ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Sommer Semester 2004 Prof. Dr. L. Thiele

Semester Project

Embedded Task Machine with BTnode and FPGA

Clemens Lombriser
and
Marc André

Advisor: Matthias Dyer

Abstract

The aim of this semester project was to create an *Embedded Task Machine*. The system would provide a means of executing applications that are too large to fit on the available hardware. Such applications would be broken down into smaller tasks and a description of how they relate to each other. *Synchronous Data Flow* is used as the modelling language for the applications. The *Embedded Task Machine* then executes the tasks one after another on reconfigurable hardware. The system at hand consists of a task repository, a memory management module for the interexchanged data, an execution unit, and a scheduler.

This report describes the architecture and the the detailed design of the complete project.

Acknowledgment

We would like to thank Prof. Lothar Thiele for making it possible for us to work on this interesting project. We also would like to thank our tutor, Matthias Dyer, who supported us on any problems we had and Jan Beutel who helped out when Matthias was not present.

We would not have come this far if Roman Plessl would not have taken time for us in the last few weeks of his Master's Thesis. The discussions we had with him about the concepts of his work were very helpful and helped us a lot to start off our project.

Last but not least we also would like to thank the rest of the TIK-team for the various small things they did to support us.

Clemens Lombriser

Marc André

Problem Task (German)

Einleitung

Der BTnode und das FPGA Modul

An unserem Institut wurden in Forschungsarbeiten, sowie in früheren Semester- und Diplomarbeiten diverse Hardware entwickelt, welche im Zusammenhang mit eingebetteten Systemen und mobilem Computing stehen.

Zum einen gibt es den BTnode [1], ein kleiner "wireless node", mit einem Mikrocontroller und einem Bluetooth Modul. Der BTnode wird in verschiedenen Forschungsprojekten eingesetzt (z.B. Wireless Sensor Networks [2] oder Wearable Computing [3]).

Zum anderen wurde in einer Diplomarbeit ein FPGA Modul entwickelt, welches für den batteriebetriebenen Einsatz gedacht ist. Das Modul, bestehend aus einem Xilinx SpartanII FPGA, einem CPLD, SRAM und Flash ist so konstruiert, dass es mit dem BTnode über eine serielle Schnittstelle kommunizieren kann. Der Verbund von BTnode und FPGA erlaubt nun eine äusserst flexible Nutzung.

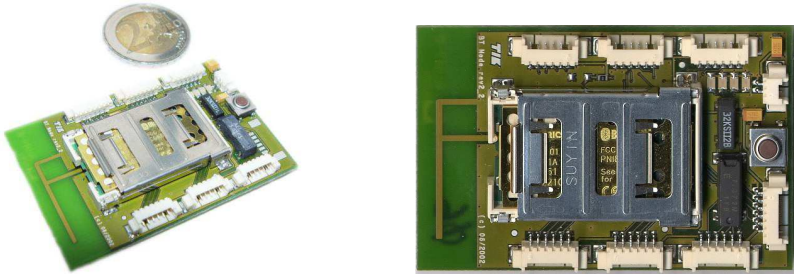

Abbildung 1: BTnode Rev 2.2 and FPGA Module

Es ist zum Beispiel möglich, dass mehrere FPGA Konfigurationen auf dem FPGA Modul gespeichert sind, welche dann auf Befehl des Mikrocon-

trollers auf dem BTnode in das FPGA geladen werden. So kann auch ein einfacher Scheduling Algorithmus auf dem BTnode implementiert werden, welcher den Ablauf einer Applikation bestehend aus mehreren Konfigurationen (Tasks) steuert. Eine Erweiterung dieses Scenarios ist, dass die Tasks nicht von Anfang an schon zur Verfügung stehen, sondern zuerst noch über die drahtlose Verbindung (Bluetooth) heruntergeladen werden müssen.

Eine Anwendung eines solchen Scenarios sind Sensor Netzwerke. Sensor Netzwerke bestehen aus einer Vielzahl von kommunizierenden Sensorknoten. Der BTnode und das FPGA Modul können zusammen einen solchen Sensorknoten darstellen. Der FPGA, welcher besonders effizient ist für die digitale Signalverarbeitung, sammelt Sensordaten und bereitet sie auf (Vorverarbeitung, Komprimierung, usw.). Der BTnode sendet dann auf Wunsch diese Daten zu einem Host.

Embedded Machine

Die Embedded Machine kommt ins Spiel, wenn eine Applikation als Ganzes zu gross für den FPGA ist. Die Applikation kann trotzdem ausgeführt werden, in dem man sie in kleinere Tasks aufteilt und definiert, wie die Task miteinander kommunizieren und interagieren. Diese Definition nennen wir *Koordinationssprache*. Eine Applikation besteht in diesem Fall aus zwei Teilen: aus der Koordinationsbeschreibung und aus den Tasks.

Wird die Koordinationssprache nicht kompiliert sondern interpretiert, geht die Information, wie die Tasks miteinander agieren, nicht verloren und kann zur Lade- und Laufzeit verwendet werden.

In [4] wurde das Prinzip der Embedded Machine für die Ausführung von Prozess Netzwerken auf eingebetteter rekonfiguierbarer Logik erklärt (siehe auch Abb. 2). Ein Prozess Netzwerk, besteht aus mehreren Prozessen (Tasks), welche in einem oder mehreren Slots im FPGA laufen können. Ein Scheduler, der in einer CPU läuft, entscheidet zur Laufzeit, welche Tasks als nächstes ausgeführt werden.

Die Realisierung der Embedded Machine auf dem FPGA Modul ist Hauptteil der Masterarbeit MA-2004-03 [5]. Das Ziel dort war es den Memorymanager und die Grundlagen für die Rekonfiguration der Taskslots zu implementieren. In dieser Arbeit wird jedoch ein IPAQ PDA als CPU verwendet.

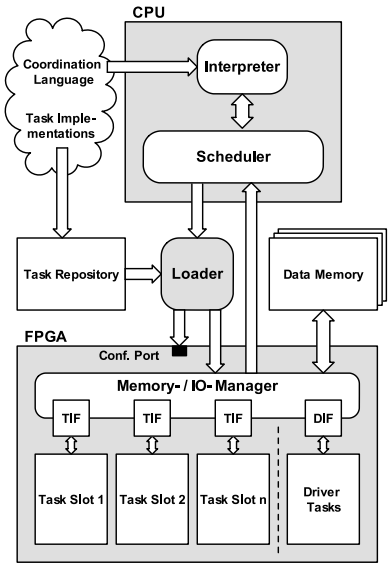

Abbildung 2: Embedded Machine Runtime System

Aufgabenstellung

Das Ziel dieser Arbeit ist den BTnode und das FPGA Modul zusammenzuschliessen und die entsprechende Services und Protokolle für die folgende Funktionen zu implementieren:

- Speichern einer FPGA-Konfiguration vom BTnode ins Flash vom FPGA Modul.
- (Partielle) Rekonfiguration des FPGAs aus dem Flash.
- Command/Event Kommunikation zwischen BTnode und FPGA.

Eine mögliche Verbindung zwischen BTnode und FPGA Modul ist die serielle Schnittstelle. Um die verschiedenen Dienste über eine serielle Verbindung zu multiplexen, braucht es ein durchdachtes Protokoll.

Es werden zur Zeit eine neue Version von BTnodes entwickelt. Diese haben zwar ein neues leistungsfähigeres Bluetooth Interface basieren aber auf demselben AVR Mikrokontroller wie die bisherigen BTnodes. Da die neuen BTnodes erst diesen Sommer erscheinen, soll diese Arbeit noch die bisherigen BTnodes verwenden.

Um Bluetooth mit den bisherigen BTnodes zu nutzen, wurde ein eigenes Dispatcher-basiertes Betriebssystem verwendet. Für die neuen BTnodes

wird das multitasking Betriebssystem Ethernut [6] verwendet. Dieses RTOS läuft auch tadellos auf den bisherigen BTnodes, es fehlt zur Zeit nur noch die Unterstützung der Bluetooth Funktionen.

Teilaufgaben

1. Erstellen Sie in den ersten zwei Wochen zusammen mit Ihrem Betreuer einen realistischen Zeitplan, welcher Meilensteine festlegt. überlegen Sie sich, wie Sie die Arbeit effizient aufteilen können.

2. Machen Sie sich mit den beiden Hardwareplattformen (BTnode und FPGA Modul) vertraut.

3. Arbeiten Sie sich in die Grundlagen der AVR Mikrokontroller bzw. FPGA/CPLD Programmierung ein.

4. Definieren Sie ein Protokoll für die Kommunikation zwischen BTnode und FPGA Modul und erstellen Sie ein Konzept, wie dieses im BTnode bzw. im CPLD/FPGA verarbeitet wird.

5. Implementieren Sie das Protokoll im BTnode und im FPGA Modul.

6. Definieren und Implementieren Sie ein oder mehrere Beispielsscenarien, mit welchen Sie das Funktionieren der Kommunikation demonstrieren können.

7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Semesterarbeit

Allgemeines

- Der Verlauf des Projektes Semesterarbeit 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.

- Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit 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. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (EMail).

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *2. Juli 2004* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.

Contents

1 Introduction **1**
1.1 Notation 3

2 System Architecture **5**
2.1 The Execution Model 5
2.1.1 Modules of the Execution Model 5
2.1.2 Mapping the modules to the hardware 6
2.2 MC-Protocol 7
2.2.1 Commands to the CPLD component 7
2.2.2 Commands to the FPGA 8
2.2.3 Response from FPGA module 10
2.3 FPGA-Protocol 11
2.3.1 Data flow 11
2.4 FIFO structure in the SRAM 12
2.5 Task slots in Flash 14

3 CPLD-Design **15**
3.1 CPLDTop 15
3.2 CPLDControl 16
3.3 RS232Core 16
3.3.1 Description 16
3.3.2 Usage 18
3.4 FlashCore 19
3.4.1 Description 19
3.4.2 Usage 19
3.4.3 The state machine 22
3.5 FPGAConfigurator 22
3.5.1 Description 22

3.5.2 Usage 22
3.6 FPGAComm 24
3.6.1 Description 24
3.6.2 Usage 24

4 FPGA-Design 26
4.1 FPGATop 26
4.2 FPGAControl 27
4.2.1 Description 27
4.3 CPLDComm 28
4.3.1 Description 28
4.3.2 Usage 29
4.4 OutputControl 31
4.5 SRAMCore 31
4.5.1 Description 31
4.5.2 Usage 31
4.6 MemoryControl 33
4.6.1 Description 33
4.6.2 Usage 34
4.7 FIFOControl 34
4.7.1 Description 34
4.7.2 Usage 36
4.8 TaskControl 37
4.8.1 Description 37
4.9 Task 38
4.9.1 Description 38
4.9.2 The TaskWrapper 40
4.9.3 TaskTestbench 41
4.9.4 Requirements for a task 41
4.10 Drivers 41
4.10.1 Input Driver 42
4.10.2 Output Driver 42
4.10.3 DriverWrapper 42

5 MC Software 44
5.1 Introduction 44
5.2 Synchronous Data Flow (SDF) 44

5.2.1 SDF formalism . . . 46
5.2.2 Scheduling . . . 46
5.3 Ethernut . . . 49
5.4 User Interface . . . 49
5.5 Messages . . . 52
5.6 Scheduler . . . 53
5.6.1 Data structures . . . 53
5.6.2 Communication with the BTnodeFPGA board . . . 55
5.7 Loading tasks . . . 55

6 Evaluation **57**
6.1 Time Measurements . . . 57
6.2 Power Measurements . . . 58
6.3 Demo application . . . 58
6.3.1 Setup . . . 59
6.3.2 Running the Demo . . . 60

7 Summary **62**

A How to start the Embedded Task Machine **63**
A.1 Cabling . . . 63
A.1.1 BTnodeFPGA . . . 63
A.1.2 BTnode . . . 63
A.2 Task setup . . . 66
A.3 Running the Scheduler . . . 66

B Future Work **67**
B.1 Partial Reconfiguration . . . 67
B.2 Several Task Slots . . . 68
B.3 Extension to Kahn Process Networks . . . 68
B.4 BRAM access . . . 69
B.5 Bluetooth . . . 69
B.6 Error handling . . . 69
B.7 Improved `FIFOControl` architecture . . . 69

Bibliography **72**

List of Figures

1 BTnode Rev 2.2 and FPGA Module VII
2 Embedded Machine Runtime System IX

1.1 Mealy automaton 3
1.2 Overview diagram of a state machine 4

2.1 Overview of the *Embedded Task Machine* 6
2.2 Wires between CPLD and FPGA 11
2.3 Timing details of a single write cycle from CPLD to FPGA . 12
2.4 SRAM FIFO structure 13
2.5 Organization of the task slots in the Flash 14

3.1 CPLD Architecture 16
3.2 FSM of the `CPLDControl` entity 17
3.3 Timing diagram of the Flash access 20
3.4 `FlashCore` FSM 21
3.5 Overview of the Finite State Machine of the `FlashCore` . . . 22
3.6 `FPGAConfigurator` FSM 23
3.7 `FPGAConfigurator` FSM Interface 23
3.8 Overview of module `FPGAComm` 25
3.9 `FPGAComm` FSM 25

4.1 FPGA Architecture 27
4.2 Symbolic state chart of `FPGAControl` 28
4.3 Overview of the module `CPLDComm` 29
4.4 `CPLDComm` FSM 30
4.5 Interface of the `SRAMCore` entity 32
4.6 `SRAMCore` handshake 33
4.7 Structure of the `MemoryControl` entity 34

4.8 `FIFOControl` interface 35
4.9 `TaskControl` state chart 38

5.1 SDF graph 45
5.2 Schedule sequence parser diagram 54
5.3 possible linkage of the `EXECUTION_STEP` structures 55

6.1 SDF graph of the demo application 59

A.1 `BTnodeFPGA` `BTnode` connection schematic 64
A.2 The connections used at the BTnodeFPGA-board 64
A.3 The connections used at the BTnode-board, rev2.2. 64
A.4 Interconnections when using the *Embedded Task Machine* . . 65
A.5 Development phase wiring 65

B.1 Proposed `FIFOControl` structure 70

List of Tables

2.1 Resources on the hardware modules 6
2.2 CPLD commands 8
2.3 List of response messages codes 11

3.1 `FlashCore` operation codes 20

4.1 SRAMCore handshake timings 32

5.1 Commands accepted by the RS232 software 50
5.2 Commands accepted by the BTnode user interface 56

6.1 time requirements of significant processes 57
6.2 power requirements of significant processes 58

A.1 `BTnode` UART settings 66
A.2 `BTnodeFPGA` UART settings 66

Chapter 1

Introduction

The next generation computing environment is called *ubiquitous computing*. It will substantially change the way people interact with computers, because any person will continuously be interacting with hundreds of nearby computers, which are interconnected and not visible to the user. Computers will be assisting people requiring no or just minimal interaction over new kinds interfaces that are subject of current research.

In order to invisibly integrate computers or at least not impeding into the environment, devices have to be small. This requirement forces a designer of a system to reduce the number of components to a minimum, and to use every module optimally not only in terms of area, but also over time. Reconfigurable hardware allows to reduce the size of the hardware by executing only the parts that are actually needed at every point in time. Other hardware configurations can be temporally stored using much less area on a mass storage device.

The aim of this thesis is to develop the software for a prototype system which can accept applications over a wireless connection and execute them. If the applications require more hardware over time than the system can provide permanently, the applications will be split into a number of tasks which then will be executed sequentially on the reconfigurable hardware part.

The execution model for the applications for the *Embedded Task Machine* defines an application to be a set of tasks which do some kind of operation on

data, and a description of how this data is passed from one task to another. This description is given in some *coordination language*.

The *Execution Model* of the system specifies four main entities: The *task repository*, which holds the different tasks of the application, the *relation memory*, where the data communicated between the tasks is stored, the *task runner*, where the tasks are actually executed, and finally the *scheduler* which controls the sequence in which the tasks are executed.

The report at hand describes how the system has been implemented. It is organized in the following chapters:

1. **System Architecture**
   This chapter explains how the system is build up. It explains what the most important parts are and where they are placed on the hardware. Additionally it explains the protocols used on the external interfaces of the hardware modules.

2. **CPLD-Design**
   The CPLD is one of the two programmable modules on the `BTnodeFPGA` board. This chapter explains the entities placed on the CPLD and how they collaborate.

3. **FPGA-Design**
   The FPGA is the reconfigurable module and thus the heart of the system. This is the hardware on which the tasks of the application run. The chapter also explains the additional entities that are implemented on the FPGA to support the tasks.

4. **Microcontroller Software**
   The microcontroller is the "mind"of the whole system. It is used to schedule all the tasks and initializes the hardware for the applications. It also provides a user interface which can be used to describe the application using a language which builds a *Synchronous Data Flow* graph.

5. **Evaluation**
   Some measurements on timings and power consumption have been carried out on the implemented and running system. The results are presented in this chapter.

Additionally, a demo application has been implemented. The chapter explains what calculations are performed by the application and how it can be run on the system.

6. **Summary**
   This chapter concludes the report and summarizes the achievements of the project.

7. **How to Start the Embedded Task Machine**
   When using the system, the hardware has to be correctly interconnected. This chapter shows how it has to be done.

8. **Future Work**
   Some ideas to what can further be done with the system is described in this chapter.

1.1 Notation

All modules in the FPGA and the CPLD have been designed as finite state machines (FSM). Since the state machines are all explained along with this text, a uniform way to describe their interface is introduced here.

A state machine generally can be written as a Mealy Automaton. A symbolic view of a Mealy Automaton is show in Figure 1.1. This view of an automaton is used to introduce a diagram, which shows all involved signals of the state machine and their influence on the automaton. Figure 1.2 shows a detailed description of this diagram.

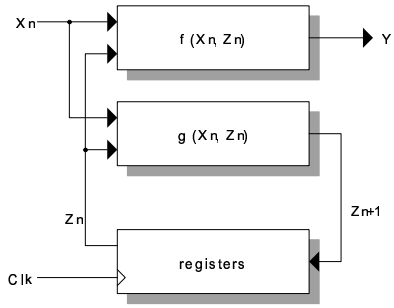

Figure 1.1: Mealy automaton

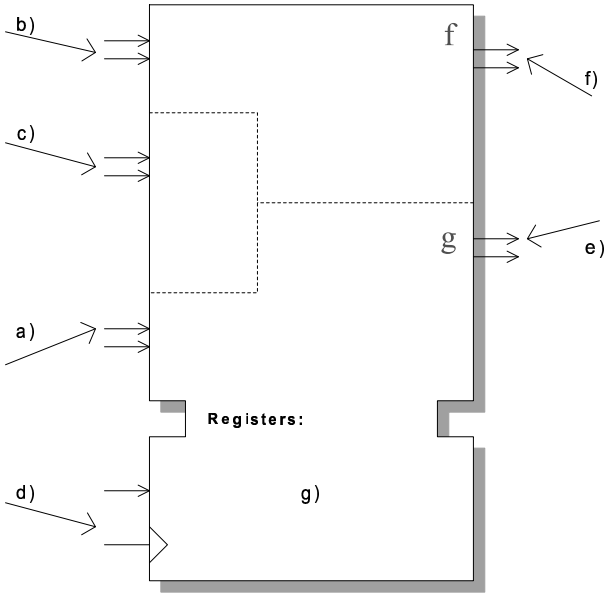

Figure 1.2: Overview diagram of a state machine
a) Input signals that can only affect the next state
b) Input signals that can directly change the output
c) Input signals that can directly affect both the output and the next state
d) Reset and clock signals
e) Output signals that only change with the clock (generated from the state)
f) Output signals that may change between clock cycles
g) Optionally: Registers used to store the state of the FSM

Chapter 2

System Architecture

The *Embedded Task Machine* runs on two hardware modules: the `BTnode`[1] and the `BTnodeFPGA`[7]. These modules have been developed at the TIK institute and used for earlier projects. The idea behind the project was to use the FPGA on the `BTnodeFPGA` board as a computing device, since the microcontroller of the `BTnode` would be too weak for many signal processing applications. The execution of this kind of applications is the intended purpose of the system, which is supposed to run in a network of intelligent sensing devices. These sensors would process much of the data before sending it to some controlling device, reducing the data volume transmitted and spreading the computation load over many devices.

Since the sensor hardware may be too small to hold an entire application configuration, the application is split into smaller parts, called *tasks*, and a description of how these tasks exchange data, called the *coordination description*. These two information items are then send to the executing device, where the tasks then can be scheduled to run separately over time on the hardware. This way area is saved by requiring a longer execution time.

2.1 The Execution Model

2.1.1 Modules of the Execution Model

In order to be able to execute an application which is split into different tasks and a coordination description, different modules are needed. A first module is the *task repository* which holds the different tasks to be executed in

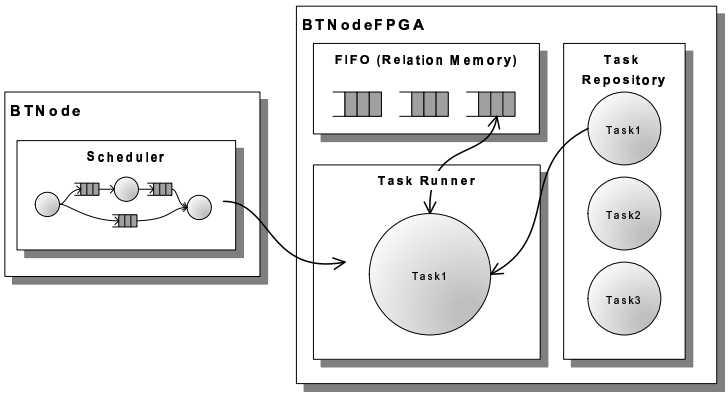

Figure 2.1: Overview of the *Embedded Task Machine*

a form which is smaller than when it is actually executed. A second module is the *relation memory*, where all the data is kept, which is interchanged between the tasks. The relation memory must be able to keep the data while the tasks are switched and executed. A third module would be the *task runner*, this is where the tasks are actually executed. It must be a reconfigurable device with a short enough reconfiguration time. Last, but not least, a *scheduler* is needed, which controls, at what time which task runs. This module is interpreting the coordination description.

2.1.2 Mapping the modules to the hardware

In order to design the system, the different modules of the Execution Model need to be mapped to the available hardware. The resources on the hardware modules are listed in Table 2.1. The different chips on the hardware modules have different properties which should be exploited for the system.

Obviously the task runner module needs to be placed on the FPGA of the `BTnodeFPGA`, since this is the only reconfigurable hardware module which

`BTnode`	`BTnodeFPGA`
ATMega128L Microcontroller	Xilinx CoolRunner CPLD
ROK 101 007 Bluetooth module	Xilinx Spartan II FPGA
AMIC 500kBit or 2MBit SRAM	AMD 1MByte Flash
	AMIC 4MBit SRAM

Table 2.1: Resources on the hardware modules

provides enough space to be able to execute tasks of sufficient size.

For the relation memory it is best to use the 4MBit SRAM since it has a direct connection to the FPGA, where the data is needed. A memory manager is needed to control the access of the data stored in the SRAM. It has to be placed between the task and the SRAM and is also placed on the FPGA.

The task repository is assigned to the Flash memory. The Flash does not loose the stored data on a power loss and thus provides a save place for the rather big configuration files which do not need to be retransmitted after a restart of the `BTnodeFPGA`. The administration of the different task slots on the Flash is implemented in the CPLD. The advantage of the CPLD is that it does not loose its configuration on a power-down as the FPGA does. The CPLD can also be used for an initialization of the whole board. The CPLD also gets the duty of reconfiguring the FPGA.

The scheduler is programmed for the microcontroller on the `BTnode`, since its work is highly data dependent and thus well suited to be executed on a microcontroller. The scheduler also calculates the sizes of the FIFOs in the relation memory and initializes the memory access structures used by the memory manager on the FPGA.

2.2 MC-Protocol

The microcontroller (MC) on the `BTnode` and the CPLD on the `BTnodeFPGA` communicate using an UART serial interface. The microcontroller uses this connection to control the the whole `BTnodeFPGA` board. The board also can send back messages through this connection. The protocol has two layers, differentiated between CPLD and FPGA commands.

2.2.1 Commands to the CPLD component

In order to create a fast communication the protocol header size is kept very small. I does not even contain any checksums. The length varies depending on the command. Each command consists of one byte header and a custom number of bytes as parameter:

4	4	
`CPLD` Cmd	Slot	Parameters

The parameter "Slot" indicates the Flash slot addressed for most CPLD commands. If this parameter is not used, it should be set to 0x0. The CPLD understands the commands listed in Table 2.2. If a command is expected to be sent to the FPGA, the CPLD command `FPGA Cmd` must be sent as further explained in Table 2.2.

Command	Code	Slot	Parameter
`Store Task`	0xA	yes	24 | Len of Data | Data
`Conf Slot`	0x9	yes	none
`Read Task`	0x5	yes	none
`Erase Chip`	0xC	no	none
`Erase Slot`	0x3	yes	none
`FPGA Cmd`	0xF	no	8 | 8 | Len of Cmd | Cmd | Param Len of Cmd defines size of Cmd and Param.

Table 2.2: Commands interpreted by the CPLD. Column Slot shows if this field is used with this command. If it is not used, this parameter must be set to 0x0.

2.2.2 Commands to the FPGA

The commands to the FPGA have two layers, the first layer contains the `FPGA Cmd` command to the CPLD, which routes the next `Len of Cmd` bytes to the FPGA. The first of these bytes contains the command for the FPGA, the remaining its parameters. The following commands can be sent to the FPGA:

Run Task

FPGA-Command code: 0x01
This command is used to start an already loaded task for a given number of cycles. The FIFO buffers used for the task and the input/output tasks are supplied with the command message. If the task has completed it replies with `TASKTERMINATED`. The command may reply with a `NACK` if a task is already running. A list of response codes can be found in Section 2.2.3.
The syntax of the command is given here:

8	8	8	8	8	8	8	16
0x01	`Port1`	`Port2`	`Port3`	`Port4`	`Port5`	`Port6`	`Cycles`

Note: `Cycles` indicates the number of iterations the task has to be executed (and not the clock cycles!). The Port-parameters have the following setup:

2	6
Driver config	FIFO-Number

Driver config defines if this port is used by input or output driver. Please make sure you always configure an input and an output port. If you don't need the drivers, attach it to an unused FIFO.

Value	Description
00	No driver attached
10	Output driver attached
11	Input driver attached
01	Not allowed

FIFO-Number is the FIFO number to be mapped to this port. See Section 2.4 for more detail.

Write SRAM

FPGA-Command code: 0x10
You can directly write to the SRAM on the `BTnodeFPGA`. Since data is written to the SRAM in 16-bit WORDS, the supplied data must have an even byte length. The `Length` parameter holds the number of WORDs to be

written. The first WORD is written to the address passed along in the `Address` parameter. Because the address has a 18-bit width, it needs two additional bits which are taken from the `length` parameter byte. With this setup a maximum of 64 WORDs (128 Bytes) can be written at once. The command replies with an `ACK` if it has successfully completed.
The syntax of the command is given here:

8	6	18	
0x10	Length	Address	Data

Note: Only the 6 most significant bits of the second byte contain the `Length` parameter. Length means data size in 16-bit WORDs.

Read SRAM

FPGA-Command code: 0x11
This command reads data from the SRAM. The `Length` parameter holds the number of WORDs to be read. The first WORD will be read from the supplied address. Because the address has a size of 18-bit it uses the two most least significant bits of the second byte. Each WORD is returned separately in a data packet. The packet header is DATA. See Section 2.2.3 for more detail.
The syntax of the command is given here:

8	6	18
0x11	Length	Address

Note: `Length` indicates the data size in 16-bit WORDs.

2.2.3 Response from FPGA module

The CPLD may answer to a command from the MC either with an `ACK` or a `NACK`. In these cases, there is no additional data attached. If the FPGA sends information, the message contains a message code and one WORD of data. Possible message codes can be found in Table 2.3.
The packets have the following setup:

8	16
Message Code	Data

Message	Code	Data	Description
ACK	0x33	no	
NACK	0x66	no	
TASKFINISHED	0xCC	no	The running task has finished.
DATA	0xAA	yes	Data returned answering a FPGA command. E.g. SRAM read.
OUTPUT	0x99	yes	Data sent by output driver.

Table 2.3: List of response messages codes

2.3 FPGA-Protocol

There are 16 wires available between the CPLD and the FPGA. Eight out of them are used to create a 8-bit data bus. 5 are used for communication control and one is used to reset the FPGA. Two lines are left unused. An overview of the connections can be seen in Figure 2.2.

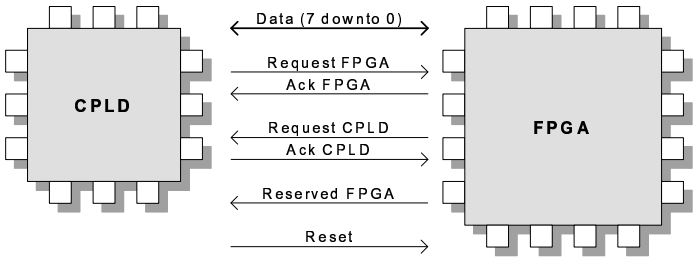

Figure 2.2: Wires between CPLD and FPGA

2.3.1 Data flow

The protocol setup is designed to work with differing clocks on the CPLD and the FPGA. This requires a longer handshake but since most communication will be forwarded trough the RS232 interface anyway, this does not matter. The transmission of one byte requires at least 5 clock cycles.

Communication starts by setting request line high. The receiver accepts by setting its acknowledge line high. If the receiver is still busy, it will not

accept until it is ready to do so. When the acknowledge line is high, data can be placed on the data lines and request line turns low. Communication is terminated by setting the acknowledge line to low. The data line access must remain on high impedance if the remote acknowledge line is not high. A timing diagram of the whole communication is shown in Figure 2.3.

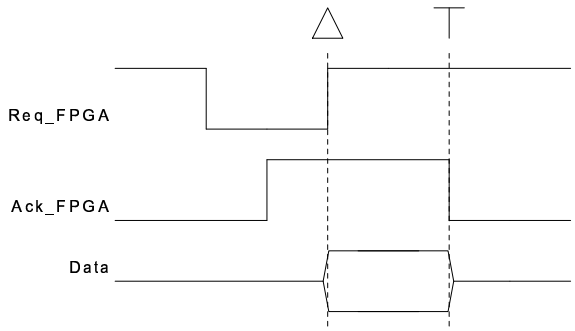

Figure 2.3: Timing details of a single write cycle from CPLD to FPGA

The implementation of the protocol slightly differs between the CPLD and the FPGA. In case both modules try to send to the bus, the CPLD has a higher priority. There is an additional line named `FPGA_Res`, which the FPGA uses to signal a multi-byte packet is being sent to the CPLD. This line is used by the CPLD to ensure that the packet will be transmitted without interruption to the RS232 module. This way a complete packet can safely be sent from the FPGA to the MC. The signal does not change the communication priority between the CPLD and the FPGA in any way.

2.4 FIFO structure in the SRAM

The FIFOs is where the drivers and the tasks store their input and output data. This data is stored in the SRAM on the module, since it has to survive a reconfiguration of the FPGA but not a power down of the BTnodeFPGA board.
The data structure used to store the structure of these FIFOs has been imported from [5] and can be examined in Figure 2.4. The FIFO access information is stored in the lowest address space of the SRAM, occupying

4 addresses per FIFO. The rest of the address space is free for FIFO data. For every FIFO the following items are stored:

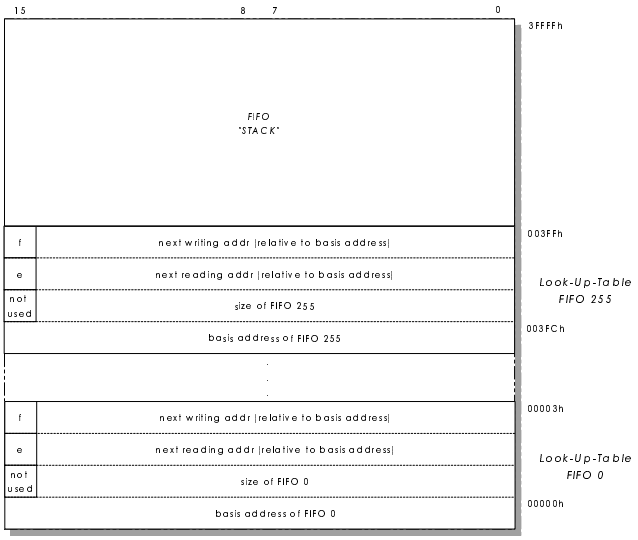

Figure 2.4: SRAM FIFO structure (image from [5])

- The **base address** indicates where the FIFO starts in the SRAM memory. Since the SRAM address is 18bit wide, and the SRAM word only 16 bit, the base address needs to be multiplied by 4 to get the actual address. A FIFO can thus only start at every fourth address.
- The second entry is the **FIFO size**. It indicates how many 16 bit words a FIFO can store. Together with the base address, the FIFO size determines the address range a FIFO occupies in the SRAM. This way, the FIFO structure would allow other data to be stored in the unused space of the SRAM.
- The **next reading address** is a relative address that indicates which data item is to be read next. After every read, it is incremented and wrapped around to 0 if the maximum relative address (the FIFO size) is reached. The address is given relative to the base address and must be added to it.
- Similarly, the **next writing address** points to the data item which can be overwritten on the next write access. This address is relative to the base address as well.

- In case the *next reading address* and the *next write address* point to the same address, two **full and empty bits** are stored at the addresses most significant bits to indicate whether the FIFO is full or empty.

The FIFO structure is accessed by the FIFOControl (see Section 4.7) to determine the FIFO locations. The FIFO structure must be created before starting the task by the microcontroller using the FPGA commands listed in Section 2.2.2.

2.5 Task slots in Flash

The `BTnodeFPGA` comes with an AM29LV081B[8] Flash chip that has 1 MByte of storage space. It is organized in 16 blocks of 64 KBytes. This is enough space to store 5 tasks with a full FPGA configuration size of 167'053 bytes, which uses three slots on the Flash. A detailed organization scheme of the Flash can be found in Figure 2.5.

Sector	Sector Size	Address-Range	Task Slot
SA0	64 Kb	00000h-0FFFFh	unused
SA1	64 Kb	10000h-1FFFFh	Slot 0
SA2	64 Kb	20000h-2FFFFh	
SA3	64 Kb	30000h-3FFFFh	
SA4	64 Kb	40000h-4FFFFh	Slot 1
SA5	64 Kb	50000h-5FFFFh	
SA6	64 Kb	60000h-6FFFFh	
SA7	64 Kb	70000h-7FFFFh	Slot 2
SA8	64 Kb	80000h-8FFFFh	
SA9	64 Kb	90000h-9FFFFh	
SA10	64 Kb	A0000h-AFFFFh	Slot 3
SA11	64 Kb	B0000h-BFFFFh	
SA12	64 Kb	C0000h-CFFFFh	
SA13	64 Kb	D0000h-DFFFFh	Slot 4
SA14	64 Kb	E0000h-EFFFFh	
SA15	64 Kb	F0000h-FFFFFh	

Figure 2.5: Organization of the task slots in the Flash

Chapter 3

CPLD-Design

The CPLD is the controlling unit on the `BTnodeFPGA`[7] board. Over an UART interface, it receives the commands to be executed by the different components. Its duties include writing and reading to the Flash and configuring the FPGA.

3.1 CPLDTop

The `CPLDTop` entity is main entity of the CPLD. It does not include any logic but only interconnects the different entities of the CPLD and the external interfaces as shown in Figure 3.1.
The entities interconnected by `CPLDTop` are:

- The `RS232Core`, which handles all communication over the UART interface
- The `FlashCore` realizes all transactions to the Flash storage of the `BTnodeFPGA` board
- The `FPGAConfigurator` controls the configuration interface of the FPGA during the reconfiguration process
- The `FPGAComm` forwards the communication between the FPGA and the microcontroller to the `RS232Core`
- The `CPLDControl` finally controls all operations within the CPLD

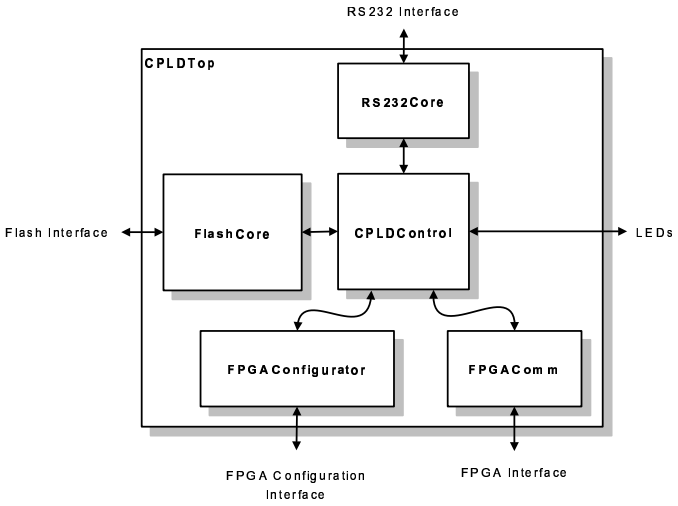

Figure 3.1: The architecture of the CPLD

3.2 CPLDControl

The `CPLDControl` is the "mind"of the CPLD. It accepts commands coming from the `BTnode` and controls their execution. It essentially executes the MC-Protocol described in Section 2.2.

The Finite State Machine (FSM) of the `CPLDControl` is shown in Figure 3.2.

3.3 RS232Core

3.3.1 Description

The `RS232Core` handles the communication over the UART interface of the `BTnodeFPGA` board. The entity has originally been programmed by Silvan Wegmann and further improved Roman Plessl. It is easily adapted to different baud rates and entity clocks. The baud rate for this project has been set to 115'200 bps with a clock of 18.432 MHz[1]. This is the highest baud rate that most devices support. Higher baud rates can be used by changing the generics of the instantiation of the `RS232Core` in the `CPLDTop` entity code. However, this should well considered because different handshakes between

[1]The original `BTnodeFPGA` oscillator with 10Mhz has been replaced on this board. The UART interface can better be controlled with the new oscillator

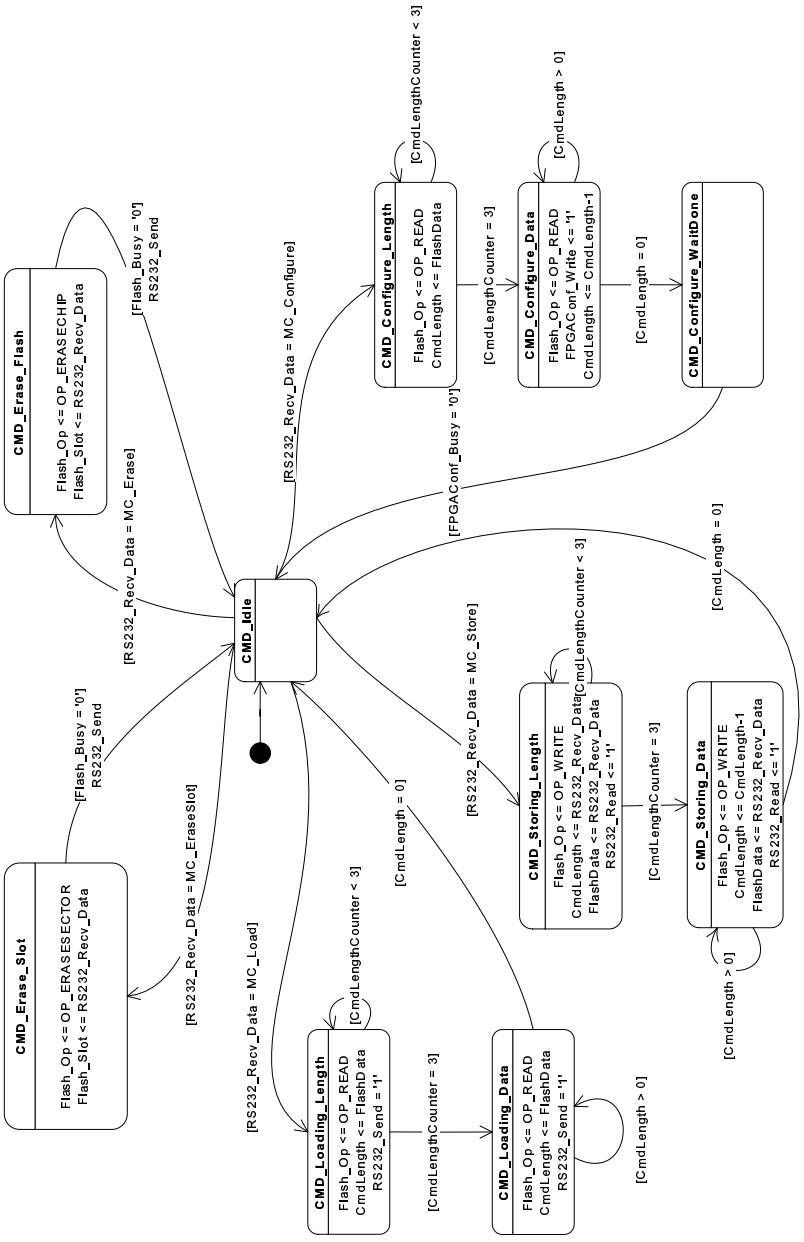

Figure 3.2: FSM of the `CPLDControl` entity

the entities have been simplified with the assumption of the RS232Core having a long delay to send and receive data.

The final interface for the `RS232Core` module is the following:

```
entity rs232core is
  generic (
    Clockfrequency : natural := 18432000;
    Baudrate       : natural := 115200
    );

  port (
    -- internal ports
    Send       : in  std_logic;
    WE         : in  std_logic;
    Send_Ready : out std_logic;
    Send_Data  : in  std_logic_vector(7 downto 0);

    Read       : in  std_logic;
    Data_Ready : out std_logic;
    Recv_Data  : out std_logic_vector(7 downto 0);

    -- external ports
    RX         : in  std_logic;
    TX         : out std_logic;

    -- standard ports
    ClkxCI     : in  std_logic;
    RstxRBI    : in  std_logic
    );
end rs232core;
```

Listing 3.1: Entity declaration of the `RS232Core`

3.3.2 Usage

If data has been **received**, the `Data_Ready` signal turns high to indicate that the data lying on `Recv_Data` is valid. The `Data_Ready` signal turns low again as soon as it is acknowledged by the other part setting the `Read` signal to high. The `Recv_Data` will then remain valid until the next whole byte has been received over the RS232 interface. If a new byte has arrived, the `Recv_Data` buffer data will be replaced, regardless if the existing data has been acknowledged or not.

To **send** data, the send buffer first has to be filled with data. This is done by setting the `WE` signal high while the data to be sent lies on `Send_Data`. The sending process is started when the `Send` signal is set to high. The `Send` and `WE` signals can be set concurrently. `Send_Ready` turns to low when sending and will return to high when all data has been sent.

`RS232Core` implements two separate buffers for reading and writing such that these operations can be carried out concurrently.

3.4 FlashCore

3.4.1 Description

The `FlashCore` is used to read, write (program) and erase the `AM29LV081B` Flash chip. It is the only entity with direct contact to the Flash chip to hide the details of the protocol to the other entities on the CPLD.

The `AM29LV081B` needs exact timing settings. These can be found in the `AM29LV081B` data sheet[8]. The `FlashCore` FSM uses a clock at half the speed of the original clock. A timing diagram of the Flash access procedure can be examined in Figure 3.3. `FlashCore` has been designed such that no external entity needs to know about the timing details of the Flash chip.

3.4.2 Usage

The Flash memory has been organized in task slots. Further details are explained in Section 2.5. Data cannot be written explicitly to certain addresses of the Flash. Read and write operations always start at the beginning of a slot. An internal counter increments the memory address after each Flash read/write cycle.

The `FlashCore` has a simple internal interface: `OpxSI` defines the current operation of the entity. This signal may only be changed when `BusyxSO` is low. If an operation is completed, the FSM is returned to its idle state by setting `OpxSI` to 000 (`None` code). The available operation codes are listed in Table 3.1. The `SlotNumberxDI` signal defines the slot number on which the operation has to be performed. This number must be between 0 to 4 and needs to be valid for the whole duration of the operation. `ReadDataxSI` is used initiate the read or write operation of the next byte of the stream.

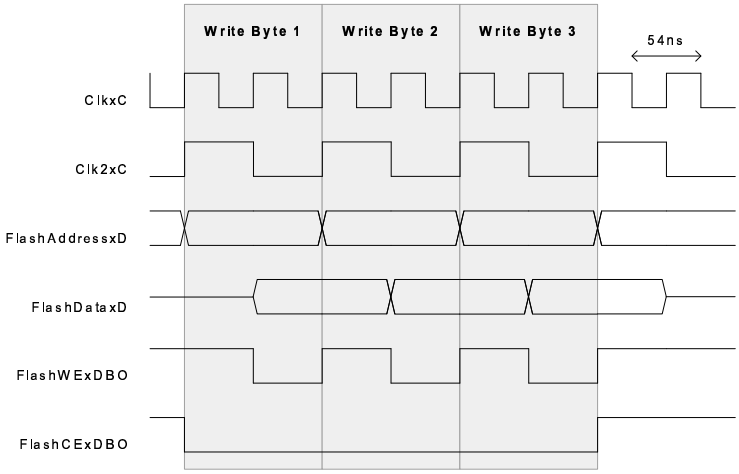

Figure 3.3: Timing diagram of the Flash access. This diagram shows a write operation of three bytes. The `FlashWE#` pin is directly connected to `Clk2xC` during the write process. This makes it possible to implement the write operation of one byte in a single FSM state. `FlashDataxD` has to be delayed.

The signal must be set high for one clock cycle. `BusyxSO` is high when a Flash operation is in progress, inputs should not be changed during this time for safe operation. `DataxDO` and `DataxDI` are used to access written or read data. `DataxDI` is buffered, so the input signal can be changed one clock cycle after `ReadDataxSI` had been high at the rising clock edge.

Operation	Code
`None`	000
`Read`	001
`Write`	010
`EraseSector`	100
`EraseChip`	101

Table 3.1: `FlashCore` operation codes

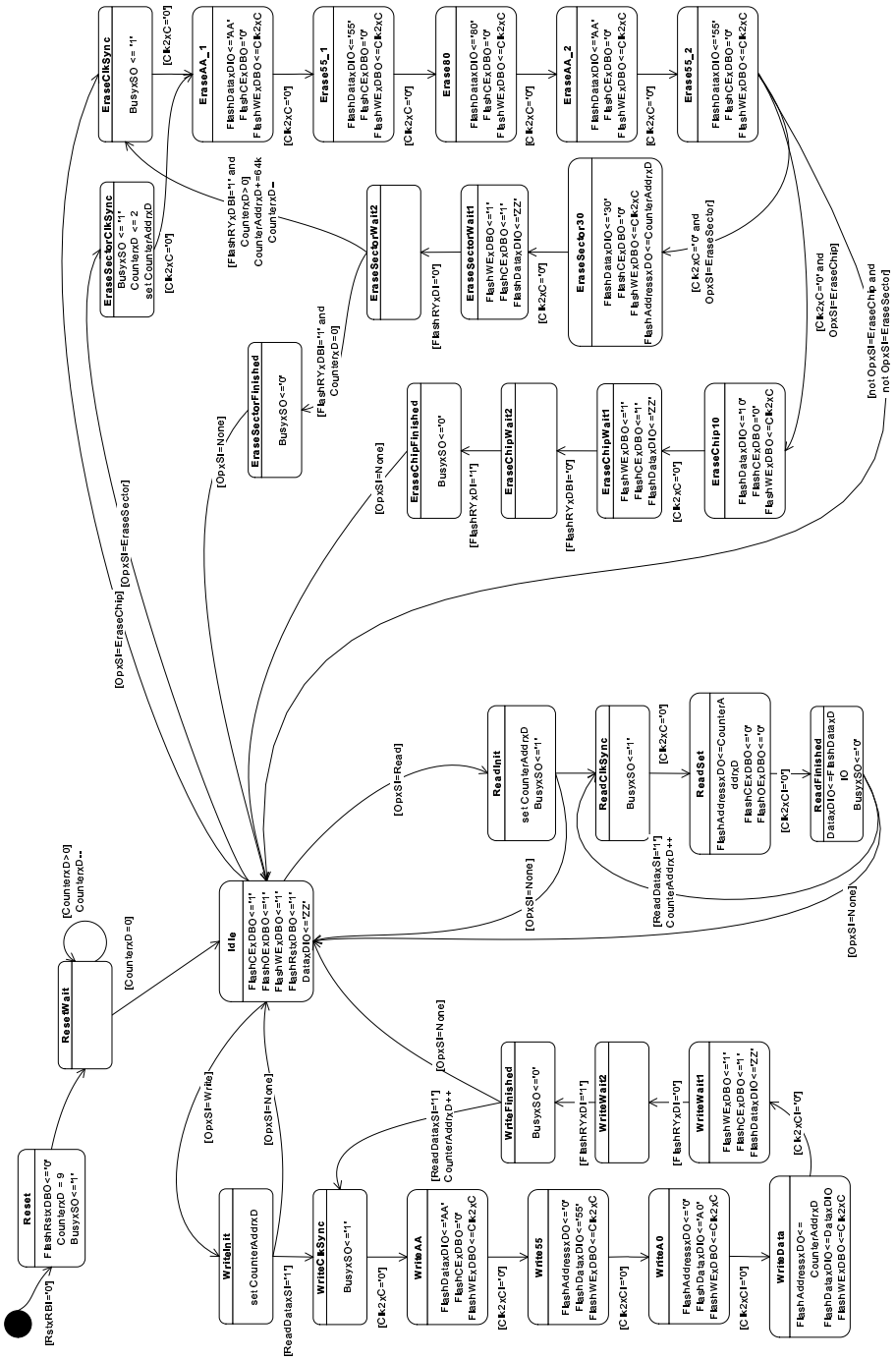

Figure 3.4: Finite State Machine of the `FlashCore`. Note the implementation of the three commands: write, read and erase.

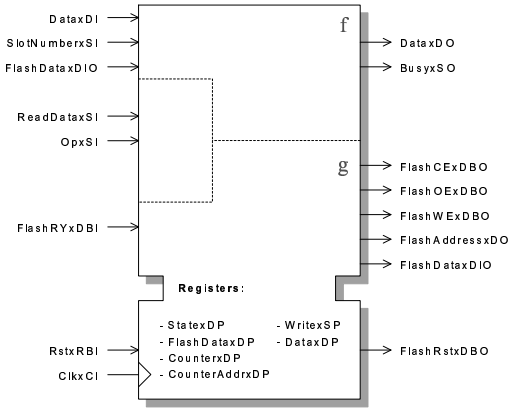

Figure 3.5: Overview of the Finite State Machine of the `FlashCore`

3.4.3 The state machine

The `FlashCore` FSM is build out of 31 states. An interface overview is presented in Figure 3.5. The detailed FSM can be examined in Figure 3.4.

3.5 FPGAConfigurator

3.5.1 Description

The `FPGAConfigurator` entity reconfigures the FPGA connected to the CPLD using the configuration port in *Slave Parallel Mode*. The configuration data is provided by the `FlashCore` module which is synchronously controlled by the `CPLDControl` entity during the configuration process. Thus, the `FPGAConfigurator` just drives the configuration controls ports and does not care about the configuration data ports on the FPGA.
The configuration process for a `Spartan2` FPGA is described in more detail in [9]. The process has been modeled as the FSM as shown in Figure 3.5.1 and Figure 3.5.1.

3.5.2 Usage

To start a configuration of the FPGA, `ConfigurexSI` has to be set to high and should remain high. `BusyxSO` will then immediately turn high and not return to low until the FPGA is ready to accept the configuration data. As

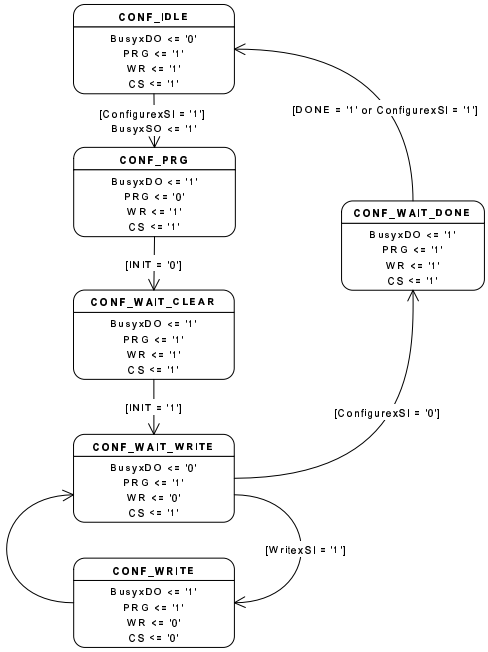

Figure 3.6: The FSM model for the FPGA configuration process

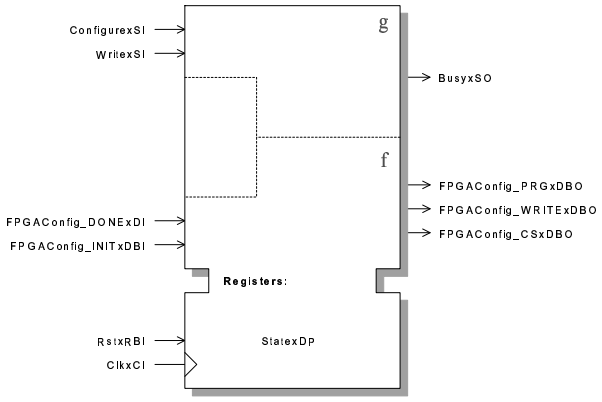

Figure 3.7: The interface of the `FPGAConfiguratorFSM`

soon as BusyxSO is low, the configuration data writing cycles start. The controlling unit should make sure that valid data lies on the parallel slave configuration data ports of the FPGA and set the WritexSI signal to high. The FPGAConfigurator then signals the FPGA to read the configuration data. BusyxSO will turn high for this time and return to low again as soon as the FPGA is ready for the next configuration data byte.
When all configuration data has been written, the ConfigurexSI signal can be set to low. The FPGAConfigurator will answer by setting BusyxSO to high, and waiting for the FPGA to complete the reconfiguration. Then, BusyxSO is set low again, and FPGAConfigurator is ready for another configuration process

Note: The configuration data is not checked to be valid. If the configuration is not accepted by the FPGA, it does indicate this by leaving its FPGAConfig_INITxDBI signal low. FPGAConfigurator does ignore this signal in this state. Thus it may happen that the FPGA has not been reconfigured and remains in its previous configuration.

3.6 FPGAComm

3.6.1 Description

The FPGAComm module is used to implement the interface to the CPLD. The custom protocol described in Section 2.3 is used for communication. The state machine used is shown in Figure 3.9. An interface description can be examined in Figure 3.8.

3.6.2 Usage

The state machine works asynchronously. All communication lines are buffered. To write a byte to the FPGA, DataxDI must be set and SendxSI tied to high for one clock cycle. BusyxSO then goes low until the byte could be successfully sent. Newly received data is signaled by the NewDataxSI signal turning high. Data then can be read from DataxDO. ReadxSI must be tied high for one clock cycle to acknowledge the reading of the received byte. Because the state machine works asynchronously, it is possible to receive a byte before the byte waiting to be sent is actually sent.

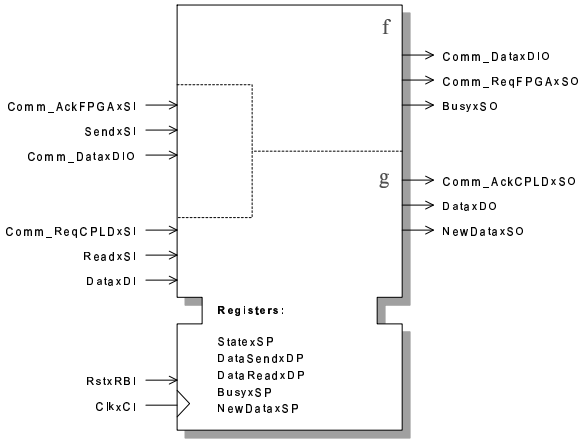

Figure 3.8: Overview of module `FPGAComm`

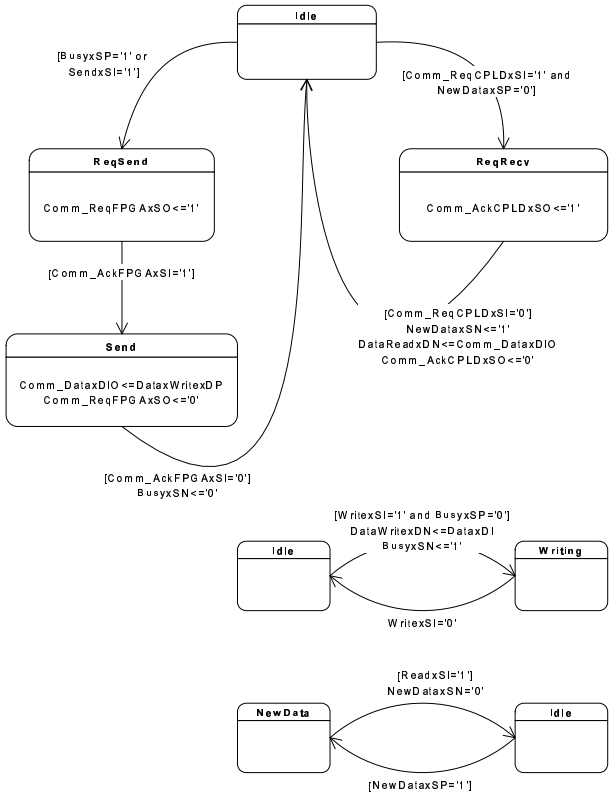

Figure 3.9: `FPGAComm` FSM

Chapter 4

FPGA-Design

The FPGA is the execution unit for the tasks. It starts and stops the tasks and controls their memory accesses. The FPGA also holds drivers for its external interfaces, where input data for the applications can be collected.

4.1 FPGATop

The `FPGATop` module interconnects the different entities of the FPGA design. Figure 4.1 shows the structure of the architecture. A short description of the different entities is given below:

- `CPLDComm` handles the communication to the CPLD
- `SRAMCore` reads and writes to the SRAM on the `BTnodeFPGA` board
- `MemoryControl` handles concurrent access requests to the `SRAMCore` by the `FIFOControl` or the `FPGAControl` entities
- `FIFOControl` acts as a cache for FIFO accesses by the `Task`, `InputDriver`, and `OutputDriver` entities
- The `InputDriver` generates or reads data from an external interface of the FPGA and feeds it to a FIFO
- The `OutputDriver` sends data from a FIFO over the CPLD to the microcontroller
- The `FPGAControl` accepts commands passed by the CPLD from the microcontroller and executes these on the FPGA

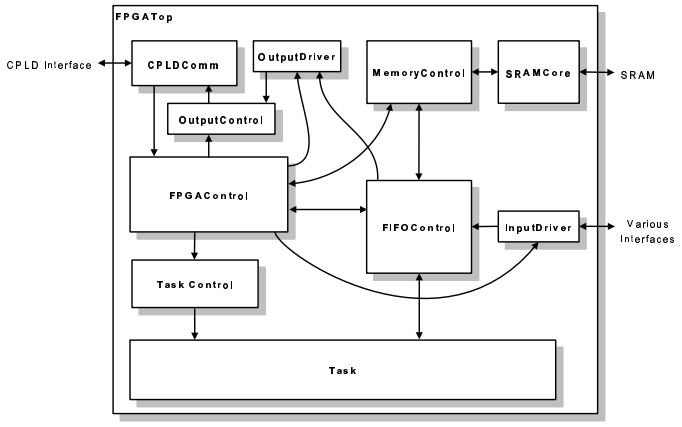

Figure 4.1: FPGA Architecture

- The `Task` entity is where the custom tasks are configured to and executed

4.2 FPGAControl

4.2.1 Description

`FPGAControl` is the main controller on the FPGA. It has two general purposes: Firstly, it accepts incoming commands from the CPLD and executes them. Secondly, it stores back the cached FIFO structure data and signals the termination of the task to the CPLD when the running task has finished.

Three commands are currently understood by the FPGAControl: the commands to read and write to the SRAM and to start the task. How these commands can be sent to the FPGA is described in Section 2.2.2.

`Read SRAM` and `Write SRAM`

These two commands are used to read and write directly to the SRAM. Writing is usually used to initialize the FIFOs before an application is started. Reading has been implemented for debugging purposes.

`Start task`

Before a task can be started, the FIFO structure data used by the task must be loaded into the cache. Section 4.7 describes the procedure in fur-

ther detail. The required FIFO numbers are submitted as parameters of the `StartTask` command. After the initialization of the FIFOs, a signal to `TaskControl` allows it to start the task.

Figure 4.2 shows a symbolic state chart of `FPGAControl`. It is not specified when the transitions occur to simplify the figure.

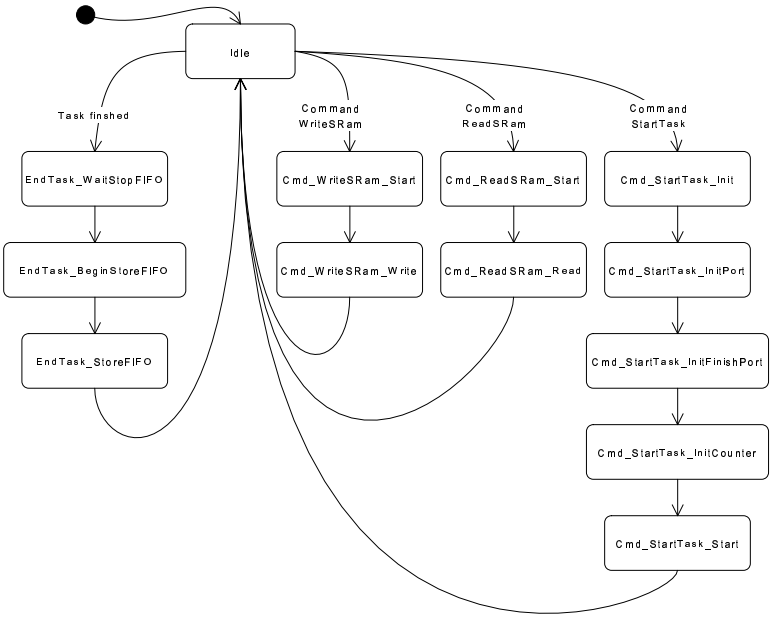

Figure 4.2: Symbolic state chart of `FPGAControl`

4.3 CPLDComm

4.3.1 Description

The `CPLDComm` module is used to handle the interface to the CPLD which uses the FPGA-Protocol described in Section 2.3. `CPLDComm` is very similar to the `FPGAComm` on the CPLD described in Section 3.6. The most important difference is that the access priorities differ. If the CPLD requests to send a byte, it has priority over the FPGA. Because the FPGA won't accept a new byte if the old one hasn't read and the CPLD has priority to send, it is important that every data that has arrived is taken from the receiver's buffer. If the FPGA tries to send a byte and doesn't read received data a deadlock

may happen! The FSM modeling the protocol is shown in Figure 4.4. An interface description can be found in figure 4.3.

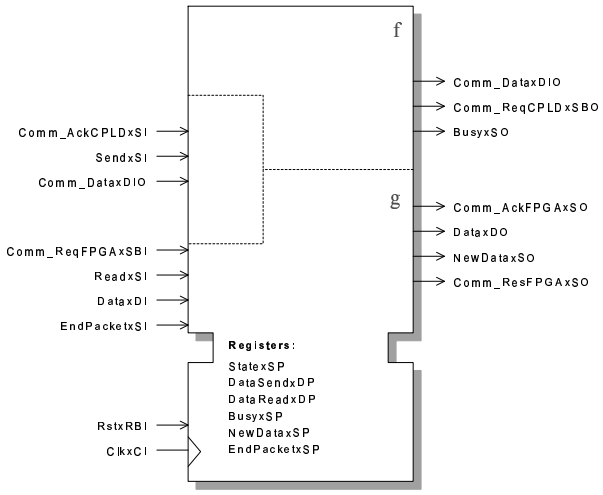

Figure 4.3: Overview of the module `CPLDComm`

4.3.2 Usage

The state machine works asynchronously. All communication lines are buffered. To write a byte to the CPLD, `DataxDI` must be set and `SendxSI` tied high for one clock cycle. `BusyxSO` goes low when the byte could be successfully sent. Newly received data is signaled by the signal `NewDataxSI` turning high. Data can then be read from `DataxDO`. Then, `ReadxSI` must be tied high for one clock cycle to acknowledge the byte. Because the state machine works asynchronously, it is possible to receive a byte before the the byte to send could be sent.

The FPGA-Protocol provides a special line for the FPGA to reserve a communication line to the `Btnode`. It is automatically used and therefore the last byte of any packet must be marked by additionally setting `EndPacketxSI` to high when sending the last byte by triggering `SendxSI`. This must even be done for a single byte packet.

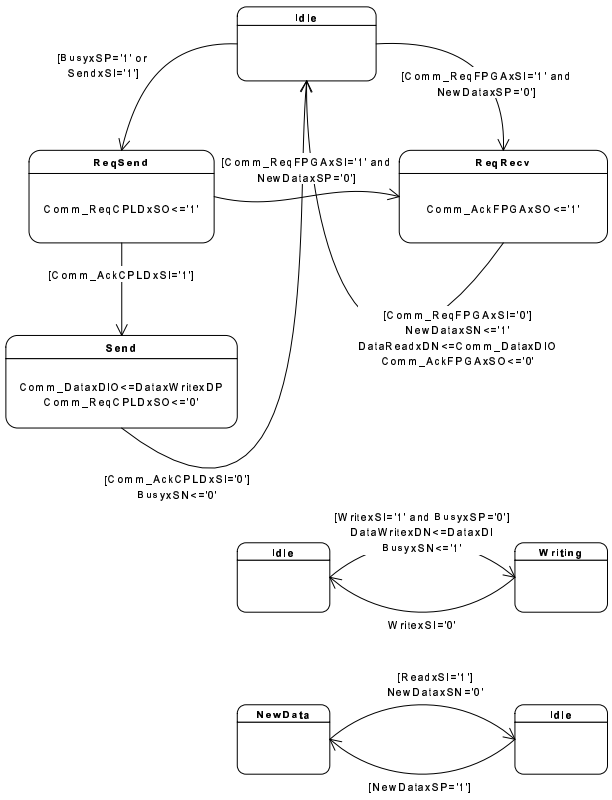

Figure 4.4: `CPLDComm` FSM

4.4 OutputControl

The `OutputControl` entity is used to share the `CPLDComm` communication entity between the the `FPGAControl` and the output driver. The same packeting mechanism as used on the `CPLDComm` interface is used to reserve the interface for a multi-byte data transmission. `FPGAControl` has a higher priority when writing to the data bus.

The advantage of using the same interface as the `CPLDComm` entity is that the connected modules do not have to be changed when directly connected to the `CPLDComm`. The entities do not notice that other modules are currently sending data since the `OutputControl` signals to the requesting entity that the external interface busy until its first byte has been transmitted.

4.5 SRAMCore

4.5.1 Description

The `SRAMCore` entity controls the SRAM of the `BTnodeFPGA` board that is connected to the FPGA. A detailed description of how the data in the SRAM can be accessed can be found in [10].

A problem has been encountered on `BTnodeFPGA` boards with a 18 MHz clock: since their clock cycle is 54ns, and the read and write cycles of the SRAM takes 55ns according to the specifications. Thus the SRAM should be slightly to slow for this clock. Two clock cycles are thus needed to access data on the SRAM. However, on the `BTnodeFPGA` board used for development, the SRAM worked with one clock cycle of the 54ns clock, too. The `SRAMCore` entity supports both access cycles, the required code for a two cycle access is commented in the source code.

4.5.2 Usage

To **write** a byte, the data and the address have to be set to `DataxDI` and to `AddressxDI` respectively. When `WritexSI` is set to high, the address and data must not change until the writing procedure has been acknowledged by the `SRAMCore` setting its `AckxSO` signal to high. The data can be considered written as soon as the `WritexSI` and the `AckxSO` signal have been high at a rising clock edge. If the `WritexSI` signal is kept high after `AckxSO` has been set, a new write cycle is initiated. Thus before changing any data, the

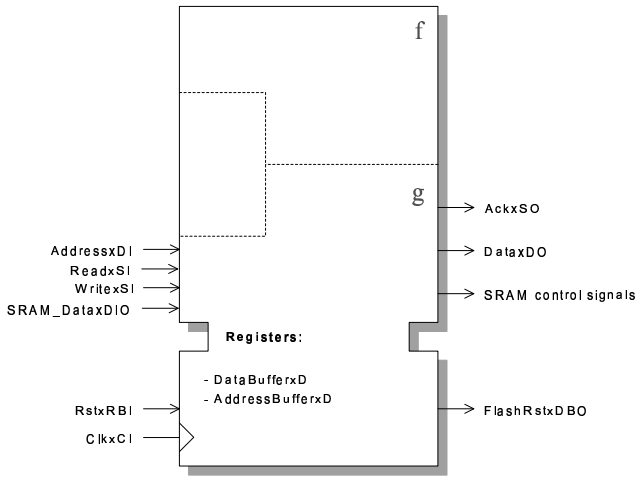

Figure 4.5: Interface of the `SRAMCore` entity

`WritexSI` signal must be removed immediately after the `AckxSO` signal has gone to low.

Setup times		Hold times	
t_{ws}	≥ 1 cycle	t_{wh}	0
t_{as}	≥ 1 cycle	t_{ah}	0
t_{ds}	-	t_{dh}	≥ 1 cycle

Table 4.1: SRAMCore handshake timings

To **read** data, the address has to be set on `AddressxDI` and the `ReadxSI` signal has to be set to high. The address should not be altered until `SRAMCore` acknowledges the reading procedure by setting the `AckxSO` signal to high. A read cycle is considered complete when the `AckxSO` and the `ReadxSI` signal have been high at a rising clock edge. After this event, the data liess on `DataxDO` for at least one clock cycle. If the `ReadxSI` remains on high after the `AckxSO` has been set to low, a new read cycle is initiated. The data is buffered in the `SRAMCore` and will not change until another operation (read or write) is started.

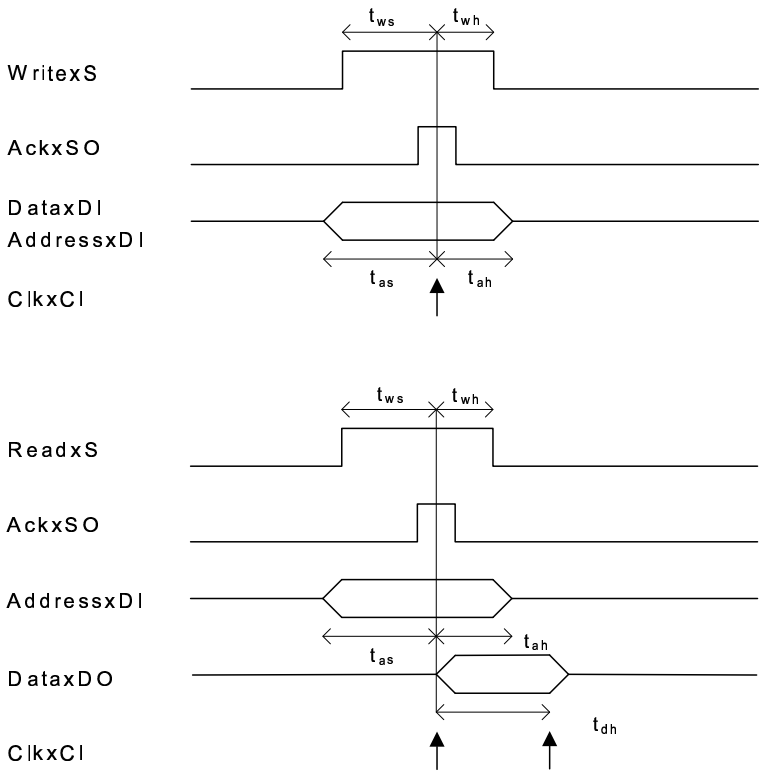

Figure 4.6: `SRAMCore` handshake, timing values are listed in Table 4.1

Note: If both the `ReadxSI` and the `WritexSI` signal are set, the read operation is executed since it does not change the data stored in the SRAM.

4.6 MemoryControl

4.6.1 Description

The `MemoryControl` entity coordinates the access to the SRAM. Two units can write to the SRAM: `FPGAControl` and `FIFOControl`. `MemoryControl` takes care that only one of these two units is accessing the SRAM at any point in time. It blocks the interface to an entity while another is accessing the SRAM.

Priority is given to the `FIFOControl` since the task or the drivers are doing the actual work and should not be impeded by configuration messages received from or sent to the controller.

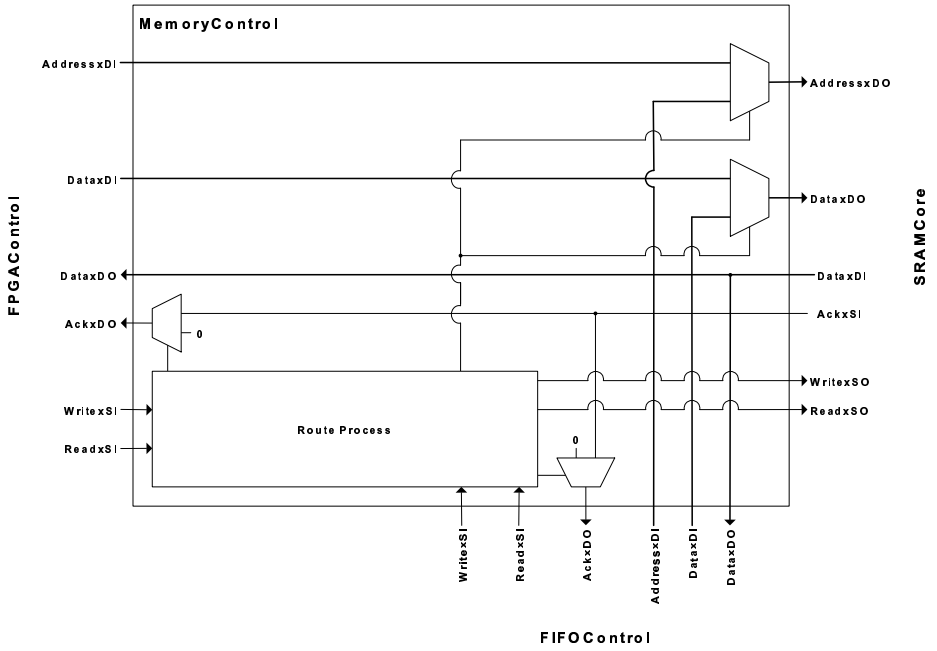

Figure 4.7: Structure of the `MemoryControl` entity

4.6.2 Usage

`FIFOControl` and `MemoryControl` can both use the interfaces of the `MemoryControl` as if they would directly be connected to the `SRAMCore` (see Figure 4.6 for details on the protocol used). There is no additional delay inserted, since `MemoryControl` observes the handshake and knows when it can switch from one to the other entity.

4.7 FIFOControl

4.7.1 Description

This entity wraps two modules that have been inherited from an earlier project[5]. These are the `Address_Register` and the `LUT_Address` entities. These modules handle the data structure of the FIFOs in the SRAM. This structure is explained in more detail in Section 2.4.

To reduce the latency of a memory access, the FIFO configurations are loaded into a cache in the `Address_Register` module. The module keeps track of every read and write access and increments the associated pointers.

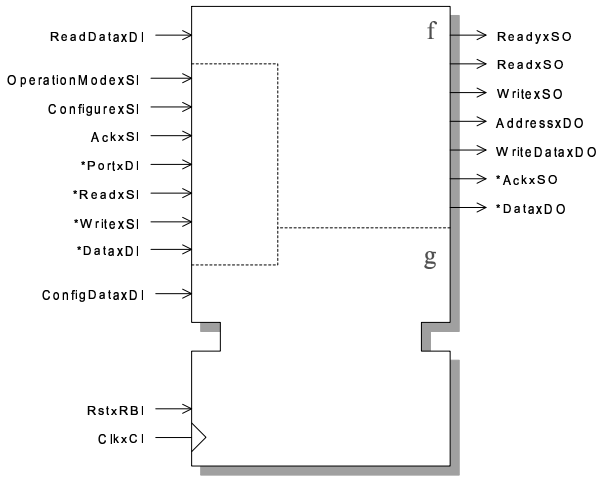

Figure 4.8: `FIFOControl` interface. Signals marked with a * are common interface signals for the different consumers

It also handles wrap overs[1]. The FIFO data structure in the SRAM is not altered. Before changing the task, the modified data in the `Address_Register` needs to be written back into the SRAM.
The `Address_Register` only keeps hold of as many FIFOs as are allowed to be ports of a task, plus two for the input and output drivers. Thus it automatically maps the port number to the corresponding FIFO in the SRAM. This mapping is conducted during the initialization process when the FIFO data is loaded into the different ports in the `Address_Register` module. The proper addresses for loading the FIFO data structure are generated by the `LUT_Address` module. The mapping is also stored into a buffer which is used to again generate the FIFO data structure addresses in the write back phase after a task has finished its execution cycles.

The `FIFOControl` entity has five interfaces. It provides three interfaces to the entities who use the FIFOs: the task, the input driver, and the output driver. These interfaces are basically used for requests to write or read something to a FIFO. Using the interface to the `MemoryControl` entity,

[1]When data is written to the last entry of the reserved FIFO space, the pointer is set back to its beginning

the FIFOControl can access the memory after having computed the physical address the request needs. The last interface connects to FPGAControl and is used for configuration purposes as further explained below.

4.7.2 Usage

Before the FIFOControl can be used, the FIFO data structure must already be present in the SRAM. This configuration has to be done externally using the MC-Protocol as defined in Section 2.2. The required structure is explained in Section 2.4.
If the FIFO data structure is present in the SRAM, the FIFOControl can be configured with other commands over the MC-Protocol. The FPGAControl extracts the necessary information from these commands and configures the FIFOControl unit. After the configuration is done, the FIFOs can be accessed by the task, the input, and the output drivers. After the task has finished its cycles, the FIFOControl needs to be instructed to write back its data before a new task FIFO configuration can be loaded into the FIFOControl.

These steps are now explained in further detail:

Configuration

When the ReadyxSO signal is high, the FIFOControl can be driven into configuration mode by setting OperationModexSI to LoadFIFO. ReadyxSO will turn to low and will return to high as soon as FIFOControl is ready to accept its first configuration byte. Now it expects the FIFO number of the first port, which can be submitted by setting ConfigDataxDI to the FIFO number and setting ConfigurexSI to high for one clock cycle. The ConfigDataxDI must remain high until ReadyxSO turns back to high again. After the impulse on ConfigurexSI, FIFOControl loads the data associated with that FIFO[2] into its buffer for its first port.

Subsequent impulses on ConfigurexSI will initiate the configuration of the next port with the FIFO data for the FIFO number given by ConfigDataxDI. New impulses are not accepted until ReadyxSO turns back to high. This configuration procedure is halted by changing OperationModexSI to AccessFIFO. If more ports are loaded than there are physically provided, a wrap around

[2]The physical address of for the first data item (base address) is computed by multiplying the FIFO number by 4

occurs and the configuration of the first port will be overwritten. Thus the configurator is responsible of not assigning too many FIFOs to the ports.

Write Back

After the task has finished its execution, the FIFO state must be written back to the SRAM before a new configuration is loaded. This can be done by driving `OperationModexSI` to `StoreFIFO` when `ReadyxSO` is high. `FIFOControl` will now start to write the FIFO configurations into the SRAM. `OperationModexSI` must remain on `LoadFIFO` until `ReadyxSO` returns to high. Changing `OperationModexSI` before the time will force `FIFOControl` back into idle mode, but leaving all FIFOs not yet written to the SRAM as they are.

FIFO Access

After a configuration has been loaded into `FIFOControl`,`OperationModexSI` can be changed to `AccessFIFO`, and FIFO accesses can be made. The handshake works like the one presented for the `SRAMCore` and displayed in Figure 4.6, with the `PortxDI` signal replacing `AddressxDI`. Priorities are given in the following order: Task Read, Task Write, InputDriver Write, OutputDriver Read.
It is important that `*PortxDI` does not change while `*WritexSI` or `*ReadxSI` are driven high.

4.8 TaskControl

4.8.1 Description

`TaskControl` is used to control a task. It receives a cycle count and a start signal from `FPGAControl` and executes the task for the given number of cycles. When the task has finished its iterations, `RunningxSO` turns low to signal the task has terminated.

The cycle count holds the number of times the task will be restarted after termination. Figure 4.9 shows the state chart.

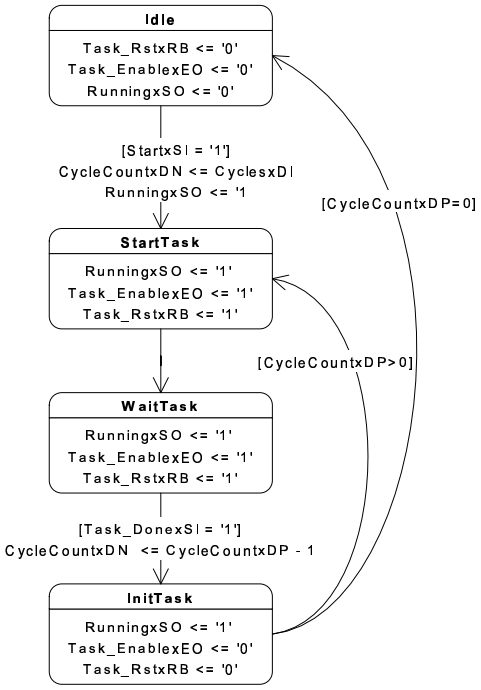

Figure 4.9: `TaskControl` state chart

4.9 Task

4.9.1 Description

The task module is where the actual work is done. Here custom modules can be placed which use the interface shown in Listing 4.1.

A task is controlled through the *execution control ports*. These are used to start and stop the execution of a task. In every iteration, the following signal forms are expected:

1. `RstxRBI` is driven low to reset the task. A task iteration must only depend on data stored in the FIFOs and no internal state that should be kept from one iteration to another. `DonexSO` must now turn to low, indicating the task is ready to run.

2. When the system is ready to run the task, `RstxRBI` goes to high. The execution of the task starts when `EnablexEI` turns high. On the next rising clock edge, the task should start its execution. If `EnablexEI` turns to low during the execution, no state change must occur in the task. This signal provides a means of halting in its execution.

3. When a task is done computing, it sets its `DonexSO` to high, indicating that it has no further calculations to do. If there are more iterations scheduled for the task, the procedure restarts at point 1.

```
entity Task is
  port (

    -- data access interface
    ReadxSO           : out std_logic;
    WritexSO          : out std_logic;
    AckxSI            : in  std_logic;
    PortxDO           : out FIFOIDType;

    DataxDO           : out std_logic_vector( 15 downto 0 );
    DataxDI           : in  std_logic_vector( 15 downto 0 );

    -- debug ports
    LEDxDO            : out std_logic_vector( 3 downto 0 );

    -- execution control ports
    ClkxCI       : in  std_logic;
    EnablexEI    : in  std_logic;
    DonexSO      : out std_logic;
    RstxRBI      : in  std_logic
  );
end Task;
```

Listing 4.1: `Task` interface

`ClkxCI` supplies a clock signal at 18.432Mhz, which gives a clock period of 54.25ns. Registers should only trigger at the rising clock edge for full compatibility with the rest of implementation. As stated above, the rising clock edge is only valid when `EnablexEI` is set to high at the same time and should be ignored otherwise.

The *data access interface* allows a task to access the FIFOs on its ports. The protocol is the same as for the `SRAMCore` as explained in Section 4.5.2 and illustrated in Figure 4.6. The handshake can most easily be implemented using a state driven VHDL design.
An code example is given in Listing 4.2.

```
case StatexSP is
  ...
        when stRead =>
          PortxDO <= INPORT;
          ReadxSO <= '1';

          if AckxSI = '1' then
            StatexDN <= stReadWait;
          end if;

        when stReadWait =>
          StatexDN <= stDoSomething;
          DATA     <= DataxDI;
          ReadxSO  <= '0';

        ...

        when stWrite =>
          WritexSO <= '1';
          PortxDO  <= OUTPORT;
          DataxDO  <= DATA;

          if AckxSI = '1' then
            StatexDN <= stDoSomethingElse;
            WritexSO <= '0';
          end if;
  ...
end case;
```

Listing 4.2: Sample FIFO access

The additional state when reading is needed because the data is buffered. It is important to drive the `WritexSO` signal to low after receiving the acknowledgement since otherwise a new write cycle would be initiated on the state transition, corrupting the data in the FIFO.

4.9.2 The TaskWrapper

For the ease of implementation of a task into the FPGA structure, the `TaskWrapper` module has been introduced. This entity has the same interface as the task and does nothing else than connecting its ports with an instance of the actual task. This way, a task can be inserted into its slot by just replacing the component name and instance type, without searching through too much code.

The task wrapper has additional functionality when it comes to partial reconfiguration. Its ports can be fixed for a design such that only the task can be newly configured into the device, saving time and memory space in the Flash holding the task configurations.

4.9.3 TaskTestbench

Since the interface for the tasks is always the same, a test bench for behavioral simulation has been created. It supports FIFO read and write access and runs the tasks for a given number of iterations. It correctly simulates the other hand of the interfaces and checks for under- and overflow on the FIFOs.

4.9.4 Requirements for a task

- The calculations should fit into a clock period of 54ns
- `fpgapackage.vhd` must be included in the design file for the `FIFOIDType` type
- The files `txt_util.vhd` and `TaskTestbench.vhd` provide a test bench for the tasks

4.10 Drivers

Drivers are very similar to tasks. The difference is that they are running in parallel to the tasks and are allowed to write to the CPLD and/or access extension ports. In the current system, it is possible to run one input and one output tasks at a time.

Drivers access FIFOs in a similar way as tasks do. A driver is suspended if the FIFO is full/empty. In the current implementation, tasks do not need to know if their calculated result is read by any other task or by the output driver.

Drivers may only access one FIFO. The port number of this FIFO is supplied by the microcontroller and submitted with the `StartTask` command parameters. See Section 2.2.2 for a description of the `StartTask` command. The Output driver may be configured to use the same FIFO that is beeing used by a parallel running task or a different one. It is important to assure a FIFO is only cached once. Because the FIFO cache is written back to the

SRAM when the task has finished, drivers are also stopped for this time. `EnableFIFOxEI` is used to signal this procedure. It would require partial reconfiguration of the FPGA to be able to run drivers continuously.

4.10.1 Input Driver

The interface of an input driver is shown in Listing 4.3. So far no input driver has been implemented.

```
entity DriverIn is
   port (
    -- Interface to access FIFO. Port is controlled
    -- by FPGAControl
    WritexSO          : out std_logic;
    AckxSI            : in  std_logic;
    DataxDO           : out std_logic_vector( 15 downto 0 );

    -- IO ports
    LEDxDO            : out std_logic_vector( 3 downto 0 );
    ExtxDIO           : inout std_logic_vector( 23 downto 0 );

    -- execution control ports
    ClkxCI            : in  std_logic;
    EnableFIFOxEI     : in  std_logic;
    RstxRBI           : in  std_logic
   );
end DriverIn;
```

Listing 4.3: Input Driver interface

4.10.2 Output Driver

The interface of an output driver is shown in Listing 4.4. Currently an output driver exists, which reads data from its FIFO and transmits it to the microcontroller.

4.10.3 DriverWrapper

Similar to the `TaskWrapper` (see Section 4.9.2), a `DriverWrapper` is provided to simplify the implementation of new drivers.

```
entity DriverOut is
   port (
    -- Interface to access FIFO. Port is controlled
    -- by FPGAControl
    ReadxSO           : out std_logic;
    AckxSI            : in  std_logic;
    DataxDI           : in  std_logic_vector( 15 downto 0 );

    -- Serial Output
    Comm_SendxSO      : out std_logic;
    Comm_DataSendxDO  : out std_logic_vector( 7 downto 0 );
    Comm_BusyxSI      : in  std_logic;
    Comm_EndPacketxSO : out std_logic;

    -- IO ports
    LEDxDO            : out std_logic_vector( 3 downto 0 );
    ExtxDIO           : inout std_logic_vector( 23 downto 0 );

    -- execution control ports
    ClkxCI            : in  std_logic;
    EnableFIFOxEI     : in  std_logic;
    RstxRBI           : in  std_logic
   );
end DriverOut;
```

Listing 4.4: Output Driver interface

Chapter 5

MC Software

5.1 Introduction

The microcontroller on the `BTnode` board is the "mind"of the whole system. It controls when which task is to be executed and sends the commands to configure the task environment.
To facilitate the implementation of the controlling functions on the microcontroller, an existing operating system has been chosen, which has already been used at the institute for several projects on the `BTnode`. It provides an ideal environment for the implementation of the software part of the project.

5.2 Synchronous Data Flow (SDF)

Data flow programs can be described using directed graphs. An example of an SDF graph is shown in Figure 5.1. *Nodes* represent functions and *arcs* carry data from one node to the other. The execution of the function of a node is called *firing*. When firing, a node consumes a certain amount of data tokens from all of its input arcs and produces a number of data tokens at its outgoing arcs. The number of consumed or produced data tokens is noted at the arc start or end point. A node may only fire when sufficient data tokens are available on its input arcs.

Synchronous Data Flow[11] demands that the number of consumed and produced data tokens is known a priori for all nodes. This requirement allows to compute a schedule for the nodes before the execution starts. This schedule is called *static*, meaning that it will not change during the execu-

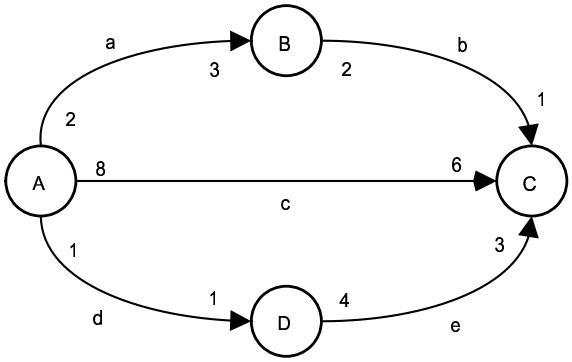

Figure 5.1: SDF graph

tion of the graph.

In the project, a SDF execution algorithm has been implemented. A large application, whose implementation is too large to fit on the given hardware is broken down into smaller *tasks*, which represent the nodes of a SDF graph and are executed one after another on the reconfigurable device. FIFOs are provided to exchange data and represent the arcs in the SDF.

SDF has mainly been chosen for the advantage of the ability to precompute a schedule. This way, a task is not preempted and its state must not be preserved[1]. The preemption process would significantly complicate the whole process.

Some asynchronous behavior is introduced with the input and output driver tasks, which communicate over some interfaces of the `BTnodeFPGA` board (input driver) or with the microcontroller (output driver). The drivers are always running and can produce or consume data tokens with an inconsistent rate. These drivers pose the only aberration to the SDF model. However, assuming a constant rate for these tasks should not be a too harsh approximation.

[1]If a task is blocked during the execution and is exchanged for another, the partially processed input data already removed from the input FIFOs must somehow be saved and restored before continuing the task

5.2.1 SDF formalism

Any SDF graph can be described by a *topology matrix* Γ [11]. In this matrix, every row describes an arc and every column represents a node of the SDF graph. A negative entry in the matrix indicates that a node consumes tokens from an arc, whereas the production of tokens is noted as a positive number. Note that the sum of a row specifying an arc starting and ending on the same node (a *self-loop*) needs to be zero for a correctly constructed graph. If the sum would be positive, data would be accumulated on this arc without ever being removed. A negative sum would constantly reduce the amount of data on the arc and eventually reach a value which prohibits the node of ever being executed again.

	A	B	C	D
a	2	-3	0	0
b	0	2	-1	0
c	8	0	-6	0
d	1	0	0	-1
e	4	0	-3	0

topology matrix Γ for the SDF graph shown in Figure 5.1

A *schedule* for the SDF graph can be represented by a vector $q \in N^s$ where s is the number of nodes of the SDF graph. Every element of the node indicates how many times the associated node is to be executed during a single schedule iteration.

A necessary condition for the existence of a schedule with bounded FIFO memory requirements is: $\text{rank}(\Gamma) = s - 1$. This follows from the fact that in a schedule with bounded memory, a multiplication of the topology matrix with the schedule vector needs to give a zero vector ($\Gamma q = 0$) in order to avoid an accumulation of tokens on the arcs. The prove can be found in [12].

5.2.2 Scheduling

A simple scheduling algorithm as presented in [11] has been chosen for implementation. It basically works as follows:

1. set $q_i = 0$

2. set $q_1 = 1$, the first node runs one time

3. if q_i is connected to q_j through arc a and q_i is scheduled (nonzero), set $q_j = q_i \left| \frac{\gamma_{a,i}}{\gamma_{a,j}} \right|$. If q_j is already scheduled with a different number, the graph cannot be scheduled

4. repeat step 3 until nothing can be changed anymore. If not all nodes could be scheduled, the SDF graph contains subgraphs which are not connected to others. Repeat the algorithm for the rest of the nodes in the schedule.

5. multiply q by the least common multiple of all denominators of the elements. The resulting schedule contains now only natural numbers

As an example, the schedule of the SDF graph pictured in Figure 5.1 is calculated in the following steps:

1. Initialization:
$$q' = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

2. after evaluating row 1:
$$q'' = \begin{pmatrix} 1 \\ \frac{2}{3} \\ 0 \\ 0 \end{pmatrix}$$

3. after evaluation of all rows
$$q''' = \begin{pmatrix} 1 \\ \frac{2}{3} \\ \frac{4}{3} \\ 1 \end{pmatrix}$$

4. to get rid of the fractions, we multiply q''' by the least common multiple of all denominators, in this case 3:

$$q = \begin{pmatrix} 3 \\ 2 \\ 4 \\ 3 \end{pmatrix}$$

This algorithm generates a schedule with arbitrary order. However, this project needs a *single appearance schedule* (SAS), where every node should be executed only once and as many times as possible. This will not give a schedule with optimal memory requirements, but be optimal in terms of runtime, since the least FPGA reconfigurations need to be made. A simple algorithm to find this sequence would be [13]:

1. schedule all root nodes, respectively those with no input
2. iteratively schedule all nodes, whose parents have already been scheduled
3. all nodes are scheduled with q_i, that is, with all their iterations per schedule

The produced schedules are usually written in some form of sequence like XYXXYZZ. A short hand notation exists for these sequences which consists of two shortcuts:

- If a task is executed consecutively a number of times, this number is written in front of the task and it is not repeated, this way, XYYYZ can be written as X3YZ.
- Loops can be expressed using parentheses and the iteration count, this means XYXY can be written as 2(XY).

This short hand notation allows to express complex schedules using only a few characters and is supported by the scheduler implemented along with this thesis.

In the example used in this section, the presented algorithms return two possible sequences: $3A2B3D4C$ or $3A3D2B4C$.

5.3 Ethernut

Ethernut [6] is an Open Source Hardware and Software Project for building Embedded Ethernet Devices. The hardware part is a small board with similar modules as on the BTnode. The operating system, Nut/OS, developed for the Ethernut board, is being adapted at the TIK for the BTnode, so that it can handle the Bluetooth interface.

Nut/OS has been chosen for this project as background operating system so that a Bluetooth communication could be added to the system. The drivers needed were still under development when this project was conducted. The interfaces of the different threads have been designed to make it possible to use a Bluetooth connection to upload an application for the BTnodeFPGA board.

The main reason for choosing Nut/OS was the fact that it supports threads. The scheduling is priority based and the threads need to yield themselves in order to give other threads a chance to run. This system is very basic, but suffices for the processes needed for this project.

5.4 User Interface

The user interface consists of a terminal application that runs over a software controlled UART interface. Both modules have been written at the TIK for other projects and have been imported to this project.

A user of the *Embedded Task Machine* can connect to the system through the software UART interface of the BTnode using communication tools such as telnet on UNIX or Hyperterminal on Windows. The user then can enter commands on a console provided by the terminal application on the BTnode.

The serial interface needs to be set to the fairly slow rate of 2400 baud/second to assure correct communication to and from the board. Higher rates may work, but sporadic errors are introduced into the streams going both ways. However, since only human typed commands and its responses are passed over this interface, the communication speed should be sufficient.

The terminal application has been extended with some commands to build the structure of a SDF graph, to evaluate it, and to execute it. These commands essentially wrap the messages the scheduler accepts to perform these actions. The answer messages from the scheduler are checked to ensure the command has correctly executed in the scheduler. On a failure,

`help`	Lists all available commands (short list TAB-TAB)
`threads`	Lists all running threads in Nut/OS
`alive`	Tests if the `BTnodeFPGA` board is active by reconfiguring FPGA with the task in slot 0
`process`	Defines a new task
`relation`	Defines a new arc between the tasks
`input`	Enters initial data into a FIFO
`sequence`	Defines the task execution sequence (schedule)
`start`	Starts the execution of the application
`configure`	Creates the FIFO setup in the RAM

Table 5.1: Commands accepted by the RS232 software

a message is printed to inform the user. The implemented commands are listed in Table 5.2.

The sequence of commands for an execution of an application is the following:

1. first some calls of `process` to define all the tasks
2. some calls of `relation` to interconnect the tasks
3. *optional:* initial data can be provided with `input`
4. `sequence` enters the schedule of the tasks
5. `configure` sets up the data structure in the RAM
6. The execution of the application finally is started with `start`

The most important commands used to build the SDF structure are explained here:

process ID Slot

The different task configurations of the application that will be run on the `BTnodeFPGA` must be loaded in advance using the program described in Section 5.7. These configurations then reside in the different task slots of the Flash memory. With the `process` command, an `ID` can be assigned to a configuration in one of the slots. The `ID` can be any single upper- or lowercase letter.

relation InputID IPortNum OutputID OPortNum Size

The data flow between the tasks can be defined using the `relation` command. These relations are implemented as FIFOs[2] storing the data produced by the task `OutputID` on its port `OPortNum`. The data will be read by the task identified by `InputID` on its port `IPortNum`. The size of the FIFO is set using `Size` and must be large enough to hold all the data produced by the `OutputID` task during its maximum amount of consequent iterations in the schedule.

Connections to the input and the output driver can be made by using the reserved process ID `I` for the input driver, and `O` for the output driver. The port numbers do not matter in this case since the drivers can only connect to one FIFO.

sequence SequenceString

`SequenceString` defines the execution sequence of the different tasks of an application. The syntax used for the sequence is further explained in Section 5.2.2.

input FIFONum Data

Using the input command, initial data can be stored to any of the defined relations. If the `relation` command has been successfully executed, a FIFO number is assigned to this relation and can be used as parameter `FIFONum` with the `input` command. The data is to be provided as decimal or hexadecimal[3] numbers, separated by spaces. The values stored in the FIFOs are 16-bit wide. Larger numbers will be truncated to that size.

[2] **F**irst **I**n **F**irst **O**ut buffers

[3] hexadecimal numbers use the prefix 0x as in the C programming language

As an example, the commands needed to create the SDF graph structure for the SDF graph presented in Figure 5.1 are presented here.

```
>process A 0
>process B 1
>process C 2
>process D 3
>relation B 0 A 0 6
>relation C 0 A 1 24
>relation D 0 A 2 3
>relation C 1 B 1 4
>relation C 2 D 1 12
>sequence 3A2B3D4C
>configure
>start
```

When the schedule is done executing, the terminal returns and displays the time needed for the whole schedule iteration. The execution can then be restarted with the same settings by issuing another `start` command.

5.5 Messages

One of the biggest drawbacks of `Nut/OS` is the lack of a message passing system. A rudimentary system has been implemented to provide a means of communication between the different threads of the *Embedded Task Machine* project. The requirements were the following:

- simple use
- no use of heap memory for efficiency considerations
- can be used to synchronize threads

The implemented system reserves for every thread a fixed amount of memory for message data and two semaphores for the synchronization of thread access to this data. The first semaphore blocks a sending thread if an older message has not been processed. The second semaphore blocks a reading thread until a message has been sent. Non-blocking functions have not been implemented since the thread scheduling algorithm of `Nut/OS` requires

a thread of high priority to suspend itself in order to allow a low priority thread to run.

The message passing system is implemented through four functions:

- All data needed is set up with `InitMessageQueues`. It must be called before any other message function is called.
- `SendMessage` sends a message. The function blocks if the receiving thread has not yet read its last message.
- Messages can be retrieved using the `RecvMessage` function. This function blocks until a message has been sent to the thread.
- `ClearMessage` must be called by the thread that received the message when it is done processing it. Otherwise, the message will not be cleared and no further message can be sent to this thread by any other threads. If `ClearMessage` is not called, any thread trying to send a message to the thread with the uncleared message will also be blocked.

5.6 Scheduler

The scheduler thread does the actual work on the `BTnode`. It receives messages with commands which it executes. Basically these messages are the same as the UI commands listed in Table 5.2 (except `help` and `threads`, which are system commands). The most important data structures the scheduler works on are three linked lists describing the processes, the FIFOs, and the execution steps. For each of these lists, structures have been created, which hold the corresponding data.

5.6.1 Data structures

The `PROCESS` structure contains the `ID` assigned to a task and its location in the Flash, or its `Slot`. It further contains a list of pointers to the FIFOs connected to its ports. No difference is made here between input and output ports. All the information stored in the `PROCESS` structure is received through command messages.

The `FIFO` structure contains all data needed to allocate the FIFO memory on the `BTnodeFPGA` board. The data field `uiSize` is received by the scheduler with the `relation` command message. This field defines the number of addresses that will be reserved in the SRAM for this FIFO. It must be calculated along with the execution sequence entered with the `sequence` command.

If the FIFO connects to a driver, its field `cDriverFIFO` is set to the corresponding value, indicating that the driver must be rerouted to use this FIFO when configuring the FPGA for the new task. Together with the `uiID` value, it defines the routing information to the `BTnodeFPGA` board, where it is used by `FIFOControl` to configure its FIFO structure caches.

In case initial data is assigned to the FIFO by the `input` command message, this data is placed on the `pData` pointer and its length stored in the `uiDataLength` field.

The list of `EXECUTION_STEP` structures is created by interpreting the `SequenceString` passed along with the `sequence` command message. A diagram of how the parser works is shown in Figure 5.2. It creates a linked list of this structure which then can be used to execute the given schedule.

The structure can stand for two possible execution steps: a task or a loop. If it is a task, the `pProcess` pointer is valid and the associated task is executed `uiIterations` times. The execution then passes on to `pNext`.

If the structure represents a loop, execution continues with the next step pointed to by `pLoop`. The step behind `pLoop` is called `uiIterations` times, then `pNext` is called. In order to allow an execution of the loop without using the stack, the last step of the loop points back to the root loop structure.

If `pNext` is not valid, the sequence has ended and the execution stops. Figure 5.3 shows a possible execution list build with `EXECUTION_STEP` structures.

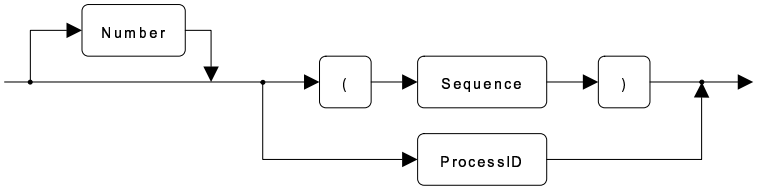

Figure 5.2: Schedule sequence parser diagram

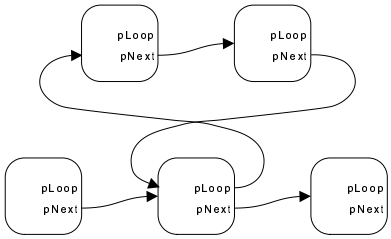

Figure 5.3: possible linkage of the `EXECUTION_STEP` structures

5.6.2 Communication with the BTnodeFPGA board

The scheduler writes commands to the `BTnodeFPGA` board directly to its UART interface. Answers of the module are received by a special thread which separates the output driver's data messages from the control messages send to the `BTnode`. Only control messages are passed on to the scheduler. The output driver data is converted to ASCII and directly printed to `stdout` for the lack of any target it could be send to.

5.7 Loading tasks

The `BTnode` software does not support loading of the FPGA configuration files holding the tasks. The main reason for the lack of this feature is that the software controlled UART interface is not reliable enough to run at high speed and introduces random errors into the stream.

As a workaround, an earlier test program written for direct communication with the `BTnodeFPGA` board can be used to preload the tasks into the `BTnodeFPGA` Flash before connecting the board to the `BTnode` and entering the SDF graph information. The tool is called `RS232Comm` and supports commands listed in Table 5.1. Instead of typing in the whole command, only the letter in parantheses must be used. Everything after the first space character is considered to be an argument for the command.

The following sequence of commands is used to load some tasks to the Flash. `load` takes two arguments: the slot to load the configuration to and the filename of the configuration.

(E)rase_Flash	Erases the whole Flash chip
(S)lot_Erase	Erases just one task slot on the Flash
(L)oad	Loads a FPGA configuration file into a slot
(R)ead	Reads back a slot into a file
(C)onfigure_FPGA	Configures the FPGA with a configuration from a slot
(B)yte_Send	Sends a byte over the RS232 interface
Send_(F)PGA_COMMAND	Sends a command for the FPGA
(W)rite_SRAM	Stores data from a file to the SRAM
Re(a)d_SRAM	Reads data from the SRAM into a file
E(x)ecute_Task	Executes a task
(H)elp	Prints the available commands
(Q)uit	Exits the program

Table 5.2: Commands accepted by the BTnode user interface

```
>e
>l 0 task1.bit
>l 1 task2.bit
...
```

To make sure the configuration has correctly been loaded into the Flash, a read back can be made using the command `r 0 test1.bit`. The newly created or overwritten file can then be checked against the original using tools as `diff` on Unix or `WinDiff` on Windows.

Chapter 6

Evaluation

6.1 Time Measurements

It is very important to consider on what ressources the different tasks of an application are to be run. It may happen, that a ressource may very well be much faster executing a task than the main central ressource, but if all the setup and result retrieval procedure needs too much time, it may be more efficient not to use the faster ressource.

Table 6.1 shows the measured time for the most significant processes when running an application. The times have been measured with a clock of 18.432 MHz on the `BTnodeFPGA` and the UART interface running at 115 kBaud.

The longest time is required to load a task configuration. This is because a full configuration file for the FPGA has a size of 160 kBytes which needs 14.5 s just to be transmitted over the UART interface. The same data is sent during a FPGA reconfiguration from the Flash to the FPGA and takes only 40 ms.

task loading	14500 ms
FPGA reconfiguration	40 ms
FIFO configuration	4.6 ms
starting a task	1 ms

Table 6.1: time requirements of significant processes

Task loading	62 mA	279 mW
FPGA reconfiguration	91 mA	410 mW
Demo application run	83 mA	374 mW
Idle	40 mA	180 mW

Table 6.2: power requirements of significant processes

6.2 Power Measurements

Since the purpose of the *Embedded Task Machine* is to use it as an intelligent sensor using batteries as the power supply, its power consumption is important .The `BTnode` and the `BTnodeFPGA` modules have been designed as low-power devices and thus should not consume too much energy.

The power requirements have been evaluated by measuring the current flow in the V_{DD} path to the `BTnodeFPGA` board. The current required by the `BTnode` has not been measured. The supply voltage was 4.5 Volts.

The results are shown in Table 6.2. The FPGA reconfiguration current has been measured by repetitively issuing reconfiguration commands using the `RS232Comm` software. To measure the current during the execution of a schedule, the demo application has been used which was controlled by the `BTnode`, which will be presented in Section 6.3. This demo application is not well suited for a power measurement, since the different tasks do not do much work and the FPGA is reconfigured very frequently. The result is that the reconfiguration current takes a major part of the measurement.

6.3 Demo application

To demonstrate the correct functionality of the *Embedded Task Machine*, a simple demo application has been written, which utilizes all implemented parts of the execution model. A simple division checker has been chosen to be implemented.

Figure 6.1 shows the SDF graph of the application. The task `A` is a counter which stores its current value and the increment in a self-loop. It produces the same data to two buffers. One holds the reference numbers, the other one is the input to the divider, which is represented as task `B`.

The result of the division is then multiplied by the same factor in task `C` and compared with the reference numbers in task `D`. The errors are added

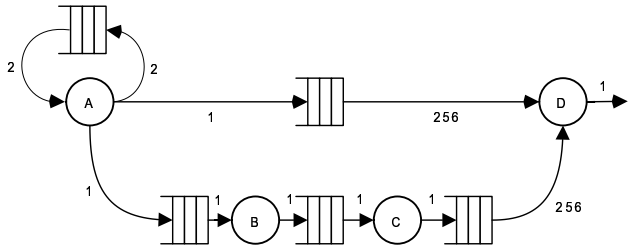

Figure 6.1: SDF graph of the demo application

up over 256 calculations and then written to the output task's FIFO.

6.3.1 Setup

In order to run the demo, the hardware first has to be configured. For this purpose, the devices have to be interconnected as shown in Figure A.5. In this configuration, the CPLD configuration can be uploaded over the JTAG chain of the `BTnodeFPGA` and the scheduler software can be loaded to the `BTnode` via the programmer.

As soon as the hardware has been configured, the tasks of the demo application have to be uploaded. The PC's COM port has to be connected to the `BTnodeFPGA` on the UART interface which is usually connected to the `BTnode`. The settings needed are listed in Table A.1.2. With the `RS232Comm` application (described in Section 5.7) the tasks can be loaded into the task repository slots with the following command sequence:

```
>e
>l 1 divmulcounter.bit
>l 2 divmuldivider.bit
>l 3 divmulmultiplier.bit
>l 4 divmulstat256.bit
```

The setup described here has only to be done to set up all the hardware. After turning off the power, the configuration up to this point will not be lost, and only the description in Section 6.3.2 has to be followed to run the application.

Note: There must be a configuration for the FPGA stored in *slot 0*. This configuration must include the support system for the task, but not

necessarily a task that can be run. The support system is needed by the `configure` command (see Section 6.3.2) to be able to write FIFO configuration data to the SRAM.

6.3.2 Running the Demo

If all hardware is configured as described in Section 6.3.1, the demo application can be run with the module interconnections shown in Figure A.4. The settings of the serial interface for the PC connected to the `BTnode` are given in Table A.1.2.

When the `BTnode` is started, `Nut/OS` enters the terminal application where commands can be entered. Details about these commands are given in Section 5.4. The SDF graph for the demo application is entered using the following commands:

```
>process A 0
>process B 1
>process C 2
>process D 3
>relation A 0 A 0 2
>relation D 0 A 1 256
>relation B 0 A 2 256
>relation C 0 B 1 256
>relation D 1 C 1 256
>relation O 0 D 2 256
>sequence 100(256A256B256CD)
```

Note: In the last `relation` command, the first parameter is the letter O, not the number zero as everywhere else.

The SDF graph has now completely been entered. The next step is to supply some initial data to the self-loop of task `A`. There are two items to be written: The increment (`0x0001`), and the current count (`0x0000`). After specifying this data, the demo application is ready to be run. The following commands will lead to the execution of the application:

```
>input 1 0x0001 0x0000
>configure
>start
```

After the execution, the application can be re-run with the last counter value by issuing another `start` command. Alternatively, the three commands above can be used to set another range for the counter and execute the SDF graph. Note that the `configure` command needs to be executed to write the FIFO data to the SRAM on the `BTnodeFPGA` board, and is thus needed to be executed after the `input` command.

During the execution of the application, the output task sends back the collected error information, in this case 100 times the following line:

```
Received from output driver: 0x0080
```

The division/multiplication tasks produce 128 errors. This is because the division task divides by 2 by simply cutting off the last bit. During multiplication, a zero is inserted at this place and thus the check fails on half of the supplied values.

Chapter 7

Summary

A proof of concept system of the *Embedded Task Machine* has successfully been implemented on the `BTnode` and the `BTnodeFPGA` board. It is able to execute applications that consists of a set of tasks and a specification of how they interact. The system includes a *task repository*, a *relation memory*, a *task runner*, and a *scheduler* which can execute a precalculated static schedule. An algorithm to calculate this schedule is presented in the report.

To verify the functionality of the system, a demo application has been built which needs all the different support functions in order to be correctly executed. The application has successfully been run on the system. During the development of the demo application, sample tasks and a adjustable task testbench have been developed which will hopefully assist in building applications for the *Embedded Task Machine*.

Appendix A

How to start the Embedded Task Machine

A.1 Cabling

Depending on how the *Embedded Task Machine* is used, a different wiring of the modules has to be used. It is possible to connect the `BTnodeFPGA` module to the `BTnode` or directly to a PC.

A.1.1 BTnodeFPGA

The `BTnodeFPPGA` is either connected to the `BTnode` using a wiring as shown in Figure A.1 or directly to a serial port on a PC. In the second case a level shifter must be used. A JTAG programmer must also be connected to set up the CPLD. The ports to use are shown in Figure A.2.

A.1.2 BTnode

The `BTnode` is connected to the `BTnodeFPGA` using a serial connection. A diagram of the required wiring is shown in Figure A.1. A level shifter has to be used to connect the BTnode to the RS232 interface of a PC. The programmer interface of the `BTnode` is only needed for development purposes. Figure A.3 shows the location of the ports on the `BTnode`.

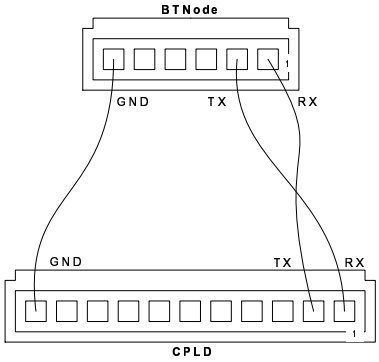

Figure A.1: The cable between `BTnodeFPGA` and `BTnode`. The connectors are shown in front view.

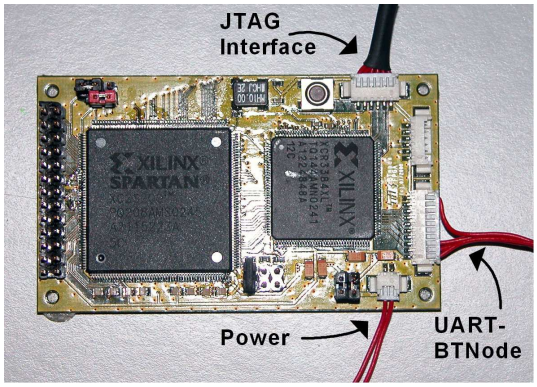

Figure A.2: The connections used at the BTnodeFPGA-board

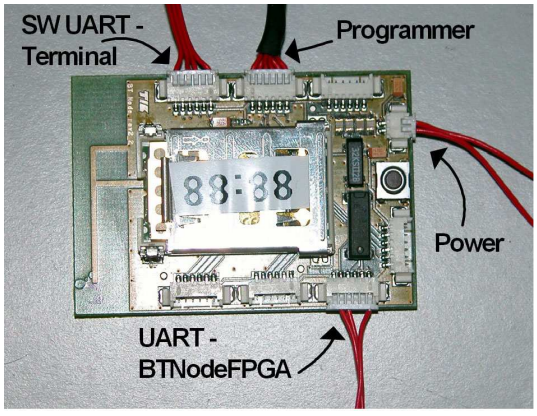

Figure A.3: The connections used at the BTnode-board, rev2.2.

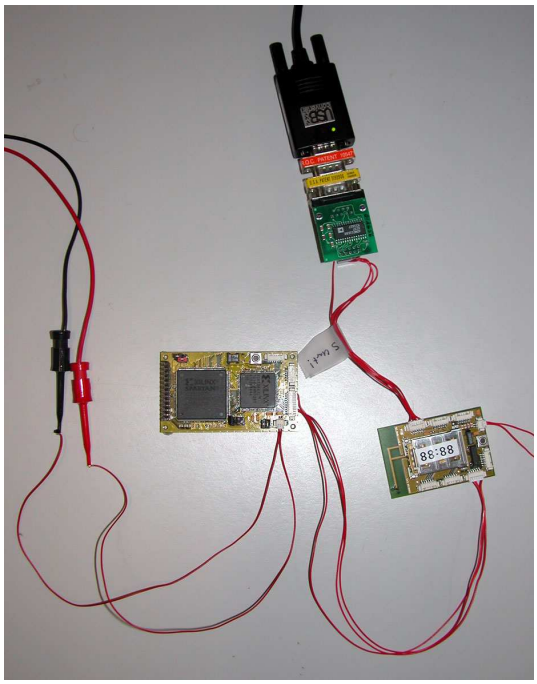

Figure A.4: Interconnections when using the *Embedded Task Machine*

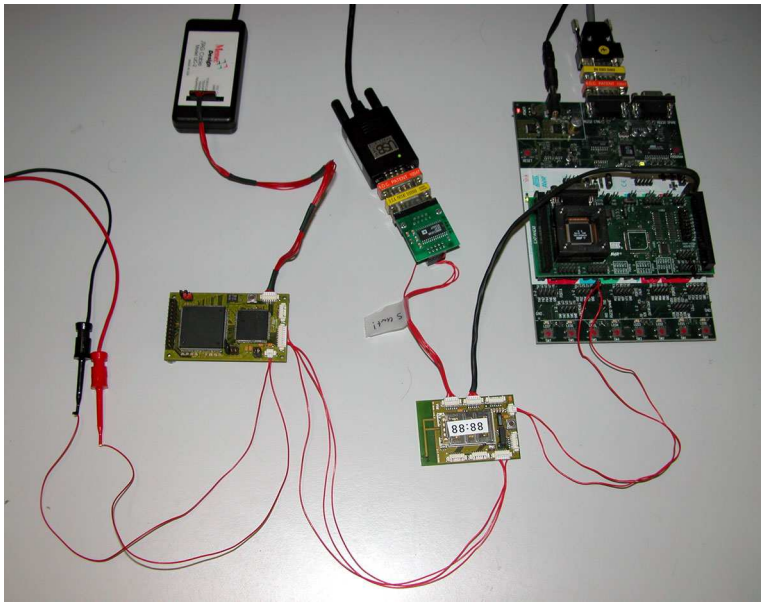

Figure A.5: Wiring during the development phase. The new devices are the JTAG programmer for the `BTnodeFPGA` and the programmer for the `BTnode`

Property	Value
Baud rate	2400
Data bits	8
Parity	None
Stop bits	1

Table A.1: Connection properties for the UART connection to the `BTnode`.

Property	Value
Baud rate	115200
Data bits	8
Parity	None
Stop bits	1

Table A.2: Connection properties for the UART connection to the `BTnodeFPGA`

A.2 Task setup

Before the *Embedded Task Machine* can be started, the tasks must be loaded into the task repository (Flash) on the `BTnodeFPGA` board. This can only be done using a PC. The `BTnodeFPGA` has to be connected using a level shifter to the PCs serial port and the `RS232Comm` application has to be started. See Section 5.7 for a description of this program.

The `e` command erases the complete Flash chip. Then the `l` command can be used to load task files to the different task slots on the Flash.

A.3 Running the Scheduler

The `BTnodeFPGA` needs to be connected to the `BTnode` and the `BTnode` to the PCs serial port. A list of commands understood by the scheduler is listed in Section 5.4. How the demo application is run is explained in further detail in Section 6.3.

Appendix B

Future Work

Many things have been implemented during the progress of this project. The *Embedded Task Machine* has been shown to be working and satisfactorily performs on the applications supplied. However, there are more features that could be added to the system, some of which are presented in this chapter.

B.1 Partial Reconfiguration

In the current system, the whole FPGA is reconfigured when the tasks are exchanged. Since the task support system as well as the input and output drivers are always present in all the configurations loaded onto the FPGA, it would be nice to be able to leave these parts of the FPGA untouched during the reconfiguration process. This would have different advantages.

Firstly, the configuration files would become smaller, allowing more tasks to be stored in the task repository on the Flash. The smaller configurations would also be faster to load onto the FPGA, increasing not only the application's maximum size, but also its execution speed.

Secondly, a partial reconfiguration would also allow the input and output drivers to run continuously, omitting a possible data loss during the reconfiguration time. Since the external interfaces are not served during at least the 40ms of reconfiguration time, a noticeable data loss is introduced into a stream with the current system.

A drawback is that the current FIFO access structure would not work with the partial reconfiguration. A concept for an adapted `FIFOControl` structure is presented in Section B.7.

B.2 Several Task Slots

Along with the partial reconfiguration comes the wish for several task slots. These slots would be separately reconfigured and could be run simultaneously. This would give the scheduler more possibilities to place the tasks in space and time.

Placing and executing several tasks has been investigated in [14]. Problems have been encountered around the placing of the tasks into the different slots. Different configuration files are needed for different slots and could not be exchanged. For a flexible schedule, many tasks configurations would thus be needed, occupying a lot of space on the Flash.

An additional problem is the size of the FPGA. It has to be well considered if the size of a single task slot would be big enough to supply enough area for a reasonable task complexity.

The `FIFOControl` entity would again not be able to handle several tasks at its current configuration and would have to be replaced by the structure presented in Section B.7.

B.3 Extension to Kahn Process Networks

The current *Embedded Task Machine* executes an SDF graph, which allows to compute a static schedule before the execution of an application. After loading the task, it is executed for a fixed number of iterations. The model requires that all the rates on the graph are known a priori. A *Kahn Process Network* (KPN) would run a task until one of its inputs is empty on a read access, or one of the outputs is full on a write access. The KPN requires dynamic scheduling that recalculates the next task to be loaded at runtime.

The KPN model introduces a major problem: since a task is preempted during its execution, its internal state must be stored until it is loaded again. In [5] a solution with a scan chain has been presented. It is an elegant solution but no tests have been made to determine if it actually works and how much additional wiring resources are used up such a system.

A simple solution would be to leave it up to the task to save its internal state. This would require a deeper insight into the system by the engineer and is thus not an acceptable solution.

Additionally a mechanism would have to be provided to give the scheduler some information about the FIFO states after the execution of the

tasks. The scheduler needs to know this information in order to compute the optimal successor task.

B.4 BRAM access

In order to provide more fast memory to the task, access to the FPGAs BRAM resources could somehow be granted. A task would have additional memory during its execution and such a storage would not have to be modeled as self-loops in a SDF graph.

B.5 Bluetooth

The next step for the project would be to make it possible to use the Bluetooth interface on the `BTnode` module to load applications to the *Embedded Task Machine*. This should not pose too harsh problems as soon as the drivers for the Bluetooth protocol stack are available.

What remains to be developed is some kind of buffer mechanism to receive the task configurations over Bluetooth and sending it to the task repository on the `BTnodeFPGA` Flash. An interpreter for the coordination description would also be needed. The interpreter would have to generate the messages for the scheduler thread described in Section 5.6.

B.6 Error handling

A big drawback of the current system is that it does not include any real error generating and handling system. Some processes have a success/failure feedback but not a real error handling. Many processes do not even detect if they have failed and the system could run into a deadlock.

Although the experiences made so far have been very positive, and the system seems to be generally stable, it would be nice to have more security on this behalf.

B.7 Improved `FIFOControl` architecture

The current `FIFOControl` functionality lacks two features: it cannot be extended for several independently running tasks and it does not support a

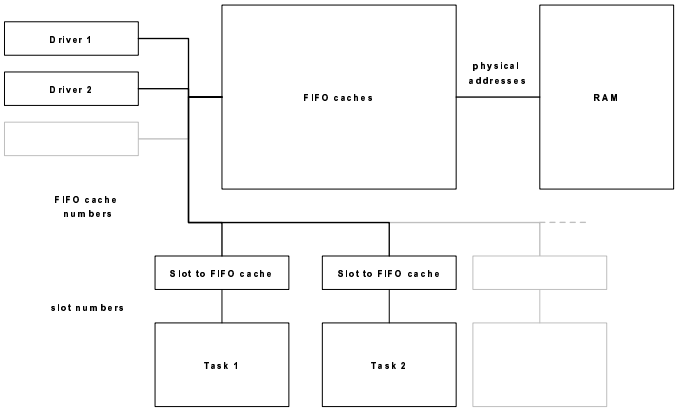

Figure B.1: Proposed `FIFOControl` structure

separate partial reconfiguration of a single task. This section presents a solution for the `FIFOControl` architecture that would provide the desired features.

The current system requires the drivers to change their FIFO caches if the task is connected to a FIFO used by a driver. A task can only access the first few FIFO caches in the `FIFOControl`, the other FIFO caches are reserved for the drivers when running on different FIFOs than the ones the task uses.

A pretty simple solution with the additional features would be to introduce an additional set of registers which map the task ports to the FIFO caches. This way, the port number the task provides on a memory access would first be translated to a FIFO cache number using a multiplexer, then the physical address would be calculated using the cached FIFO access structure. The new structure is shown in Figure B.1.

A driver would always access the same FIFO and thus this FIFO cache would never have to be written to the SRAM or read from there once the application has started. The `FIFOControl` would have to be changed in a way that it supports driver memory accesses during a new configuration of the tasks port mapping and the FIFO access structures loading procedure.

This solution has the same drawback as the current solution does: it does not scale well. It utilizes a lot of registers for the cache and for the mappings. Would even more concurrent task be allowed, the number of registers would increase very fast. The total number of registers needed is

given by the following formula:

$$r_{total} = (n \cdot s + d) \cdot 4 \cdot 18 + n \cdot s \log_2(F_a) + n \log_2(s) \log_2(n \cdot s + d)$$

where n denotes the number of concurrent tasks, s is the allowed number of ports per task, d is the number of drivers connected to one FIFO, and F_a is the total number of FIFOs that can be used in an application which is confined by the MC-Protocol as $2^6 = 64$.

This means that a system with 3 concurrent tasks, 4 allowed ports per task, and 2 drivers would use 1104 registers, which is quite a lot, considering the FPGA on the `BTnodeFPGA` has 4704 slice flip flops. In this case, the space left for the tasks on the FPGA would be very slim.

Bibliography

[1] BTnodes – A Distributed Environment for Prototyping Ad Hoc Networks. http://www.btnode.ethz.ch.

[2] NCCR - MICS – Project IP9 Abstract. http://www.mics.ch/micsProjects.php?groupName=IP9&action=abstract.

[3] Wearable Computing Lab. ETH Zrich. http://www.wearable.ethz.ch.

[4] Matthias Dyer and Marco Platzner and Lothar Thiele. Efficient Execution of Process Networks on a Reconfigurable Hardware Virtual Machine. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page to appear, April 2003.

[5] Roman Plessl. Downloadable Hardware and DSP for Mobile FPGA. Master's thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2004.

[6] Ethernut – Embedded Ethernet. http://www.ethernut.de.

[7] Peter Fercher. Btnode fpga: Mobiler fpga mit bluetooth kommunikation. diploma thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 2003. Under the supervision of Professor Lothar Thiele.

[8] *Am29LV081B Data Sheet.*

[9] *Spartan-II 2.5V FPGA Family:Functional Description.*

[10] *LP62S16256E-I Series: 256K X 16 BIT LOW VOLTAGE CMOS SRAM.*

[11] David G. Messerschmitt Edward A.Lee. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[12] David G. Messerschmitt Edward A.Lee. Static scheduling of synchronous data flow programs for digital signal processing. In *IEEE Trans. Comput.*, 1987.

[13] Shuvra S. Bhattacharyya, Joseph T. Buck, Soonhoi Ha, Edward A. Lee. A compiler scheduling framework for minimizing memory requirements of multirate dsp systems represented as dataflow graphs. Memorandum No. UCD/ERL M93/31, Department of Electrical Engineering and Computer Sciences, University of California, Bekeley, April 1993.

[14] Simon Steinegger. Hardware Tasks auf FPGAs. Master's thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2004.