Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

DIPLOMA THESIS

Design of a Flexible Data Path for Heterogenous Coarse-Grain Reconfigurable Logic Circuits

Submitted at the Faculty of Electrical Engineering and Information Technology, Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ. Prof. Dr. habil. Christoph Grimm
Dipl.-Ing. Johann Glaser

by

Martin Schmölzer, BSc
Matr.Nr. 0426583
Graffgasse 10/3/6/3601, 1100 Wien

Vienna, May 30, 2014

Kurzfassung

Konventionelle ASICs werden typischerweise für eine genau spezifizierte Anwendung entwickelt. Verändern sich nach Produktion die Anforderungen, muss mit hohem Zeitaufwand und hohen Kosten ein neuer ASIC entwickelt und produziert werden. Rekonfigurierbare Logikbausteine wie z.B. FPGAs können an Stelle eines ASIC eingesetzt werden, allerdings haben diese Bausteine üblicherweise gravierende Nachteile in Bezug auf Chipfläche, Energiebedarf und Geschwindigkeit. Alternativ dazu kann der ASIC mit *grobgranular rekonfigurierbarer Logik* ausgestattet werden, was eine Anpassung an veränderte Anforderungen mit einem gewissen Spielraum ermöglicht. Anhand des der High-Level-Synthese zugrundeliegenden *'Finite State Machine with Data Path'*-Modells wurde eine rekonfigurierbare Schaltungsarchitektur entwickelt, die eine möglichst hohe Flexibilität in Bezug auf den zu erzielenden Funktionsumfang aufweist. Zum Einsatz kommen rekonfigurierbare endliche Zustandsautomaten, verschiedene digitale Schaltungsblöcke sowie ein rekonfigurierbares Verbindungsnetzwerk zur Verbindung der einzelnen Schaltungsblöcke. Im Rahmen einer praktischen Anwendung, dem so genannten "Logic Exerciser", mit dem durch Rekonfiguration seiner grobgranular rekonfigurierbaren Schaltung digitale Signale erzeugt und analysiert werden können, wird die Flexibilität nachgewiesen.

Abstract

Conventional ASICs are typically designed for a precisely specified application. If the requirements change after production, a new ASIC has to be developed and manufactured, which are very time-consuming and costly tasks. Reconfigurable logic such as FPGAs can be used instead, but these devices usually have significant disadvantages in terms of chip area, power efficiency and performance. Alternatively, *coarse-grain reconfigurable logic* can be included in the ASIC, which allows for adaptations to changing requirements within certain bounds, but retains many of the ASIC's advantages such as power efficiency and speed. Based on the *'Finite State Machine with Data Path'* model, which forms the basis for high level synthesis, a highly flexible corase-grain reconfigurable digital circuit architecture, consisting of reconfigurable finite state machines, various arithmetic and logical cells as well as a reconfigurable interconnect was developed. An example circuit, the so-called "Logic Exerciser", which is able to analyze and generate arbitrary digital signals by reconfiguration of its coarse-grain reconfigurable logic, is implemented to demonstrate the flexibility of the implemented solution.

Table of Contents

1 Introduction **1**

2 Digital Integrated Circuits **4**
2.1 (Re)configurable Logic Circuits . . . 5
2.1.1 Fine-Grain Reconfigurable Logic . . . 6
2.1.2 Coarse-Grain Reconfigurable Logic . . . 6
2.2 Design of Digital Integrated Circuits . . . 7
2.2.1 High Level Synthesis . . . 9
2.3 Coarse-Grain Reconfigurable Logic . . . 9
2.3.1 Reconfigurable Finite State Machines . . . 10
2.3.2 Reconfigurable Interconnect Technology . . . 14
2.3.3 Yosys . . . 17

3 Design of a Coarse-Grain Reconfigurable Architecture **20**
3.1 The Logic Exerciser Device . . . 20
3.2 Digital Circuit Structure . . . 22
3.2.1 Configuration and Parameterization . . . 24
3.3 Design Flow . . . 25
3.3.1 Specification of Coarse-Grain Cells . . . 26
3.3.2 Synthesizing an FSM into a TR-FSM . . . 27
3.3.3 Generation of the Reconfigurable Logic Module . . . 28
3.3.4 Post-Synthesis Design Flow . . . 30
3.4 Example Applications . . . 30
3.4.1 SPI Sniffer . . . 30
3.4.2 Protocol Analyzer for ADT7310 Sensor . . . 31
3.4.3 SPI Master . . . 32
3.4.4 I^2C Master . . . 32
3.4.5 Protocol Exerciser for ADT7410 Sensor . . . 33
3.4.6 Hardware Testbench . . . 34

4 Implementation of the Logic Exerciser **37**
4.1 Hardware Platform . . . 37
4.1.1 Microsemi SmartFusion . . . 37
4.1.2 Xilinx Zynq-7000 . . . 39

4.2 Digital Circuit . . . 40
4.2.1 Bus Interface . . . 41
4.2.2 Configuration Interface . . . 43
4.2.3 GPIO Port . . . 44
4.2.4 Reconfigurable Logic Module . . . 45
4.3 Firmware . . . 54
4.3.1 Communication between Logic Exerciser and Host . . . 54
4.4 Host Software . . . 55
4.4.1 Logic Exerciser Control Program . . . 55
4.4.2 Bitstream Generation for Data Path Cells . . . 56

5 Results and Discussion **58**
5.1 Example Applications . . . 58
5.1.1 SPI Sniffer . . . 59
5.1.2 Protocol Analyzer for ADT7310 Sensor . . . 60
5.1.3 SPI Master/MAX6682 Exerciser . . . 62
5.1.4 I^2C Master . . . 64
5.1.5 Protocol Exerciser for ADT7410 Sensor . . . 65
5.1.6 Hardware Testbench . . . 67
5.2 Cell Usage . . . 68
5.3 Resource Usage . . . 70

6 Summary **74**
6.1 Outlook . . . 76
6.1.1 Design Flow Automation . . . 76
6.1.2 Additions to the Tcl Interface . . . 77
6.1.3 Interrupt-Driven Retrieval of FIFO Data . . . 77

Literature **78**

Internet References **81**

Abbreviations

AHB	AMBA High-speed Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
AXI	AMBA eXtensible Interface
ASIC	Application Specific Integrated Circuit
CAM	Content Addressable Memory
CAN	Controller Area Network
CHLL	Configurable High-Level Logic
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DUT	Device Under Test
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSMD	Finite State Machine with Data Path
GAL	Generic Array Logic
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High Level Synthesis
I^2C	Inter-Integrated Circuit
IC	Integrated Circuit
ISO	International Organization for Standardization
I/O	Input/Output
IP	Internet Protocol
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LSB	Least Significant Bit
LUT	Look-Up Table
MAC	Media Access Controller
MSB	Most Significant Bit
MSS	Microcontroller Subsystem

OSI	Open System Interconnection
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PHY	PHYsical layer
PAL	Programmable Array Logic
PL	Programmable Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PS	Processing System
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memory
RTL	Register Transfer Level
SoC	System on Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TR-FSM	Transition-Based Reconfigurable Finite State Machine
UART	Universal Asynchronous Receiver Transmitter
UDP	Universal Datagram Protocol
USB	Universal Serial Bus
VHDL	Very high speed integrated circuit Hardware Description Language
VLSI	Very Large Scale Integration

1 Introduction

"Since the creation of human life and evolution of human intelligence, there has been no invention more important than the IC for the development and progress of mankind at least from the mid 20th century onwards."

–Arjun N. Saxena [Sax09, p. x]

Today, Integrated Circuits (ICs) are essential and ubiquitous parts of our modern civilization, influencing almost every part of our lives. In fact, it is most likely easier to find examples of technology that are *not* influenced by ICs than to list the vast number of things that either contain ICs themselves or have been manufactured by some kind of machinery which is controlled by ICs.

The invention of the IC dates back to the year 1958, when Jack Kilby demonstrated his first working example of an Integrated Circuit, a mesa transistor made from germanium. In 1959, the first monolithic (all elements of the circuit are manufactured on the same substrate) silicon IC was built by Robert Noyce with the help of technology invented by Jean Hoerni and Kurt Lehovec [Sax09]. The inventions by Kilby and Noyce contained only a small number of electronic devices, but with the adoption of the technology by the industry, this began to change rapidly: today's ICs can contain more than 2 billion transistors in a single die [38].

Modern Application Specific Integrated Circuits (ASICs) are a type of IC which are typically designed for a precisely specified application. In the early 1980s, the design and manufacturing of digital integrated circuits became more and more sophisticated, quickly replacing analog technology that was state of the art at that time [Raz01]. Although being unmatched in terms of performance and power efficiency, ASICs do have several disadvantages:

- The development of an ASIC is very time-consuming and the manufacturing process is complicated and error-prone. This means that the development costs for an ASIC are very high – typically, ASICs are economically advantageous only when produced in high quantities.
- Due to the high manufacturing costs, it is essential that the first design iteration produces a working product.
- If the requirements change or a mistake is found after an ASIC has been manufactured, large portions of the time-consuming and expensive design and manufacturing processes have to

be repeated. If the ASIC has already been deployed within another product (e.g. soldered onto printed circuit boards that are installed in control cabinets around the world), the follow-up costs may be high enough to bankrupt the manufacturer.

In recent years, reconfigurable digital logic devices gained popularity because they have key advantages over the design of an ASIC such as a much shorter time-to-market, greatly reduced nonrecurring engineering costs and the ability to redesign the functionality of the implemented logic circuit if requirements change or bugs are found [Wak01]. The latest – and currently most flexible – of these technologies is the Field Programmable Gate Array (FPGA), which contains relatively simple programmable logic blocks connected by a rich routing network [SML09]. With these devices it is possible to implement almost any digital logic circuit as long as the performance requirements can be met and a device with enough logic resources is available. FPGAs are also favored for products which are produced in low numbers that do not warrant the expensive development of an ASIC. However, the flexibility of FPGAs comes at the cost of a large silicon area overhead, large propagation delays and poor power efficiency [Har01].

Yet, for some applications, reconfigurability is a crucial feature. Consider a so-called Central Processing Unit (CPU) Offload Engine for a wireless sensor node as introduced by [GHDG09], which is designed to handle simple tasks such as gathering environment values from digital sensors while the CPU is in a low-power standby state; should the necessity arise to change the algorithm that is implemented by the CPU Offload Engine, a new ASIC would have to be produced, incurring huge manufacturing costs.

One possible approach to solve this conflict of goals are *Coarse-Grain Reconfigurable Logic Circuits*. This is an emerging research topic that aims to combine the advantages of current reconfigurable logic devices (outstanding flexibility, ease of use) and non-reconfigurable ASICs (excellent performance, low power consumption). Reconfigurable logic of this type employs larger functional blocks than the fine-grained Configurable Logic Blocks (CLBs) found in FPGAs and interconnect switches that operate on word-level signals instead on bit-level signaling as found in FPGAs.

The design of these logic circuits is a relatively new academic field. Although several architectures targeted at special applications such as multimedia processing exist [LPCM07], [MSK+99], a *generic* coarse-grain reconfigurable architecture which is able to implement arbitrary digital logic circuits is still missing. Therefore, the goals of this thesis are:

- Research of readily available tools and components for the design of coarse-grain reconfigurable circuits.
- Establishment of a systematic approach for the design of coarse-grain reconfigurable architectures.
- Implementation of a reconfigurable circuit and demonstration of its ability to implement generic real-world applications.
- Provision of suggestions for further research on coarse-grain reconfigurable logic architectures as well as ideas for the enhancement and automation of the design flow.

To achieve these objectives, a model for generic coarse-grain reconfigurable logic circuits – the Finite State Machine with Data Path (FSMD) model – is derived from the theory of High Level

Synthesis (HLS). This model provides us with two distinct logic blocks, the *control unit* and the *data path*, but lacks mechanisms for reconfigurability. Additions to the model are therefore proposed and existing tools and components that may be used to implement them are evaluated.

Next, a device called the "Logic Exerciser", which utilizes a reconfigurable logic circuit to analyze and generate arbitrary digital signals and will serve as a prototype for the coarse-grain reconfigurable architecture, is introduced. The specifications for this device and several example applications, which will be used to gain information about the operations that need to be implemented by the cells in the reconfigurable logic, are defined. A suitable System on Chip (SoC) hardware platform for the implementation of the Logic Exerciser is selected and the necessary software and logic to connect the reconfigurable circuit to an off-the-shelf CPU is implemented. Then, the real-world example applications specified earlier are developed, and based upon the information obtained during this phase, the cells which comprise the reconfigurable logic are specified and implemented. Finally, the same example applications are realized by the Logic Exerciser's resulting coarse-grain reconfigurable logic circuit.

The remainder of this thesis is structured as follows:

Chapter 2 begins with a short introduction to the manufacturing process of ASICs, followed by a classification of various reconfigurable and non-reconfigurable IC devices. Next, the concept of High Level Synthesis is described, which is based on the FSMD model introduced by [GR94]. This model is established as the basis for the development of a coarse-grain reconfigurable circuit architecture. Finally, existing tools for the development of coarse-grain reconfigurable circuits are presented.

In chapter 3, extensions to the FSMD model which are required for the model to serve as a basis for coarse-grain reconfigurable logic circuits are discussed. Furthermore, the requirements for the Logic Exerciser are defined and its implementation is planned in a top-down manner, including the introduction of the three main components of the reconfigurable logic circuit (The Transition-Based Reconfigurable Finite State Machine (TR-FSM), Data Path and Interconnect). Finally, several real-world example applications which shall be implemented by the coarse-grain reconfigurable logic circuit are specified.

Chapter 4 gives a detailed description of the implemented solution based on the requirements stated in chapter 3. The hardware platform for the Logic Exerciser as well as the soft- and firmware required to communicate with the reconfigurable logic circuit is presented. Finally, implementation details of the cells present in the coarse-grain reconfigurable logic circuit are illustrated.

In chapter 5, each of the real-world example applications is implemented to demonstrate the flexibility of the coarse-grain reconfigurable logic circuit.

Finally, chapter 6 summarizes the work conducted in the course of this thesis and provides suggestions for future enhancements of both the design flow and the implemented logic.

2 Digital Integrated Circuits

The production of an ASIC is a very complex process involving a multitude of steps performed in a clean room with no or very little margin of error. The manufacturing process starts with a *wafer*, usually cut from a very pure single-crystal silicon ingot which has a very low number of impurities and defects. Today, wafer diameters of up to 300 mm and more are common [51] while each *die* (the silicon "chip") has a much smaller surface area of up to a few hundred square millimeters, which means that multiple dies can be manufactured from a single wafer. After the wafers have been polished to remove defects that were introduced during the cutting process, a series of processing steps can be applied to the wafer, which typically include photolithography, oxidation, doping, deposition, etching and polishing [Raz01].

The design and manufacturing methods for ASICs can be divided into three different categories [EH91], each having their advantages and disadvantages:

Full Custom: As the name implies, the cells as well as the manufacturing steps – including masks for all layers – are geared towards the product being designed. It is also possible to integrate arbitrary analog blocks into the design. Optimizations with regard to area, power usage and speed can be performed at will. The downside of this approach is its enormous complexity and very high costs, making it profitable only for devices with very high sales volumes or very specialized needs. The development of a full custom ASIC also poses a higher risk of manufacturing non-functional devices than other design methods.

Standard Cell: To reduce or even eliminate the risk of manufacturing non-functional devices, cells from a cell library – which have already been verified and optimized – are used instead of custom cells. Typically, the cell library contains a large number of logic functions such as AND, OR, NOT gates as well as storage elements such as flip flops, each in numerous variants. The manufacturing costs are still high because the masks for all layers have to be designed, but there is a significantly lower risk of manufacturing erroneous devices.

Gate Array: In this technology, basic cells are arranged in a regular structure. A large portion of the masks can be developed independently of the logic design, they are used to pre-fabricate wafers containing incomplete dies, which are then finalized by designing only the masks required to establish connectivity between the basic cells on the die, and performing the corresponding manufacturing steps. This means that the Gate Array approach is more cost-effective than Standard Cell or Full Custom designs, but it is also less flexible and not as much optimizable.

However, all of the variants described above have one thing in common: the cost for designing ASICs are very high, and once they are manufactured, it is no longer possible to change their functionality. To eliminate these disadvantages, extensive research on *configurable* (also called *programmable*) digital ICs has already been conducted.

2.1 (Re)configurable Logic Circuits

Configurable digital circuits date back to the 1970s, when a mask-programmable device, the TMS2000 [Tex73], was developed by Texas Instruments. This device was "programmed" by altering the metal interconnection layer during the manufacturing process, making the device configurable, but not *user-configurable*. Later, user-configurable variants utilizing antifuse technology emerged. These devices, called Programmable Logic Arrays (PLAs), consisted of a programmable AND structure and a programmable OR structure connected by configurable connections [Wak01]. Due to this structure, it is possible to directly map combinational logic in disjunctive normal form (also called "Sum of Products") onto these devices. Another variant of this circuit is the Programmable Array Logic (PAL), which uses a fixed OR structure instead of a programmable one [Wak01]. One of the first programmable logic devices that were able to implement sequential logic was the Generic Array Logic (GAL), which is similar to Programmable Array Logic devices but also includes flip flops. These devices were also the first to provide the ability for erasure and re-configuration [Wak01].

Today, devices of these types are typically referred to under the term Programmable Logic Device (PLD). Since the complexity of digital circuits evolved rapidly, the two-level AND-OR structure of these devices was soon unable to meet the growing requirements. This led to the invention of Complex Programmable Logic Devices (CPLDs), which consist of several PLD units integrated into one device and connected by a user-configurable interconnect. Another approach to meet the growing demand for powerful reconfigurable logic devices is to design a device with even smaller programmable logic blocks than those found in CPLDs and add a large, distributed and programmable interconnect. These devices are called FPGAs [Wak01].

FPGAs contain relatively simple Configurable Logic Blocks (CLBs) connected by a rich routing network [SML09]. For example, a CLB of the Xilinx 7 Series FPGA technology consists of four six-input Look-Up Tables (LUTs), which are able to implement arbitrary boolean logic functions, eight single-bit storage elements (flip-flops or latches) as well as multiplexers and carry logic [56]. The routing network of FPGAs typically operates on bit-level signals.

The behavior of an FPGA is determined by its *configuration bitstream*, which needs to be stored in on-chip memory. Static Random Access Memory (SRAM) is the most widely used type of memory for this purpose, although other technologies such as Flash-based FPGAs exist. The configuration bitstream is generated as the last step in the FPGA design flow and defines the behavior of the CLBs, the connections between the CLBs and Input/Output (I/O) cells, the characteristics of the I/O cells themselves (for example, multiple I/O standards may be selected for each pin) and customizable features such as the slew rate. After generation of the configuration bitstream, it has to be written into the FPGA's memory, which can be done manually (used typically during development, e.g. via a Joint Test Action Group (JTAG) interface), automatically (e.g. by storing it in off-chip non-volatile memory which is read by the FPGA device) or externally (e.g. an external device such as a microprocessor sends the configuration bitstream to the FPGA).

The CPLD and the FPGA are the only reconfigurable devices of those described above that remain in use today, with CPLDs being typically used for simpler applications and FPGAs being used for more complex applications. From the manufacturing point of view, all reconfigurable logic devices are also ASICs, the only difference is the fact that their purpose is not determined during production of the dies, but by the user.

2.1.1 Fine-Grain Reconfigurable Logic

Most modern PLD technologies have one thing in common: they focus on providing methods for the efficient implementation of relatively small combinational logic circuits (e.g. Sum-of-Product terms in CPLDs, LUTs in FPGAs) and offer single-bit registers (e.g. D-Flip-Flops) to realize sequential logic. Larger logic circuits are realized by interconnecting the individual logic blocks via a rich routing network which focuses on bit-level signals, hence the term *"Fine-Grain"*. In these technologies, the functional elements which are used to implement the actual logic only account for a small percentage of the total chip area while the routing network occupies up to 80–90 % of the available area, the remainder being used for the on-chip configuration memory [DeH96]. In addition, fine-grain reconfigurable logic devices typically suffer from poor performance due to large propagation delays (introduced by the routing network) and poor power efficiency [Har01]. However, the key advantage of these devices is their ability to implement almost any digital circuit as long as a device with suitable resources is available and the requirements for a high clock frequency are not too stringent.

2.1.2 Coarse-Grain Reconfigurable Logic

Coarse-grain reconfigurable logic is an emerging research topic which aims to mitigate the disadvantages of fine grain reconfigurable logic by making slight tradeoffs in terms of flexibility, but maintaining performance and energy efficiency comparable to ASICs. This is achieved by implementing larger, customized (and therefore highly optimizable) functional blocks that operate on word-level signals and interconnect switches optimized for these blocks. Existing work on this type of reconfigurable logic circuits often focuses on specialized applications such as multimedia or mobile communications. Examples include:

The CHESS array [MSK+99] focuses on the acceleration of multimedia applications such as video decoding. It consists of an array of Arithmetic Logic Units (ALUs) which are capable of performing arithmetic and logic operations. Each ALU is connected to a *switchbox*, which establishes connectivity with other ALU cells. These functional blocks can be reconfigured to perform alternate functions, e.g. a switchbox can be used as a small memory unit and an ALU can be used as a memory interface.

The Dynamically Reconfigurable Architecture for Mobile Systems [ABGS00] implements Reconfigurable Processing Units (RPUs) that contain two Reconfigurable Arithmetic Processing Units (RAPs), one so-called "Spreading Data Path" which implements specialized operations required for Quadrature Phase Shift Keying (QPSK) modulation often found in mobile communication networks, two dual-port Random Access Memorys (RAMs) and a communication protocol controller. The RAPs are reconfigurable 8-bit data paths which use another dual-port RAM as a LUT to perform fast arithmetic operations.

The Multimedia Oriented Reconfigurable Array [LPCM07] is a reconfigurable logic circuit for multimedia applications. Its reconfigurable cells (which consist of an input stage, a dual-port SRAM, a Processing Element (PE) and an output stage) are optimized for a pipelined data flow suitable for stream-oriented processing. The cells have two basic modes of operation: data loading, during which the contents of the dual-port SRAM can be modified, and execution, which is started as soon as all operands have been stored in the cell's memory. The processing elements are able to perform arithmetic operations such as addition, subtraction, accumulation, multiplication and multiply-accumulation.

The Embedded Field Programmable Gate Array [43] Intellectual Property core offered by the french company Menta S.A.S. allows ASIC designers to include reconfigurable logic in their designs. It consists of configurable logic blocks – similar to those found in FPGAs – which are connected via a programmable interconnect. One of the key features of the eFPGA is the ability to include customized functional elements of arbitrary complexity among the standard configurable logic blocks during the design phase. Also, the width of the interconnect buses is configurable (in contrast to most dedicated FPGA architectures, where it is focused on single-bit signals), thus creating a mixed environment of fine- and coarse-grain elements.

Another term closely related to this research topic is *Reconfigurable Computing*. [HHW90] suggests a major deviation from the von Neumann principles by replacing the instruction stream and the instruction sequencer in traditional CPUs by a data stream and a data sequencer. Sequential machine code is replaced by combinational machine code, which stores the configuration for reconfigurable hardware. This type of machine, called the Xputer, achieves performance that is far beyond that of traditional computers for a specific class of algorithms called Systolizable Algorithms, which are applicable in areas such as digital signal processing, image, speech, radar, sonar or seismic processing [HHW90].

2.2 Design of Digital Integrated Circuits

Many of the techniques and concepts for the design of Integrated Circuit (IC) can be demonstrated with help of the Y-Chart, which was first introduced by D. Gajski and R. Kuhn [GK83]. This chart shows three different representations of the IC on several different levels of abstraction:

Behavioral Representation: As the name implies, the behavioral representation deals with the behavior of a design. The highest level comprises the system specification, which is often written in natural language and includes requirements of the final product, often serving as part of the contract between a customer and the IC manufacturer. On the algorithmic level, the behavior of the IC is described in high-level languages such as SystemC, typically devoid of hardware implementation details. On the Register Transfer Level (RTL), the system can be described by the flow of data between registers, memories or ALUs. This is typically done with the help of Hardware Description Languages (HDLs) such as VHDL or Verilog.

Geometrical Representation: Here, the focus lies on the structure of the design in space (e.g. pins, bonding wires, packaging) or silicon (cells, cell geometry, masks used to manufacture the IC) – the purpose of the design, as specified by the behavioral description, is ignored as much as possible [GK83].

Structural Representation: This representation is a bridge between the behavioral and the geometrical representations – the behavioral representation is mapped onto a set of components such as CPUs, buses, flip flops or transistors. Prior to the availability of synthesis technology, ICs were designed on the structural representation either by hand or with the help of schematic capture tools. Also, analog components – which are essential to many digital ICs – are often designed in this way.

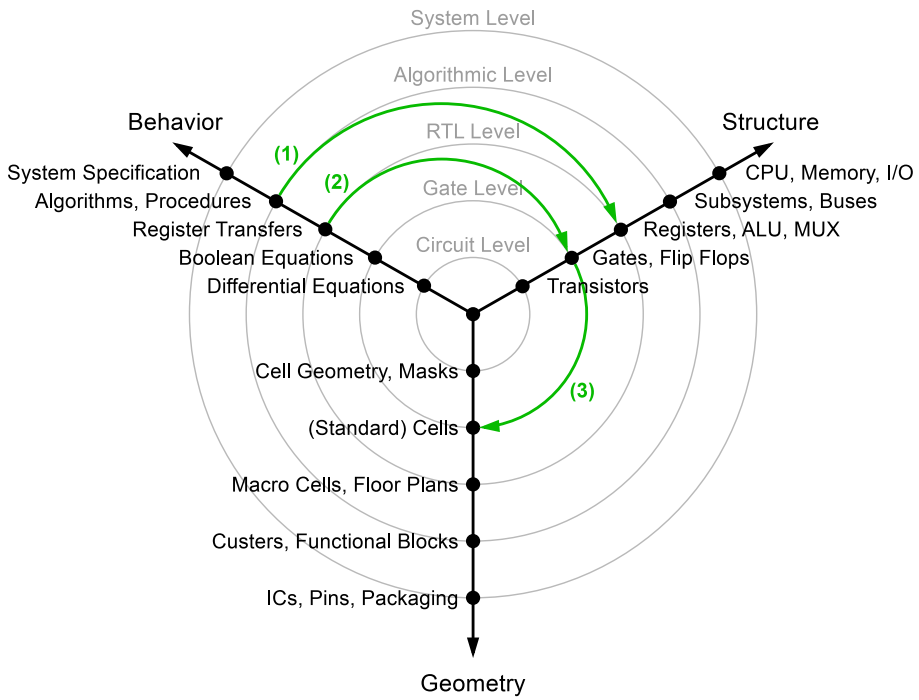


Figure 2.1: Extended Gajski-Kuhn Y-Chart

Figure 2.1 shows an extended version of the Gajski-Kuhn Y-Chart. Examples of transitions between the representations and between the different levels are marked by the green arrows and can be made with the help of IC design tools:

(1) High Level Synthesis: A behavioral description of an algorithm is synthesized into an RTL structure. This process is described in detail in section 2.2.1.

(2) RTL Synthesis: A behavioral description on the Register Transfer Level is synthesized to logic gates and flip flops. The most widely used Hardware Description Languages (HDLs) – Verilog and VHDL – were not initially designed for this purpose, instead, they were devised to document and simulate digital circuits [Wak01]. A few years later, the first RTL synthesis tools were available. While the implementations of synthesis tools may vary, a few basic steps involve *compilation* of the HDL code into an internal representation, *optimization* of this internal representation, and *technology mapping*, which is the process of mapping the internal representation onto elements of the target device family (e.g. ASIC standard cells, FPGA CLBs).

(3) Placement & Routing: This process marks the transition from the structural representation to the geometrical representation – the output of RTL Synthesis is mapped to cells that are available in the target architecture.

2.2.1 High Level Synthesis

Currently, a large portion of digital IC design is done on the Register Transfer Level (RTL). This was made possible by two key inventions: Hardware Description Languages (HDLs) and RTL Synthesis Technology. At this level, the design is typically created by writing HDL code – specifically, only a subset of *synthesizable* language constructs can be used – in a form that often mixes control logic with data processing.

However, for today's complex Very Large Scale Integration (VLSI) designs, the ability to describe an integrated circuit on the Algorithmic Level is desired to increase the productivity of digital design engineers – just like the invention of high-level programming languages such as C++ did for software engineers. A method to achieve this is High Level Synthesis (HLS), whose input is an algorithm independent of implementation details such as hardware properties and capabilities [Cil11]. An HLS tool performs two tasks to generate a system consisting of control logic and data path elements: *resource allocation* and *scheduling*. During resource allocation, the operators present in the input algorithm are identified and data path *cells* (e.g. registers, ALUs, memories) are generated depending on the sequentiality of the algorithm. Scheduling then binds the operations to specific clock cycles to implement the sequential data flow described by the algorithm [CGMT09, Cil11].

[GR94] introduces the FSMD model – a universal representation of hardware designs – as the basis for digital logic generated by High Level Synthesis tools. It consists of a control unit, which implements one or more Finite State Machines (FSMs) and a data path, which implements storage units (e.g. registers, counters, memories), functional units (e.g. ALUs, shifters, comparators) and interconnection units (e.g. wires, buses, multiplexers). During synthesis, the behavioral description of the design is mapped into the FSMD model, which is shown in figure 2.2 [GR94].

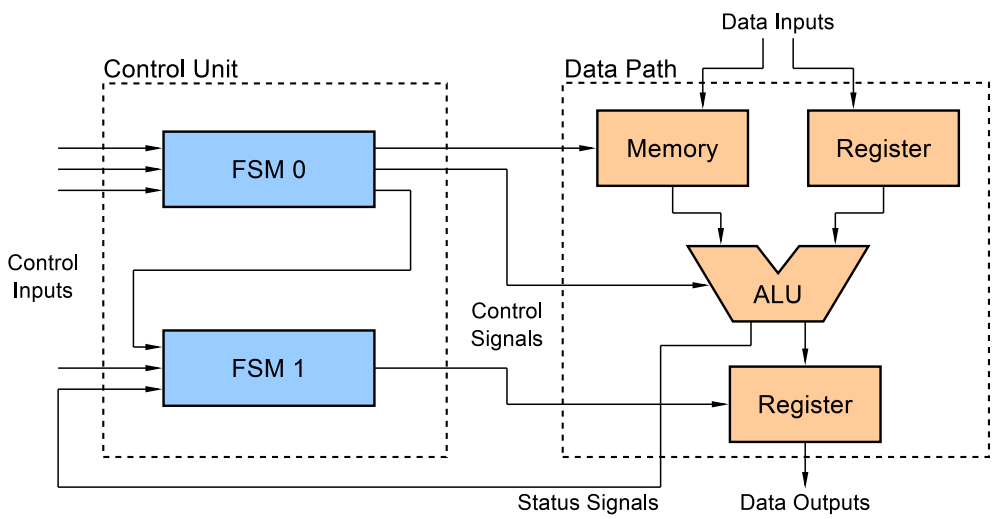


Figure 2.2: Finite State Machine with Data Path Model

2.3 Coarse-Grain Reconfigurable Logic

The FSMD model introduced in the previous section shares several similarities with coarse-grain reconfigurable architectures, making it a well-suited starting point for the development of such

an architecture:

- Strict separation of Control and Data Path Logic.
- The operation of data path elements is governed by the Control Logic.
- The connection between data path resources is based on word-level signals.

Still, the FSMD model does not provide any mechanism for reconfigurability. Several problems need to be addressed before a coarse-grain reconfigurable architecture based on the FSMD model which is able to implement real-world applications can be designed. For example, the FSMs in the control unit shown in figure 2.2 are generated in the scheduling phase of the HLS and mapped to logic cells. The connection between the control unit and the data path as well as the connection between the individual data path cells is fixed. Also, the cells of the data path are customized for the specific algorithm that serves as input to the HLS tool.

In summary, for our purposes, the FSMD model needs to be extended in the following ways:

- Reconfigurable FSMs must be used to implement the control unit.
- The connections between the control unit and the data path as well as between the individual data path cells must be reconfigurable and have a high degree of freedom.
- The cells of the data path must be able to implement a vast range of arithmetic and logic operations and provide ample storage mechanisms.

Possible solutions to these problems are investigated in the following sections.

2.3.1 Reconfigurable Finite State Machines

As mentioned in section 2.2.1, the control unit of a circuit synthesized by a High Level Synthesis (HLS) tool consists of one or more FSMs [GR94]. When synthesizing FSMs for implementation in an ASIC, the synthesis tool uses logic minimization algorithms to implement the transfer function and the output function as combinational logic [GDHG10]. For reconfigurable logic, this approach requires a fine-grained structure which implements the combinational logic (e.g. LUTs) connected to a state register. A more efficient approach is to provide a discrete implementation of a reconfigurable FSM. In this section, we will establish a few theoretical basics about FSMs, then present several approaches to implement discrete reconfigurable FSMs.

There are two basic types of FSMs: the *Moore* FSM and the *Mealy* FSM, named after their respective inventors, Edward F. Moore and George H. Mealy. In a Moore FSM, the output signals depend only on the current state, whereas the outputs of a Mealy FSM depend on the current state and the values of the input signals. Because it offers more flexibility, we will focus on the Mealy FSM for our coarse-grain reconfigurable logic architecture.

Formally, a Mealy Finite State Machine M can be defined as a seven-tuple $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$, where:

- Σ is the input alphabet,

- Ψ is the output alphabet,
- Q is the finite set of states,
- $\delta : Q \times \Sigma \mapsto Q$ is the transfer function,
- $\lambda : Q \times \Sigma \mapsto \Psi$ is the output function,
- s is the initial state,
- F is the set of final states ($F \subseteq Q$).

For FSMs implemented in digital circuits, each state can be a final state ($F = Q$), so our FSM definition is reduced to a six-tuple $M = (\Sigma, \Psi, Q, \delta, \lambda, s)$. To implement an FSM in a digital circuit, the transfer function δ and the output function λ is implemented by combinational logic, and the current state is encoded in a *state register*, as shown in figure 2.3. For a FSM with n_I input signals, 2^{n_I} transitions are possible in each state. With a state register width of n_S bits, there are 2^{n_S} possible states and therefore $n_T = 2^{n_I+n_S}$ possible transitions for the entire FSM.

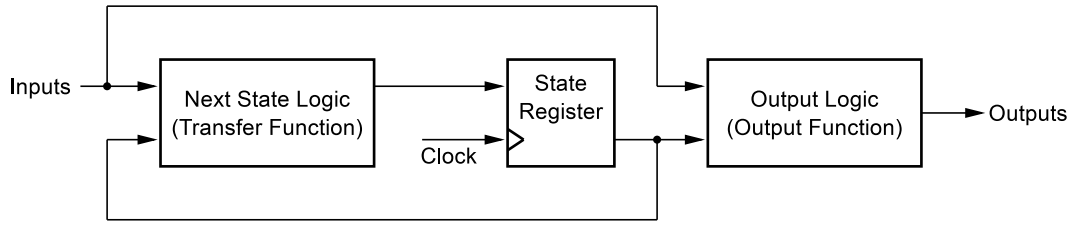


Figure 2.3: Generic Implementation of a Mealy FSM

In the following sections, we will explore several discrete implementations for reconfigurable FSMs that can be implemented in our coarse-grain reconfigurable logic circuit.

2.3.1.1 Random Access Memory Based Reconfigurable FSMs

A straightforward approach for the realization of a reconfigurable FSM is to replace the combinational logic which implements the transfer function and the output function by a RAM. The FSM input signals and the outputs of the state register (the current state) are used as address inputs to the RAM, while the contents of the RAM are split into the FSM output signals and the next state, which is fed back into the state register, as shown in figure 2.4. By changing the contents of the RAM, the FSM can be reconfigured. However, the disadvantage of this approach is that it requires a rather large RAM, which must contain one word consisting of $n_S + n_O$ bits for every possible transition [GDHG10]: for a small reconfigurable RAM-based FSM with six input signals, six output signals and four state bits, this approach requires 10240 memory bits. This approach is used by the CHESS array [MSK+99] introduced in section 2.1.2.

2.3.1.2 Run-Time Context Reconfiguration Based FSM

The reconfiguration of the FSM *context* at run-time, as introduced by [MV07], aims to minimize the combinational logic required to implement the FSM's transfer and output functions. Reconfigurable logic is used to implement only those parts of the transfer and output functions which are needed to calculate the next state from the input signals. After each state transition, the logic is reconfigured, creating a separate context for each of the implemented states. A dedicated

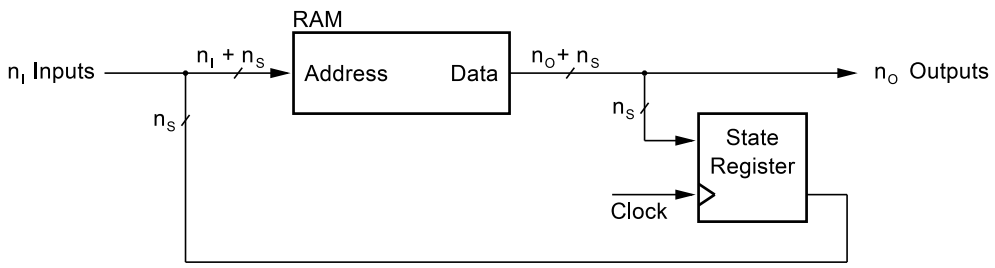


Figure 2.4: RAM-based FSM

configuration memory is used to store the bitstreams for each context. This approach requires considerably less logic or storage elements than other designs, but the periodic reconfiguration is disadvantageous for low-power applications [GDHG10]. Figure 2.5 shows the principle of this design.

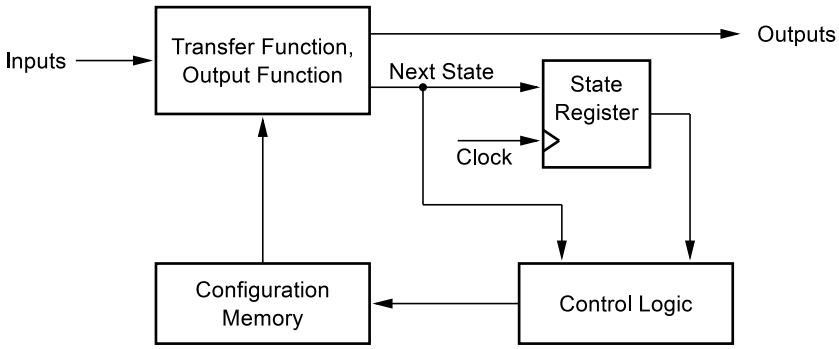


Figure 2.5: Run-Time Context Reconfiguration Based FSM

2.3.1.3 Transition-Based Reconfigurable Finite State Machine

In section 2.3.1, we established the maximum number of transitions in an FSM to be $n_T \leq 2^{n_I+n_S}$. However, most real-world FSMs implement a much lower number of transitions that may only depend on a subset of the input signals. The design of the TR-FSM exploits this fact by focusing on the transitions, rather than the implementation of the entire transfer and output functions. This is done by providing reconfigurable blocks called *Transition Rows*, which implement the transitions of the FSM. The modular design of the TR-FSM allows the inclusion of Transition Rows with varying *size* parameters, which determine the number of input signals that trigger the transition. Figure 2.6 shows the structure of the TR-FSM. A transition row (green) consists of the following reconfigurable logic blocks (yellow) [GDHG10]:

Input Switch Matrix (ISM): The Input Switch Matrix extracts a subset of the TR-FSM's input signals (those signals that can trigger a transition in the current state) and forwards it to the Input Pattern Gate (IPG).

State Selection Gate (SSG): The State Selection Gate contains a reconfigurable register that holds the encoding of the state in which the transition implemented by the transition row is allowed to occur. The contents of this register are compared to the contents of the State

Register (SR) – if they match, a logic "`1`" is output by the SSG, enabling the Input Pattern Gate.

Input Pattern Gate (IPG): The Input Pattern Gate determines if the conditions for the execution of the transition are met by comparing the input signals forwarded by the Input Switch Matrix to the values stored in a reconfigurable register, and asserts an output signal if they match.

Next State Register (NSR): The Next State Register contains the encoded destination state for the transition, which will be written into the State Register once the transition is takes place.

Output Pattern Register (OPR): The Output Pattern Register contains the state of the output signals for the given transition.

In transition rows with a size of zero, which are used to implement forced transitions that do not depend on any input signals, the ISM and IPG components are not present. Instead, the output of the SSG is directly connected to the next state selection block. In addition to the Transition Rows, the TR-FSM also contains several "global" components (orange), which are also depicted in figure 2.6:

State Register (SR): The state register is used to store the current state of the implemented FSM. There are no limitations to the actual encoding of the states, with one exception: there must be at least one unused state encoding which will be written to the SSGs of unused transition rows to prevent them from becoming active [GGHG11].

Output and Next State Selection Block (SEL): This module contains a multiplexer whose one-hot selection input[1] is driven by the outputs of the Input Pattern Gates. It is used to select the contents of the Next State Register as the input to the State Register and the contents of the Output Pattern Register as the active FSM output [GDHG10].

Output Register (OR): The optional Output Register was added as a performance optimization. In the TR-FSM, the path between the inputs and the outputs is formed exclusively by combinational logic. This means that a level change at the input ports has to propagate through the entire logic elements during a single clock cycle, leading to a long critical path and possibly to a lower achievable clock frequency of the entire design. Any combinational logic connected to the inputs or outputs of the TR-FSM further stresses the timing requirements. By registering the TR-FSM outputs, this effect is mitigated: although the outputs set by the TR-FSM now appear one clock cycle later, a higher achievable clock frequency of the entire design may be possible. The output registers can be activated by setting a specific configuration bit in the TR-FSM's configuration bitstream.

The VHDL implementation of the TR-FSM makes extensive use of generics, which allows to specify the size of the state register, the number of input and output signals as well as the number of transition rows and their respective sizes. The TR-FSM has already been implemented as part

[1] One-hot refers to a signal vector in which only one bit is asserted at a time. In this case, the width of the multiplexer selection input is equal to the number of transition rows in the TR-FSM.

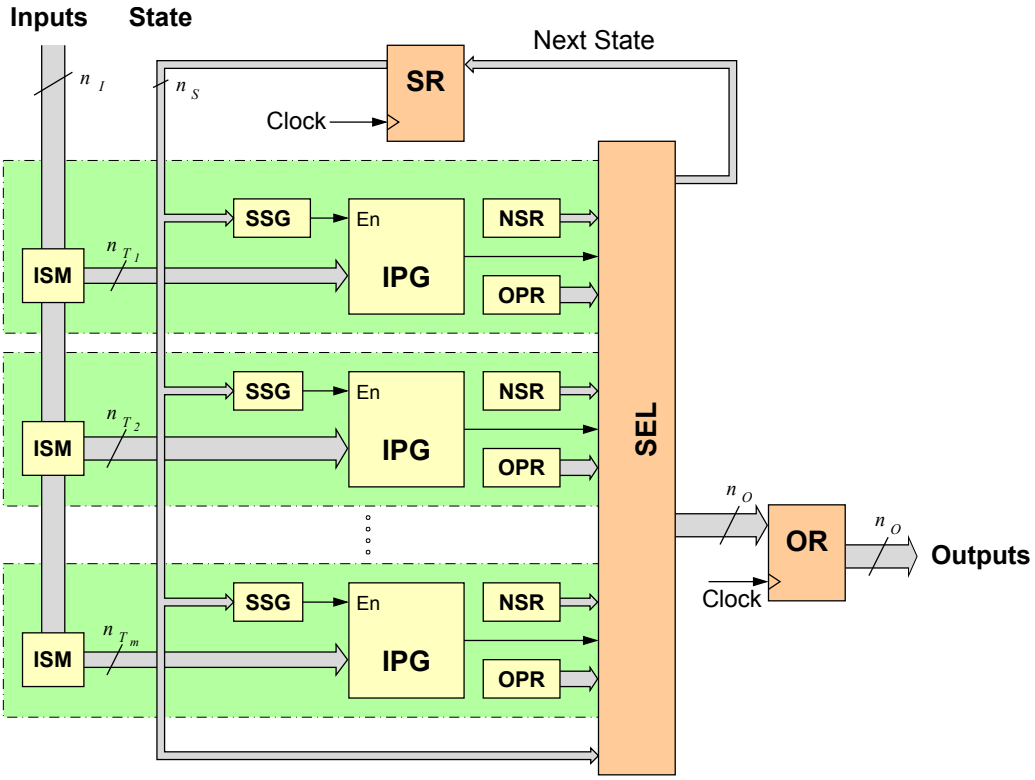


Figure 2.6: TR-FSM Structure (© 2010 by Johann Glaser)

of a test chip manufactured in an automotive-qualified general purpose 130 nm CMOS process [GGHG11].

A software tool named trfsmgen, which is written in the Pascal programming language and includes a Tool Command Language (Tcl) interface is available to generate the configuration bitstreams for the TR-FSM.

2.3.2 Reconfigurable Interconnect Technology

The resulting netlist or HDL code which is generated by HLS contains fixed mappings of input and output port pairs for the cells in the design. This is also the case when manually designing digital logic with the help of a hardware description language. When performing the placement and routing for an ASIC, these associations are transformed into the metal layer masks which define the connection "wires" between the transistor in the device. To be able to modify the connections between the cells after the design has been implemented in hardware, switchable elements (which form the reconfigurable interconnect) must be inserted at the appropriate locations in the design. This section introduces two different techniques for these interconnects.

2.3.2.1 Grid-Structured Interconnect

Several existing coarse-grain reconfigurable architectures designed for computational tasks use a two-dimensional grid arrangement of homogeneous processing elements, typically using direct connections to the adjacent cells with an added global routing network. An example of a grid-structured interconnect is shown in figure 2.7.

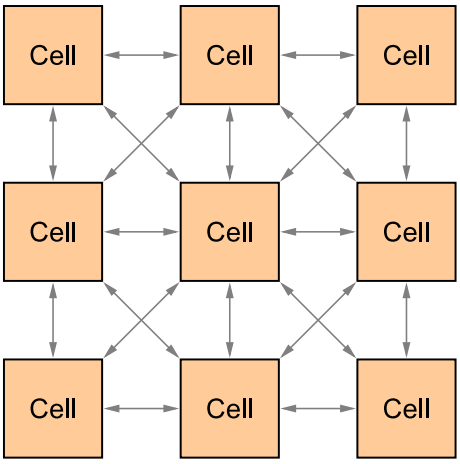


Figure 2.7: Exemplary Grid-Structured Interconnect

Three of the designs introduced in section 2.1.2 use a grid-structured interconnect:

CHESS: In the CHESS array [MSK+99], the functional blocks (ALUs and so-called switchboxes) are arranged in an alternating fashion, creating a layout comparable to a chess board. Each ALU is connected to a neighboring switchbox, which has access to 64 different horizontal and vertical connections of different length. The switchboxes also contain registers, which can be enabled for longer connections (with greater propagation delays) to keep a high clock speed.

DReAM: Each Reconfigurable Processing Unit (RPU) of the Dynamically Reconfigurable Architecture for Mobile Systems [ABGS00] is directly connected to its four neighbors via 16-bit bus signals. With the help of switching boxes, connectivity to global connection lines can be established. A hand-shaking protocol is used to transmit data between the RPUs.

MORA: Each processing element in the Multimedia Oriented Reconfigurable Array [LPCM07] has two input and four output data ports which can be connected to the ports of its eight neighboring cells via 8-bit unidirectional horizontally, vertically, and diagonally laid out data and address buses. Connections to more distant processing elements are realized by a second-level interleaved interconnect structure.

However, these architectures – having been developed for specialized applications such as multimedia processing or mobile communications – are not very well suited for the implementation of general-purpose logic. There are multiple downsides:

- The grid structure of processing elements negatively affects the routing degree of freedom: while neighboring cells can be connected easily and with low latency, longer connections

may need to be registered, causing additional propagation delays [MSK+99]. In contrast to multimedia applications, where many algorithms have regular execution patterns, we can't make any assumptions about execution patterns for general-purpose digital logic, thus efficient any-to-any connections are required.

- The necessity to explicitly write data into memory cells embedded in the processing elements before executing arithmetic or logic operations would impact the performance of many circuits (e.g. when several operations are executed by different functional units with the same operands, they would first have to be copied from one register to another).
- None of the presented architectures are suited for the efficient implementation of customized FSMs and the distribution of signals to and from the other functional units, which is a crucial feature for general-purpose reconfigurable logic.

2.3.2.2 InterSynth

In homogeneous architectures such as these introduced in section 2.1.2, the grid-structured interconnection technology fits well. A general-purpose coarse-grain reconfigurable circuit, however, has to be classified as a *heterogenous* architecture, which consists of many different functional units. Therefore, no assumptions can be made about the regularity of the structure, i.e. the interconnect must be able to provide random connections between cell ports. InterSynth [WGS+12] is a software tool which was designed for the generation of an optimized interconnect between heterogenous cells. An example-driven approach is used to gather information about the requirements for the interconnect: a set of application netlists (which contain a structural description of the application by instantiating cells of the reconfigurable logic circuit) is analyzed. Then, various optimizations are applied and finally, the interconnect along with the instantiations of the cells is generated as synthesizable Verilog code.

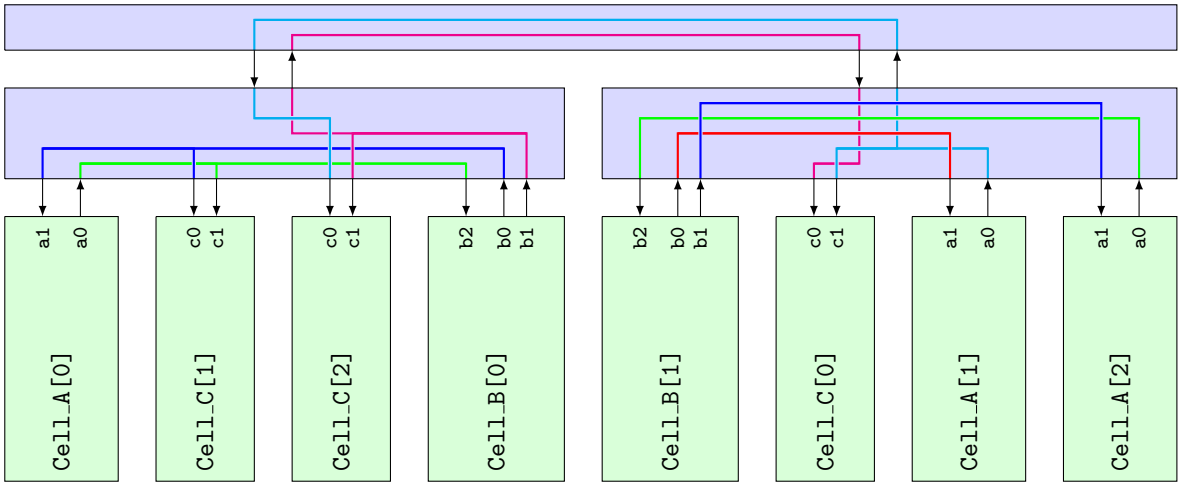


Figure 2.8: Exemplary interconnect tree generated by InterSynth

Every port of a cell in the coarse-grain reconfigurable logic is assigned a *connection type*, which are characterized by the number of bits and optionally by semantics (e.g. data bit or control signal). For each connection type, a separate interconnect is generated, which can either be implemented in a tree topology with a user-specified height (as shown in figure 2.8) or as a

fully multiplexed switch [54]. The tree topology can be tweaked by an optimizer based on the Kernighan-Lin algorithm [WGS+12]. The fully multiplexed switch allows every input port of a specific connection type to be routed to every output port of the switch, thus implementing true any-to-any connections for the cell ports [54], but with a very large hardware overhead.

InterSynth operates in two different phases, the *Pre-Silicon* and *Post-Silicon* phase [WGS+12]:

Pre-Silicon Phase: In this phase, the specifications of the reconfigurable module are derived from the example application netlists, the optimization algorithm is run and the reconfigurable module is generated and exported as a Verilog HDL file. This file is then synthesized in the following ASIC design flow. This phase also generates the bitstream data for all example application netlists that have been processes so far.

Post-Silicon Phase: This phase may be executed upon completion of the Pre-Silicon Phase to implement other applications than those that have been processed during the Pre-Silicon phase.

If the interconnect was implemented as a tree topology, the example applications that have been processed in this phase plus any application whose cell count and cell connection requirements do not exceed the implemented design can be realized by reconfiguration of the generated module. If the fully multiplexed switch topology was used, any application can be realized as long as its cell count does not exceed the specifications of the generated module.

2.3.3 Yosys

In the previous sections, several options for the extension of the FSMD model were introduced. However, without specialized tools, the actual process of integrating the functional units comprising the data path and the reconfigurable FSMs which implement the control unit requires a large amount of manual effort. Especially the manual specification of the behavior of the control module is a tedious task: during normal ASIC and FPGA design flows, finite state machines are described in an HDL, which is then simulated together with the remaining HDL modules. The HDL code can be adapted very quickly, which is not quite the case for reconfigurable FSMs, e.g. when the designer chooses to alter the number of input and/or output ports, the following tasks have to be completed before the design can be simulated again:

1. Alter the instantiation of the reconfigurable FSM in the HDL code.
2. Match the specification of the FSM in the tool which generates the configuration to the new instantiation parameters.
3. Change the behavior of the FSM, e.g. by modifying the transition table.
4. Re-generate the configuration of the reconfigurable FSM.
5. Re-run the simulation.

When simulating FSMs written in an HDL, only steps 1, 3 and 5 need to be executed. During development of a digital logic circuit, which is often an iterative process, this has to be done many times. Therefore, it is preferable to work with the FSM description written in an HDL, and switch to the reconfigurable FSM not until the point where the design is nearing completion. At this point, a tool which translates the HDL description of the FSMs to the format which is understood by the tools which generate the configuration of the reconfigurable FSMs is desirable. This can be added to an extensible synthesis tool which either has a "plug-in" interface or which has an open source code that is available for modification.

Today, the development of FPGA and ASIC designs is mainly done with the aid of commercial synthesis tools. These computer-aided design tools provide support for many design methods and are mature, yielding stable and efficient logic circuits for many target architectures. However, these tools are almost exclusively available under proprietary licenses (and therefore cannot be modified) and usually lack extension interfaces. Some existing open-source synthesis tools exist, but these are still under development and therefore unable to implement real-world designs. Due to these reasons, Clifford Wolf developed *Yosys* (**Y**osys **O**pen **SY**nthesis **S**uite), an extensible framework for digital synthesis as a basis for custom synthesis flows and as an environment for the research on new synthesis algorithms [WG13].

Yosys' main task is synthesis of Verilog HDL code "with the maximum of openness and extensibility" [WG13, p. 2]. This is achieved by a modular approach: the HDL code is first converted to an internal format called *Register Transfer Level Intermediate Language* (RTLIL) by the appropriate HDL front-end. Then, *passes* may analyze or modify the RTLIL representation of the design. Finally, a back-end is used to generate the output data in an appropriate format (e.g. a netlist). Some steps (optimization of gate-level netlists and mapping of combinational logic) are handled by an external tool called Berkeley ABC [WG13]. Because Yosys' source code is available under an open source license, customized passes can be added at will.

Yosys is able to perform mappings of the synthesized logic in various formats [WG13]:

ASIC Cell Mapping: Yosys can map the synthesized design to an ASIC standard cell library specified in the Liberty format. This is done by running the `dfflibmap` and `abc` passes: the `dfflibmap` pass replaces all flip flops in the design with those specified in the library. Finally, the `abc` pass invokes the Berkeley ABC tool to map the combinational logic to cells in the library (e.g. NOT, NAND, NOR).

FPGA LUT Mapping: The synthesized design can be mapped to LUTs and flip flop cells that are found in the target FPGA device. Initially, Yosys maps the synthesized design to an internal cell library, then Berkeley ABC is used to map this internal representation to generic n-input LUT cells and positive-edge flip flops. Finally, Yosys replaces these generic cells with the cells found in the target FPGA.

Coarse-Grain Mapping: Mapping the synthesized design to ASIC cells or FPGA LUTs is fundamentally different than mapping it to the cells of a coarse-grain reconfigurable logic circuit: when mapping to ASIC cells or internal cells which are then mapped to LUTs, the granularity of these cells is smaller or equal to the operators commonly found in HDLs. However, coarse-grain cells often implement more complex functions than these operators. Therefore, Yosys provides the `extract` pass which uses an algorithm that is based on the Ullmann Algorithm for Subgraph Isomorphisms to find occurrences of the coarse-grain cells

(defined in a netlist library) in the synthesized design. Yosys is also able to generate the configuration files for InterSynth (introduced in section 2.3.2.2) [55].

Yosys is controlled by executing synthesis scripts which contain a series of commands. Each command either executes a front-end, pass, or back-end. Commands can also be specified on the command line, allowing the integration with other work flow management systems [WG13].

Two Yosys features are particularly interesting for the design of coarse-grain reconfigurable logic circuits: the extensibility which stems from Yosys' open source nature and allows for the implementation of custom passes and the already present support for coarse-grain design flows, i.e. the extraction of coarse-grain cells and the ability to automatically generate InterSynth netlists from structural Verilog designs.

3 Design of a Coarse-Grain Reconfigurable Architecture

The design of software and hardware is often done as an incremental process. First, the requirements of the product are specified. Then, the design is broken down into smaller parts, the structure of the system is derived and interfaces between the modules are defined. Depending on the complexity of the design, more detailed requirements for the individual modules may be specified. During the following implementation phase, each module is then developed and tested, and the system is integrated. Finally, the resulting product undergoes verification to make sure it meets the requirements specified earlier.

In this chapter, the requirements of the Logic Exerciser device are presented and the system structure is planned in a top-down manner. Furthermore, the design flow for coarse-grain reconfigurable logic circuits is outlined. Also, a set of *Example Applications*, which serve as the specification of the reconfigurable logic module, are defined at the end of this chapter. These example applications are required for two reasons: to gain insight about the functionality and types of reconfigurable cells in the coarse-grain reconfigurable circuit and to demonstrate the flexibility of the resulting design.

3.1 The Logic Exerciser Device

The motivation for the development of the Logic Exerciser is to gather information about the arithmetic, logic and storage operations that need to be implemented by our coarse-grain reconfigurable architecture. Therefore, the main goal is to define a universal set of cells comprising the reconfigurable circuit so it is able to implement a broad range of applications.

The requirements can be summarized as follows:

1. **Universality:** The Logic Exerciser shall be able to analyze and generate digital signals with a primary focus on, but not limited to, serial bus interfaces. Here, "Universal" means that no specialized, dedicated hardware modules are included which handle signal analysis and generation for a particular interface or technology. Instead, a coarse-grain reconfigurable logic circuit shall be used to implement the signal analysis and generation.

2. **Connection to Host PC:** The Logic Exerciser shall be connected to a standard PC over readily available standardized low-cost interfaces. The purpose of this connection is the downloading of configuration data to the device and the transfer of generated or analyzed data to the PC.

3. **Graphical User Interface:** For the host PC, an application with a graphical user interface shall be developed, whose purpose is the presentation of data generated or captured by the Logic Exerciser.

4. **Off-the-shelf Hardware:** While coarse-grain reconfigurable hardware is normally targeted at ASICs, this would be far too expensive for this first real-world application which can be viewed as a proof-of-concept prototype. The only viable alternative to the development of an ASIC is an FPGA, which by itself constitutes reconfigurable hardware. However, this reconfigurability must not be used to implement the coarse-grain reconfigurable logic circuit, i.e. the FPGA's bitstream is not allowed to be changed when the Logic Exerciser is reconfigured between applications.

One of the requirements of the Logic Exerciser is its ability to communicate with a host PC, receiving commands from the software running on the host and transmitting captured and/or generated data to the host (requirement 2). Any standard Personal Computer shall be able to serve as host, thus the selection of the communication interface between the logic exerciser and the host is limited to interfaces commonly found in standard PCs. These include:

- Peripheral Component Interconnect (PCI) and its successor, PCI Express (PCIe),
- Universal Serial Bus (USB),
- Ethernet.

Older interfaces such as RS-232 (UART) or Parallel Port were not considered since they are very rare in modern PCs by now.

PCI and PCIe were ruled out because they are internal interfaces, but the logic exerciser is designed to be an external device. While there are methods to extend the PCIe bus to the outside of a PC casing, the possible length of the cabling is very limited. Also, the development of a PCIe peripheral is error-prone, as slight errors in the signaling can cause the operating system of the host PC to become unstable, constituting a very uncomfortable development environment.

USB may be a good choice because of its relatively simple structure and good data throughput. However, Ethernet fits the connectivity requirements of the Logic Exerciser better because it can be connected to a Local Area Network, thus enabling multiple hosts to access the Logic Exerciser without having to move the device or even change the cabling.

Ethernet itself is a low-level protocol situated at the lower levels of the ISO OSI model. For effective communication between hosts, at least two higher-level protocols are encapsulated within the Ethernet frames (Internet Protocol (IP) combined with Universal Datagram Protocol (UDP) or Transmission Control Protocol (TCP)). This requires considerable management effort and is typically done by software running on a general-purpose CPU, although there are exceptions[1].

[1] For example, a so-called TCP offload engine is available in some high-end network interface cards designed for high-speed connections where the management of the TCP/IP stack requires a considerable amount of processing power which would be better spent with other tasks.

The selection of Ethernet as the connection between the host and the coarse-grain reconfigurable logic therefore mandates the inclusion of a CPU. Typically, a system designed this way is called a System on Chip (SoC). Because the hardware selection is limited to FPGAs due to financial reasons (requirement 4), two options are available:

Soft-Core CPUs: The RTL code of the CPU is synthesized for the targeted logic device. Several FPGA vendors offer soft-core CPUs optimized for their devices (e.g. Xilinx MicroBlaze [59], Altera Nios II [31]). Also, multiple open-source soft-core processors are available (e.g. LEON [AGW10], OpenMSP430 [49] and OpenRISC [50]).

The main advantages of a soft-core CPU is the fact that the designer is able to choose the CPU which best fits the requirements and the ability to extend the CPU to include custom instructions or optimizations. The downside of this approach is that the logic resources of the FPGA are shared among the CPU and the user-designed logic circuit. Also, the clock frequency of the resulting design depends on the timing requirements of both parts of the design.

Hard-Core CPUs: This variant does not use the reconfigurable logic of the FPGA to implement the processor – instead, a dedicated and optimized cell comparable to an ASIC is included in the FPGA with a defined interface to the reconfigurable logic portion. The advantage of this variant is much better performance of the optimized CPU block without imposing additional timing constraints on the logic implemented in the FPGA. Also, the CPU does not need to be synthesized during the design of the logic circuit, speeding up the design flow.

Because the main use case for the Logic Exerciser is the analysis of generic digital circuits (requirement 1; comparable to a digital logic analyzer), it can be expected that data is generated at a high rate, the highest possible being one transmission per clock cycle. A soft-core CPU which is operating at the same clock speed as the reconfigurable logic circuit would have trouble keeping up the transmission of data to the host PC. For this reason, an FPGA with a powerful hard-core CPU is the preferred implementation basis for the Logic Exerciser. Most devices which contain CPUs of this type also offer dedicated communication logic e.g. for Ethernet or USB, which again is beneficial to keeping the reconfigurable portion of the FPGA reserved for the coarse-grain logic circuit.

3.2 Digital Circuit Structure

Now, with the requirements for the Logic Exerciser device having been established in the previous section, the structure of the digital circuit can be planned. First, we need to define a few terms that will be used throughout the remainder of this thesis:

- The **Reconfigurable Logic Module** is the part of the digital circuit which implements the coarse-grain reconfigurable logic.
- An **Application** is a task or a set of tasks which is implemented by the Reconfigurable Logic Module.

- A **Cell** is a functional unit within the Reconfigurable Logic Module. Examples include a reconfigurable FSM, a register, a memory or ALU. The behavior of a cell may be fixed (e.g. an ALU which implements a pre-defined set of operations) or the cell may be reconfigurable by itself (e.g. the behavior of an FSM is determined by its bitstream, the contents of a memory cell may be altered).
- The **Interconnect** implements the reconfigurable connections between the individual cells and between the cells and the top level ports of the Reconfigurable Logic Module.

In the previous section, we have already derived from the requirements of the Logic Exerciser that the system CPU is tasked with the management of the TCP/IP stack and the configuration of the coarse-grain reconfigurable logic circuit, requiring communication between those two components. There are two different ways to achieve this:

- A General Purpose Input/Output (GPIO) port may be used to implement a very basic communication scheme between the CPU and the reconfigurable logic. Each participant drives a signal vector of e.g. one or two bytes and a separate "*Enable*" or "*Valid*" signal which triggers an interrupt or is continuously polled. While this approach is easy to implement, it is not very efficient.
- Most modern embedded CPUs are equipped with an on-chip bus which is used to implement communication between the CPU core itself and various *peripherals* such as a USB controller, a memory interface, an Analog/Digital Converter or a timer. Interfaces to these on-chip buses are more difficult to design, but offer a much higher data throughput at a much lower processor utilization than the first approach involving a GPIO port.

Because of its advantages, the second approach implementing a bus interface will be used in the Logic Exerciser. In SoC FPGAs, the on-chip bus is typically exposed to the reconfigurable fabric in which the user-defined logic can be implemented. This allows us to attach the digital circuit for the Logic Exerciser as an independent slave peripheral to the CPU. As part of this peripheral, a GPIO port acting as an interface between the Logic Exerciser and the Device Under Test (DUT), needs to be included.

These considerations lead to the definition of four key components that constitute the Logic Exerciser peripheral, which are also shown in figure 3.1:

Reconfigurable Logic Module: This is the most important part of the design. It contains the cells which implement the coarse-grain reconfigurable logic circuit. The module has two different external interfaces: the GPIO port which is used to enable communication between the reconfigurable logic and external logic as well as a Configuration and Parameterization interface.

Configuration Interface: This module is responsible for the transfer of the configuration data as well as parameters received from the host PC to the Reconfigurable Logic Module.

Bus Interface: The bus interface is used to connect the Reconfigurable Logic Module and the Configuration Interface to the on-chip bus (which is dependent on the type of CPU used in the design). The bus interface makes registers implemented in the Logic Exerciser peripheral accessible to the system CPU.

GPIO Port: The digital GPIO port handles the configuration of the I/O cells that interface with the DUT with regard to direction (input or output) and value (driving logic '0' or '1').

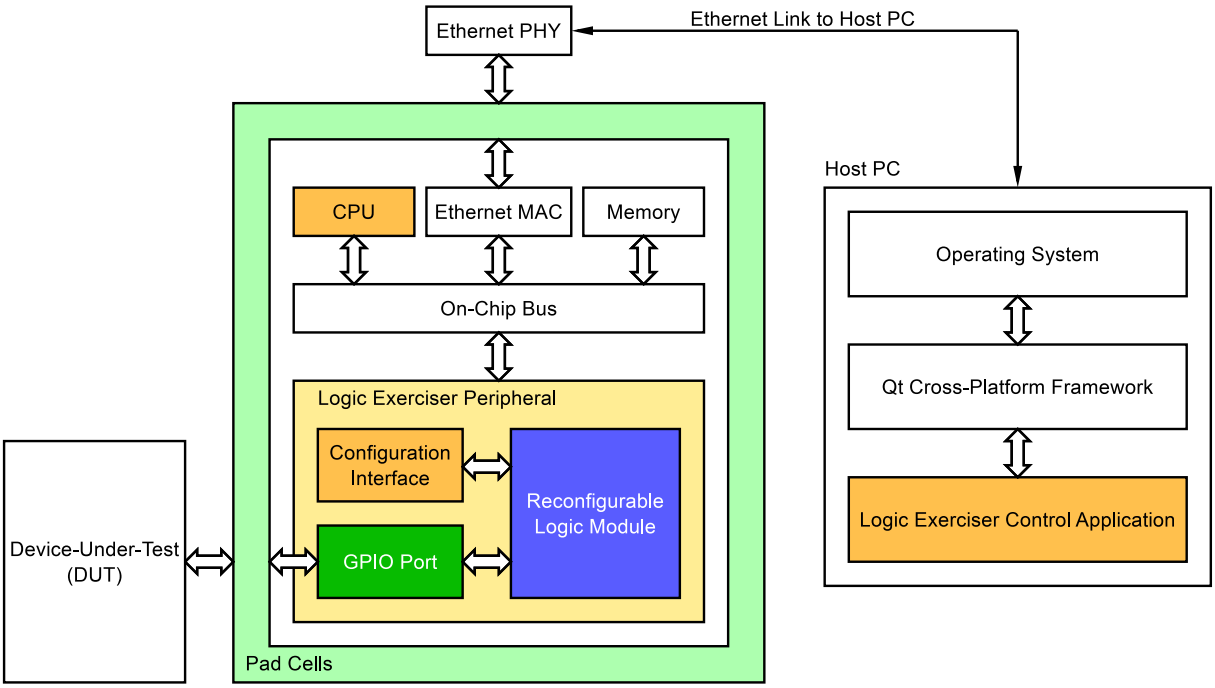


Figure 3.1: Logic Exerciser System Structure

Once the cells which comprise the Reconfigurable Logic Module have been defined, the module can be generated automatically by specifying multiple example application netlists and processing them with InterSynth [WGS+12], which was introduced in section 2.3.2.2. The remaining components need to be implemented manually using a Hardware Description Language.

3.2.1 Configuration and Parameterization

The behavior of a coarse-grain reconfigurable circuit can be altered in two ways: configuration and parameterization. There are fundamental differences between these two procedures, which need to be implemented in customized digital logic interfacing with the reconfigurable logic module:

Configuration: The configuration of the reconfigurable logic module alters its behavior. This is done by modifying the connections between the cells of the coarse-grain reconfigurable circuit and, where supported by the cells, their behavior (e.g. the behavior of a reconfigurable state machine or the contents of a memory can be altered directly). Similar to existing reconfigurable architectures, configuration needs to be done only once at system start-up, and the configuration is retained until the circuit is either powered down or needs to be reconfigured again to implement a different application.

Parameterization: Parameterization does not directly alter the behavior of the circuit. Instead, it provides values that are either needed for the correct operation of the circuit (e.g. "magic numbers") or user data which is processed by the application. Examples may include a

threshold value which is compared to measurement data retrieved from a sensor to check if an alarm should be raised.

Due to the different requirements of the configuration and the parameterization interface, two different approaches are used: The configuration of the reconfigurable logic module can be done in a serial manner by shifting a *configuration bitstream* into configuration shift registers. For the parameterization interface, a more flexible approach is needed: the system CPU needs the ability to quickly change parts of the parameterization data to provide new data to the coarse-grain reconfigurable logic circuit or turn on/off certain features. Therefore, the storage of parameterization data should be word-accessible by the CPU and a way to notify the reconfigurable logic module of changes to the parameterization data is suggested.

3.3 Design Flow

Elaborate methods exist for the design of digital circuits for Fine-Grain Reconfigurable Logic or ASICs, which usually include describing the design with the help of hardware description languages on the algorithmic or RTL levels, then synthesizing the HDL description. For Coarse-Grain Reconfigurable Logic, this is not yet possible – while RTL synthesis can be used during the design of the actual architecture, the development of applications which are implemented by the reconfigurable circuit requires manual effort. Because InterSynth – which is used to implement the Reconfigurable Logic Module – uses an example-driven approach [WGS+12], the design of the coarse-grain reconfigurable logic circuit and the design of the actual applications which can be implemented by it are closely interwoven.

The first and most important part of the design of a coarse-grain reconfigurable architecture is the identification of the arithmetic, logic and storage operations which need to be implemented by the cells in the Reconfigurable Logic Module. This process and the granularity as well as the computational power of these cells heavily depend on the requirements of the reconfigurable system, as a trade-off between power efficiency, speed and chip area for both the cells and the interconnect must be found.

Once the cells have been defined and implemented, a set of example applications should be created. As a starting point for application development, a purely structural Verilog or VHDL representation of each application can be created by manually instantiating the data path cells and FSMs. At this point, the FSMs which control the data path can be implemented in the Verilog language for the sake of simplicity. After the functionality of each application has been verified by established methods (e.g. simulation, logic equivalence checking, formal verification), the FSMs can be replaced by TR-FSMs. At this stage, the InterSynth netlists can also be prepared. This currently requires manual effort by "translating" the structural Verilog or VHDL representation to the InterSynth format (which is quite similar to a structural HDL description).

From the process described in this section, it can be seen that there is a circular dependency between the definition and implementation of the cells which form the coarse-grain reconfigurable logic, and the implementation of the example applications: it is difficult to develop the cells without prior knowledge about the applications which are to be implemented with the help of these cells, and it is impossible to develop the applications in the proposed structural manner when the cells are not specified sufficiently. Therefore, for the development of the first couple of

example applications, the designer can deviate from the purely structural approach mentioned earlier and include behavioral/RTL operations in the HDL design. This simplifies the definition of the cells and the example applications can be refined to a structural-only description later or the InterSynth netlists can be prepared directly.

3.3.1 Specification of Coarse-Grain Cells

The specification of the cells in the Reconfigurable Logic Module is the most important task a designer will face when developing a coarse-grain reconfigurable logic circuit. While the reconfigurable FSMs and the interconnect bear a large amount of inherent flexibility, the data path cells do not. Because the amount of necessary routing resources increases for each additional cell present in the reconfigurable circuit, it may seem desirable to implement very large and complex cells that implement a broad range of functionality. However, this decreases the utilization of the individual cells. Consider an application which wants to perform an addition and a multiplication at the same time: in an RTL design which is synthesized for a fine-grain reconfigurable device or an ASIC, these two operations would be synthesized as separate functional units. With an ALU which can only perform one operation per clock cycle, these operations would have to be done sequentially, and most of the resources in the ALU are not utilized in both clock cycles. Replicating the ALUs restores the ability to perform both operations in the same clock cycle, but doing this on a large scale lowers the logic utilization even more. On the other hand, very small granularity (e.g. each operation is implemented in its own "Mini-ALU") leads to larger routing overhead.

Besides choosing the correct granularity of the cells, the designer needs to decide which operations shall be implemented in the first place. For this process, we propose an empirical approach which analyzes the language features of common hardware description languages. For example, the Verilog HDL defines (amongst others) the following operations [Soc06]:

Arithmetic Operations: Arithmetic operations include addition, subtraction, multiplication, division, and modulus.

Bitwise Operations: A bitwise operation is executed by performing the operation on each pair of corresponding bits in the operands. This includes the following operations: NOT, AND, OR, XOR, XNOR.

Shift Operations: The bits of an operand are shifted by one to the left or right. These can be divided into arithmetic and logical shift operations.

Conditional Statements: This language feature facilitates behavior that depends on the result of a specific operation. Three different constructs are supported by Verilog: a ternary operator ('?'), "if-else", and "case".

The arithmetic and bitwise operations could be implemented in a single or several different ALU cells. For the implementation of the shift operations, a shift register may be suited best. Simple conditional statements that only test the value of single bits may be handled by FSMs. Finally, an additional element must be considered which is not listed above: the temporal domain. Many real-world applications depend on actions which either occur or need to be executed at a specific time. Again, FSMs can be used to wait for an event to occur (e.g. an input signal changes), but

the generation of an event after a pre-determined number of clock cycles needs to be implemented by a dedicated timer cell. A special case of the addition (+1) and subtraction (−1) operations can be handled by a combined timer/counter cell.

3.3.2 Synthesizing an FSM into a TR-FSM

RTL Synthesis tools usually include support for FSM Detection and Extraction, which detects finite state machines described in the HDL code and transforms them into an internal representation (e.g. state transition tables) [Gio95]. Then, various optimizations can be performed and finally, combinational logic is generated to implement the FSM's transfer and output functions [GDHG10]. When synthesizing the HDL description of a finite state machine for implementation in a TR-FSM, the last step has to be implemented differently – instead of generating combinational logic, the state transitions need to be mapped onto the TR-FSM's transition rows. This mapping process is already implemented in the trfsmgen tool introduced earlier in section 2.3.1.3, which is able to read FSM definitions from a text file in the KISS format [Uni92]. This format describes the transitions of a finite state machine in the following table-like format:

```
.start_kiss
.i 3
.o 3
.p 5
.s 4
.r state0
0-- state0 state0 010
1-- state0 state1 100
01- state1 state2 001
--1 state2 state3 111
1-- state3 state0 000
.end_kiss
```

This KISS file describes an FSM with three input state ("`.i 3`"), three output states ("`.o 3`"), five state transitions ("`.p 5`"), four states ("`.s 4`") with the reset state being "`state0`". The remaining lines describe the state transitions: the first column represents the input signal pattern, which may include the don't care symbol ("`-`") that triggers the transition. The second column marks the state in which the transition may occur, followed by the destination state in the third column. The fourth and last column contains the output signal values for this transition.

While the FSM behavior could be described by manually writing a KISS file or by utilizing the Tcl interface of trfsmgen, the ability to describe them in common HDLs is highly desirable as this method provides the designer with much better abstraction and the advantage of being able to simulate the HDL description of the FSM during design, long before generating the TR-FSM bitstream. Therefore, the so-called `FSM_EXPORT` pass was developed for Yosys, which has been introduced in section 2.3.3. Yosys already has support for FSM detection, extraction, optimization and recoding [WG13]. The `FSM_EXPORT` pass is typically run after the FSM optimization passes and generates a KISS file for each FSM in the design. It is possible to set the Verilog attribute "`fsm_export`" on the state register of the FSMs to define the name of each exported KISS file or mark individual FSMs for export directly in the source code.

Now, together with the specifications of the targeted TR-FSM (input and output width, state register size and number of available transition rows), the resulting KISS file can be used as an input for trfsmgen to generate the bitstream for the TR-FSM instance.

3.3.3 Generation of the Reconfigurable Logic Module

As mentioned earlier in section 3.3, the generation of the Reconfigurable Logic Module is done with the help of InterSynth. Being an example-driven tool [WGS+12], multiple netlists of example applications that shall be implemented by the reconfigurable logic module need to be specified.

3.3.3.1 Extensions to InterSynth

Before InterSynth could be used in a project of this scale, several features had to be added. Many of the Verilog modules described in the configuration files have a large number of ports, but InterSynth initially required each instantiation to be defined on a single line of text within the configuration file. Also, certain cells can have their own bitstream, which is integrated with the InterSynth bitstream. This bitstream also had to be specified in the netlist configuration file on the same line as the instantiation as either a decimal, octal, hexadecimal or binary number. These restrictions would have resulted in configuration files with line lengths of more than 1000 characters, making them highly impractical to create or edit manually.

As a result of these shortcomings, three features were added to the InterSynth code:

Line Continuation: This feature allows to split the commands in InterSynth configuration files over multiple lines of text. By adding the continuation character '\' at the end of a line, the configuration file parser is instructed to add the following line to the current command.

Removal of Line Length Limit: Initially, InterSynth statically allocated memory for 1024 characters when reading each line of the configuration files. This limit was removed by dynamically allocating the exact amount of memory needed for the actual lines, and releasing it after it is no longer used.

External Bitstream Files: This feature allows to specify an external file which contains the bitstream of an instantiated module. This eliminates the need to include the bitstream in the netlist configuration file.

The following code example shows the instantiation of components prior and after the implementation of these features:

```
node test_inst test data_i sig1 data_o sig2 config_i 'b010001010111111101001101

node test_inst test \
  data_i    sig1    \
  data_o    sig2    \
  config_i  file:cam_bitstream.txt
```

3.3.3.2 InterSynth Configuration Files

The structure of the reconfigurable logic module (as generated by InterSynth) is defined by one or more configuration files, which contain the following information [54]:

Connection Types: Connection types are distinguished between their signal width and semantics (e.g. control signal, data signal, clock enable, ...). Each cell port is assigned a connection type, and will therefore be connectable to other ports of the same connection type. A separate interconnect is generated for each connection type.

Cell Types: A cell type lists the I/O ports of each cell and assigns the connection type and direction (input or output) to each port. Some of the ports have special semantics: they are *configuration ports*. These ports are not routed via the generated interconnect, instead they assigned their own portion of the Reconfigurable Logic Module's bitstream.

I/O Ports: The top-level I/O ports of the Reconfigurable Logic Module are also listed in the configuration files. These can either be connected directly to the cells within the generated module or connected to the interconnect.

Netlists: For each example application that is to be processed by InterSynth, a netlist must be defined, which contains instantiations of the cell types defined earlier. Each port of the cells is connected to a signal, much like the instantiations in common Hardware Description Languages.

It makes sense to separate the netlists from the remaining information ("base configuration") for clarity. The netlists can be easily ported from the structural HDL description which was developed earlier for verification purposes. At this stage, it is also necessary to generate the bitstreams for any components in the reconfigurable logic module that have their own configuration bitstream which will be included in the "global" bitstream. This does not count for the TR-FSMs, which have their own configuration interface that needs to be included on the top level of the reconfigurable logic module.

Once the configuration files are complete, InterSynth is started, the configuration is loaded and a series of commands is executed:

1. The "**presilicon**" command generates and optimizes the reconfigurable logic module, but does not yet write any output data.
2. The "**write_data**" command stores the internal state of the program. This can later be used to generate bitstreams for *new* applications with the help of the post-silicon design flow.
3. The "**write_verilog**" command writes the Verilog RTL description of the reconfigurable logic module to a file.
4. The "**write_bitdata**" command exports the bitstream for each application to a text file.

After these steps have been executed successfully, the Reconfigurable Logic Module can be included in the following design flow, which depends on the targeted ASIC (or FPGA when developing a prototype design) architecture.

3.3.4 Post-Synthesis Design Flow

After the production of the ASIC containing the Reconfigurable Logic Module, it is often desired to implement new applications. The ability to do so is limited by two criteria:

- A sufficient number of cells for the new application must be present in the Reconfigurable Logic Module. This is not specific to coarse-grain reconfigurable logic circuits, the same requirement exists for every application implemented in any reconfigurable device.
- The routing degree of freedom must be adequate for the new application. In a fully multiplexed interconnect, this is always the case, but this would induce a very large overhead in silicon area. The tree-structured interconnect generated by InterSynth offers efficient resource utilization while still providing a good routing degree of freedom [WGS+12].

The process of designing an application after the coarse-grain reconfigurable logic circuit has been manufactured is simpler than the pre-silicon design flow described in the sections above because the circular dependency between cell and application development does not apply – the specification and implementation of the cells has already been completed. If this was done well, the designer merely has to define the behavior of the reconfigurable FSMs, generate the cell bitstreams and implement the netlists which are used as inputs to InterSynth.

3.4 Example Applications

In section 3.3, the methods for the implementation of the reconfigurable logic module were outlined and the example-driven approach by InterSynth was described. Therefore, before the implementation of the coarse-grain reconfigurable logic circuit can begin, it is necessary to define several example applications which will later be used by InterSynth during the pre-silicon design flow.

The definition of several example applications is also beneficial for the specification of the coarse-grain cells used to implement the reconfigurable logic module. While the empirical approach described in section 3.3.1 already provides valuable information about the types of operations which need to be implemented by the cells, the actual implementation of real-world applications yields even more knowledge about the required features of each cell.

3.4.1 SPI Sniffer

The reconfigurable logic module, when configured as an Serial Peripheral Interface (SPI) Sniffer, shall act as a device which listens to the communication between two SPI devices. The primary purpose is to extract the raw data exchanged between the communicating devices for debugging purposes, but also for further analysis by another application implemented by the reconfigurable logic module.

The Serial Peripheral Interface (SPI) is a synchronous master/slave serial bus interface developed and specified by Motorola [46]. It is integrated in many microcontrollers and digital sensors. The physical layer defines four logic signals:

- SCK (Serial Clock)

- SS (Slave Select)
- MOSI (Master Output, Slave Input)
- MISO (Master Input, Slave Output)

The master initiates communication by asserting a logic '0' to the Slave Select line. Then, it generates clock cycles on the SCK line. During each SCK cycle, a full-duplex communication takes place: the master transmits a data bit on the MOSI line, and the slave transmits a data bit on the MISO line. After the desired number of bits has been transferred, the master ends the transmission by asserting a logic '1' to the Slave Select line.

The SPI standard specifies two parameters: clock phase (CPHA) and clock polarity (CPOL), which affect the sampling point of the MOSI and MISO signals with regard to the edges of the SCK signal [46].

Figure 3.2 shows an SPI transmission. In this application, the reconfigurable logic module must perform three basic tasks:

- Check whether an SPI transmission is currently active by evaluating the SS signal.
- Detect the positive edges of the SCK signal and sample the values of the MOSI and MISO lines.
- Keep track of the number of bits transmitted and store the data in a suitable format (e.g. byte-oriented).

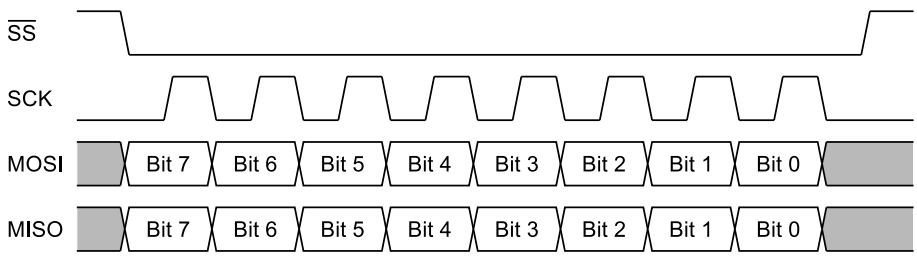


Figure 3.2: SPI transmission; CPHA = 0, CPOL = 0

3.4.2 Protocol Analyzer for ADT7310 Sensor

The purpose of this application is to demonstrate the possibility of a hierarchical design. On top of the existing SPI sniffer application, the protocol analyzer application is responsible for the monitoring of an ongoing SPI communication between two devices and the extraction of information relevant to a specific function. The communication between a microcontroller which periodically polls an ADT7310 device (a smart digital temperature sensor designed by Analog Devices Inc.) via the SPI interface is used as data source for the protocol analyzer application.

The ADT7310 is equipped with an SPI-compatible interface which exposes several of the sensor's internal registers. The sensor can be configured and polled via this interface by reading and/or writing to these registers. For example, the retrieval of the temperature value is done in two steps [32]:

1. The SS line is asserted by the SPI master and a command byte is sent to the ADT7310 sensor. It contains the register address for the following transaction and a read/write bit.

2. After the command byte has been sent, the actual transfer of the information is done. Depending on the command byte transferred earlier, either an eight or sixteen bit long SPI transfer is executed. For write transfers, the ADT7310 sensor will store the data received on the MOSI line in the register selected by the register address, and for read transfers it will provide the contents of the register on the MISO line. At the end of this cycle, the SS line is de-asserted again.

Reading the temperature value from the sensor therefore requires the transmission of a command byte containing the address of the temperature value register along with the read/write bit set to indicate a read access. A typical communication between an SPI master and the ADT7310 sensor might continuously read the sensor's Temperature Value Register in predefined intervals and act on the received information (e.g. raise an alarm if the temperature is not within certain bounds).

This application extends the SPI Sniffer introduced in the previous section by processing the raw MOSI and MISO bytes that were extracted from the communicating devices. This data shall then be analyzed and the temperature values sent to the host PC. The implementation of this application requires the ability to synchronize different parts of the reconfigurable logic module which are governed by different FSMs: the SPI Sniffer application shall be designed to store the raw SPI data for transmission to the host PC, implying that a buffering mechanism is needed. However, the data is also required by the cells which implement the Protocol Analyzer, therefore neither part of the design may execute a consuming read operation without interfering with the other part. This requirement forms the need for the development of a flexible buffering mechanism that allows different cells to access the same data in an independent way.

3.4.3 SPI Master

The previous example applications focused on the reception and analysis of digital data. However, the Logic Exerciser shall also be able to generate digital signals. Therefore, in this application, the focus lies on the implementation of a simple SPI master which actively polls a digital sensor with an SPI interface.

The MAX6682 smart temperature sensor designed by Maxim Integrated Products is used as DUT for this example application. This sensor converts an external thermistor's temperature-dependent resistance into an 11-bit signed value which can be read via an SPI-compatible interface [42]. This interface is read-only; it is not necessary to send register addresses or commands to the sensor, instead the master simply generates 11 SCK cycles after asserting the Chip Select signal. The sensor then replies with the latest temperature value on the MISO signal [42].

3.4.4 I^2C Master

The complexity of the example applications introduced in the previous sections is quite limited. However, because the main goal for the development of the coarse-grain reconfigurable logic circuit is universality (requirement 1 on page 20), the following example applications will be more complex. A good candidate for a more challenging application is an Inter-Integrated Circuit (I^2C) master.

The I^{2}C Master application shall implement the ability to generate I^{2}C frames of arbitrary length. It shall include a simple interface for data exchange and the generation of the start and stop conditions, which can be used by another example application.

The I^{2}C bus is a simple serial, master-slave oriented, bidirectional 2-wire bus designed by Philips Semiconductors (now NXP Semiconductors) implemented in over 1000 different integrated circuits manufactured by more than 50 companies. It is a byte-oriented bus requiring only two signals called SDA (Serial DAta line) and SCL (Serial Clock Line). These signals are connected to a positive supply voltage via a pull-up resistor or a current source [47].

A level change on the SDA line is only permitted while the SCL line is low. Two exceptions from this rule form the so-called *start condition* (a negative edge on the SDA line while SCL is high) and the *stop condition* (a positive edge on the SDA line while SCL is high). These conditions are always generated by the bus master. Between the start and stop conditions, one or more bytes may be transferred. After eight bits, the sender and receiver switch roles and an acknowledgment (ACK) bit is transferred. Once the master has generated a start condition, it sends the slave address and the Read/Write bit. If a slave with the transmitted address is present, it transmits an ACK bit. Depending on the transfer type (read or write), the master or the slave will then transmit one or more data bytes [47].

Figure 3.3 shows an I^{2}C transmission.

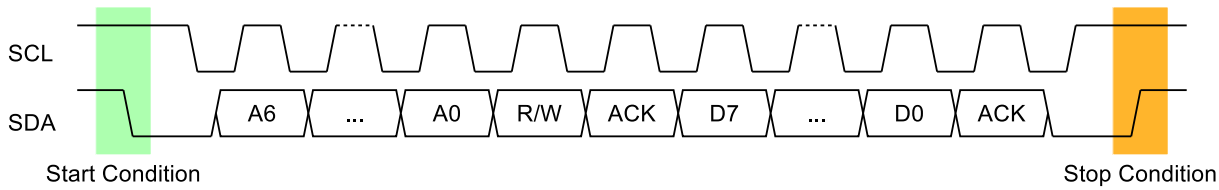


Figure 3.3: I^{2}C transmission

3.4.5 Protocol Exerciser for ADT7410 Sensor

In this example application, the I^{2}C Master introduced in the previous section is extended by a controller which periodically retrieves measured values from an ADT7410 digital temperature sensor attached to the Logic Exerciser device.

The ADT7410 is a smart temperature sensor designed by Analog Devices, Inc. [33]. It bears similarities to the ADT7310 device, which has already been described briefly in section 3.4.2. Instead of an SPI interface, the ADT7410 sensor features an I^{2}C interface to access its internal registers. This example application extends the I^{2}C Master Application to generate I^{2}C transactions which comply with the communication protocol of the ADT7410 sensor.

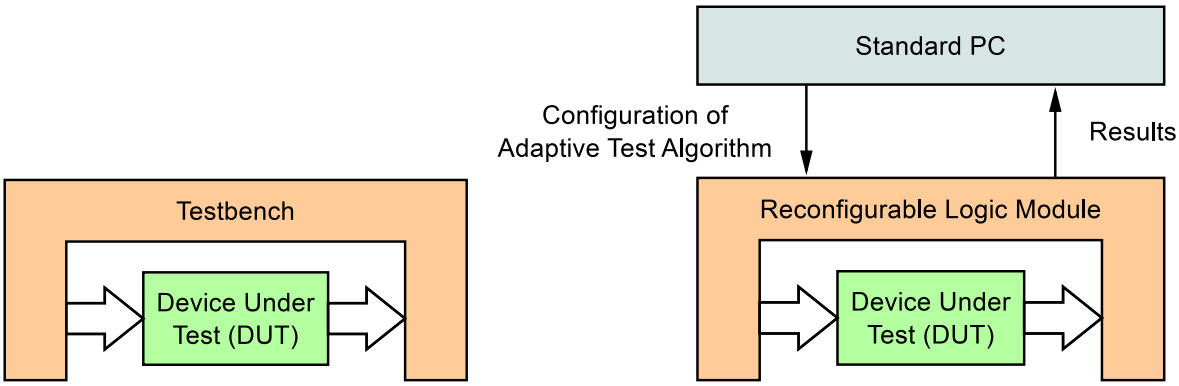
Instead of the full-duplex communication implemented by SPI, where a separate data line for both communicating devices is available, the I^{2}C interface only supports half-duplex communication, i.e. while an SPI device can receive and transmit data concurrently, an I^{2}C bus can only be driven by one device. Therefore, the sequence of data transmissions when polling the ADT7410 sensor differs significantly from the ADT7310:

- After the start condition has been generated by the master, an address byte containing the 7-bit I^2C address of the ADT7410 sensor and the read/write bit is sent. Then, the internal address (one byte) of the destination register within the sensor is transmitted.
- For write transfers, up to two data bytes can be sent immediately and the stop condition is generated by the master.
- For read transfers, this is different: in the current transfer, the master is sending data to the slave. To reverse these roles, a stop condition has to be generated, immediately followed by a start condition and another address byte. In the following cycles, the ADT7410 sensor generates the data bits.
- The final stop condition generated by the master terminates the transfer.

The goal for the implementation of this application is to demonstrate the ability of the reconfigurable logic module to implement a hierarchical design. In this case, all of the cells used by the I^2C Master are controlled by the cells of the Protocol Exerciser application, further increasing the complexity of the

3.4.6 Hardware Testbench

The previous example applications mostly dealt with serial bus interfaces such as SPI or I^2C or the generation and analysis of packets/data related to specific communication protocols. However, one of the requirements of the Logic Exerciser device is the ability to act as arbitrary digital logic by reconfiguring its reconfigurable logic module (requirement 1 on page 20). To demonstrate the capability of implementing generic circuits, this application, which is designed to act as a hardware implementation of a testbench for an existing digital logic module, was specified.



(a) Testbench Concept (b) Hardware Testbench (Logic Exerciser)

Figure 3.4: Testbench with Device Under Test

Once a module has been implemented with the help of a hardware description language, it is often simulated with specialized software to verify its functionality. However, most modules require *test stimuli* for correct operation, e.g. when simulating an ALU, a clock signal, the operands and an operation might need to be applied to the module. Also, the outputs of the module need to be compared to values that are known to be correct. These two tasks are typically handled by a

testbench, which is often also written for individual modules, parts of the design structure or the entire design in a hardware description language. The concept is shown in figure 3.4(a). Figure 3.4(a) shows the concept of a testbench implemented in a hardware description language, which applies test vectors to the Device Under Test (DUT) and checks the results generated by the DUT against known good values. However, in certain cases, there are simulation/synthesis mismatches, resulting in a different behavior of the implemented design compared to the simulation. Also, accurately simulating a design at later stages in the design flow (e.g. after RTL synthesis or after Placement and Routing has been completed), the runtime of the simulation of a complex module increases drastically. Therefore, it is desirable to be able to implement the testbench in hardware, so it can be run in real-time to verify the DUT. To implement a hardware testbench, the reconfigurable logic module must implement three tasks:

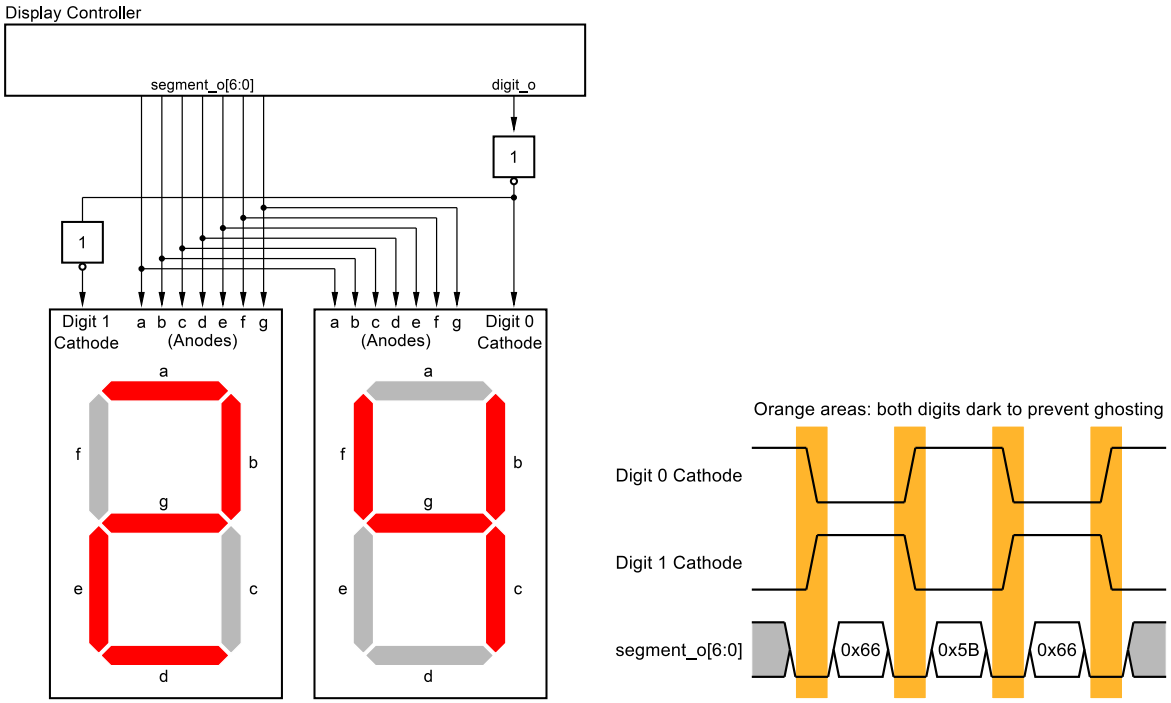
- An adaptive test algorithm, which applies inputs to the DUT and then dynamically reacts to its outputs.
- Reception of input data from the host PC.
- Logging or transmission of test results to the host PC for further analysis.

Figure 3.4(b) illustrates this concept.

As DUT for the hardware testbench application, a small microcontroller written in the VHDL language was chosen. This microcontroller implements a set-point adjuster for an automotive cruise controller - the desired set-point is applied via an 8-bit input vector, the microcontroller then increments or decrements the current set-point at a predefined rate, causing the vehicle equipped with the cruise control to accelerate or decelerate at a comfortable rate. The DUT is implemented in hardware (in this case, the same FPGA which also contains the Logic Exerciser is used, but it may also be implemented in any other device) and the Logic Exerciser is used to run tests comparable to a hardware testbench written in a hardware description language and executed by a simulator.

The current set-point is output from the microcontroller via a multiplexed two-digit seven segment display. Each digit consists of seven LEDs arranged to form a *digit*. Each Light Emitting Diode (LED) can be turned on and off independently to display numbers from 0 to 9 and some letters, including 'A' to 'F', therefore the seven segment display can be used to display hexadecimal numbers. To employ more than one seven segment display, one could simply connect each LED to a GPIO output pin on the display controller. This, however, would require $n \cdot m$ pins for n digits containing m segments each. A more resource-friendly approach is to *multiplex* the display. In this approach, the LEDs on all digits are driven by the same signal, and the digits are turned on and off repeatedly just in time to display the correct arrangement of LEDs for each specific digit. This approach uses only $(n + m)$ pins for n digits containing m segments each. Due to an effect called *persistence of vision*, the human eye can not see that they are in fact turned on and off at a high rate [BGG03].

A phenomenon that can occur in multiplexed segment displays is *ghosting* or *shadowing* – a faint number is shown in parallel to the actual number displayed. This can happen when the wrong segment value is applied to an enabled digit for a short amount of time. To prevent this, the display controller needs to turn off **all** digits for a brief period before switching to the next digit.



(a) Two Digit Seven Segment Display

(b) Multiplexed Control Signals

Figure 3.5: Seven Segment Display

Figure 3.5(a) shows the signals generated by the DUT. Because the hardware used in this example uses a single output pin to drive both cathodes via two inverters, the segment output is used to turn off the active digit when changing to the next digit.

On the right digit (digit 0), segments `b`, `c`, `f`, and `g` are illuminated. Converted to binary format – with segment `a` represented by the least-significant bit and segment `g` represented by the most-significant bit, this translates to the hexadecimal number of `0x66`. On the left digit (digit 1), segments `a`, `b`, `d`, `e`, and `g` are illuminated. In binary notation, this translates to the hexadecimal number of `0x5B`.

The goal for the hardware testbench application is to apply a test vector to the microcontroller (the desired set-point), then check the output on the seven segment display. The values displayed on the seven segment display shall then be transferred to the host for further analysis.

This means that the reconfigurable logic module must implement *de-multiplexing* of the seven segment display to regenerate the current set-point.

4 Implementation of the Logic Exerciser

After the requirements of a product have been specified and the system structure has been defined, the actual implementation process can begin. At the beginning, a target hardware platform that meets both functional and non-functional requirements needs to be selected. Then, based on the selected hardware platform, the implementation of the individual modules is planned and executed. In accordance with these established design principles, in this chapter we will first select a suitable hardware platform. Then, we will refine the structure of the digital circuit which was introduced in section 3.2 and provide implementation details for its components.

4.1 Hardware Platform

This section describes the hardware platform used as the basis of the Logic Exerciser.

Although the coarse-grain reconfigurable logic module is exclusively targeted at ASICs, FPGAs were used during the implementation. This decision was made because the development of an ASIC is much more expensive than the development using an FPGA. The reconfigurability of the FPGA is **not** used to implement reconfigurability of the implemented digital circuit, e.g. the circuit is reconfigurable without modifying the FPGA's own configuration bitstream.

Development started with the Microsemi SmartFusion Evaluation Kit, but during the implementation process, it became clear that while the processing power of the hard-macro CPU included in the SmartFusion device was sufficient, the FPGA resources were too limited. This resulted in a switch to the Xilinx ZedBoard development kit. Both platforms are introduced briefly in this section.

4.1.1 Microsemi SmartFusion

The SmartFusion SoC FPGA is manufactured by the Microsemi Corporation. It contains a Microcontroller Subsystem (MSS) based on an ARM Cortex-M3 CPU core, programmable analog blocks and an FPGA fabric based on the Microsemi ProASIC3 architecture [45].

The microcontroller subsystem includes an AMBA AHB bus matrix, which is accessible from the FPGA fabric, thus enabling the development of custom microcontroller peripherals attached to the CPU's AHB bus.

The MSS also includes an Ethernet Media Access Controller (MAC) peripheral [45].

To immediately start development, the commercially available low-cost SmartFusion Evaluation Kit [44] was chosen. It provides all necessary hardware features required to implement the logic exerciser:

- The SmartFusion SoC device,
- An RJ45 jack connected to a 10/100 MBit/s PHYsical layer (PHY) chip interfacing with the SmartFusion's Ethernet MAC,
- Sufficient digital I/O pins routed to a 100 pin mixed-signal header.

However, the mixed signal header on the SmartFusion Evaluation Kit is not very well suited for the connection of hardware that interfaces with the logic exerciser. Therefore, the breakout board shown in figure 4.1 was developed. The design resources for this board are available under an open source license [37].

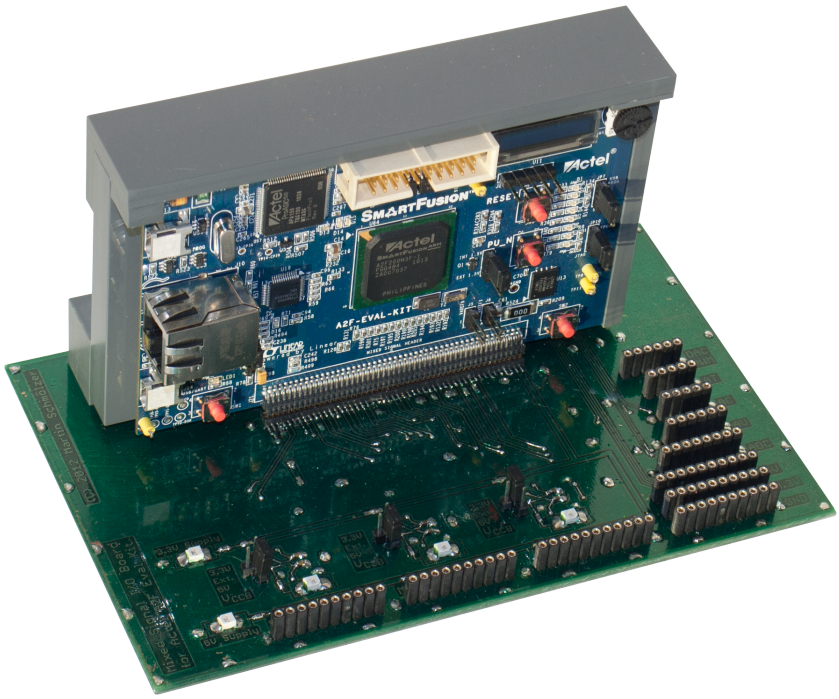


Figure 4.1: SmartFusion Breakout Board with Evaluation Kit installed

The breakout board features a 100 pin female connector that matches the 100 pin mixed signal header on the SmartFusion Evaluation Kit, which is vertically inserted into the breakout board's connector. A retaining bracket (gray plastic) was developed to hold the SmartFusion Evaluation Kit in place and keep the connectors on both logic boards free from mechanical stress.

The analog and digital I/O signals provided by the 100 pin mixed signal header are connected to several user connectors. Because the SmartFusion device offers I/O standards of up to 3.3 Volts, level shifters were included on the breakout board for each of the digital I/O pins. This allows the system to interface with devices that operate with I/O standards of up to 5.5 Volts [53].

The analog I/O pins of the 100 pin mixed-signal header – although not used by the logic exerciser – were also routed to User I/O connectors for use in other projects employing the SmartFusion Evaluation Kit.

4.1.2 Xilinx Zynq-7000

The Zynq-7000 is a high-performance SoC FPGA designed and manufactured by Xilinx, Inc. It consists of the Processing System (PS) and the Programmable Logic (PL) components.

The Processing System consists a dual-core ARM Cortex-A9 Application Processor Unit, on-chip caches and memory, multiple external memory interfaces and several I/O peripherals and interfaces (Gigabit Ethernet MAC, USB, Controller Area Network (CAN), Universal Asynchronous Receiver Transmitter (UART), SPI, I^2C amongst others) [57].

The Programmable Logic is based on the Xilinx Artix-7 FPGA technology [57]. The Zynq device allows to connect logic implemented in the PL to the PS via several AMBA AXI interfaces.

Again, a commercially available development kit, the *ZedBoard*, shown in figure 4.2 was chosen.

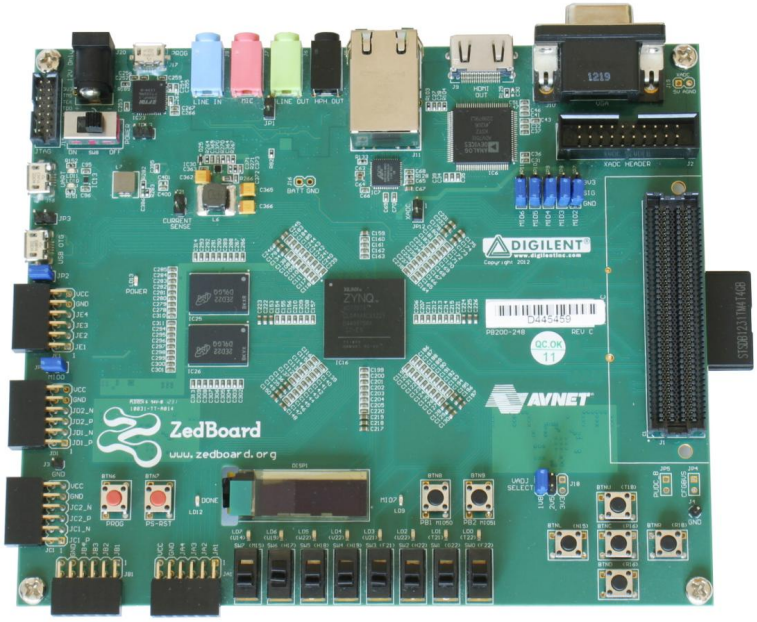


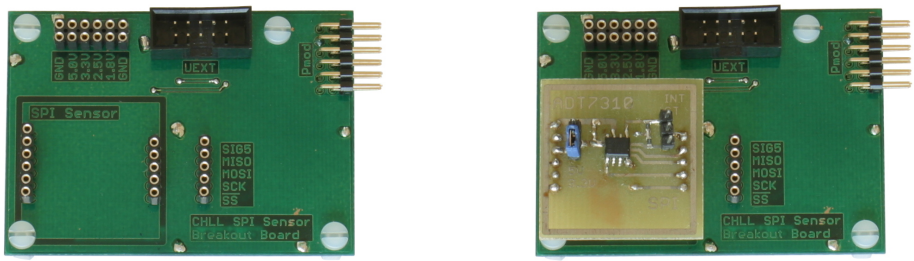
Figure 4.2: ZedBoard Development Board

The ZedBoard exposes the Zynq's digital I/O pins via two separate connector types:

- The Low Pin Count FPGA Mezzanine Connector (LPC-FMC) on the right-hand side of the ZedBoard exposes 68 I/O pins [35].
- The five **Digilent Pmod Compatible Headers** along the edge in the left-hand bottom corner of the ZedBoard expose a total of 40 I/O pins, of which 32 are connected the Zynq PL and 8 are connected to the PS [35].

Because the LPC-FMC is difficult to solder by hand, the much simpler Pmod headers were selected as the interface between the ZedBoard and the DUT. For this purpose, a set of small Breakout Boards was developed. These breakout boards comprise the interface between external digital circuits and the ZedBoard. In the course of the Configurable High-Level Logic (CHLL) project, various modules containing a variety of digital sensors had already been developed and manufactured at the Institute of Computer Technology [GGHG11]. To connect these sensor modules to the ZedBoard, and ultimately interface them with the logic exerciser, three different adapter boards were designed:

- The **I²C** and **SPI Sensor Breakout Boards** are shown in figures 4.3 and 4.4, respectively. These boards contain a standardized connector that matches the CHLL sensor modules. Furthermore, these boards feature a Digilent Pmod compatible pin header and an Olimex UEXT [30] connector, which can be used to connect the board to many Olimex microcontroller evaluation kits such as the STM32-P103, as well as an additional connector that can either be used to directly connect wires or an oscilloscope/logic analyzer to the board.
- The **Pmod I/O Extension Board** is shown in figure 4.5. This board was developed to interface with devices that employ other logic levels than that supported by the Zynq device. Like in the SmartFusion breakout board described in section 4.1.1, level shifters are used to translate the 3.3 V I/O standard on the Pmod compatible headers to other I/O voltages[1].



(a) Without Sensor

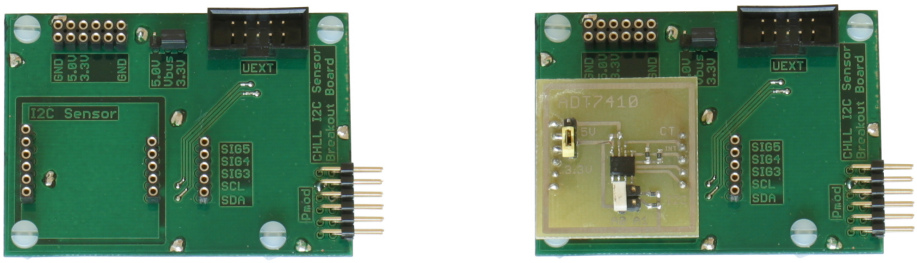
(b) ADT7310 Sensor Installed

Figure 4.3: CHLL SPI Sensor Breakout Board

4.2 Digital Circuit

In chapter 3, we discussed the requirements of the Logic Exerciser device and planned the implementation accordingly. With the hardware platform being selected in the previous section, we can now delve into the implementation details of both aspects of the digital circuit: the coarse-grain reconfigurable logic module and the supporting circuitry.

[1] Texas Instruments TXB0108 voltage-level translators are used to achieve this. They can convert digital signals to a voltage level between 1.65 V to 5.5 V [53].



(a) Without Sensor

(b) ADT7410 Sensor Installed

Figure 4.4: CHLL I^{2}C Sensor Breakout Board

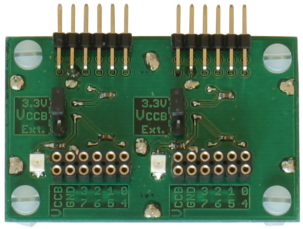


Figure 4.5: Pmod I/O Extension Board

4.2.1 Bus Interface

The bus interface was introduced in section 3.2. It implements the glue logic between the Advanced Microcontroller Bus Architecture (AMBA) found in the SoC FPGA and the reconfigurable logic module as well as the configuration interface. As already mentioned in section 4.1, the hardware platform had to be changed from the Microsemi SmartFusion to the Xilinx Zynq-7000 device during the implementation process. Therefore, only a description of the AMBA AXI bus interface present in the Xilinx device is given.

This device includes several AMBA AXI ports to allow circuits implemented in the PL section to communicate with the PS section of the device. The AXI protocol is the latest generation of AMBA interfaces targeted at high-performance, high-frequency designs [34]. The Zynq-7000 devices include several 32-bit AMBA eXtensible Interface (AXI) 3.0 interfaces [58]. The "full" AXI interface provides advanced features such as burst transfers and transaction ordering [34], which are not required in the logic exerciser implementation. Instead, the bus interface implements the AXI4-Lite version of the interface, and a *Protocol Converter* provided by Xilinx as an intellectual property block is used to translate AXI3 transfers to AXI4-Lite.

The AXI specification defines five independent transaction channels, each consisting of a set of handshake signals as well as information signals [34]:

- Read Address Channel,
- Read Data Channel,
- Write Address Channel,
- Write Data Channel,

- Write Response Channel.

The handshake mechanism of each channel consists of a signal pair:

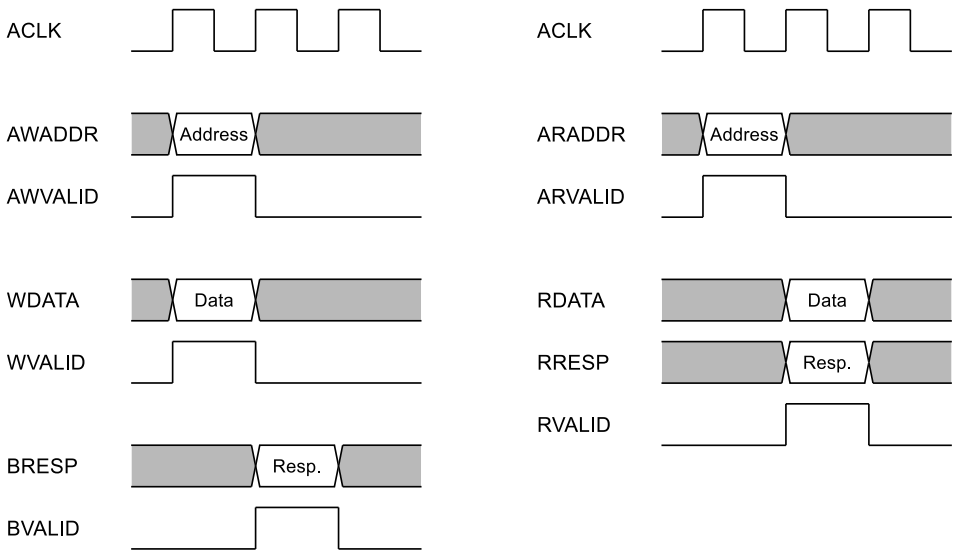
`READY`: This signal is driven by the channel receiver. If this signal is driven high, the receiver is able to process new information, e.g. an address or data.

`VALID`: This signal is driven by the channel sender. It is driven high when the data or address transmitted on the appropriate signal is valid, i.e. it should be processed by the receiver.

Together, the handshake signal pair can be used by the sender as well as the receiver of information to implement *flow control*, i.e. to influence the rate at which information is sent over the channel.

A read access consists of two transactions: first, address and control information is transferred on the Read Address Channel by the master. The slave then transmits data on the Read Data Channel [34].

A write access consists of three transactions: address and control information as well as data is transferred on the appropriate channels. This may happen during the same clock cycle or during different clock cycles. The slave might have to buffer address or data until both have been transmitted by the master. After the write access has completed, a write response is sent on the Write Response Channel [34]. Simplified examples of AXI write and read accesses are shown in figure 4.6.



(a) AXI Write Access (b) AXI Read Access

Figure 4.6: AXI Write Access

During development of the Bus Interface, the transactions initiated by the AXI master (the system CPU) need to be simulated to verify correct behavior of the AXI slave implemented by the Logic Exerciser's Bus Interface. This can be done by simulating the RTL code of the entire AXI master and at least parts of the system CPU. However, this would drastically increase simulation times because both the AXI master and the CPU are very complex logic circuits. Therefore, the device

vendors usually provide simulation models, which are executed by the simulator and generate the same signals as the actual hardware. In the case of on-chip buses, these simulation models are called "Bus Functional Model". However, Xilinx does not offer it free of charge, instead, a paid license which costs about USD 800,00 has to be purchased. To avoid having to license this module form Xilinx, a simulation model implementing an AXI4-Lite master was developed instead. This simulation model is – like most of the HDL code in the project – written in the Verilog Hardware Description Language and makes heavy use of non-synthesizable code constructs. It is instantiated in the testbench for the HDL design and emulates the bus accesses generated by the firmware program running on the Zynq-7000 Processing System.

4.2.2 Configuration Interface

The Configuration Interface, which was introduced in section 3.2.1, implements procedures to shift the configuration bitstreams of the Reconfigurable Logic Module into the appropriate configuration shift registers.

The individual configuration bitstreams can be quite large (up to 2000 bits), but the on-chip bus only allows to write up to 32 bits at once, which means that the bitstream has to be shifted into the configuration registers in chunks of up to 32 bits. The Configuration Interface provides two registers for this task:

Configuration and Status Register (CSR)

Writing to this register sets up a new configuration process. Table 4.1 describes the meaning of each bit within the register:

Bit(s)	Size	Description
31	1	Busy Bit
30	1	Configuration Mode for Interconnect
29	1	Configuration Mode for TR-FSMs
28	1	Reserved (always reads zero, write has no effect
27:24	4	Configuration Destination (0 = Interconnect, 1+ = TR-FSMs)
23:00	24	Bitstream Size: $2^{24} - 1 = 16.777.215$ Bits possible

Table 4.1: Description of the Configuration and Status Register (CSR)

Configuration Data Output Register (CDOR)

Writing to this register triggers the shift operation of a configuration bitstream chunk. While the shift operation is in progress, the Busy Bit in the CSR will read back a logic '`1`'. The content of the CDOR is shown in table 4.2.

Bit(s)	Size	Description
31:00	32	Configuration Data Chunk

Table 4.2: Description of the Configuration Data Output Register (CDOR)

The configuration of a component can be described by the following algorithm:

1. Split bitstream into chunks of 32 bits (the last chunk may be smaller),
2. Write the configuration destination, bitstream size and enabled configuration mode bit to the appropriate fields within the CSR,
3. Write a bitstream chunk to the CDOR,
4. Repeatedly read the CSR and wait until the read returns the Busy Bit as a logic '`0`',
5. Go to step 3 if more bitstream chunks need to be written.
6. Disable the configuration mode by writing a logic '`0`' to the appropriate bit in the CSR.

4.2.3 GPIO Port

The General Purpose Input/Output (GPIO) Port is the interface between the DUT and the reconfigurable logic module. Each pin must be configurable as either an output or input, which is achieved by using the bidirectional buffers available in the FPGA device.

To avoid problems induced by metastability, two-stage input synchronizers are connected to each input signal.

The GPIO port includes five 32 bit wide registers that are accessible via the on-chip AMBA bus. Each bit in these registers corresponds to one of the 32 GPIO port's physical pins. These registers perform the following functions:

`GPIO_DIR_SOURCE`: Selects the source of the pin direction signal. This may either be the reconfigurable logic module (bit set to logic '`1`') or the `GPIO_DIR` register (bit set to logic '`0`'), allowing the firmware to control the pin direction.

`GPIO_DIR`: Used to set the direction of the individual GPIO pins to input ('`0`') or output ('`1`'). Only active when the direction source is configured accordingly.

`GPIO_PORT_SOURCE`: Selects the source of the logic level of a pin that is configured as an output. This may either be the reconfigurable logic module ('`1`') or the `GPIO_PORT` register, allowing the firmware to control the logic level of an output pin.

`GPIO_PORT`: Used to set the logic level of the individual GPIO pins that are configured as an output. Only active when the port source is configured accordingly.

`GPIO_PIN`: This read-only register reflects the state of the pins **after** the input synchronizers.

The structure of the GPIO port, including the input synchronizers and bidirectional buffers, is shown in figure 4.7.

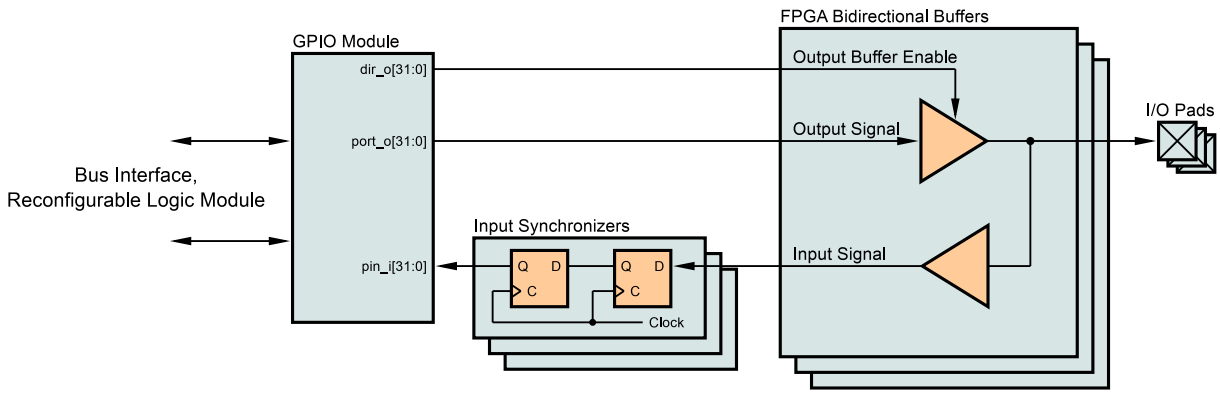


Figure 4.7: GPIO Port Structure

4.2.4 Reconfigurable Logic Module

The Reconfigurable Logic Module is the core of the digital circuit. The TR-FSMs, all data path cells and the reconfigurable interconnect are contained in this module. Internally, it is connected to the bus interface, the configuration interface and the GPIO port.

This module is derived from the example application netlists by InterSynth during the Pre-Silicon design phase. The structure of the generated module is determined by the InterSynth configuration files: The base configuration lists the instantiation details for each Verilog module as well as the top-level ports. For each example application, one additional configuration file, a *netlist*, which describes the connections between the ports of the instantiated modules, is processed.

One requirement for the automatic generation of the Reconfigurable Logic Module by InterSynth is the exclusive usage of pre-defined *connection types*. Each connection type is defined on compatible signaling (bit width) and other semantics (clock enable, other control signal). For each connection type, InterSynth generates a separate reconfigurable interconnect [WGS+12].

In the Reconfigurable Logic Module, the following connection types are used:

- Bit
- Nibble (4 Bits)
- Byte (8 Bits)
- Word (16 Bits)
- Double Word (32 Bits)

A distinction between different types of the same width was **not** made to achieve maximum flexibility. Theoretically, a second connection type with a width of one bit could have been introduced and the semantics defined as “control bit” and “data bit”, possibly resulting in a smaller overall design size, but this would have caused conflicts with the TR-FSM input and output signals: While the port semantics of a data path component can be easily determined, in the case of the TR-FSM it is possible that an output drives a data input in one example application and a control input in another.

4.2.4.1 Data Path Cells

Each of the example applications defined in section 3.4 was first implemented as a monolithic Verilog design, from which the operations supported by the cells were derived. In this section, each of the implemented data path cells is described. The TR-FSMs that are used to control the data path have already been introduced in section 2.3.1.3 and are not further dealt with in this section. The development of these cells was done in an iterative process.

GPIO Splitter

As mentioned in section 4.2.3, the GPIO port contains 32 physical pins that can either be configured as an input or output pin. However, the reconfigurable logic module uses pre-defined *connection types* of 1, 4, 8, 16 and 32 bits. Without additional circuitry, it would therefore be impossible to connect signals of other connection types than the 32-bit wide Double Word to the GPIO port, which would be of little use in real-world applications.

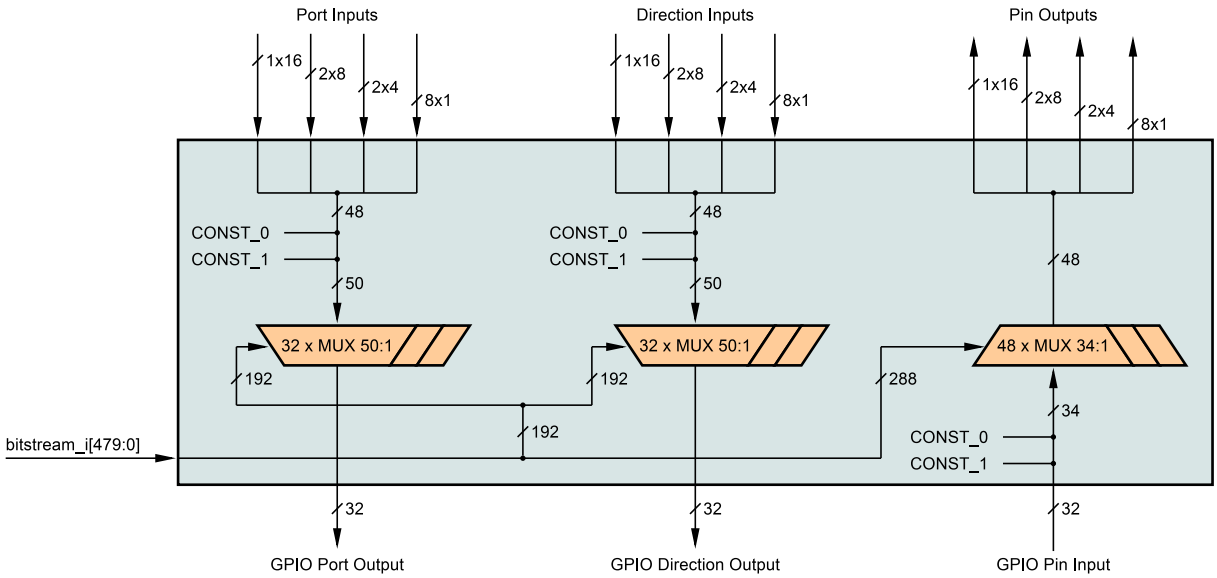


Figure 4.8: GPIO Splitter Module

Therefore, a single GPIO Splitter instance, which is shown in figure 4.8, is present in the Reconfigurable Logic Module. It has two interfaces: the GPIO side, and the reconfigurable logic side. On the GPIO side, it is connected to the Port, Direction and Pin signals described in section 4.2.3, while on the reconfigurable logic side multiple input and output ports of each connection type are present.

A set of multiplexers is present that can route any of the input bits or a constant '`0`' or '`1`' to any of the output signal bits. The selection signals of the multiplexers are concatenated and form the bitstream of the GPIO Splitter cell.

Port Converter

As mentioned in section 4.2.4, the connections between components of the data path are distinguished between their bit width (the *connection type*). A problem arises when a signal of a certain connection type needs to be connected to a port of a different connection type, e.g. a byte captured from a serial communication line needs to be stored in a First In First Out (FIFO) Buffer, which only has a 32-bit wide input port.

The Port Converter cell was therefore devised to convert between ports of different connection types. It has one input and output port of each connection type that is present in the Reconfigurable Logic Module. A two-stage conversion circuit is present in the cell:

In the **Input Stage**, a set of four multiplexers determines the data source for the conversion process.

The **Intermediate Signal Vector** is a concatenation of the multiplexer outputs of the input stage with the addition of two constant bits: a logic '0' and '1'.

In the **Output Stage**, a set of 32 multiplexers selects one of the input signal bits for each output bit. This means that the Port Converter can also be used to implement other tasks – e.g. a circuit with functionality equivalent to a barrel shifter could be implemented by configuring the multiplexers of the output stage accordingly. It is also possible to mix parts of different input signals.

The bitstream of the Port Converter cell is the concatenation of the individual selection signals of all multiplexers in the cell.

Figure 4.9 shows the Port Converter cell.

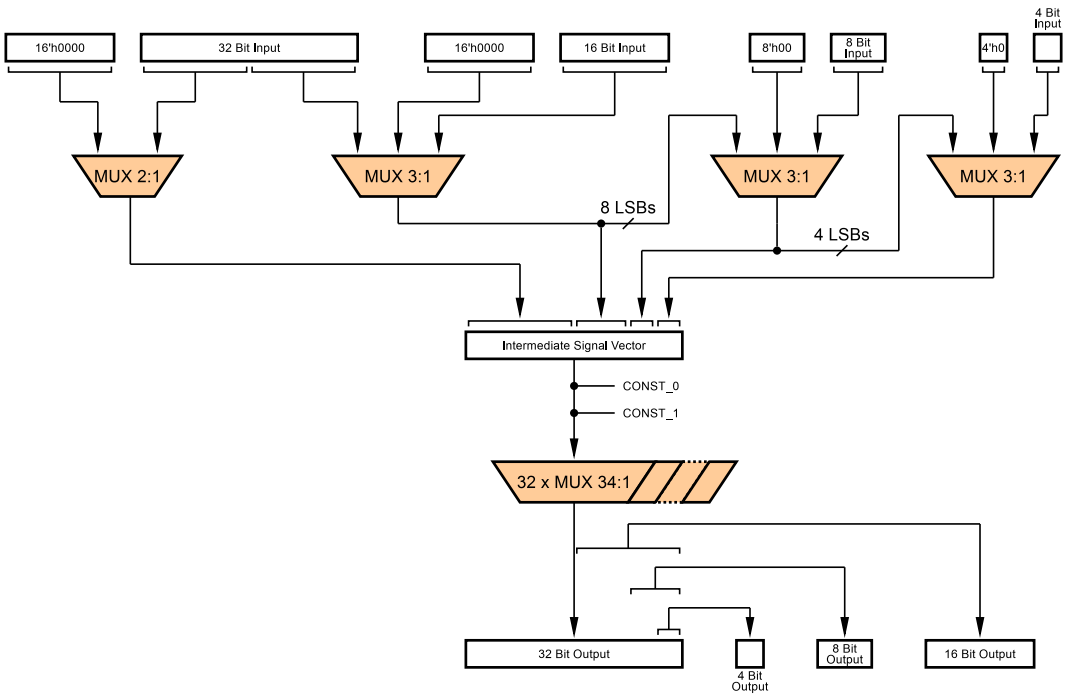


Figure 4.9: Port Converter Module

Aggregator Register

The Aggregator Register is a 32-bit wide register that can be used to assemble values from multiple data sources. This can be compared with the concatenation operator in hardware description languages, where e.g. two 8-bit signal vectors are concatenated to form a 16-bit vector. It is composed of a 32-bit wide register with one input, output and load-enable signal for each connection type that is used in the reconfigurable logic module. By asserting any of the load-enable signals, the corresponding input is loaded into the register at a certain offset which is determined by the *shift offset input*. This way, values may be assembled sequentially during multiple clock cycles.

Figure 4.10 illustrates three cycles in which various values are loaded into the Aggregator Register.

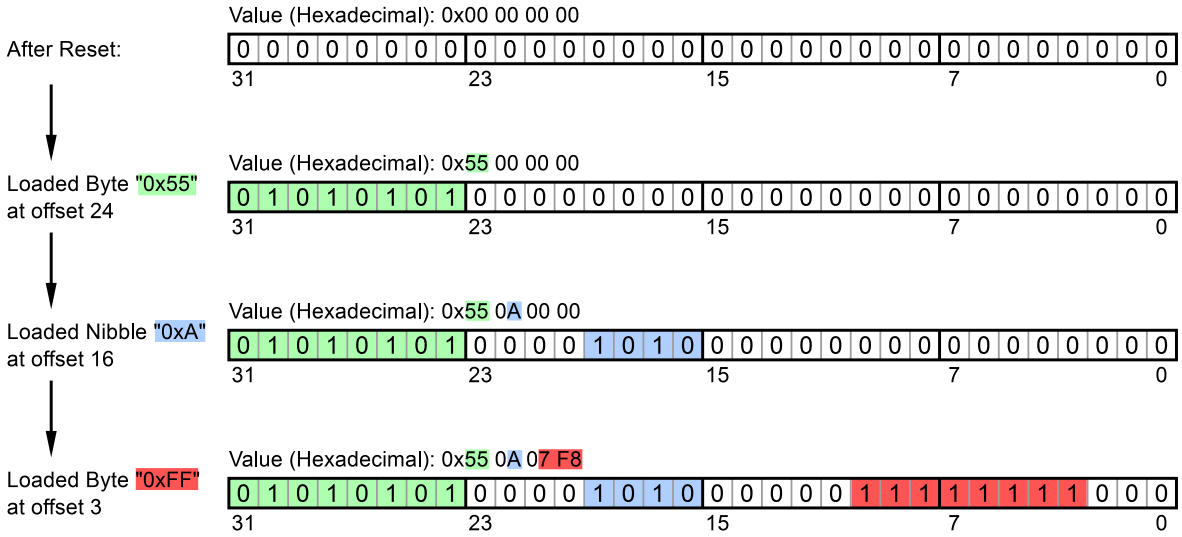


Figure 4.10: Aggregator Register Cell

Shift Register

The Shift Register cell, which is shown in figure 4.11, is a universal shift register featuring separate clear, parallel load and shift-left/right functions. It has one parallel input with a width of 32 bits which is used exclusively in conjunction with the parallel load feature and one single-bit serial data input which serves as input to the Least Significant Bit (LSB) or Most Significant Bit (MSB) when the shift-right and shift-left functions are used. There is a parallel data output signal for each connection type present in the Reconfigurable Logic Module. The MSB and LSB of the value stored in the shift register are also readable at their respective output signals.

This cell has to meet a unique requirement: while the physical size of the shift register is 32 bits, it must be configurable to behave exactly like a shift register with a *logical size* of 1 to 32. This is achieved by introducing another input, the "size" vector. Depending on this configuration input, some of the most significant bits may be masked out to logic zeroes. Also, the MSB output signal will reflect the *logical* MSB instead of the *physical* MSB (Bit 31).

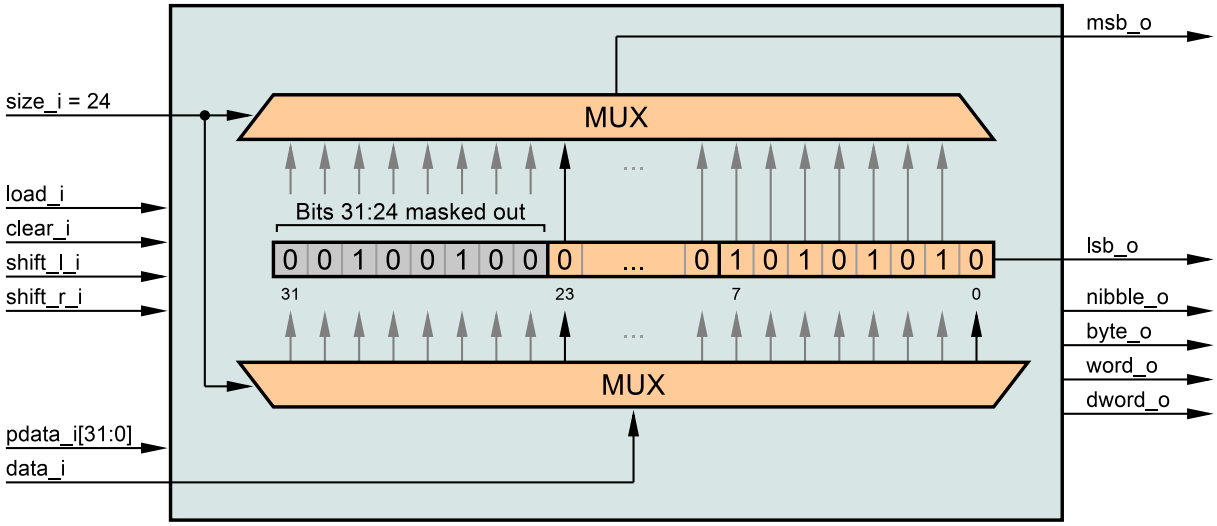


Figure 4.11: Shift Register Cell

Parameter Register

The Parameter Register, which is shown in figure 4.12, is used to store constant values of varying width. Instances of this component are connected to a set of top-level ports of the Reconfigurable Logic Module which in turn are connected to the Bus Interface described in section 4.2.1. This makes the parameter registers writable by the system CPU and readable by the logic inside the Reconfigurable Logic Module. Write accesses are always 32 bits wide (i.e. each write access overwrites the entire contents).

Physically, the Parameter Register has a width of 32 bits and possesses several output ports of each connection type that is present within the reconfigurable logic module.

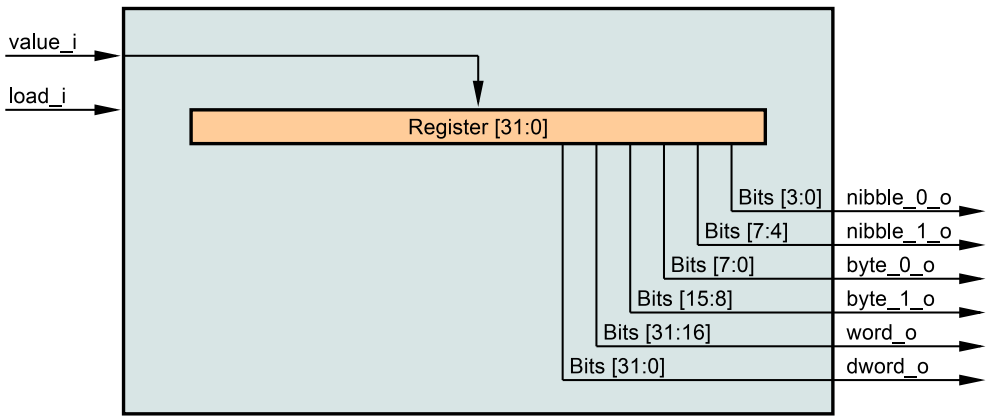


Figure 4.12: Parameter Register Cell

Arithmetic Logic Unit

The ALU cell implements various computations:

- Addition,
- Subtraction,
- Multiplication,
- Comparison,
- Logical AND,
- Logical OR.

The cell has two 32-bit wide operand inputs and five control inputs that trigger the computations listed above. All operations are performed within a single clock cycle. The result of these operations is available at a 32-bit wide output signal. Various status outputs are present that indicate specific conditions that may occur during execution of an operation:

- Carry Flag
- Zero Flag
- Equal Flag: set during comparison operation when both operands are equal
- Greater Flag: set during comparison operation when operand A is greater than operand B
- Lesser Flag: set during comparison operation when operand A is smaller than operand B

Content Addressable Memory

In software as well as in digital logic circuits, often a decision must be made depending on the value of a variable or signal. With the TR-FSM and the data path components introduced earlier, two possibilities to implement such functionality are available:

- A FSM may set its output value depending on an input value. This method is preferred when some action depends on the value of a single bit.

- For multi-bit signals, the ALU cell described above can be used to compare the signal to a known reference, e.g. a value stored in a parameter register.

With these two methods, a large number of decision-making processes can already be implemented. However, it is still impossible to efficiently compare a signal to *multiple* reference signals or map one set of signals to a different set (a large number of parameter registers and big multiplexers would have to be used for these tasks).

Therefore, the Content Addressable Memory (CAM) cell was developed. A CAM, also known as an associative memory, is a memory capable of searching its contents for occurrences of an input data word. Compared to a RAM, which retrieves content after an address word is applied, a CAM retrieves the address after a data word is applied. In an extended version, instead (or in addition to) returning the address, a second data word is retrieved [MTM01].

Figure 4.13 shows the CAM cell. The contents of the individual CAM memory cells are determined by the respective parts of the CAM's bitstream.

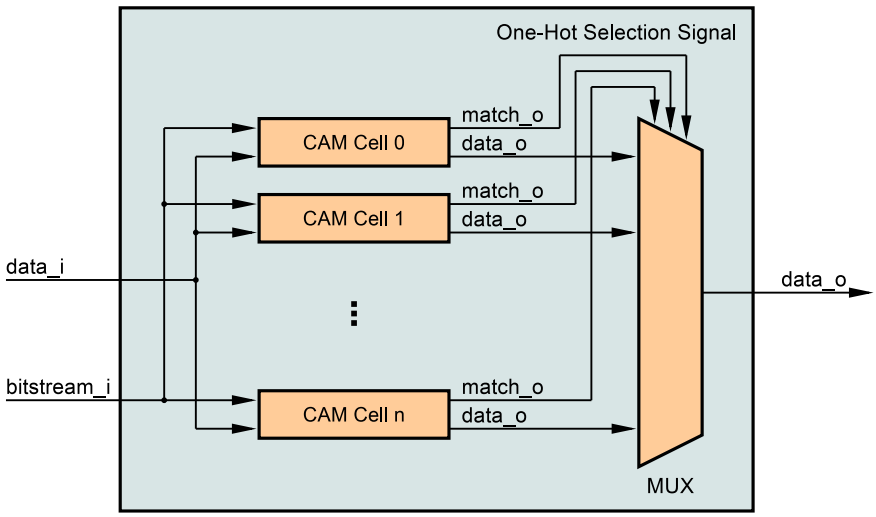


Figure 4.13: Content Addressable Memory cell

General Purpose Counter

The General Purpose Counter cell implements a simple downward-counting timer/counter capable of generating an underflow signal.

It contains two registers: the *reload value* and the *current value*.

By applying the reload value to a parallel input and asserting the reload signal, the reload value is stored in the corresponding register. This process also overwrites the contents of the current value register with the reload value.

When the *count enable* signal is asserted, the counter decrements the current value register by one on each clock cycle. Upon reaching zero, an underflow event is generated and the reload value is again loaded into the current value register.

The counter cell has room for improvement: the ability to count upwards is missing – while the current design is sufficient for the example applications described in section 5.1, it may be insufficient for other applications where more flexibility (e.g. count occurrences of an event, then further process this value) is required.

To save resources, several versions with different bit widths (8, 16 and 32) are present in the Reconfigurable Logic Module. These different versions are synthesized from the same HDL code by making use of the parameterization feature present in the Verilog language.

D Flip-Flop with Enable Input

A D flip-flop (also called *data* or *delay* flip flop) normally stores the value on its data input during each clock cycle, propagating it to a data output. This allows to store a single bit for exactly one clock cycle. By adding a small multiplexer which – depending on the state of the Enable signal – feeds either the data input or output to the flip flop, the possibility to retain the value for an indefinite amount of clock cycles is created.

This cell is used whenever it is necessary to store the value of a single bit, where the usage of the fully-featured aggregator or shift register cell is not justified.

Figure 4.14 shows the structure of the D Flip-Flop cell.

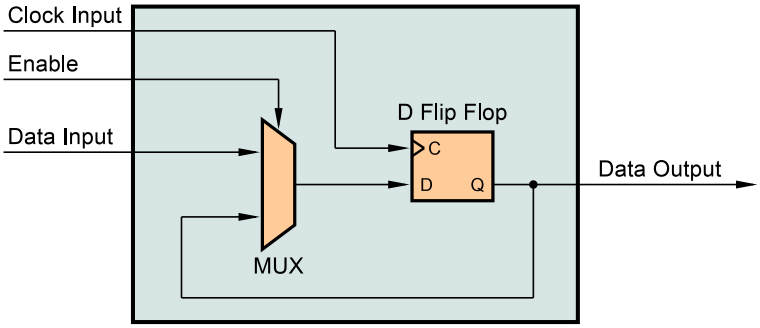


Figure 4.14: D Flip-Flop with Enable Input

Multiplexer

During development of the example applications, the need to apply signals from different sources to an input port of a cell arose. One example is a shift register which needs to be loaded with values from different parameter registers. This problem can be solved by including instances of this Multiplexer cell in the design.

The Multiplexer has four data inputs and a single data output of a specific connection type. It also features a 2-bit wide selection signal, which is splitted into two 1-bit lines. This allows the selection signal to be driven by two outputs of a TR-FSM instance or a single output of a TR-FSM while the other bit is tied to a constant logic 0 or 1, effectively operating as a multiplexer with only two inputs and freeing up a TR-FSM output signal for other uses.

Several versions of the multiplexer are present in the Reconfigurable Logic Module, each implementing a different connection type on the input and output ports. Figure 4.15 shows the structure of this cell.

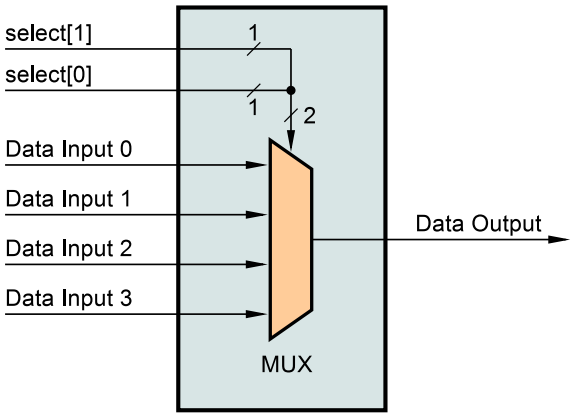


Figure 4.15: Multiplexer Cell

Dual-Port FIFO Buffer

When transferring data that was generated by the DUT or the Logic Exerciser from the reconfigurable logic module to the firmware running on the host CPU, it is required to buffer the data in some dedicated storage container. If the various registers in the Reconfigurable Logic Module were directly accessible via the bus interface, the CPU might capture their contents at the wrong moment (e.g. when the read access is delayed due to a network management task), sending erroneous data to the host. To eliminate this possibility, the *Dual-Port FIFO Buffer*, which is capable of storing a number of 32-bit values, was designed.

The FIFO cell is implemented as a circular buffer with one write pointer and two independent read pointers. This means that one write interface and two read interfaces – the *primary read interface* is connected to the bus interface, the *secondary read interface* is exposed to the reconfigurable logic – are available. It is therefore possible to extract data for further processing on one read interface before it is read by the system CPU. New data can only be written if it would not overwrite data that has not been read via both read interfaces. If only one read interface is used, the read enable signal of the other must be tied to a logic '`1`', otherwise the FIFO buffer will remain in a stalled state once it has been filled.

The Dual-Port FIFO Buffer also implements two status signals to indicate whether it is empty or full. The structure of the cell is shown in figure 4.16.

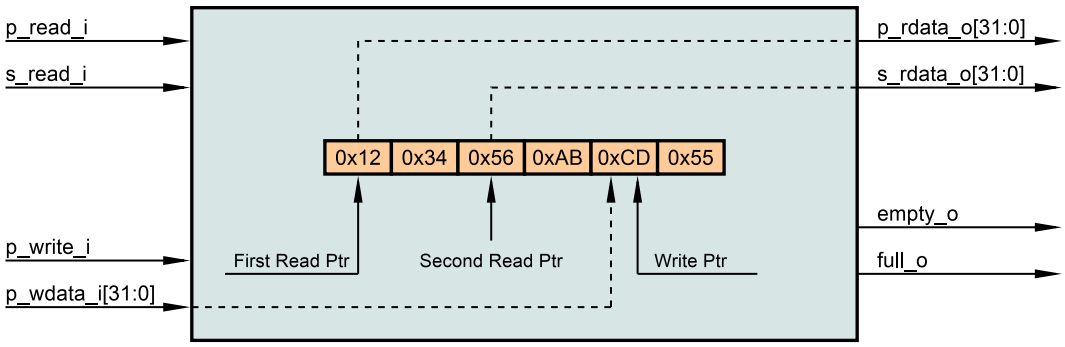


Figure 4.16: Dual-Port FIFO Buffer

"Constant" Cell

This cell provides constants for each connection type present in the reconfigurable logic module. This is required because all input ports of the cells must be connected to a valid signal source at all times. For every connection type present in the reconfigurable logic module, the "Constant" cell has one output port which is fixed to a logic '`0`' value. For the "Bit" connection type, a constant with a value of '`1`' is also provided. The constants generated by this module are used to disable certain features of the cells which are unused in an application (for example, an application may include an ALU cell but never use the multiplication operation. In this case, the corresponding input is disabled by connecting it to the constant '`0`' output of the "Constant" cell).

4.3 Firmware

This section describes the logic exerciser firmware running on the system CPU. It has two basic tasks:

- Handle communication between the host and the logic exerciser,
- Read and write the logic exerciser peripheral's registers, either when executing commands received from the host or when new data has been captured or generated by the logic exerciser.

4.3.1 Communication between Logic Exerciser and Host

The layered Open System Interconnection (OSI) model devised by the International Organization for Standardization (ISO) is the basis for many modern network architectures. This model defines standard procedures for the interconnection of network systems [GG04]. It introduces seven layers, each providing interfaces to higher and lower layers. Figure 4.17 shows the OSI model with the three topmost layers combined into one.

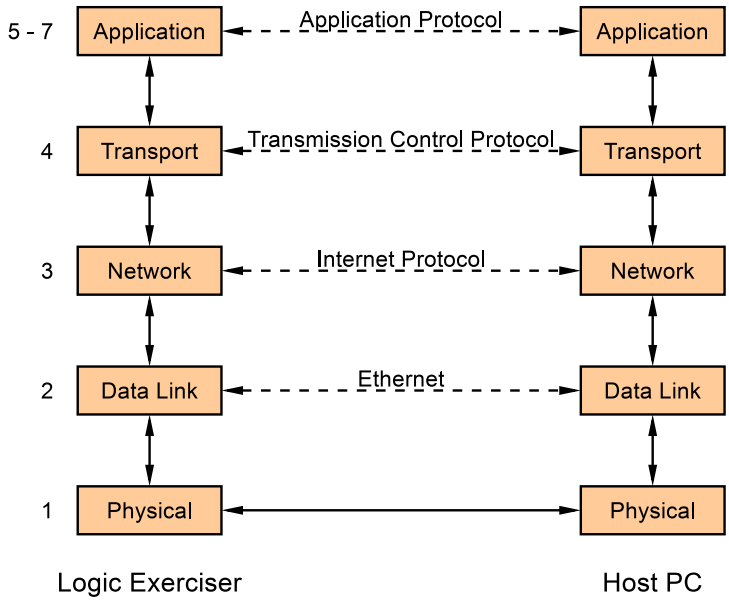


Figure 4.17: ISO OSI model

The *Application Protocol* (Layers 5-7) was implemented in the firmware. The remaining protocols are provided by a TCP/IP Stack. As shown in figure 4.17, the application protocol interfaces with the Transport Layer. Two options are available here: The UDP and the TCP. Because TCP automatically handles retransmissions and flow control, it was chosen as the transport layer protocol used by the logic exerciser.

The Lightweight IP (lwIP) stack [39] was chosen to be used in the firmware implementation. This open-source TCP/IP stack can be used with or without an operating system. It provides three different Application Programming Interfaces (APIs): *netconn*, *socket* and *raw*. When run without an operating system, only the raw API is available for use [40].

In the logic exerciser firmware, multi-tasking is not needed – instead, the main requirements for the firmware are high data throughput and low latency. With these requirements in mind, a so-called *bare-metal* firmware program utilizing the lwIP raw API was developed.

After power-on, the firmware configures the network interface and begins listening for incoming connections. After the host establishes the TCP connection, a stream-based full-duplex communication channel is available for data transmission between the host and the logic exerciser. The host may then begin sending *commands* to the logic exerciser, which may include writing to the parameter registers in the reconfigurable logic module, configuration of the reconfigurable logic module, configuration of the GPIO port and enabling/disabling of data acquisition. The Logic Exerciser may begin sending *messages* containing generated or acquired data to the host.

4.4 Host Software

The host software consists of two programs written in the C++ programming language: a Graphical User Interface (GUI) utility designed for the end-user of the logic exerciser, and a command-line program called `bsgen` that integrates with the design flow described in section 3.3. The source code of both programs is structured in a single directory, as some modules are used by both projects. The CMake cross-platform build system [36] is used to manage the build process of both programs. Both programs are described briefly in this section.

4.4.1 Logic Exerciser Control Program

This program is used to communicate with the logic exerciser in two different kinds: *Configuration* and *Visualization*. Configuration refers to the process of downloading configuration and parameterization data to the Logic Exerciser device. This includes importing the bitstream generated by the tools used in the coarse-grain reconfigurable logic design flow. For this role, the program features a Tcl interface which is capable of executing external script files that contain configuration commands. Visualization is the process of displaying data received from the logic exerciser to the user.

The Logic Exerciser Control Program makes extensive use of the Qt cross-platform application and User Interface framework [52] to abstract GUI and networking functions as well as cross-platform support for different operating systems (Linux, Apple Mac OS X and Microsoft Windows). Figure 4.18 shows the application's GUI. The upper portion of the GUI window is used to enter the IP address and the TCP port of the Logic Exerciser. The middle part shows three *channels* in a tabular view. The data source for these channels are the FIFO cells in the reconfigurable logic module, which are connected to the bus interface and read by the Logic Exerciser firmware. The data read from the FIFOs is then transmitted to the host software via the TCP connection and displayed by the GUI.

Tcl Commands Supported by the Logic Exerciser Control Program

`channel_name <channel #> <name>`

Sets the channel name in the Visualization GUI.

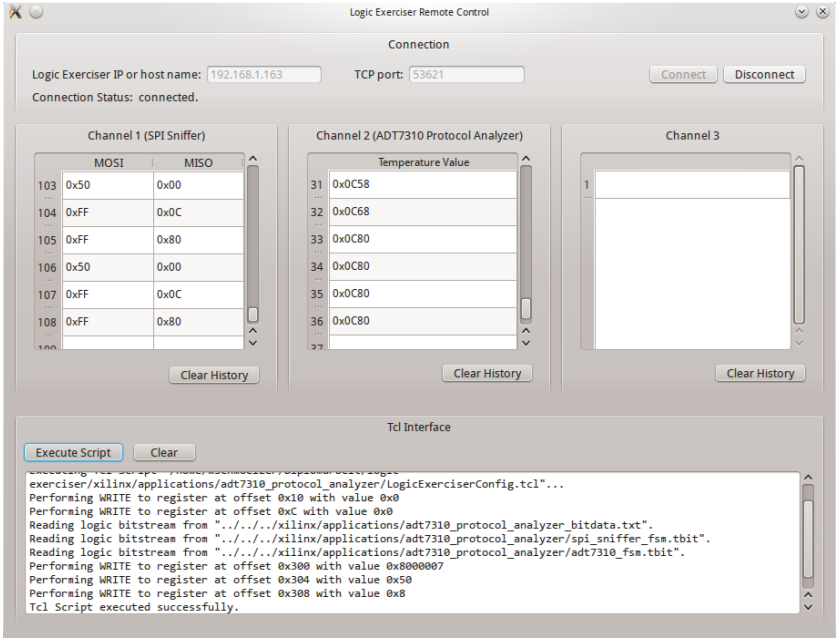


Figure 4.18: GUI Logic Exerciser Control Application

`register <REGISTER_NAME> <OPERATION> <value>`

Performs a write access to one of the Logic Exerciser's internal registers. Valid register names are: `GPIO_DIR_SOURCE`, `GPIO_DIR`, `GPIO_PORT_SOURCE`, `GPIO_PORT` and `PARAM_0` to `PARAM_4`.
Valid operations are `WRITE`, `WRITE_AND` and `WRITE_OR`.
Values may be specified in any format supported by the Tcl interpreter.

`cfgmode [-enable trfsm | interconnect ] [ -disable trfsm | interconnect ]`

Enables or disables the configuration mode for either the TR-FSMs, the interconnect, or both.

`configure_logic <destination #> <bitstream file>`

Shift configuration bitstream into either the interconnect and data path components (destination `0`) or a single TR-FSM instance (destination >`0`).

4.4.2 Bitstream Generation for Data Path Cells

The program `bsgen` is used to generate the bitstream for the following data path cells:

- GPIO Splitter,
- Port Converter,
- Content Addressable Memory.

It is also used to convert the bitdata files generated by InterSynth to the text format that is understood by the GUI control program and the Verilog testbenches used during development of the reconfigurable logic module.

The program includes a Tcl interface using the Tcl/Tk 8.6 library [41]. It can read Tcl script files or execute Tcl commands supplied via the command-line, e.g. from within a Makefile, thus integrating well into existing design flows. The following sections describe the available commands.

The internal representation of the data path cell used to generate the bitstream is hereafter referred to as a *configuration model*. Each configuration model must be given a unique name.

Tcl Commands Supported by the bsgen Tool

```
create_gpio_splitter <name>
```

Create a new configuration model for a GPIO Splitter cell.

```
gpio_input -gpio <x>
           -bank <WORD_0 | BYTE_1 | BYTE_0 | NIBBLE_1 | NIBBLE_0 | BIT_0>
           -pin <y>
```

Configures an input multiplexer of the GPIO Splitter cell. This causes the input pin with the index `<x>` (0..31) to be routed to pin `<y>` within a certain bank.

```
gpio_output -bank <WORD_0 | BYTE_1 | BYTE_0 | NIBBLE_1 | NIBBLE_0 | BIT_0>
            -pin <x>
            -gpio <y>
```

Configures an output multiplexer of the GPIO Splitter cell. This routes the input pin `<x>` from the specified bank to be routed to the output pin `<y>`

```
create_port_converter <name>
```

Creates a new configuration model for a Port Converter cell.

```
config_input_stage <INPUT_32_BIT | INPUT_16_BIT | INPUT_8_BIT | INPUT_4_BIT>
```

Sets the input stage of the Port Converter to either the 32-, 16-, 8- or 4-bit input port.

```
config_output_stage <output bit> <input bit>
```

Configures an output pin to either reflect the state of an input pin or an internally generated constant (by passing `CONST_0` or `CONST_1` as the `<input bit>` argument.

```
create_cam <name> -input_width <int> -output_width <int> -cells <int>
```

This command creates a new configuration model for a CAM cell with a given input and output width (up to 32 bits) and a given number of cells.

```
cam_cell <input value> <input mask> <output value>}
```

Adds a cell with given input value, input mask and output value to the CAM configuration model.

```
write_bitstream [ -format human-readable | text | verilog | intersynth ]
                [ -o </path/to/file> ]
```

This command can be used for any of the cells described above. It writes the cell bitstream to a stream (stdout or file) in a specified format.

5 Results and Discussion

In this chapter, the ability of the presented coarse-grain reconfigurable logic circuit to implement real-world applications is evaluated. The example applications which have been specified in section 3.4 were designed with the tools and methods presented in the previous chapters. These applications are implemented by the reconfigurable logic module of the Logic Exerciser device. In each case, supplemental hardware, digital logic or software was used to act as a DUT. The first part of this chapter describes the implementation details of each example applications, while the second part provides ideas for future enhancements and further automation of the design flow.

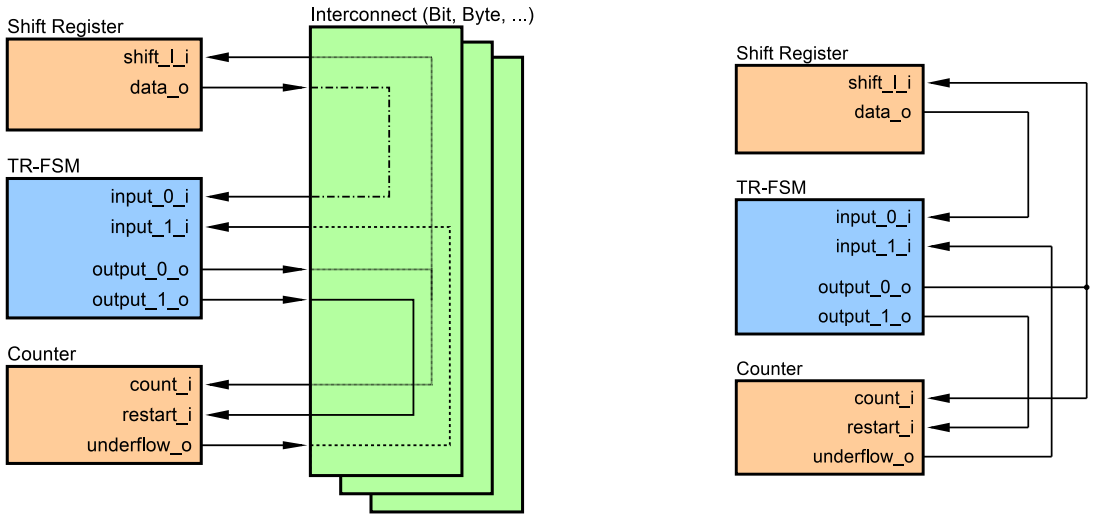
5.1 Example Applications

Each example application was first implemented as a behavioral Verilog description to gather information about the required cells which comprise the reconfigurable logic module. From these examples, the required cells (as described in section 4.2.4) were derived, implemented in Verilog and verified by simulation. Each example application was then re-implemented as a structural Verilog description instantiating only the reconfigurable cells. Like the individual cells, these applications were also verified using simulation and custom testbenches. They were then manually converted to the netlist format used by InterSynth. This tool was used to analyze, optimize and implement the reconfigurable logic module. The number of instances of each cell present in the resulting reconfigurable logic module depends on the cell usage of the example applications processed during the pre-synthesis design flow. Each example application is realized by re-arranging the connections between these cells and by modifying the behavior of certain cells that can be reconfigured by themselves.

The description of the example applications includes block diagrams of the connections between the cells (as generated by InterSynth). To reduce the complexity of these diagrams, a few simplifications have been made:

- The interconnect is not shown as illustrated in figure 5.1.
- The parameter registers, which typically store constant values, are not shown. Instead, a constant input signal is shown as an arrow to an input port with its value shown in Verilog format (e.g. `1'b0`, `8'h1A`, `16'd2587`).

- Inputs and outputs to the cells that are irrelevant to a specific example application (and may be connected to a constant '0' or '1' signal) are omitted.



(a) Actual topology with interconnect shown

(b) Simplified block diagram

Figure 5.1: Example application diagrams

5.1.1 SPI Sniffer

The SPI Sniffer example application implements a device that listens to all four SPI signals and extracts the data transmitted over the interface. The extracted data can be used by another application, e.g. the Protocol Analyzer described in section 5.1.2, or simply displayed on the host for debugging purposes (for example when designing software or hardware that communicates over an SPI interface). To simplify the development of the example application, the implemented logic is limited to the CPHA and CPOL parameters both being equal to zero. Four pins of the GPIO port are configured as inputs and are used to route the SPI signals into the reconfigurable logic module.

The structure of the SPI Sniffer application is shown in figure 5.2. The following cells are used in this application:

- One TR-FSM,
- Two shift registers to gather the data transmitted on the MOSI and MISO lines,
- One counter to keep track of the number of bits transmitted,
- One aggregator register to assemble words consisting of MOSI and MISO data,
- Two multiplexers to switch between MOSI and MISO data as aggregator inputs.

A single FSM is used to track the state of the SPI interface as well as control the data path up to the point where data is stored in the CPU-readable FIFO buffer. The reload value of the counter cell is set to the number of bits in a frame, e.g. eight for most cases. Each time the FSM detects a valid SCK edge, the state of the MISO and MOSI signals is stored in the shift registers and the frame counter is decremented. Once the frame counter signals an underflow, the MOSI and

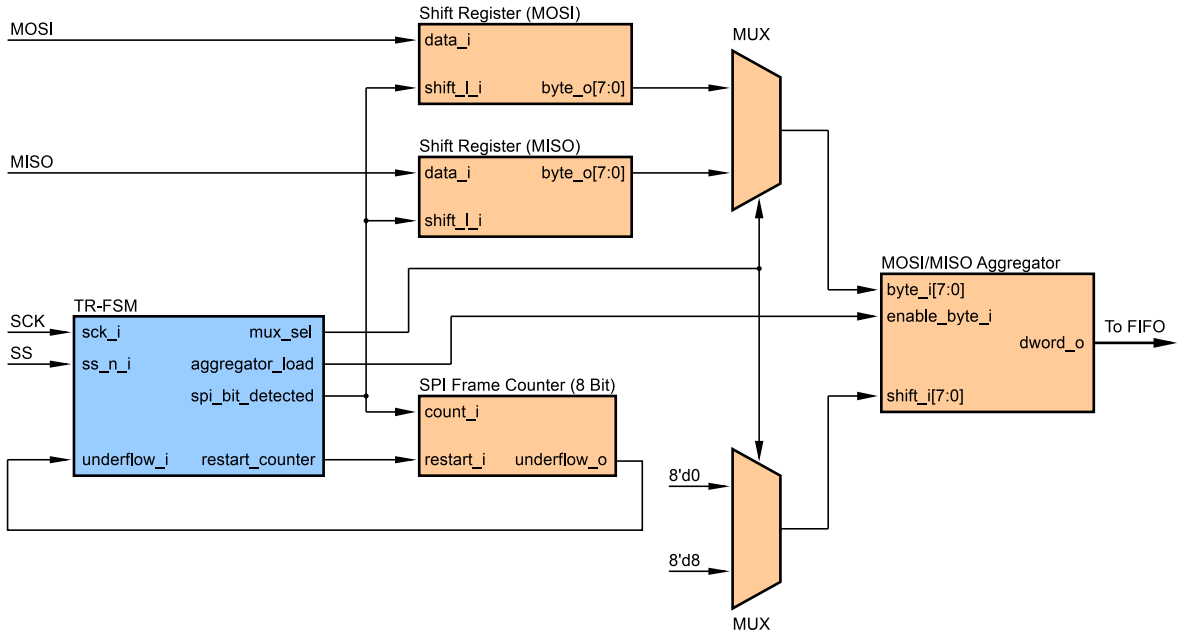


Figure 5.2: Block diagram of SPI Sniffer Application

MISO values of one entire frame have been stored in the shift registers. These values are then combined in the aggregator register and finally stored in a CPU-readable FIFO buffer, where they can be further processed and/or transmitted to the host.

5.1.2 Protocol Analyzer for ADT7310 Sensor

The Protocol Analyzer application is an extension to the SPI Sniffer application described in section 5.1.1. The SPI Sniffer is used to listen to the communication between an SPI master and an ADT7310 sensor, and the Protocol Analyzer keeps track of the register access commands sent by the master and extracts the raw temperature values transmitted by the sensor. This is done by reading the "sniffed" raw SPI data bytes from the secondary read interface of the dual-port FIFO buffer cell and by deriving the sequence of commands issued by the SPI master to the ADT7310 sensor from these SPI data bytes. The Olimex STM32-P103 development kit [48] is used in conjunction with the CHLL ADT7310 sensor attached to the SPI breakout board (introduced in section 4.1.2) to provide SPI communication for the Protocol Analyzer to listen to.

Figure 5.3 illustrates the ADT7310 Protocol Analyzer application. In addition to the cells used by the SPI Sniffer application, the following cells of the reconfigurable logic module are used in the Protocol Analyzer application:

- One TR-FSM,
- Two port converters used to extract the MOSI and MISO data from the FIFO values,
- One ALU to compare MOSI data to known ADT7310 commands,
- One Aggregator Register to assemble the temperature word from two MISO bytes,
- One Multiplexer to switch between the aggregator shift values.

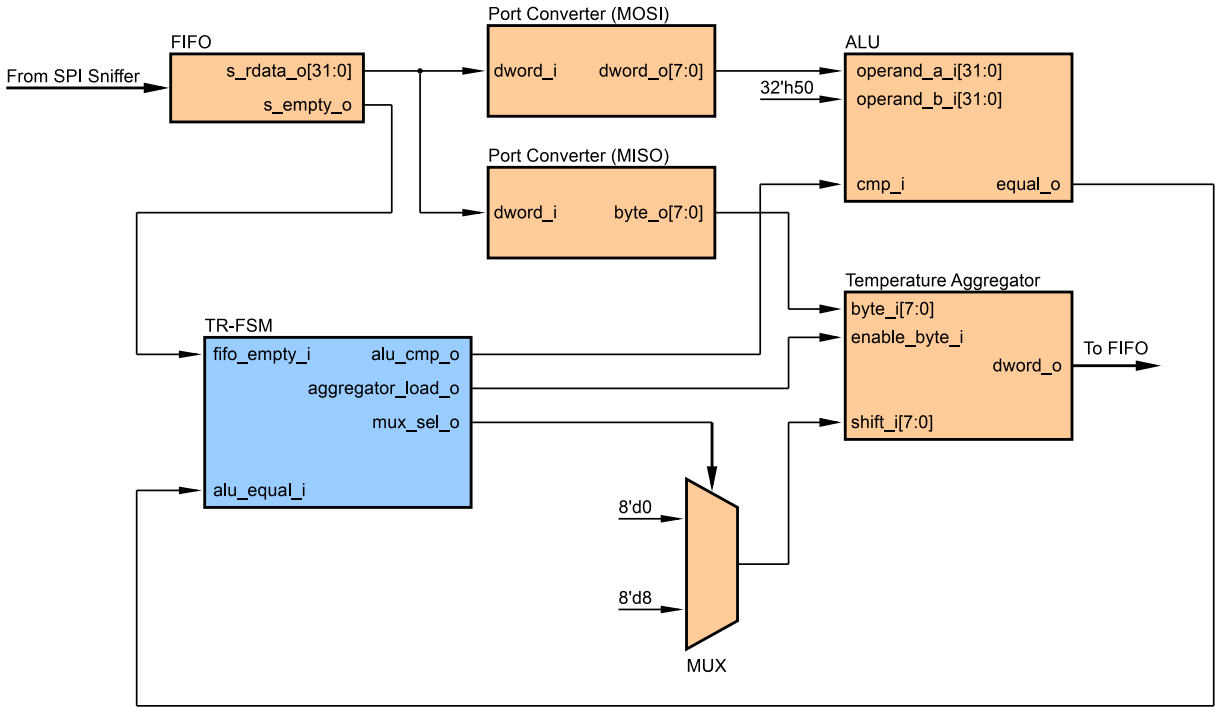


Figure 5.3: Block diagram of ADT7310 Protocol Analyzer Application

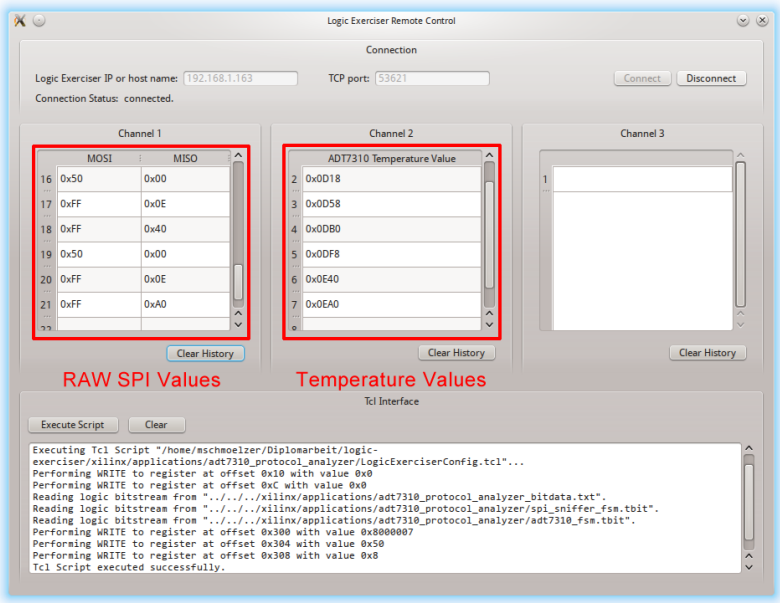


Figure 5.4: GUI Logic Exerciser Control Application

Once the Protocol Analyzer FSM detects available data on the secondary read interface of the FIFO cell, it uses an ALU peripheral to compare the MOSI data to the command used by the SPI master to read the sensor's Temperature Value Register. Then, the 16-bit raw temperature value is assembled with the help of an Aggregator Register. Finally, the 16-bit temperature value is

stored in another CPU-readable FIFO buffer, from which it is transmitted to the host, as shown in figure 5.4. According to the ADT7310 data sheet [32], the temperature values in this picture are in the 26 to 30 °C range $\left(\frac{\text{sensor value}>>3}{16}\right)$.

5.1.3 SPI Master/MAX6682 Exerciser

This application implements an SPI master that continuously queries the sensor and raises an alarm if the temperature exceeds a pre-determined value (temperature guardian). In addition, the received temperature values are transmitted to the host. Two pins of the GPIO port are configured as outputs (SS and SCK) and one pin is configured as an input (MISO). Because the SPI-compatible interface of the MAX6682 sensor is read-only, the MOSI signal can be left unconnected.

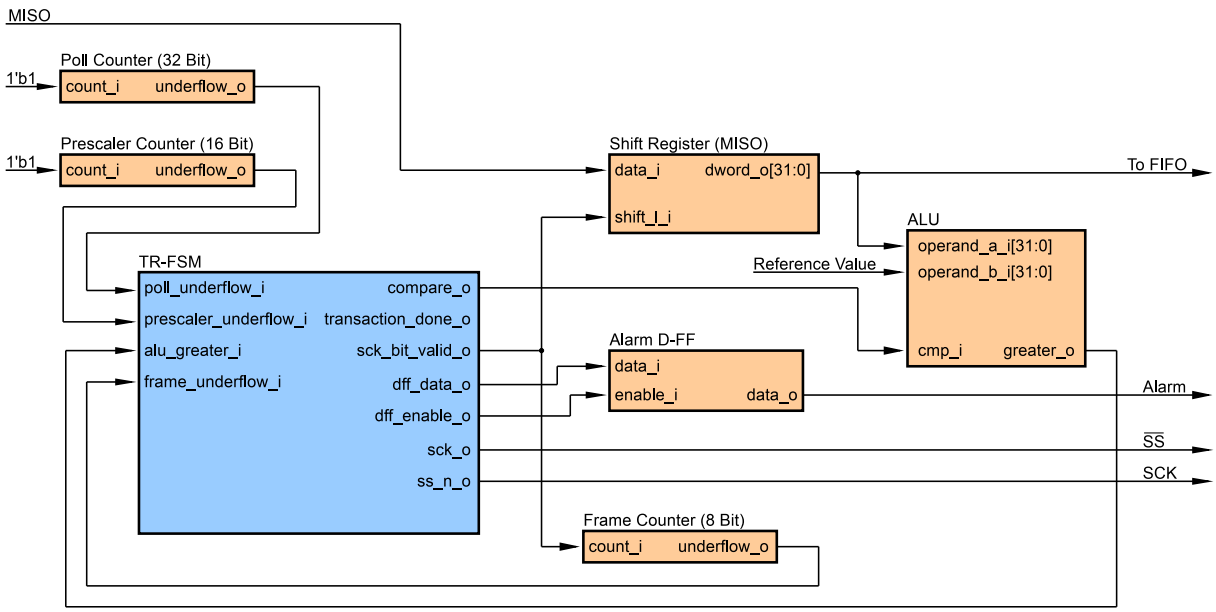


Figure 5.5: Block diagram of MAX6682 Exerciser Application

Figure 5.5 illustrates the MAX6682 Exerciser application. The following cells of the reconfigurable logic module are used in this application:

- One TR-FSM,
- One 32-bit counter used to periodically trigger the retrieval of the temperature value,
- One 16-bit counter used as a prescaler to set the SCK frequency,
- One 8-bit counter used as an SPI frame counter,
- One shift register used to store the temperature value,
- One D-Flip-Flop to used to store the alarm state.

The 32-bit counter is continuously left in a running state to periodically generate underflow events (the interval at which these events occur is determined by the reload value of the counter). The FSM waits for these events. Once an underflow is detected, it asserts the SS signal, generates an SCK cycle, advances the frame counter and finally stores the state of the MISO signal in the shift

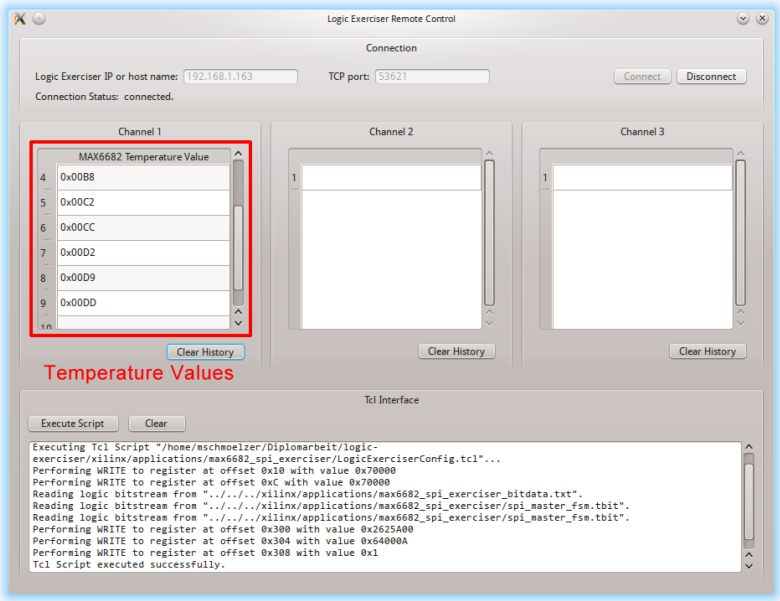


Figure 5.6: GUI Logic Exerciser Control Application

register. This is repeated until the frame counter, which is pre-loaded with the constant number 11 (the number of data bits sent by the sensor), generates an underflow. Finally, the SS signal is de-asserted.

Now, the value in the shift register is compared to a reference value stored in a parameter register, and an alarm signal is asserted if the received temperature value is larger than the reference value. This is done by the FSM with the help of a Flip-Flop, allowing the alarm signal to persist until the temperature drops below the reference value. The alarm signal could then be connected to a microcontroller GPIO port capable of generating edge-triggered interrupts to act on the alarm condition.

Figure 5.6 shows the GUI on the host displaying the retrieved temperature values from the MAX6682 sensor.

Compared to a purely microcontroller-based approach, this implementation of a temperature guardian has two main advantages:

- It allows the system CPU to remain in an energy conserving state while the hardware autonomously monitors the temperature by polling the sensor,
- It is not necessary to generate more than eleven SCK cycles, which would be the case with many commercial microcontrollers that generate SPI frame sizes in multiples of 8. This saves energy in both the host **and** the sensor.

Because the MAX6682 is a read-only device, only the reception of data is implemented. The application could easily be extended to send data on the MOSI line by introducing a second shift register and making some modifications to the FSM.

5.1.4 I²C Master

This application implements an I²C master which is able to address slaves which support the 7-bit slave address format defined in [47] and interact with them by sending and receiving frames of arbitrary length. As already mentioned in section 3.4.4, the complexity of the I²C bus with regard to the implementation effort in the coarse-grain reconfigurable logic circuit designed for the Logic Exerciser is much higher than in the previous example applications. However, compared to the application described in the previous sections, the I²C Master does not use more instances of the reconfigurable cells – most of the complexity is contained in the two FSMs that govern the data path cells.

The structure of this example application is shown in figure 5.7. It utilizes the following cells of the reconfigurable logic module:

- Two TR-FSMs,
- One 8-bit counter used to implement a configurable delay between SCL and SDA transitions,
- Another 8-bit counter used to count the I²C bits,
- One 16-bit counter used as a prescaler to set the SCL frequency,
- Two D-Flip-Flops used to store the state of the SCL and SDA drivers,
- One Shift Register used to store the I²C address, R/W bit and data.

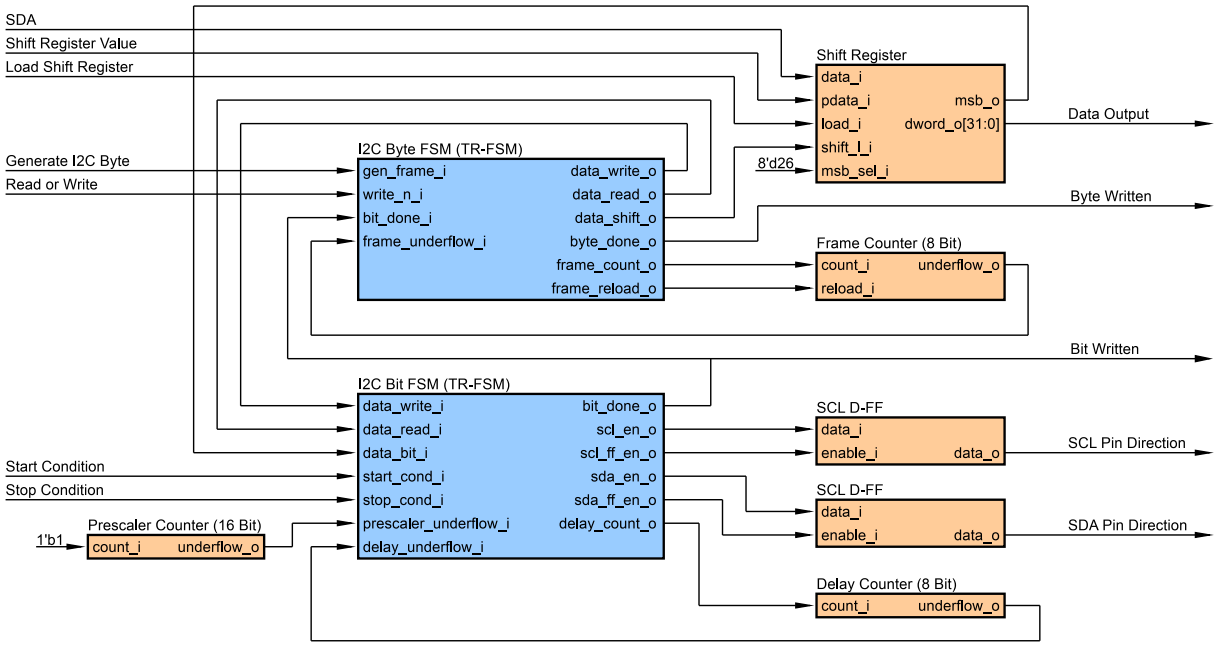


Figure 5.7: Block diagram of I²C Master Application

The two FSMs present in this application are organized in a hierarchical structure. One is responsible for generating bit-level signals (start and stop conditions, reading and writing bits) while the other keeps track of the transmission of a byte (address or data) plus the ACK/NACK bit. An I²C transaction is started by loading the address and data into the shift register. Because the physical size of the shift register is 32 bits, a maximum of three I²C bytes consisting of the 7-bit address, two data bytes and three ACK/NACK bits can be processed sequentially, before another

access to the shift register is necessary (for three I^{2}C bytes, this calculates to $7+2\cdot8+3 = 26 \leq 32$ bits; another I^{2}C byte does not fit). The data source for the shift register might be a CPU-writable FIFO buffer or parameter register.

The start and stop conditions are generated by applying a logic '`1`' to the respective input of the first FSM (called the "I^{2}C Bit FSM"). The transmission of a byte – address or data – is started by applying a logic '`1`' to the input of the second FSM (called the "I^{2}C Byte FSM"). This FSM then uses the I^{2}C Bit FSM to generate eight bits (the semantics – address or data – are determined by the shift register contents) and one ACK/NACK bit.

The I^{2}C Bit FSM employs an 8-bit counter to introduce a configurable delay between generating a negative edge on the SCL line and the level change on the SDA line. This is necessary to ensure that no unwanted start or stop conditions occur when the SDA line rises faster than the SCL line. A 16-bit counter is used as a prescaler for the I^{2}C Bit FSM; its reload value determines the SCL frequency.

Because the SCL and SDA lines are connected to a positive supply voltage via pull-up resistors or current sources [47], a dominant logic '`0`' is generated by actively driving the respective signal low. A recessive logic '`1`', on the other hand, is generated by turning off the signal driver. In the Logic Exerciser, the GPIO pins for these lines are configured as inputs when listening or "generating" a logic '`1`', and configured as outputs when generating a logic '`0`'. Two D-Flip-Flops are used to buffer the GPIO direction signals for the SCL and SDA pins. This allows the FSM to switch states and do other processing during an SCL cycle.

5.1.5 Protocol Exerciser for ADT7410 Sensor

The structure of the Protocol Analyzer application is illustrated in figure 5.8. In addition to the cells used by the I^{2}C Master application described in section 5.1.4, the following cells of the reconfigurable logic module are used in the Protocol Exerciser application:

- One TR-FSM,
- One 32-bit multiplexer used to load different values into the I^{2}C Master's shift register,
- One 32-bit counter used to periodically trigger the retrieval of the temperature value,
- One port converter used to extract the temperature word from the I^{2}C input data,
- One FIFO used to store the temperature values.

To retrieve a temperature value from the sensor, two separate I^{2}C transactions must be made:

- The first transaction writes the value `0x00` to the sensor's *address pointer register*. This marks the sensor's *temperature value register* as the data source for the next I^{2}C read access.

- The second transaction reads the contents of the *temperature value register* from the sensor, completing the retrieval of a single 16-bit raw temperature value.

The application uses two parameter registers as storage for these pre-defined I^{2}C transactions. The FSM waits for the poll timer to generate an underflow, then executes both transactions sequentially. Then, the 16-bit raw temperature value is stored in the I^{2}C Master's shift register, but mixed with other data bits (e.g. ACK/NACK bits). A port converter is therefore used to

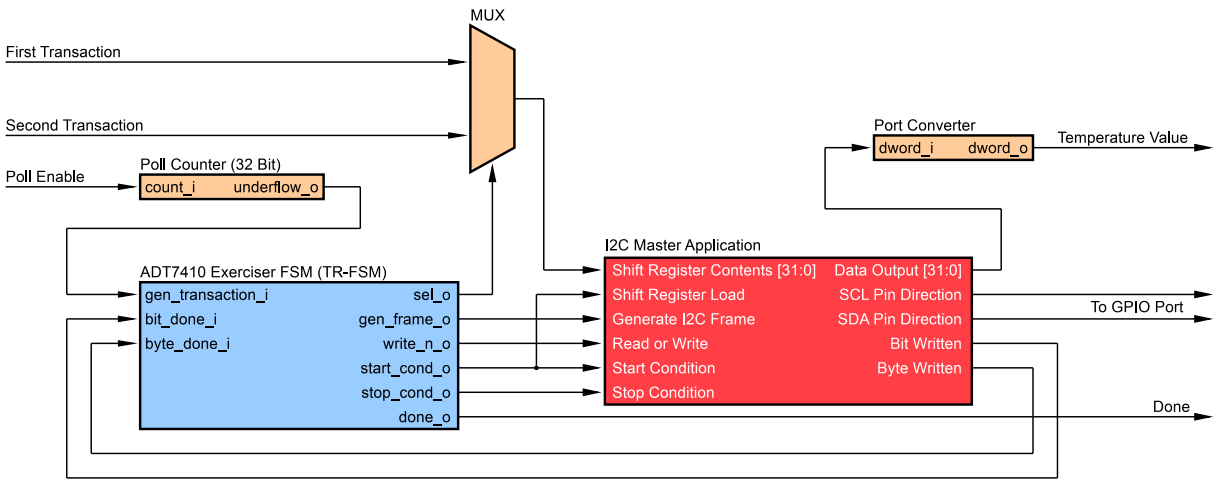


Figure 5.8: Block diagram of ADT7410 Protocol Exerciser Application

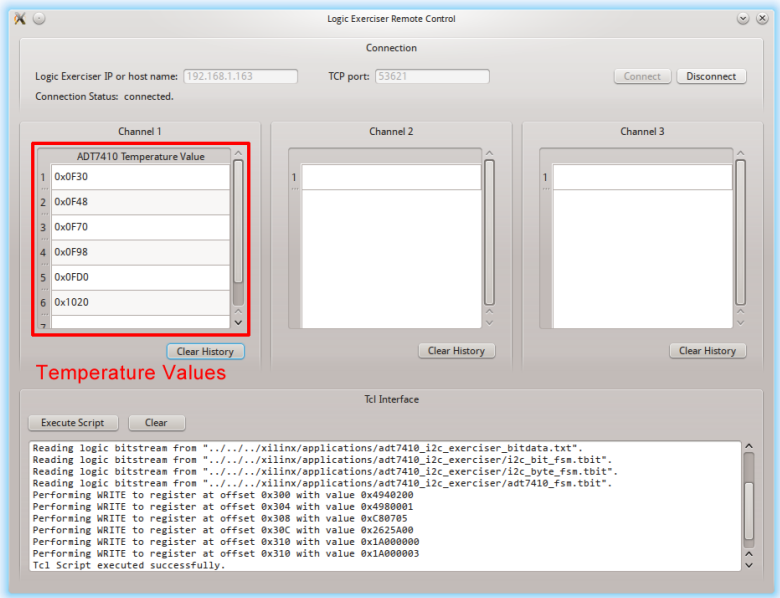


Figure 5.9: GUI Logic Exerciser Control Application

mask out the unneeded information and route the 16 bits that represent the raw temperature value to its physical output bits `15:0`.

The temperature value is then stored in a CPU-readable FIFO buffer for transmission to the host. Figure 5.9 shows the GUI on the host displaying the temperature values retrieved from the ADT7410 sensor while it is being warmed up with a heat gun. According to the ADT7410 data sheet [33], these values are in the 31 °C range $\left(\frac{\text{sensor value}>>3}{16}\right)$.

This application could easily be extended to raise an alarm when the temperature exceeds a certain value, like it was implemented in the MAX6682 Exerciser Application described in section 5.1.3.

5.1.6 Hardware Testbench

The DUT is implemented in the same FPGA as the Logic Exerciser. Therefore, the signals used to drive the seven segment display are connected to the reconfigurable logic module internally. However, to verify the correct behavior of the application, these signals are also routed to the GPIO port.

The structure of the Hardware Testbench example application is illustrated in figure 5.10. The following cells are used by this application:

- Two parameter registers,
- One TR-FSM,
- One 16-bit counter,
- One Content Addressable Memory,
- Two aggregator registers,
- One 8-bit multiplexer,
- One Arithmetic Logic Unit,
- One CPU-readable FIFO buffer.

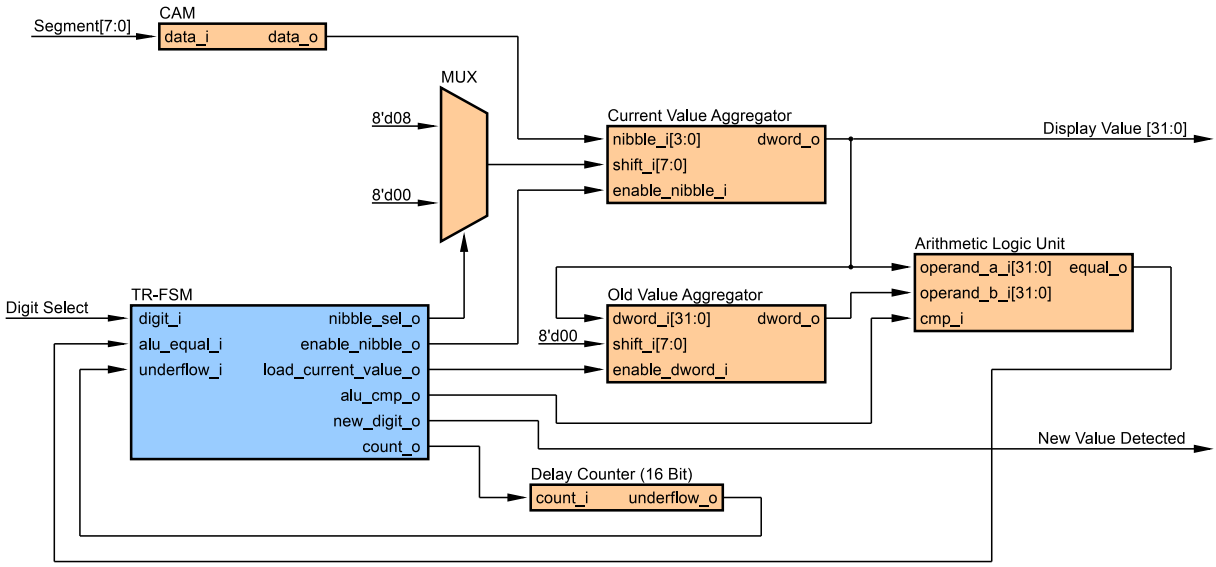


Figure 5.10: Block diagram of Hardware Testbench Example Application

The FSM waits for the first digit to be enabled by the display controller. It then starts the 16-bit counter and waits for it to generate an underflow. This accounts for the delay between the edge of the `digit_o` signal and the correct value being output via the `segment_o` signal as shown in figure 3.5(b).

The segment value is fed into the content addressable memory which is configured to output the binary value of the number represented by the segment value. The output of the content addressable memory is fed into an aggregator register and stored at the correct position – either at bits [3:0] or [7:4], depending on which digit was examined.

This process is repeated for the second digit.

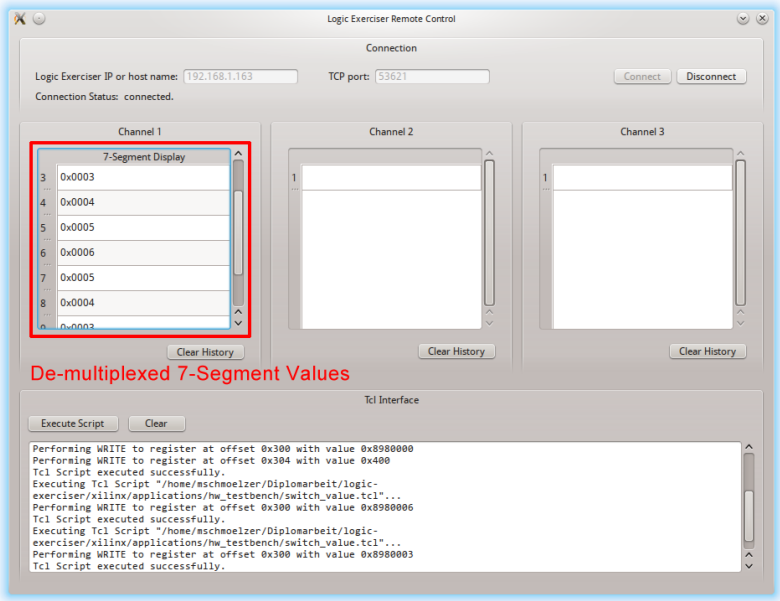


Figure 5.11: GUI Logic Exerciser Control Application

After the second digit has been processed, the FSM uses the Arithmetic Logic Unit to compare the binary representation of the digits to the previous value (also stored in an Aggregator register). If it differs, it is considered new and stored in a CPU-readable FIFO buffer for transmission to the host. If it is equal to the previous value, the number output by the DUT did not change and it is discarded.

Figure 5.11 illustrates the test vectors in the Logic Exerciser GUI.

5.2 Cell Usage

Table 5.1 shows the cell usage of each example application introduced in the previous sections. For applications which extend another application, the cumulative cell count of both applications is listed. From this table, we can draw the following conclusions:

- Every example application uses at least one TR-FSM. This observation is in-line with the basic model of our coarse-grain reconfigurable circuit, which has been established in section 2.3 as an extended version of the FSMD model.
- Every application uses at least one Timer/Counter cell (described in section 4.2.4.1), which are implemented in a very simple way (i.e. the cell is only able to count downwards from a pre-defined reload value until it reaches zero). It can be assumed that other applications exist for which the existing cell is insufficient. The development of an enhanced Timer/-Counter cell with additional functionality is therefore recommended.
- The CAM cell is used by only one application (the Hardware Testbench). This can be attributed to the fact that the Hardware Testbench is the only example application which implements large multi-vector decision trees.

			Example Applications					
Cell	**Count**	**Config**	**1**[a]	**2**[b]	**3**[c]	**4**[d]	**5**[e]	**6**[f]
"Small" TR-FSM	2	Yes	1	2	1	1	2	1
"Large" TR-FSM	1	Yes				1	1	
GPIO Splitter	1	Yes	1	1	1	1	1	1
Parameter Register	5	No	2	3	3	5	5	1
Aggregator	2	No	1	2				2
Shift Register	2	Yes	2	2	1	1	1	
CAM	1	Yes						1
FIFO	2	No	1	2	1	1	1	1
D-Flip-Flop	2	No			1	2	2	
ALU	1	No		1				1
Port Converter	2	Yes		2			1	
8 Bit Counter	2	No	1	1	1	2	2	
16 Bit Counter	2	No			1	1	1	1
32 Bit Counter	1	No			1		1	
8 Bit Multiplexer	3	No	2	3				1
32 Bit Multiplexer	1	No					1	
"Constant" Cell	1	No	1	1	1	1	1	1

[a]SPI Sniffer
[b]ADT7310 Protocol Analyzer
[c]SPI Master/MAX6682 Exerciser
[d]I^2C Master
[e]Protocol Exerciser for ADT7410 Sensor
[f]Hardware Testbench

Table 5.1: Cell usage for all implemented example applications

- Four of the six implemented applications employ at least one multiplexer cell. These multiplexers are used to switch between two or more data sources for one port of a specific connection type. Therefore, these cells can be seen as an extension to the reconfigurable interconnect generated by InterSynth. This functionality could be integrated into the generated interconnect by enhancing certain output ports with a method to change their source port during run-time.

- All example applications use constants provided by the "Constant" cell, which contains ports for each connection type that carry a constant all-one or all-zeroes signal vector. These constant signals are potentially required at every leaf node of the generated interconnect, which means that they have to be routed through large portions or even the entire interconnect tree. A more resource-friendly solution would be to locally generate these constants in the leaf switches.

5.3 Resource Usage

Tables 5.2 and 5.3 show the Look-Up Table (LUT) and Flip-Flop (FF) usage of the reconfigurable logic module with the interconnect implemented in both variants that are supported by InterSynth: the tree structure and the fully-multiplexed switch structure [WGS+12]. These numbers were obtained by synthesizing the design for the Xilinx Zynq-7000 XC7Z020CLG484-1 FPGA device with the Xilinx Vivado 2014.1 design tools [60].

Although these results were gathered from an FPGA implementation, it is assumed that the ratio between interconnect and cell usage (measured in LUTs and FFs) is roughly comparable to ASIC chip area specifications.

Cell	**LUTs**	**FFs**	**Count**	**Total LUTs**	**Total FFs**	**LUT %**	**FF %**
"Small" TR-FSM	920	784	2	1840	1568	15.33 %	23.51 %
"Large" TR-FSM	1914	1837	1	1914	1837	15.95 %	27.54 %
GPIO Splitter	1724	0	1	1724	480	14.37 %	7.20 %
Port Converter	422	32	2	844	462	7.03 %	6.93 %
Parameter Register	1	33	5	5	165	0.04 %	2.47 %
Shift Register	109	33	2	218	66	1.82 %	0.99 %
Aggregator	266	32	2	532	64	4.43 %	0.96 %
FIFO	159	339	2	318	678	2.65 %	10.16 %
CAM	219	4	1	219	324	1.83 %	4.86 %
D-Flip-Flop	2	1	2	4	2	0.03 %	0.03 %
ALU	14	1	1	14	1	0.12 %	0.01 %
8 Bit Counter	18	17	2	54	34	0.30 %	0.51 %
16 Bit Counter	38	33	2	76	66	0.63 %	0.99 %
32 Bit Counter	74	65	1	74	65	0.62 %	0.97 %
Cells Total			**26**	**7818**	**5812**	**65.16 %**	**87.14 %**
InterSynth (Tree)				4125	858	34.84 %	12.86 %
8 Bit Multiplexer	8	0	3	24	0	0.20 %	0.00 %
32 Bit Multiplexer	32	0	1	32	0	0.27 %	0.00 %
Interconnect Total (Tree)				**4181**	**858**	**35.31 %**	**12.86 %**
Total				**11999**	**6670**	**100.00 %**	**100.00 %**

Table 5.2: LUT and FF usage of the cells and the tree-structured Interconnect

It can be clearly seen that the tree-structured interconnect has a much more efficient area usage than the fully-multiplexed interconnect, requiring less than half the LUTs, but roughly the same number of FFs. Both are more efficient than bit-level FPGA interconnect implementations, which often use up to 90 % of the die area [DeH96].

Cell	LUTs	FFs	Count	Total LUTs	Total FFs	LUT %	FF %
"Small" TR-FSM	920	784	2	1840	1568	10.46 %	23.22 %
"Large" TR-FSM	1914	1837	1	1914	1837	10.89 %	27.20 %
GPIO Splitter	1724	0	1	1724	480	9.80 %	7.11 %
Port Converter	422	32	2	844	462	4.80 %	6.84 %
Parameter Register	1	33	5	5	165	0.03 %	2.44 %
Shift Register	109	33	2	218	66	1.24 %	0.98 %
Aggregator	266	32	2	532	64	3.03 %	0.95 %
FIFO	159	339	2	318	678	1.81 %	10.04 %
CAM	219	4	1	219	324	1.25 %	4.80 %
D-Flip-Flop	2	1	2	4	2	0.02 %	0.03 %
ALU	14	1	1	14	1	0.08 %	0.01 %
8 Bit Counter	18	17	2	54	34	0.20 %	0.50 %
16 Bit Counter	38	33	2	76	66	0.43 %	0.98 %
32 Bit Counter	74	65	1	74	65	0.42 %	0.96 %
Cells Total			**26**	**7818**	**5812**	**44.46 %**	**86.05 %**
InterSynth (Fully Multiplexed)				9709	942	55.22 %	13.95 %
8 Bit Multiplexer	8	0	3	24	0	0.20 %	0.00 %
32 Bit Multiplexer	32	0	1	32	0	0.27 %	0.00 %
Interconnect Total (Fully Multiplexed)				**9765**	**942**	**55.54 %**	**13.95 %**
Total				**17583**	**6754**	**100.00 %**	**100.00 %**

Table 5.3: LUT and FF usage of the cells and the fully-multiplexed Interconnect

To compare the coarse-grain reconfigurable logic circuit to an FPGA implementation, all example applications described in section 5.1 were implemented in the same Xilinx Zynq-7000 XC7Z020CLG484-1 FPGA device that was used for the implementation of the reconfigurable logic module. However, it is not sufficient to simply compare the LUTs and FFs used by the reconfigurable logic module and the discrete FPGA implementation, as the FPGA synthesis tool generates just the amount of logic required for the implementation of the intended behavior of the design. This cannot be done for the coarse-grain reconfigurable circuit, where all cells have fixed functions (e.g. consider a design which at some point performs an addition: during FPGA synthesis, only an adder will be generated while the coarse-grain reconfigurable logic implements this with the help of a much larger ALU cell). For a thorough comparison, the effort for configuration storage must also be considered.

According to the Xilinx Zynq Product Table [61], the FPGA used in the comparison has 53,200 LUTs, 106,400 FFs, 140 Block RAMs containing 36 kBits each and 220 Digital Signal Processing (DSP) elements. The bitstream used to configure the whole device has a length of 32,364,512

bits. Neither the Block RAMs nor the DSP elements are used in the design, so their part of the FPGA bitstream has to be determined and left out in the comparison. Unfortunately, FPGA manufacturers usually do not provide the necessary information, so an estimation has to be made:

- No information is available on the portion of the bitstream which is responsible for the configuration of the DSP elements. Therefore, 25 % of the FPGA bitstream are deducted to account for the unknown number of configuration bits of the DSP elements and the related portions of the interconnect.
- The Block RAM contents can be initialized by the FPGA bitstream, which means that at least 36,868 bits of configuration data is used for one Block RAM. Additional bits are used for the configuration of the FPGA's fine-grain interconnect which connects to the Block RAM ports. The number of Block RAM bits is multiplied by 1.2 to account for the corresponding bitstream size; this number is then also subtracted from the bitstream length.

For the given device, this leaves us with an estimated bitstream size of 18,080,232 bits (approximately 56 % of the total bitstream) which configure the LUTs and FFs in the device. This number is believed to favor the FPGA in the following comparison. The number of configuration bits used in a particular application is calculated by determining the ratio between the resources used by the application and the number of available resources in the FPGA.

Each application, when implemented by the coarse-grain reconfigurable logic module, requires 4154 bits of configuration data. In table 5.4, we can see that the larger applications require up to $\frac{204252}{4154} = 49.17$ times more configuration bits when implemented in the FPGA instead of the coarse-grain reconfigurable logic circuit.

Application	**LUTs**	**FFs**	**Config Bits**	**Config Factor**
SPI Sniffer (5.1.1)	337	448	114531	27.57
ADT7310 Protocol Analyzer (5.1.2) (includes SPI Sniffer)	601	851	204252	49.17
SPI Master (5.1.3)	307	542	104335	25.12
I^2C Master (5.1.4)	199	173	67631	16.28
ADT7410 Protocol Exerciser (5.1.5) (includes I^2C Master)	417	522	141719	34.12
Hardware Testbench (5.1.6)	308	453	104675	25.20
Total	**1633**	**2368**	**554981**	**133.60**
Reconfigurable Logic Module (Tree)	**11999**	**6670**	**4154**	

Table 5.4: LUT and FF usage of the example applications implemented in the FPGA

Now, to compare these numbers to an ASIC, we will take a look at the chip area required for the configuration storage. FPGAs typically use SRAM memory to store their configuration data. In the reconfigurable logic module, D-Flip-Flops, which require more chip area, are used. In an

exemplary ASIC manufacturing process[1], the ratio between the footprints of a D-Flip-Flop and an area-optimized SRAM bit is approximately 9.4. This means that in this process, the area used for the 6670 D-Flip-Flops (which includes storage of configuration **and** the FFs in the cells) present in the reconfigurable logic module is equivalent to 62,698 FPGA configuration bits. The smallest FPGA implementation of an example application (the I^2C Master) uses approximately the same chip area for configuration storage as the entire set of D-Flip-Flops contained in the reconfigurable logic module. The chip area consumed by the FPGA configuration for the larger example applications (e.g. the ADT7310 Protocol Analyzer or the ADT7410 Protocol Exerciser) uses up to three times as much chip area as the D-Flip-Flops of the reconfigurable logic module.

For a complete analysis, the reconfigurable logic module would have to be implemented in the exemplary ASIC manufacturing process, which was out of scope for this master's thesis, but is recommended for future work on the topic of coarse-grain reconfigurable logic circuits.

[1] An industrial grade 0.35 μm CMOS standard cell process with 4 metal layers is used for this comparison.

6 Summary

In the course of this thesis, the design and implementation of a coarse-grain reconfigurable logic architecture is presented. The motivation behind the development of this circuit architecture is a conflict of goals in current digital circuits: ASICs deliver high performance, combined with good power efficiency but almost non-existent reconfiguration capabilities. Current reconfigurable digital logic devices, on the other hand, offer very good flexibility but – compared to ASICs – suffer from poor performance and high energy consumption.

However, for some applications, reconfigurability is a crucial feature and a conflict of goals arises if another requirement is either high performance or energy efficiency. Also, re-designing an ASIC is a very time-consuming process and the manufacturing process is very expensive. Coarse-grain reconfigurable logic aims to combine the advantages of ASICs with those of currently available reconfigurable logic devices. This is done by implementing larger functional units which are connected by optimized word-level interconnects.

The FSMD model, which was originally devised for the high-level synthesis of digital logic circuits, was established as the foundation of a coarse-grain reconfigurable logic architecture. This model represents an arbitrary digital circuit and consists of two basic components: the *control unit* and the *data path*. The control unit consists of one or more finite state machines which control various arithmetic and logical elements to perform the actual data processing. Three key extensions to the FSMD model have been identified to make it a suitable basis for a reconfigurable architecture:

- The finite state machines in the control unit must be replaced by dedicated reconfigurable finite state machines.
- The cells which comprise the data path must implement all data processing operations required for the given application domain.
- The connections between all cells (the finite state machines in the control unit and the cells in the data path) must be reconfigurable.

The *Logic Exerciser*, a device which is connected to a host PC and designed to analyze and generate digital signals, has been selected as a prototype case for the newly developed reconfigurable architecture. The analysis and generation of signals by the Logic Exerciser is realized by a coarse-grain reconfigurable logic module, while an embedded CPU is used to handle its configuration as well as the communication between the Logic Exerciser and the host PC.

An example-driven approach was conducted to develop the cells which comprise this module: several example applications were implemented as monolithic designs with the help of the Verilog hardware description language. Then, the functionality implemented by these modules was analyzed and the cells of the reconfigurable logic module were specified from the information learned in this step.

After the implementation of the cells, the example applications, which were chosen with focus on real-world usefulness, were re-implemented in the coarse-grain reconfigurable architecture:

- The **SPI Sniffer** is designed as a debug tool for devices communicating via a Serial Peripheral Interface (SPI). The application listens to the four wires which are used by the SPI and transfers the captured data to the host PC.
- As an extension to the SPI Sniffer, the **Protocol Analyzer** was developed. This application analyzes the data which is transmitted between a microcontroller and a smart digital temperature sensor and extracts the temperature values which have been measured by the sensor.
- An **SPI Master** was implemented by the coarse-grain reconfigurable circuit. It is capable of transmitting and receiving data to a standard SPI slave.
- An **I^2C Master** was designed. The reconfigurable logic circuit implements the bus access at the electrical level as well as the transmission and reception of arbitrary data.
- On top of the I^2C Master application, a **Protocol Exerciser** was developed. This circuit implements the sequence of commands which needs to be sent via the I^2C bus to continuously poll a smart digital sensor.
- The **Hardware Testbench** application is designed to apply test vectors to a Device Under Test (DUT) and check its outputs for correct execution. The main advantage to existing hardware test equipment is the fact that the test vectors are not simply pairs of input- and expected output signals, but instead an adaptive test algorithm can be implemented directly in hardware.

Two other crucial modules were designed: a flexible GPIO port which can be partially reconfigured by the coarse-grain reconfigurable logic and a configuration and parameterization interface which acts as a bridge between an embedded CPU and the reconfigurable logic module. The example applications show that the the coarse-grain reconfigurable logic developed in the course of this thesis is able to implement diverse and feature-complete real-world tasks.

The resulting reconfigurable logic module was compared to a discrete FPGA implementation of the example applications. This comparison shows a clear advantage in terms of chip area for the coarse-grain reconfigurable logic.

In addition to the design of the reconfigurable circuit, methods and suggestions for the design of coarse-grain reconfigurable logic circuits were developed.

6.1 Outlook

The wide variation of example applications implemented by the reconfigurable logic circuit already shows the high flexibility of the designed coarse-grain reconfigurable logic circuit. However, as in most complex projects, there is always room for improvement. In this section, suggestions for future enhancements which either improve the design flow for coarse-grain reconfigurable circuits or provide specific ideas for implementation details are provided.

6.1.1 Design Flow Automation

During the design of the cells which comprise the reconfigurable logic module as well as during the development of the example applications, several tasks which bear large potential for automation had to be done manually due to lack of existing tools. For the development of an ideal design flow with a maximum of user-friendliness, the automation of several processes would be desirable:

- Automatic generation of wrappers: During the pre-silicon design flow, InterSynth is used to process netlists containing example applications. Because InterSynth does not support the generation of Verilog modules with instantiation parameters, these had to be wrapped in another Verilog module which does not use instantiation parameters. Tool support for the generation of these wrappers is desirable.
- Automatic generation of InterSynth netlists: Currently, the example netlists used by InterSynth during the pre-silicon and post-silicon design flow phases need to be written by hand. However, it is much more comfortable to be able to write them in a standard hardware description language such as Verilog. Yosys has basic support for the generation of InterSynth netlists via its "`write_intersynth`" pass, but during development of the example applications, the output generated by this pass did not yield correct and complete netlists.
- More streamlined replacement of hand-coded FSMs with TR-FSMs: In the design flow described in this thesis, the FSMs that control the data path cells were first written in the Verilog HDL, then synthesized with the help of Yosys, exported in the KISS format and finally imported by trfsmgen to generate the configuration bitstream for the corresponding TR-FSM. Then, the FSM instantiation needs to be manually replaced by the TR-FSM. During development, changes to the Verilog FSM description would sometimes change the order of the FSM input and/or output signals in Yosys, requiring manual adjustments. Tool support for an integrated approach to synthesize the module containing instances of FSMs described in Verilog and the subsequent export of a new Verilog file with FSM instantiation replaced by the corresponding TR-FSM instantiations would greatly simplify the design of new applications.
- Coarse-grain synthesis: ultimately, the ability to synthesize arbitrary Verilog RTL code or even a high-level algorithmic description to a set of coarse-grain reconfigurable cells and TR-FSMs – as long as the available cells provide the required operations – would possibly be a breakthrough for coarse-grain reconfigurable logic circuits, eventually leading to a widespread adoption of the technology. Yosys already implements coarse-grain mapping, which is a key feature for a fully-fledged coarse-grain synthesis, in its "`extract`" pass [WG13].

6.1.2 Additions to the Tcl Interface

The Tcl interfaces of both the Logic Exerciser Control program introduced in section 4.4.1 as well as the Bitstream Generation program for data path cells introduced in section 4.4.2 both feature a Tcl interface:

Logic Exerciser Control Program: The Tcl interface of the Logic Exerciser control software is currently used only for automating the configuration and parameterization of the reconfigurable logic module (e.g. downloading the configuration bitstream and writing data to the parameter registers). The analyzed or generated data sent from the Logic Exerciser to the host is displayed by the GUI of the program, where it needs to be manually verified by the user. To automate this process, this data could be exposed to the Tcl interface, creating a very powerful automated test environment, where – in addition to the adaptive test algorithm implemented by the reconfigurable logic module – the full flexibility of the Tcl interpreter on the host system could be harnessed.

Bitstream Generation for Data Path Cells (bsgen): The Tcl interface of the bsgen program is used to execute commands which ultimately generate the bitstream for the CAM, Port Converter and GPIO Splitter cells. Currently, because only one bitstream can be generated during one invocation of the program, the process of generating bitstreams for multiple cells involves calling the bsgen program with different configuration files (one for each cell). However, the library which was used to implement the Tcl interface [41] has support for custom variable types. Therefore, new variable types representing the cells could be registered within the Tcl library, and a single cell bitstream generation script can be written for each example application.

6.1.3 Interrupt-Driven Retrieval of FIFO Data

As mentioned in section 3.2, the logic exerciser peripheral exposes several of its internal registers to the system's AMBA bus, thus allowing the system CPU to read and write these registers. Two conditions trigger accesses to these registers:

- A configuration or register write command is received over the TCP transmission.
- No commands are pending and the firmware is currently not handling any tasks related to the management of the TCP protocol. In this case, the FIFO status register is continuously polled, and if available, data from the corresponding FIFO cell is queued for transmission to the host PC.

The busy-waiting implementation of the second condition listed above is generally frowned upon, because it prevents the CPU from executing other tasks and the switch to a power saving mode. However, this solution provides the lowest possible response time between the storage of a new data value in a FIFO and the extraction of the value by the CPU. Also, there are no stringent power saving requirements that need to be adhered to by the Logic Exerciser. For applications that do have such requirements, an interrupt-driven approach can be easily added to the circuit, in which the FIFO cells generate an interrupt whenever a new value is added. This interrupt can then be used to wake up the system CPU from a low-power state and trigger the retrieval of the value stored in the FIFO.

Literature

[ABGS00] Ahmad Alsolaim, Jürgen Becker, Manfred Glesner, and Janusz Starzyk. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–214, Napa Valley, CA, USA, April 2000.

[AGW10] Jan Andersson, Jiri Gaisler, and Roland Weigland. Next generation multipurpose microprocessor. In *DASIA 2010 Data Systems in Aerospace*, volume 682 of *ESA Special Publication*, Noordwijk, Netherlands, August 2010.

[BGG03] Vicki Bruce, Patrick R. Green, and Mark A. Georgeson. *Visual Perception: Physiology, Psychology and Ecology.* Psychology Press, 27 Church Road, Hove, BN3 2FA, United Kingdom, 2003.

[CGMT09] Phillipe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, August 2009.

[Cil11] Michael D. Ciletti. *Advanced Digital Design with the Verilog HDL.* Prentice Hall, 2011.

[DeH96] André DeHon. Reconfigurable architectures for general-purpose computing. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, October 1996.

[EH91] Norman G. Einspruch and Jeffrey L. Hilbert. *Application Specific Integrated Circuit (ASIC) Technology.* Academic Press, 1991.

[GDHG10] Johann Glaser, Markus Damm, Jan Haase, and Christoph Grimm. A dedicated reconfigurable architecture for finite state machines. In *6th International Symposium, ARC 2010, Bangkok, Thailand. Proceedings*, pages 122–133, March 2010.

[GG04] Ian A. Glover and Peter A. Grant. *Digital Communications.* Prentice Hall, second edition edition, 2004.

[GGHG11] Johann Glaser, Klaus Gravogl, Jan Haase, and Christoph Grimm. A reconfigurable architecture for ultra-low power wireless sensors. *The Mediterranean Journal of Electronics and Communications*, 7(3):255–266, July 2011.

[GHDG09] Johann Glaser, Jan Haase, Markus Damm, and Christoph Grimm. Investigating

power-reduction for a reconfigurable sensor interface. In *17th Austrian Workshop on Microelectronics. Proceedings*, October 2009.

[Gio95] Jean-Charles Giomi. Finite state machine extraction from hardware description languages. In *Proceedings of the Eighth Annual IEEE International ASIC Conference and Exhibit*, Austin, TX, USA, September 1995.

[GK83] Daniel D. Gajski and Robert H. Kuhn. Guest editor's introduction – new vlsi tools. *IEEE Computer*, 16(12):11–14, December 1983.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, 1994.

[Har01] Rainer Hartenstein. A decade of reconfigurable computing: a visionary retrospective. *Design, Automation and Test in Europe. Conference and Exhibition. Proceedings.*, pages 642–649, 2001.

[HHW90] R.W. Hartenstein, A. G. Hirschbiel, and M. Weber. Xputers: An open family of non-von neumann architectures. In *Architektur von Rechensystemen, Tagungsband, 11. ITG/GI-Fachtagung*, München, Germany, 1990.

[LPCM07] Marco Lanuzza, Stefania Perri, Pasquale Corsonello, and Martin Margala. A new reconfigurable coarse-grain architecture for multimedia applications. In *Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 119–126, August 2007.

[MSK+99] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *ACM/SIGDA seventh international symposium on Field Programmable Gate Arrays*, New York, NY, USA, 1999.

[MTM01] Hisatada Miyatake, Masahiro Tanaka, and Yotaro Mori. A design for high-speed low-power cmos fully parallel content-addressable memory macros. *IEEE Journal of Solid-State Circuits*, 36(6):956–968, June 2001.

[MV07] Graeme Milligan and Wim Vanderbauwhede. Implementation of finite state machines on a reconfigurable device. In *Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 386–396, Edinburgh, Scotland, August 2007.

[Raz01] Behzad Razavi. *Design of Analog CMOS Integrated Circuits.* McGraw-Hill, international edition 2001 edition, 2001.

[Sax09] Arjun N. Saxena. *Invention of Integrated Circuits – Untold Important Facts.* World Scientific Publishing Co. Pte. Ltd., 2009.

[SML09] Mihai Sima, Michael McGuire, and Julien Lamoureux. Coarse-grain reconfigurable architectures – taxonomy. *Communications, Computers and Signal Processing*, pages 975–978, August 2009.

[Soc06] IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language.* The Institute of Electrical and Electronics Engineers, Inc., 2006.

[Tex73] Texas Instruments. *MOS Programmable Logic Arrays. Application Bulletin CA.158.*, 1973.

[Uni92] University of California, Berkeley. *Berkeley Logic Interchange Format (BLIF)*, July 1992.

[Wak01] John F. Wakerly. *Digital Design: Principles and Practices.* Prentice Hall, 3rd edition edition, 2001.

[WG13] Clifford Wolf and Johann Glaser. Yosys - a free verilog synthesis suite. In *Proceedings of Austrochip 2013*, 2013.

[WGS+12] Clifford Wolf, Johann Glaser, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-driven interconnect synthesis for heterogenous coarse-grain reconfigurable logic. *Forum on Specification and Design Languages (FDL)*, pages 194–201, September 2012.

Internet References

[30] *Olimex Universal EXTension connector (UEXT)*. Available at: https://www.olimex.com/Products/Modules/UEXT/resources/UEXT_rev_B.pdf.

[31] Altera Nios II Processor: The World's Most Versatile Embedded Processor. Available at: http://www.altera.com/devices/processor/nios2/ni2-index.html.

[32] Analog Devices Inc. *ADT7310 Data Sheet*, rev. a edition, 2011.

[33] Analog Devices Inc. *ADT7410 Data Sheet*, rev. a edition, 2011.

[34] ARM Ltd. *AMBA AXI and ACE Protocol Specification*, issue e edition, February 2013.

[35] Avnet Electronics Marketing. *ZedBoard Hardware User's Guide*, v1.6 edition, January 2014.

[36] CMake, the cross-platform, open-source build system. Available at: http://cmake.org.

[37] Design Files for the SmartFusion Breakout Board. Available at: https://github.com/mschmoelzer/smartfusion-breakout-board.

[38] Intel Xeon Processor E7 and E3-1200 Families. Available at: http://newsroom.intel.com/docs/DOC-1963.

[39] lwIP - A Lightweight TCP/IP stack - Summary. Available at: http://savannah.nongnu.org/projects/lwip.

[40] lwIP Wiki: Porting For Bare Metal. Available at: http://lwip.wikia.com/wiki/Porting_For_Bare_Metal.

[41] Main Tcl/Tk 8.6 Info Page. Available at: http://www.tcl.tk/software/tcltk/8.6.html.

[42] Maxim Integrated Products. *MAX6682 Data Sheet*, rev. 0 edition, 2002.

[43] Menta S.A.S.: eFPGA Core IP Product Brief. Available at: http://www.menta.fr/down/ProductBrief_eFPGA_Core.pdf.

[44] Microsemi Corporation. *SmartFusion Evaluation Kit User's Guide*, rev. 7 edition, 2012.

[45] Microsemi Corporation. *SmartFusion System-on-Chip (SoC) FPGA Datasheet*, rev. 12 edition, October 2013.

[46] Motorola Inc. *SPI Block Guide*, February 2003.

[47] NXP Semiconductors. *I^2C-bus specification and user manual*, rev. 5 edition, 10 2012.

[48] Olimex Ltd. Available at: https://www.olimex.com/Products/ARM/ST/STM32-P103/.

[49] OpenCores: openmsp430 Overview. Available at: http://opencores.org/project,openmsp430.

[50] OpenCores: OpenRISC Overview. Available at: http://opencores.org/or1k/Main_Page.

[51] Press Release: Infineon Succeeds in Producing Chips on New 300-Millimeter Thin Wafer Technology for Power Semiconductors. Available at: http://www.infineon.com/cms/en/corporate/press/news/releases/2011/INFXX201110-002.html.

[52] Qt Project. Available at: http://qt-project.org.

[53] Texas Instruments Incorporated. *8-Bit Bidirectional Voltage-Level Translator With Auto-Direction Sensing and ±15 kV ESD Protection*, rev. e edition, May 2012.

[54] C. Wolf. Intersynth user manual. April 2012. Available at: http://www.clifford.at/intersynth/intersynth_manual.pdf.

[55] C. Wolf. Yosys manual. December 2013. Available at: http://www.clifford.at/yosys/files/yosys_manual.pdf.

[56] Xilinx Inc. *7 Series FPGAs Configurable Logic Block*, v1.5 edition, August 2013.

[57] Xilinx Inc. *Zynq-7000 All Programmable SoC Overview*, v1.6 edition, December 2013.

[58] Xilinx Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual*, v1.7 edition, February 2014.

[59] Xilinx LogiCORE IP MicorBlaze Micro Controller System (v1.1). Available at: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ds865_microblaze_mcs.pdf.

[60] Xilinx Vivado 2014.1 Design Tools. Available at: http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2014-1.html.

[61] Xilinx Zynq-7000 Product Table. Available at: http://www.xilinx.com/publications/prod_mktg/zynq7000/Zynq-7000-combined-product-table.pdf.