAM based structures, Documentation at [33]

  Usually, an FPGA consumes more power than an additional microcontroller but there are many advantages for sensor nodes:

  - computation power: current FPGAs reach nearly the computation power of ASICs for data-path oriented applications.
  - flexibility: in contrast to ASIC an FPGA is reconfigurable. By using partial reconfiguration it is possible to change only a part of the FPGA configuration.
  - multifunctionality: an FPGA can be used for a multitude of applications. There exist a lot of commercial or free IP-Cores which can be included into a custom design. Applications could be digital filters, en- or decoder, cryptography or even a complete CPU implementation.
  - parallelism : multiple algorithms or applications could be executed concurrently.

  Power estimations are done in the Diploma Thesis of Fercher [12].

- **CPLD:** *Xilinx Coolrunner^{TM} XPLA3* XCR3384XL, 384 Macrocells, Package TQ144, Speed Grade -12, Documentation at [32]

  A CPLD device saves his configuration in a non-volatile memory and doesn't need to be reconfigured at startup sequence.

- **FLASH:** *AMD 29LV801*, AM29LV801B-70EC, 8 MBit, 16 x 64 kByte sectors, 70 ns, 200 nA ultra-low-power mode, Documentation at [1]

  Flash memory is a non-volatile memory and stores in the design of the BTNODEFPGA the configuration bitstreams of an FPGA.

- **SRAM** *AMIC LP62S16256*, LP62S16256EV-55LLT, 4 MBit, 256k x 16 Bit data words, Documentation at [2]

  The external SRAM cell is a volatile memory. The advantage of this cell is that the values are kept during reconfiguration versus the values stored in BlockRAM memory on the FPGA.

4.2.2 IPAQ

- **IPAQ:** The used IPAQ is a model of the h3970 series from HP [15], containing an Intel XScale processor [16] running at 400 MHz. As an operation system we run Windows CE 4.2 .NET Edition [22].
- **IPAQ to BTNodeFPGA Bridge:** PCB developed by Matthias Dyer at TIK, ETH Zürich. The bridge allows a physical connection between the IPAQs expansion slot and the BTNODEFPGA board.

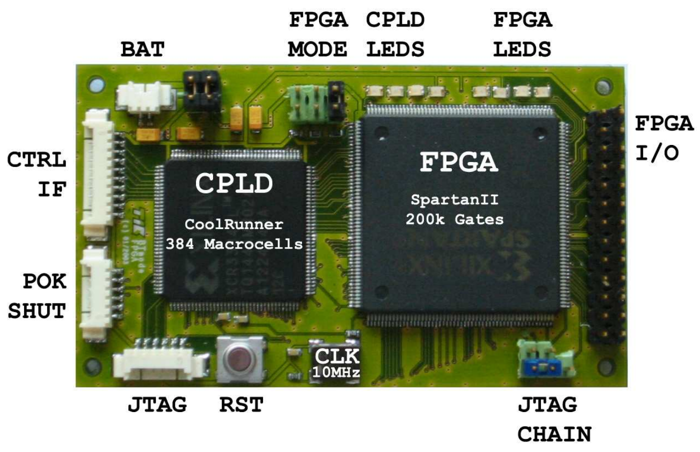


Figure 4-3
BTNODEFPGA*: electronic parts on the front side.*
On the left the Coolrunner CPLD, on the right the Spartan-II FPGA. On the backside of the PCB are the FLASH and the SRAM. The clock oscillator is replaced by a 18.432 MHz variant which allows much better serial communication.

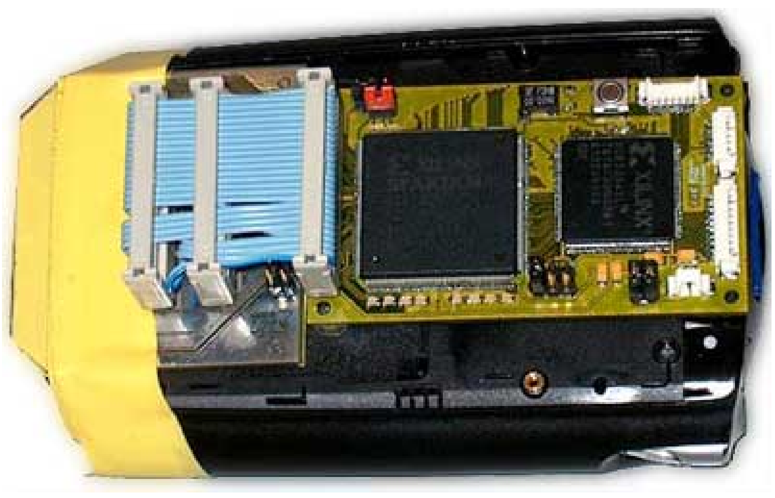


Figure 4-4
BTNODEFPGA *- Bridge: Connection to IPAQ*
The bridge and the cable connects the SRAM interface of the IPAQ expansion port to the IO pins of the BTNODEFPGA *board.*

4.2.3 Behaviour of the BTNODEFPGA *board*

In the figures 4-1, 4-3 and 4-5 the setup of the IPAQ, BTNODEFPGA and the connectors are shown. In this section we list the behaviour of the components, the advantages and some drawbacks for the KPNexecution framework.

- Power Supply (**BAT**): Power Input (2 pin molex). Used with a power supply adjusted to 6 Volts and a capacitor in parallel to have current peaks under control during reconfiguration.
- Clock (**CLK**): Two of the boards use a clock oscillator of 18.432 MHz (160 overclocking of a 115'200 baud serial clock), another two boards have a 10 MHz clock. A drawback in the design is that three of the four available clock pins of the FPGA can't be used without modifying (patching) the board.
- Reset (**RST**): Reset button connected to the CPLD. A reset button at the FPGA is missing so that in the most cases the reset signal from the button has to be routed through the CPLD to the FPGA by one wire of the «CPLD↔FPGA bus». The reset button is not debounced so a configurable pull-up resistor at the input pin of the CPLD is necessary.
- FPGA Mode Jumpers (**FPGA MODE**): The FPGA can be configured in different modes (parallel, serial, JTAG). The shown configurations in figure 4-5 are used in the design of the KPNexecution framework.
- JTAG Chain Jumpers (**JTAG CHAIN**): By setting the jumpers (fig. 4-5) the boundary scan includes the CPLD and the FPGA.
- CPLD and FPGA Leds (**CPLD LEDS, FPGA LEDS**): The meanings of the Leds and their assignment is described in the scheme.
- IPAQ Connection (**FPGA I/O**): To keep the IPAQ extension board bridge design easy the pin order has to be fixed with the FPGA pinout. Unfortunately the internal clock of the IPAQ is not available - using the SRAM interface modus it is possible to generate in the FPGA an inbound internal clock. The pin order of the IPAQ interface is on the right side of fig. 4-5.
- Serial Interface (**CTRL IF**): Control Interface usually connected to the BTNODEnow used as a serial interface (*RS-232*) to a host PC (10 pin molex). Only two data pins are used for the *RX* and the *TX* of the serial protocol, the others stay unconnected. A level shifter transforms the serial signal level from 12 to 3.3 Volts which is the level of the BTNODEFPGA board.

  To communicate with a PC a crossed RS232 cable (null modem cable) is necessary.
- A drawback in test and development cycle are the missing test points (pins) in the communication channel between FPGA and CPLD. As a solution the «UC/SERIAL»-interface could be used with a 10 pin molex connector.

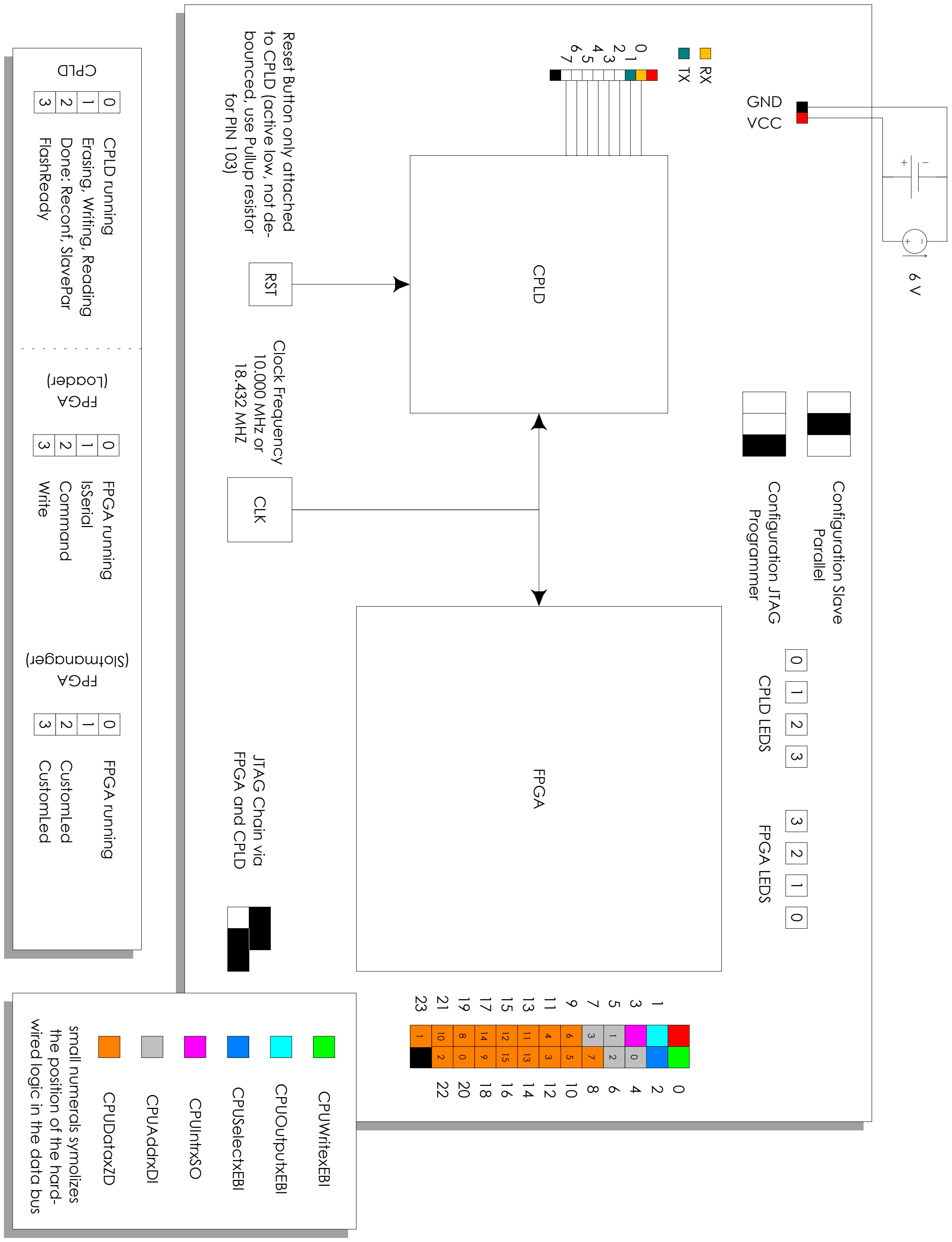


Figure 4-5
BTNodeFPGA: Additional Documentation and KPNExecution Framework Specifics. On the top the interface to RS232, on the bottom the interface to the IPAQ is shown. The table on the left side shows the meanings of the status leds.

4.3 Slotmanager

The Kahn Process Network execution framework on the FPGA is called «slotmanager» and contains multiple sub elements as shown in figure 4-6.

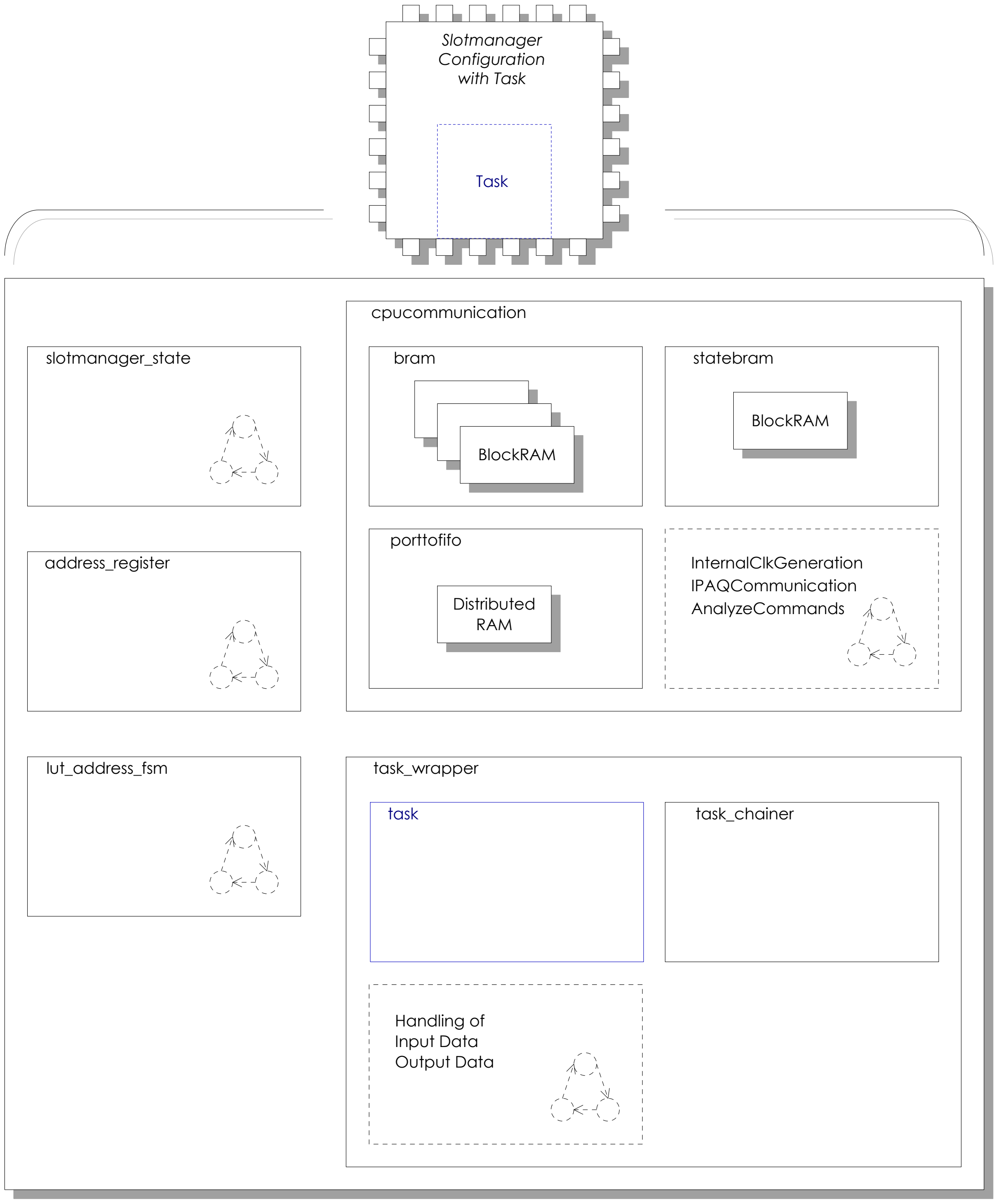


Figure 4-6
Slotmanager: Overview of the slotmanager design. The task is blue-coloured in both representations of the slotmanager. The boxes represent subparts of the manager, the connected circles one or several FSM. The names of the subparts item are the same as given for the VHDL entities.

slotmanager_state: Contains the main FSM of the Slotmanger and regulates the slot state: *Configuration*, *Run*, *Block*, *Load Task Internal States*, *Save Task Internal States*, *Task Switch*.

The *Configuration*-state initialises and reloads values for a register structure which manages the bounded FIFOs on the external SRAM. This structures are located in the **address_register** and **lut_address_fsm** sub parts.

cpucommunication: The cpucommunication part consists of the communication interface to the IPAQ and is able to fetch and analyse the command protocol. The functionality related to this communication protocol is also included in this part — PortID to FifoID “virtualising”, the configuration mechanism for the SRAM FIFO structures, additional BlockRAM for sharing data items between the FPGA and the IPAQ and BlockRAM for state saving.

lut_address_fsm: FSM for automatically addressing the SRAM look-up-table (LUT) for the used FIFOs in the configuration phase. The memory address will be generated with the PortID which is translated to a FifoID and finally to the external SRAM memory address.

address_register: Register which saves and computes t the information about the FIFO address and states in the external SRAM for each port.

The register file stores the information as follows.

- empty or full state
- next reading address in the FIFO
- next writing address in the FIFO
- basis address of the FIFO
- size of the FIFO needed for implementing the «wrap around» for FIFO structures.

task_wrapper: Wrapper which integrates the task in the framework. The wrapper includes some functional structures to save and restore internal task states too.

4.4 Upload of new bitstreams

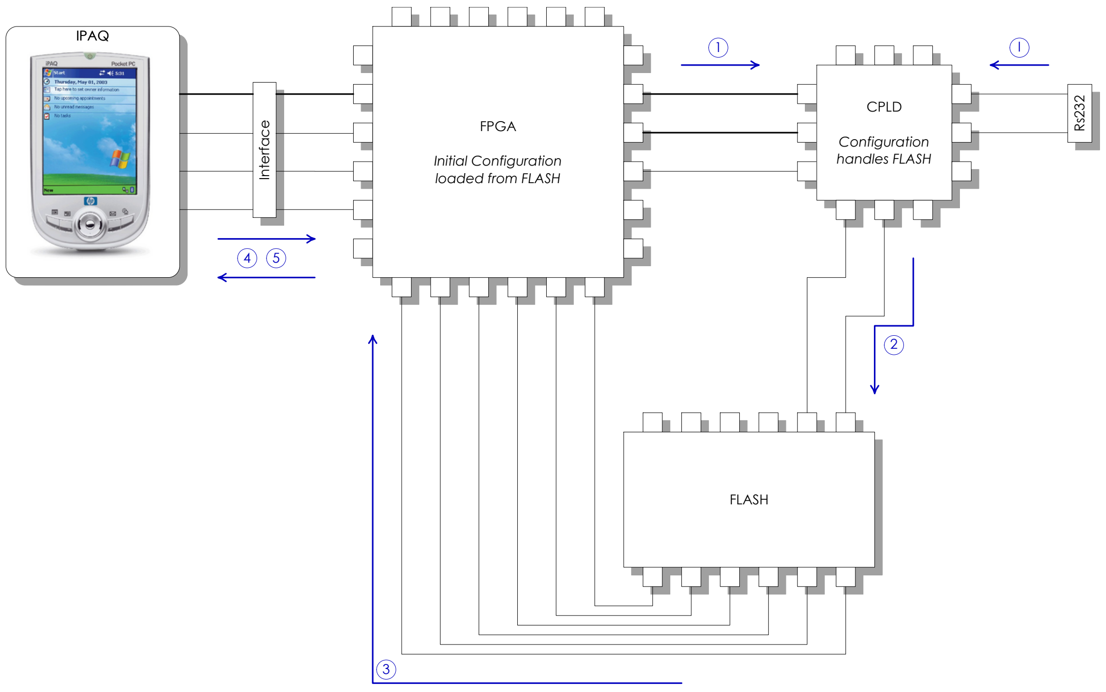

IPAQ
Interface
FPGA
Initial Configuration loaded from FLASH
CPLD
Configuration handles FLASH
Rs232
FLASH
1
I
2
3
4
5

Figure 4-7
Upload of new bitstreams - Configuration flow:
Before the slotmanager and the tasks start their execution, the tasks have to be downloaded to the BTNODEFPGA *board. An initial configuration – the* Loader *– is configured at startup time.*

1. At power-up sequence an initial bitstream is loaded from the FLASH memory to the FPGA. This configuration bitstream contains a *Loader* for the FPGA device which implements an upload communication protocol between the FPGA and the CPLD. The new FPGA configuration are streamed out by the IPAQ and feed-through to the CPLD.

I. As a alternative way to the previous method, a configuration bitstream can be transfered from a PC to the CPLD by sending the data with a *RS-232* protocol.

2. The CPLD includes a configuration which handles the FLASH program procedure. The bitstream – arriving from FPGA or RS-232 – are transfered to the memory device.

3. The IPAQ, the Loader, the Slotmanager or a PC via RS-232 can send a reconfiguration command to the CPLD. As a consequence of this instruction a reconfiguration with a new Slotmanager containing a new task will be executed.

4. Mapping control-flow graphs to hardware devices is very challenging. In our design we do the KPN graph interpretation in software on the IPAQ device. Thus, at the beginning of each new execution phase this information has to be exchanged.

5. The runtime information – blocking states – of a task has to be feed-back to the IPAQ device. Follow-up actions will happen.

4.5 Reconfiguration of the FPGA

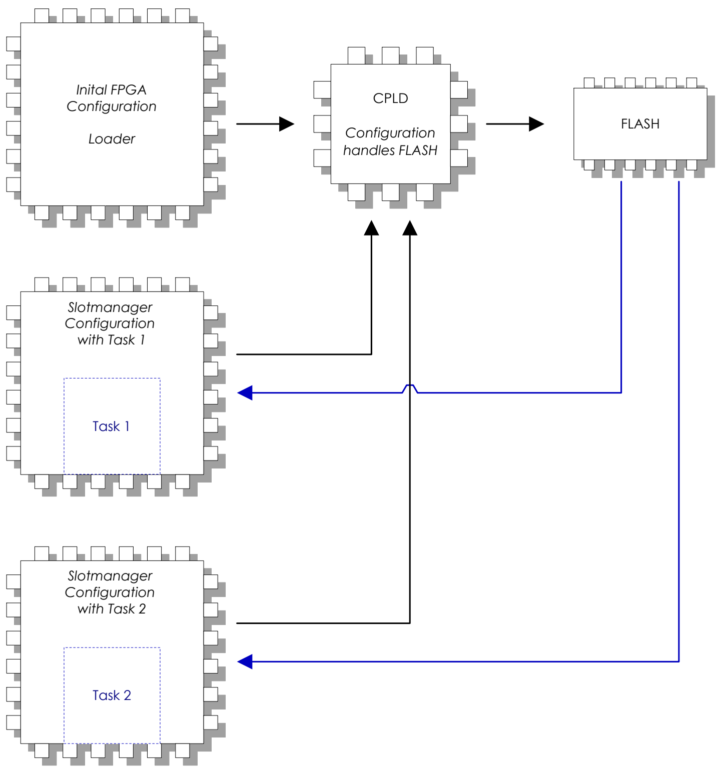

Inital FPGA Configuration
Loader
CPLD
Configuration handles FLASH
FLASH
Slotmanager Configuration with Task 1
Task 1
Slotmanager Configuration with Task 2
Task 2

Figure 4-8
Slotmanager: Reconfiguration of the FPGA
The Loader *or the* Slotmanager *could initiate a reconfiguration procedure at the CPLD. The reconfigure command allows an address relocation to sixteen predefined FLASH addresses – one out of five bitstreams is selected and transfered to the reconfiguration port of the FPGA.*

With the current implementation it is even possible, that the Slotmanager deactivates itself by initiating a reconfiguration of the FPGA. The system is self-reconfigurable.

4.5.1 Reconfiguration Toolflow

1. Generate Task: Use VHDL Task Template and write an user task for the KPN.
2. If necessary or wished use the insertion toolflow for the scan chain, a mechanism which allows to load and store internal states of Kahn tasks.
3. Connect the user task with the task wrapper of to the slotmanager. Keep caution on the reset polarity and the reset pin configuration (feed-through from CPLD in the current design).
4. activate “set unused pins to: float” in the ISE FPGA implementation flow – to don’t affecting the other devices at the FLASH address and data bus.
5. Synthesis, Translate, Map, Place & Route, Generate Bit File

6. Modify the Bit File with bistreambuilder.pl to obtain a programmable form.
7. Upload the modified bitstream to the IPAQ with the Microsoft Active Sync Tool. Default Directory is the folder «bitstreams».
8. Run the IPAQ Program «DataStreamWriter.exe» and select the new bitstream.

5 Detailed Implementation

5.1 Overview

In this section we explain the implementation details of our realisation of the «slot-manger» and Kahn Process Network execution framework:

At the beginning we introduce in section 5.2 several concepts which are fundamental and given by the restrictions of the model of computation or by the hardware device constraints. The «slotmanager» and its implementation in VHDL is described in section 5.3. A *Task* initially needs to be loaded into the Flash memory. The design of the «loader» includes an FPGA design and a CPLD design (sections 5.4 and 5.5). A mechanism to store and reload internal states of a *Task* needed for reconfiguration is figured in section 5.6. Handshaking between the devices and acquisition of asynchronous data is explained in section 5.7. Our Kahn Task Template is proposed in 5.8. The Windows CE DataStreamer program is explained in 5.9. Before a new Task and bitstream could be saved into the Flash and could be used for reconfiguration, it is necessary to run a tool-flow chain explained in section 5.10.

5.2 Concepts

5.2.1 Implementation of the FIFOs in SRAM

In the Kahn Process Network the communication between tasks is realised by using unbounded circular buffers — named FIFOs in the following sections.

In our execution framework unbounded FIFOs aren't feasible so we implemented bounded FIFOs for the «slotmanager». Bounded FIFOs have only a limited memory space with a «wrap-around» mechanism at the upper memory space boundary. The FIFO items are stored in the external SRAM device so that the values will be saved during the FPGA is reconfigured. To account for the complete data loss during configuration we use the lowest part of the SRAM as a Look-Up-Table to store the FIFO parameters (size, read and write addresses) as shown in the figure 5-1. The LUT entries are loaded during the task initialisation phase from the lower SRAM address into a register file in the slotmanager. The LUT values are modified during task execution and stored back when a task switch occurs.

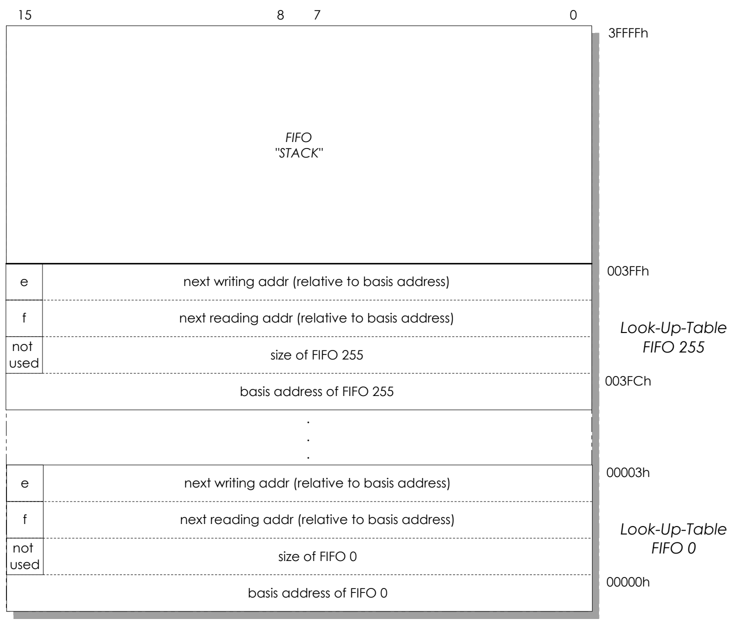

15
8
7
0
3FFFFh
FIFO
"STACK"
003FFh
e
next writing addr (relative to basis address)
f
next reading addr (relative to basis address)
not used
size of FIFO 255
Look-Up-Table
FIFO 255
003FCh
basis address of FIFO 255
00003h
e
next writing addr (relative to basis address)
f
next reading addr (relative to basis address)
not used
size of FIFO 0
Look-Up-Table
FIFO 0
00000h
basis address of FIFO 0

Figure 5-1
SRAM:
Implementation of the **FIFO***: The lower 1024 words (256 x 4 x 16 bits) of the SRAM contain a Look-Up-Table for the FIFO configurations. The configuration consists of the base address (shifted by 2 bits, so that only at each fourth address a new FIFO could be started), the size of the FIFO (maximum are 15 bits, max. 1 MBit), the last reading and writing address relative to the base address . All addresses need a calculation, because the address space of SRAM is 18 bits but a SRAM data word is only able to hold 16 bits. The size, reading and writing FIFO address are relative addresses.*

5.2.2 FLASH Configuration

A bitstream for the *Xilinx SPARTAN^{TM} II* FPGA has a size of 165 kBytes — generated by the ISE toolflow with the default configuration. To save this bitstream in the Flash memory we need at least 3 x 64 kByte slots. Using the Flash in 64 kBytes [1, p.10] sectors has several advantages:

Each sectors could be erased or reprogrammed without affecting the others and there is a possibility to store and configure several configurations. The first three sectors should contain the default configuration of the FPGA at startup sequence which implements the *Loader*.

If all 16 sectors of the Flash are used 5 bitstreams could be stored exactly. The proposed storing scheme is listed in table 5-1.

5.2.3 Internal Clock Generation

The IPAQ extension port has no clock interface or clock pin for the internal IPAQ clock (`MEMCLK`) at all. For in and out data transfer we connect to the programmable

bitstream select	used sectors for bitstream storage		
0	**0**	**1**	**2**
1	1	2	3
2	2	3	4
3	**3**	**4**	**5**
4	4	5	6
5	5	6	7
6	**6**	**7**	**8**
7	7	8	9

bitstream select	used sectors for bitstream storage		
8	8	9	10
9	**9**	**10**	**11**
A	10	11	12
B	11	12	13
C	**12**	**13**	**14**
D	13	14	15
E	14	15	0
F	15	0	1

*Table 5-1: Bitstream storing scheme for Flash memory. Our recommendation is to store the initial (*Loader*) bitstream at address **0**, the first custom task in a bitstream at **3**, … and the fourth task bitstream starts at the **C** sector.*

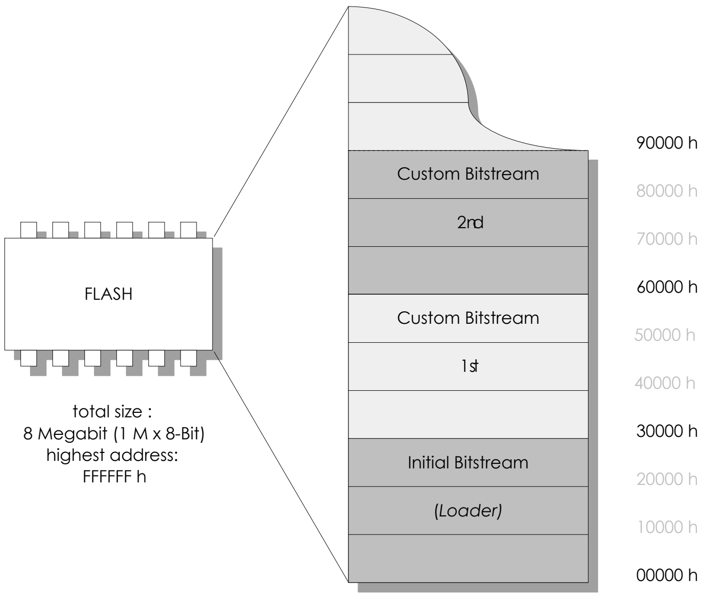
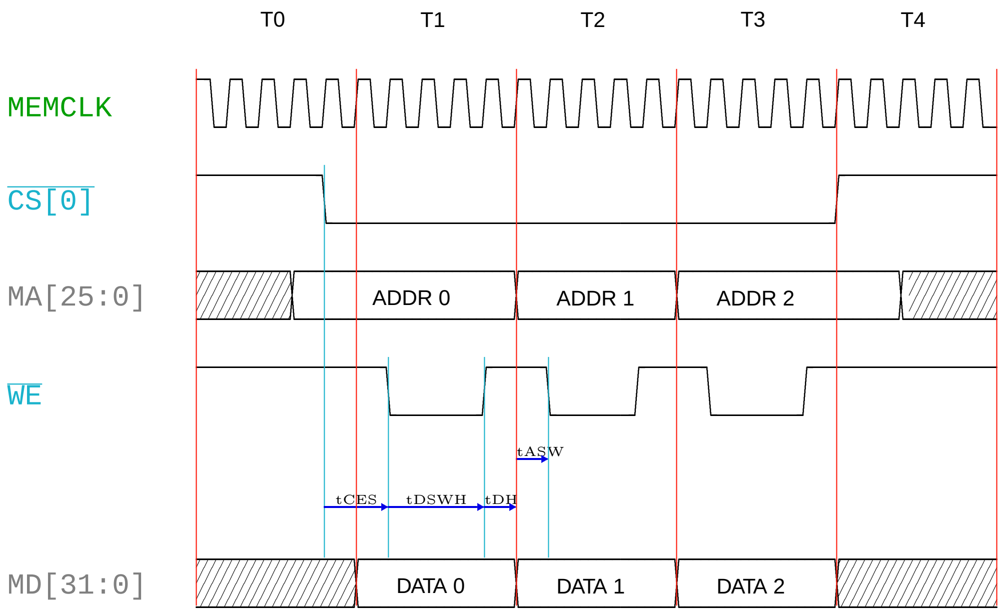


Figure 5-2
Flash: Storing scheme for FPGA configuration bitstreams. The memory could store up to 5 bitstreams. The sector (F) with the highest addresses (F0000h) is empty at all.

asynchronous SRAM interface modus of the XScale — included in the memory manager — and use a direct mapped memory region with our «expansion slot bridge». The behaviour of the SRAM interface is as expected: data addresses will be valid at the falling clock edge, data words at the rising clock edge. In figure 5-3 the timing constraints are shown — the minimal timings are listed in IPAQ clock cycles.



Required clock cycles between transitions:

tCES:	`2 MEMCLKs`	*tDSWH*:	`4 MEMCLKs`
tDH:	`1 MEMCLK`	*tASW*:	`1 MEMCLK`

Figure 5-3
Waveforms of the IPAQ extension interface in SRAM mode.

We have measured several output waves of the IPAQ expansion port interface with a logic analyser (measurement results in fig. 5-4). We recognise that the ChipSelectEnable (nCS) wave is 128 ns low in reality. The falling edge of the WriteEnable (nWE) signal triggers 10 ns later.

The IPAQ is master of the SRAM interface. To determine a *write* or *read* command we have to sample the waveform with the BTNODEFPGA board clock. For a write operation, the data are valid 4 `MEMCLK` before ChipSelectEnable (nCS) rises. Even, if we use double-edged triggered flip-flops at 18.432 MHz we will certainly miss a data word.

$$\frac{400\ MHz}{2\ *\ 18.432\ MHz} \cong 11 \geq 4(!)$$

Only one clock pin of the *Xilinx SPARTAN*^{TM} *II* on the BTNODEFPGA board is connected to a clock source (the oscillator) but non of the other specialised clock pins are connected to the I/O pins of the board — these pins have low slews, low delays and are optimised to connect to the 4 internal clock nets.

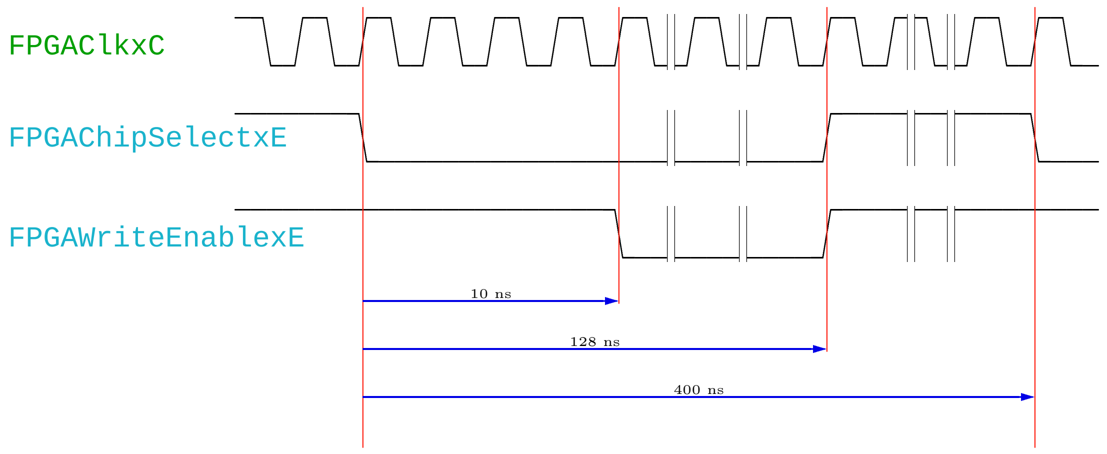


Figure 5-4
IPAQ extension port waveforms, the FPGA clock and the timing data relative to this clock on the BTNODEFPGA *board.*

As a solution of this dilemma we «generate» an internal clock signal composed of the three input enable signals: nCS (named `CPUCSelectxE` in the design), nWE (`CPUWritexE`) and nOE (`CPUOutputxE`). We use three normal I/O pins as «clock input» and build with the aid of some logic and a dedicated clock buffer an internal clock (fig. 5-5).

Valid data words on IPAQs tristate data bus are loaded to flip-flops triggered by this generated clock signal. The address is fetched at the falling edge, the data at the rising edge — as expected from a SRAM device.

The information if the clock pulse originates from a «write» or «read» command will be stored in an additional 1-bit register.

The fetching process of a sample measurement is shown in figure 5-6.

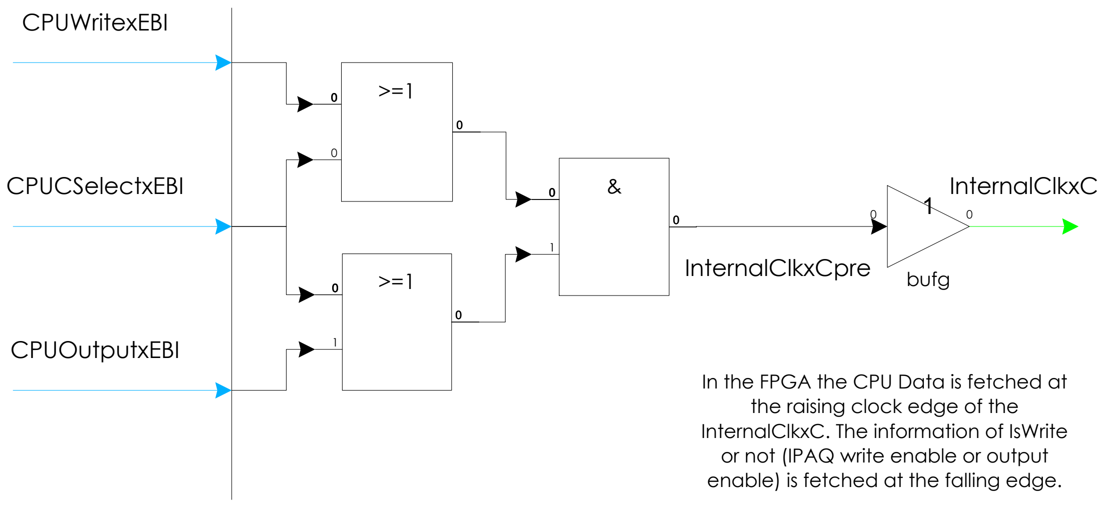

CPUWritexEBI
CPUCSelectxEBI
CPUOutputxEBI
>=1
>=1
&
InternalClkxCpre
InternalClkxC
bufg
In the FPGA the CPU Data is fetched at the raising clock edge of the InternalClkxC. The information of IsWrite or not (IPAQ write enable or output enable) is fetched at the falling edge.

Figure 5-5
Internal Clock Generation in the FPGA: `CPUCSelectxE`, `CPUWritexE` and `ChipSelectE` are enable signal from the IPAQ and connected via the «bridge» to the FPGA IO Pins (at the bottom of figure 4-5). The internal clock is generated by the following logic function:

$$(\texttt{CPUWritexEBI} + \texttt{CPUCSelectxEBI}) \bullet (\texttt{CPUOutputxEBI} + \texttt{CPUCSelectxEBI})$$

Finally, the generated clock is asserted to a dedicated clock net (one of four availables).

5.3 Slotmanager

The «slotmanager» is the execution framework for the KPN tasks. The frameworks includes mechanisms to load and execute different tasks, the managing of the external devices (CPLD, Flash, SRAM) and some structures which enables the reconfiguration process of the task slot.

As seen in figure 5-7 the slotmanager is divided in multiple parts:

***slotmanager_state* sec. 5.3.1** includes the main FSM of the slotmanager

***lut_address_fsm* sec. 5.3.2** an managing unit for the LUT stored in SRAM

***address_register* sec. 5.3.3** a FIFO implementation for bounded memory

***cpucommunication* sec. 5.3.4** the communication with the IPAQ and finally the

***task_wrapper* sec. 5.3.5** the task itself.

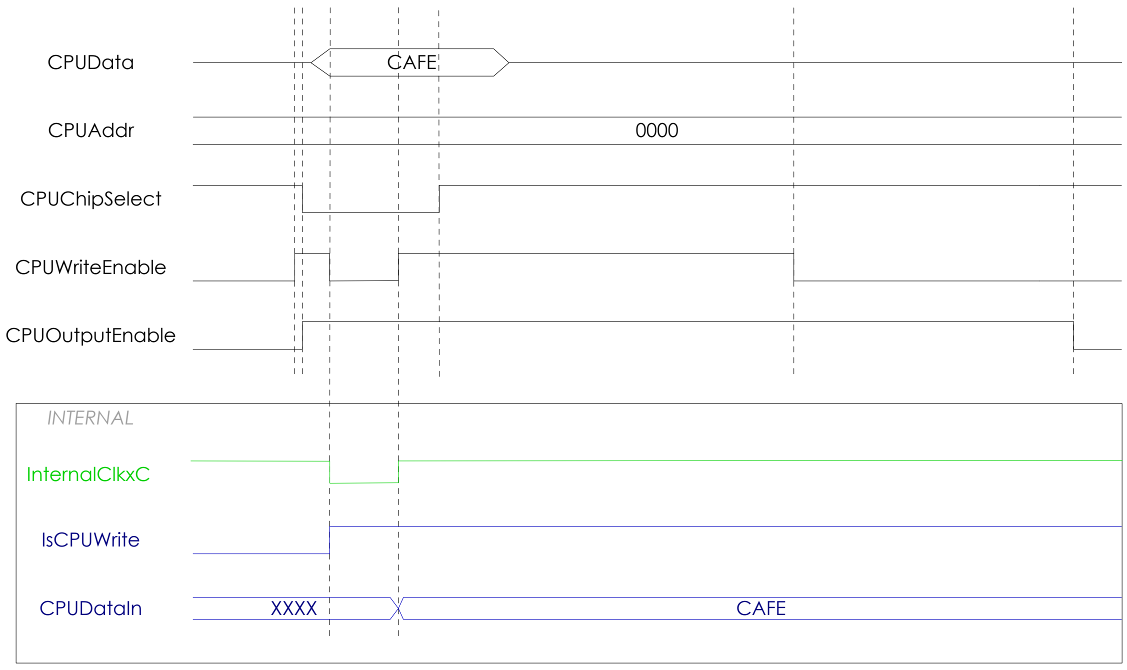


Figure 5-6
FPGA - Waveforms of fetching an IPAQ data word
`InternalClk` *is generated by the formula of figure 5-5. In reality, we measure that both signal WriteEnable and OutputEnable are changing their levels several times at a* write *or* read *command. By analysing also the level of the ChipSelect we can make a distinction between these different transitions. In the sample wave the InternalClk signal is triggered by a* write *command. This information is kept in the variable IsWrite at the falling edge. The new data word is latched at the rising edge.*

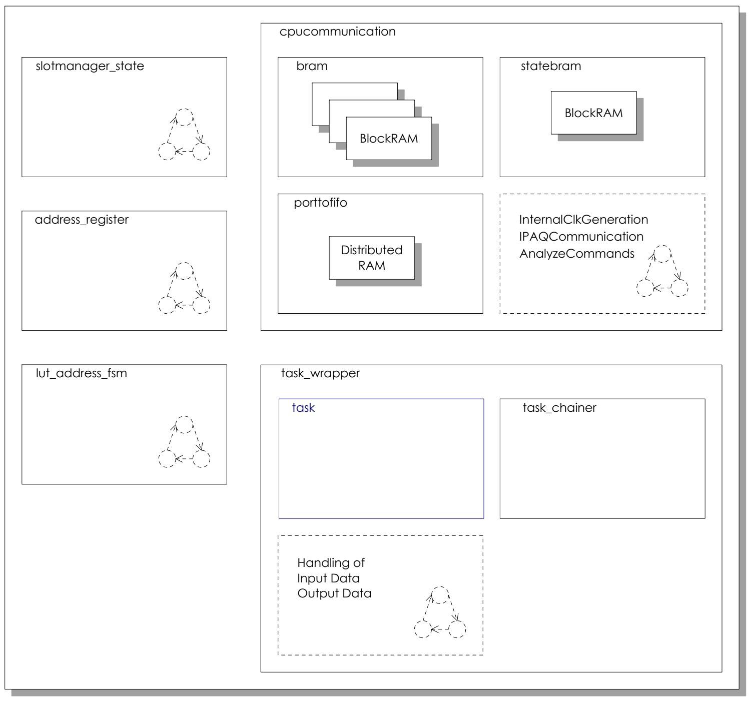


Figure 5-7
Slotmanager - Overview of Slotmanager, the sub parts and their implementation are described in the following sections.

5.3.1 slotmanager_state

The part slotmanager_state includes the main and the configuration Final State Machine of the «slotmanager» — brief it is the heart of the running system.

The main FSM contains the slotmanager state and is described in figure 5-8, the configuration FSM is needed after reconfiguration and shown in figure 5-9.

VHDL ENTITY FOR SLOTMANAGER_STATE

```
entity slotmanager_state is
  port (
    ClkxCI                     : in  std_logic;                  -- clock
    RstxRBI                    : in  std_logic;                  -- reset (active low)
    ---------------------------------------------
    -- SLOT STATE
    ---------------------------------------------
    SlotManagerStatexSO        : out SlotManagerState;           -- state of the main FSM
    ConfigSlotManagerStatexSO  : out ConfigSlotManagerState;     -- state of the config FSM
    LutAddressFSMxSI           : in  LUTAddressFSMState;         -- addresses for SRAM LUT
    BLOCKINGxSI                : in  std_logic_vector((NUMBER_OF_PORTS_PER_PROCESS*2)-1 downto 0);
    PortIDxDO                  : out PortIDType;                 -- generated PortIDs for "fillup"
    ---------------------------------------------
    -- READ / WRITE and ACKS
    ---------------------------------------------
    PortReadxSI                : in  std_logic;                  -- read action (ReadToken)
    PortWritexSI               : in  std_logic;                  -- write action (WriteToken)
    PortReadWriteAckxEI        : in  std_logic;                  -- acknowledge of Token action
    ---------------------------------------------
    -- TRIGGER SIGNALS
    ---------------------------------------------
    -- start all
    RunNowxEI                  : in   std_logic;                 -- start main FSM
    -- cpu to fpga communication done / new
    CommFPGAConfigDonexEI      : in   std_logic;                 -- communication done
    CommFPGANewConfigxEI       : in   std_logic;                 -- new bitstream (!) asserted
    -- generated output trigger signals
    GetAddrNowxEO              : out  std_logic;                 -- enable: LUT addresses
    FetchCPUDataxEO            : out  std_logic;                 -- enable: comm. to CPU
    -- shift state
    ContextSwitchxSO           : out  std_logic;                 -- context switch enable
    ContextSwitchDonexSI       : in   std_logic                  -- context switch done
  );
end slotmanager_state;
```

Listing 5-1
Slotmanager State Entity

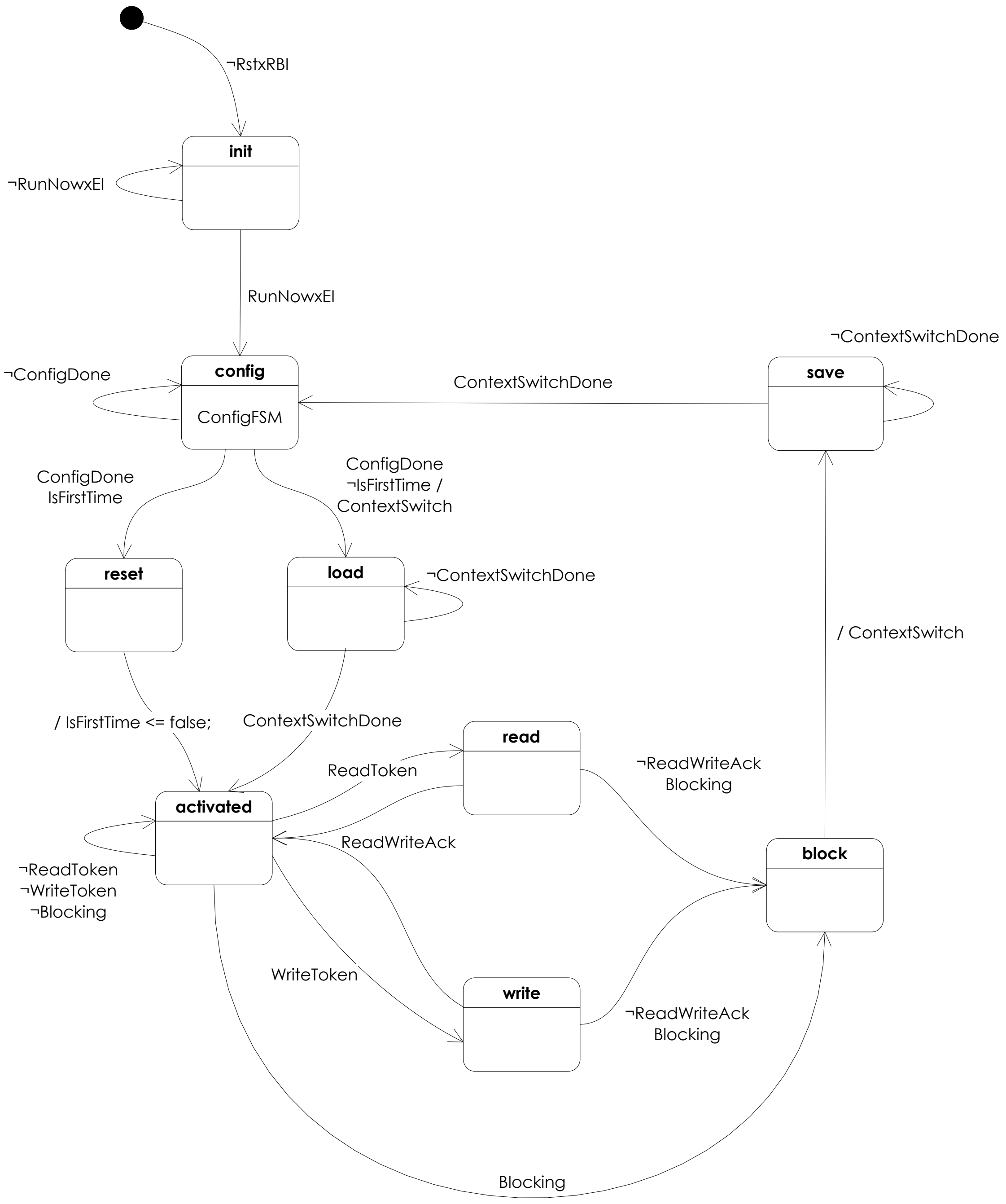


Figure 5-8
Slotmanager - FSM:
This FSM is the main final state machine of the «slotmanager». The FSM stays in the init-state till a explicit RunNow *is asserted from the CPU Communication Interface. In the config-state the configuration FSM (fig. 5-9) will be started and the main FSM waits till this process has finalised. A check will be done if the active task of the slotmanager has completed a cycle (config → load → activated → block → save → config) without being replaced. If this condition is true the task will be loaded a first time and the internal states will be reseted in the reset-state, otherwise the internal states will be loaded from state saving memory in the load-state. When the slotmanager is in the activated-state the task is running and produces* read *or* write *operations which will typically be acknowledged. Otherwise the task is in block-state and can be replaced by another task. To store the internal states of the task, the internal states are «shifted out» in the save-state. Finally the config-state is reached. Now another task could be loaded if available else the task is blocked endlessly.*

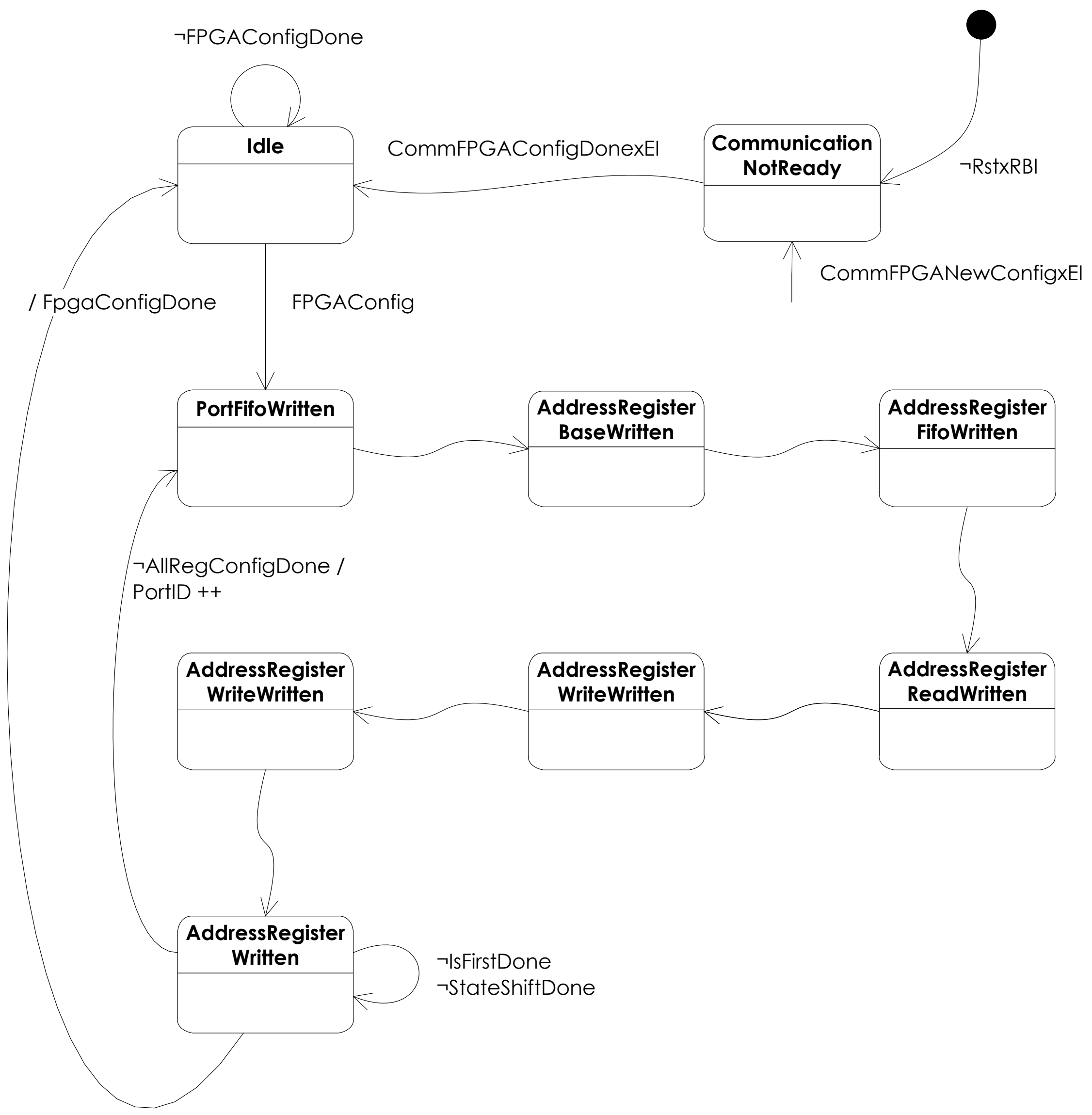


Figure 5-9
Slotmanager - Configuration FSM:
The configuration FSM stays in CommunicationNotReady-state until the CommFPGAConfigDone *arrives from the CPU Communication Interface. This signal indicates that a new configuration has been loaded (reconfiguration done) and the communication between the IPAQ and the FPGA — graph dependencies in the KPNnetwork — has been done. The* FPGAConfig *signal is triggered by the main FSM (fig. 5-8) and always asserted after the* CommFPGAConfigDone *chronologically. The* FPGAConfig *causes the FSM to switch from idle-state to a cycle with several states:*
If the translation map (PortID → FifoID) is set up in the PortToFifoWritten-state, in the following five states the register field will be automatically filled for each PortID with their values as described in section 5.3.3. The addresses are generated in the address_lut_fsm and triggered by the following five states: AddressRegisterBaseWritten, AddressRegisterFifoWritten, AddressRegisterReadWritten, AddressRegisterWriteWritten and AddressRegisterWritten. The cycle will end if for all registers with their corresponding PortIDs are initalised.

5.3.2 lut_address_fsm

The purpose of this part is to generate the LUT addresses for the lowest part of the SRAM from asserted FifoIDs and an internal FSM. The FSM loops in a cycle for the following states: Idle → BaseAddr → FifoHigh → ReadAddr → WriteAddr → Idle. The first FSM transition needs a trigger by the `GetAddrNowxEI` signal.

VHDL ENTITY FOR LUT_ADDRESS_FSM

```
 entity lut_address_fsm is
   port (
     ClkxCI            : in   std_logic;                          -- clock
     RstxRBI           : in   std_logic;                          -- reset (active low)
     -- trigger signal
     GetAddrNowxEI     : in   std_logic;                          -- enable the address FSM
     -- FIFO IN -> ADDRESS OUT
     FifoIDxDI         : in   std_logic_vector(7 downto 0);       -- selects the FIFO
     LutAddressAddrxDO: out  std_logic_vector(17 downto 0);      -- address of SRAM
     -- LUT ADDRESS FSM STATE OUT
     LutAddressFSMxSO  : out  LUTAddressFSMState                  -- address FSM state
     );
end lut_address_fsm;
```

Listing 5-2
Entity for the LUT Address FSM

Depending on the FSM state the following SRAM Addresses are generated (i.e. if FifoIDxDI is 200 (=C8h), in the four following cycles the SRAM addresses 801, 802, 803 and 804 will be generated):

```
   LUTBaseAddrRegxD  <= FifoIDxDI & "00";
   LUTFifoHighRegxD  <= FifoIDxDI & "01";
   LUTReadAddrRegxD  <= FifoIDxDI & "10";
   LUTWriteAddrRegxD <= FifoIDxDI & "11";
```

Listing 5-3
Generation of LUT Address for the FIFOs

5.3.3 address_register

In short, the part address_register of the «slotmanager» is a large register field. This registers contain all FIFO managing data, which are loaded in an initalise process from the lowest part of the SRAM — the Look-Up-Table (LUT) as proposed in fig. 5-1. As a recapitulation the FIFO in SRAM is arranged in the following way:

IV data word:
1-bit flag for «empty FIFO»
15-bit next writing address

III data word:
1-bit flag for «full FIFO»
15-bit next reading address

II data word:
1-bit flag not used
15-bit size of FIFO

I data word:
16-bit base address

Primarily, the base address of the FIFO is loaded. Because data words in SRAM are 16-bit wide but the addresses of the memory cell are 18-bit, we shift the loaded binary address by 2. With this setup only, each fourth address could be the basic address of a new FIFO (details in figure 5-10 on page 44).

The FIFO size, the next reading or writing address are stored in relative values to the base address in the LUT. When these words are loaded to the *address_register*, they are converted to absolute values according to the base address.

A FIFO could have a maximal size of 2^{15} — with the maximal sized FIFOs, the external SRAM cell has only a space for 8 FIFO allocations.

The reading address contains the next read address in the FIFO, the writing address the next write address respectively.

If the highest FIFO address is reached by a reading or writing operation, a «wrap-around» of the address will happen: A comparator of the reading (comparator `WrapAroundReadxS`) or writing (`WrapAroundWritexS`) address with the highest FIFO address signals a flag when the highest address occurs. As a consequence the address counter is reseted to the lower boundary value of the FIFO.

The number a parallel register fields is proportional to the number of available and permitted *ports-per-task*.

The constant `NUMBER_OF_PORTS_PER_PROCESS` implies this value. In the current and tested implementation the available ports is set to 8. The internal register are build by a VHDL generate statement, so that the allowed ports-per-task could be changed easily if wished or necessary (caution: augmenting this value will massively increase the area overhead of the slotmanager).

At runtime the PortID acts as a switch in the register field and selects the accurate register entries and values.

A schematic of the «Address Generation Unit» is given in figure 5-10. A VHDL code snipplets of the unit is given in the listing 5-4.

VHDL ENTITY FOR ADDRESS_REGISTER

```
entity address_register is
  generic (
    NUMBER_OF_PORTS_PER_PROCESS : natural := 8
    );
  port (
    ClkxCI                : in  std_logic;                -- clock
    RstxRBI               : in  std_logic;                 -- reset (active low)
    -- data transfer
    DataInxD              : in  std_logic_vector(15 downto 0);  -- SRAM DATA BUS
    DataOutxD             : out std_logic_vector(15 downto 0);  -- SRAM DATA BUS
    AddressOutxD          : out std_logic_vector(17 downto 0);  -- SRAM ADDR BUS
    -- port id switch
    PortIDxDI             : in  PortIDType;               -- PortID switch
    -- FSM
    SlotManagerStatexSI   : in  SlotManagerState;         -- Load, Store, ReadToken, WriteToken
    LutAddressFSMxSI      : in  LUTAddressFSMState;       --    BaseAddr,   FifoHigh,    ReadAddr,
WriteAddr

    -- Empty Full Flags
    BLOCKINGxSO           : out std_logic_vector(NUMBER_OF_PORTS_PER_PROCESS*2)-1 downto 0);
                                                                    -- one of all PORTs is in
blocking state
    EmptyFullFIFOxSO      : out std_logic                 -- current PORT is in blocking state
  );
end address_register;

architecture rtl of address_register is
  type BaseAddrRegType is array                                     -- base addresses
          (NUMBER_OF_PORTS_PER_PROCESS-1 downto 0) of std_logic_vector(17 downto 0);
  type FIFOHighRegType is array                                     -- FIFO upper boundaries
          (NUMBER_OF_PORTS_PER_PROCESS-1 downto 0) of std_logic_vector(17 downto 0);
  type ReadAddrRegType is array                                     -- next reading addresses
          (NUMBER_OF_PORTS_PER_PROCESS-1 downto 0) of std_logic_vector(17 downto 0);
  type WriteAddrRegType is array                                    -- next writing addresses
          (NUMBER_OF_PORTS_PER_PROCESS-1 downto 0) of std_logic_vector(17 downto 0);
  type FullOrEmptyRegType is array                                   -- FIFO full or empty infor-
mation
          (NUMBER_OF_PORTS_PER_PROCESS-1 downto 0) of std_logic_vector(1 downto 0);

  type StdLogicArrayType is array                                    -- generated comarison ar-
ray
          (NUMBER_OF_PORTS_PER_PROCESS-1 downto 0) of std_logic;

  ...
```

Listing 5-4
Entity – Address Register

In the Kahn Process Network the FIFO (or any manager) must not know about the fill level of the unit. In the adapted model of Parks a FIFO still doesn't know the fill level, but a task will be in blocking read or blocking write state, if one of the FIFOs run at full capacity or a underflow occurs. In our design we have implemented such an behaviour of the task with a Final State Machine as shown in figure 5-11.

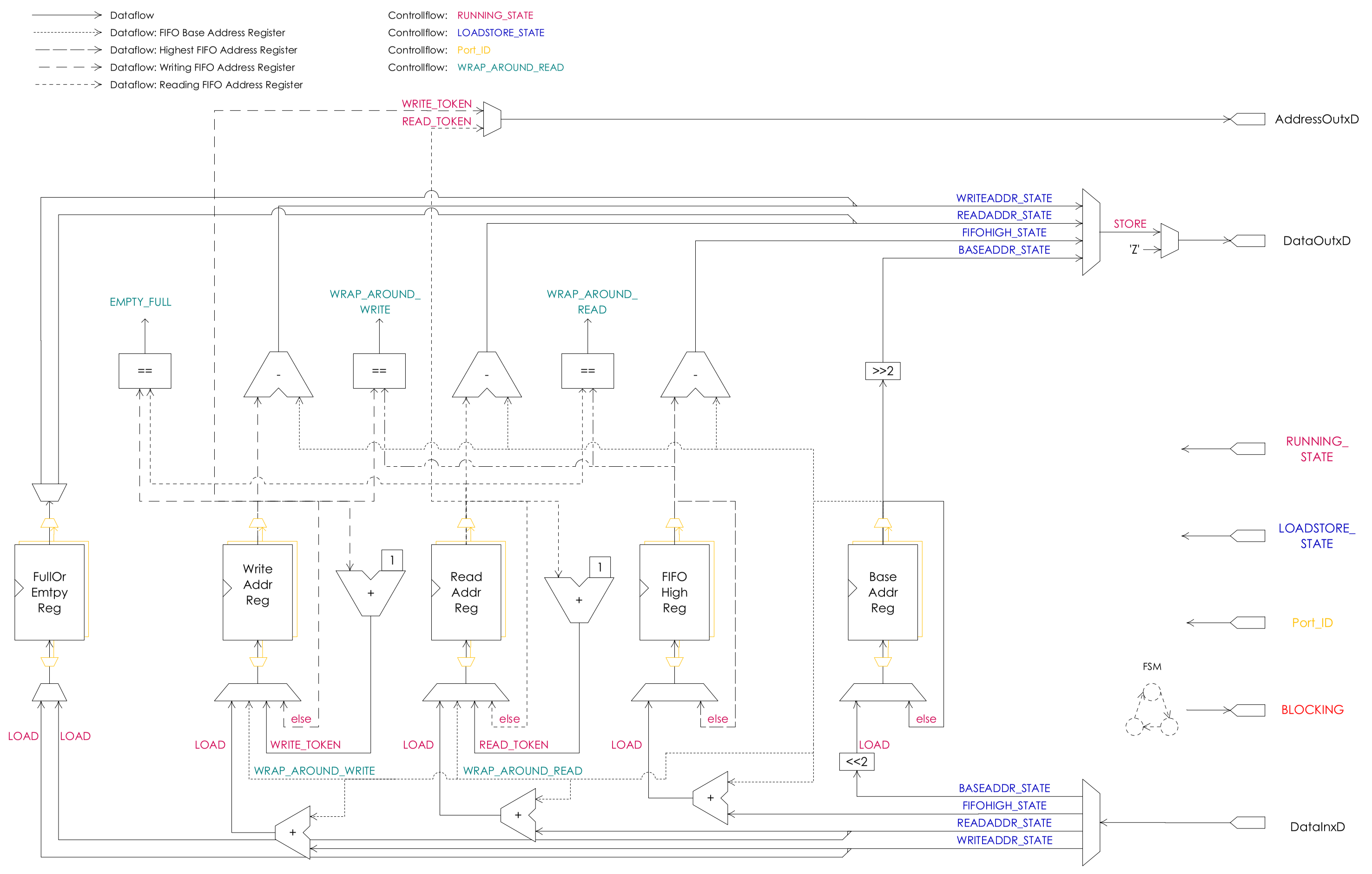


Figure 5-10
Address Generation Unit:
The different operation modes are coloured and explained in legend above the schematic.
*In brief, a description of the data flow: In a first step the **BaseAddr** of the FIFO is loaded into the Base Addr Reg. The address is shifted by 2 as already mentioned. In a second step the **FifoHigh** value is loaded. This value is added together with the base address and stored in the FIFO High Reg. The **ReadAddr** and **WriteAddr** are modified in the same way: the absolute value is added together with the relative counters. The FullOrEmptyReg is loaded by 2 bits (Empty Flag, Full Flag) which are stored physically in front of the read and write address entries. The FullOrEmptyReg is modified by the EMPTY_FULL flag and a FSM as figured in fig. 5-11.*

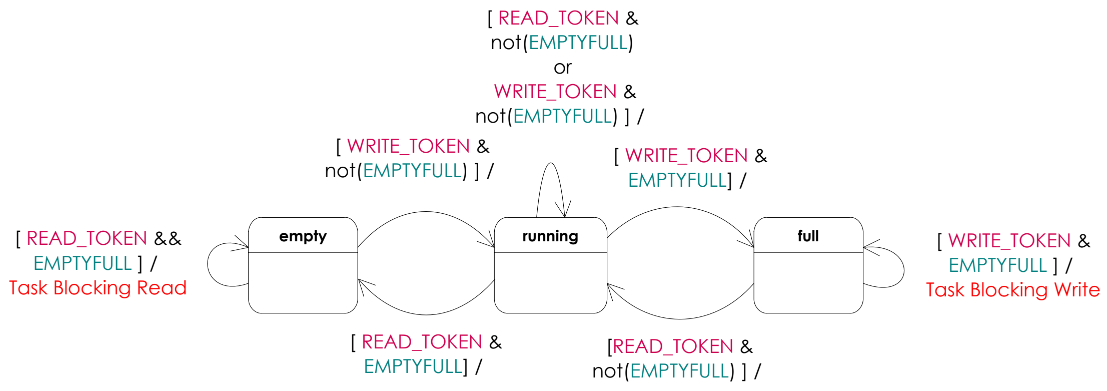


Figure 5-11
Address Generation Unit - FSM:
A task reads (READ_TOKEN) or writes (WRITE_TOKEN) FIFO items until a FIFO overflows or underflows. This exception occurs when the read address is equivalent to the write address and signalled by the EMPTY_FULL flag.
If that «exception» occurs either one additional item could be read/written or the FIFO is blocked. When the task is blocked, the blocking signal is asserted and spread over the «slotmanager». Usually, this effects a reconfiguration of the task slot.

5.3.4 cpucommunication

The cpucommunication part arrange the communication between the «slotmanager» and the IPAQ.

An internal clock is generated as proposed in section 5.2.3 and a enhanced FSM fetches the data words from the IPAQ tristate data bus and switches several internal enable signals.

VHDL ENTITY FOR CPUCOMMUNICATION

```
entity cpucommunication is
  port (
    ClkxCI             : in  std_logic;                          -- clock
    InternalClkxCO     : out std_logic;                          -- internal generated clock
    RstxRBI            : in std_logic;                           -- reset (active low)

    -- CPU Communication (Interface to IPAQ)
    CPUDataxZD         : inout std_logic_vector(15 downto 0);    -- tristate data bus
    CPUAddrxDI         : in    std_logic_vector(3 downto 0);     -- address bus
    CPUCSelectxEBI     : in    std_logic;                        -- chip select
    CPUWritexEBI       : in    std_logic;                        -- write enable
    CPUOutputxEBI      : in    std_logic;                        -- output enable
    CPUIntrxSO         : out   std_logic;                        -- interrupt to IPAQ

    -- Enable Fetch Data From CPU
    FetchCPUDataxEI     : in std_logic;                          -- enables fetching of cpu data

    -- Signaling to slotmanager_state: Config (CPU to FPGA) is DONE
    CommFPGAConfigDonexEO   : out std_logic;                     -- configuration is done
    CommFPGANewConfigxEO    : out std_logic;                     -- new configuration in pipeline

    -- SRAM Connections
    SRAMAddrxDO      : out std_logic_vector(17 downto 0);        -- SRAM addresses
    SRAMDataInxDI    : in  std_logic_vector(16 downto 1);        -- SRAM data in
    SRAMDataOutxDO   : out std_logic_vector(16 downto 1);        -- SRAM data in
    SRAMReadxEO      : out std_logic;                            -- SRAM read enable
    SRAMWritexEO    : out std_logic;                         -- SRAM write enable

    -- State Shift BRAM Connections
    -- Serial In / Out Interface
    StateBRAMSerialRAMxEI      : in   std_logic;                       -- BRAM enable
    StateBRAMSerialRAMWritexEI : in   std_logic;                       -- BRAM write enable
    StateBRAMSerialAddrxDI     : in   std_logic_vector(11 downto 0); -- BRAM address
    StateBRAMSerialDataInxDI   : in   std_logic;                       -- BRAM data in (1 bit)
    StateBRAMSerialDataOutxDO  : out  std_logic;                       -- BRAM data out (1 bit)

    -- Testsignals (for Modelsim)
    StateBRAMReadxTEO   : out std_logic;                     -- State BRAM read enable
    StateBRAMWritexTEO  : out std_logic;                     -- State BRAM write enable

    -- Transformation Port ID -> Fifo ID implemented in Distributed RAM
    TransformPortIDxDI   : in  PortIDType;                   -- Port ID
    TransformFifoIDxDO   : out std_logic_vector(7 downto 0); -- FIFO ID
    TransformIsReadxSO   : out std_logic;                    -- operation is read
    TransformIsWritexSO  : out std_logic;                    -- operation is write

    -- Task Blocking Information
    BLOCKINGxSI : in std_logic_vector(15 downto 0)          -- contains a '1' when BLOCKING
    );
end cpucommunication;
```

Listing 5-5
CPU Communication Entitiy

CPU DATA OUTPUT ON THE TRISTATE BUS

```
— applying signals on the tristate bus (CPU bus)
CPUDataxZD <= CPUDataOutxD when CPUOutputxEBI = '0' and CPUAddrxDI = DATA_REG else
              BLOCKINGxSI when CPUOutputxEBI = '0' and CPUAddrxDI = BLOCKING_REG else
              (others => 'Z');
```

Listing 5-6
CPU Output Data

5.3.4.1 Port to FIFO

In the Kahn Process Network the tasks communicate to each other via FIFOs which connect task in- and outputs. We call this in- and outputs Ports to abstract the data-flow direction. A Port of a specific task can be either an *Input* or a *Output* but not both. For practical reasons we enumerate the Ports with a **PortID** which is in the current implementation a natural number between 0 and 7 (NUMBER_OF_PORTS_PER_PROCESS).

In a graphical representation of the KPN the FIFOs are often not named or enumerated but for our application, we need to count and enumerate them with a **FifoID** — a number between 0 and 255.

In the current design we offer 8 PortID but 256 FIFOs that can be used. As a consequence we need a structure which enables the mapping from «virtual» to «real» IDs.

The part *PorttoFIFO* offers such a mechanism which maps the «virtual» PortIDs to «real» FifoID using a Look-Up-Table. This table is implemented in a dual-ported distributed SRAM structure (built with LUT4 Elements of the CLBs).

The graph resolution, enumerating of the PortID and FifoIDs is done in the IPAQ hand-held and afterwards loaded in the slotmanager for each initiated task. The Look-Up-Table is built as shown in figure 5-12.

The distributed dual-ported RAM offers two separate address ports, one for read and another for write operations. The address ports are completely asynchronous. We use the write port for writing the LUTs from the IPAQ interface (internally generated clock boundary) and the read port for the task execution (task clock boundary).

The WriteEnable enables a synchronous write operation of the RAM block words. The enable signal is controlled by the CPU communication protocol and a FSM.

VHDL ENTITY FOR PORTTOFIFO

```
entity porttofifo is
  port (
    ClkxCI           : in std_logic;                       — clock
    WritexEI         : in std_logic                        — write enable of the “RAM”
    ReadAddressxDI   : in std_logic_vector(3 downto 0);    — read address in
    WriteAddressxDI  : in std_logic_vector(3 downto 0);    — write address in
    DataInxDI        : in std_logic_vector(10 downto 0);   — data in
    DataOutOnexDO    : out std_logic_vector(10 downto 0);  — not connected
    DataOutTwoxDO    : out std_logic_vector(10 downto 0);  — data out
    );
end porttofifo;
```

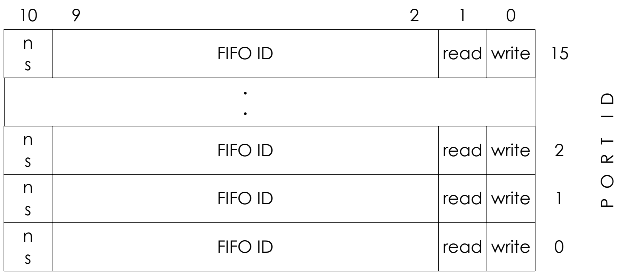


Figure 5-12
PortID to FifoID Translation Table - For each PortID the Translation Table contains a row composed of the following entries:
The tenth bit of each row (MSB) is the normal or special FIFO *flag. This flag contains the information if a FIFO is located in the external SRAM cell or the internal BlockRAM. The flag is on high level '1' if the FIFO is implemented in the BlockRAM, '0' otherwise. The idea behind this implementation is that such a BlockRAM-FIFO could be the in- or output of a driver task (i.e. an audio task).*
One of the $2^8 = 256$ *available FifoIDs is mapped to the bits 9 downto 2.*
The bits 1 and 0 signal that for the mapped FifoID, data items only could be read or written respectively. In the KPN model and the implemented framework a port can only read or write to a FIFO exclusively.

Listing 5-7
CPU Communication: Port to FIFO

5.3.4.2 StateBRAM

The StateBRAM is an instance of a dual ported *Xilinx SPARTAN^{TM} II* BlockRAM. In this case the two clock boundaries occurs again — the first is the internally generated CPU clock, the second the BTNODEFPGA task clock.

We use the BRAM as an asynchronous communication channel between these two clocks with the advantages that address and data ports of the instantiated element are parameterisable. We need a 16-bit data bus for the CPU communication at the internal clock side and a 1-bit data bus for task switching at the task side (serially shifting out the state bits of a task).

The ram and write enables of the StateBRAM can be asserted by specific CPU commands.

In the design only one BRAM is allocated for the StateBRAM which offers to store the task states of 4096 x 1 bits. To fetch this data at the CPU communication side 255 x 16 bits are necessarily.

VHDL ENTITY FOR STATEBRAM

```
entity statebram is
  port (
    -- Serial Part for Task Interface (1 bit in/out)
    SerialClkxCI       : in   std_logic;                         -- serial ram clock
    SerialRAMxEI       : in   std_logic;                         -- serial ram enable
    SerialRAMWritexEI  : in   std_logic;                         -- serial ram write enable
    SerialAddrxDI      : in   std_logic_vector(11 downto 0);     -- serial ram addr
    SerialDataInxDI    : in   std_logic;                        -- serial data in
    SerialDataOutxDO   : out  std_logic;                        -- serial data out

    -- Parallel Part for CPU Interface (16 bit in/out)
    ParallelClkxCI       : in   std_logic;                      -- parallel ram clock
    ParallelRAMxEI       : in   std_logic;                      -- parallel ram enable
    ParallelRAMWritexEI  : in   std_logic;                      -- parallel ram write enable
    ParallelAddrxDI      : in   std_logic_vector(7 downto 0);   -- parallel ram addr
    ParallelDataInxDI    : in   std_logic_vector(15 downto 0);  -- parallel data in
    ParallelDataOutxDO   : out  std_logic_vector(15 downto 0)   -- parallel data out
    );
end statebram;
```

Listing 5-8
CPU Communication: StateBRAM

5.3.4.3 BRAM for Driver Tasks

Driver Task like an audio task are slots which needs the FIFO items directly on the FPGA. To respect such an application NUMBER_OF_FIFO_BRAM BlockRAMs are instantiated which offers a bounded easy accessible SRAM space. The BRAM can be read and written by the CPU directly by using the communication protocol.

VHDL ENTITY FOR BRAM

```
entity bram is
  generic (
    NUMBER_OF_FIFO_BRAM : natural := 8                  -- number of blockrams
    );
  port (
    ClkxCI       : in   std_logic;                      -- bram clock
    RAMxEI       : in   std_logic;                      -- bram enable
    RAMWritexEI  : in   std_logic;                      -- bram write enable
    AddrxDI      : in   std_logic_vector(10 downto 0);  -- bram addr
    DataInxDI    : in   std_logic_vector(15 downto 0);  -- bram data in
    DataOutxDO   : out  std_logic_vector(15 downto 0)   -- bram data out
    );
end bram;
```

Listing 5-9
CPU Communication: BRAM

5.3.4.4 CPU Protocol

During execution of slotmanager the following commands (tab. 5-2) are available on the CPU data bus.

Command	Subcommand	Encoding
		1111'1100'0000'0000
		5432'1098'7654'3210
Not used		000-'----'----'----
Read / Write State BRAM		001-'NNNN'NNNN'----
	Read	0010'NNNN'NNNN'----
	Write	0011'NNNN'NNNN'----
Read / Write SRAM		010-'NNNN'NNNN'--AA
	Read	0100'NNNN'NNNN'--AA
	Write	0101'NNNN'NNNN'--AA
Read / Write BRAM		011-'NNNN'NNNN'--AA
	Read	0110'NNNN'NNNN'--AA
	Write	0111'NNNN'NNNN'--AA
Configure normal fifo ports		011-'PPPP'FFFF'FFFF
	Read	0110'PPPP'FFFF'FFFF
	Write	0111'PPPP'FFFF'FFFF
Configure special fifo ports		101-'PPPP'FFFF'FFFF
	Read	1010'PPPP'FFFF'FFFF
	Write	1011'PPPP'FFFF'FFFF
Transfer Command		110-'----'----'----
	StateBRAM Write Enable	110-'-1--'----'---1
	StateBRAM Write Disable	110-'-1--'----'---0
	StateBRAM Read Enable	110-'--1-'----'---1
	StateBRAM Read Disable	110-'--1-'----'---0
	BRAM Write Enable	110-'---1'----'---1
	BRAM Write Disable	110-'---1'----'---0
	BRAM Read Enable	110-'----'1---'---1
	BRAM Read Disable	110-'----'1---'---0
	FPGA Config Flag Enable	110-'----'-1--'---1
	FPGA Config Flag Disable	110-'----'-1--'---0
	FPGA New Bitstream Flag Enable [1]	110-'----'--1-'---1
	FPGA New Bitstream Flag Disable [1]	110-'----'--1-'---0
Not used		111-'----'----'----

Table 5-2: Communication IPAQ to FPGA KPN: Command Words to manage and handle the KPN execution framework on the FPGA. The capitals symbolise these data bits:
N: Number of Words
A: MSBs of the SRAM Address (Bit 18 and Bit 17)
P: PortID
F: FifoID

[1] *not implemented until now*

5.3.5 task_wrapper

The task_wrapper connects a user Task with the slotmanager. The communication between the task and the framework is done by *read* and *write* operation as proposed in the KPN implementation of Parks [24].

Two signal wires are routed directly to 2 leds of the BTNODEFPGA and can be used for flagging computation phases to the user. An overview of the wrapper is given in figure 5-35 on 80.

In the modul exists additional logic for automatically load and store internal states. Brief — a detailed view is in section 5.6 — we shift the internal states serial in and out; an analogy is a linear shift register. In our implementation we use a BRAM as a storage, the ContextSwitch for FSM communication and the task_chainer for automation of this shifting process.

VHDL ENTITY FOR TASK_WRAPPER

```
entity task_wrapper is
  port (
    ClkxCI  : in std_logic;                                  -- clock
    RstxRBI : in std_logic;                                  -- reset (active low)
    -------------------------------
    -- Task to FIFO Communication
    -------------------------------
    PortIDxDO           : out PortIDType;                    -- Port ID
    PortReadxSO         : out std_logic;                     -- Likes to Read from Port
    PortWritexSO        : out std_logic;                     -- Likes to Write to Port
    PortReadWriteAckxEI : in  std_logic;                     -- Ackn. Read / Write
    PortDataInxDI       : in  std_logic_vector(15 downto 0); -- Data In
    PortDataOutxDO      : out std_logic_vector(15 downto 0); -- Data Out
    -------------------------------
    -- Context Switch
    -------------------------------
    ContextSwitchxSI     : in  std_logic;                    -- do context switch
    ContextSwitchDonexSO : out std_logic;                    -- context switch done
    -------------------------------
    -- Flip-Flop Enable
    -------------------------------
    FlipFlopxEI : in std_logic;                              -- General FlipFlop Enable
    -------------------------------
    -- StateBRAM
    -------------------------------
    StateBRAMSerialRAMxEO      : out std_logic;              -- StateBRAM Enable
    StateBRAMSerialRAMWritexEO : out std_logic;              -- StateBRAM Write Enable
    StateBRAMSerialAddrxDO     : out std_logic_vector(11 downto 0); -- StateBRAM Address
    StateBRAMSerialDataInxDI   : in  std_logic;              -- StateBRAM Data in (1bit)
    StateBRAMSerialDataOutxDO  : out std_logic;              -- StateBRAM Data out (1bit)
    -------------------------------
    -- CUSTOM LED
    -------------------------------
    CustomLed2xSO : out std_logic;                           -- LED 2 of BTNodeFPGA
    CustomLed3xSO : out std_logic                            -- LED 3 of BTNodeFPGA
  );
```

Listing 5-10
Task Wrapper

The idea behind the module task_chainer is, that we have implemented a script which is able to insert a «scan-chain» automatically. The length of the chain could be saved in a variable.

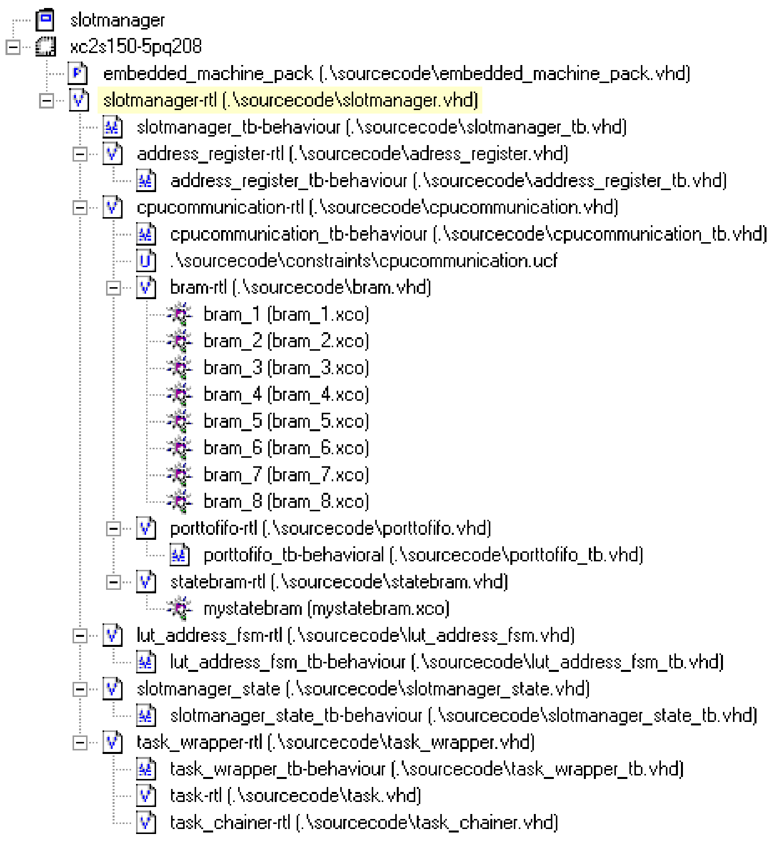
In the execution framework the «reach-end-of-chain» of the shifting process has to be signalised to the slotmanager. This is done by a counter and a comperator-function which is generated in VHDL (std_match(counter,chain_length)). The natural value of chain_length could be inserted from the script.

VHDL CODE OF TASK_CHAINER

```
entity task_chainer is
  generic (
    CHAIN_LENGTH : std_logic_vector(11 downto 0) := X"FFF" -- length of the chain
    );
  port (
    ClkxCI        : in std_logic                              -- clock
    RstxRBI       : in std_logic;                             -- reset (active low)
    -- counter value
    CounterxDO    : out std_logic_vector(11 downto 0);        -- used as StateBRAM address
    CounterxEI    : in  std_logic;                            -- enable counter
    -- all shifted
    AllShiftedxEO : out std_logic);                           -- all states shifted
end task_chainer;

architecture rtl of task_chainer is

  signal CountxDP      : std_logic_vector(11 downto 0) := X"000";
  signal AllShiftedxE : std_logic;

begin  -- rtl

  AllShiftedxEO <= AllShiftedxE;
  CounterxDO <= COUNTxDP;

  -- purpose: Flagging with aid of counter
  -- type : sequential
  -- inputs : ClkxCI, RstxRBI, StateBRAMLengthChainxD
  -- outputs: StateBRAMAllShiftedxE
  flaggingCounter : process (ClkxCI, RstxRBI)
  begin  -- process flaggingCounter
    if RstxRBI = '0' then        -- asynchronous reset (active low)
      CountxDP <= (others => '0');
    elsif ClkxCI'event and ClkxCI = '1' then  -- rising clock edge
      if CounterxEI = '1' then
        CountxDP <= CountxDP + 1;
      elsif AllShiftedxE = '1' then
        CountxDP <= (others => '0');
      else
        CountxDP <= CountxDP;
      end if;
    end if;
  end process flaggingCounter;

  -- purpose: flags when counter reaches chain length (SCRIPT!)
  -- type : combinational
  -- inputs : CountxDP, StateBRAMLengthChainxD
  -- outputs: StateBRAMAllShiftedxE
  flagging : process (CountxDP)
  begin  -- process flagging
    AllShiftedxE <= '0';
    if std_match(CountxDP, CHAIN_LENGTH) then
      AllShiftedxE <= '1';
    end if;
  end process flagging;

end rtl;
```

Listing 5-11
Task Chainer: A mechanism to determine to end of the shifting process automatically.

5.3.6 Synthesis & Routing Data



```
slotmanager
xc2s150-5pq208
  embedded_machine_pack (.\sourcecode\embedded_machine_pack.vhd)
  slotmanager-rtl (.\sourcecode\slotmanager.vhd)
    slotmanager_tb-behaviour (.\sourcecode\slotmanager_tb.vhd)
    address_register-rtl (.\sourcecode\adress_register.vhd)
      address_register_tb-behaviour (.\sourcecode\address_register_tb.vhd)
    cpucommunication-rtl (.\sourcecode\cpucommunication.vhd)
      cpucommunication_tb-behaviour (.\sourcecode\cpucommunication_tb.vhd)
      .\sourcecode\constraints\cpucommunication.ucf
      bram-rtl (.\sourcecode\bram.vhd)
        bram_1 (bram_1.xco)
        bram_2 (bram_2.xco)
        bram_3 (bram_3.xco)
        bram_4 (bram_4.xco)
        bram_5 (bram_5.xco)
        bram_6 (bram_6.xco)
        bram_7 (bram_7.xco)
        bram_8 (bram_8.xco)
      porttofifo-rtl (.\sourcecode\porttofifo.vhd)
        porttofifo_tb-behavioral (.\sourcecode\porttofifo_tb.vhd)
      statebram-rtl (.\sourcecode\statebram.vhd)
        mystatebram (mystatebram.xco)
    lut_address_fsm-rtl (.\sourcecode\lut_address_fsm.vhd)
      lut_address_fsm_tb-behaviour (.\sourcecode\lut_address_fsm_tb.vhd)
    slotmanager_state (.\sourcecode\slotmanager_state.vhd)
      slotmanager_state_tb-behaviour (.\sourcecode\slotmanager_state_tb.vhd)
    task_wrapper-rtl (.\sourcecode\task_wrapper.vhd)
      task_wrapper_tb-behaviour (.\sourcecode\task_wrapper_tb.vhd)
      task-rtl (.\sourcecode\task.vhd)
      task_chainer-rtl (.\sourcecode\task_chainer.vhd)
```

Figure 5-13
Slotmanager Design : Project View in the ISE

Data-sheet of the FPGA (Slotmanager) after Synthesis with *Xilinx XST* and fitting process with *Xilinx FIT* .

■ **slotmanager**

- CPUWritexEBI
  - *buffer_type* = ibuf
  - *clock_signal* = yes
- CPUOutputxEBI
  - *buffer_type* = ibuf
  - *clock_signal* = yes
- InternalClkxC
  - *clock_signal* = yes

■ **address_register**

- NUMBER_OF_PORTS_PER_PROCESS = 8

■ **cpucommunication**

- FetchFSM
  - *States* : 10
  - *Transitions* : 28
  - *Input* : 12
  - *Output* : 9
  - *Clock* : InternalClkxC

- **bram**
  - NUMBER_OF_FIFO_BRAM = 1
- **porttofifo**
  - INSTANCES
    - 11 x *RAM16X1D*
- **lut_address_fsm**
  - AddressFSM
    - *States* : 5
    - *Transitions* : 6
    - *Input* : 1
    - *Output* : 5
- **slotmanager_state**
  - SlotManagerFSM
    - *States* : 9
    - *Transitions* : 20
    - *Input* : 8
    - *Output* : 9
  - ConfigSlotManagerFSM
    - *States* : 8
    - *Transitions* : 19
    - *Input* : 4
    - *Output* : 7
- **task_chainer**
  - CHAIN_LENGTH = <u>000000001111

Overview of the used elements, CLBs and timings:

- **Elements**
  - *Register*: 9x1-bit, 16x2-bit, 1x3-bit, 2x8-bit, 1x10-bit, 1x16-bit and 33x18-bit
  - *Comparator*: 1x8bit and 3x18-bit comparators equal
  - *Adders*: 1x8-bit and 3x18-bit
  - *Subtractors*: 1x18-bit
  - *Counters*: 1x8-bit and 1x12-bit
  - *MUXs*: 648
  - *Flip-Flops*: 736
  - *Tristates*: 2x16-bit and 1x18-bit tristate buffer
  - *Clock Buffers*: 2
  - *RAMs*: 11xRAM16X1D, 1xRAMB4_S16 and 1xRAMB4_S1_S16
  - *I/O Pins*: 83 (+ 1 Clock Pin)
- **CLBs**
  - *Slice flip-flops*: 720 / 4707 (15%)
  - *Used 4-LUTs*: 1074 / 4704 (22%)
  - *Occupied Slices*: 838 / 2352 (35%)
  - *Bonded IOB*: 83 / 140 (59%)
  - *IOB flip-flops*: 16 / 140
  - *GCLKs*: 2 / 4 (50%)
  - *GCLKIOBs*: 1 / 4 (25%)
- **Clocks**
  - ClkxCI
    - *Max. Period*: 42.8 ns
    - *Fanout*: 373
    - *Net Skew*: 0.489 ns
    - *Max Delay*: 0.759 ns
    - *Logic Levels*: 18
  - InternalClkxC
    - *Max. Period*: 23.3 ns
    - *Fanout*: 62
    - *Net Skew*: 0.493 ns
    - *Max Delay*: 0.763 ns
    - *Logic Levels*: 1

5.4 Loader on FPGA

5.4.1 System Overview

In addition to the FPGA configuration for the KPNexecution framework there exists a special configuration of the FPGA device at the moment – the «Loader». This configuration acts as a glue-logic for the loading setup between the IPAQ, the CPLD and the FLASH device.

The loader consists of two parts: A CPU part which is able to communicate with the IPAQ over the I/O interface, and a SERIAL part which includes an RS-232 interface. The RS-232 interface was thought for stand-alone execution of the Slotmanager and as a debug interface.

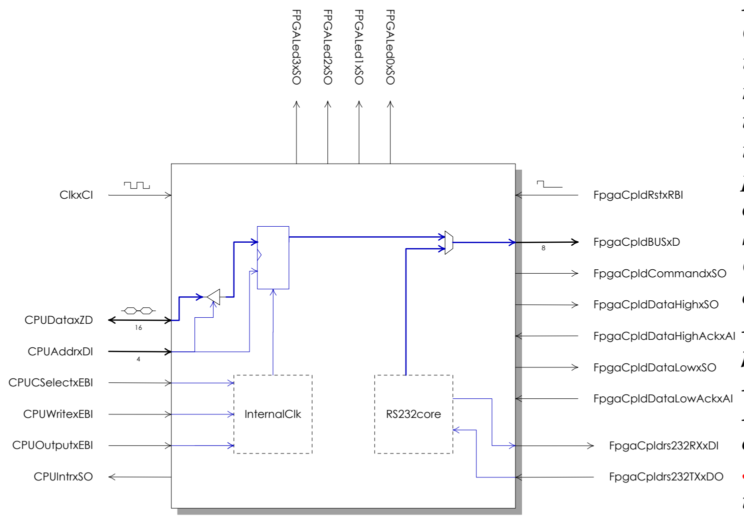


Figure 5-14
Loader on FPGA - DataFlow:
On the left hand side of the figure the CPU interface, on the right hand side the interface to the CPLD is shown. The Loader is subdivided into 2 functional parts a CPU and a SERIAL communication part.
Serial Data is fed through the CPLD to the RS-232 core; the data here are evaluated by a FSM and the data words transferred back to the CPLD.
For communication with the IPAQ the internal clock is generated as described in section 5.2.3. A data word on the tristate bus `CPUDataxZD` will be fetched if the `CPUAddrxDI` has the value '0' and the internal clock triggers a rising clock edge.

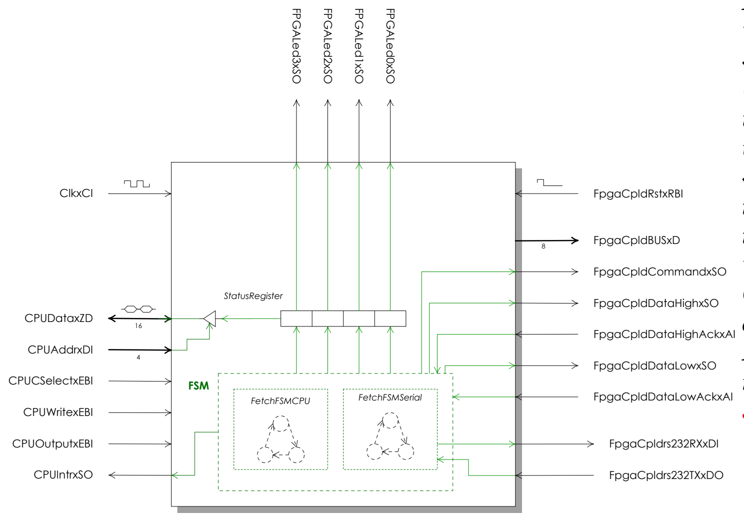


Figure 5-15
Loader on FPGA - ControlFlow:
The Loader will assert the «Status Register» to the `CPUDataxZD` bus if the address wires `CPUAddrxDI` builds the value "F". Some flags of the «Status Register» are connected to the Leds for visual signalling to a user.
The loader consists of two FSM. One for the CPU part and another for the SERIAL part. The FSM is explained more precise in sec. 5.4.3 on page 58 and 5.4.4 on page 62.

5.4.2 Commands (Serial and CPU Protocol)

Synchronisation Word	Encoding
`COMM_COMMAND_1`	C h
`COMM_COMMAND_2`	A h
`COMM_COMMAND_3`	F h
`COMM_COMMAND_4`	E h
`COMM_COMMAND_5`	A h
`COMM_COMMAND_6`	F h

Table 5-3: Communication To FPGA:
Synchronisation words which are used in a preamble before the effective commando word or bitstream will be transferred.

Commando Word for FSM transitions	Encoding
`COMM_COMMAND_DUMMY`	0 h
`COMM_COMMAND_RESET`	1 h
`COMM_COMMAND_MANUFACTURER_ID` [1]	2 h
`COMM_COMMAND_DEVICE_ID` [1]	3 h
`COMM_COMMAND_SECTOR_PROT_VERIFY` [1]	4 h
`COMM_COMMAND_PROGRAM`	5 h
`COMM_COMMAND_UNLOCK_BYPASS` [1]	6 h
`COMM_COMMAND_UNLOCK_BYPASS_PROG` [1]	7 h
`COMM_COMMAND_UNLOCK_BYPASS_RESET` [1]	8 h
`COMM_COMMAND_CHIP_ERASE`	9 h
`COMM_COMMAND_SECTOR_ERASE`	A h
`COMM_COMMAND_ERASE_SUSPEND` [1]	B h
`COMM_COMMAND_ERASE_RESUME` [1]	C h
`COMM_COMMAND_READ_BACK` [3]	D h
`COMM_COMMAND_SLAVE_PARALLEL`	E h
`COMM_COMMAND_INTERRUPT` [2]	F h

Table 5-4: Communication to FPGA and CPLD - Command Words:
*The commands triggers the FPGA and CPLD FSM and handles finally the FLASH Memory. **COMM_COMMAND_RESET** until **COMM_COMMAND_ERASE_RESUME** are the commands offered by the internally included FSM of the FLASH.*
***COMM_COMMAND_READ_BACK** could be used when a read-back port is foreseen. **COMM_COMMAND_SLAVE_PARALLEL** enables the Slave Parallel Interface of the Spartan II to reconfigure the FPGA Configuration. **COMM_COMMAND_INTERRUPT** will generate an Interrupt at the CPU Interface.*

[1] *not activated (commented out)*
[2] *not implemented until now*
[3] *not implemented until now - feedback channel not available*

Select Sector	Commando Word	Encoding	Starting Address	End Address
0	COMM_SECTOR_0	0 h	00000 h	28C44 h
1	COMM_SECTOR_1	1 h	10000 h	38C44 h
2	COMM_SECTOR_2	2 h	20000 h	48C44 h
3	COMM_SECTOR_3	3 h	30000 h	58C44 h
4	COMM_SECTOR_4	4 h	40000 h	68C44 h
5	COMM_SECTOR_5	5 h	50000 h	78C44 h
6	COMM_SECTOR_6	6 h	60000 h	88C44 h
7	COMM_SECTOR_7	7 h	70000 h	98C44 h
8	COMM_SECTOR_8	8 h	80000 h	A8C44 h
9	COMM_SECTOR_9	9 h	90000 h	B8C44 h
10	COMM_SECTOR_10	A h	A0000 h	C8C44 h
11	COMM_SECTOR_11	B h	B0000 h	D8C44 h
12	COMM_SECTOR_12	C h	C0000 h	E8C44 h
13	COMM_SECTOR_13	D h	D0000 h	F8C44 h
14	COMM_SECTOR_14	E h	E0000 h	08C44 h
15	COMM_SECTOR_15	F h	F0000 h	18C44 h

Table 5-5: Communication to CPLD and FPGA - Select Sectors:
The sector «information» included in a command word enables an address relocation in the CPLD. As proposed in the table 5-1 on page 33 this scheme is also used for storing multiple bitstreams on the FLASH.

5.4.3 Loader - 8 Bit Transfer Implementation

The data bus at the IPAQ expansion port has a width of 16-bit. To transfer a new bitstream from the IPAQ to the FLASH memory in 16-bit chunks, we have to transfer these data words over a data bus from the FPGA to the CPLD.

Normally, a data transfer from one to another device requires a (minimal) handshaking or a flagging — like a DataReady or a DataAcknowledge signal. Unfortunately, there are only 16 connections between the FPGA and the CPLD on the BTNODEFPGA board and there is no possibility to use another communication channel to the CPLD without affecting the other devices. As a solution we use only 8-bits as a data bus, the other 8-bits are used for inter-device control flow.

During the implementation phase of our system there were some problems with the complex inter-chip communication. To be sure that the communication from the IPAQ via the self-made bridge and the FPGA to the CPLD is working, we use an 1 Byte data transfer mode. After assertion of a command and control sequence to the FPGA only the *higher* 8-bits of the IPAQ data bus interface is used for bitstream transfer; the *lower* 8-bits contain dummy words (the same values as the higher bits).

With this method we have created a «tunnel mode» which allows us to write the new bitstream from the IPAQ application directly to the CPLD and the FLASH memory without any data conversion.

An advantage of this transfer mode is the lax timing. There is plenty of time for the Flash to finish its writing procedure before a new data word arrives at the input.

5.4.3.1 Communication Protocol

The FPGA acts as a «sender», the CPLD as a «receiver» of the byte stream. The timing diagram for the sender implementation is figured in 5-17. A detailed implementation of the receiver in the CPLD is given in section 5.5 on page 66.

5.4.3.2 Synthesis & Routing Data

In the following the implementation data of the FPGA (Loader) after Synthesis with *Xilinx XST* and fitting process with *Xilinx FIT* are listed.

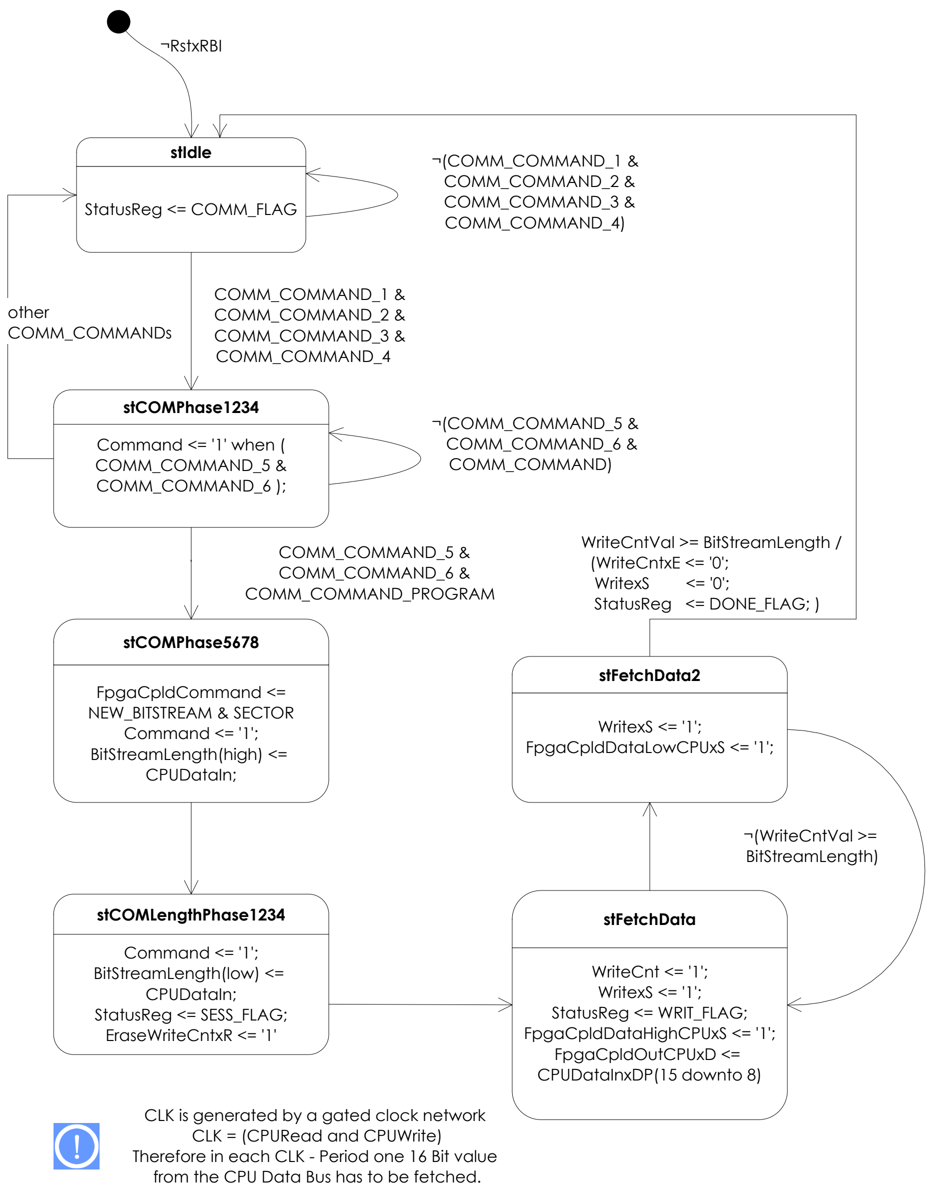

¬RstxRBI
stIdle
StatusReg <= COMM_FLAG
¬(COMM_COMMAND_1 & COMM_COMMAND_2 & COMM_COMMAND_3 & COMM_COMMAND_4)
other COMM_COMMANDs
COMM_COMMAND_1 & COMM_COMMAND_2 & COMM_COMMAND_3 & COMM_COMMAND_4
stCOMPhase1234
Command <= '1' when (COMM_COMMAND_5 & COMM_COMMAND_6);
¬(COMM_COMMAND_5 & COMM_COMMAND_6 & COMM_COMMAND)
COMM_COMMAND_5 & COMM_COMMAND_6 & COMM_COMMAND_PROGRAM
WriteCntVal >= BitStreamLength / (WriteCntxE <= '0'; WritexS <= '0'; StatusReg <= DONE_FLAG;)
stCOMPhase5678
FpgaCpldCommand <= NEW_BITSTREAM & SECTOR Command <= '1'; BitStreamLength(high) <= CPUDataIn;
stFetchData2
WritexS <= '1'; FpgaCpldDataLowCPUxS <= '1';
¬(WriteCntVal >= BitStreamLength)
stCOMLengthPhase1234
Command <= '1'; BitStreamLength(low) <= CPUDataIn; StatusReg <= SESS_FLAG; EraseWriteCntxR <= '1'
stFetchData
WriteCnt <= '1'; WritexS <= '1'; StatusReg <= WRIT_FLAG; FpgaCpldDataHighCPUxS <= '1'; FpgaCpldOutCPUxD <= CPUDataInxDP(15 downto 8)
CLK is generated by a gated clock network CLK = (CPURead and CPUWrite) Therefore in each CLK - Period one 16 Bit value from the CPU Data Bus has to be fetched.

Figure 5-16
Loader on FPGA: FSM for the CPU Protocol (8 Bit)
The FSM waits in the Idle-state until a 16 bit data word from the IPAQ arrives on `CPUDataxZD` *which contains the synchronisation sequence "CAFE" (tab. 5-3). If such a word arrives the automata switches to the COMPhase1234-state. The next data word on the IPAQ ↔ FPGA bus should contain two additional synchronisation half-bytes "AF", the execution command for the CPLD (tab. 5-4) and the selected sector if necessary (tab. 5-5). If such a command arrives the FSM will transfer the command and the sector information to the CPLD.*
If the command is COMM_COMMAND_PROGRAM, a transition to the COMPhase5678-state will be done, otherwise to the Idle-state. In the COMPhase5678- and COMLengthPhase1234-state the length of the bitstream is fetched from the FPGA, internal counters are reseted and a communication session to the CPLD is initiated. FetchData and FetchData2 are called `BitStreamLength` *times until the complete bitstream is transferred to the CPLD and the FLASH memory.*

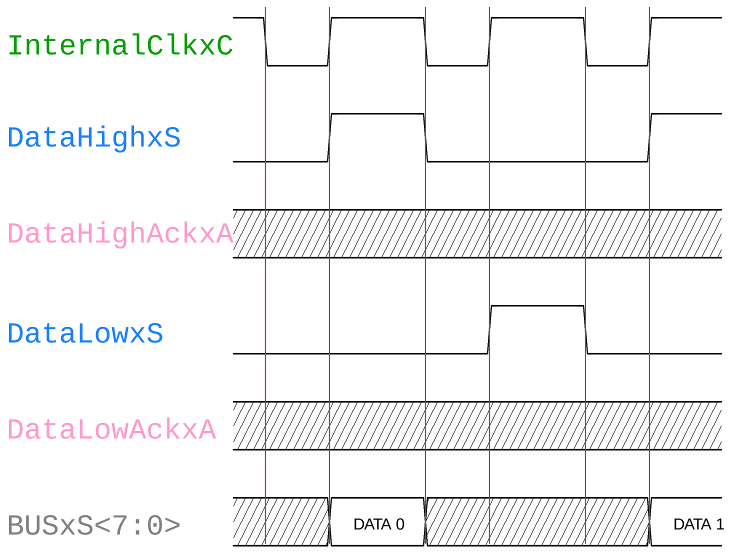


Figure 5-17
Data Transfer Protocol
InternalClkxC is the generated clock in the FPGA and only internally used. DataHighxS and DataLowxS are the communication flags which are transfered from the FPGA to the CPLD device. The communication channel from the CPLD to the FPGA (DataHighAckxA and DataLowAckxA) are not used in this setup.

- **fpga_flash_loader**
  - CPUWritexEBI
    - *buffer_type* = ibuf
    - *clock_signal* = yes
  - CPUOutputxEBI
    - *buffer_type* = ibuf
    - *clock_signal* = yes
  - InternalClkxC
    - *clock_signal* = yes
  - FSM FetchFsmCPUxDP
    - *States* : 6
    - *Transitions* : 12
    - *Inputs* : 6
    - *Outputs* : 6
    - *Encoding* : one-hot
  - FSM FetchFsmSERIALxDP
    - *States* : 12
    - *Transitions* : 38
    - *Inputs* : 9
    - *Outputs* : 14
    - *Encoding* : one-hot

- **rs232core**
  - CONSTANTS
    - *Clockfreq* = 18432000
    - *Baudrate* = 115200
  - FSM senderstate
    - *States* : 2
    - *Transitions* : 2
    - *Inputs* : 2
    - *Outputs* : 2
    - *Encoding* : one-hot
  - FSM receiverstate
    - *States* : 5
    - *Transitions* : 10
    - *Inputs* : 3
    - *Outputs* : 5
    - *Encoding* : one-hot
  - shiftreg
    - *WIDTH* = 10
  - counter
    - *WIDTH* = 13

- **Elements**
  - *Registers*:
    2x1-bit, 2x4-bit, 1x8-bit, 1x10-bit, 4x13-bit, 2x16-bit and 2x32-bit
  - *Comparator*:
    4x32-bit comparators greater-equal

- **CLBs**
  - *Slice flip-flops*:
    208 / 4707 (4%)
  - *Used 4-LUTs*:
    507 / 4704 (10%)
  - *Occupied Slices*:
    267 / 2352 (11%)
  - *Bonded IOBs*:

■ **Clocks**

- ClkxCI
  - *Max. Period*: 26.6 ns
  - *Fanout*: 91
  - *Net Skew*: 0.163 ns
  - *Max Delay*: 0.763 ns
- InternalClkxC
  - *Max. Period*: 13.8 ns
  - *Fanout*: 53
  - *Net Skew*: 0.169 ns
  - *Max Delay*: 0.763 ns

■ **Timings**

- *Max. Pin Delay*: 9.527 ns

5.4.4 Loader - 16 Bit Transfer Implementation

This implementation was the primary implementation of the data transfer between IPAQ, FPGA and CPLD.

The throughput to the Flash memory is doubled versus the last proposed transfer mode: Now, the IPAQ data stream consists of 2 Bytes bitstream data consequently. The new bitstream has a preamble for the synchronisation with the FSMs, a bitstream length and the – raw but snipped – bitstream configuration data. There is no need to insert dummy bytes.

The lack of bus wires between the FPGA and the CPLD for a 16-bit data transfer still exists. As a solution we transfer the IPAQ data word from the FPGA to the CPLD in two 8-bit values with the aid of a four phases handshaking protocol. A four phase handshaking protocol allows asynchronous and reliable data transfers (see also section 5.7 on page 76).

The implementation idea is a partial asynchronous data transfer. The «receiver» on the CPLD is a Moore automata and therefore synchronous with the board clock. The «sender» is implemented as a Mealy automata — the state transition and output data depend also on the inputs between clock edges. The corresponding Final State Machines are shown in these figures: The Moore automata of the Loader on FPGA is explained in two parts. Figure 5-18 contains the FSM for the CPU communication, figure 5-21 for the SERIAL communication. The «sender» state machine is sketched in figure 5-19.

The handshaking protocol is implemented with four command lines between the devices: `DataHighxS` and `DataLowxS` signal new and valid data words (8-bit) on the data bus `BUSxD` to the CPLD. The receiving of these words are acknowledged with the `DataHighAckxA` and `DataLowAckxA` signals. The timing diagram is shown in figure 5-20.

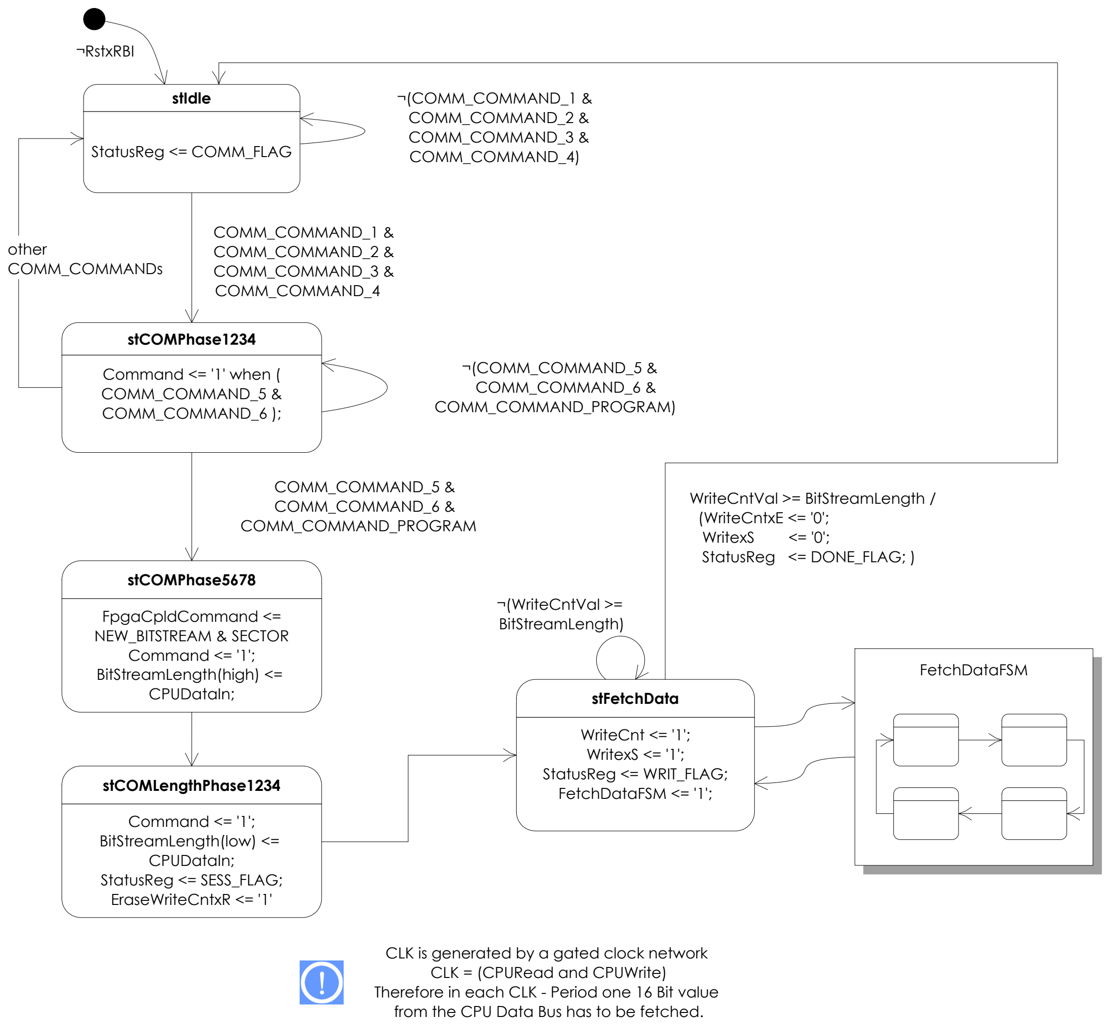

¬RstxRBI
stIdle
StatusReg <= COMM_FLAG
¬(COMM_COMMAND_1 & COMM_COMMAND_2 & COMM_COMMAND_3 & COMM_COMMAND_4)
other COMM_COMMANDs
COMM_COMMAND_1 & COMM_COMMAND_2 & COMM_COMMAND_3 & COMM_COMMAND_4
stCOMPhase1234
Command <= '1' when (COMM_COMMAND_5 & COMM_COMMAND_6);
¬(COMM_COMMAND_5 & COMM_COMMAND_6 & COMM_COMMAND_PROGRAM)
COMM_COMMAND_5 & COMM_COMMAND_6 & COMM_COMMAND_PROGRAM
WriteCntVal >= BitStreamLength / (WriteCntxE <= '0'; WritexS <= '0'; StatusReg <= DONE_FLAG;)
stCOMPhase5678
FpgaCpldCommand <= NEW_BITSTREAM & SECTOR Command <= '1'; BitStreamLength(high) <= CPUDataIn;
¬(WriteCntVal >= BitStreamLength)
stFetchData
WriteCnt <= '1'; WritexS <= '1'; StatusReg <= WRIT_FLAG; FetchDataFSM <= '1';
FetchDataFSM
stCOMLengthPhase1234
Command <= '1'; BitStreamLength(low) <= CPUDataIn; StatusReg <= SESS_FLAG; EraseWriteCntxR <= '1'
CLK is generated by a gated clock network CLK = (CPURead and CPUWrite) Therefore in each CLK - Period one 16 Bit value from the CPU Data Bus has to be fetched.

Figure 5-18
Loader on FPGA: FSM for the CPU Protocol (16 Bit)
The FSM waits in the Idle-state until a 16 bit data word from the IPAQ arrives on `CPUDataxZD` *which contains the synchronisation sequence "CAFE". If such a word arrives the automata switches to the COMPhase1234-state. The next data word should contain two additional synchronisation half-bytes "AF", the execution command for the CPLD and the selected sector if necessary. If such a command arrives the FSM will transfer the command and the sector information to the CPLD.*
If the command is `COMM_COMMAND_PROGRAM`*, a transition to the COMPhase5678-state will be done, otherwise to the Idle-state. In the COMPhase5678- and COMLengthPhase1234-state the length of the bitstream is fetched from the FPGA, internal counters are reseted and a communication session to the CPLD is initiated.*
The Mealy «sender» FSM will be activated if the FSM reaches the FetchData-state. The bitstream data items will be received `BitStreamLength`*/2 times and – controlled by the «sender» FSM – transferred to the CPLD.*

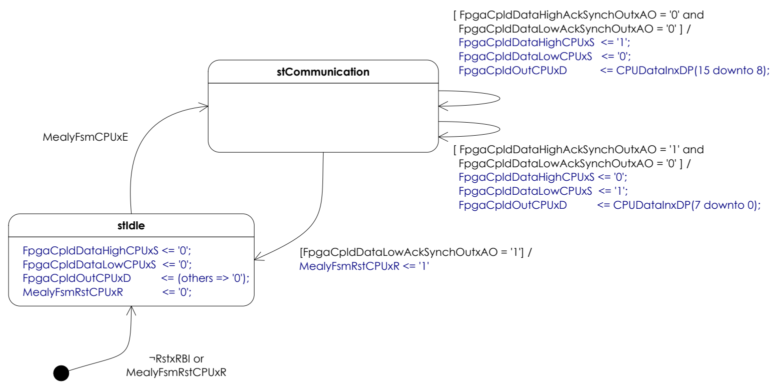

[FpgaCpldDataHighAckSynchOutxAO = '0' and
FpgaCpldDataLowAckSynchOutxAO = '0'] /
FpgaCpldDataHighCPUxS <= '1';
FpgaCpldDataLowCPUxS <= '0';
FpgaCpldOutCPUxD <= CPUDataInxDP(15 downto 8);
stCommunication
MealyFsmCPUxE
[FpgaCpldDataHighAckSynchOutxAO = '1' and
FpgaCpldDataLowAckSynchOutxAO = '0'] /
FpgaCpldDataHighCPUxS <= '0';
FpgaCpldDataLowCPUxS <= '1';
FpgaCpldOutCPUxD <= CPUDataInxDP(7 downto 0);
stIdle
FpgaCpldDataHighCPUxS <= '0';
FpgaCpldDataLowCPUxS <= '0';
FpgaCpldOutCPUxD <= (others => '0');
MealyFsmRstCPUxR <= '0';
[FpgaCpldDataLowAckSynchOutxAO = '1'] /
MealyFsmRstCPUxR <= '1'
¬RstxRBI or
MealyFsmRstCPUxR

Figure 5-19
Loader on FPGA: Mealy automata for data sender
This Mealy automata controls the data transfer from the FPGA to the CPLD device. In the initial Idle-state there is no valid data word at the data bus and the two control signals are on low level '0'. If the Mealy FSM is enabled by the CPU FSM with the `MealyFSMCPUxE` transition the automata switchs to the Communication-state. The output of this state depends on the acknowledge *signals and as a last transition* — DataLowAckxA = '1' — *the automata resets itself back to the Idle-state.*

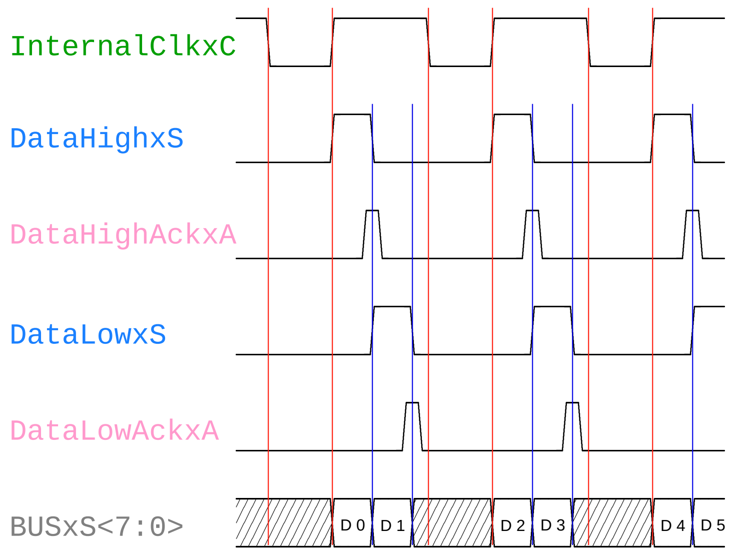


Figure 5-20
Transfer Protocol
Transfer of two 8-bit values in only one InternalClk high phase with the aid of a four phase handshaking protocol. The transferred bytes are symbolised by D0 . . . D5.

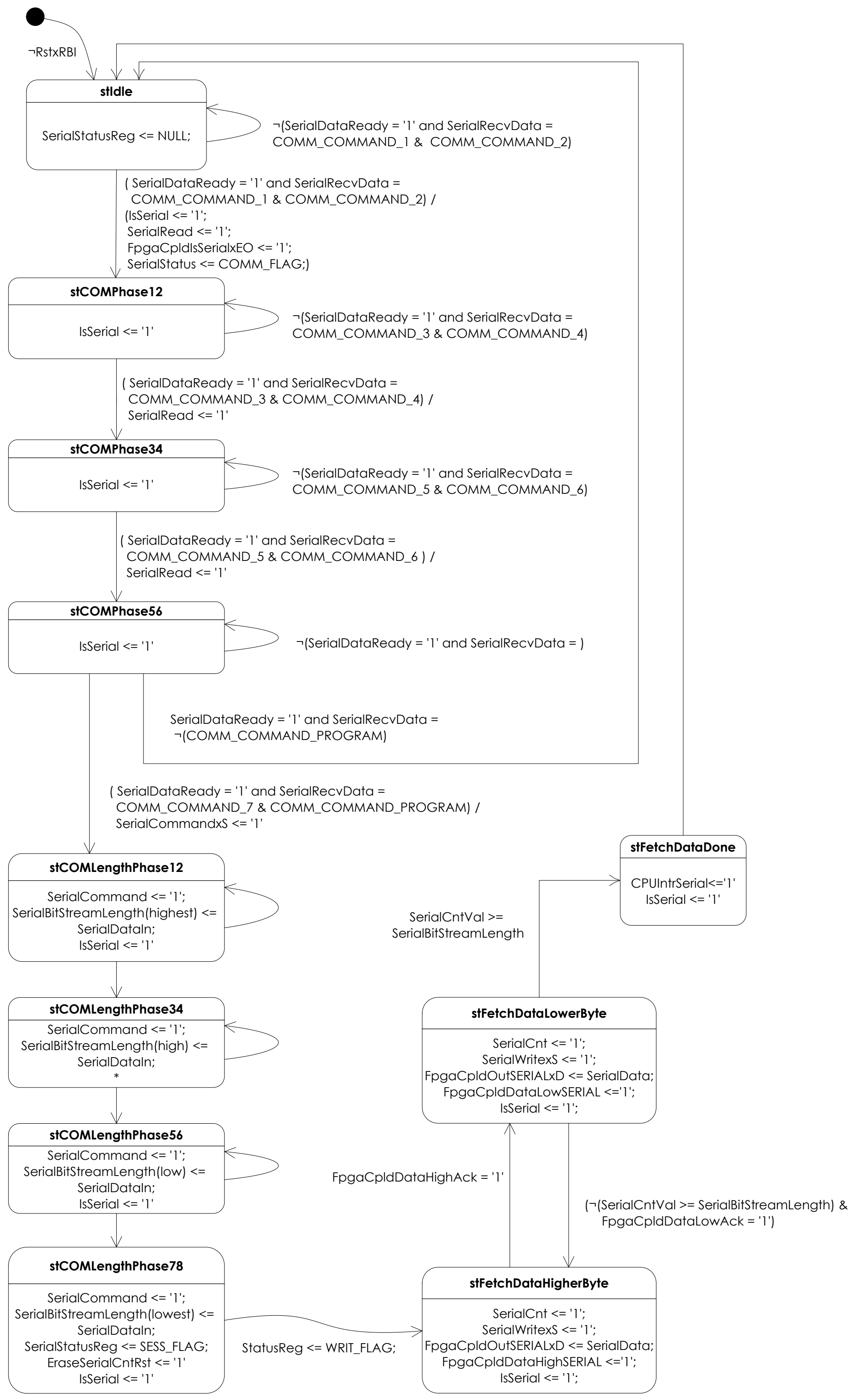

¬RstxRBI
stIdle
SerialStatusReg <= NULL;
¬(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_1 & COMM_COMMAND_2)
(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_1 & COMM_COMMAND_2) / (IsSerial <= '1'; SerialRead <= '1'; FpgaCpldIsSerialxEO <= '1'; SerialStatus <= COMM_FLAG;)
stCOMPhase12
IsSerial <= '1'
¬(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_3 & COMM_COMMAND_4)
(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_3 & COMM_COMMAND_4) / SerialRead <= '1'
stCOMPhase34
IsSerial <= '1'
¬(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_5 & COMM_COMMAND_6)
(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_5 & COMM_COMMAND_6) / SerialRead <= '1'
stCOMPhase56
IsSerial <= '1'
¬(SerialDataReady = '1' and SerialRecvData =)
SerialDataReady = '1' and SerialRecvData = ¬(COMM_COMMAND_PROGRAM)
(SerialDataReady = '1' and SerialRecvData = COMM_COMMAND_7 & COMM_COMMAND_PROGRAM) / SerialCommandxS <= '1'
stFetchDataDone
CPUIntrSerial<='1'
IsSerial <= '1'
stCOMLengthPhase12
SerialCommand <= '1'; SerialBitStreamLength(highest) <= SerialDataIn; IsSerial <= '1'
SerialCntVal >= SerialBitStreamLength
stCOMLengthPhase34
SerialCommand <= '1'; SerialBitStreamLength(high) <= SerialDataIn; *
stFetchDataLowerByte
SerialCnt <= '1'; SerialWritexS <= '1'; FpgaCpldOutSERIALxD <= SerialData; FpgaCpldDataLowSERIAL <='1'; IsSerial <= '1';
stCOMLengthPhase56
SerialCommand <= '1'; SerialBitStreamLength(low) <= SerialDataIn; IsSerial <= '1'
FpgaCpldDataHighAck = '1'
(¬(SerialCntVal >= SerialBitStreamLength) & FpgaCpldDataLowAck = '1')
stCOMLengthPhase78
SerialCommand <= '1'; SerialBitStreamLength(lowest) <= SerialDataIn; SerialStatusReg <= SESS_FLAG; EraseSerialCntRst <= '1' IsSerial <= '1'
StatusReg <= WRIT_FLAG;
stFetchDataHigherByte
SerialCnt <= '1'; SerialWritexS <= '1'; FpgaCpldOutSERIALxD <= SerialData; FpgaCpldDataHighSERIAL <='1'; IsSerial <= '1';

Figure 5-21
Loader on FPGA: FSM for the SERIAL Protocol
Analogically to the FSM receiving the IPAQ data the SERIAL FSM receives data from the implemented RS-232 interface. Different to the CPU FSM (as explained in fig. 5-18) is that the transmitted serial data are 8-bit; therefore all FSM states need a "second" state, the FSM is clocked by the board oscillator and the Mealy FSM is not needed for the communication. All control signals are generated by Moore states.

5.5 CPLD Design

A CPLD device keeps its functionality at loss of power because the configuration will be stored in internal EEPROM. Usually, CPLD are used as «glue-logic» between multiple hardware devices. In contrast to the SRAM based Xilinx FPGAs they can't be reconfigured at runtime.

This behaviour is very useful for a reconfigurable design: The CPLD contains a Final State Machine which handles the FLASH, the FPGA communication protocol and — as the main application — the reconfiguration process of the FPGA. Doing a reconfiguration causes an information loss of all internal states in the FPGA (flip-flop, BlockRAM etc.), so it isn't feasible to do the rewriting without a second «controlling device».

5.5.1 Application of the CPLD

We use the CPLD for initiating the «slave parallel» reconfiguration mode of the *Xilinx SPARTAN^{TM} II* device, generating addresses and transferring the data from the Flash memory and controlling the end of the reconfiguration process and finally the startup process.

The communication protocol between the FPGA and the CPLD is kept simple. The 16-bit bus between the devices is splitted in a 8-bit data bus and a 6-bit for control flow (1 reset, 1 new command, 4 data transfer) and 2-bit feed-through wires for the RS-232 interface.

- **New Command**: A new command (fig. 5-22-a) is initiated by a high `FpgaCpldCommandxS` signal level. At the same time the command data word is asserted to the `FpgaCpldBUSxD`.
- **Data Transfer**: The data transfer — for program a new bitstream into the Flash — is implemented as a unidirectional transfer from FPGA to CPLD.

  After we assert the new command, the data words are transferred by a «pushing protocol» using the `FpgaCpldDataHigh` and `FpgaCpldDataLow` signal (fig. 5-22-b). The data are transferred in the first phase (`FpgaCpldDataHigh` is high), in the second phase they are used for state transitions in the CPLD FSM.

  In the second implementation of the *loader* (see also sec. 5.4.4) a 16-bit data word of the IPAQ is transferred in 2 x 8-bit values per `InternalClk` cycle. In this mode the CPLD part is strictly synchronous (Moore FSM), but the FPGA part of the implementation has asynchronous transitions (Mealy FSM). To handle this asynchronous data transfer a 4 phase handshaking protocol is implemented. Additional to the `FpgaCpldDataHigh` and `FpgaCpldDataLow` signals two acknowledge wires – «ACKs» – `FpgaCpldDataHighAck` and `FpgaCpldDataLowAck` are used for transitions.
- **RS-232 Transfer**: To use the BTNodeFPGA board without having a IPAQ or to have a «fall-back mode» for debugging, the loader implementation contains

a RS-232 mode. Because the IPAQ bridge is still connected in the debug mode, the FPGA I/O pins are all taken.

As a communication channel we use the «UC»-interface of the CPLD and feed the serial signal wires through the device to the inter-device bus. The RS-232 doesn't influence the CPLD behaviour. The data transfer is displayed in the fig. 5-23.

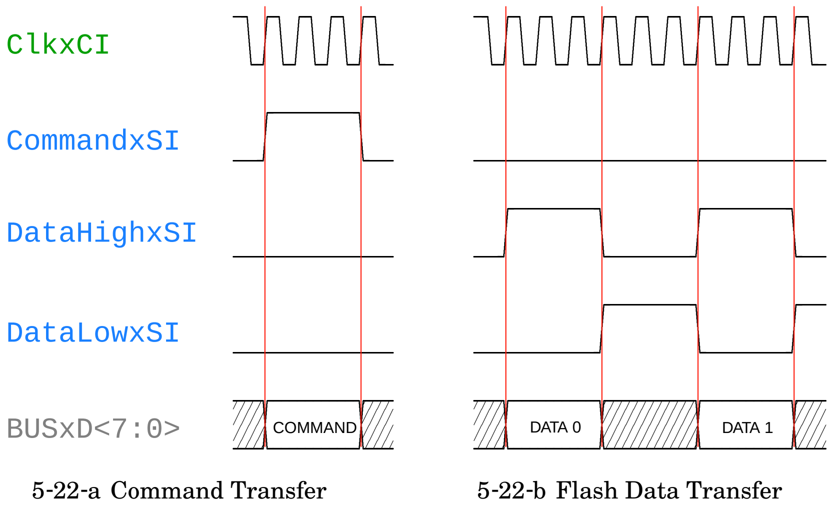


Figure 5-22
CPLD Commands: Command & Data Transfer

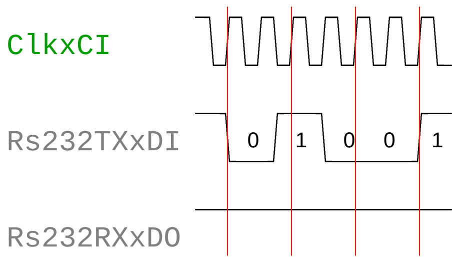


Figure 5-23
RS-232 data transfer through the CPLD:
The interface is only feed through to the inter-device bus and to the FPGA. The RS-232 data do not influence the CPLD. As shown in the figure, the serial clock is quite different from the clock oscillator (0.115 MHz versus 18.432 MHz).

5.5.2 Final States Machine

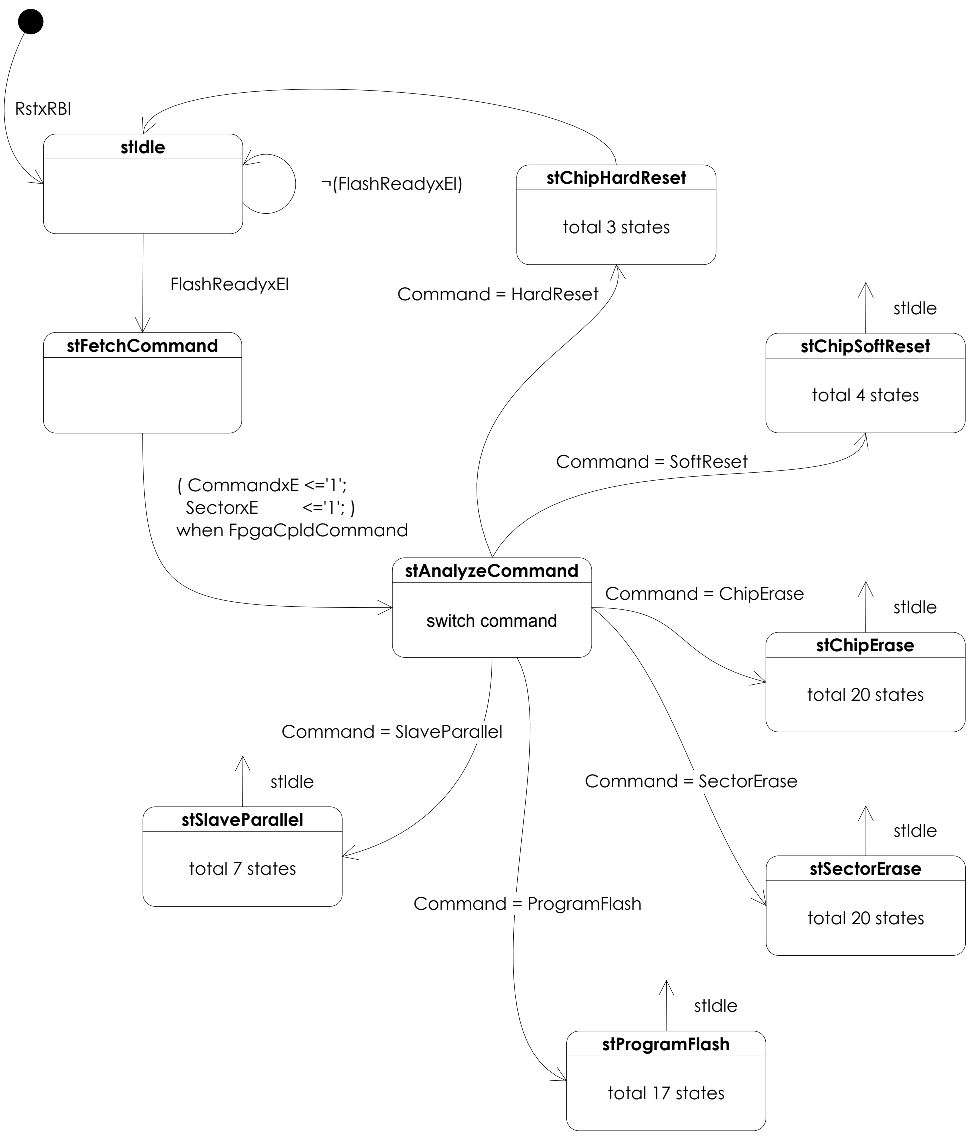


Figure 5-24
CPLD - FSM as used in the 8-bit mode.
The FSM stays in the initial Idle-state until the Flash memory on the board is in the ready state. Afterwards, the commands of the FPGA (IPAQ) or the host computer connected via RS-233 are interpreted. The FSM is listen to the inter-chip communication until a new command and sector is applied in the FetchCommand-state. The new command is analysed in the AnalyzeCommand-state which consists of a jump table.
The ChipHardReset-state and the ChipSoftReset-state reset the internal Flash Final State Machine to its initial – reading *– state. The complete Flash is erased in the ChipErase-state: The Flash memory always needs an erasing procedure — setting all bits to the high level '1' — before the memory can be programmed by discharging the desired bits to low level '0'. Erasing is also possible in sectors of 64 kBytes. This mode is performed in the SectorErase-state. The programming procedure will be done in the ProgramFlash-state – exactly, in 20 sub states looped for bitstream-length times (i.e. 166'980 times). The addresses of the Flash – with the stored bitstream – are selected by the «selected sector» command. The SlaveParallel-state initiates the reconfiguration process of the FPGA. One of the available bitstreams in the Flash memory is selected and loaded via the slave parallel mode.*

5.5.3 Synthesis & Routing Data

Data-sheet of the CPLD after Synthesis with *Xilinx XST* and fitting process *Xilinx FIT* . The implementation data include the *Sector Erase* functionality of the FSM. All asynchronous inputs are directly connected to a synchroniser as proposed in section 5.7.

- **Statemachine FlashFSM**
  - *States*: 70
  - *Transition*: 86 FIXME
  - *Inputs*: 29 FIXME
  - *Outputs*: 31 FIXME
  - *Encoding*: Gray Encoding
  - *Register*: T flip-flops
- **Tristates**
  - *FlashAddr*:
    20-bit tristate buffer
  - *FlashData*:
    8-bit tristate buffer
- **Elements**
  - *ROM (Sector)*:
    16x40-bit ROM (Sector Select)
  - *Register*:
    2x4-bit, 7-bit, 2x20-bit
  - *Comparator*:
    2x20-bit comparators
  - *Counters*:
    4-bit, 8-bit, 2x20-bit and 29-bit
  - *Flip-Flops*:
    200
  - *IO*:
    61 (+ 1 Clock Pin)
- **Macrocell**
  - *used*:
    202 / 384 (53%)
  - *product terms*:
    711 / 1344 (53%)
  - *registers used*:
    122 / 384 (32%)
  - *pins used*:
    62 / 114 (55%)
  - *inputs used*:
    642 / 960 (67%)
- **Timings**
  - *Min. Clock Period*:
    38.400 ns
  - *Max. Clock Frequency*:
    26.042 MHz
  - *Clock to Setup*:
    38.400 ns
  - *Pad to Pad Delay*:
    12.000 ns
  - *Setup to Clock at the Pad*:
    3.000 ns
  - *Clk Pad to Output Pad Delay*:
    45.800 ns

5.6 Scan Chain

This section describes an idea and implementation of saving the internal states of an user task. Similar in concept to the flip-flop scan chains for VLSI designs, the developed technique includes all or a specified selection of the flip-flops in serial chain at FPGA logic level. The scan chain can be inserted after design implementation (post-synthesis) and is in that case independent from users knowledge about the execution environment.

5.6.1 Introduction

A main problem to be solved when a task switch occurs is to retain information of the running task. The current state should be fetched and saved for a restore at the next instantiation.

There are several approaches to address this problem:

1. The «cheapest» solution of this problem can be used if the function of the task doesn't depend on the last internal states or just from the initial state. In this case the internal states of the process don't need to be saved and no further actions are needed.
2. One possible solution is that the designer knows the execution environment of the user task and explicitly uses a memory structure like a FIFO, a Block-RAM or an external memory cell for state saving and reloading. When a task switch occurs, the task stores to this memory in the «destructor». When the task is reloaded in the execution environment the states are reloaded by the «constructor». Figure 5-25 sketches such a environment.

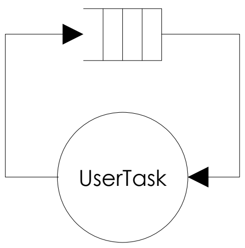


Figure 5-25
User Task saves his internal state explicitly to an external memory cell (i.e. FIFO or BlockRAM)

3. Another solution, is the implementation of a «scan chain» adapted from an idea for circuit verification in VLSI designs. The implementation of a scan chain builds a large linear shift register containing all (or selected) task flip-flops and adds a second operation mode - the shift mode - besides the first operational mode.

   The execution environment needs a memory for state saving as mentioned in the previous solution but in this proposal the scan chain can be inserted in the net-list after synthesis. The designer of the task doesn't have to know about the existence of such a state saving functionality.

A further advantage is that the memory cells could adaptively be instantiated depending on the length of the scan chain and that the shift duration can be recognised by the chain generation process.

4. Usually, a FPGA contains — like every VLSI chip — a test verification structure (like a built in self test (BIST) or scan chain) but for user application they are deactivated or not available. As an alternative some devices allow to read back the actual configuration via JTAG interface. Such a bit-stream contains initial values of LUTs, contents of flip-flops or internal RAM structures and is available at *Xilinx SPARTAN^{TM} II* [34] or at *Xilinx VIRTEX^{TM}* [35] devices. A drawback of this method is, that the read-back needs a special configuration in the design flow and that the internal states only could be analysed and stored on a computer, not in a execution environment on the FPGA.
5. Another «read back function» is the instantiation a component which allows an access to the internal configuration access port on FPGA. Such an element exists on the *Xilinx VIRTEX^{TM} II* devices (Element: ICAP_VIRTEX2). With the ICAP element it is possible to fetch the states of selected FPGA elements as done in [5].

We have chosen to implement a scan chain cause of the following advantages:

The task designer doesn't need to know about the state saving functionality; the states can be saved directly in the task execution environment (i.e. to a BlockRAM) by a FSM and no further processing on a host computer is needed. The interface is kept very easy as we will see in the next sections. The memory size and the duration of the whole task switch can be determined by a scan chain generation script.

In section 5.6.2 the original idea of the scan chain in VLSI designs is presented. A proposed paper and the differences in our design is written in section 5.6.3. The implementation of the scan chain in the *Xilinx SPARTAN^{TM} II* FPGA is shown in section 5.6.4.

5.6.2 Scan Chain in VLSI designs

In VLSI designs the common approach is to simulate the design at multiple levels for validating the hardware design (behavioural model written in a high level language, behavioural simulation of a HDL, post-synthesis simulation) because there is a very high observability and controllability. *Observability* refers to the ability to access all internal state of the circuit; *controllability* is the ability to modify the run-time state of the circuit.

After fabricating the design as a chip the mentioned advantages will be lost if no specialised test structures are foreseen (design for testability). In the ideal case the hardware execution should provide approximately the same level of observability and controllability as a software simulator. Two essential methods are known:

- *Built In Self Test (BIST)*: Test structure which tests a part of the chip itself (at startup sequence or during running state) and signals the result of this test to the system (normally used for memory cells)

- *Scan Chain*: Test structure which connects all flip-flops instances of a chip as 1 bit linear shift register for production failure (ATPG, stuck-at-tests) and functional tests (observability and controllability of internal states)

These capabilities should be provided automatically by the design tools:

For scan chain insertion the used flip-flops are replaced by their scan-able counterpart ((2:1 MUX in front of D-input) and assembled into N chains, where N is typically 2 <= N <= 64). ATPG vectors are automatically generated by improved tools able to interpret synthesised net-lists and to produce sequences of test vectors which should trigger each wire and element at least one time at low level another time at high level.

5.6.3 Scan Chain in FPGAs and Related Work

The idea of using the well-known scan chain from VLSI hardware designs to increase FPGAs observability and controllability has been investigated by Wheeler et al. in [29]:

Further the benefits and area/speed costs of a scan chain in FPGA designs are listed in the paper of Wheeler et al:

1. The FPGA does not require any special capabilities to implement design-level scan - it can be added to any user design in any FPGA.
2. The amount of data scanned out of the circuit is much smaller and easier to manipulate than for configuration read-back bit-streams, since scan bit-stream contains only the desired circuit state information.
3. Determining the position of signal values in the scan bit-stream is straightforward since it is easy to determine the order in which the memory elements are arranged in the scan chain.
4. Scan allows the state of the circuit to be set to known values.
5. Scan Chain insertion can be done automatically by script.

The biggest downside to scan is the large area and speed penalty it causes. Scan chains in VLSI designs requires an area overhead of 5-30%. The area overhead of a scan chain in FPGA designs costs additional 66-130% in area overhead on average. Adding scan logic reduces the speed of circuit by 20% on average.

Different from the proposed approach from Wheeler [29] we use the scan chain not to increase the observability or controllability, but we use the instrument of a scan chain for shift in and out the state of some flip-flops. The overhead of the chain could be reduced by only selecting the necessary flip-flops (i.e. special name given (like SaveStateFSMxDP)). For writing the user task we prefer VHDL or Verilog which are compatible with our tool-flow in contrast to the used JHDL [6] in the Wheelers paper.

5.6.4 Scan Chain Implementation

The «scan chain» is 1-bit-wide Linear Shift Register (LSR) wiring the memory elements such as flip-flops, so that the state bits contained in these elements exit the circuit serially through the Scan Out pin whenever the Scan Enable control signal is asserted. New state data for the FPGA enters the circuit serially on the Scan In pin. When Scan Enable is deasserted, the circuit returns to normal operation.

5.6.4.1 Design Primitives

FPGA flip-flops can be inserted into a scan chain by simply attaching a multiplexor (MUX) before the data input of the FF and a MUX (or logic gates) in front of the Enable pins.

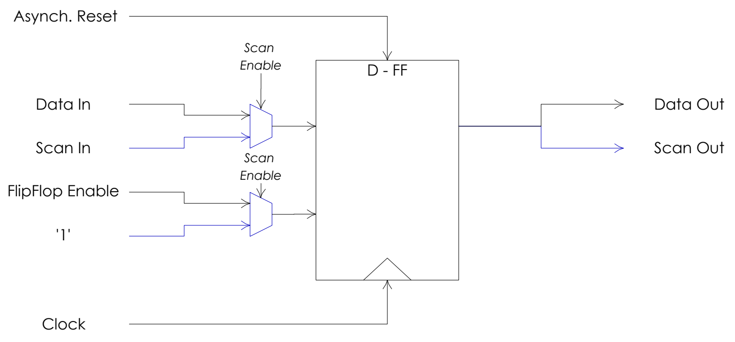


Figure 5-26
Modifying a FlipFlop for scan: Adding two MUX in front of the Data Input and the FlipFlop Enable.

The Scan In signal in the figure 5-26 is the Scan Out from the previous FF in the scan chain. The Scan Out becomes the Scan In for the next storage cell in the chain. Thus when the signal Scan Enable is asserted, the memories in the circuit form a single-bit-wide shift register; when Scan Enable is deasserted the circuit resumes normal operation. While Scan Enable is asserted, the FF must be enabled and allow its bit to be shifted out. The multiplexor and the constant *true* value in front of the Enable input serve this purpose.

In the worst case the area overhead for replacing the normal flip-flops with this scan-able FF is two MUXes (in the *Xilinx SPARTAN*^{TM} *II* architecture implemented with a 2-LUT and a 3-LUT). In many real world instances, the two MUX can be included in the (empty) LUT in front of a FF or can be merged with the combinational logic for Data Input.

5.6.5 Tool Flow

The scan chain generation is done with a tool-chain consisting of several scripts: The Xilinx synthesis tool (XST) generates a compressed net-list format usually called NGC. To modify this net-list we need to transform it to EDIF (`ngc2edif`), which is also the output of Synplicity Synplify. Afterwards we transform the EDIF net-list to an XML format for tree parsing (DOM - Tree) and modify the tree with elaborated

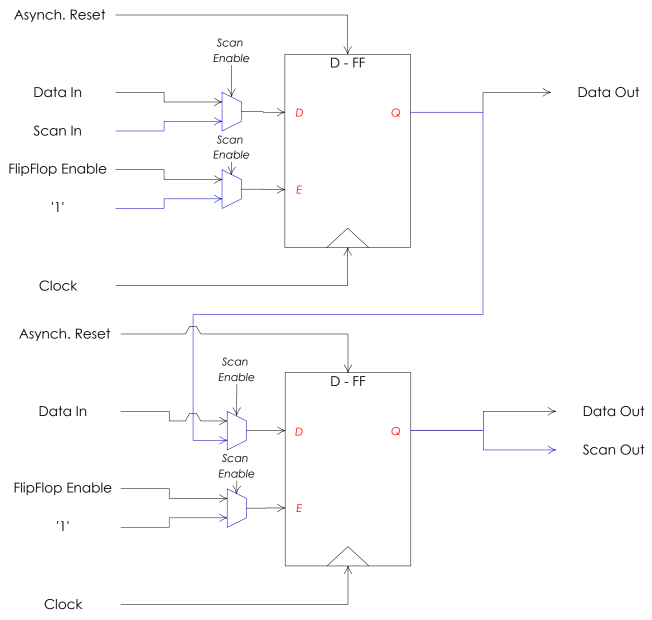


Figure 5-27
Concept: Scan chain with two connected scan-able flip flops.

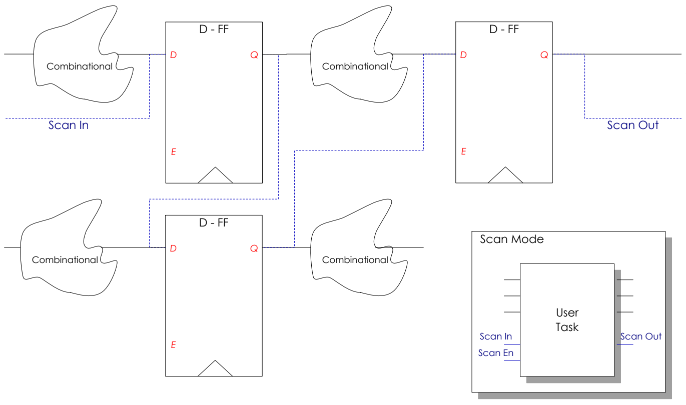


Figure 5-28
Concept: In the scan mode some flip-flops are directly connected (Output Q –> Input D), any combinational logic is missing. Often the sequence (i.e. counter) is another than in non-scan operation but the order doesn't matter in this mode.
In the User Task three additional pins are added: two Inputs Scan In *and* Scan Enable *and one Output* Scan Out.

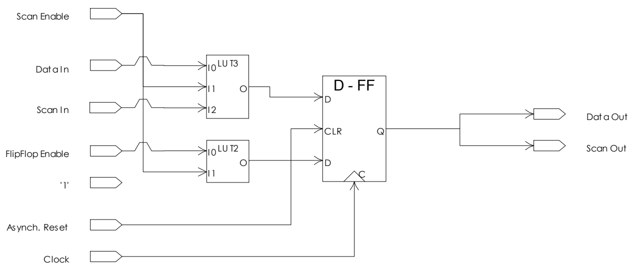


Figure 5-29
Realisation of the scan chain with a Xilinx SPARTAN^{TM} II *LUT2 and a LUT3 element.*

tools `Edif2XML`. `Modify_improved` replaces all flip flops by a version with a «enable» which is a requirement for the following scan chain insertion; the chain is inserted by `Modify_ScanChain`. The modified net-list needs to be back-transformed from the XML format to EDIF `XML2Edif` for the following translation step.

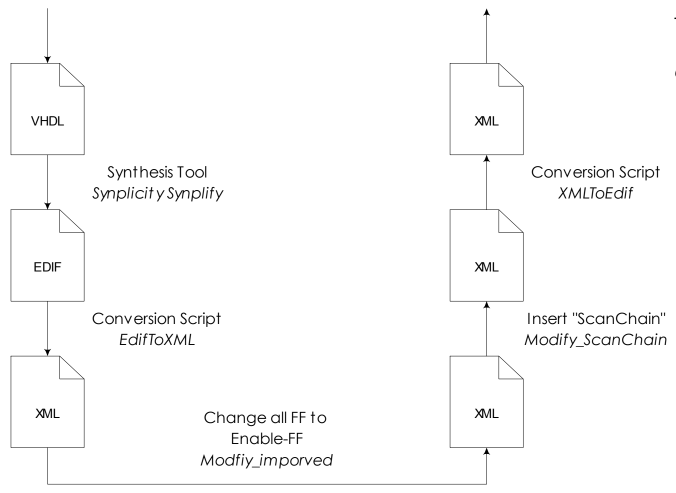


Figure 5-30
Tool-flow: Insertion of a ScanChain in an already synthesised design.

5.6.6 Conclusions

The proposed and implemented scan chains allow to store internal states of a task in an execution environment without special FPGA elements. The routing and speed overhead is not neglectable but any other method (without the explicit saving) has similar drawbacks.

5.7 Acquisition of Asynchronous Data - Handshaking

5.7.1 Acquisition of a single input signal

The final state machines of an implementation needs stable input signal during the setupTime of the state transitions. To avoid illegal states – brief glitching – of the FSM all asynchronous input signals need a synchroniser! The «synchroniser» should consist of one additional flip-flop at least. We use always a stage of 2 flip-flops as proposed in [19].

We underestimate this design rule and as a consequence we had several weeks to debug the CPLD design of the «Loader» (the FlashReady signal of the FLASH device was totally asynchronous to the CPLD clock).

5.7.2 Acquisition of vectored input signals

Multiple chip designs normally have problems with acquiring vectored data because of the data is changing in the first device during the setup time of the flip-flops in a second device [19]. One solution for data transfer between clock boundaries is to use a handshake protocol.

Handshaking excludes inconsistent data words from ever being admitted into the receiving circuit. The main idea is to avoid sampling the input while it might be changing. Instead, the updating and the sampling of the data vector get coordinated by observing a handshake protocol (with multiple FSMs) that involves both the producing and the consuming subsystems.

Full handshaking is essentially symmetrical and requires two control lines, termed **request** `REQ` and **acknowledge** `ACK`.

In the application design of the BTNODEFPGA a handshaking protocol with four-phases (return-to-zero (RZ)) is used for the communication between FPGA and CPLD. The reason for using such a handshaking protocol is the construct and generation of the «internally clock» from IPAQs **WriteEnable**, **OutputEnable** and **ChipSelect** signals. In the worst case the data (commands and bitstream data) in the `InternalClk`-Domain in the FPGA will be changing during the setupTime of the FSM in the CPLD.

As a solution a `REQ` - `ACK` protocol completed with two synchroniser is implemented in the FPGA as well as in the CPLD. The suggestion from the lecture script [19] is implemented, as can be seen in the figures 5-31, 5-32 and 5-33

5.7.3 Metastability at Clock Boundaries

Data sheets describe the logic behaviour of bistable such as flip-flops by the way of a truth table. The desired data retention of flip-flops is essentially obtained from cross-coupling two inverting amplifiers. In between the two stable equilibrium points that reflect the binary state '0' and '1' respectively, there necessarily exists a third and unstable equilibrium point. Marginal triggering implies bringing a bistable close to that point and leaving it to recover. The bistable is then said to hand in an evanescent **metastable state** as shown in figure 5-34

Solution: As a cooking receipt a design with two flip-flops in series will stop the metastability in most of the cases.

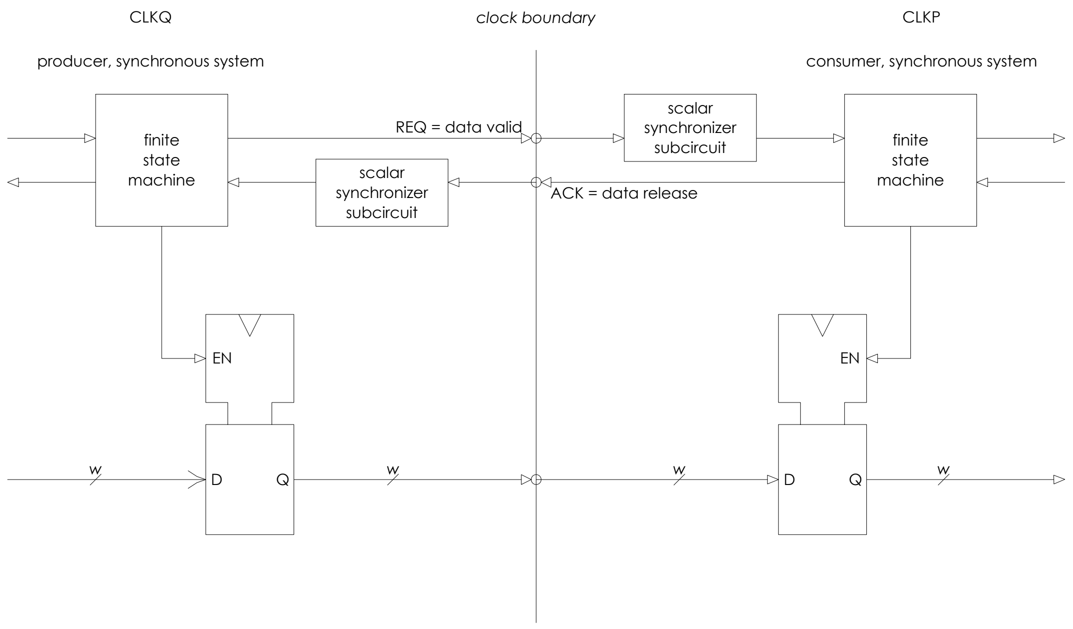

CLKQ
clock boundary
CLKP
producer, synchronous system
consumer, synchronous system
finite state machine
REQ = data valid
scalar synchronizer subcircuit
finite state machine
scalar synchronizer subcircuit
ACK = data release
EN
EN
w
D
Q
w
w
D
Q
w

Figure 5-31
Handshaking over Clock Boundaries
In the upper part the two final state machines are shown handling the handshaking protocol with their REQs and ACKs and the enable strobes of the receiving flip-flops. The bits of the data vector could be received without timing-driven changes, because the data word is valid for three clock periods at least.

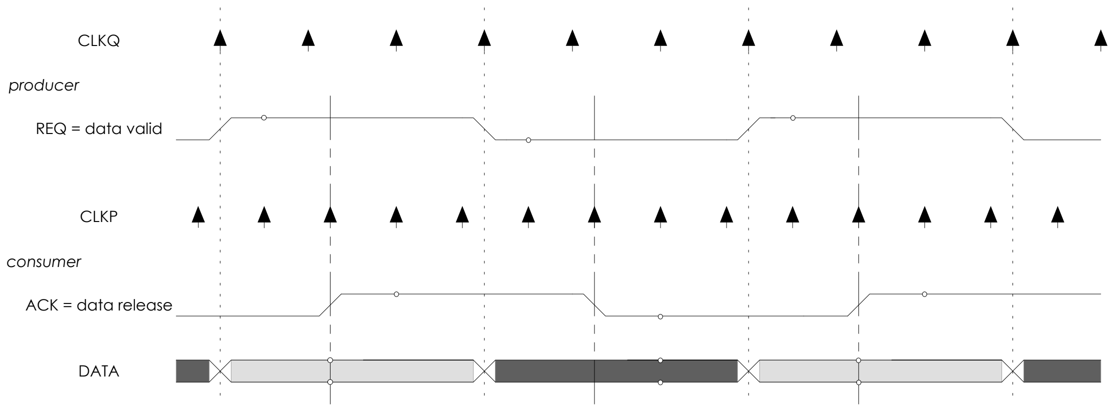
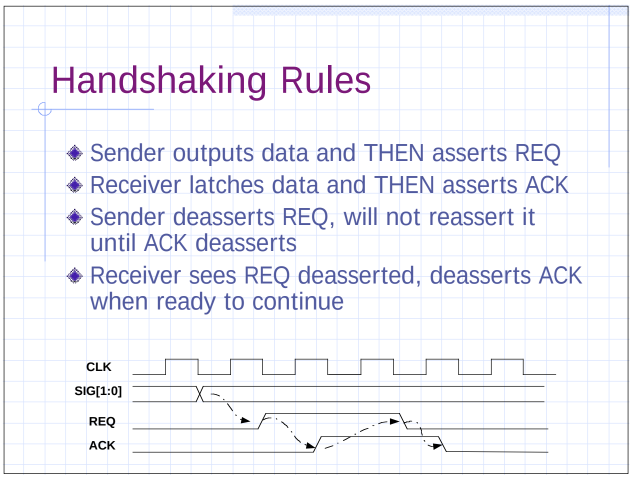


Figure 5-32
Handshaking over Clock Boundaries: Protocol
In the upper part the clock boundary of the producer, in the lower part the clock boundary of the consumer with their rising clock edges are shown. The small circle represents the point of time when the high logic level is recognised by the other device. In this four-phased scheme the transferred data are valid only in the light grey shadowed regions.



Handshaking Rules

- Sender outputs data and THEN asserts REQ
- Receiver latches data and THEN asserts ACK
- Sender deasserts REQ, will not reassert it until ACK deasserts
- Receiver sees REQ deasserted, deasserts ACK when ready to continue

CLK
SIG[1:0]
REQ
ACK

Figure 5-33
Handshaking over Clock Boundaries: Synchronisation Howto Action and Event Diagram

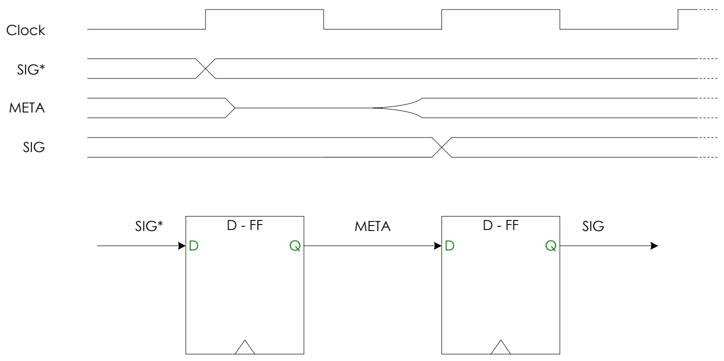


Figure 5-34
Metastability Wave in CMOS Technology and their resolution in time to the known logic level 0 or 1. SIG is the logic level before the first flip-flop, the META signal is in-between the to flip-flops and SIG is the output signal after the synchroniser.*

5.8 Kahn Task Template

The figure 5-35 and the listing 5-12 show the Task Template in VHDL. CustomLed2xSO and CustomLed3xSO can be used.

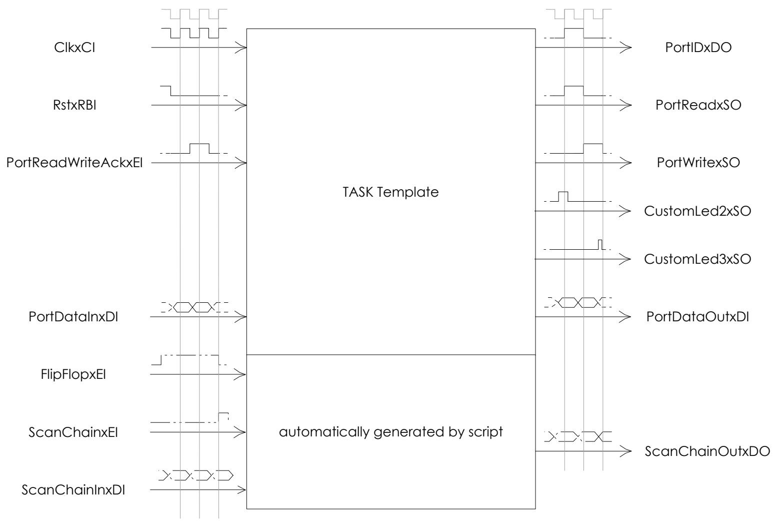


Figure 5-35
Task Template: Structural Overview
The upper part of the entity has to be implemented by the user. The lower part (enable all flip-flops and scan chain) could be generated automatically by a Perl-script *or explicitly programmed by the user.*
The signals `CustomLed2xSO` *and* `CustomLed3xSO` *are direct connected with the FPGA-Leds 2 and 3 of the* BTNODEFPGA *- Board and can be used to signal some status information to the user.*

```
entity task is
  port (
    ClkxCI               : in  std_logic;                      -- Clock
    RstxRBI              : in  std_logic;                      -- Reset
    PortIDxDO            : out PortIDType;                     -- Port ID
    PortReadxSO          : out std_logic;                      -- Request: Read from Port
    PortWritexSO         : out std_logic;                      -- Request: Write to Port
    PortReadWriteAckxEI  : in  std_logic;                      -- Ack: Read / Write Req
    PortDataInxDI        : in  std_logic_vector(15 downto 0);  -- Data In
    PortDataOutxDO       : out std_logic_vector(15 downto 0);  -- Data Out
    ScanChainInxDI       : in  std_logic;                      -- Scan Chain In
    ScanChainOutxDO      : out std_logic;                      -- Scan Chain Out
    ScanChainxEI         : in  std_logic;                      -- Scan Chain Enable
    FlipFlopxEI          : in  std_logic;                      -- General FlipFlop Enable
    CustomLed2xSO        : out std_logic;                      -- Custom LED for Task
    CustomLed3xSO        : out std_logic                       -- Custom LED for Task
   );
end task;
```

Listing 5-12
Task Entity of the VHDL Task Template

5.9 IPAQ Program

The IPAQ tool is a Windows CE program written in C++ and running in the command line modus offered by the shell. The program is the counterpart to the FPGA Loader on the BTNODEFPGA — it acts as a «Data Streamer».

The «Data Streamer» reads modified and stored bitstreams on the IPAQ file system and writes them in 16-bit data chunks to the extension bus. The bitstreams are read in the Streamer with a “looped” file read function until the end-of-file occurs.

A graphical user interface for the Streamer is not implemented so far.

The IPAQ data streamer uses a library («UserSpace Driver» developed at TIK) for communicate to the FPGA via the connected extension port.

The mentioned driver maps the expansion port (data and control flow) to a memory mapped region in the main memory and offers reading and writing function from and to this sector. The «block transfer mode» of the interface allows to transfer multiple data items with one command. Consequently, the throughput is much higher than calling the writing procedure in a loop multiple times.

For debugging reasons we use the more lax timing by calling the write function in a loop in the «Data Streamer» application.

A deeper and more detailed insight in the program is given in the doxygen output in the appendix section B.

5.10 Toolflow

5.10.1 Bitstream Generation

1. Create a new Task: Kahn Process Networkare normally used for data streaming applications and our framework is optimised for such tasks. The first step is to design a new task with such a behaviour.
2. Translate the design idea to a hardware description language. Available and used languages are VHDL or Verilog. Use the VHDL Task Template (sec. 5.8) as the top entity of the task or as a data wrapper for data transfers.
3. The scan chain can be inserted by a separate toolflow as described in section 5.6.5. Before the chain could be inserted, a netlist has to be generated by *Xilinx XST* .
4. If the task has rather hard area or timing constraints, compile the stand-alone task with *Synplicity Synplify Pro* as described in section 6.4.
5. Connect the user task with the task wrapper of to the slotmanager. Keep caution on the reset polarity and the reset pin configuration – the reset button is feed-through from CPLD in the current design.

   For insert the task in the framework, the following methods are possible:

   - Integrate the adapted VHDL Task Template in the Project Navigator.
   - A Verilog File with the same connectivity as the VHDL template could be inserted in the Project Navigator.
   - Integration of a EDIF netlist: The task as a EDIF netlist will be integrated in the ISE flow, but the netlist will not appear in the Project Navigator. For the integration with the current tools, it is necessary that the EDIF netlist is located in the same directory as the Project file (`.npl`). The EDIF netlist will be integrated in the translate-step.

6. Activate the option `set unused pins to: float` in the ISE FPGA implementation flow:

   In the properties of `Generate Programming File` there is a «tab» with `Configuration Options`. The last but one option `Unused IOB Pins` change the option from `Pull Down` to `float`.

   This is necessary to donŠt affect the FLASH address and data bus by unused FPGA pins.
7. Run the implementation flow for FPGA configurations: Synthesis, Translate, Map, Place & Route, Generate Bit File
8. The bitstream can be uploaded to the BTNodeFPGA board by the JTAG boundary-scan interface of *Xilinx Impact* , but usually we like to reconfigure the BTNodeFPGA board by the IPAQ.
9. Modify the generated bitstream (`.bit` file) with bistreambuilder.pl to obtain a programmable form for our framework with and on the IPAQ. The operation modes of the program is explained in the appendix C.1 on page 148.
10. Upload the modified bitstream to the IPAQ with the Microsoft Active Sync Tool. The default directory for the Windows CE program and new bitstreams is the folder «bitstreams».
11. The Windows CE Program «DataStreamWriter.exe» can read and stream-out modified bitstreams. In the current implementation the names of the bitstreams are hard-coded as could be seen in sec. B — a GUI for our Streamer on the IPAQ would be nice.

5.10.2 Modifying of Bitstream

In brief, a short overview how a FPGA configuration bitstream is modified.

```
 Original Bitstream    Modified Bitstream
    .. .. .. ..
    .. .. .. ..           CA FE AF 5x        Additional Synchronisation Word
    .. .. .. ..           xx xx xx xx        Length of Bitstream (Bytes)
    FF FF FF FF           FF FF FF FF        Synchronisation Word 1
    AA 99 55 66           AA 99 55 66        Synchronisation Word 2
                          00 00 00 00        Additional Synchronization Word
    30 00 80 01           30 00 80 01        First Real Data
    .. .. .. ..           .. .. .. ..
    30 00 00 01           30 00 00 01        Write To CRC
    xx xx xx xx           xx xx xx xx        CRC Value
    .. .. .. ..           .. .. .. ..
    30 00 80 01           30 00 80 01        Write next 4 Bytes to CMD Register
    00 00 00 05           00 00 00 05        Begin Start-Up Sequence
    .. .. .. ..           .. .. .. ..
    30 00 00 01           30 00 00 01        Final Write To CRC
    xx xx xx xx           xx xx xx xx        Final CRC Value
    00 00 00 00           00 00 00 00        16 zero Bytes for Slave Parallel
    00 00 00 00           00 00 00 00
    00 00 00 00           00 00 00 00
    00 00 00 00           00 00 00 00
```

Listing 5-13
Modified Bitstreams

6

Application and Performance

6.1 Possible Applications

In generally, Kahn Process Network are well suited for *streaming* and *signal processing applications*. Examples of these classes are audio- and video processing operations (like encoding, decoding or applying of filters), cryptography (like symetrical encryption of a data stream) or dataflow operations in generally (like packing or subdividing of a datastream into packets).

Our platform with the implemented «Kahn Hardware Task» execution unit is proper for high performance computation task which will calculate the all-up function faster in hardware than in software. As an example we would like give an short overview of the AES encryption:

> Data will be encrypted in several computation «rounds» which mostly consists of several multiplications and feed-backs of the current «states». General Purpose Processors usally have one or two multiplication units with several cycles of latency in the most cases.
>
> By implementing AES in hardware several hardware multiplier could be instanciated in parallel. The state feed-back could be implemented by a feed-back path. If in the hardware implementation is piplelined in each clock cycle a new data token could be encrypted.

Other examples in the class «faster in hardware» would be:

- symetrical encryption: DES, AES
- check-sums: CRC32, MD5
- image processing: filters like erosion, dilution

Additionally, there exist a plenty of applications having computation steps which could be runned in parallel – and therefore also faster executed on a FPGA.

- bit-pattern matching (image processing, processing of internet data streams)
- filters (FIR-Filter, gauss filter for vision)

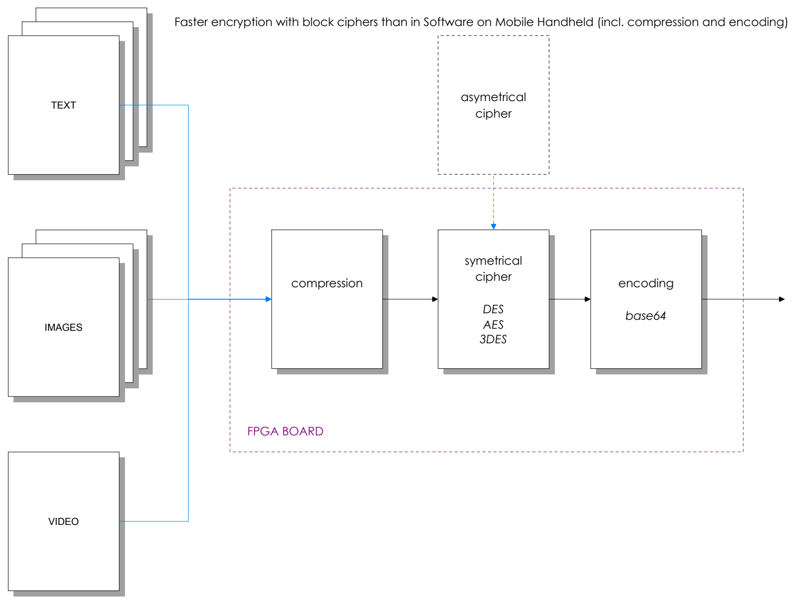

Faster encryption with block ciphers than in Software on Mobile Handheld (incl. compression and encoding)
TEXT
IMAGES
VIDEO
asymetrical cipher
compression
symetrical cipher
DES
AES
3DES
encoding
base64
FPGA BOARD

Figure 6-1
Application: Cipher encoding in HW

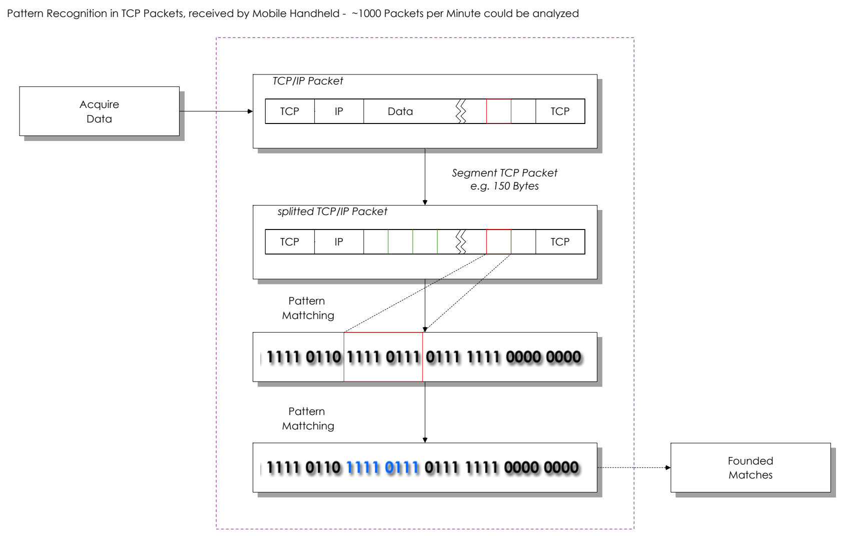


Figure 6-2
Application: Pattern recognition in Mobile NetFlow Data

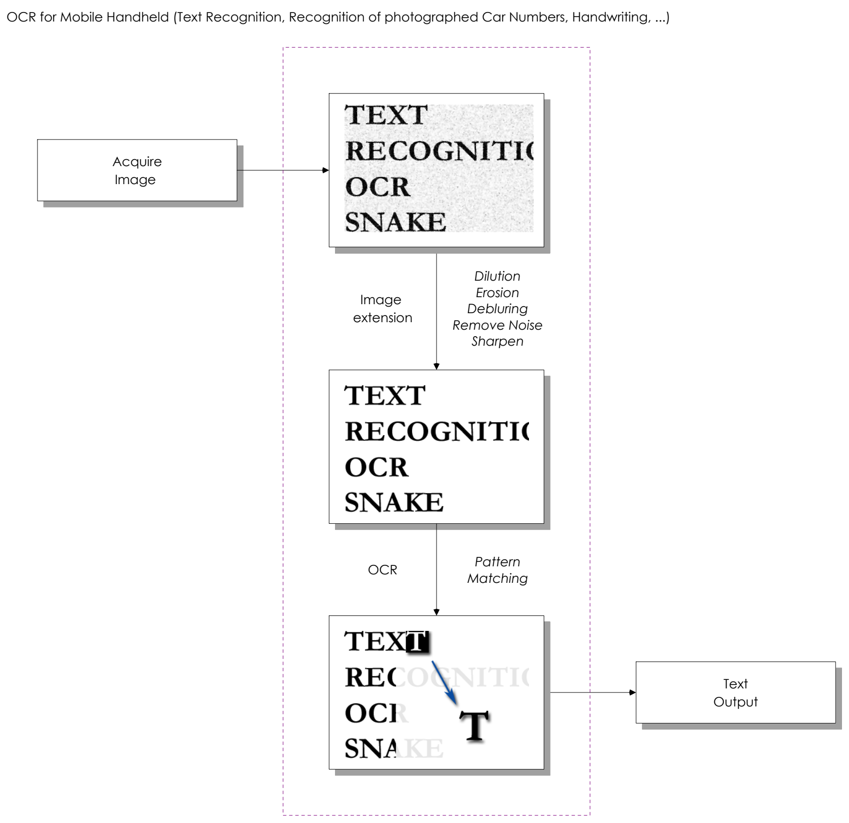

OCR for Mobile Handheld (Text Recognition, Recognition of photographed Car Numbers, Handwriting, ...)
Acquire Image
TEXT RECOGNITIO OCR SNAKE
Image extension
Dilution
Erosion
Debluring
Remove Noise
Sharpen
TEXT RECOGNITIO OCR SNAKE
OCR
Pattern Matching
TEXT RECOGNITIO OCR SNAKE
T
Text Output

Figure 6-3
Application: Pattern recognition on scanned or photographed images

6.2 Example Cores inclusive Performance

6.2.1 AES Core - encoding and decoding

Design	http://www.opencores.org/cores/aes_core/ http://www.asics.ws
Author	Rudolf Usselmann, rudi@asics.ws
Language	
of Design	Verilog
of Wrapper	VHDL
Testbench	included with reasonable test vectors
Docu	included with overview, block schema, timing diagrams

Table 6-1: AES Core Datas

6.2.1.1 Behaviour

This implementation is with a 128 bit key expansion module only.

AES

The AES cipher core consists of a key expansion module, an initial permutation module, a round permutation module and a final permutation module. The round permutation module will loop internally to perform 10 iteration (for 128 bit keys).

Inverse AES

The AES inverse cipher core consists of a key expansion module, a key reversal buffer, an initial permutation module, a round permutation module and a final permutation module.

The key reversal buffer first stores keys for all rounds and then presents them in reverse order to the inverse cipher rounds.

The round permutation module will loop internally to perform 10 iteration (for 128 bit keys).

6.2.1.2 Throughput

The forward cipher block can perform a complete encrypt sequence in 12 clock cycles (10 cycles for the 10 rounds, plus one cycle for initial key expansion, and one cycle for the output stage).

The inverse cipher block can perform a complete decrypt sequence in 12 cycles 10 cycles for the 10 rounds, plus one cycle for initial key loading, and one cycle for the output stage).

The inverse cipher, however, requires that the key is loaded before decryption can be performed. This is because it uses the last expanded key first and the first expanded key last. Once the key has been loaded, the expanded versions are generated and stored in an internal buffer.

Additional 8 cylces ($\frac{128Bit}{16Bit}$) are necessary cause of input and output datawidth of the Wrapper.

$$12\ Cycles\ (AES) + 2 * 8\ Cyles\ (Input, Output) = 28\ Cylces$$

With a used clock of 18.432 MHz:

$$18.432.000\ Hz * 2\ Bytes/28 = 1.316.000\ MByte/s$$

Communication-Overhead: Communication / Algorithm

$$16\ Cylces/12\ Cycles = 4/3 = 1.333(!)$$

6.2.1.3 Encoder

synthesis done with Synplify Pro

```
Frequency, Slack:
=================

                      Requested   Estimated   Requested   Estimated
Starting Clock        Frequency   Frequency   Period      Period      Slack
---------------------------------------------------------------------------
slotmanager|ClkxCI    19.0 MHz    34.4 MHz    52.632      29.069      23.563
InternalClk           19.0 MHz    33.7 MHz    52.632      29.712      22.919
===========================================================================


Elements
========

Register bits not including I/Os:   1338 (38%)

Internal tri-state buffer usage summary
BUFTs + BUFEs: 256 of 1728 (14%)

RAM/ROM usage summary
Dual Port Rams (RAM16X1D_1): 10
32x1 ROMs (ROM32X1): 488
Block Rams : 12 of 14 ( 86%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 3321 (96%)
```

map done with Xilinx map

```
Logic Utilization:
  Number of Slice Flip Flops:     1,338 out of  4,704   28%
  Number of 4 input LUTs:         2,251 out of  4,704   47%
Logic Distribution:
    Number of occupied Slices:                      2,026 out of  2,352   86%
    Number of Slices containing only related logic: 2,026 out of  2,026  100%
    Number of Slices containing unrelated logic:        0 out of  2,026    0%
Total Number 4 input LUTs:        3,288 out of  4,704   69%
      Number used as logic:                     2,251
      Number used as a route-thru:                 41
      Number used for Dual Port RAMs:              20
      (Two LUTs used per Dual Port RAM)
      Number used as 16x1 ROMs:                   976
   Number of bonded IOBs:            83 out of    140   59%
   Number of Tbufs:                 256 out of  2,464   10%
   Number of Block RAMs:             14 out of     14  100%
   Number of GCLKs:                   2 out of      4   50%
   Number of GCLKIOBs:                1 out of      4   25%

Total equivalent gate count for design:  290,454
Additional JTAG gate count for IOBs:       4,032
```

place and route done with Xilinx par

```
Device utilization summary:

   Number of External GCLKIOBs         1 out of 4      25%
   Number of External IOBs            83 out of 140    59%
      Number of LOCed External IOBs   83 out of 83    100%

   Number of BLOCKRAMs                14 out of 14    100%
   Number of SLICEs                 2026 out of 2352   86%

   Number of GCLKs                     2 out of 4      50%
   Number of TBUFs                   256 out of 2464   10%

+-------------------------+----------+--------+-------------+-------------+
|        Clock Net        | Resource | Fanout |Net Skew(ns) |Max Delay(ns)|
+-------------------------+----------+--------+-------------+-------------+
|         ClkxCI_c        |  Global  |  783   |  0.512      |  0.774      |
+-------------------------+----------+--------+-------------+-------------+
|cpucommunication_1/      |          |        |             |             |
|      internalclkxc      |  Global  |   52   |  0.501      |  0.763      |
+-------------------------+----------+--------+-------------+-------------+

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        2.966
   The MAXIMUM PIN DELAY IS:                              10.397
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   8.761

   Listing Pin Delays by value: (nsec)

    d < 2.00   < d < 4.00  < d < 6.00  < d < 8.00  < d < 11.00  d >= 11.00
   ---------   ---------   ---------   ---------   ---------    ---------
        5088        6474        3193         858          87            0

---------------------------------------------------------------------------------
  Constraint                                  | Requested   | Actual      | Logic
                                              |             |             | Levels
---------------------------------------------------------------------------------
  NET "ClkxCI_ibuf/IBUFG" PERIOD =  54.253    | 54.253ns    | 36.119ns    | 10
  nS   HIGH 50.000000 %                       |             |             |
---------------------------------------------------------------------------------
  NET "CPUOutputxEBI" PERIOD =  400 nS    LO  | N/A         | N/A         | N/A
  W 120 nS                                      |              |              |
-------------------------------------------------------------------------------------
  NET "CPUWritexEBI" PERIOD =  400 nS   LOW     | N/A          | N/A          | N/A
   120 nS                                       |              |              |
-------------------------------------------------------------------------------------
  TS_ClkxCI = PERIOD TIMEGRP "ClkxCI"  52.6     | N/A          | N/A          | N/A
  32 nS   HIGH 50.000000 %                      |              |              |
-------------------------------------------------------------------------------------
  OFFSET = OUT 52.632 nS  AFTER COMP "ClkxC     | 52.632ns     | 34.226ns     | 14
  I"                                            |              |              |
-------------------------------------------------------------------------------------
  OFFSET = IN 52.632 nS  BEFORE COMP "ClkxC     | 52.632ns     | 13.173ns     | 2
  I"                                            |              |              |
-------------------------------------------------------------------------------------
```

6.2.1.4 Decoder

synthesis done with Synplify Pro

```
Frequency, Slack:
=================

                      Requested    Estimated    Requested    Estimated
Starting Clock        Frequency    Frequency    Period       Period       Slack
--------------------------------------------------------------------------------
slotmanager|ClkxCI    19.0 MHz     33.3 MHz     52.632       30.043       22.588
InternalClk           19.0 MHz     35.6 MHz     52.632       28.104       24.527
================================================================================

Elements
========
I/O Register bits:                 0
Register bits not including I/Os:  1475 (42%)

Internal tri-state buffer usage summary
BUFTs + BUFEs: 256 of 1728 (14%)

RAM/ROM usage summary
Dual Port Rams (RAM16X1D): 128
Dual Port Rams (RAM16X1D_1): 10
32x1 ROMs (ROM32X1): 488
Block Rams : 12 of 12 (100%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 3947 (114%)
```

map done with Xilinx map

```
Logic Utilization:
  Number of Slice Flip Flops:      1,475 out of  4,704   31%
  Number of 4 input LUTs:          2,626 out of  4,704   55%
Logic Distribution:
    Number of occupied Slices:                        2,350 out of  2,352   99%
    Number of Slices containing only related logic:   2,341 out of  2,350   99%
    Number of Slices containing unrelated logic:          9 out of  2,350    1%
Total Number 4 input LUTs:         3,919 out of  4,704   83%
      Number used as logic:                      2,626
      Number used as a route-thru:                  41
      Number used for Dual Port RAMs:              276
      (Two LUTs used per Dual Port RAM)
      Number used as 16x1 ROMs:                    976
   Number of bonded IOBs:             83 out of    140   59%
   Number of Tbufs:                  256 out of  2,464   10%
   Number of Block RAMs:              14 out of     14  100%
   Number of GCLKs:                    2 out of      4   50%
   Number of GCLKIOBs:                 1 out of      4   25%

Total equivalent gate count for design:  310,178
Additional JTAG gate count for IOBs:  4,032
```

place and route done with Xilinx par

```
Device utilization summary:

   Number of External GCLKIOBs         1 out of 4      25%
   Number of External IOBs            83 out of 140    59%
      Number of LOCed External IOBs   83 out of 83    100%

   Number of BLOCKRAMs                14 out of 14    100%
   Number of SLICEs                 2350 out of 2352   99%

   Number of GCLKs                     2 out of 4      50%
   Number of TBUFs                   256 out of 2464   10%

+-----------------------------+----------+---------+------------+-------------+
|          Clock Net          | Resource | Fanout  |Net Skew(ns)|Max Delay(ns)|
+-----------------------------+----------+---------+------------+-------------+
|cpucommunication_1/inter     |          |         |            |             |
|                  nalclkxc   |  Global  |   51    |  0.501     |  0.763      |
+-----------------------------+----------+---------+------------+-------------+
|          ClkxCI_c           |  Global  |  904    |  0.507     |  0.769      |
+-----------------------------+----------+---------+------------+-------------+

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        3.127
   The MAXIMUM PIN DELAY IS:                              10.488
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   9.339

   Listing Pin Delays by value: (nsec)

    d < 2.00   < d < 4.00  < d < 6.00  < d < 8.00  < d < 11.00  d >= 11.00
   ---------   ---------   ---------   ---------   ---------   ---------
        4944        8154        4521         868          91           0

--------------------------------------------------------------------------------
  Constraint                                | Requested    | Actual       | Logic
                                            |              |              | Levels
--------------------------------------------------------------------------------
  NET "ClkxCI_ibuf/IBUFG" PERIOD =  54.253  | 54.253ns     | 39.304ns     | 9
  nS   HIGH 50.000000 %                     |              |              |
--------------------------------------------------------------------------------
  NET "CPUOutputxEBI" PERIOD =  400 nS   LO | N/A          | N/A          | N/A
  W 120 nS                                  |              |              |
--------------------------------------------------------------------------------
  NET "CPUWritexEBI" PERIOD =  400 nS   LOW | N/A          | N/A          | N/A
   120 nS                                   |              |              |
--------------------------------------------------------------------------------
  TS_ClkxCI = PERIOD TIMEGRP "ClkxCI"   52.6 | N/A         | N/A          | N/A
  32 nS   HIGH 50.000000 %                  |              |              |
--------------------------------------------------------------------------------
  OFFSET = OUT 52.632 nS  AFTER COMP "ClkxC | 52.632ns     | 38.859ns     | 8
  I"                                        |              |              |
--------------------------------------------------------------------------------
  OFFSET = IN 52.632 nS  BEFORE COMP "ClkxC | 52.632ns     | 13.790ns     | 2
  I"                                        |              |              |
--------------------------------------------------------------------------------
```

6.2.1.5 Encoder: Implementaition results of available synthesis tools

	1	2	3	4	5	6
Logic Utilization:						
- Number of Slice Flip Flops:	1,338	28%	1,457		out of	4,704
- Number of 4 input LUTs:	2,251	47%	4,923	104%	out of	4,704
Logic Distribution:						
- Number of occupied Slices:	2,026	86%	2,818	119%	out of	2,352
- Number of Slices cont. only related logic:	2,026	100%	2,776	98%	out of	2,026
- Number of Slices cont. unrelated logic:	0	0%	41	1%	out of	2,026
Total Number 4 input LUTs:	3,288	69%	4,974	105%	out of	4,704
- Number used as logic:	2,251		4,923			
- Number used as a route-thru:	41		31			
- Number used for Dual Port RAMs:	20		20			
- (Two LUTs used per Dual Port RAM)						
- Number used as 16x1 ROMs:	976					
Number of bonded IOBs:	83	59%	83	59%	out of	140
- IOB Flip Flops:			16			
Number of Tbufs:	256	10%	18	1%	out of	2,464
Number of Block RAMs:	14	100%	14	100%	out of	14
Number of GCLKs:	2	50%	2	50%	out of	4
Number of GCLKIOBs:	1	25%	1	25%	out of	4
Clock	34.4	MHz	25.2	MHz		
Clock (Internal Generated Clock (IPAQ))	33.7	MHz	?	MHz		
Minimum input arrival time before clock	?	?	8.7	ns		
Maximum output required time after clock	23.7	ns	40.6	ns		
Maximum combinational path delay:	?	?	17.0	ns		

1 & 2: Synthesis Tool A (1 are absolute, 2 relative values)
3 & 4: Synthesis Tool B (3 are absolute, 4 relative values)
5 & 6: FPGA device specific

Table 6-2: Results after synthesis with available synthesis tools and after place & route

6.2.2 DES Core - only encoding

Encoding with DES (64 Bit)

Design	`http://www.yordas.demon.co.uk/crypto/`
Author	Chris Eilbeck, chris@yordas.demon.co.uk
Language of Design of Wrapper	 VHDL VHDL
Testbench	adapted with reasonable test vectors
Docu	Brief Description with some numbers (implementation done on an older Xilinx FPGA)

Table 6-3: DES Core Datas

6.2.2.1 Behaviour

Implementaion of a DES Core (an encoder using the ECB mode) in VHDL.

2 Versions:

1. pipelined without Xilinx Elements (pipelined-des)
2. pipelined with Xilinx Elements used (optimised-des)

6.2.2.2 Throughput

$$1\ Cylce\ (DES) + 2 * 4\ Cycles\ (Input, Output) = 9\ Cycles$$

With a used clock of 18.432 MHz:

$$18.432.000\ Hz * 2\ Bytes/9 = 4.096.000\ MByte/s$$

Communication-Overhead: Communication / Algorithm

$$8\ Cylces/1\ Cycles = 8/1 = 8(!)$$

6.2.2.3 Pipelined Version

synthesis done with Synplify Pro

```
Frequency, Slack:
=================

                      Requested    Estimated    Requested    Estimated
Starting Clock        Frequency    Frequency    Period       Period       Slack
--------------------------------------------------------------------------------
slotmanager|ClkxCI    19.0 MHz     33.5 MHz     52.632       29.876       22.755
InternalClk           19.0 MHz     34.1 MHz     52.632       29.366       23.266
================================================================================

Elements:
=========

I/O Register bits:                 0
Register bits not including I/Os:  1265 (36%)

RAM/ROM usage summary
Dual Port Rams (RAM16X1D_1): 10
32x1 ROMs (ROM32X1): 928
Block Rams : 12 of 12 (100%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 3974 (114%)
```

map done with Xilinx map

```
Logic Utilization:
  Number of Slice Flip Flops:      1,265 out of  4,704   26%
  Number of 4 input LUTs:          2,033 out of  4,704   43%
Logic Distribution:
    Number of occupied Slices:                          2,335 out of  2,352   99%
    Number of Slices containing only related logic:     2,335 out of  2,335  100%
    Number of Slices containing unrelated logic:            0 out of  2,335    0%
Total Number 4 input LUTs:         3,950 out of  4,704   83%
      Number used as logic:                     2,033
      Number used as a route-thru:                 41
      Number used for Dual Port RAMs:              20
      (Two LUTs used per Dual Port RAM)
      Number used as 16x1 ROMs:                 1,856
   Number of bonded IOBs:             83 out of    140   59%
   Number of Block RAMs:              14 out of     14  100%
   Number of GCLKs:                    2 out of      4   50%
   Number of GCLKIOBs:                 1 out of      4   25%

Total equivalent gate count for design:  316,866
Additional JTAG gate count for IOBs:  4,032
```

place and route done with Xilinx par

```
Device utilization summary:

   Number of External GCLKIOBs          1 out of 4       25%
   Number of External IOBs             83 out of 140     59%
      Number of LOCed External IOBs    83 out of 83     100%

   Number of BLOCKRAMs                 14 out of 14     100%
   Number of SLICEs                  2335 out of 2352    99%

   Number of GCLKs                      2 out of 4       50%

+------------------------------+----------+--------+-------------+-------------+
|          Clock Net          | Resource  | Fanout  |Net Skew(ns)|Max Delay(ns)|
+-----------------------------+-----------+---------+------------+-------------+
|          ClkxCI_c           |  Global   |  693    |  0.501     |  0.763      |
+-----------------------------+-----------+---------+------------+-------------+
|cpucommunication_1/inter     |           |         |            |             |
|                  nalclkxc   |  Global   |   50    |  0.500     |  0.762      |
+-----------------------------+-----------+---------+------------+-------------+

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        3.027
   The MAXIMUM PIN DELAY IS:                               10.620
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:    9.384

   Listing Pin Delays by value: (nsec)

    d < 2.00   < d < 4.00  < d < 6.00  < d < 8.00  < d < 11.00  d >= 11.00
   ---------   ---------   ---------   ---------   ---------   ---------
        5101        8065        3995         611         119           0

------------------------------------------------------------------------------------
  Constraint                                  | Requested   | Actual      | Logic
                                              |             |             | Levels
------------------------------------------------------------------------------------
  NET "ClkxCI_ibuf/IBUFG" PERIOD =   54.253   | 54.253ns    | 44.573ns    | 9
  nS   HIGH 50.000000 %                       |             |             |
------------------------------------------------------------------------------------
  NET "CPUOutputxEBI" PERIOD =  400 nS    LO  | N/A         | N/A         | N/A
  W 120 nS                                    |             |             |
------------------------------------------------------------------------------------
  NET "CPUWritexEBI" PERIOD =  400 nS   LOW   | N/A         | N/A         | N/A
   120 nS                                     |             |             |
------------------------------------------------------------------------------------
  TS_ClkxCI = PERIOD TIMEGRP "ClkxCI"   52.6  | N/A         | N/A         | N/A
  32 nS   HIGH 50.000000 %                    |             |             |
------------------------------------------------------------------------------------
  OFFSET = OUT 52.632 nS  AFTER COMP "ClkxC   | 52.632ns    | 37.585ns    | 9
  I"                                          |             |             |
------------------------------------------------------------------------------------
  OFFSET = IN 52.632 nS  BEFORE COMP "ClkxC   | 52.632ns    | 14.237ns    | 2
  I"                                          |             |             |
------------------------------------------------------------------------------------
```

6.2.2.4 Pipelined and Optimized Version with XILINX instances

synthesis done by Synplify Pro

```
Frequency, Slack:
=================

                      Requested    Estimated    Requested    Estimated
Starting Clock        Frequency    Frequency    Period       Period       Slack
-------------------------------------------------------------------------------
slotmanager|ClkxCI    19.0 MHz     34.8 MHz     52.632       28.729       23.903
InternalClk           19.0 MHz     37.2 MHz     52.632       26.847       25.785
===============================================================================

Elements:
=========

I/O Register bits:                 0
Register bits not including I/Os:  1312 (37%)

RAM/ROM usage summary
```

```
Dual Port Rams (RAM16X1D_1): 10
32x1 ROMs (ROM32X1): 1024

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 4209 (121%)
```

mapping done by Xilinx map

```
Logic Utilization:
  Number of Slice Flip Flops:      1,312 out of  4,704   27%
  Number of 4 input LUTs:          1,991 out of  4,704   42%
Logic Distribution:
    Number of occupied Slices:                      2,350 out of  2,352   99%
    Number of Slices containing only related logic:  1,252 out of  2,350   53%
    Number of Slices containing unrelated logic:     1,098 out of  2,350   46%
Total Number 4 input LUTs:         4,281 out of  4,704   91%
      Number used as logic:                     1,991
      Number used as a route-thru:                222
      Number used for Dual Port RAMs:              20
      (Two LUTs used per Dual Port RAM)
      Number used as 16x1 ROMs:                 2,048
   Number of bonded IOBs:             83 out of    140   59%
   Number of Block RAMs:               2 out of     14   14%
   Number of GCLKs:                    2 out of      4   50%
   Number of GCLKIOBs:                 1 out of      4   25%

Total equivalent gate count for design:  128,344
Additional JTAG gate count for IOBs:  4,032
```

place and route done with Xilinx par

```
Device utilization summary:

   Number of External GCLKIOBs          1 out of 4      25%
   Number of External IOBs             83 out of 140    59%
      Number of LOCed External IOBs    83 out of 83    100%

   Number of BLOCKRAMs                  2 out of 14     14%
   Number of SLICEs                  2350 out of 2352   99%

   Number of GCLKs                      2 out of 4      50%

+----------------------------+----------+---------+-------------+-------------+
|         Clock Net          | Resource | Fanout  |Net Skew(ns) |Max Delay(ns)|
+----------------------------+----------+---------+-------------+-------------+
|          ClkxCI_c          |  Global  | 1216    |  0.512      |  0.782      |
+----------------------------+----------+---------+-------------+-------------+
|cpucommunication_1/inter    |          |         |             |             |
|                   nalclkxc |  Global  |   63    |  0.497      |  0.761      |
+----------------------------+----------+---------+-------------+-------------+

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:          2.688
   The MAXIMUM PIN DELAY IS:                                 7.928
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:     7.074

   Listing Pin Delays by value: (nsec)

    d < 1.00    < d < 2.00   < d < 3.00   < d < 4.00   < d < 8.00   d >= 8.00
   ----------   ----------   ----------   ----------   ----------   ----------
         2551         4476         6218         4277         3674            0

--------------------------------------------------------------------------------
  Constraint                                   | Requested  | Actual     | Logic
```

```
                                                     |               |               | Levels
-----------------------------------------------------------------------------------------------
 NET "ClkxCI_ibuf/IBUFG" PERIOD =  54.253            | 54.253ns      | 40.167ns      | 9
 nS   HIGH 50.000000 %                               |               |               |
-----------------------------------------------------------------------------------------------
 NET "CPUOutputxEBI" PERIOD =  400 nS   LO           | N/A           | N/A           | N/A
 W 120 nS                                            |               |               |
-----------------------------------------------------------------------------------------------
 NET "CPUWritexEBI" PERIOD =  400 nS   LOW           | N/A           | N/A           | N/A
  120 nS                                             |               |               |
-----------------------------------------------------------------------------------------------
 TS_ClkxCI = PERIOD TIMEGRP "ClkxCI"  52.6           | N/A           | N/A           | N/A
 32 nS   HIGH 50.000000 %                            |               |               |
-----------------------------------------------------------------------------------------------
 OFFSET = OUT 52.632 nS  AFTER COMP "ClkxC           | 52.632ns      | 35.164ns      | 8
 I"                                                  |               |               |
-----------------------------------------------------------------------------------------------
 OFFSET = IN 52.632 nS  BEFORE COMP "ClkxC           | 52.632ns      | 15.301ns      | 2
 I"                                                  |               |               |
-----------------------------------------------------------------------------------------------
```

6.2.3 CRC32 Core

CRC32 Checksum

Design	http://www.xilinx.com/bvdocs/appnotes/xapp209.pdf ftp://ftp.xilinx.com/pub/applications/xapp/xapp209.zip
Author	Chris Borrelli, Xilinx,
Language	
of Design	Verilog
of Wrapper	VHDL
of Script	Perl (Generates Polynomial Coeffients)
Testbench	none (correct implementation?)
Docu	XAPP: * Introduction to CRC * Polynoms (sample, standard polynoms) * LFSR * Block Diagrams * Explanation of perl script * Timing Diagrams

Table 6-4: CRC32 Core Datas

6.2.3.1 Behaviour

Script can produce a lot of different versions. Only 2 versions are used/wrapped:

2 Versions:

1. CRC16_8
2. CRC_CCIT_8

Wrapper
Internal Clock-Multiplyer (Clk2X) is used, cause the input datawidth is only 8 Bit, and 16 Bits are available from the KPN Wrapper. Using internally the doubled clock frequency the two bits could be used in one external cycle....

Phases:

Input Length of Datastream (wish to receive CRC32)

Input Data

Output After <Length> Cycles the CRC32 value is written to the output

6.2.3.2 Throughput

$$0 Cylces\ (CRC) + 1\ Cycles\ (Input) = 1\ Cycle$$

- CRC is computed parallel to data input
- 1 Cycle is used for length of data acquisition
- 1 Cycle is used for data output

With a used clock of 18.432 MHz:

$$18.432.000\ Hz * 2\ Bytes/1 = 18.432.000\ MByte/s$$

Communication-Overhead: Communication / Algorithm

$$1\ Cylces/1\ Cycles = 1 = 1(!)$$

6.2.3.3 CRC16_8

synthesis done with Xilinx xst

```
Frequency, Slack:
=================

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 66.862ns (Maximum Frequency: 14.956MHz)
   Minimum input arrival time before clock: 8.736ns
   Maximum output required time after clock: 40.568ns
   Maximum combinational path delay: 17.013ns

Elements:
=========

Macro Statistics :
```

```
# Registers                          : 68
#       1-bit register               : 9
#       10-bit register              : 1
#       16-bit register              : 4
#       18-bit register              : 33
#       2-bit register               : 16
#       3-bit register               : 1
#       8-bit register               : 4
# Counters                           : 2
#       12-bit up counter            : 1
#       8-bit up counter             : 1
# Multiplexers                       : 30
#       1-bit 8-to-1 multiplexer     : 2
#       18-bit 8-to-1 multiplexer    : 4
#       2-bit 8-to-1 multiplexer     : 1
#       2-to-1 multiplexer           : 23
# Tristates                          : 3
#       16-bit tristate buffer       : 2
#       18-bit tristate buffer       : 1
# Adders/Subtractors                 : 6
#       16-bit adder                 : 1
#       18-bit adder                 : 3
#       18-bit subtractor            : 1
#       8-bit adder                  : 1
# Comparators                        : 5
#       16-bit comparator less       : 1
#       18-bit comparator equal      : 3
#       8-bit comparator equal       : 1
# Xors                               : 4
#       1-bit xor3                   : 1
#       1-bit xor7                   : 1
#       1-bit xor8                   : 1
#       1-bit xor9                   : 1

 Number of Slices:                  710  out of   2352    30%
 Number of Slice Flip Flops:        793  out of   4704    16%
 Number of 4 input LUTs:           1199  out of   4704    25%
 Number of bonded IOBs:              84  out of    144    58%
 Number of TBUFs:                    18  out of   2352     0%
 Number of BRAMs:                     2  out of     14    14%
 Number of GCLKs:                     3  out of      4    75%
```

map done with Xilinx map

```
Logic Utilization:
  Number of Slice Flip Flops:       777 out of  4,704   16%
  Number of 4 input LUTs:         1,143 out of  4,704   24%
Logic Distribution:
    Number of occupied Slices:                          885 out of  2,352   37%
    Number of Slices containing only related logic:     885 out of    885  100%
    Number of Slices containing unrelated logic:          0 out of    885    0%
Total Number 4 input LUTs:        1,205 out of  4,704   25%
      Number used as logic:                     1,143
      Number used as a route-thru:                 42
      Number used for Dual Port RAMs:              20
      (Two LUTs used per Dual Port RAM)
   Number of bonded IOBs:            83 out of    140   59%
      IOB Flip Flops:                              16
   Number of Tbufs:                  18 out of  2,464    1%
   Number of Block RAMs:              2 out of     14   14%
   Number of GCLKs:                   3 out of      4   75%
   Number of GCLKIOBs:                1 out of      4   25%
   Number of DLLs:                    1 out of      4   25%
```

place and route done with Xilinx par

```
Device utilization summary:
   Number of External GCLKIOBs         1 out of 4       25%
   Number of External IOBs            83 out of 140     59%
      Number of LOCed External IOBs   83 out of 83     100%

   Number of BLOCKRAMs                 2 out of 14      14%
   Number of SLICEs                  885 out of 2352    37%

   Number of DLLs                      1 out of 4       25%
   Number of GCLKs                     3 out of 4       75%
   Number of TBUFs                    18 out of 2464     1%

+-----------------------------+----------+--------+------------+-------------+
|          Clock Net          | Resource | Fanout |Net Skew(ns)|Max Delay(ns)|
+-----------------------------+----------+--------+------------+-------------+
|cpucommunication_1_Inter     |          |        |            |             |
|                 nalClkxCO   |  Global  |   62   |  0.493     |  0.794      |
+-----------------------------+----------+--------+------------+-------------+
|task_wrapper_1_task_1_cr     |          |        |            |             |
|  c16_8_wrapper_1_Clk2xC     |  Global  |   33   |  0.117     |  0.759      |
+-----------------------------+----------+--------+------------+-------------+
|        ClkxCI_IBUFG         |   Local  |  371   |  1.823     |  5.423      |
+-----------------------------+----------+--------+------------+-------------+

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        2.480
   The MAXIMUM PIN DELAY IS:                              10.901
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   7.536

   Listing Pin Delays by value: (nsec)

    d < 2.00   < d < 4.00  < d < 6.00  < d < 8.00  < d < 11.00  d >= 11.00
   ---------   ---------   ---------   ---------   ---------   ---------
        3156        2124         822         275          47           0
```

6.2.3.4 CRC_CCIT_8

synthesis done with Xilinx xst

```
Frequency, Slack:
=================

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 66.862ns (Maximum Frequency: 14.956MHz)
   Minimum input arrival time before clock: 8.736ns
   Maximum output required time after clock: 40.568ns
   Maximum combinational path delay: 17.013ns

Elements:
=========

Macro Statistics :
# Registers                       : 68
#      1-bit register             : 9
#      10-bit register            : 1
#      16-bit register            : 4
#      18-bit register            : 33
#      2-bit register             : 16
#      3-bit register             : 1
#      8-bit register             : 4
# Counters                         : 2
#       12-bit up counter          : 1
#       8-bit up counter           : 1
# Multiplexers                     : 30
#       1-bit 8-to-1 multiplexer   : 2
#       18-bit 8-to-1 multiplexer  : 4
#       2-bit 8-to-1 multiplexer   : 1
#       2-to-1 multiplexer         : 23
# Tristates                        : 3
#       16-bit tristate buffer     : 2
#       18-bit tristate buffer     : 1
# Adders/Subtractors               : 6
#       16-bit adder               : 1
#       18-bit adder               : 3
#       18-bit subtractor          : 1
#       8-bit adder                : 1
# Comparators                      : 5
#       16-bit comparator less     : 1
#       18-bit comparator equal    : 3
#       8-bit comparator equal     : 1
# Xors                             : 4
#       1-bit xor3                 : 1
#       1-bit xor7                 : 1
#       1-bit xor8                 : 1
#       1-bit xor9                 : 1

Cell Usage :
# BELS                             : 1697
#       GND                        : 3
#       LUT1                       : 46
#       LUT2                       : 110
#       LUT3                       : 506
#       LUT4                       : 526
#       MUXCY                      : 154
#       MUXF5                      : 167
#       MUXF6                      : 76
#       VCC                        : 1
#       XORCY                      : 108
# FlipFlops/Latches                : 793
#       FDC                        : 96
#       FDC_1                      : 1
#       FDCE                       : 670
#       FDCPE                      : 20
#       FDP                        : 6
# RAMS                             : 13
#       RAM16X1D                   : 11
#       RAMB4_S16                  : 1
#       RAMB4_S1_S16               : 1
# Tri-States                       : 18
#       BUFT                       : 18
# Clock Buffers                    : 3
#       BUFG                       : 3
# IO Buffers                       : 84
#       IBUF                       : 11
#       IBUFG                      : 1
#       IOBUF                      : 32
#       OBUF                       : 40
# DLLs                             : 1
#       CLKDLL                     : 1

 Number of Slices:                  710  out of   2352    30%
 Number of Slice Flip Flops:        793  out of   4704    16%
 Number of 4 input LUTs:           1199  out of   4704    25%
 Number of bonded IOBs:              84  out of    144    58%
 Number of TBUFs:                    18  out of   2352     0%
 Number of BRAMs:                     2  out of     14    14%
 Number of GCLKs:                        3  out of      4     75%
```

map done with Xilinx map

```
Logic Utilization:
  Number of Slice Flip Flops:       777 out of  4,704   16%
  Number of 4 input LUTs:         1,141 out of  4,704   24%
Logic Distribution:
    Number of occupied Slices:                         884 out of  2,352   37%
    Number of Slices containing only related logic:    884 out of    884  100%
    Number of Slices containing unrelated logic:         0 out of    884    0%
Total Number 4 input LUTs:        1,203 out of  4,704   25%
      Number used as logic:                     1,141
      Number used as a route-thru:                 42
      Number used for Dual Port RAMs:              20
      (Two LUTs used per Dual Port RAM)
   Number of bonded IOBs:            83 out of    140   59%
      IOB Flip Flops:                              16
   Number of Tbufs:                  18 out of  2,464    1%
   Number of Block RAMs:              2 out of     14   14%
   Number of GCLKs:                   3 out of      4   75%
   Number of GCLKIOBs:                1 out of      4   25%
   Number of DLLs:                    1 out of      4   25%

Total equivalent gate count for design:  55,900
Additional JTAG gate count for IOBs:  4,032
```

place and route done with Xilinx par

```
Device utilization summary:

   Number of External GCLKIOBs          1 out of 4       25%
   Number of External IOBs             83 out of 140     59%
      Number of LOCed External IOBs    83 out of 83     100%

   Number of BLOCKRAMs                  2 out of 14      14%
   Number of SLICEs                   884 out of 2352    37%

   Number of DLLs                       1 out of 4       25%
   Number of GCLKs                      3 out of 4       75%
   Number of TBUFs                     18 out of 2464     1%

+-----------------------------+----------+--------+-------------+-------------+
|          Clock Net          | Resource | Fanout |Net Skew(ns)|Max Delay(ns)|
+-----------------------------+----------+--------+-------------+-------------+
|cpucommunication_1_Inter     |          |        |             |             |
|                 nalClkxCO   |  Global  |   62   |  0.493      |  0.793      |
+-----------------------------+----------+--------+-------------+-------------+
|task_wrapper_1_task_1_cr     |          |        |             |             |
|c_ccit_8_wrapper_1_Clk2x     |          |        |             |             |
|                         C   |  Global  |   36   |  0.107      |  0.763      |
+-----------------------------+----------+--------+-------------+-------------+
|       ClkxCI_IBUFG          |  Local   |  371   |  2.357      |  6.367      |
+-----------------------------+----------+--------+-------------+-------------+

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        2.557
   The MAXIMUM PIN DELAY IS:                              10.385
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   7.473

   Listing Pin Delays by value: (nsec)

    d < 2.00    < d < 4.00   < d < 6.00   < d < 8.00   < d < 11.00   d >= 11.00
   ---------   ---------    ---------    ---------    ---------     ---------
        3118        2164          813          210          118            0
```

6.3 Performances

6.3.1 Reconfiguration-Performance

Clock	18 MHz	max. 50 MHz[a]
Loader (FPGA): used slices	267 / 2352	11 %
Reconfiguration times:		
ERASING of FLASH	11 s	
FPGA → FLASH	7.8 s	min. 1.6 s[b]
FLASH → FPGA	9 ms	

Table 6-5: Performance of the Loader and for Reconfiguration

[a] maximum configuration clock of the FPGA is specified with 50 MHz

[b] minimal configuration time of FLASH memory is: 166980 * 0.9 μs – bitstream length * physical program delay

6.3.2 Slotmanagager-Performance

Clock	18.432 MHz	
Slotmanager: used slices	838 / 2352	35 %
Slotmanager: used flip-flops	736	
Slot size: in available slices	1514 / 2352	65 %
Slot size: in available BRAMs	12 / 14	85 %

Table 6-6: Performance of Slotmanger on BTNODEFPGA-*Board*

6.3.3 ScanChain-Performance

Clock	18.432 MHz
Maximal Length	4.096 Items
Scan Mode Duration (n items)	n clock cycles

Table 6-7: Performance of the Scan Chain

6.3.4 Cores-Performance

- Reconfiguration Time
- Initalize
- Performances of the Cores

Clock	18.432 MHz
AES (encryption)	
• Throughput	1.316 MB/s
• Cylcles per Token	12 cycles
• Communication-Overhead	16 cycles
AES (decryption)	
• Throughput	1.316 MB/s
• Cylcles per Token	12 cycles
• Communication-Overhead	16 cycles
DES (encryption - pipelined)	
• Throughput	4.096 MB/s
• Cylcles per Token	12 cycles
• Communication-Overhead	16 cycles
CRC32 (Checksum)	
• Throughput	18.432 MB/s
• Cylcles per Token	1 cycles
• Communication-Overhead	1 cycles

Table 6-8: Performance of Example Cores for Slotmanger

6.4 Synthesis Tutorial for the Demo Tasks

How to synthesize a (big) core inluding the slot and task manager for the Spartan II XCS-200 device:

1. Start Xilinx ISE, build new project and add hardware description (VHDL, Verilog) and constraint (UCF) files.
2. Synthesise with the synthesis tool (Xilinx XST or Synplicity Synplify Pro). If the core is rather big or uses 14 of 14 BlockRAMs, in the later processing problems will occour (i.e. in the mapping step). For this reason we are doing the synthesis for a smaller device (xc2s150), which is approximately the same (same chip process, internals, but only 12 BlockRAMS and ....)
3. Change the Device in Project Properties from xc2s200 to xc2s150 (right mouse ckick on the the first item in the Source Window)
4. Synthesise the design with this chip constraints
5. If synthesis succeds, save the ISE Project and close the project.
6. Use an editor to modify the Xilinx ISE Project File (.npl): Replace the line `DEVICE xc2s150` to `DEVICE xc2s200`
7. Reopen the ISE Project file in ISE
8. Use the Implement Design switch or manually select Translate, Map, Place and Route

7 Status and Future Work

7.1 Status of the Project Tasks

System At first we give a short digest of the current system status:

- Our mobile handheld is extended with a computation node based on a FPGA board.
- A reconfiguration of the extension – activated from a user program on the IPAQ – is well-working.
- The «Slotmanager» is our proof-of-concept for the execution of a Kahn Task on such a computation node.

IPAQ We have developed a Windows CE program which acts as «reconfigurator», i.e. the program allows a user or a user program to (a) erase the FLASH to clean-up all bitstreams, (b) program a new bitstream to the FLASH, (c) activate the reconfiguration mode of the FPGA and configure the device with a bitstream, (d) do these steps for multiple bitstreams and sectors in the FLASH (up to 5 full-bitstreams can be stored in the memory).

IPAQ → FPGA The communication interface between IPAQ and our framework is working and has been well tested. Two different communication protocols are implemented (*Slotmanager* and *Loader*), but only the *Loader* protocol is integrated in a software environment and tested.

A «virtual machine» and a «scheduler» are needed to integrate the *Slotmanager* protocol in a IPAQ application. A first approach has been done in a parallel semester term project.

An improvement of the communication bottleneck (IPAQ → FPGA) could be achieved by using the «PC Card» mode of the expansion slot instead of the used «asynchronous SRAM» interface.

FPGA Two FPGA designs are developed - the «Loader» and the «Slotmanager».

The CPU Interface (i.e. IPAQ part) of the Loader has been tested and is full functional working. The serial part of the Loader simulates accurately but is not tested (s.a. Tools → Streamer).

The Slotmanager has been simulated and partially tested on our platform but the final tests and a debugging phase have not taking place until now.

FPGA → CPLD We have implemented two communication protocols between the FPGA and the CPLD as described in sections 5.4.3 and 5.4.4.

The 8 bit mode is debugged and working, the 16 bit mode needs some fixes (after the insertion of synchroniser as described in section 5.7.1 the design should work!).

CPLD The FLASH command «SECTOR ERASE» is included in a branch of the CPLD design. With this additional states fitting problems occurs which should be resolved by using a *Synplicity Synplify Pro* synthesis and an improved area/timing driven fit process.

CPLD → FLASH An improved programming process is proposed in the following section 7.2.1.

Reconfiguration Implementation works as designed. No improvements are outstanding.

Cores 5 functional cores are adapted (wrapped) to run as a Kahn process. AES, DES and CRC32 are open-source cores and explained in section 6.2. The adpcm-core was developed by Matthias Dyer and illustrated in his master thesis. The md5 core is erroneous and no further used.

Tools

Streamer/Listener This Perl program for sending and receiving data over a serial RS-232 connection has been tested to work well between 2 PCs. Problems (too long «high phases» between the data words) are encountered by using the streamer with the FPGA board, the two designs (rs-232 core on FPGA and software serial streamer) seem not work together.

7.1.1 Survey: Actual State of Project Tasks

A tabularly survey of the actual state:

Tasks	Status
Windows CE Program	
DataStreamWriter	7
Cores of Kahn Tasks	
AES	6
DES	6
CRC32	6
ADPCM	6
MD5	3
Slotmanager	
CPU Protocol	3
AGU (Address Generation Unit)	6
Saving of Task States	4-6
Loader	
CPU Protocol	7
Serial Protocol	5
IPAQ Streamer	7
IPAQ / CPLD Communication	7
FLASH ERASE	7
FLASH PROGRAM	7
Reconfiguration of FPGA	7
Reconf. initated from IPAQ	7
Flash Simulation Model	3

Table 7-1: Survey: Actual state of Project Tasks

1. Idea
2. Concept Done
3. Behavioural Implementation in VHDL
4. Tested Behavioural Implementation (Modelsim)
5. Synthesised
6. Place & Route Done
7. Working on or with BTNODEFPGA-Board

7.2 Future Work

Ideas In this section we like to give an overview for future work:

- Extend the «one-kahn-task-slot-system» to a multiple task slot system.
- Improve the reconfiguration process with the partial reconfiguration idea (smaller partial bitstreams, reconfiguration during another task is executed in another task slot concurrently).
- Integration of the Slotmanger and the Loader in an application («Scheduler» & «Virtual Machine») running on IPAQs Windows CE.
- Workings in the field of scheduling of bounded Kahn Process Networks (scheduling and mathematical analysis).

7.2.1 Improved FLASH Programming Procedure

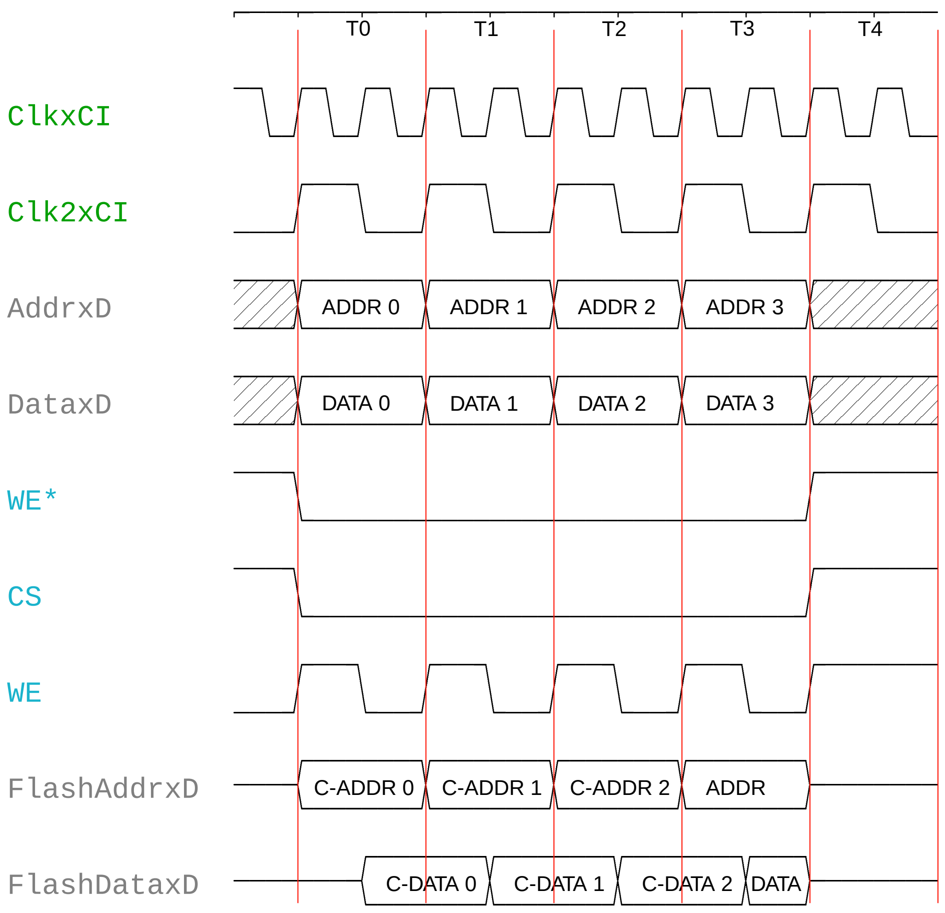
At the moment the FLASH programming procedure needs 4 x 3 states (plus some additional for control logic) to write a single byte from the CPLD to the non-volatile memory. The three state for each transfer are: *address valid*, *data valid & write enable* and *write enable de-asserted*.

These 12 states transitions could be improved in a 4 state scheme as proposed in the timing diagram 7-1. Keep in mind that each FLASH program procedure needs a delay of 9 μs at the new-byte-assertion until the internal FSM of the FLASH has written the data to the memory storage.

A code-snipplet in VHDL for this FSM improvement is listed is 7-14.

```
Sequential: process (ClkxCI, RstxRI)
begin  -- process Sequential
  if RstxRI = '0' then                     -- asynchronous reset (active high)
    FSMxDP <= stInit;
  elsif ClkxCI'event and ClkxCI = '1' then   -- rising clock edge
    Clk2xC <= not Clk2xC;
    if Clk2xC = '0' then
      FSMxDP <= FSMxDN;
    else
      FlashDataxD <= DataxD;
    end if;
  end if;
end process Sequential;

Comb: process (FSMxDP)
begin  -- process Comb
  case FSMxDP) is
    when stInit =>
      --
    when stWrite =>
      --
  end case;
end process Comb;
```

Listing 7-14
VHDL for writing to the FLASH



C-ADDR:	Address word for triggering the FLASH FSM
C-DATA :	Data word for triggering the FLASH FSM

Figure 7-1
Waveforms for writing to the FLASH

7.2.2 Scan-Chain Script Improvements

In the VHDL *synthesis step* allocated or instanciated flip-flop will be mapped to design primitive elements which are available in the *Xilinx SPARTAN^{TM} II* device. These design primitives will be *mapped* at least to 12 different configuration for the flip-flops in the FPGA.

The current implementation of the «scan-chain» insertion script considers only two possible flip-flop configurations. An improved scheme is listed in the right column of the table.

7-1-a Implemented Replacement

FlipFlop	**Replacement FlipFlop**
FDC	FDCE
FDCE	FDCE

7-1-b Improved Replacement

FlipFlop	Replacement FlipFlop
FDR	FDRE
FDRE	FDRE
FDS	FDRSE
FDRS	FDRSE
FDRSE	FDRSE
FD	FDCE
FDC	FDCE
FDE	FDCE
FDCE	FDCE
FDP	FDCPE
FDPE	FDCPE
FDCPE	FDCPE

Table 7-2: «scan-able» flip-flop equivalents
FFs and their «scan-able» equivalent which are implemented in the current script. In the right column an improved replacement scheme is listed.

A

Appendix

A.1 Development Environment

This section does offer an overview of the software being used to develop the system. To generate the software and the bitstreams to be downloaded to the BTNODEFPGA, the following software tools have been used:

Project Navigator / ISE 6.1 The ISE [36] is a tool suite used to synthesize and implement designs for the *Xilinx* FPGAs and CPLDs.

We have used this tool suite to implement small designs and also to debug our designs. The *Xilinx XST* generates much more output informations in the Project Navigator at synthesis time that the other synthesis alternatives. The Project Navigator is a GUI that merges the various command line tools of the ISE into one front end.

Synplify Pro 7.3.3 Synplicitys Synplify Pro is a synthesis tool which translates VHDL and Verilog projects to a device (FPGA, CPLD) netlist. The result of Synplify Pro is in the most cases faster and smaller than the result of the *Xilinx XST* synthesis tool. Synplify has a graphical user interface for the visualization of the translated netlist, a timing analyzer and additional improved tools like the FSM explorer (graph, states, transitions).

HexCmp 2.3 Fairdells HexCmp is a program, which combines together the features of a binary file comparison application and a convenient hex editor. HexCmp was used to compare binary files (bitstreams) with the aid of color highlighting and synchronous scrolling in the panels.

In addition to these commercial tool suites, a number of freely available programs have been used. To modify bitstreams, for the rs-232 streamer and scan-chain insertion *Perl* has been used. *XML* and the *libxml* have been used to insert the scan chain in the EDIF netlist. The editor of choice was the almighty *Emacs*.

A.2 VHDL Coding Guidelines

For VHDL coding we have used the coding guidelines of the Microelectronics Design Center of ETH Zürich [10]. As a recapitulation we like to give a short overview over the main rules:

Constant Names

- Use *upper-case* letters and "_" only (e.g., `WIDTH`, `RAM_DEPTH`, `LFSR_INIT`).
- Avoid "_" in *generics* (synthesis attaches generic names to other names with "_" as delimiter).

Signal Names

- Start with an *upper-case* letter.
- Have a *suffix* with syntax "x[CRESDTAZ][IO]?B?" ("[...]" denotes a choice, "?" means optional).
- The suffix part "[CRESDTAZ]" indicates the class of the signal:

Class	Char	Example	Description
clock	C	`ClkxC`	clock
reset	R	`RstxRB`	asynchronous reset
enable	E	`LoadCntxE`, `StartCtrlxE`	trigger some synchronous event
control/status	S	`SelInputxS`, `FullxS`	static control signals, status signals
data/address	D	`SamplexD`, `RamAdrxD`	data and address signals
test	T	`ScanEnxT`, `RamIsolxT`	test signals
asynchronous	A	`StrobexA`	asynchronous signals
three-state	Z	`ExternBusxZ`	three-state bus signals

Table A-1: Coding Style for VHDL as proposed by the DZ

- The suffix part "[IO]?" indicates *input* and *output* signals of an entity (e.g., `CoeffxDI`, `FullxSO`)
- The suffix part "B?" indicates active *low* signals.

Variable Names

- Start with a *lower-case* letter (e.g., temp, i, currentState).
- Have *no suffix* (as opposed to signal names).

Type Names

- Have a *suffix* "Type" or a name that implies a type (e.g. stateType, stdLogicArray).

FSM Names

- Have a *prefix* "st" and a name that implies the state(e.g. stErase, stProgram).

File Names

- Have the same name as the contained design unit (possibly with the first or all letters in lower case).
- Have file suffix ".vhd" or ".vhdl".

A.3 VHDL Error Hotlist

When intensely modelling circuits in VHDL, the following pitfalls are commonly encountered:

1. default assignments

   It must not be possible to go through a VHDL process statement without having all signals assigned a defined value. The most secure approach is to make a default assignment for all signals used at the beginning of the process.
2. reset polarity

   Some systems use active high reset signals, others use active low signals. If you don't respect the actual reset polarity, you may risk your system stuck in the reset state. And, *no*, this is not easily seen in the behaviour of the system!
3. sensitivity list

   The sensitivity list is a syntax element of minor relevance for the synthesis process. It is, however, very important for the simulation tool. The simulation can only be guaranteed to behave as the synthesized circuit does if attention is payed on correct sensitivity lists.
4. correct user constraints file (`*.ucf`)

   Having the wrong `*.ucf` file assigned to the project is fatal: often, no error message is spawned during the implementation process, but the design might still not work as the pins are not located as expected.

A.4 Scan Chain

Two additional multiplexer are needed to modify an «operational» flip-flop to its «scan-able» equivalent as described in section 5.6.4.1.

In the implementation of the scan-chain the multiplexer are realized by two LUT-Elements of the *Xilinx SPARTAN*^{TM} *II* . A LUT2 element is added for the multiplexer in front of the FF-Enable input (tab. A-1-a) and initalized with the value "E". The data input of the flip-flop is modified with a LUT3 (tab. A-1-b) in front of the FF-Data input and contains the value "E2" as a logic function.

A-1-a LUT2

Signal	Port
FlipFlopEnable	IO
ScanEnable	I1
MuxOutput (D)	0

A-1-b LUT3

Signal	Port
DataIn	IO
ScanChainIn	I1
ScanEnable	I2
MuxOutput (CE)	0

Table A-2: LUT Elements ScanChain: Mapping of the Signals to the In- and Outports of the LUTs Elment of Xilinx SPARTAN^{TM} II .

An running example of the «scan-chain» is shown in figure A-1: As a proof-of-concept a counter is implemented and modified with the scan-chain insertion script. After increasing the value for a certain time the «scan enable» is asserted and the execution mode changes from operational to the scan mode. As could be seen in the figure the states are shifted in on the one side, on the other side the bits are serial shifted out.

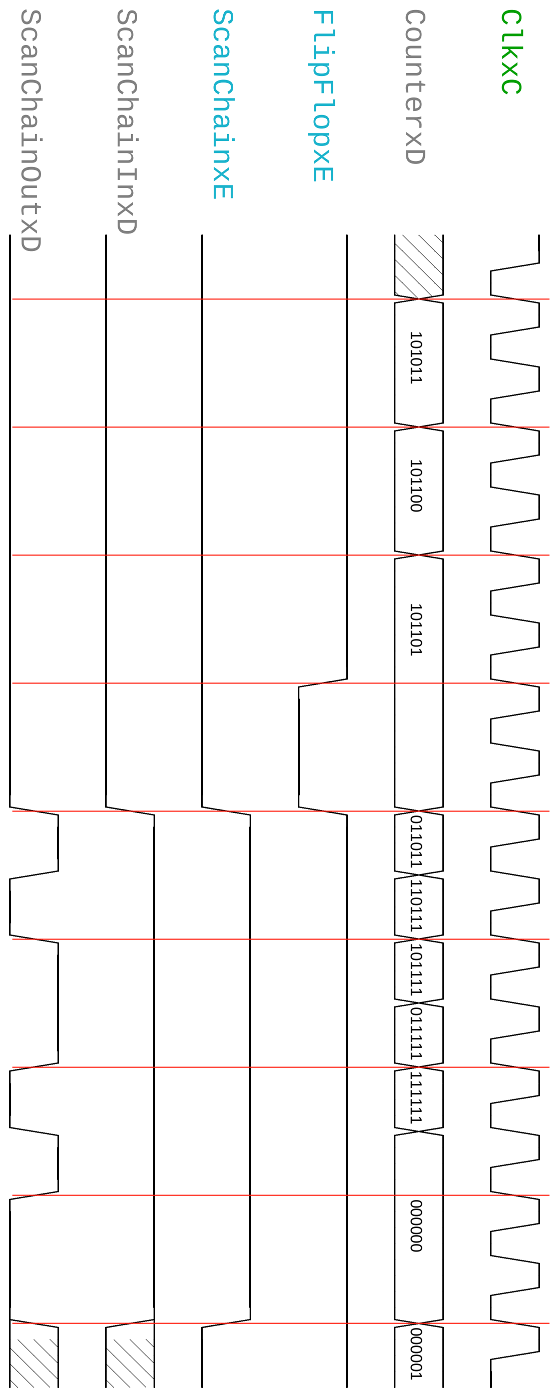


Figure A-1
Waveform of a scan chain in action

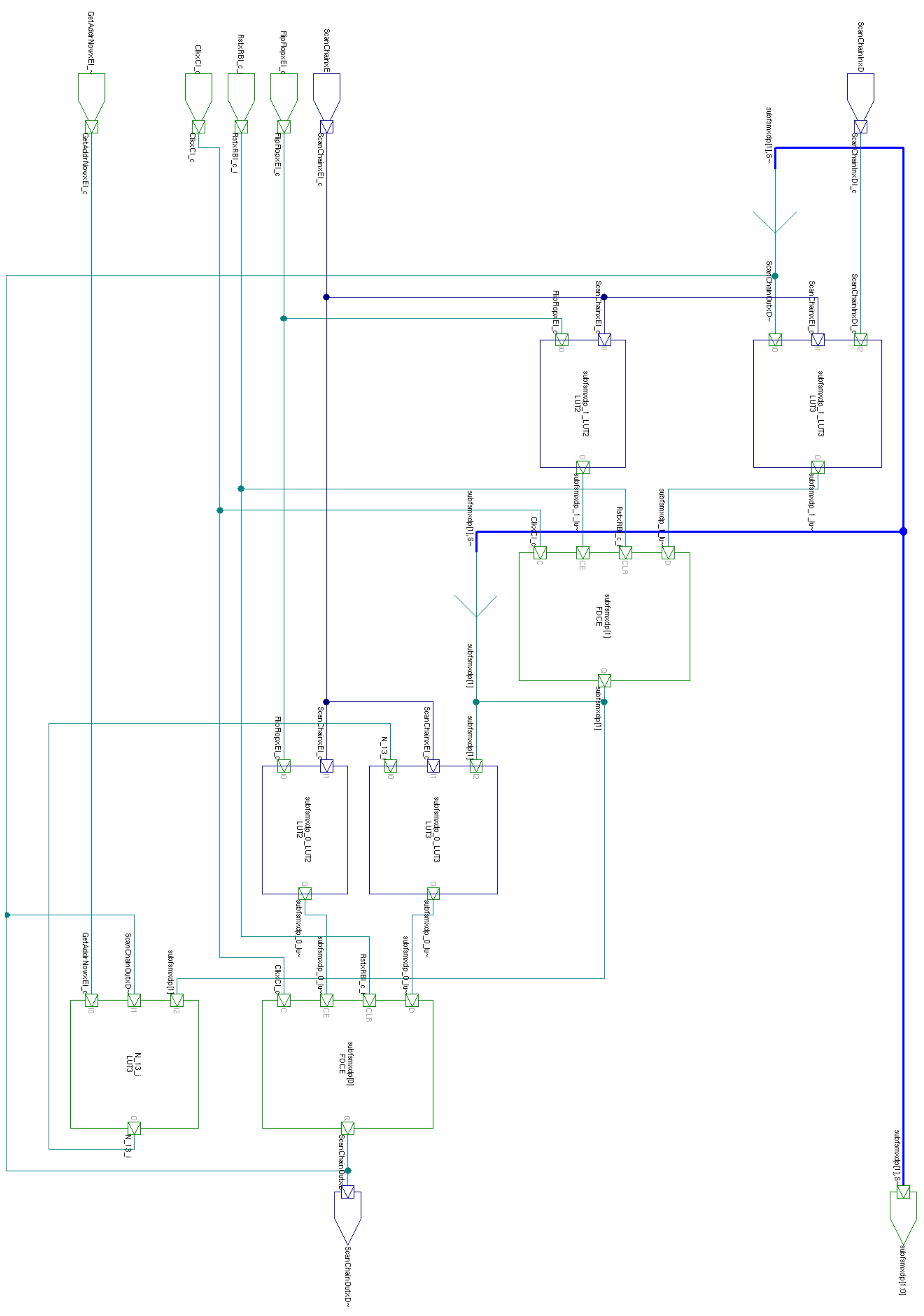


Figure A-2
Example of a scan chain: A netlist improved with a scan chain and visualised with Synopsys «Design Vision» GUI.

A.5 Entities and UCF Files

Writing `*.ucf`-files could be a very error-prone task: We have modified and updated the placement and timing constraints for the CPLD and the FPGA. For the sake of completeness we include the constraint files of these two devices as a reference book.

A.5.1 FPGA: Xilinx SPARTAN™ II *XC2S200*

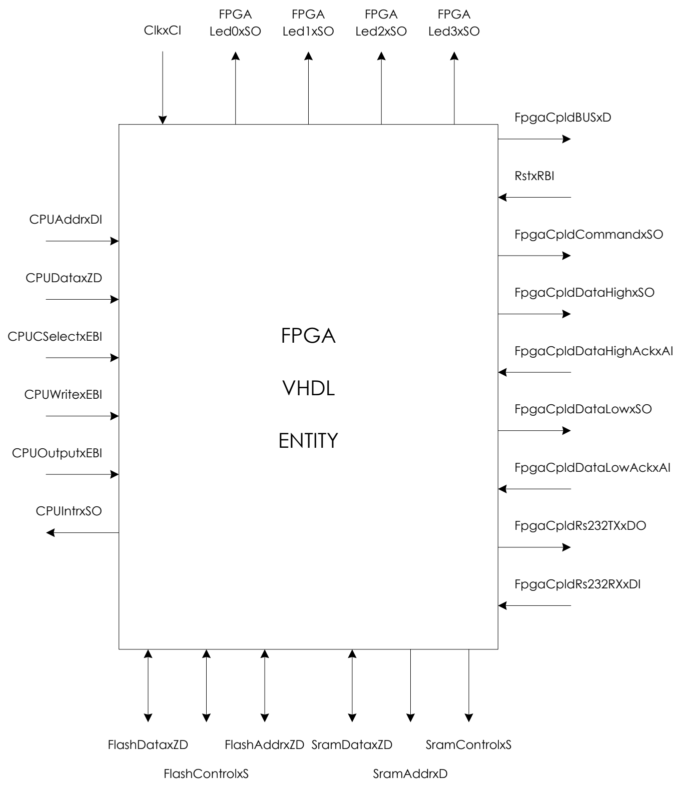


Figure A-3
Entity:
Xilinx SPARTAN™ II *XC2S200 for the* BTNODEFPGA *board*

```
######## Clock and Reset ########
# clk
NET "ClkxCI"              LOC = "P80";

# reset switch
# use the button of the CPLD !!!

######## LEDS ########

# leds for fpga
NET "FPGALed0xSO"         LOC = "P203";
NET "FPGALed1xSO"         LOC = "P204";
NET "FPGALed2xSO"         LOC = "P205";
NET "FPGALed3xSO"         LOC = "P206";

######## FPGA <--> CPLD #########

NET "FpgaCpldBUSxD<0>"      LOC="p60";
NET "FpgaCpldBUSxD<1>"      LOC="p61";
NET "FpgaCpldBUSxD<2>"      LOC="p62";
NET "FpgaCpldBUSxD<3>"      LOC="p63";
NET "FpgaCpldBUSxD<4>"      LOC="p67";
NET "FpgaCpldBUSxD<5>"      LOC="p68";
NET "FpgaCpldBUSxD<6>"      LOC="p69";
NET "FpgaCpldBUSxD<7>"      LOC="p70";

NET "RstxRBI"                 LOC="p71";
NET "FpgaCpldCommandxSO"      LOC="p73";
NET "FpgaCpldDataHighxSO"     LOC="p74";
NET "FpgaCpldDataHighAckxAI"     LOC="p75";
NET "FpgaCpldDataLowxSO"         LOC="p81";
NET "FpgaCpldDataLowAckxAI"      LOC="p82";
NET "FpgaCpldRs232RXxDI"         LOC="p83";
NET "FpgaCpldRs232TXxDO"         LOC="p84";

############# SRAM ###############

# SRAM addresses
NET "SRAMAddrxDO<0>"     LOC = "P14";
NET "SRAMAddrxDO<1>"     LOC = "P9";
NET "SRAMAddrxDO<2>"     LOC = "P7";
NET "SRAMAddrxDO<3>"     LOC = "P5";
NET "SRAMAddrxDO<4>"     LOC = "P3";
NET "SRAMAddrxDO<5>"     LOC = "P4";
NET "SRAMAddrxDO<6>"     LOC = "P6";
NET "SRAMAddrxDO<7>"     LOC = "P8";
NET "SRAMAddrxDO<8>"     LOC = "P41";
NET "SRAMAddrxDO<9>"     LOC = "P42";
NET "SRAMAddrxDO<10>"    LOC = "P43";
NET "SRAMAddrxDO<11>"    LOC = "P44";
NET "SRAMAddrxDO<12>"    LOC = "P45";
NET "SRAMAddrxDO<13>"    LOC = "P59";
NET "SRAMAddrxDO<14>"    LOC = "P58";
NET "SRAMAddrxDO<15>"    LOC = "P57";
NET "SRAMAddrxDO<16>"    LOC = "P49";
NET "SRAMAddrxDO<17>"    LOC = "P48";

# SRAM status signals
NET "SRAMCExEBO"         LOC = "P16";
NET "SRAMHBxEBO"         LOC = "P15";
NET "SRAMLBxEBO"         LOC = "P17";
NET "SRAMOExEBO"         LOC = "P10";
NET "SRAMWExEBO"         LOC = "P47";

# SRAM data
NET "SRAMDataxZD<0>"     LOC = "P18";
NET "SRAMDataxZD<1>"     LOC = "P21";
NET "SRAMDataxZD<2>"     LOC = "P23";
NET "SRAMDataxZD<3>"     LOC = "P27";
NET "SRAMDataxZD<4>"     LOC = "P30";
NET "SRAMDataxZD<5>"     LOC = "P33";
NET "SRAMDataxZD<6>"     LOC = "P35";
NET "SRAMDataxZD<7>"     LOC = "P46";
NET "SRAMDataxZD<8>"     LOC = "P37";
NET "SRAMDataxZD<9>"     LOC = "P36";
NET "SRAMDataxZD<10>"    LOC = "P34";
NET "SRAMDataxZD<11>"    LOC = "P31";
NET "SRAMDataxZD<12>"    LOC = "P29";
NET "SRAMDataxZD<13>"    LOC = "P24";
NET "SRAMDataxZD<14>"    LOC = "P22";
NET "SRAMDataxZD<15>"    LOC = "P20";

######## FPGA IO ##########

NET "CPUWritexEBI"         LOC="p195";
NET "CPUWritexEBI"         pullup;

NET "CPUOutputxEBI"        LOC="p194";
NET "CPUOutputxEBI"        pullup;

NET "CPUCSelectxEBI"       LOC="p193";
NET "CPUCSelectxEBI"       pullup;

NET "CPUIntrxSO"           LOC="p192";

NET "CPUAddrxDI<0>"        LOC="p191";   # NET "FPGA_IO<4>"   LOC="191";
NET "CPUAddrxDI<1>"        LOC="p189";   # NET "FPGA_IO<5>"   LOC="189";
NET "CPUAddrxDI<2>"        LOC="p188";   # NET "FPGA_IO<6>"   LOC="188";
NET "CPUAddrxDI<3>"        LOC="p187";   # NET "FPGA_IO<7>"   LOC="187";

NET "CPUDataxZD<0>"        LOC="p165";   # NET "FPGA_IO<20>"  LOC="p165";
NET "CPUDataxZD<1>"        LOC="p162";   # NET "FPGA_IO<23>"  LOC="p162";
NET "CPUDataxZD<2>"        LOC="p163";   # NET "FPGA_IO<22>"  LOC="p163";
NET "CPUDataxZD<3>"        LOC="p176";   # NET "FPGA_IO<12>"  LOC="p176";
NET "CPUDataxZD<4>"        LOC="p178";   # NET "FPGA_IO<11>"  LOC="p178";
NET "CPUDataxZD<5>"        LOC="p179";   # NET "FPGA_IO<10>"  LOC="p179";
NET "CPUDataxZD<6>"        LOC="p180";   # NET "FPGA_IO<9>"   LOC="p180";
NET "CPUDataxZD<7>"        LOC="p181";   # NET "FPGA_IO<8>"   LOC="p181";
NET "CPUDataxZD<8>"        LOC="p166";   # NET "FPGA_IO<19>"  LOC="p166";
NET "CPUDataxZD<9>"        LOC="p167";   # NET "FPGA_IO<18>"  LOC="p167";
NET "CPUDataxZD<10>"       LOC="p164";   # NET "FPGA_IO<21>"  LOC="p164";
NET "CPUDataxZD<11>"       LOC="p175";   # NET "FPGA_IO<13>"  LOC="p175";
NET "CPUDataxZD<12>"       LOC="p173";   # NET "FPGA_IO<15>"  LOC="p173";
NET "CPUDataxZD<13>"       LOC="p174";   # NET "FPGA_IO<14>"  LOC="p174";
NET "CPUDataxZD<14>"       LOC="p168";   # NET "FPGA_IO<17>"  LOC="p168";
NET "CPUDataxZD<15>"       LOC="p172";   # NET "FPGA_IO<16>"  LOC="p172";

######### FLASH ############
```

```
# FLASH FLAGS
# NET "FlashReadyxE"        LOC="p122";
# NET "FlashWritexEB"       LOC="p125";
# NET "FlashResetxRB"       LOC="p123";
# NET "FlashOutputxEB"      LOC="p88";
# NET "FlashChipxEB"        LOC="p87";

# FLASH DATA
# NET "FlashDataxZD<0>"     LOC="p89";
# NET "FlashDataxZD<1>"     LOC="p90";
# NET "FlashDataxZD<2>"     LOC="p94";
# NET "FlashDataxZD<3>"     LOC="p95";
# NET "FlashDataxZD<4>"     LOC="p96";
# NET "FlashDataxZD<5>"     LOC="p97";
# NET "FlashDataxZD<6>"     LOC="p98";
# NET "FlashDataxZD<7>"     LOC="p99";

# FLASH ADDR
# NET "FlashAddrxZD<0>"     LOC="p86";
# NET "FlashAddrxZD<1>"     LOC="p109";
# NET "FlashAddrxZD<2>"     LOC="p110";
# NET "FlashAddrxZD<3>"     LOC="p111";
# NET "FlashAddrxZD<4>"     LOC="p112";
# NET "FlashAddrxZD<5>"     LOC="p113";
# NET "FlashAddrxZD<6>"     LOC="p114";
# NET "FlashAddrxZD<7>"     LOC="p120";
# NET "FlashAddrxZD<8>"     LOC="p127";
# NET "FlashAddrxZD<9>"     LOC="p129";
# NET "FlashAddrxZD<10>"    LOC="p100";
# NET "FlashAddrxZD<11>"    LOC="p132";
# NET "FlashAddrxZD<12>"    LOC="p133";
# NET "FlashAddrxZD<13>"    LOC="p134";
# NET "FlashAddrxZD<14>"    LOC="p136";
# NET "FlashAddrxZD<15>"    LOC="p138";
# NET "FlashAddrxZD<16>"    LOC="p139";
# NET "FlashAddrxZD<17>"    LOC="p102";
# NET "FlashAddrxZD<18>"    LOC="p121";
# NET "FlashAddrxZD<19>"    LOC="p101";
###########
## TIMINGS ##
###########

NET "ClkxCI" PERIOD = 18432 kHz HIGH 50 %;

NET "CPUWritexEBI" PERIOD = 400 ns LOW 120 ns;

NET "CPUOutputxEBI" PERIOD = 400 ns LOW 120 ns;

NET "cpucommunication_1_InternalClkxCO" PERIOD = 400 ns LOW 120 ns;
```

Listing A-15
UCF File for the FPGA and ISE 6.1

A.5.2 *CPLD:* Xilinx Coolrunner$^{\text{TM}}$ XPLA3 *XCR3384XL*

```
# clk and reset
NET "ClkxCI" LOC = "P128";
NET "ClkxCI" PERIOD = 18.432 MHz;

# reset switch (button pulls to gnd)
NET "RstxRBI" LOC = "P103";
NET "RstxRBI" pullup;

# leds for cpld
NET "CPLDLedxS0" LOC = "P37";
NET "CPLDLedxS1" LOC = "P38";
NET "CPLDLedxS2" LOC = "P39";
NET "CPLDLedxS3" LOC = "P40";

# flash: FlashDataxZD ( 7 downto 0)
NET "FlashDataxZD<0>" LOC = "P69";
NET "FlashDataxZD<1>" LOC = "P68";
NET "FlashDataxZD<2>" LOC = "P67";
NET "FlashDataxZD<3>" LOC = "P66";
NET "FlashDataxZD<4>" LOC = "P65";
NET "FlashDataxZD<5>" LOC = "P63";
NET "FlashDataxZD<6>" LOC = "P62";
NET "FlashDataxZD<7>" LOC = "P61";

# flash: FlashAddrxZD (19 downto 0)
NET "FlashAddrxZD<0>"  LOC = "P72";
NET "FlashAddrxZD<1>"  LOC = "P113";
NET "FlashAddrxZD<2>"  LOC = "P114";
NET "FlashAddrxZD<3>"  LOC = "P116";
```

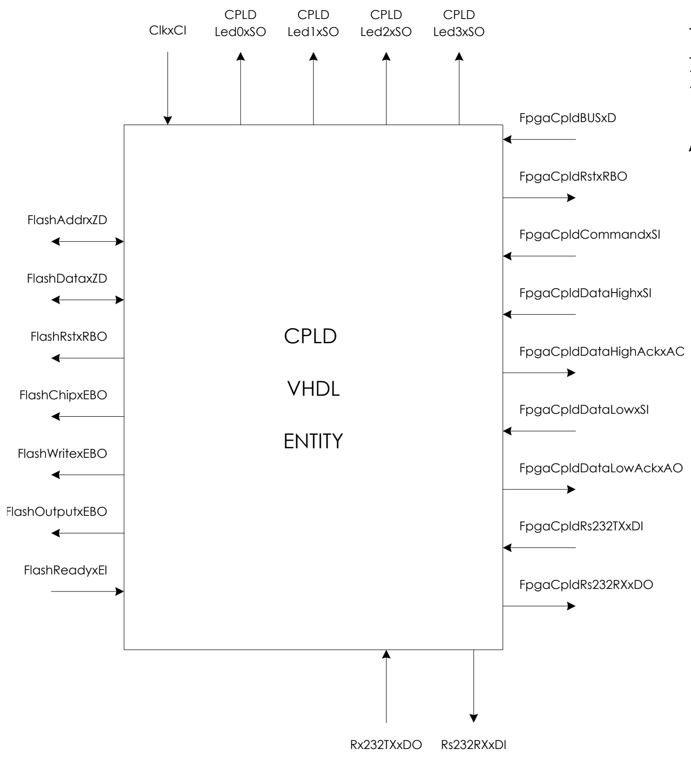


Figure A-4
Entity:
Xilinx Coolrunner^{TM} XPLA3 *XCR3384XL* *for the* BTNODEFPGA *board*

```
NET "FlashAddrxZD<4>"  LOC = "P117";
NET "FlashAddrxZD<5>"  LOC = "P118";
NET "FlashAddrxZD<6>"  LOC = "P119";
NET "FlashAddrxZD<7>"  LOC = "P120";
NET "FlashAddrxZD<8>"  LOC = "P136";
NET "FlashAddrxZD<9>"  LOC = "P137";
NET "FlashAddrxZD<10>" LOC = "P60";
NET "FlashAddrxZD<11>" LOC = "P138";
NET "FlashAddrxZD<12>" LOC = "P139";
NET "FlashAddrxZD<13>" LOC = "P140";
NET "FlashAddrxZD<14>" LOC = "P141";
NET "FlashAddrxZD<15>" LOC = "P142";
NET "FlashAddrxZD<16>" LOC = "P143";
NET "FlashAddrxZD<17>" LOC = "P55";
NET "FlashAddrxZD<18>" LOC = "P122";
NET "FlashAddrxZD<19>" LOC = "P56";

# flash control signals
NET "FlashReadyxEI"             LOC = "P132";
NET "FlashChipxEBO"             LOC = "P71";
NET "FlashOutputxEBO"           LOC = "P70";
NET "FlashRstxRBO"              LOC = "P133";
NET "FlashWritexEBO"            LOC = "P134";

# flash slave parallel interface
NET "FpgaSlaveParProgramxEBO"   LOC = "P2";
NET "FpgaSlaveParDonexEI"       LOC = "P4";
NET "FpgaSlaveParInitxEI"       LOC = "P1";
#NET "FpgaSlaveParBusyxE"       LOC = "P7"; # not used below 50 MHz
NET "FpgaSlaveParWRxEBO"        LOC = "P5";
NET "FpgaSlaveParCSxEBO"        LOC = "P6";

# rs232 interface (to Host PC)
NET "Rs232RXxDI"        LOC = "P74";
NET "Rs232TXxDO"        LOC = "P75";

# uc interface (to BTnode)
# NET "uc<0>"   LOC = "P74";
# NET "uc<1>"   LOC = "P75";
# NET "uc<2>"   LOC = "P77";
# NET "uc<3>"   LOC = "P78";
# NET "uc<4>"   LOC = "P79";
# NET "uc<5>"   LOC = "P80";
# NET "uc<6>"   LOC = "P81";
# NET "uc<7>"   LOC = "P82";
# FPGA CPLD BUS (15 downto 0)
NET "FpgaCpldBUSxD<0>"          LOC = "P36";
NET "FpgaCpldBUSxD<1>"          LOC = "P35";
NET "FpgaCpldBUSxD<2>"          LOC = "P34";
NET "FpgaCpldBUSxD<3>"          LOC = "P32";
NET "FpgaCpldBUSxD<4>"          LOC = "P31";
NET "FpgaCpldBUSxD<5>"          LOC = "P30";
NET "FpgaCpldBUSxD<6>"          LOC = "P29";
NET "FpgaCpldBUSxD<7>"          LOC = "P28";

NET "FpgaCpldRstxRBO"           LOC = "p27";
NET "FpgaCpldCommandxSI"        LOC = "p26";
NET "FpgaCpldDataHighxSI"       LOC = "p25";
NET "FpgaCpldDataHighAckxAO"    LOC = "p23";
NET "FpgaCpldDataLowxSI"        LOC = "p21";
NET "FpgaCpldDataLowAckxAO"     LOC = "p20";
NET "FpgaCpldRs232RXxDO"        LOC = "p19";
NET "FpgaCpldRs232TXxDI"        LOC = "p18";

# NET "FpgaCpldBUSxZD<0>"              LOC = "P36";
# NET "FpgaCpldBUSxZD<1>"              LOC = "P35";
# NET "FpgaCpldBUSxZD<2>"              LOC = "P34";
# NET "FpgaCpldBUSxZD<3>"              LOC = "P32";
# NET "FpgaCpldBUSxZD<4>"              LOC = "P31";
# NET "FpgaCpldBUSxZD<5>"              LOC = "P30";
# NET "FpgaCpldBUSxZD<6>"              LOC = "P29";
# NET "FpgaCpldBUSxZD<7>"              LOC = "P28";

# NET "FpgaCpldBUSxZD<8>"              LOC = "P27";
# NET "FpgaCpldBUSxZD<9>"              LOC = "P26";
# NET "FpgaCpldBUSxZD<10>"             LOC = "P25";
# NET "FpgaCpldBUSxZD<11>"             LOC = "P23";
# NET "FpgaCpldBUSxZD<12>"             LOC = "P21";
# NET "FpgaCpldBUSxZD<13>"             LOC = "P20";
# NET "FpgaCpldBUSxZD<14>"             LOC = "P19";
# NET "FpgaCpldBUSxZD<15>"             LOC = "P18";
```

Listing A-16
UCF File for the CPLD and ISE 6.1

B

IPAQ Driver and Data Streamer

Embedded Machine on FPGA IPAQ Driver and Data Streamer

Reference Manual

Generated by Roman Plessl
and Doxygen 1.3.6

Version 1.10, April 21, 2004

Contents

1 Data Structure Index 1

2 File Index 1

3 Page Index 1

4 Data Structure Documentation 1

5 File Documentation 2

6 Page Documentation 21

1 Data Structure Index

1.1 Embedded Machine on FPGA - IPAQ Driver and Data Streamer Data Structures

Here are the data structures with brief descriptions:

CFpga (Functions to access the FPGAModule) 1

ISTData (Interrupt Service Thread Struct) 2

2 File Index

2.1 Embedded Machine on FPGA - IPAQ Driver and Data Streamer File List

Here is a list of all documented files with brief descriptions:

DataStreamWriter.cpp (Embedded Machine FPGA Loader (DataStreamWriter)) 2

EmbeddedMachine.h (Constants for the "Embedded Machine on FPGA" on the IPAQ) 11

FPGADriver.cpp (Embedded Machine FPGA Communication Driver for Windows CE) 12

FPGADriver.h (Embedded Machine FPGA Communication Driver for Windows CE) 20

3 Page Index

3.1 Embedded Machine on FPGA - IPAQ Driver and Data Streamer Related Pages

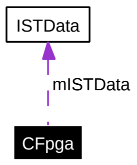
Here is a list of all related documentation pages:

Todo List 21

4 Data Structure Documentation

4.1 CFpga Class Reference

Collaboration diagram for CFpga:



4.1.1 Detailed Description

The CFpga Class enables an access to the FPGAModule connected to an IPAQ running Windows CE. The implemented functionality has read and write functions for the memory-mapped expansion pack, an hardware interrupt handler and runtime functions (constructor, destructor). All functions are implemented to run in the user-space and can be used by user applications.

Public Member Functions

- int **Init** (void)
- int **DeInit** (void)
- int **Read** (UINT pipe, PUSHORT pBuffer, UINT ucb)
- int **Write** (UINT pipe, PUSHORT pBuffer, UINT ucb)
- int **SlowWrite** (UINT pipe, PUSHORT pBuffer, UINT ucb)

4.2 ISTData Struct Reference

4.2.1 Detailed Description

The IST (Interrupt Service Thread) Struct merges parameters and links to thread instances, which are needed for the interrupt handler at runtime.

Parameters:

hThread : Handle to a ThreadObject (self-reference to the running instance of the IST)

sysIntr : Name of hardware interrupt (for the expansion slot we need SYSINTR_OPT)

hEvent : Handle to a EventObject (event handler links "hardware" interrupt and "software" event)

bAbort : Boolean if this IST instance should be aborted (done at DeInit() or by a second thread)

nPriority : Integer sets the priority of the Windows CE Thread (range 0 - 255, s.a. the priorities listed in FPGA_IST_PRIORITY)

Data Fields

- HANDLE **hThread**
- DWORD **sysIntr**
- HANDLE **hEvent**
- volatile BOOL **bAbort**
- int **nPriority**

5 File Documentation

5.1 DataStreamWriter.cpp File Reference

5.1.1 Detailed Description

Embedded Machine FPGA Loader (DataStreamWriter):

This Windows CE program implements a streamer for "FPGA configuration bitstreams". A new bitstream is created, loaded via a PC and Microsoft ActiveSync to the IPAQ device and streamed out – over the expansion slot – to the FPGA board (BTNodeFPGA board) afterwards.

Author:
Roman Plessl

Version:
1.7

Date:
2004/04/21

Id
DataStreamWriter.cpp,v 1.7 2004/04/21 18:54:58 rplessl Exp

Defines

- #define **WIN32_LEAN_AND_MEAN**
- #define **ZONE_ERROR** DEBUGZONE(0)
- #define **ZONE_WARNING** DEBUGZONE(1)
- #define **ZONE_FUNC** DEBUGZONE(2)

Functions

- int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int CmdShow)
  Main Program.

5.1.2 Function Documentation

5.1.2.1 int WINAPI WinMain (HINSTANCE *hInstance*, HINSTANCE *hPrevInstance*, LPTSTR *lpCmdLine*, int *CmdShow*)

Parameters:

HINSTANCE hInstance : Handle of the currently running instance of the application

HINSTANCE hPrevInstance : Handle of the previously running instance of the same applicaiton. In the 32-bit API of Windows CE, this parameter is always NULL (because every program runs in its own address space)

LPTSTR lpCmdLine : command line arguments for the program (in UNICODE)

int CmdShow : declare the style of the main window when the program first starts (i.e. maximized, minimized and that like)

Returns:
Message wParam : indicates success ’0’ : indicates that the function terminates before entering the message loop

This main program acts as an Data Streamer to the FPGA board. At the beginning, an instance of the userspace driver is created and initalized with the accurate values.

```
Initializing FPGADriver ...
--> suceeded
```

If the creation and initalize process suceeds a simple application — commandline and char inputs as a user-interface — appears which allows to choose one of the different operation and streaming modes.

```
Writing bitstream in short words
to FPGA device ...

Choice BitStream:
0 - Reseting Flash
1 - Erase Flash
2 - New BitStream
3 - Active SlaveParallel
5 - New BitStream non verbose
6 - New BitStream   SELECT (SELECT SECTOR)
7 - Active SlavePar SELECT (SELECT SECTOR)
Q - Break
```

By select one of this entries a bitstream is read from the filesystem and written to the expansion port. Afterwards the write procedure the Data Streamser terminates its communication phase and the user-space driver is deinitalized.

```
DeInit FPGADriver ...
suceeded!
```

The program has to be terminated by pressing SPACE + ENTER

```
Press SPACE ENTER to exit!
```

```
// Handle to the FPGADriver
CFpga* pMydevice;

// Use of debug output to host
DEBUGREGISTER(hInstance);

// This text goes to the console window on the IPAQ
printf("FPGA Test\n");
// This text goes to the debug window on the host (if connected and in debug mode)
DEBUGMSG(ZONE_FUNC, (TEXT("FPGA Test.\n")));

pMydevice = new CFpga();


printf("Initializing FPGADriver ... ");
if(!pMydevice->Init()){
  printf("\n --> failed!\n");
  return 0;
}
printf("\n --> suceeded!\n");

/* Get data from file */

FILE *stream;
int numread; // number of read bytes

/* data width of the expansion slot is 16 bit
   we read in two BYTEs and convert them to a
   16 bit unsigned short value (USHORT)

   CAUTION on byte order (big/little endian)
*/

USHORT data[1];
BYTE tempdata[2];

/* Output user-informations and stay in switch function*/

char *line = "__";
char c;

printf("Writing bitstream in short words\n");
printf("to FPGA device ...\n");

printf("\n");

printf("Choice BitStream:\n ");
printf(" 0 - Reseting Flash\n ");
printf(" 1 - Erase Flash\n ");
printf(" 2 - New BitStream\n ");
printf(" 3 - Active SlaveParallel\n ");
printf(" 5 - New BitStream non verbose\n ");
printf(" 6 - New BitStream   SELECT \n ");
printf(" 7 - Active SlavePar SELECT \n ");
printf(" Q - Break \n\n > ");

gets (line);

switch(c = *line)
  {
case '0':
    /* Open file in binary mode: */
    if( (stream = fopen( "Bitstreams/reseting.bit", "rb" )) != NULL )
      {
        printf("open file (RESET FLASH) \n");

        while( !feof( stream ) )
          {
            // read 2 bytes from file <stream>
            numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

            if (numread > 0){
              // construct a 16 bit USHORT from 2 BYTEs
              // big to little endian conversion
              data[0]=(tempdata[0]<<8) + tempdata[1];

              // Debug out
              printf("DATA: %Xh \n",data[0]);

              // Write to FPGA:
              // Address : 0
              // Data    : data[0] (16 bit USHORT value)
              // Size    : 1 USHORT
              pMydevice->Write(DATA_REG,&data[0],1);
            }
          }
        printf("close file (RESET FLASH)\n");
        fclose( stream );
      }
    else
      {
        printf("Error opening file (RESET FLASH)\n");
      }
    break;
case '1':
    /* Open file in binary mode: */
    if( (stream = fopen( "Bitstreams/erasing.bit", "rb" )) != NULL )
      {
        printf("open file (ERASE CHIP) \n");

        while( !feof( stream ) )
          {
            // read 2 bytes from file <stream>
            numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

            if (numread > 0){
                    // construct a 16 bit USHORT from 2 BYTEs
                    // big to little endian conversion
                    data[0]=(tempdata[0]<<8) + tempdata[1];

                    // Debug out
                    printf("DATA: %Xh \n",data[0]);

                    // Write to FPGA:
                    // Address : 0
                    // Data    : data[0] (16 bit USHORT value)
                    // Size    : 1 USHORT
                    pMydevice->Write(DATA_REG,&data[0],1);
                  }
                }
              printf("close file (ERASE CHIP)\n");
              fclose( stream );
            }
          else
            {
              printf("Error opening file (ERASE CHIP)\n");
            }
          break;

        case '2':
        default:
          /* Open file in binary mode: */
          if( (stream = fopen( "Bitstreams/fpga_blinkenlight_mod.bit", "rb" )) != NULL )
            {
              printf("open file (NEW BITSTREAM)\n");

              UINT counter = 0;

              while( !feof( stream ) )
                {
                  // read 2 bytes from file <stream>
                  numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

                  if (numread > 0){
                    // construct a 16 bit USHORT from 2 BYTEs
                    // big to little endian conversion
                    data[0]=(tempdata[0]<<8) + tempdata[1];


                    // Debug out
                    if((counter % 100) == 0){
                            printf("DATA: %.4Xh %i\n",data[0], counter);
                    }

                    // Write to FPGA:
                    // Address : 0
                    // Data    : data[0] (16 bit USHORT value)
                    // Size    : 1 USHORT
                    pMydevice->Write(DATA_REG,&data[0],1);

                    counter++;
                      }
                }
              printf("close file (NEW BITSTREAM)\n");
              fclose( stream );
            }
          else
            {
              printf("Error opening file (NEW BITSTREAM)\n");
            }
          break;

        case '3':
```

```
      /* Open file in binary mode: */
      if( (stream = fopen( "Bitstreams/slaveparallel_0.bit", "rb" )) != NULL )
        {
          printf("open file (SLAVE PARALLEL)\n");

          while( !feof( stream ) )
            {
              // read 2 bytes from file <stream>
              numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

              if (numread > 0){
                // construct a 16 bit USHORT from 2 BYTEs
                // big to little endian conversion
                data[0]=(tempdata[0]<<8) + tempdata[1];

                // Debug out
                printf("DATA: %Xh \n",data[0]);

                // Write to FPGA:
                // Address : 0
                // Data:     data[0] (16 bit USHORT value)
                // Size:     1 USHORT
                pMydevice->Write(DATA_REG,&data[0],1);
              }
            }
          printf("close file (SLAVE PARALLEL)\n");
          fclose( stream );
        }
      else
        {
          printf("Error opening file\n");
        }
      break;
 case '5':





 /* Open file in binary mode: */
      if( (stream = fopen( "Bitstreams/fpga_blinkenlight_mod.bit", "rb" )) != NULL )
        {
          printf("open file (NEW BITSTREAM)\n");

          UINT counter = 0;

          while( !feof( stream ) )
            {
              // read 2 bytes from file <stream>
              numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

              if (numread > 0){
                // construct a 16 bit USHORT from 2 BYTEs
                // big to little endian conversion
                data[0]=(tempdata[0]<<8) + tempdata[1];

            // Write to FPGA:
                // Address : 0
                // Data    : data[0] (16 bit USHORT value)
                // Size    : 1 USHORT
                pMydevice->Write(DATA_REG,&data[0],1);

                counter++;
                  }
            }
          printf("close file (NEW BITSTREAM)\n");
          fclose( stream );
```

```
  }
 else
  {
   printf("Error opening file (NEW BITSTREAM)\n");
      }
 break;

case '6':
 printf(" Select Sector: \n ");
 printf(" [0]: loader [3]: blinkenlight \n ");
 printf(" [6]: flashinglight \n ");
 printf("\n > ");

 gets (line);

 switch(c = *line)
  {
  case '0':
  default:
   /* Open file in binary mode: */
   if( (stream = fopen( "Bitstreams/fpga_flash_loader_mod_0.bit", "rb" )) != NULL )
    {
     printf("open file fpga_flash_loader_mod_0.bit \n");

     UINT counter = 0;

     while( !feof( stream ) )
      {
       // read 2 bytes from file <stream>
       numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

       if (numread > 0){
        // construct a 16 bit USHORT from 2 BYTEs
        // big to little endian conversion
        data[0]=(tempdata[0]<<8) + tempdata[1];

        // Write to FPGA:
        // Address : 0
        // Data    : data[0] (16 bit USHORT value)
        // Size    : 1 USHORT
        pMydevice->Write(DATA_REG,&data[0],1);

        counter++;
       }
      }
     printf("close file fpga_flash_loader_mod_0.bit \n");
     fclose( stream );
    }
   else
    {
     printf("Error opening file fpga_flash_loader_mod_0.bit\n");
    }
   break;
  case '3':
   /* Open file in binary mode: */
   if( (stream = fopen( "Bitstreams/fpga_blinkenlights_mod_3.bit", "rb" )) != NULL )
    {
     printf("open file fpga_blinkenlights_mod_3.bit \n");

     UINT counter = 0;

     while( !feof( stream ) )
      {
       // read 2 bytes from file <stream>
       numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

       if (numread > 0){
```

```
                        // construct a 16 bit USHORT from 2 BYTEs
                        // big to little endian conversion
                        data[0]=(tempdata[0]<<8) + tempdata[1];

                        // Write to FPGA:
                        // Address : 0
                        // Data    : data[0] (16 bit USHORT value)
                        // Size    : 1 USHORT
                        pMydevice->Write(DATA_REG,&data[0],1);

                        counter++;
                      }
                    }
                  printf("close file fpga_blinkenlights_mod_3.bit \n");
                  fclose( stream );
                }
              else
                {
                  printf("Error opening file fpga_blinkenlights_mod_3.bit\n");
                }
              break;
            case '6':
              /* Open file in binary mode: */
              if( (stream = fopen( "Bitstreams/fpga_flashinglights_mod_6.bit", "rb" )) != NULL )
                {
                  printf("open file fpga_flashinglights_mod_6.bit \n");

                  UINT counter = 0;

                  while( !feof( stream ) )
                    {
                      // read 2 bytes from file <stream>
                      numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

                      if (numread > 0){
                        // construct a 16 bit USHORT from 2 BYTEs
                        // big to little endian conversion
                        data[0]=(tempdata[0]<<8) + tempdata[1];

                        // Write to FPGA:
                        // Address : 0
                        // Data    : data[0] (16 bit USHORT value)
                        // Size    : 1 USHORT
                        pMydevice->Write(DATA_REG,&data[0],1);

                        counter++;
                      }
                    }
                  printf("close file fpga_flashinglights_mod_6.bit \n");
                  fclose( stream );
                }
              else
                {
                  printf("Error opening file fpga_flashinglights_mod_6.bit\n");
                }
              break;
            }
          break;

        case '7':
          printf(" Select Sector: \n ");
          printf(" [0-F]\n ");
          printf("\n > ");

          gets (line);

          switch(c = *line)
```

```
      {
      case '0':
      default:
        /* Open file in binary mode: */
        if( (stream = fopen( "Bitstreams/slaveparallel_0.bit", "rb" )) != NULL )
          {
            printf("open file (SLAVE PARALLEL)\n");

            while( !feof( stream ) )
              {
                // read 2 bytes from file <stream>
                numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );

                if (numread > 0){
                  // construct a 16 bit USHORT from 2 BYTEs
                  // big to little endian conversion
                  data[0]=(tempdata[0]<<8) + tempdata[1];

                  // Debug out
                  printf("DATA: %Xh \n",data[0]);

                  // Write to FPGA:
                  // Address : 0
                  // Data:     data[0] (16 bit USHORT value)
                  // Size:     1 USHORT
                  pMydevice->Write(DATA_REG,&data[0],1);
                }
              }
            printf("close file (SLAVE PARALLEL)\n");
            fclose( stream );
          }
        else
          {
            printf("Error opening file\n");
          }
        break;

      case '3':
        if( (stream = fopen( "Bitstreams/slaveparallel_3.bit", "rb" )) != NULL )
          {
            printf("open file (SLAVE PARALLEL)\n");
            while( !feof( stream ) )
              {
                numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );
                if (numread > 0){
                  data[0]=(tempdata[0]<<8) + tempdata[1];
                  printf("DATA: %Xh \n",data[0]);
                  pMydevice->Write(DATA_REG,&data[0],1);
                }
              }
            printf("close file (SLAVE PARALLEL)\n");
            fclose( stream );
          }
        else
          {
            printf("Error opening file\n");
          }
        break;

      case '6':
        if( (stream = fopen( "Bitstreams/slaveparallel_6.bit", "rb" )) != NULL )
          {
            printf("open file (SLAVE PARALLEL)\n");
            while( !feof( stream ) )
              {
                numread = fread(&(tempdata[0]), sizeof(BYTE), 2, stream );
                if (numread > 0){
                    data[0]=(tempdata[0]<<8) + tempdata[1];
                    printf("DATA: %Xh \n",data[0]);
                    pMydevice->Write(DATA_REG,&data[0],1);
                  }
                }
              printf("close file (SLAVE PARALLEL)\n");
              fclose( stream );
            }
          else
            {
              printf("Error opening file\n");
            }
          break;
        }
      break;

    case ' ':
    case 'Q':
    case 'q':
      break;
    }

  printf("writing done! \n");

  /* READ Status Register of the FPGA Framework
     implemented but not tested ....
  */

  // pMydevice->Read(STATUS_REG,&data[0],1);
  // printf("DATA: %Xh \n", data[0]);


   printf("DeInit FPGADriver ...");
   if(!pMydevice->DeInit()){
     printf("failed!\n");
     return 0;
   }
   printf("suceeded!\n");

   delete pMydevice;
   pMydevice = NULL;

   printf("Press SPACE ENTER to exit!\n");
   while(getchar()!=32);

   return 1;
}
```

5.2 EmbeddedMachine.h File Reference

5.2.1 Detailed Description

Constants for the "Embedded Machine on FPGA" project for software on IPAQ

Author:
Roman Plessl

Version:
1.5

Date:
2004/04/21

$Id $

Defines

- #define DATA_REG 0
  Data Register in FPGA.
- #define BLOCKING_REG 4
  Blocking Register in FPGA.
- #define STATUS_REG 15
  Status Register in FPGA.

5.2.2 Define Documentation

5.2.2.1 #define BLOCKING_REG 4

Status Register in FPGA is mapped to the Address 4 (equals 4 in HEX). The Address is switched by internal FPGA Addresses and the wires connected to the FPGA Pins (CPUAddrxDI)

5.2.2.2 #define DATA_REG 0

Data Register in FPGA is mapped to the Address 0 (equals 0 in HEX). The Address is switched by internal FPGA Addresses and the wires connected to the FPGA Pins (CPUAddrxDI)

5.2.2.3 #define STATUS_REG 15

Status Register in FPGA is mapped to the Address 15 (equals F in HEX). The Address is switched by internal FPGA Addresses and the wires connected to the FPGA Pins (CPUAddrxDI)

5.3 FPGADriver.cpp File Reference

5.3.1 Detailed Description

Source for the FPGADriver.dll. This DLL is loaded by user apps to access the FPGAModule, which is memory-mapped to the MCS4 region of the expansion pack.
Driver was initially designed as a Windows CE Stream Device Driver, which could be automatically loaded upon expansion pack insertion. However, due to debugging reasons and simpler handling, the driver now can only be loaded by a user apps in user space.

Author:
Matthias Dyer
Roman Plessl

Version:
1.9

Date:
2004/04/21

Todo
Interruptenable as parameter e.g. in constructor

Id
FPGADriver.cpp,v 1.9 2004/04/21 18:54:58 rplessl Exp

Defines

- #define WIN32_LEAN_AND_MEAN
  Exclude rarely-used stuff from Windows headers.
- #define DTAG TEXT ("FPGADrv: ")
  Debug zone support.
- #define ZONE_ERROR DEBUGZONE(0)
  Debug zone constant.

- #define ZONE_WARNING DEBUGZONE(1)
  Debug zone constant.
- #define ZONE_FUNC DEBUGZONE(2)
  Debug zone constant.
- #define ZONE_INIT DEBUGZONE(3)
  Debug zone constant.
- #define ZONE_DRVCALLS DEBUGZONE(4)
  Debug zone constant.
- #define FPGA_IO_BASE 0x10000000
  FPGA_IO_BASE: Base Address in IPAQ Address Space.
- #define FPGA_IO_SPACE 64
  FPGA_IO_SPACE: Size of Space.
- #define FPGA_IST_PRIORITY 152
  FPGA_IST_PRIORITY: Interrupt Service Thread Priority.
- #define FPGA_INT_ENABLE 1
  FPGA_INT_ENABLE : Switch (Enable, Disable) of Interrupt.
- #define SLOW_FACTOR 100
  slowing parameter

Functions

- DWORD FPGA_IST (void $*$dat)
  Interrupt Service Thread Function.
- BOOL APIENTRY DllMain (HANDLE hModule, DWORD ul_reason_for_call, LPVOID lp-Reserved)
  DllMain - only used for logging here.
- CFpga ()
  Constructor.
- int Init ()
  Init.
- int DeInit ()
  DeInit.
- int Read (UINT pipe, PUSHORT pBuffer, UINT ucb)
  Read.
- int Write (UINT pipe, PUSHORT pBuffer, UINT ucb)
  Write.
- int SlowWrite (UINT pipe, PUSHORT pBuffer, UINT ucb)
  SlowWrite.
- void Slower (void)
  Slower.
- void ReadStatusReg (void)
  Read Status Reg.

5.3.2 Define Documentation

5.3.2.1 #define DTAG TEXT ("FPGADrv: ")

DTAG : Used as a prefix string for all debug zone messages.

5.3.2.2 #define FPGA_IST_PRIORITY 152

FPGA_IST_PRIORITY is a WinCE Threads Priority
WinCE Threads Priority

- 0 - 96 : real time (above drivers)
- 97 - 152 : WinCE based drivers

- 153 - 247 : real time (below drivers)
- 248 - 255 : non real time (applications)

more enhanced thread list from http://www.windowsfordevices.com/articles/AT3859908246.html
Real-Time Thread Priorities: CeSetThreadPriority

- 0-19 Open Real Time Above Drivers
- 20 Permedia Vertical Retrace
- 21-98 Open Real Time Above Drivers
- 99 Power management Resume Thread
- 100-108 USB OHCI UHCI, Serial
- 109-129 Irsir1, NDIS, Touch
- 130 KITL
- 131 VMini
- 132 CxPort
- 133-144 Open Device Drivers
- 145 PS2 Keyboard
- 146-147 Open Device Drivers
- 148 IRComm
- 149 Open Device Drivers
- 150 TAPI
- 151-152 Open Device Drivers
- 153-247 Open Real Time Below Drivers

Normal Thread Priorities: SetThreadPriority

- 248 Power Management
- 249 WaveDev, TVIA5000, Mouse, PnP ,Power
- 250 WaveAPI
- 251 Power Manager Battery Thread
- 252-255 Open

5.3.2.3 #define ZONE_DRVCALLS DEBUGZONE(4)

DEBUGZONE(4)
ZONE_DRVCALLS: DRIVER CALLS exception (needed for debugging a DLL (additonal information))

5.3.2.4 #define ZONE_ERROR DEBUGZONE(0)

DEBUGZONE(0)
ZONE_ERROR : ERROR exception (needed for debugging a DLL (additonal information))

5.3.2.5 #define ZONE_FUNC DEBUGZONE(2)

DEBUGZONE(2)
ZONE_FUNC : FUNCTION exception (needed for debugging a DLL (additonal information))

5.3.2.6 #define ZONE_INIT DEBUGZONE(3)

DEBUGZONE(3)
ZONE_INIT : INIT exception (needed for debugging a DLL (additonal information))

5.3.2.7 #define ZONE_WARNING DEBUGZONE(1)

DEBUGZONE(1)
ZONE_WARNING : WARNING exception (needed for debugging a DLL (additonal information))

5.3.3 Function Documentation

5.3.3.1 CFpga ()

Constuctor of the class CFPGA

```
  pFpgaRegs = NULL;
  return;
}
\end{verbatim}\normalsize
\hypertarget{FPGADriver_8cpp_a16}{
\index{FPGADriver.cpp@{FPGADriver.cpp}!DeInit@{DeInit}}
\index{DeInit@{DeInit}!FPGADriver.cpp@{FPGADriver.cpp}}
\paragraph[DeInit]{\setlength{\rightskip}{0pt plus 5cm}int De\-Init ()}\hfill}
\label{FPGADriver_8cpp_a16}


Deactivate, disable and destroy the Interrupt Service Thread.

\begin{Desc}
\item[Returns:]1 : if success

0 : else \end{Desc}



\footnotesize\begin{lstlisting}[language=C++]347 {
  DEBUGMSG(ZONE_FUNC, (DTAG TEXT("DeInit++\r\n")));
  if(mISTData.hThread){
    // stopping the IST
    mISTData.bAbort = TRUE;
    SetEvent(mISTData.hEvent);
    InterruptDisable(mISTData.sysIntr);
    CloseHandle(mISTData.hEvent);
    CloseHandle(mISTData.hThread);
  }
  //unmap
  VirtualFree((void*)pFpgaRegs, FPGA_IO_SPACE, MEM_RELEASE);

  DEBUGMSG(ZONE_FUNC, (DTAG TEXT("DeInit--\r\n")));
  return 1;
};
```

5.3.3.2 BOOL APIENTRY DllMain (HANDLE *hModule*, DWORD *ul_reason_for_call*, LPVOID *lpReserved*)

The DllMain function is an optional entry point into a dynamic-link library (DLL). If the function is used, it is called by the system when processes and threads are initialized and terminated. DLLMain Initializes a non-MFC DLL.
We use this function for the debug logging functionality

Parameters:

hModule : Handle to the DLL module. The value is the base address of the DLL.

ul_reason_for_call : Indicates why the DLL entry-point function is being called.

lpReserved : additional parameters for ul_reason_for_call

Returns:

’true’ : if initialization succeeds (ul_reason_for_call = DLL_PROCESS_ATTACH)
’false’: if initialization fails (ul_reason_for_call = DLL_PROCESS_ATTACH)
NULL : in other configurations

```
  switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
      DEBUGREGISTER((HINSTANCE)hModule);
      DEBUGMSG(ZONE_INIT, (DTAG TEXT("DLL_PROCESS_ATTACH.\r\n")));
      break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
      DEBUGMSG(ZONE_INIT, (DTAG TEXT("DLL_PROCESS_DETACH.\r\n")));
      break;
    }
  return TRUE;
}
```

5.3.3.3 DWORD FPGA_IST (void ∗ *dat*)

Parameters:

dat : ISTData∗ struct with priority, event handle and int number

The Interrupt handling of the XScale processor and its processing in Windows CE is shown in the figure. An hardware interrupt on the expansion port (FPGA board) is usally thrown as an edge triggered impuls, but the XScale interrupt unit reacts on level transitions. To fix this "double-counting" of the interrupt signals a T-flipflop between source and destination is used for "translation". An output change on the Q output of the T-FF needs two interrupt triggers at the D input.

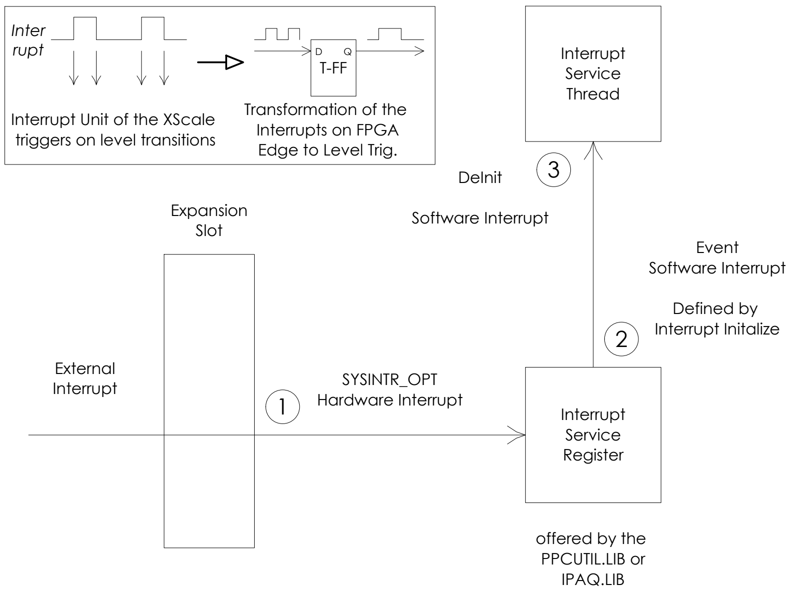

Inter rupt
Interrupt Unit of the XScale triggers on level transitions
D Q
T-FF
Transformation of the Interrupts on FPGA Edge to Level Trig.
Interrupt Service Thread
DeInit
3
Software Interrupt
Expansion Slot
Event Software Interrupt
Defined by Interrupt Initalize
2
External Interrupt
1
SYSINTR_OPT Hardware Interrupt
Interrupt Service Register
offered by the PPCUTIL.LIB or IPAQ.LIB

Figure 1: Interrupt Unit on the IPAQ

(1) symbolises the Hardware Interrupt from the expansion slot
(2) the link between Hardware Interrupt and Software Event
(3) figures a DeInit process of the link created in (2)

```
ISTData* pData=(ISTData*)dat;
DEBUGMSG (ZONE_FUNC, (DTAG TEXT("IST++\r\n")));
if(!CeSetThreadPriority(GetCurrentThread(),pData->nPriority)){
  DEBUGMSG (ZONE_ERROR, (DTAG TEXT("IST: Set Priority Error: %d\r\n"),GetLastError()));
  return -1;
}
// connect "hardware" interrupt with "software" event
// link creation showed as (2) in the image
if(!InterruptInitialize(pData->sysIntr, pData->hEvent, NULL, 0)){
  DEBUGMSG (ZONE_ERROR, (DTAG TEXT("IST: Intr Init Error: %d\r\n"),GetLastError()));
  return -1;
}
// main loop
// pData->bAbort is throwen when Thread is stopped or killed (DeInit process)
while(!pData->bAbort){
  // wait for the interrupt event
  DEBUGMSG (ZONE_FUNC, (DTAG TEXT("IST: Waiting for Interrupt...\r\n")));
  // wait till interrupt event will appear
  WaitForSingleObject(pData->hEvent, INFINITE);
  if(pData->bAbort){
    // pData->bAbort is throwen by the DeInit function
    // abort function showed as (3) in the image
    DEBUGMSG (ZONE_FUNC, (DTAG TEXT("IST: abort\r\n")));
    InterruptDone(pData->sysIntr);
    break;
  }
   // handle the interrupt ==========================
  DEBUGMSG (ZONE_FUNC, (DTAG TEXT("IST: Interrupt Event!\r\n")));

  // <begin> of user interrupt handle

```

```
    // <end> of user interrupt handle

    InterruptDone(pData->sysIntr);
  } // end main loop
  DEBUGMSG (ZONE_FUNC, (DTAG TEXT("IST--\r\n")));
  return 0;
}
```

5.3.3.4 int Init ()

Initializes access. Allocates and maps MCS4 region. If needed creates starts Interrupt Service Thread.

Returns:

1 : if success

0 : else

```
  DEBUGMSG(ZONE_FUNC, (DTAG TEXT("Init++\r\n")));
  //
  if(!IsSleevePresent()){
    DEBUGMSG (ZONE_INIT | ZONE_FUNC | ZONE_ERROR,
              (DTAG TEXT("Init failure. Sleeve not present.\r\n")));
    SetLastError(ERROR_DEV_NOT_EXIST);
    return 0;  // Fail init
  }

  // allocate and map mcs4 region
  pFpgaRegs = (PULONG)VirtualAlloc(NULL, FPGA_IO_SPACE, MEM_RESERVE, PAGE_NOACCESS);
  if(pFpgaRegs==NULL){
    DEBUGMSG (ZONE_INIT | ZONE_FUNC | ZONE_ERROR,
              (DTAG TEXT("Init failure (VirtualAlloc). Out of memory\r\n")));
    SetLastError(ERROR_OUTOFMEMORY);
    return 0;  // Fail init
  }
  if(!VirtualCopy((VOID*)pFpgaRegs,(VOID*)(FPGA_IO_BASE/256), FPGA_IO_SPACE,
                  PAGE_READWRITE | PAGE_NOCACHE | PAGE_PHYSICAL)){
    DEBUGMSG (ZONE_INIT | ZONE_FUNC | ZONE_ERROR,
              (DTAG TEXT("Init failure (VirtualCopy).\r\n")));
    return 0;  // Fail init
  }

  // Power On
  SleevePower();

  if(FPGA_INT_ENABLE){
    // Starting IST
    mISTData.bAbort = FALSE;
    mISTData.sysIntr = SYSINTR_OPT;
    mISTData.nPriority = FPGA_IST_PRIORITY;
    // Create Event Handler
    mISTData.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
    // Create Thread Handler with the functionality of FPGA_IST and the mISTData parameters
    mISTData.hThread = CreateThread(NULL, 0, &FPGA_IST, &mISTData, 0, NULL);
    if(mISTData.hThread==NULL){
      DEBUGMSG(ZONE_ERROR, (DTAG TEXT("Init: Error creating IST.\r\n")));
    }
    PPC_SET_INTERRUPT_ENABLED(NULL,TRUE);
  }

  DEBUGMSG (ZONE_FUNC, (DTAG TEXT("Init--\r\n")));
  return 1;
};
```

5.3.3.5 int Read (UINT *pipe*, PUSHORT *pBuffer*, UINT *ucb*)

Read(pipe, &(Buffer[0]), 64) reads 64 short words from the FPGA module at address pipe and stores it in Buffer.

Parameters:

pipe : address to select memory mapped region (range: 0h - Fh)

pBuffer : array to store the data words read

ucb : number of words (a word is 16 bits)

Returns:

0 : if bad pointer
1 : if success

```
 DEBUGMSG(ZONE_FUNC, (DTAG TEXT("Read++\r\n")));
 if(IsBadWritePtr(pBuffer,2*ucb)){
  DEBUGMSG(ZONE_ERROR, (DTAG TEXT("Read: Bad Pointer\r\n")));
  return 0;
 }
 for(UINT i=0;i<ucb;i++){
  pBuffer[i] = (USHORT)(pFpgaRegs[pipe]&0x0000ffff);
 }
 DEBUGMSG(ZONE_FUNC, (DTAG TEXT("Read--\r\n")));
 return 1;
};
```

5.3.3.6 void ReadStatusReg (void) `[inline]`

Prints the content of FPGA status register into the shell

```
    USHORT ReadData[1];

    Read(STATUS_REG,&ReadData[0],1);
    printf("DATA: %Xh \n",ReadData[0]);
};
```

5.3.3.7 void Slower (void) `[inline]`

Inline directive to delay a process (inline "for" - loop)

```
 ULONG SlowingVar = 0;
 for (UINT i=0;i<SLOW_FACTOR;i++){
  SlowingVar++;
 }
};
```

5.3.3.8 int SlowWrite (UINT *pipe*, PUSHORT *pBuffer*, UINT *ucb*)

SlowWrite(pipe, &(Buffer[0]), 64) writes 64 short words from Buffer to the FPGA module. The address pipe parameter is assigned to 4-bit width external address pins.
The write process is delayed by a factor SLOW_FACTOR (non functional wait loop).

Parameters:

pipe : address to select memory mapped region (range: 0h - Fh)

pBuffer : array that contains the data words for writing

ucb : number of words (a word is 16 bits)

Returns:

0 : if bad pointer
1 : if success

```
    ULONG SlowingVar = 0;

    DEBUGMSG(ZONE_FUNC, (DTAG TEXT("FPGA_Write++\r\n")));
    if(IsBadReadPtr(pBuffer,2*ucb)){
        DEBUGMSG(ZONE_ERROR, (DTAG TEXT("Write: Bad Pointer\r\n")));
```

```
                return 0;
        }
        for(UINT i=0;i<ucb;i++){
                for (UINT i=0;i<SLOW_FACTOR;i++){
                        SlowingVar++;
                }
                pFpgaRegs[pipe] = (ULONG)pBuffer[i];
        }
        DEBUGMSG(ZONE_FUNC, (DTAG TEXT("Write--\r\n")));
        return 1;
};
```

5.3.3.9 int Write (UINT *pipe*, PUSHORT *pBuffer*, UINT *ucb*)

Write(pipe, &(Buffer[0]), 64) writes 64 short words from Buffer to the FPGA module. The address pipe parameter is assigned to 4-bit width external address pins.

Parameters:

pipe : address to select memory mapped region (range: 0h - Fh)

pBuffer : array that contains the data words for writing

ucb : number of words (a word is 16 bits)

Returns:

0 : if bad pointer (i.e. Buffer Overflow)
1 : if success

```
  DEBUGMSG(ZONE_FUNC, (DTAG TEXT("FPGA_Write++\r\n")));
  if(IsBadReadPtr(pBuffer,2*ucb)){
    DEBUGMSG(ZONE_ERROR, (DTAG TEXT("Write: Bad Pointer\r\n")));
    return 0;
  }
  for(UINT i=0;i<ucb;i++){
    pFpgaRegs[pipe] = (ULONG)pBuffer[i];
  }
  DEBUGMSG(ZONE_FUNC, (DTAG TEXT("Write--\r\n")));
  return 1;
};
```

5.4 FPGADriver.h File Reference

5.4.1 Detailed Description

Author:

Matthias Dyer
Roman Plessl

Version:

1.7

Date:

2004/04/05

Id

FPGADriver.h,v 1.7 2004/04/21 18:54:58 rplessl Exp

Data Structures

- struct ISTData
  Interrupt Service Thread Struct.
- class CFpga
  Functions to access the FPGAModule.

Defines

- #define FPGA_API __declspec(dllimport)
  export DLL

Typedefs

- typedef ISTData **ISTData**

5.4.2 Define Documentation

5.4.2.1 #define FPGA_API __declspec(dllimport)

The following ifdef block is the standard way of creating macros which make exporting from a DLL simpler. All files within this DLL are compiled with the FPGADRIVER_EXPORTS symbol defined on the command line. this symbol should not be defined on any project that uses this DLL. This way any other project whose source files include this file see FPGA_API functions as being imported from a DLL, wheras this DLL sees symbols defined with this macro as being exported.

6 Page Documentation

6.1 Todo List

File FPGADriver.cpp Interruptenable as parameter e.g. in constructor

Index

BLOCKING_REG
    EmbeddedMachine.h, 12

CFpga, 1
    FPGADriver.cpp, 15

DATA_REG
    EmbeddedMachine.h, 12
DataStreamWriter.cpp, 2
DataStreamWriter.cpp
    WinMain, 3
DllMain
    FPGADriver.cpp, 15
DTAG
    FPGADriver.cpp, 13

EmbeddedMachine.h, 11
EmbeddedMachine.h
    BLOCKING_REG, 12
    DATA_REG, 12
    STATUS_REG, 12

FPGA_API
    FPGADriver.h, 21
FPGA_IST
    FPGADriver.cpp, 16
FPGA_IST_PRIORITY
    FPGADriver.cpp, 13
FPGADriver.cpp, 12
    CFpga, 15
    DllMain, 15
    DTAG, 13
    FPGA_IST, 16
    FPGA_IST_PRIORITY, 13
    Init, 18
    Read, 18
    ReadStatusReg, 19
    Slower, 19
    SlowWrite, 19
    Write, 20
    ZONE_DRVCALLS, 14
    ZONE_ERROR, 14
    ZONE_FUNC, 14
    ZONE_INIT, 15
    ZONE_WARNING, 15
FPGADriver.h, 20
    FPGA_API, 21

Init
    FPGADriver.cpp, 18
ISTData, 2

Read
    FPGADriver.cpp, 18
ReadStatusReg
    FPGADriver.cpp, 19

Slower
    FPGADriver.cpp, 19
SlowWrite
    FPGADriver.cpp, 19
STATUS_REG
    EmbeddedMachine.h, 12

WinMain
    DataStreamWriter.cpp, 3
Write
    FPGADriver.cpp, 20

ZONE_DRVCALLS
    FPGADriver.cpp, 14
ZONE_ERROR
    FPGADriver.cpp, 14
ZONE_FUNC
    FPGADriver.cpp, 14
ZONE_INIT
    FPGADriver.cpp, 15
ZONE_WARNING
    FPGADriver.cpp, 15

C

Tools

1. bitstreambuilder.pl on page 148
2. streamer.pl on page 152
3. listener.pl on page 154
4. EdifToXML.pl on page 156
5. XMLToEdif.pl on page 159

Roman Plessl > **bistreambuilder.pl-1.4**

- NAME
- DESCRIPTION
- SYNOPSIS
  - MODIFY BITSTREAM
  - RESET
  - ERASING
  - ERASING SECTOR
  - ACTIVATE SLAVE PARALLEL MODE
- SECTOR MATCHING IN FLASH
- AUTHORS
- COPYRIGHT
- COMPATIBILITY

NAME

bitstreambuilder.pl

DESCRIPTION

The primary idea of this program is the modifcation of XILINX FPGA bitstreams for SPARTAN II devices in such a manner that they can be stored in a FLASH and used for reconfigure a FPGA afterwards.

A second application field is the generation of command bitstreams for the communication between IPAQ and the BTNodeBoard (FPGA and CPLD)

bitstreambuilder.pl has the following operating modes:

* modifying of a bitstream generated for XILINX SPARTAN II FPGAs
* generate command bitstreams for contolling the BTNodeFPGA Board

Version: $Revision: 1.4 $

SYNOPSIS

usage : bitstreambuilder.pl -h|-?|--help prints this help text

```
         bitstreambuilder.pl -r|--reset <filename>
                                       generates bitfile for reseting
                                       the flash memory
```

```
         bitstreambuilder.pl -e|--erase <filename>
                                       generates bitfile for erasing
                                       the flash memory
```

```
         bitstreambuilder.pl -r|--erase_sector <filename> <sector>
                                        generates bitfile for erasing
                                        an sector of the flash memory
```

```
         bitstreambuilder.pl -m|--modify <filename> <sector>
                                        modifies bitstream for upload
                                        the configuration to FLASH
```

```
         bitstreambuilder.pl -s|--slaveparallel <filename> <sector>
                                        generates bitfile for active
                                        the slave parallel mode
```

MODIFY BITSTREAM

Modifies Bitstream in the following way:

```
     Original Bitstream   Modified Bitstream

        .. .. .. ..
        .. .. .. ..           CA FE AF 5x        Additional Synchronisation Word
        .. .. .. ..           xx xx xx xx        Length of Bitstream (Bytes)
        FF FF FF FF           FF FF FF FF        Synchronisation Word 1
        AA 99 55 66           AA 99 55 66        Synchronisation Word 2
                              00 00 00 00        Additional Synchronization Word
        30 00 80 01           30 00 80 01        First Real Data
        .. .. .. ..          .. .. .. ..
        30 00 00 01          30 00 00 01        Write To CRC
        xx xx xx xx          xx xx xx xx        CRC Value
        .. .. .. ..          .. .. .. ..
        30 00 80 01          30 00 80 01        Write next 4 Bytes to CMD Register
        00 00 00 05          00 00 00 05        Begin Start-Up Sequence
        .. .. .. ..          .. .. .. ..
        30 00 00 01          30 00 00 01        Final Write To CRC
        xx xx xx xx          xx xx xx xx        Final CRC Value
        00 00 00 00          00 00 00 00        16 zero Bytes for Slave Parallel
        00 00 00 00          00 00 00 00
        00 00 00 00          00 00 00 00
        00 00 00 00          00 00 00 00
```

Sector handling:

```
   SECTOR         COMMAND

     0          CA FE AF 50
     1          CA FE AF 51
     2          CA FE AF 52
     3          CA FE AF 53
     4          CA FE AF 54
     5          CA FE AF 55
     6          CA FE AF 56
     7          CA FE AF 57
     8          CA FE AF 58
     9          CA FE AF 59
     A          CA FE AF 5A
     B          CA FE AF 5B
     C          CA FE AF 5C
     D          CA FE AF 5D
     E          CA FE AF 5E
     F          CA FE AF 5F
```

Output:

```
   --> generating fpga .bit file for slave/parallel $BITSTREAM_FILE_MOD ..
     > reading file = $BITSTREAM_FILE
```

```
     > writing command words (CA FE AF 5x) [x <-- sector]\n";
     >         stream length (XX XX XX XX)
     >         synchro words (FF FF FF FF)
     >         synchro words (AA 99 55 66)
     >         synchro words (00 00 00 00)
```

RESET

Output:

```
    --> generating fpga .bit file for reseting the FLASH ...
```

```
      > reseting command words  (CA FE AF 10)
      >            dummy words  (00 00 00 00)
```

ERASING

Output:

```
    --> generating fpga .bit file for erasing the FLASH ...\n";
```

```
      > erasing command words   (CA FE AF 90)\n";
      >           dummy words   (00 00 00 00)\n";
```

ERASING SECTOR

Output:

```
    --> generating fpga .bit file for erasing the FLASH ...\n";
```

```
      > erasing command words   (CA FE AF Ax [x <-- sector]\n";
      >           dummy words   (00 00 00 00)\n";
```

Sector handling:

```
SECTOR          COMMAND
=========================
   0          CA FE AF A0
   1          CA FE AF A1
   2          CA FE AF A2
   3          CA FE AF A3
   4          CA FE AF A4
   5          CA FE AF A5
   6          CA FE AF A6
   7          CA FE AF A7
   8          CA FE AF A8
   9          CA FE AF A9
   A          CA FE AF AA
   B          CA FE AF AB
   C          CA FE AF AC
   D          CA FE AF AD
   E          CA FE AF AE
   F          CA FE AF AF
```

ACTIVATE SLAVE PARALLEL MODE

Output:

```
  --> generating fpga .bit file for activating slave/parallel ...
```

```
    > slave parallel command words  (CA FE AF Ex) [x <-- sector]
    >                  dummy words  (00 00 00 00)
```

Sector handling:

```
SECTOR          COMMAND
=========================
   0          CA FE AF E0
   1          CA FE AF E1
   2          CA FE AF E2
   3          CA FE AF E3
   4          CA FE AF E4
   5          CA FE AF E5
   6          CA FE AF E6
   7          CA FE AF E7
   8          CA FE AF E8
   9          CA FE AF E9
   A          CA FE AF EA
   B          CA FE AF EB
   C          CA FE AF EC
   D          CA FE AF ED
   E          CA FE AF EE
   F          CA FE AF EF
```

SECTOR MATCHING IN FLASH

A bitstream normally has a size of 164 KB and therefore needs 3 x 64 KB bytes for storage in the FLASH. The following sequence displays the relation between *SECTOR NUMBER* and used *FLASH SECTORS*

```
  SECTOR      USED FLASH SECTORS
 =========================
    0           0   1   2
    1           1   2   3
    2           2   3   4
    3           3   4   5
    4           4   5   6
    5           5   6   7
    6           6   7   8
    7           7   8   9
    8           8   9  10
    9           9  10  11
    A          10  11  12
    B          11  12  13
    C          12  13  14
    D          13  14  15
    E          14  15   0
    F          15   0   1
```

AUTHORS

Roman Plessl, rplessl@ee.ethz.ch.

COPYRIGHT

Copyright (C) 2004, Roman Plessl. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

COMPATIBILITY

Most of the code in this module has been stable since version $Revision: 1.4 $ Except for items indicated as *Experimental*, I do not expect functional changes which are not fully backwards compatible.

29 March 2004.

Roman Plessl > streamer.pl-1.13

NAME
DESCRIPTION
SYNOPSIS
    CONFIGURE SERIAL PORT
        baudrate
        parity
        databits
        stopbits
        handshake
    STREAM BINARY FILE
SEE ALSO
AUTHORS
COPYRIGHT
COMPATIBILITY

NAME

RS232 Streamer

DESCRIPTION

streamer.pl is a small application written in Perl for configuring and using the serial interface (rs232) of a pc using windows.

streamer.pl has the following operating modes:

* configuration of the serial interface (baudrate, data- and controlflow, handshaking)

* streaming a binary file to the serial port

Version: $Revision: 1.13 $

SYNOPSIS

```
  require 5.003;
  use Win32::SerialPort qw( :STAT 0.19 );
```

```
  usage : perl streamer.pl -h|-?|--help            prints this help text
```

```
          perl streamer.pl   -c|--config
                           [-b|--baud <baudrate>]
                           [-d|--databits <databits>]
                           [-p|--parity <parity>]
                           [-f|--flowcontrol|--handshake <handshake]
                           [-s|--stopbits <stopbits>]
                                                 generates configuration file
```

```
          perl streamer.pl <filename>             uploads <filename> over the
                                                 serial port $PortName
```

CONFIGURE SERIAL PORT

The default serial outport of streamer.pl is the first serial port called COM1 in the Windows environment. Other COM ports can be used by changing the variable `$PortName` in the script.

The configuration is stored in a file called *COMx_rs232.cfg*.

The following values are allowed to the configuration, defaults are:

```
   my $baudrate = 9600;
   my $databits = 8;
   my $parity   = "none";
   my $handshake= "none";
   my $stopbits = 1;
```

baudrate

Any legal value.

parity

One of the following: ``none'', ``odd'', ``even'', ``mark'', ``space''. If you select anything except ``none'', you will need to set parity_enable in the script.

databits

An integer from 5 to 8.

stopbits

Legal values are 1, 1.5, and 2. But 1.5 only works with 5 databits, 2 does not work with 5 databits, and other combinations may not work on all hardware if parity is also used.

handshake

The handshake setting is recommended but no longer required. Select one of the following: ``none'', ``rts'', ``xoff'', ``dtr''.

STREAM BINARY FILE

In this mode the file `filename` is opened and streamed to the serial port (configurated with the config modus). A simple progress bar shows the streamed position in the input file (unit 10 KB).

SEE ALSO

Win32::SerialPort - Bill Birthsel Win32 API for serial ports

Win32::CommPort - the low-level API calls which supports the Serial Port module

AUTHORS

Roman Plessl, rplessl@ee.ethz.ch.

COPYRIGHT

Copyright (C) 2004, Roman Plessl. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

COMPATIBILITY

Most of the code in this module has been stable since version $Revision: 1.13 $ Except for items indicated as *Experimental*, I do not expect functional changes which are not fully backwards compatible.

03 April 2004.

Roman Plessl > **listener.pl-1.4**

- NAME
- DESCRIPTION
- SYNOPSIS
  - CONFIGURE SERIAL PORT
    - baudrate
    - parity
    - databits
    - stopbits
    - handshake
  - RECEIVE BINARY FILE
- SEE ALSO
- AUTHORS
- COPYRIGHT
- COMPATIBILITY

NAME

RS232 Listener

DESCRIPTION

listener.pl is a small application written in Perl for configuring and using the serial interface (rs232) of a pc using windows. listener.pl is the counterpart to streamer.pl which is a file streamer over the serial port.

lister.pl has the following operating modes:

* configuration of the serial interface (baudrate, data- and controlflow, handshaking)

* receiving of a binary file from the serial port

Version: $Revision: 1.4 $

SYNOPSIS

```
require 5.003;
use Win32::SerialPort qw( :STAT 0.19 );
```

```
usage : perl listener.pl -h|-?|--help            prints this help text
```

```
        perl listener.pl   -c|--config
                          [-b|--baud <baudrate>]
                          [-d|--databits <databits>]
                          [-p|--parity <parity>]
                          [-f|--flowcontrol|--handshake <handshake]
                          [-s|--stopbits <stopbits>]
                                               generates configuration file
```

```
        perl listener.pl <filename>            downloads <filename> from the
                                               serial port $PortName
```

CONFIGURE SERIAL PORT

The default serial outport of streamer.pl is the first serial port called COM1 in the Windows environment. Other COM ports can be used by changing the variable `$PortName` in the script.

The configuration is stored in a file called *COMx_rs232.cfg*.

The following values are allowed to the configuration, defaults are:

```
  my $baudrate = 9600;
  my $databits = 8;
  my $parity   = "none";
  my $handshake= "none";
  my $stopbits = 1;
```

baudrate

Any legal value.

parity

One of the following: ``none'', ``odd'', ``even'', ``mark'', ``space''. If you select anything except ``none'', you will need to set parity_enable in the script.

databits

An integer from 5 to 8.

stopbits

Legal values are 1, 1.5, and 2. But 1.5 only works with 5 databits, 2 does not work with 5 databits, and other combinations may not work on all hardware if parity is also used.

handshake

The handshake setting is recommended but no longer required. Select one of the following: ``none'', ``rts'', ``xoff'', ``dtr''.

RECEIVE BINARY FILE

In this mode the file `filename` is opened and received from the serial port (configurated with the config modus). A simple progress bar shows the streamed position at the input file (unit 10 KB).

The input read function is blocking; unfortunately, it seems that in some cases the blocking mechanism not always acts as expected, so an additional timeout counter is instanciated.

SEE ALSO

Win32::SerialPort - Bill Birthsel Win32 API for serial ports

Win32::CommPort - the low-level API calls which support the Serial Port module

---

AUTHORS

Roman Plessl, rplessl@ee.ethz.ch.

COPYRIGHT

Copyright (C) 2004, Roman Plessl. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

COMPATIBILITY

Most of the code in this module has been stable since version $Revision: 1.4 $. Except for items indicated as *Experimental*, I do not expect functional changes which are not fully backwards compatible.

03 April 2004.

Roman Plessl > **EdifToXML.pl-0.01**

NAME
SYNOPSIS
DESCRIPTION
    Internals
    Example
SEE ALSO
AUTHORS
COPYRIGHT
COMPATIBILITY

NAME

EdifToXML.pl

SYNOPSIS

```
usage : perl EdifToXML.pl -h|-?|--help
                                       prints this help text

        perl EdifToXML.pl <filename> > <XMLfilename.xml>
                                       transforms the EDIF <filename> netlist
                                       to a XMLed representation
```

DESCRIPTION

Transforms a EDIF file from harware synthesis (Synplify Pro) to XML for simpler parsing the netlist with known tools.

Afterwards the XML representation is used for parsing (build a DOM tree), modifying and amending additional netlist parts. The reason for this script is that many XML tools are given to read in and modfiy a XML DOM Tree.

The XML format is only a representation of the EDIF netlist, not further specified and thus **not** a standard.

XMLToEdif.pl is the counterpart for transforming a modified netlist to the original EDIF representation.

Internals

The following steps are executed:

Fixes of the edif output synthesis tools (XILINX xst)

```
   # toolflow with xst: vhdl --> ngc --> ngc2edif --> I<fix> -->edif --> edifXML
   if ($text =~ /(Xilinx ngc2edif)/g) {
       $text =~ s/([<])([\d]*?)([>])/&xstfixparantheses($1,$2,$3)/ge;
       $text =~ s/([&])([\d\w]*?)/&xstfixampersand($1,$2)/ge;
       $text =~ s/(ngc2edif)/&xstfixngc2edif($1)/ge;
   }
```

```
   sub xstfixparantheses()
     {
         return "(" . $_[1] . ")";
     }
   sub xstfixampersand()
     {
         return "xstfix" . $_[1];
     }
   sub xstfixngc2edif()
     {
         return "generatedbyngc2edif";
     }
```

Save the Bus-Parenthesis (xx:xx)

```
$text =~ s/([\(])([\d]*?:[\d]*?)([\)])/&savebus($1,$2,$3)/ge;
```

```
sub savebus()
  {
      return "__SAVEBUSSTART_" . $_[1] . "_SAVEBUSSTOP__";
  }
```

Save the Single Bits of a Bus (x)

```
$text =~ s/([\(])([\d]*?)([\)])/&savesignal($1,$2,$3)/ge;
```

```
sub savesignal()
  {
      return "__SAVESIGNALSTART_" . $_[1] . "_SAVESIGNALSTOP__";
  }
```

Build Stack for Counting ``(" and ``)" and Tranform EDIF to XML

```
 $text =~ s/([\)\(])/&tagmaker($1)/ge;
```

```
 sub tagmaker()
  {
     if ($_[0] eq '(') {
         $counter++;
         $string = "<" . $counter . ">";
         return $string;
     } else {
         return "</" . $counter-- . ">";
     }
   }
```

Restore the Single Bits of a Bus (x)

```
$text =~ s/(__SAVESIGNALSTART_)([\d]*?)(_SAVESIGNALSTOP__)/&restoresignal($1,$2,$3)/ge;
```

```
sub restoresignal()
  {
      return "(" . $_[1] . ")";
  }
```

Restore the Bus-Parenthesis (xx:xx)

```
$text =~ s/(__SAVEBUSSTART_)([\d]*?:[\d]*?)(_SAVEBUSSTOP__)/&restorebus($1,$2,$3)/ge;
```

```
sub restorebus()
  {
      return "(" . $_[1] . ")";
  }
```

Example

Synthesis Output as EDIF netlist:

```
(edif aes_enc_wrapper
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
   (written
     (timeStamp 2004 3 4 11 32 51)
     (author "Synplicity, Inc.")
     (program "Synplify Pro" (version "7.3.3, Build 024R"))
    )
  )
 (library VIRTEX
   (edifLevel 0)
   (technology (numberDefinition ))
   (cell RAMB4_S16 (cellType GENERIC)
   ...
```

XML representation:

```
 <edif> aes_enc_wrapper
 <edifVersion> 2 0 0</edifVersion>
 <edifLevel> 0</edifLevel>
 <keywordMap> <keywordLevel> 0</keywordLevel></keywordMap>
 <status>
   <written>
     <timeStamp> 2004 3 4 11 32 51</timeStamp>
     <author> "Synplicity, Inc."</author>
     <program> "Synplify Pro" <version> "7.3.3, Build 024R"</version></program>
    </written>
  </status>
 <library> VIRTEX
   <edifLevel> 0</edifLevel>
   <technology> <numberDefinition> </numberDefinition></technology>
```

```
<cell> RAMB4_S16 <cellType> GENERIC</cellType>
...
```

SEE ALSO

EDIF - Electronic Design Interchange Format (EDIF): http://www.edif.org/

XML - Extensible Markup Language (XML): http://www.w3.org/XML

LibXML - The XML C parser and toolkit of Gnome: http://www.xmlsoft.org/

XML::LibXML - Perl Binding for libxml2: http://search.cpan.org/dist/XML-LibXML/

AUTHORS

Roman Plessl, rplessl@ee.ethz.ch.

COPYRIGHT

Copyright (C) 2004, Roman Plessl. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

COMPATIBILITY

Most of the code in this module has been stable since version 0.01. Except for items indicated as *Experimental*, I do not expect functional changes which are not fully backwards compatible.

08 March 2004.

Roman Plessl > XMLToEdif.pl-0.01

- NAME
- SYNOPSIS
- DESCRIPTION
  - Internals
  - Example
- SEE ALSO
- AUTHORS
- COPYRIGHT
- COMPATIBILITY

NAME

XMLToEdif.pl

SYNOPSIS

```
usage : perl XMLToEdif.pl -h|-?|--help
                                       prints this help text

        perl XMLToEdif.pl <XMLfilename.xml> > <filename.edn>
                                       transforms the XML <XMLfilename.xml>
                                       netlist to a EDIFed representation
```

DESCRIPTION

Transforms a EDIF netlist represented in a XML format back to the usually used form with parentheses [``(" and ``)"].

The XML format is produced by the counterpart of this script, EdifToXML.pl. The format is only a representation of the EDIF netlist, not further specified and thus **not** a standard.

Internals

The following steps are executed:

Transform XML to EDIF

```
 $text =~ s/([<])([\/]?)([A-Za-z]*)([>])/&paranthesesmaker($1,$2,$3,$4)/ge;
```

```
 sub paranthesesmaker()
   {
      if ($_[1] ne '/') {
          my $string = "(" . $_[2];
          return $string;
      } else {
          return ")";
      }
  }
```

Example

XML representation of the synthesis output:

```
  <edif> aes_enc_wrapper
  <edifVersion> 2 0 0</edifVersion>
  <edifLevel> 0</edifLevel>
  <keywordMap> <keywordLevel> 0</keywordLevel></keywordMap>
  <status>
    <written>
      <timeStamp> 2004 3 4 11 32 51</timeStamp>
      <author> "Synplicity, Inc."</author>
      <program> "Synplify Pro" <version> "7.3.3, Build 024R"</version></program>
     </written>
   </status>
  <library> VIRTEX
    <edifLevel> 0</edifLevel>
    <technology> <numberDefinition> </numberDefinition></technology>
    <cell> RAMB4_S16 <cellType> GENERIC</cellType>
      ...
```

Retransformed to the EDIF representation:

```
 (edif aes_enc_wrapper
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (status
    (written
      (timeStamp 2004 3 4 11 32 51)
      (author "Synplicity, Inc.")
      (program "Synplify Pro" (version "7.3.3, Build 024R"))
     )
   )
  (library VIRTEX
    (edifLevel 0)
    (technology (numberDefinition ))
    (cell RAMB4_S16 (cellType GENERIC)
    ...
```

SEE ALSO

EDIF - Electronic Design Interchange Format (EDIF): http://www.edif.org/

XML - Extensible Markup Language (XML): http://www.w3.org/XML

LibXML - The XML C parser and toolkit of Gnome: http://www.xmlsoft.org/

XML::LibXML - Perl Binding for libxml2: http://search.cpan.org/dist/XML-LibXML/

AUTHORS

Roman Plessl, rplessl@ee.ethz.ch.

COPYRIGHT

Copyright (C) 2004, Roman Plessl. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

COMPATIBILITY

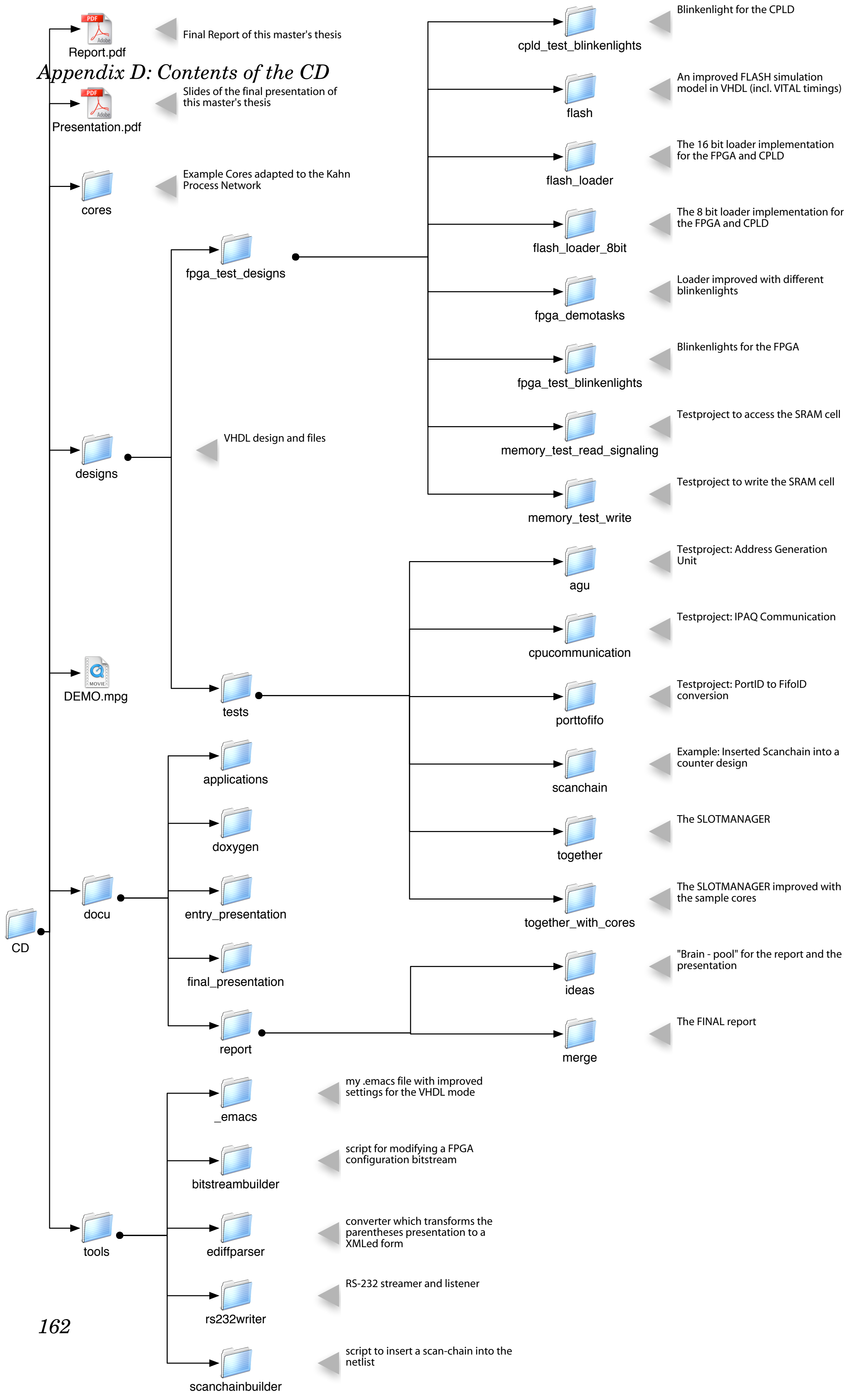
Most of the code in this module has been stable since version 0.01. Except for items indicated as *Experimental*, I do not expect functional changes which are not fully backwards compatible.

08 March 2004.

D

Contents of the CD

Appendix D: Contents of the CD



CD
Report.pdf
Final Report of this master's thesis
Presentation.pdf
Slides of the final presentation of this master's thesis
cores
Example Cores adapted to the Kahn Process Network
designs
VHDL design and files
fpga_test_designs
cpld_test_blinkenlights
Blinkenlight for the CPLD
flash
An improved FLASH simulation model in VHDL (incl. VITAL timings)
flash_loader
The 16 bit loader implementation for the FPGA and CPLD
flash_loader_8bit
The 8 bit loader implementation for the FPGA and CPLD
fpga_demotasks
Loader improved with different blinkenlights
fpga_test_blinkenlights
Blinkenlights for the FPGA
memory_test_read_signaling
Testproject to access the SRAM cell
memory_test_write
Testproject to write the SRAM cell
tests
agu
Testproject: Address Generation Unit
cpucommunication
Testproject: IPAQ Communication
porttofifo
Testproject: PortID to FifoID conversion
scanchain
Example: Inserted Scanchain into a counter design
together
The SLOTMANAGER
together_with_cores
The SLOTMANAGER improved with the sample cores
DEMO.mpg
docu
applications
doxygen
entry_presentation
final_presentation
report
ideas
"Brain - pool" for the report and the presentation
merge
The FINAL report
tools
_emacs
my .emacs file with improved settings for the VHDL mode
bitstreambuilder
script for modifying a FPGA configuration bitstream
ediffparser
converter which transforms the parentheses presentation to a XMLed form
rs232writer
RS-232 streamer and listener
scanchainbuilder
script to insert a scan-chain into the netlist

E

Glossary

ASIC Application Specific Integrated Circuits - Intergrated Circuit which is specialized to run an algorihm or a system at high-speed or low-power.

ATPG Automatically Test Pattern Generation

BIST Build in Self Test

BlockRAM Integrated SRAM cell included on a FPGA chip.

BlockRAM SRAM cell included in Xilinx FPGAs additional to the flip-flops.

BRAM see BlockRAM

C++ C++ is a general purpose computer programming language. It is a static-type-checking language supporting procedural programming, data abstraction, object-oriented programming and generic programming.

CLB Configurable Logic Blocks - Each *Xilinx SPARTAN*^{TM} *II* CLB consists of 2 slices. Each slice contains:

- two 4-input lookup tables (LUTs) which can be used to implement a 4-input and a 1-output combinational function
- two Carry and Control Logic block for faster arithmetic operations
- two storage elements that can be configured as edge-triggered D flip-flops or level-sensitive latches.

CPLD Complex Programmable Logic Device - is a reconfigurable device which contains regularly structures of AND-OR-plans and some flip-flops. CPLD are usually used as «glue-logic» and are built by a technology to keep their configuration.

DFT Design for Testability

doxygen Doxygen is a documentation system for C++, C, Java and many more programming languages. It can generate an on-line documentation browser (in

HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files.

DSP Digital Signal Processor - An programmable processor which is optimised for calculations in streaming applications.

EDIF Electronic Design Interchange Format [31]

EEPROM Electrically-Erasable Programmable Read-Only Memory, is a non-volatile storage chip.

FIFO «First-In-First-Out» - Algorithm often used at memory structures or for a scheduling algorihm. The first item received will also be the first which leaves the system.

FIFO First In First Out - A storage method that retrieves the item stored for the longest time. Contrast with LIFO (Last In First Out). Also known as circular buffer in literature.

FIFO First-In-First-Out - often used for data storage in memory cells or for scheduling algorithms.

FLASH Flash memory is a form of EEPROM that allows multiple memory locations to be erased or written in one programming operation.

FPGA Field Programmable Gate Arrays - like an ASIC, but the system is reconfigurable and so usable in multipurpose applications. The drawback versus ASICs are the power consumption and the lesser density on a device. At the moment there are only two competitors: Altera and Xilinx.

FSM Final State Machine - Mealy or Moore automata.

glue-logic A simple logic circuit that is used to connect complex logic circuits together. At the printed circuit board (PCB) level, glue logic may be implemented with simple glue chips that contain a few gates all the way to programmable logic devices (see CPLD).

ISE Integrated Software Environment. A software environment provided by *Xilinx* to synthesize and implement designs for their configurable logic devices (FPGAs, CPLDs). It also includes tools to perform timing analyses and to view the designs graphically.

JTAG Joint Test Action Group has done a specification for performing boundary-scan hardware testing at the IC level. In 1990 this specification resulted in IEEE 1149.1, a standard which has established the details of access to any chip with a so-called JTAG port.

KPN Kahn Process Networks

LUT Look Up Table used for combinational logic after mapping the net-list to real devices. LUT-4 (means Look Up Table with 4 input signal and 1 output.)

LUT Look-Up-Table - An array or matrix of values that contains data which is looked for. In FPGA devices LUTs are used for logic function calculations. In the «Slotmanager» the lowest part of the external SRAM memory is also named LUT because of the FIFO control structures which are stored there.

Mealy Mealy model of a FSM: The next state and the output function of a Mealy FSM depends on the current state, the state transition function of this state and the input datas. The automate can be modelled by these equations:

$$o(k) = g(i(k), s(k))$$

$$s(k+1) = f(i(k), s(k))$$

g is termed the output function and f transition function. s is the state transition function.

Moore Moore model of a FSM: The next state of a Moore FSM depends on the current state, the transition function and the input data. The output depends only on the current state but not on the input data. The automate can be modelled by these equations:

$$o(k) = g(s(k))$$

$$s(k+1) = f(i(k), s(k))$$

g is termed the output function and f transition function. s is the state transition function.

MSB Most Significant Bit - In a byte, every bit has a value based upon the bit's position in the byte. The bit which has the largest value is called the most significant bit.

PCB Printed Circuit Board - interconnects electronic components without discrete wires.

SDF Synchronous Data Flow as described in section

SDF Synchronous Dataflow Language

stuck-at-test simple test-model for production failures used in VLSI chip designs

Synplify Synplify. A synthesizer for several FPGA and CPLD written by Synplicity.

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory of the ETH Zürich

transition Event that triggers a FSM to change its state. A FSM description is basically a combination of states and events plus a state transition table which ties them all together.

Verilog Verilog - A Hardware Description Language for electronic design and gate level simulation by Cadence Design Systems.

VHDL VHDL - Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. A large high-level VLSI design language with Ada-like syntax. The DoD standard for hardware description, now standardised as IEEE 1076.

F

Bibliography

[1] Advanced Micro Devices. *AMD Am29LV081B*. AMD Am29LV081B, 8 Megabit (1 Mb x 8- Bit), CMOS 3.0 Volt only Uniform Sector Flash Memory: `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21525.pdf`.

[2] AMIC Technology. *AMIC LP62S16256E-I Series*. AMIC LP62S16256E-I Series, 256K X 16 BIT LOW VOLTAGE CMOS SRAM: `http://www.amictechnology.com/pdf/LP62S16256E.pdf`.

[3] T. Basten and J. Hoogerbrugge. Efficient Execution of Process Networks. In *Communicating Process Architectures – 2001*. IOS Press, 2001.

[4] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996. ISBN 0792397223.

[5] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A self-reconfiguring platform. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 565–574. LNCS 2778, Springer-Verlag, 2003. `http://www.springerlink.com/openurl.asp?genre=issue&issn=0302-9743&volume=2778`.

[6] Brigham Young University. *JHDL Home Page*. webpage, 2003. `http://www.jhdl.org/`.

[7] BTNode Project. *BTNode Homepage*. `http://www.btnode.ethz.ch`.

[8] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memeory using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993.

[9] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 605–614. LNCS 1896, Springer-Verlag, 2000. http://springerlink.metapress.com/link.asp?id=e9krabjcf4tlcdk0.

[10] DZ - Microelectronic Design Center, ETH Zürich. *Coding Style and Naming Conventions for VHDL*. http://dz.ee.ethz.ch/support/ic/vhdl.

[11] Stephen A. Edwards. *Languages for digital embedded systems*. Kluwer Academic Publications, 2000.

[12] Peter Fercher. *BTNode FPGA - a Mobile FPGA Module with Bluetooth Communication*. Master's thesis, ETH Zürich, February 2002. ftp://ftp.tik.ee.ethz.ch/pub/students/2002-2003-Wi/DA-2003-08.pdf.

[13] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *Lecture Notes in Computer Science, ESOP 2003*, volume 2618. Springer, 2003.

[14] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-Time Code. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326. ACM Press, 2002. http://portal.acm.org/citation.cfm?id=512567.

[15] Hewlett-Packard. *IPAQ h3950*.
Specifications
http://www.pocketpccentral.net/ipaq3970.htm
HP Website for H3950
http://h20000.www2.hp.com/bizsupport/TechSupport/Home.jsp?prodSeriesId=322896.

[16] Intel Corporation. *XScale® - 400 MHz Intel PXA250 Application Processor*.
Overview
http://www.intel.com/design/pca/prodbref/298620.htm
Datasheet (SRAM Interface in Chapter 6.8)
http://www.intel.com/design/pca/applicationsprocessors/manuals/278522-001.pdf.

[17] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Information Processing 74*, pages 471–475, 1974.

[18] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the eighth international workshop on Hardware/software codesign (CODES'00)*, pages 13–17. ACM Press, May 2000. http://portal.acm.org/citation.cfm?id=334015.

[19] Hubert Käslin. *Lecture Notes on VLSI II, D-ITET, ETH Zürich*, chapter Acquisition of Asynchronous Data, pages 47 – 68. Microelectronics Design Center of ETH Zürich, November 2002.

[20] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987. ISSN 0018-9340.

[21] T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *Lecture Notes in Computer Science, FPL 2003*, volume 2778, pages 595 – 605, September 2003.

[22] Microsoft Cooperation. *Windows CE*.
Microsoft Windows Embedded
http://msdn.microsoft.com/embedded/
Microsoft CE .NET
http://msdn.microsoft.com/embedded/downloads/ce.net/default.aspx
unofficial Windows CE site
http://www.windowsce.net.

[23] J. Mignolet, V. Nollet, P. Coene, S. Vernaldeand D. Verkest, and R. Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-On-Chip. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 986–991. IEEE, ISSN: 1530-1591, 2003. http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1253733&k2dockey=1253733@ieeecnfs.

[24] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Berkeley, University of California, 1995.

[25] Christian Plessl and Marco Platzner. Virtualization of Hardware – Introduction and Survey. In *Proceedings of the 4th International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, 2004. http://www.tik.ee.ethz.ch/~plessl.

[26] Ptolemy Project. *Ptolemy Project*. http://ptolemy.berkeley.edu.

[27] Herbert Walder, Samuel Nobs, and Marco Platzner. The XF-Board: A Prototyping Platform for Reconfigurable Hardware Operating Systems. In *Proceedings of the 4rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'04)*. CSREA Press, June 2004.

[28] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287. CSREA Press, June 2003.

[29] Timothy Wheeler, Paul Graham, Brent Nelson, and Brad Hutchings. Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 483–492. LNCS 2147, Springer-Verlag, 2001. http://link.springer.de/link/service/series/0558/tocs/t2147.htm.

[30] T. Wiangtong, P.Y.K Cheung, and W. Luk. A Unified Codesign Run-Time Environment for the UltraSONIC Reconfigurable Computer. In *Lecture Notes in Computer Science, FPL 2003*, volume 2778, pages 396 – 405, September 2003.

[31] Holger Willmer. EDIF - Electronic Design Interchange Format, September 1994. www.thkukuk.de/pg/Seminar/edif-holger.ps.

[32] Xilinx Inc. *Xilinx® CoolRunner™ XPLA CPLD*, .
CoolRunner Family:
http://direct.xilinx.com/bvdocs/publications/ds012.pdf
384 Macrocell CPLD:
http://direct.xilinx.com/bvdocs/publications/ds024.pdf.

[33] Xilinx Inc. *Xilinx® Spartan™ -II* FPGA, Data Sheets, .
Introduction and Ordering Information
http://direct.xilinx.com/bvdocs/publications/ds001_1.pdf
Functional Description:
http://direct.xilinx.com/bvdocs/publications/ds001_2.pdf
DC and Switching Characteristics:
http://direct.xilinx.com/bvdocs/publications/ds001_3.pdf
Pinout Tables:
http://direct.xilinx.com/bvdocs/publications/ds001_4.pdf.

[34] Xilinx Inc. *Configuration and Readback of the Spartan-II and Spartan-IIE Families*, March 2002. http://www.xilinx.com/bvdocs/appnotes/xapp176.pdf.

[35] Xilinx Inc. *Configuration and Readback of Virtex FPGAs Using (JTAG) Boundary Scan*, March 2002. http://www.xilinx.com/bvdocs/appnotes/xapp139.pdf.

[36] Xilinx Inc. *Development System Reference Guide*. Xilinx, Inc., ISE 6 edition, April 2003. http://toolbox.xilinx.com/docsan/xilinx6/books/docs/dev/dev.pdf.

[37] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Laura: Leiden Architecture Research and Exploration Tool. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 911–920. LNCS 2778, Springer-Verlag, 2003. http://springerlink.metapress.com/link.asp?id=r82hw8rm7jbl.