FPGA Firmware Qualification Framework
using AXI interconnect and extended debug facilities

Lemmen, L.

Award date:
2012

TU/e Eindhoven University of Technology

Electronic Systems Group
Department of Electrical Engineering

Master's Thesis

FPGA Firmware Qualification Framework

Using AXI Interconnect and Extended Debug Facilities

By
L. Lemmen

Supervisors
Prof. dr. H. Corporaal
dr. R.H. Mak
Ir. B. v.d. Ven
Ing. S. van den Hoek

Internal revision: 218
March 25, 2012

A Framework for Reducing Qualification and Debug Time

Son March 25, 2012

Course	: 5T514 - Graduation Project
Master	: Embedded Systems
Department	: Electrical Engineering - Eindhoven University of Technology
Chair	: Electronic Systems - Eindhoven University of Technology

Supervisors : *Eindhoven University of Technology*
Prof. dr. H. Corporaal
dr. R.H. Mak

Prodrive B.V.
Ir. B. van de Ven
S. van den Hoek

Author : L. Lemmen
Gebr. Ganslaan 16
5626 GB, Eindhoven

Prodrive B.V.
P.O. Box 28030
Science Park 5501
5692 EM Son
The Netherlands
☎ +31 (0)40 2676 200
🖷 +31 (0)40 2676 201
🖱 http://www.prodrive.nl
✉ info@prodrive.nl

Eindhoven University of Technology
P.O. Box 513
Den Dolech 2
5612 AZ Eindhoven
The Netherlands
☎ +31 (0)40 2479 111
🖷 +31 (0)40 2441 692
🖱 http://www.tue.nl
✉ info@tue.nl

Abstract

FPGA devices are flexible hardware devices. FPGAs are used in the area where focus lies on short design times and small production series. The observability and controllability of these configurable devices is limited. Therefore, it is difficult to locate and debug problems in FPGA firmware. It is desirable to have a debugging methodology together with tooling that provides a better debugging solution for FPGA firmware. Observability in a running FPGA design enables the inspection of internal signal values. Controllability allows to pause and continue the execution of parts of a design.

The observability can be improved by routing internal signals to "spare" I/O pins. Furthermore observability can be improved by instantiating monitor IP[1] *cores. These methods for increased observability are time and resource intensive.*

This research introduces the ***F****PGA* ***F****irmware* ***Q****ualification* ***F****ramework (FFQF) methodology. This methodology is supported by a template design and embedded into a framework with some interface tooling. The template consists of a library with standard interconnection and debug components. The FFQF methodology allows a communication centric monitoring approach that increases observability. The introduction of breakpoints and trace buffers enables off-line analysation of sequences. These sequences can lead to the source of to a problem situation. The breakpoints enable controllability while tracing increases observability. If the firmware is stopped at a breakpoint then it is possible to read-back the FPGA hardware state e.g. flip-flop and blockram contents. This enables observability at hardware register level.*

The expected overhead and impact of the system is determined using a model. The model is verified with a case-study that also demonstrates the usage of the framework.

The area requirements of interconnection system depend on its configuration. The configuration used in the case-study consists of two slaves and a single controlling master. This configuration requires less than 0.1% of the resources of a Virtex-6 LX240T FPGA. The system scales linear with the number of slaves for all resources except slices, which scales quadratic. In the case study project the interconnection system increases the area usage with approximately 20%.

[1] Intellectual Property

Preface

This research is conducted at Prodrive B.V. Son. Prodrive B.V. is a solution provider in electronics design, manufacturing and delivers added value services for Original Equipment Manufactures (OEMs) and Original Design Manufactures (ODMs) operating in industrial, professional and consumer markets.

FPGAs play an important role in the wide range of products designed by Prodrive B.V. The market demands shorter development time of more complex designs. This requires extensive qualification in limited time. Issues found during qualification need to be resolved as quick as possible. The nature of FPGAs is that their controllability and observability is limited and difficult to achieve. Therefore, the qualification and debugging process are time consuming. This is the reason for Prodrive B.V. to initiated this research project. The case-study design is a electrical current controller designed and produced by Prodrive B.V. The developed methodology is applicable to a wider range of products than only this controller.

Contents

Chapter 1

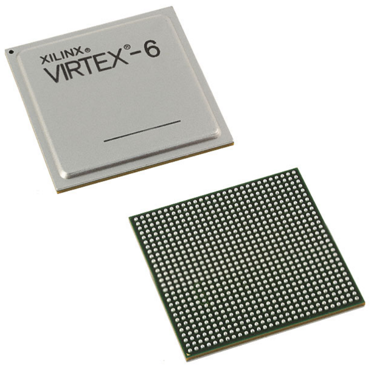
Introduction

In the industrial market numerous electronics manufacturers design products on customer request. The products or solutions designed by these manufactures are, depending on the company, mostly components which are integrated in a larger system by customers. The designs range from a single Print Circuit Board (PCB) to multiple PCBs integrated in casing running complex applications. The software or firmware running in this hardware can be of different types. The first type is software running on a single or multi-core processor as for example a Digital Signal Process (DSP) or Reduced Instruction Set Computer (RISC). This research does not focus on these type of processors. The second type of software is the firmware running in Field Programmable Gate Arrays (FPGAs). These FPGAs are reprogrammable hardware components. The trade-off between FPGAs and Processors can be made based on cost of goods and the application requirements. If for example $300+$ Input-Output pins are required in combination with 20 fully configurable high-performance parallel tasks running at $200\mathrm{MHz}$ then an FPGA is a good solution. If a sequential `C++` application using multiple standard interfaces as `IIC`, `UART` and `Timers` is required to run, then a DSP processor is a proper platform.

The parallel nature of FPGA devices makes them ideal for running multiple tasks concurrently. When FPGA designs get more complex by the increasing number of parallel tasks working together, it also gets more difficult to understand these entire designs. The designs for these FPGAs are often written in the *VHSIC*[1] *Hardware Description Language* (VHDL). This language allows using the parallel nature of the device. Figure 1.1 shows a photo of a Virtex-6 FPGA which is used during this research with some of its features. Figure 1.2 shows a power amplifier designed by Prodrive B.V. using an FPGA as controlling processor. A similar power amplifier is used for the case-study of this research.

The design cycle of the firmware running on the FPGA consists of different steps. The steps of this process are shown in Figure 1.3 and are according to the V-Model [2] design flow. In the *requirement analysis* phase the customer specifications are written in a specification document. These specifications are used as input for the *architectural* and *module design*. The architectural design describes

[1] Very-High-Speed Integrated Circuits



Component	Value	Value
Slices	37,680	-
CLB flip-flops	301,440	-
Blockram memory	14,976	Kb
DSP Slices	768	-
I/O	730	pins

Figure 1.1: Xilinx Virtex-6 FPGA (LX240T), used during this research. The CLBs are Configurable Logic Blocks which are elementary FPGA building blocks. The blocks consist of two slices with each four 6-input LUTs and 8 flip-flops [1].

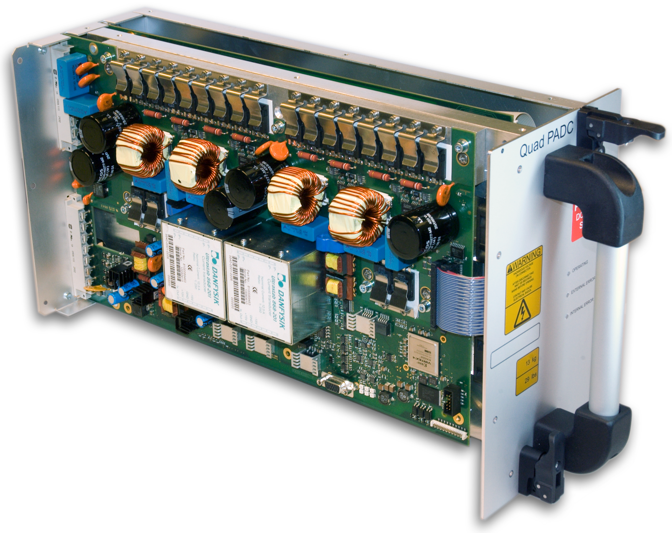


Figure 1.2: Four axis high linearity power amplifier with FPGA firmware controlling electrical current.

the top-level while the module design specifies all subsystems in detail. All this design information is written in the design document. This concludes the design phase. The design information is then used as input when *writing VHDL code*. Based on the information in the design document and specification document a qualification document is written. The qualification document should contain all test cases and expected results required to ensure the system behaves according to its specification. The subsystems or modules are qualified during the *unit tests* using a Register Transfer Level (RTL) behavioural simulation. The top-level is qualified in the *integration test*, first with behavioural RTL simulation, and thereafter running it on the actual hardware with less observability. In the *acceptance test* the entire system is tested according to the specification.

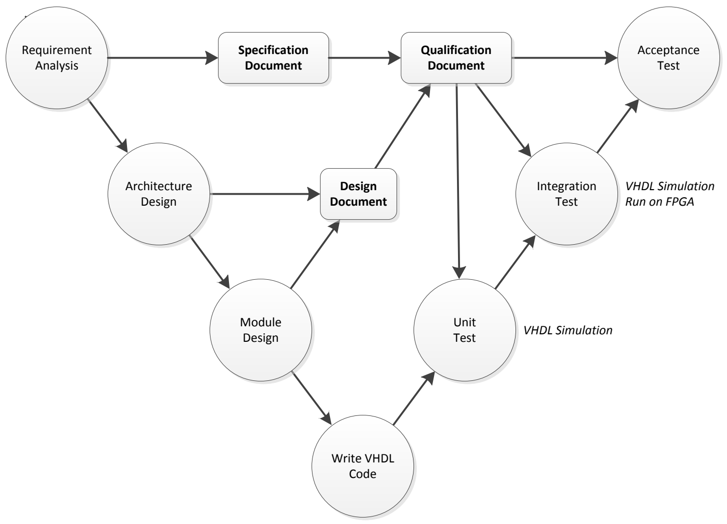


Figure 1.3: Design flow according to the V-Model, every abstraction level in the design includes a qualification step at the same level. The blocks denote the documents which contain the specifications, design descriptions and qualification results.

Since the VHDL simulations at RTL level are behavioural, they do not cover the exact hardware behaviour [3]. During the behavioural RTL simulation every single register value can be monitored at every clock cycle. Most issues that exist in the design can be found during simulation. More information on the simulation of VHDL code can be found in Section 2.3.

After simulation the complete VHDL code is synthesised into a netlist file describing all FPGA required logic blocks. The synthesis output is used as input for the implementation process. This process performs a technology mapping where the netlist is mapped to the FPGA logic. Thereafter, the process determines the location of the components and adds routing logic. The last step is to write all this information into an FPGA configuration binary file. The time consumed by these individual process steps depends on the size of the FPGA design. This process is explained in more detail in Section 2.2.

The FPGA binary file is loaded into the FPGA device for testing. At this point only the input and output of the FPGA can be used for diagnosis. In order to monitor inputs and outputs an oscilloscope or logic analyzer needs to be connected to the physical pins on the device. The process that runs in the FPGA cannot be monitored internally without modifications. If problems arise internal logic analyzers can be embedded in the design. These analyzers trigger on pre-defined conditions and store signal traces starting from the moment the trigger condition occurs. Adding and modifying the logic analyzer requires rerunning the map and placement steps. When the design is changed in order to correct the mistake, the entire process for generating a binary must be repeated before testing is possible.

The basic simulation process is easy to setup. Due to a lot of observability during simulations it tends to get slow. The qualification step of running the firmware on the FPGA is more difficult. There is limited observability and therefore possible error causes are difficult to locate. It is desirable to introduce a solution to improve the qualification of running firmware by increasing controllability and observability. The motivation and problem statement is described in more detail in section 1.1. The contributions of this research are given in section 1.2 and the remainder of this thesis is outlined in section 1.3.

1.1 Motivation and problem statement

In the industry there are designs that use an FPGA from a software perspective. In the case of Prodrive B.V. a selection of products are industrial motion and electrical current controllers. These controllers are either fitted with only an FPGA or are a hybrid platform with also a DSP or RISC processor. The firmware running in the FPGA is responsible for completing a certain function. Based on received input the FPGA generates a defined output. In case of an electrical current controller a desired electrical current set-point is configured. Based on the actual electrical current measured via the sensors an output is generated for an actuator e.g. a step count or a Pulse Width Modulated (PWM) signal. This PWM signal is fed into an amplifier controlling for example a motor. The responsibility of the manufacturer is to ensure that for all specified situations the correct output is generated. These specified situations are for instance the bandwidth requirements of the controller. The output value must be reached within $125\mu\mathrm{s}$ after setting the desired value. Furthermore, the error may not be larger than 0.5% in a stable situation. The research applies to more than only these electrical current controllers; this is only used as an example.

The key task of a design as shown in Figure 1.4 is to control the electric current in a system to a desired set-point. In order to quickly respond to changes in the environment the controller must quickly generate output based on measured input. The loop rates of systems range for instance between $150-500\mathrm{kHz}$. In order not to destroy the product in error situations constant monitoring is required. When the electrical current is above a limit value for more than for instance $1\mu\mathrm{s}$ or even $1\mathrm{ns}$ this could burn the tracks on the PCB or burn electrical components. Besides controlling and monitoring the system periodically receives new configuration parameters via a communication interface. All these processes must run in parallel at high clock speeds on a low-cost device with a lot of I/O ports. Therefore an FPGA is chosen instead of a multi-core application processor.

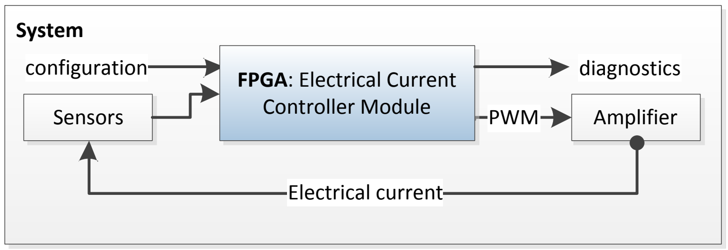


Figure 1.4: Overview of an example system, the electrical current controller is a component in the design.

The complex designs that exist in these FPGAs are build up from multiple smaller parallel building blocks as shown in Figure 1.5. These building blocks are referred to as subsystems. The subsystems are hierarchically built from process groups. Process groups are a collection of individual processes. The subsystems inside the FPGA are tightly coupled in existing designs. This coupling means that processes are connected directly via parallel registers of arbitrary width. These hierarchical levels can be chosen arbitrarily. More information on the determination of subsystems and processes is shown in Section 4.1.

There are two types of connections in every FPGA design, namely the internal connections between processes, process groups or subsystems, and the external connections between the FPGA and external hardware. The internal and external connections have different constraints, but are handled equally in VHDL.

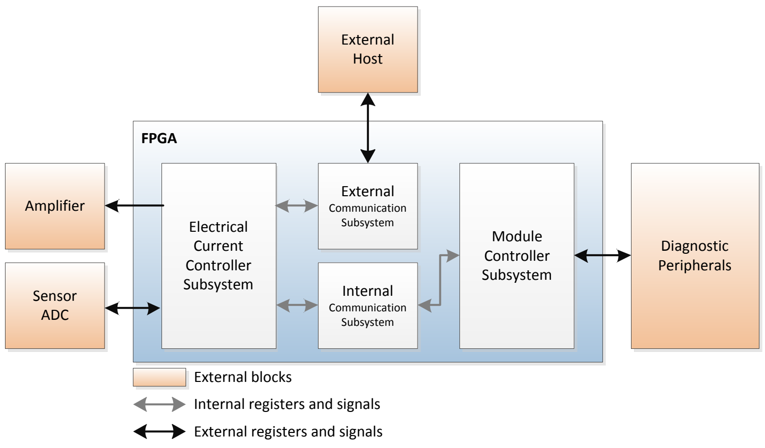
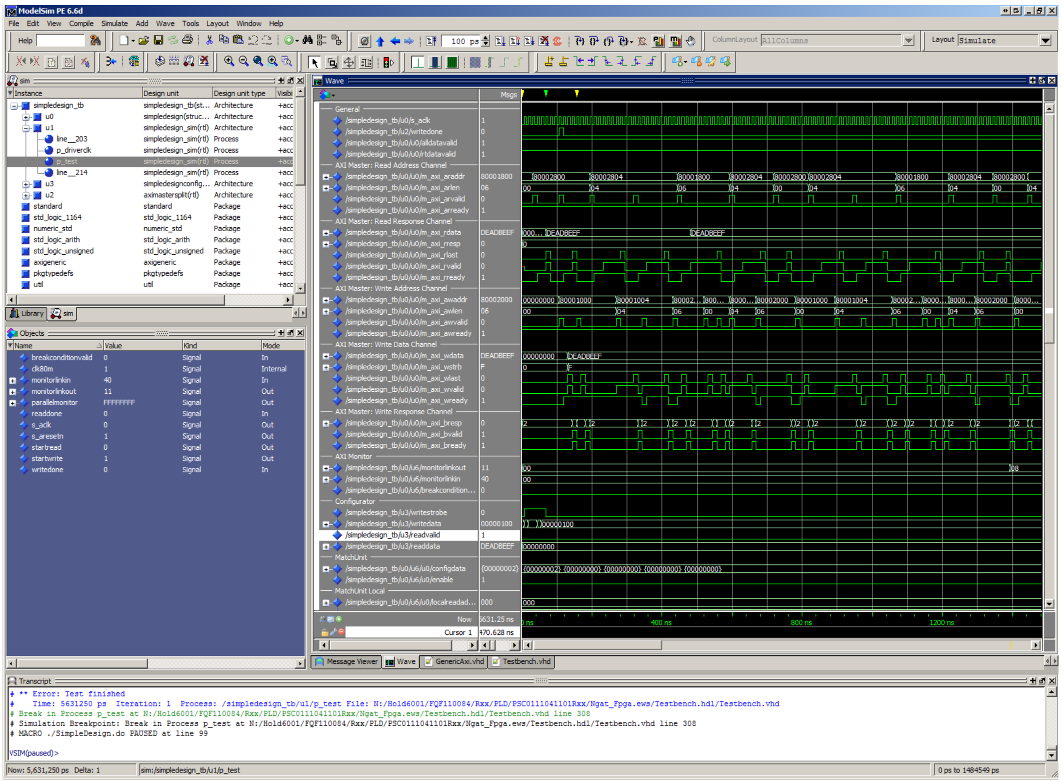

External Host
FPGA
Amplifier
Sensor ADC
Electrical Current Controller Subsystem
External Communication Subsystem
Internal Communication Subsystem
Module Controller Subsystem
Diagnostic Peripherals
External blocks
Internal registers and signals
External registers and signals

Figure 1.5: Overview of an example system, the electrical current controller is a component in the design

The individual subsystems are qualified before they are used in the final design. The qualification determines whether the subsystem behaves as expected. There are different methods used for qualification of FPGA firmware. The first method for qualification is simulation in special designed software as Modelsim [4]. This simulator takes both the source code under test and stimulus source-code as input; this stimulus source-code is referred to as the test-bench. The output of Modelsim is a time graph with all internal signals which can be used to determine the behaviour of the design. This simulation enables extended observability as shown in Figure 1.6. A detailed description of this debug process is shown in chapter 2.

The simulation of large designs with around $50,000+$ lines of code take *hours* in the order of magnitude to simulate *milliseconds* of execution time. For this reason usually only small subsystems are simulated. The top-level design which glues all subsystems together is often impractical to simulate because of the long simulation time. Due to these limited simulation possibilities most problems arise when gluing all subsystems and processes together. It takes even more time to simulate a time correct model of the FPGA firmware and therefore this is not used within Prodrive B.V.

Besides simulation another method is also used for qualification. This method is to program the FPGA with the generated firmware and apply different test (vector) input, while monitoring the output.



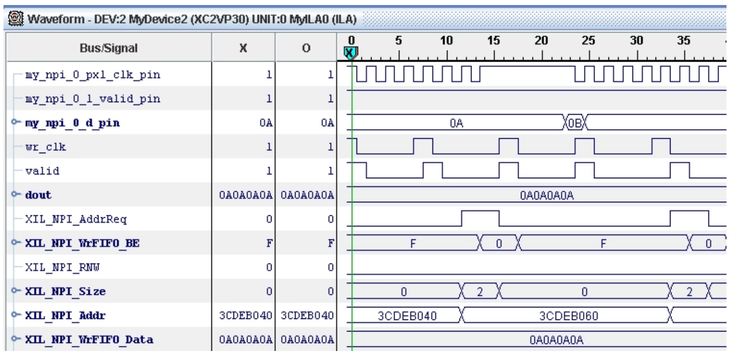
```
# ** Error: Test finished
#    Time: 5631250 ps  Iteration: 1  Process: /simpledesign_tb/u1/p_test File: N:/Hold6001/FQF110084/Rxx/PLD/PSC0111041101Rxx/Ngat_Fpga.ews/Testbench.hdl/Testbench.vhd
# Break in Process p_test at N:/Hold6001/FQF110084/Rxx/PLD/PSC0111041101Rxx/Ngat_Fpga.ews/Testbench.hdl/Testbench.vhd line 308
# Simulation Breakpoint: Break in Process p_test at N:/Hold6001/FQF110084/Rxx/PLD/PSC0111041101Rxx/Ngat_Fpga.ews/Testbench.hdl/Testbench.vhd line 308
# MACRO ./SimpleDesign.do PAUSED at line 99

VSIM(paused)>
```

Figure 1.6: Modelsim showing the signals including the clock when simulating a design for correctness with full observability regarding internal signals.

There are different problems when validating and debugging such complex FPGA firmware. An FPGA is a closed system with many completely configurable subsystems running in parallel. The final system can only be validated by triggering the system with a known input at physical input pins. The value at the output pins is then compared with the expected value. If this information does not match then there is no default method to extract more internal information from the FPGA. This means the current value of internal signals and the state of state-machines cannot be determined. This is also referred to as the current state of the firmware.

If online debugging an FPGA design is required then additional components need to be enabled or inserted into the design. In order to increase observability in the design an in-circuit online analyser as ChipScope [5] can be used. This tool is embedded into the FPGA design and interfaced via JTAG. Beforehand all to be monitored signals and trigger signals must be defined and as core embedded into the FPGA firmware. During this process the design is changed which could make the problem disappear. The ChipScope core has the possibility to trigger on a certain condition. When this condition occurs the code starts filling memory with signal values. The more signals values are stored, the less time can be buffered. The ChipScope core tends to become very resource intensive when complex trigger conditions are constructed. No interaction is possible other than getting the acquisition buffer of ChipScope and setup triggers. Acquisition or triggering during initialization of the FPGA is not possible, since the JTAG connection is required to be initialized. The ChipScope interface groups the bus interfaces used in the design. A screenshot of the ChipScope interface is given in Figure 1.7.


Waveform - DEV:2 MyDevice2 (XC2VP30) UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O
my_npi_0_pxl_clk_pin	1	1
my_npi_0_l_valid_pin	1	1
my_npi_0_d_pin	0A	0A
wr_clk	1	1
valid	1	1
dout	0A0A0A0A	0A0A0A0A
XIL_NPI_AddrReq	0	0
XIL_NPI_WrFIFO_BE	F	F
XIL_NPI_RNW	0	0
XIL_NPI_Size	0	0
XIL_NPI_Addr	3CDEB040	3CDEB040
XIL_NPI_WrFIFO_Data	0A0A0A0A	0A0A0A0A

Figure 1.7: ChipScope interface showing captured data

If a non-intrusive method for monitoring signals is required, then probes can be inserted in the firmware. When using probes the Xilinx FPGA editor is used to edit the already fitted design binary to attach external pins to internal nets. This requires spare pins on the FPGA and is very labour intensive. Furthermore, the net names are often garbled due to optimization.

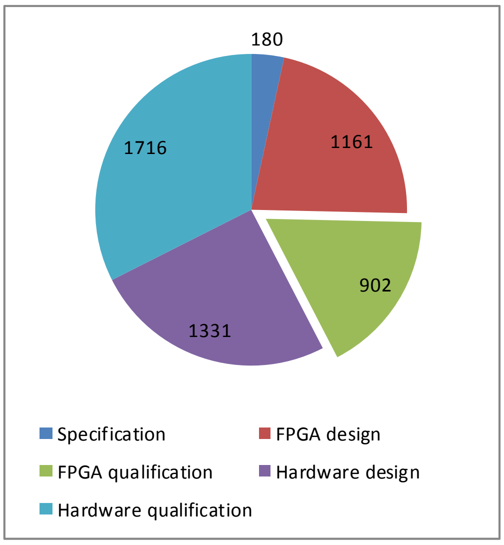


Figure 1.8: Pie-chart of the total project man-hours per phase for an electrical current controller. The FPGA qualification is a substantial part of the total project time. The debugging framework can also shorten the FPGA design time since qualification failure requires to locate and fix the mistake, which is design time as shown in Figure 1.9. The hardware qualification depends on the FPGA firmware. The project lead time can be shortened when the firmware errors are located and repaired faster.

Due to these complexities and time consuming process steps the qualification of FPGA firmware takes on average more than 17% of the project's design time. This time includes source-code simulation for individual subsystems, but also validation of FPGA output generated for given inputs, regeneration of FPGA binaries and the search time required for debugging. In Figure 1.8 a graphical representation of the project hours spent on an electrical current controller is given. The numbers shown in the graph are man-hours spent per phase. The colors in the graph refer to the project design steps explained in chapter 1. The actual time spent on qualifying a design is shown in Figure 1.9, debugging is part of this time. This diagram shows the time spent on both design and qualification in parallel. This is the time that a problem is being debugged.

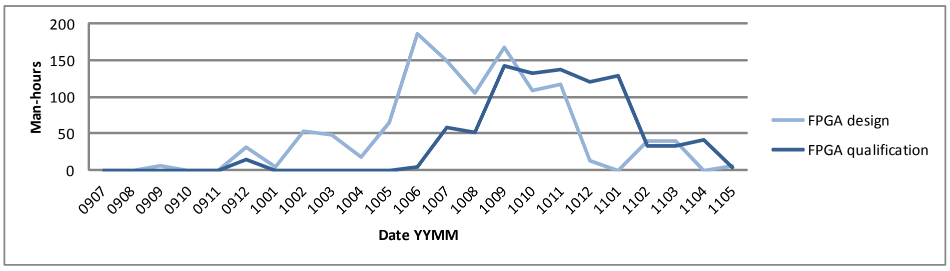


Figure 1.9: Chart showing the time when FPGA design and qualification runs in parallel. This time is the debugging phase of a project.

The goal for this project is to reduce the qualification and debug time for FPGA firmware by increasing observability and controllability of FPGA firmware.

The approach used to solve this problem is to decrease development time and increase design quality, while keeping the overhead of the system to a minimal.

The components which are responsible for the difficulties in debugging are the time and resource consuming simulation process which increases with the design complexity. This simulation cannot be fully trusted, since it abstracts from hardware and timing behaviour. The debug process on the end-result requires changing the binary; which potentially introduces other errors. The observability is limited; only for predefined signals measurements are possible. Every single change to the design requires running a time consuming process to create a new bit-stream. Error injection is only possible on defined locations using a either a ChipScope core, or specific design code needs to be inserted.

The target of FPGA firmware qualification is to validate that for every given sequence of input the correct output is generated. Due to state-space explosion this is not possible and only a subset of situations is tested. When the behaviour is not according to the specification this is marked as a bug. The debugging process is initiated to find and repair the bug. This debugging process requires information from within the FPGA in order to determine where the error arises. In most cases the designs are too large or do not have enough observability to easily locate the fault.

1.2 Contributions

The research conducted during this project contributes to the FPGA debugging world in different ways. The main contributions are:

- An **F**PGA **F**irmware **Q**ualification **F**ramework (FFQF) debugging methodology
- A template design supporting this methodology
- A framework using the template and additional tooling
- A model predicting the design impact of using the FFQF template
- A case-study checking this impact and demonstrating the framework and methodology

The FPGA debug methodology enables the possibility to monitor inter-subsystem communication. This communication centric approach of monitoring enables analyzing the input-output relation of subsystem data. This information can be used to track problems within a subsystem. If a given input does not produce the expected output then the subsystem requires further detailed analysis. This second stage of debugging is also contained in the FFQF debug methodology. The second stage of debugging enables setting breakpoints. These breakpoints are conditions that can be set at run-time. If a condition is true then the execution of a subsystem is stopped. The subsystem can be analyzed in different ways. The first way is to extract the path leading to the problem situation and repeat it in a simulator with full observability. The second method is to read-back the entire design and inspect it on flip-flop level.

The introduced template consists of a library with standard components to build the interconnection system. The library also contains the monitor and acquisition components. The framework adds tooling to this template used to interface the system and extract debug data. The tooling is currently in a premature state and ideas are presented in the future work chapter.

The impact of using this debug methodology is predicted in a model. This model shows the scalability assumptions and expected impact on design area, and communication throughput and latency. This prediction is validated for a single design via a case-study design.

1.3 Outline

The remainder of this document describes how the observability and controllability of the debug process is improved by using FFQF. The design template can easily be extended with configurable debugging modules. These modules are used to debug designs on-the-fly during qualification or when noticing run-time problems. Background information on FPGAs and current debug methods for FPGAs are shown in chapter 2. The relevant related work which is studied for this research is outlined in chapter 3. The techniques which are used to create this template and framework are explained in chapter 4, in chapter 5 this process and the techniques are evaluated and transformed into a model. The model is checked using a case-study which is described in chapter 6. The conclusion of the entire project is found in chapter 7, future work suggestions can be found in chapter 8.

Chapter 2

FPGA debug process

FPGA devices are flexible programmable hardware devices. In the industry where focus lies on short design times and flexibility of products these devices are widely used over Application-Specific Integrated Circuit (ASIC) devices. Disadvantages of FPGAs over ASICs is the cost of area and power. FPGA devices are programmed using a bit-stream configuration file. This file is constructed by converting VHDL in a format accepted by the FPGA. Background information on the internals of Xilinx FPGAs is shown in section 2.1. Information about the tooling and process of how the bit-stream is generated is shown in section 2.2. The VHDL code designed for these FPGAs can contain problems or bugs. The debugging process for FPGA firmware is explained in section 2.3.

2.1 FPGA hardware background

FPGA devices have evolved over time from a simple homogeneous grid of programmable logic blocks to a bulk of logic blocks in combination with special function blocks and memory. The Virtex-6 FPGA used in this project contains a number of different components as shown in Figure 2.1. These components are explained in this section; the information presented is a summary of [6].

The most elementary configurable building block of an FPGA is the Configurable Logic Block (CLB). These CLB blocks consists of two sub components called *slices* as shown in Figure 2.2. The slices are connected to the *switch matrix* which is used to route the signals that connect all internal FPGA components. Each individual slice contains four Look-Up Tables (LUTs), eight storage elements, wide-function multiplexers, and carry logic. The total number of available components in a CLB is shown in Table 2.1, and per FPGA in Table 2.2. In addition to this, some slices support two additional functions: storing data using distributed RAM and shifting data with 32bit registers. Slices that support these additional functions are called SLICEM; while the others are referred to as SLICEL.

Each CLB can contain zero or one SLICEM functions. Every other CLB column contains SLICEMs. In addition, the two CLB columns to the left of the DSP48E columns, shown in Figure 2.1, both

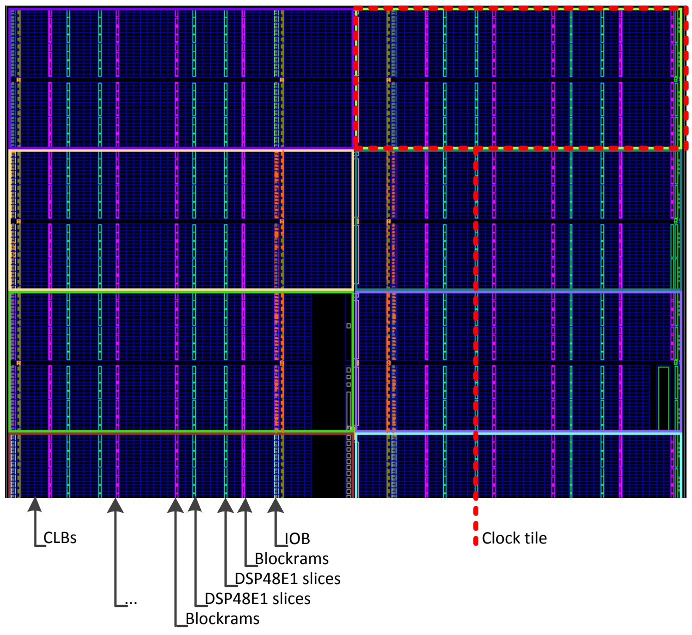


Figure 2.1: The floorplan taken from Xilinx PlanAhead shows the arrangment of logic components in the FPGA. Every clock tile can have its own independent clock signal.

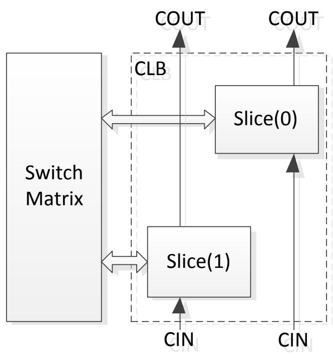


Figure 2.2: The two slices that exist in the CLB are connected to the routing logic. The components that exist in the CLB are divided over the slices and are shown in Table 2.1.

contain a SLICEL and a SLICEM.

The LUTs or function generators of the Virtex-6 are implemented as six-input look-up tables. There are six independent inputs (A inputs - A1 to A6) and two independent outputs (O5 and O6) for each of the four function generators in a slice (A, B, C, and D) as shown in Figure 2.3. The function generators can implement any arbitrarily defined six-input boolean function. Each function generator can also implement two arbitrarily defined five-input boolean functions, as long as these two functions share common inputs. Only the O6 output of the function generator is used when a six-input function is implemented. Both O5 and O6 are used for each of the five-input function generators implemented.

There are different types of distributed memory in the FPGA. This memory is either Random Access

Slices	LUTs	Flip-flops	Arithmetic	Distributed RAM$^{\text{SLICEM}}$	Shift Register$^{\text{SLICEM}}$
2	8	16	2	256bits	128bits

Table 2.1: Logic available in the Virtex-6 per CLB.

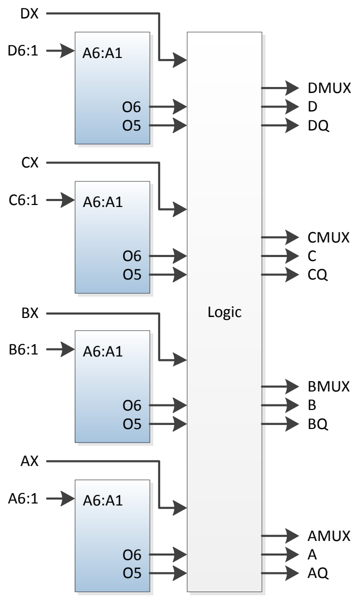


Figure 2.3: SLICEL component with the function generators in blue, the detailed version of this figure is found in [6].

Memory (RAM) or Read Only Memory (ROM). The distributed memory is generated by combining multiple LUTs in a SLICEM. The memory can be generated using special tooling which provides an interface to this memory. The ROM can be created in both the SLICEM and SLICEL blocks. The ROM is initialized at FPGA configuration and cannot be changed afterwards.

Besides the distributed memory in CLBs the FPGA also has columns with dedicated memory components. These components are referred to as blockrams [7]. A blockram resources stores up to 36Kb of data and can be configured as either two independent 18Kb RAMs, or one 36Kb RAM. The blockram resources are dual ported, and read and write synchronous. The two ports are symmetrical and totally independent, sharing only the stored data. The contents of the blockram can be configured and cleared at startup using the configuration bit-stream. The read and write operations on a blockram both require a single clock cycle per address. The blockram resources can be initiated using the Xilinx core generator software. This tool also enables higher level interfaces providing First-in First-out (FIFOs) and dual-clocked blockrams. The dual-clocked blockrams can be used as a barrier

Device	Slices	SLICELs	SLICEMs	6-input LUTs	Maximum Distributed RAM	Flip-Flops	DSP48E1 Slices	DSP48E1 Columns
LX240T	$37,680$	$23,080$	$14,600$	$150,720$	$3,770\text{Kb}$	$301,440$	768	8

Table 2.2: Elements and available memory in the Virtex-6 LX240T FPGA.

between two different clock domains.

All FPGA designs can be implemented using only CLBs, as long as enough of these components are available and the clock frequency can be set to an infinitely low value so timing errors due to routing will not occur. This is not the case for a lot of practical designs, therefore it is chosen to equip the FPGA with additional building blocks, types of these building blocks are the blockram memories described above. Another standard component is a number of columns with DSP48E1 slices [8]. These Digital Signal Processing elements are introduced to improve flexibility and utilization. The functions that the DSP slice can fulfill include multiply, multiply accumulate (MAC), multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect and wide counter. The DSP slices can also be cascaded in order to extend the width of all functions and perform complex filtering without the use of general FPGA logic. The number of DSP slices available in the Virtex-6 LX240T device is shown in Table 2.2.

Besides memory and DSP slices the Virtex-6 FPGA also contains even more dedicated components as High-Speed Serial Transceivers [9] and [10], PCI-express interfaces [11], and Ethernet MACs [12]. These dedicated blocks make sure that timing and signal requirements can be met for such high speed interfaces. They also decrease the design effort when using these components.

All configurable blocks in the FPGA together with the switch matrix or routing logic need to be configured. The configuration of the FPGA is done by means of a configuration bit-stream [13]. This bit-stream is a binary representation of all initialization values for all configurable logic. The bit-stream can be programmed into the FPGA via a debug link or the FPGA can read the configuration from (external) nonvolatile memory. The FPGA configuration data is stored in CMOS configuration latches, this requires the FPGA the be reconfigured after power down. The Xilinx tooling generates multiple types of bit-streams depending on the programming method. The FPGA configuration of the Virtex-6 LX240T FPGA consists of $73,859,552$ configuration bits. The sequence used for configuring the FPGA is shown in Figure 2.4.

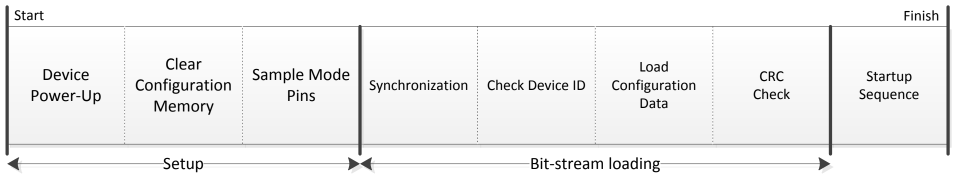


Figure 2.4: The configuration sequence for the Virtex-6 FPGA. The setup initializes the configuration process, while bit-stream loading part reads and writes the internal configuration registers using the bit-stream. The startup sequence gets the device out of shutdown.

Besides the standard hardware components there are also configurable standard software components. In Xilinx tooling there is for instance support for the generation of soft-core micro-processors. These processors are application processors implemented in the FPGA. These processors can be simple Reduced Instruction Set Computers (RISC), but also Memory Management Unit (MMU) equipped processor configurations are able of running embedded Linux.

The future FPGA devices are equipped with even more specific building blocks e.g. internal Analog

Digital Converters (ADCs) [14]. This makes it easier to use standard functions without wasting a lot of CLB logic. The latest addition is a dual-core ARM embedded into the FPGA fabric. This enables dividing tasks between a real processor and custom configured FPGA logic. The FPGA fabric has direct connections to the ARM core and both devices can share the same memory. These connections consist of approximately 3000 interface signals which also include memory mapped AMBA AXI connections. This processor can be used to replace the MicroBlaze soft-core processor which is widely used in FPGA designs. This type of Xilinx FPGA with embedded ARM Cortex-A9 multi-core is released under the name of *Zynq* [15].

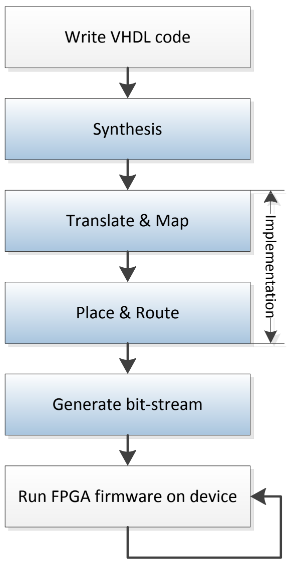


Figure 2.5: The process of converting the VHDL code to a bit-stream format accepted by the FPGA.

2.2 FPGA software background

In order to make use of FPGA components the VHDL code must be implemented and converted into an FPGA bit-stream. This implementation process is depicted in Figure 2.5. The process starts with logic *synthesis* where the VHDL code which describes the high-level behaviour is turned into a netlist with only logic gates and building blocks e.g. blockram, clock resources.

The synthesis output is fed into the next process step called *translate*. During translation all netlist files are merged into a vendor specific database file. This database contains the logical design reduced to vendor specific, e.g. Xilinx, primitives. This database is used by the *map*, or technology mapping, process to translate this design to CLBs, IOBs, etc. The output of this process is the physical design mapped to Xilinx components.

The mapped design is placed on a floor-plan of FPGA components. This floor-plan depends on the type of FPGA. After placing the components on this floor-plan they are connected with each other.

The *place & route* process takes care of this. This process takes all timing and signal requirements into account. If the design passes this step then all these requirements are met. There is only one step required in order to be able to program this design into the FPGA and that is the generation of a bit-stream. This bit-stream is generated in the *generate bit-stream* step.

In the latest FPGAs it is possible to overwrite partitions of a running FPGA. This requires all the implementation steps for only a portion of the design. This process is referred to as *partial reconfiguration*. This technique is described in [16], sample applications are shown in [17], [18], [19] and [20].

2.3 FPGA firmware debug decision tree

The design and implementation of FPGA firmware is done manually, this inevitably means that the firmware programmed into FPGA device can contain bugs. These bugs can potentially destroy hardware when for example safety mechanisms fail to activate. In order to find and repair these errors qualification and debugging is required. The debug process of FPGA firmware is depicted in Figure 2.6. This decision tree shows the paths which can be taken to ensure that the firmware meets the requirements. The FPGA firmware is written in VHDL, and then synthesised, and finally placed and routed into a configuration bit-stream.

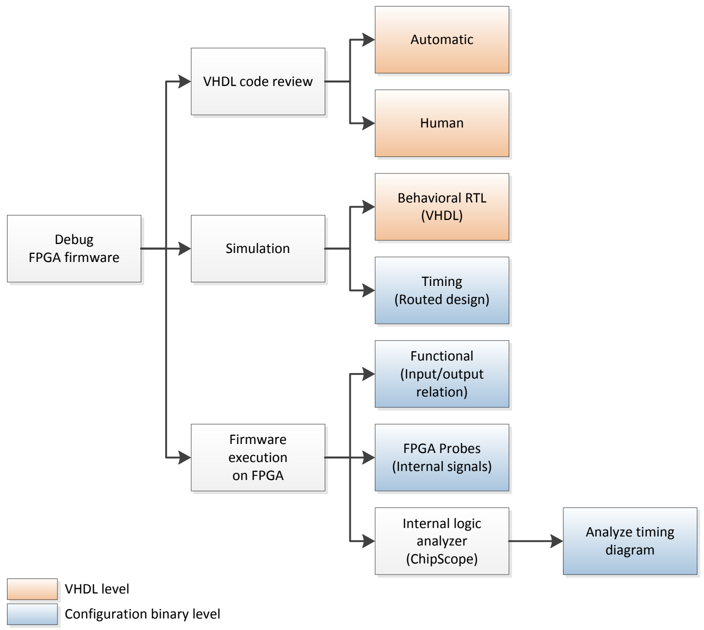

Debug FPGA firmware
VHDL code review
Simulation
Firmware execution on FPGA
Automatic
Human
Behavioral RTL (VHDL)
Timing (Routed design)
Functional (Input/output relation)
FPGA Probes (Internal signals)
Internal logic analyzer (ChipScope)
Analyze timing diagram
VHDL level
Configuration binary level

Figure 2.6: Decision tree showing the debug process for FPGA firmware. The three branches distinct between off-line analysis by code review, simulations, and online analysis while executing on the FPGA.

2.4 VHDL code review

The *VHDL code review* by *Human* is the process where other FPGA engineers read the VHDL code while keeping the architecture and module design in mind. The design document describes the behaviour which should be implemented in VHDL. The constructions of code which do not match the design specified in the design document can be found here. The VHDL code review is on conceptual level checking the relation between the design and the implementation. The review is also at VHDL construction level checking the VHDL code style for different constructions. Processes can be implemented synchronously or asynchronously, the use of a reset condition can be required, the method of initializing signals differs per FPGA, etc.

A VHDL code review can be done at every stage in the project from the moment there is VHDL code. The downside is that the reviewing engineer must be familiar with the design. Mistakes are easily missed when reading large amounts of code. The quality of the review depends on the experience and effort of the reviewing engineer.

An *Automatic* code review can be performed by linting applications. These applications check the VHDL code for mistakes, typos and typical error constructions. Problems found by the linter are bus width mismatches, assignment conflicts, unused signals, reading from output ports and setting input ports etc. The warning messages of the implementation process steps as map, place and route, and generate bit-steam are also considered as automatic code review.

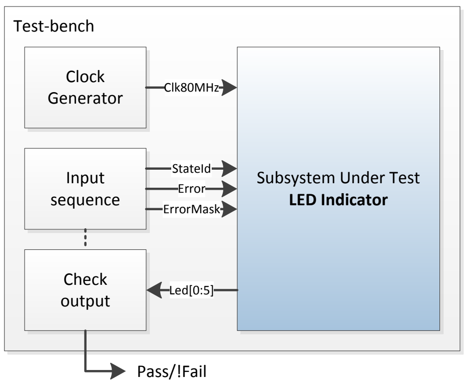


Figure 2.7: An example of a test-bench design. The clock generator generates a clock signal using VHDL statements. This clock is supplied to the subsystem under test together with a sequence of input signals. The output signals of the subsystem are checked by the check output block. If the output does not match the expected value for that input, then a fail is generated. The test-bench execution can be investigated by inspecting the clock and signal graph generated after execution.

2.5 Simulation

The VHDL code can directly be used for *Simulation*. In the industry this is referred to as *behavioural RTL* simulation. The design in such simulation process consists of two different parts. The first part is the module or subsystem under test, this is a process or process group taken directly from the design. The second part is the test-bench as shown as the white part in Figure 2.7. The test-bench consists of one or more processes that accepts extended VHDL constructions. This extension on synthesizable VHDL allows the use of delays, wait statements, file access and print commands. The test-bench generates all signals routed to the subsystem under test. This includes a clock and the desired input signals. The output signals of the subsystem are monitored. The test-bench can automatically report errors by checking the output of the subsystem under test. The sequence leading to the error can be inspected via the timing diagram. A timing diagram can show all internal signals at each moment in time of the last execution. This simulation step abstracts from internal hardware timing behaviour. The simulation is at RTL level, this means that it shows register values in a timing graph at each clock tick. All changes to registers are immediately applied, unless transport delays are defined. If the subsystem under test gets larger then the simulation time also increases, since the value for every signal is calculated for every time interval.

The simulation of VHDL code is fast and delivers a high level of observability. The VHDL code can be fed directly into the test-bench, no translation and technology mapping of the code is required. The abstraction from hardware behaviour makes it impossible to fully qualify the FPGA code since not all cases are covered.

Another type of simulation works with a fully implemented design. Fully implemented means the design went through the synthesis, place and route, and mapping steps. The output file describes the required logic and occupied location in the FPGA. These output files can be used for *timing simulation*. A timing simulation uses libraries that describe the component and wire delays of the design. During synthesis and during place and route additional netlist files containing this information are generated. These files are loaded in the test-bench instead of the VHDL file of the subsystem under test in Figure 2.7. The timing simulator cannot determine the exact delays of all logic in routing. The simulator makes use of minimal and maximal values on paths. The timing constraints are constantly monitored to see whether or not timing violations occur. This process requires much more computational power than the behavioral simulation. It has full observability but requires approximate a factor 100 more time to simulate compared to behavioral. The use of timing simulations is only required when clock domain crossings exist in designs. When all clocks are synchronous the implementation process makes sure all timing requirements are met.

2.6 Firmware execution on FPGA

Another branch that can be taken in parallel with the code review and simulation is *firmware execution on FPGA*. This means that the configuration bit-steam is loaded into the FPGA and a *functional* test is executed. During this functional test the FPGA is used as it would be used in the final product. This test can be extended with error sequences as for example disabling or overriding input signals and checking the behaviour in these situations. When for example the serial communication is stopped an error LED should light up. Signals can also be checked with an oscilloscope or multimeter when

LEDs are not available or too slow. This monitoring of functional behaviour requires no changes to the FPGA code. The code runs in its final delivery setup.

The functional test is very fast, since the code is running real-time on the FPGA. When problems are found during the functional test it is difficult to determine the cause of these problems. The functional test lacks observability, since only the output signals of the FPGA can be monitored. In some cases it is possible to read diagnostics data via the external interface.

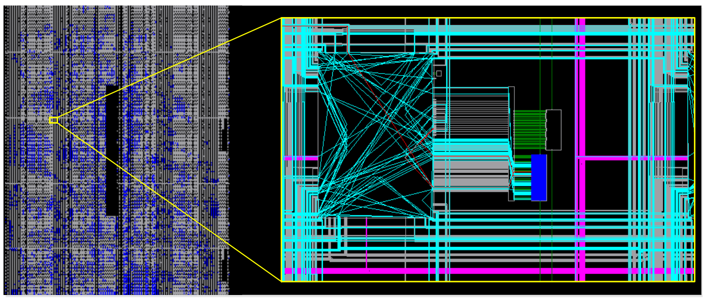


Figure 2.8: The routed FPGA bit-steam can be opened using the FPGA editor and connections from low-level nets can be made to external FPGA pins. These external "spare" pins can be monitored using for example an oscilloscope or logic analyzer.

In order to gain more knowledge about what happens internally it's can be helpful to be able to view internal FPGA signals. These internal signals can be manually routed to "spare" pins on the FPGA. These "spare" pins are unused pins on the FPGA which are routed to an external connection on the PCB. Using an FPGA editor as shown in Figure 2.8 the configuration bit-steam can be opened and these extra signals can be routed. This does not require re-running the implementation process, the bit-steam can immediately be saved. Some signals are difficult to find due to obfuscation. The implementation and synthesis process can combine signals that have equal values in order to optimize a design. When internal nets are combined they are given new names. These names are difficult, or sometimes impossible to relate back to the original signal. Another disadvantage of this method is that "spare" output pins are required on the FPGA. If these pins are not available, e.g. all I/O pins are in use, then this method cannot be used.

The use of probes has the advantage that probes can easily be added to an FPGA design when nets are not optimized and "spare" pins are available. Disadvantage is that usually only limited spare pins are available. It can take a lot of time before the correct probing location is found related to an issue, only a few bits can be analyzed at a time.

The final method mentioned in the decision tree is the adding an *internal logic analyzer* e.g. ChipScope [5] to a design. In this chapter the focus lies on ChipScope, but there are also other types of analyzers e.g. [21]. If a ChipScope analyzer core is added to the design then every build step after synthesis must be repeated. The design requires a new mapping, routing and bit-stream file. This step is time intensive for large designs e.g. $100.000+$ lines of code can take up to 2h. The advantage of these analyzers is that they connect via the JTAG interface. The JTAG interface is always embedded into the FPGA, and can be used to program the FPGA and to connect to the analyzer. No

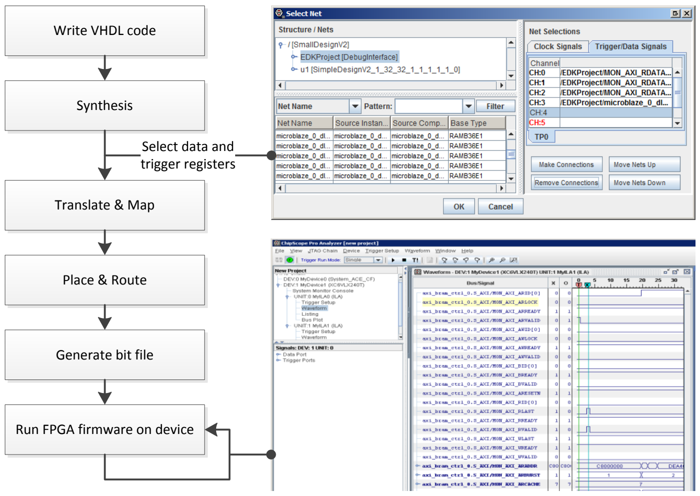


Figure 2.9: ChipScope triggers and registers to be stored after triggering are configured after synthesis. The trigger conditions e.g. values and masks are configured when the FPGA firmware is running in the FPGA. Via the JTAG connection and the and using the configuration file generated after synthesis the configurations can be made.

additional connect logic as for instance with the probes is required.

The internal logic analyzer works with triggers and buffers. During the analyzer configuration all signals required for triggering or acquisition need to be defined. Next the clock on which the core should run is selected. This clock is selected from an internal design clocks and is used as source for checking the trigger conditions. Different types of possible triggers can be selected when configuring the core. Setting the actual trigger value and method can be configured during run-time. When the analyzer is successfully configured and the bit-stream is build the design can be executed on the FPGA. A desktop tool connects via JTAG to the ChipScope core in the FPGA. Using this tool the trigger conditions can be configured, as shown in Figure 2.9. When the design triggers on an condition it starts filling internal memories with data from the selected signals. Due to the limited internal memory capacity, the more signals are selected the less number of samples can be stored. After the buffers are full the data is copied via JTAG and the timing diagram is shown. When other signals are monitored a new bit-stream is required. The design is changed when the analyzer is added and therefore it could introduce new timing problems. Parts of the design are moved in order to fit the analyzer. This changes the routing layout and can lead to data misses on the sampling clock edge of a clock domain crossing.

The advantage of analyzing a design with ChipScope or similar internal logic analyzer is it provides a lot of observability if configured correctly. Because of the JTAG connection no additional FPGA pins are required to view debug information. The disadvantage is that for every new signal that must be triggered or stored a lot of recompilation time is required. The ChipScope core not always fit in the FPGA together with the design.

2.7 Debugging methodology

The previous sections describe the current methods for analyzing, debugging and qualifying FPGA firmware. This research introduces a new debugging methodology to enable the combination of the advantages of existing debug techniques. The existing research that is studied and of which parts are reused is described in chapter 3. The FFQF debugging methodology introduces observability and controllability in FPGA firmware. The observability is introduced at two levels, it enables communication centric analysis of internally communicated information. This information can lead to the source of error situations. The second level is at flip-flop level storing the actual state of each low-level component in the FPGA. This enables viewing inside each process.

The controllability is introduced to enable stop conditions or breakpoints. At these points the design or parts of the design stop executing. When the design is stopped it is possible to load the trace with input values from the design. These traces can be loaded into an off-line simulation. This combines the speed of executing a design on the FPGA and full observability when a problem occurs.

The methodology is supported by a template design which includes a generic communication structure. The template also contains debug blocks as a monitor and replacement slaves. The design is required to meet the template requirements to be able to make use of all debug features.

2.8 Conclusion

To summarize the FPGA debug decision tree all items are added to Table 2.3. This table summarizes the advantages and disadvantages of each step in the tree. This table shows there is no optimal solution. The combination of observability during simulation and speed of run-time would improve the debug process. This combination makes the design run fast in its final environment while introducing observability in a problem situation that requires investigation. The injection can be used to overrule certain signals to trigger error situations or inject test patterns. This is introduced with the FFQF debug methodology.

Debugging method	**Requires Synthesis**	**Implement.**	**Bit-stream**	**Observability**	**Remark**
VHDL Code review (human)	No	No	No	-	
VHDL Code review (automatic)	No	No	No	-	
Behavioral RTL (VHDL) simulation	No	No	No	Partial	Fast for small designs, slow for larger designs, missing timing problems on clock domain crossings.
Timing simulation	Yes	Yes	No	Maximal	Very slow and only edge conditions shown, still not real hardware.
Functional code execution	Yes	Yes	Yes	Input / output	
FPGA probes in execution	Yes	Yes	Yes, for every change	Input / output / internal	Limited signals, only to “spare” outputs.
ChipScope analyzer	Yes	Yes, for every change	Yes, for every change	Input / output / internal	Can trigger on internal signals and view traces.

Table 2.3: A summary of the decision tree showing the differences between the FPGA firmware debug steps. The combination simulation observability and speed of run-time analysis would be ideal.

Chapter 3

Related work

There is already a lot of work done in the research and development industry regarding FPGA firmware debugging and debugging in general. During the start of this project, research has been done in the form of a pre-study [22]. In order to make debugging possible, observability and controllability are required. The research related to increasing observability is discussed in section 3.1. A selection of the debugging methodologies researched for this thesis are grouped in section 3.2.

3.1 Observability and controllability

The company *Temento Systems* sells an FPGA debug solution called *DiaLite* [21]. This tool works with fifteen different Intellectual Property (IP) blocks and scripts. These blocks and scripts make it possible to debug run-time FPGAs and add online assertions. Traces to the breakpoints are stored in order to re-run the faulty path off-line in simulation. The software connects via the JTAG chain. Their work does not support on-the-fly change of registers values and is therefore purely a viewer/monitor.

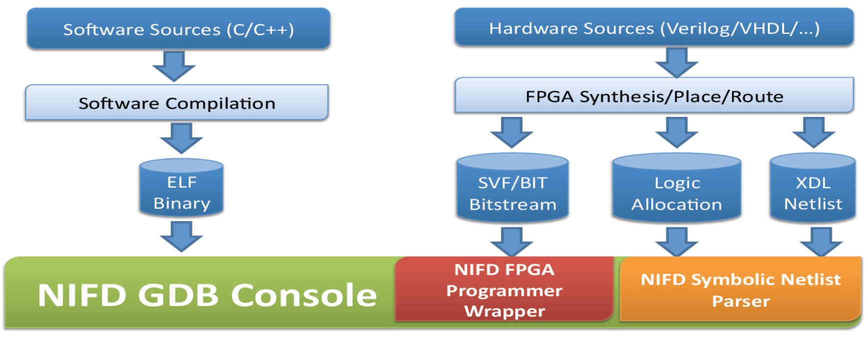
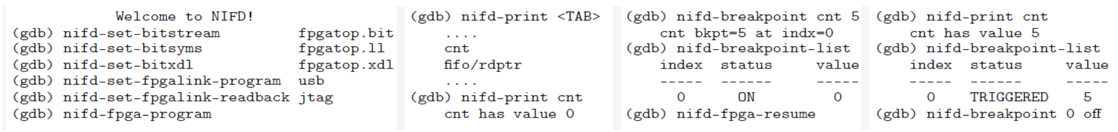

Software Sources (C/C++)
Software Compilation
ELF Binary
Hardware Sources (Verilog/VHDL/...)
FPGA Synthesis/Place/Route
SVF/BIT Bitstream
Logic Allocation
XDL Netlist
NIFD GDB Console
NIFD FPGA Programmer Wrapper
NIFD Symbolic Netlist Parser

Figure 3.1: Non Intrusive FPGA Debugger tool flow

A system with minimal overhead called *Non-Intrusive FPGA Debugger* (NIFD) is proposed in the work of *Angepat et al.* [23]. Their system creates a debug interface to the FPGA using GDB [24]. This interface is shown in Figure 3.2. The hierarchy of the original design is kept and mapped to the current state of the firmware running in the FPGA. This makes it possible to view internal registers by name as shown by the tool flow in Figure 3.1. In addition to this system it is also possible to enable breakpoints in the design and control the clocking. The read-back option is used to determine the state of the FPGA. The limitation of this system is the off-line host connecting via JTAG. The JTAG connection introduces a major communication bottleneck, as mentioned in the paper. The function of the JTAG connection can also be implemented internally using a MicroBlaze processor running Linux and GDB. This keeps the communication internal which enables higher throughput. The paper does not mention this alternative.



```
           Welcome to NIFD!
(gdb) nifd-set-bitstream         fpgatop.bit
(gdb) nifd-set-bitsyms           fpgatop.ll
(gdb) nifd-set-bitxdl            fpgatop.xdl
(gdb) nifd-set-fpgalink-program  usb
(gdb) nifd-set-fpgalink-readback jtag
(gdb) nifd-fpga-program
```

```
(gdb) nifd-print <TAB>
    ....
    cnt
    fifo/rdptr
    ....
(gdb) nifd-print cnt
    cnt has value 0
```

```
(gdb) nifd-breakpoint cnt 5
    cnt bkpt=5 at indx=0
(gdb) nifd-breakpoint-list
    index  status      value
    -----  ------      -----
      0      ON          0
(gdb) nifd-fpga-resume
```

```
(gdb) nifd-print cnt
    cnt has value 5
(gdb) nifd-breakpoint-list
    index  status      value
    -----  ------      -----
      0    TRIGGERED     5
(gdb) nifd-breakpoint 0 off
```

Figure 3.2: NIFD debug example in GDB, the figure consists of four independent columns each starting with (gdb). The first columns shows how the NIFD tool is configured. The second column shows which internal signals can be printed, in this case the value of `cnt`. In the third column a breakpoint is set when `cnt=5`, the status of the breakpoint is on. The last column shows that the breakpoint triggered.

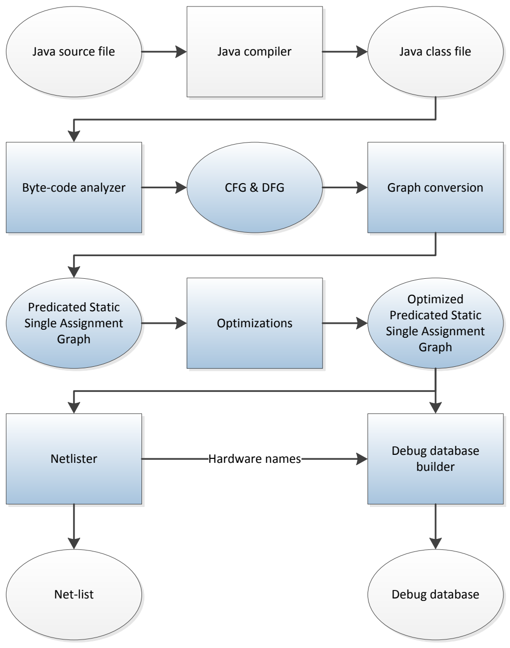
A design flow for monitor-aware network-on-chips is introduced by *Calin Ciordas et al.* in [25]. They propose a method for introducing monitor blocks at a level where the communication requirements are known. This enables minimal overhead in the monitor structure becase the monitors are adapted to the communication requirements of the network. The system assumes a network with a set of routers and standard network interfaces. This work is not applicable since the AXI network in this research makes use of a single interconnect.

The dissertation [26] of *Paul S. Graham* describes *Local Hardware Debuggers for FPGA-Based Systems*. This report describes the work of creating a software like debugging system with controllability and observability. The basis of this system is the Java-HDL (JHDL) [27]. The JHDL design environment provides an Application Programmers Interface (API) for describing FPGA circuits. The Java objects represent circuit elements in the FPGA. The read-back and state determination techniques all aim on the relation with the Java objects. Therefore this is not applicable for this research. The low-level read-back techniques can be used.

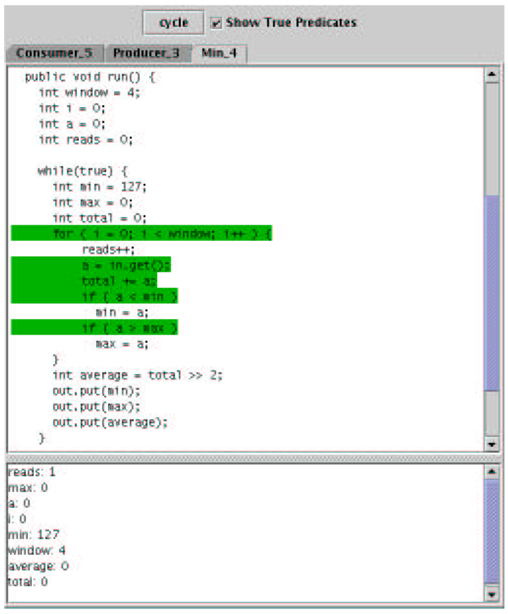
The use of hardware context switches, which can be used for debug breakpoints, is researched by *Trong-Yen Lee et al.* in [28]. This research has the focus on what information must be extracted from the FPGAs configuration registers to determine the state of the firmware. The state is used to be able to make context switches, by storing and re-storing this state at a context switch. A context switch is in their work required when programming alternative bit-streams in partitions of the FPGA using partial reconfiguration. The tool created by *Trong-Yen Lee et al.* takes the state of a portion of the FPGA via the capture mechanism in the FPGA. This captured data is read-back via an external interface called SelectMAP. This paper makes use of a *shutdown, capture and startup* when reading the data. In the FFQF research it is desired to leave parts of the FPGA operational while performing a capture on part of the FPGA. This means that a read-back is required without using the shutdown mechanism.

3.2 Debugging methodology

Debugging an FPGA device in a *software-like* fashion is explained by *Hemmert et al.* [29]. This software-like fashion means it is possible to set breakpoints on conditions, step with a given number of cycles and inspect signal values. Using high-level synthesis tools to map programs written in general-purpose languages to FPGA hardware has grown in popularity. It is becoming necessary to provide comprehensive debugging tools in order to verify the correctness of the synthesised hardware.



(a) Overview of the operations performed by synthesis tool proposed by *Hemmert et al.* in blue. The oval nodes represents generated files or formats while the rectangular nodes represent operations or tools.



```
public void run() {
  int window = 4;
  int i = 0;
  int a = 0;
  int reads = 0;

  while(true) {
    int min = 127;
    int max = 0;
    int total = 0;
    for ( i = 0; i < window; i++ ) {
        reads++;
        a = in.get();
        total += a;
        if ( a < min )
          min = a;
        if ( a > max )
          max = a;
    }
    int average = total >> 2;
    out.put(min);
    out.put(max);
    out.put(average);
  }
```

```
reads: 1
max: 0
a: 0
i: 0
min: 127
window: 4
average: 0
total: 0
```

(b) The current state in the FPGA is related to the original high level source code in *Hemmert et al.*

Figure 3.3: Synthesis operations and uses interface of the system proposed by Hemmert et al..

Currently, post-synthesis debugging is done at the circuit level. This research discusses the issues, as well as some early results, of creating a source level debugger for hardware synthesised from source code. This research provides insight in what must be added or built into synthesising compilers in order to allow the debugging of a synthesised circuit at source code level as shown in Figure 3.3(a). It discusses issues involved with both creating a hardware debug database and hardware-source level debugger. In this research high level JAVA code is used as high-level synthesis development language. A custom compiler is used to map the parallel generated instructions to an actual state in the FPGA as shown in Figure 3.3(b). This work does not fully apply to the research of this thesis since it makes use of high level synthesis and JAVA code. This thesis aims on debugging VHDL code in a software-like fashion.

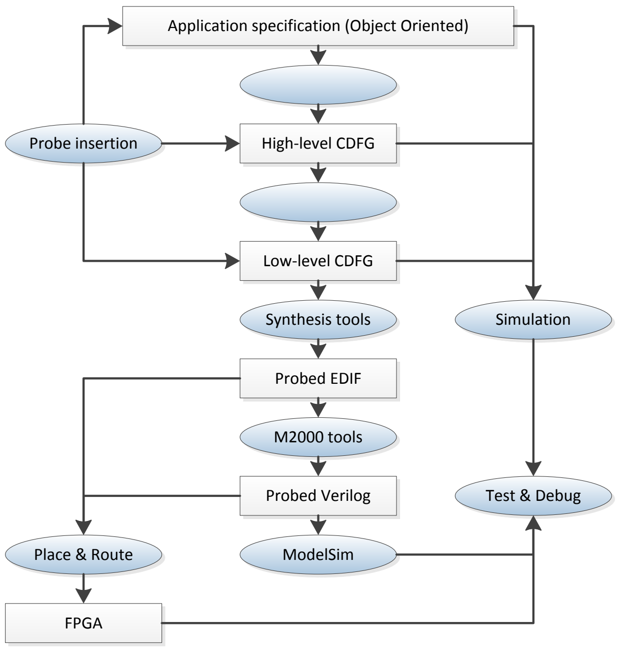


Figure 3.4: The introduction of probes in de EDIF file enables debugging and inspection of running FPGA code in the approach of Lagadec et al.

A methodology for HDL verification is proposed by *Denolf et al.* [30]. This research introduces the use of a `C` model of each subsystem. These models can be tested extensively by introducing probes in the model and applying stimuli to these probes. The behaviour is checked with the expected output of the block. Communication and functionality are separated. This project makes like *Hemmert et al.* use of high level synthesis languages in combination with debugging. In this case `C` code is synthesised into an FPGA binary and debugged.

Another High-Level synthesis approach for debugging an FPGA in a software like fashion is introduced by *Lagadec et al.* in [31]. This method uses Control Data Flow Graphs (CDFG) to represent the behaviour of the subsystems. Using an automatic injection method, breakpoint signals and monitor signals are injected into the Electronic Design Interchange Format (EDIF) file. This EDIF file is generated by the synthesis tool. The technique used for breakpoints is to insert a control module in each subsystem during qualification and remove it afterwards without changing the fundamental behaviour of the design. The process is graphically represented in Figure 3.4.

Online verification of the FPGA code against simulation results is proposed as *gNOSIS* by *Khan et al.* [32]. They propose a system which reads the entire state of the FPGA, converts this to a simulation

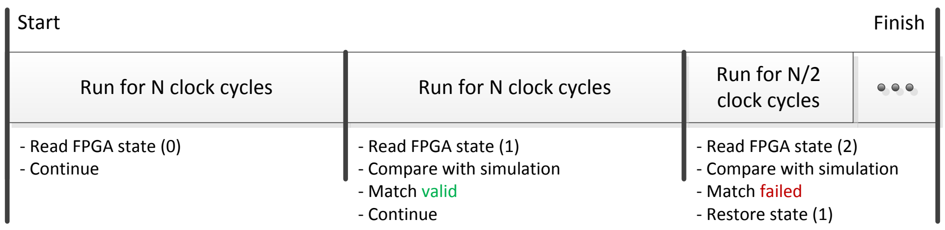


Figure 3.5: The method of gNOSIS is to run the FPGA code for N cycles and run the simulation for N cycles. These two outputs are compared. When the outputs match, the design runs for another N cycles. If the outputs of the FPGA run and the simulation do not match then the previous capture is restored and run for $\frac{N}{2}$ cycles. This sort of successive approximation approach is repeated until the exact problem cycle is found.

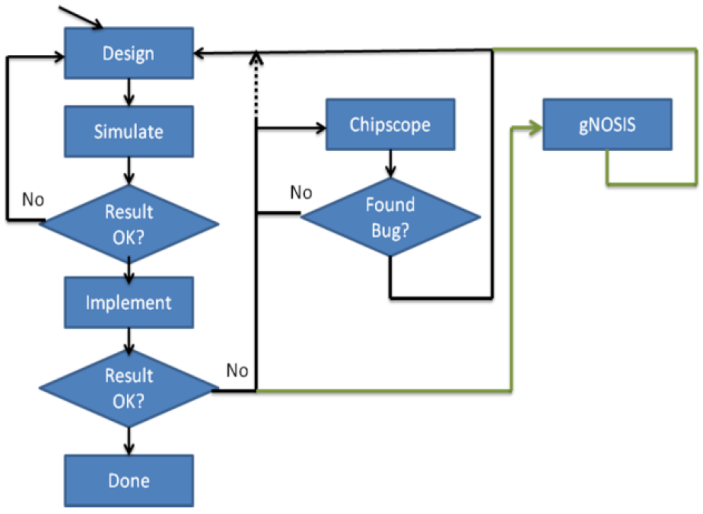


Figure 3.6: gNOSIS usage

state file and compares it with simulation output. If these files match then the design runs for another number of clock cycles and the design is compared again. On a mismatch the previous state of the FPGA is restored and the sequence is repeated in a smaller number of steps. This process is repeated until the exact clock-cycle is found where the error occurs as shown in Figure 3.5. In this research is determined how the translation from a read-back FPGA state line of source-code can be made. A drawback of the system is that it can only reproduce errors which exist between functional simulation and real hardware. The system can be used in parallel with ChipScope debugging as depicted in Figure 3.6. This introduces an extra level of observability.

A system which has a broader perspective and is designed to debug *Multi-Core Systems-on-Chip* is researched and designed by *B. Vermeulen (NXP) and K.G.W. Goossens (TU/e)* [33][34]. This system does not particularly aim on FPGA systems but on all Multi-Core / Multi-Process implementations. The system introduces both intrusive and non-intrusive debug cores in the design. Their on-chip *CSAR* debug approach stands for *Centered on* ***C****ommunication*, *Using* ***S****can chains*, *Based*

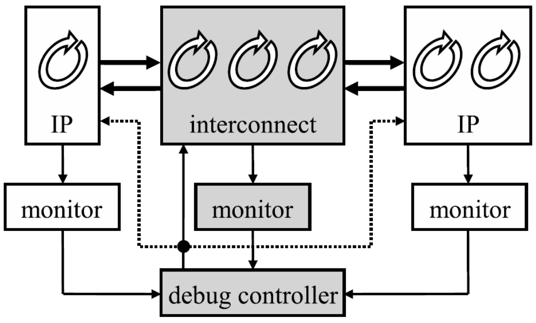


Figure 3.7: The communication centric approach introduced by B. Vermeulen (NXP) and K.G.W. Goossens (TU/e) for the CSAR system.

*on **A**bstraction* and *Implementing **R**un/Stop Control*. It aims at debugging the system at communication level as shown in Figure 3.7. This system is combined with an off-chip system called ***In**tegrated **Ci**rcuit **D**ebug **E**nvironment* (InCiDE) via the scan-chain TAP interface. This InCiDE performs the abstraction steps in the design. This means that, for example, low-level reads can be performed on addresses, but also on named registers. The register names are extracted at compile-time. The *InCiDE* system is interfaced via iTCL, an object oriented version of the TCL scripting language, commands.

3.3 Conclusion

The related work that has been studied during this project contains parts that can be reused. None of the studied research combines different levels of observability as proposed by the FFQF methodology. The different levels of controllability proposed in the related work is also combined in the FFQF methodology.

The DiaLite system [21] allows the usage of break conditions and traces. The tooling has no possibility to inspect at hardware register level and no method for injecting values. The NIFD [23] provides a debug interfaces a design via GDB. This NIFD research focusses on intrusive breakpoints and viewing signals while in break state. Interaction cannot be monitored non-intrusively. The method of reading hardware-level registers is re-usable. A method for non-intrusive communication monitoring shown in the work of *Calin Ciordas et al.* [25]. This method places monitors at routers and network interfaces. The communication infrastructure is reused when getting information from the monitor.

The researches that aims on debugging High-Level synthesis languages, i.e. [27], [29] and [31], are not applicable. The techniques used for inspecting hardware registers and capturing firmware state information is reusable. The debugging techniques that focus on differences between firmware execution and simulation are not reused. Using FFQF it is only possible to compare input and output traces of subsystems. The idea of using both intrusive and non-intrusive debug blocks shown in CSAR [34] is used with FFQF as well. The FFQF does not require the use of JTAG debug connections.

Chapter 4

FFQF template and framework

The pre-study document [22] advises to follow three orthogonal paths in the process of creating a qualification and debug framework. These different paths are shown in Figure 4.1. The *off-line debug* path refers to analyzing an FPGA design, when its execution has been stopped. This stop behaviour is initiated by a condition that is triggered internally or externally to the FPGA. The state of the FPGA is extracted and this information is used for off-line analysis. The *online* path describes debugging or analysing a running design without interfering with the behaviour i.e. non-intrusive. The last path referred to as *block reconfiguration* describes the partial online reconfiguration of partitions in the FPGA. The rest of the FPGA remains operational while a separate part is reprogrammed with a new design.

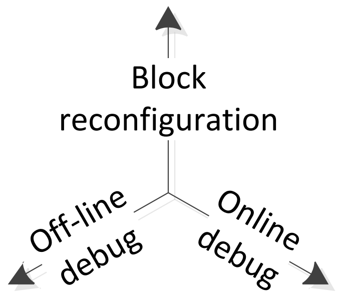


Figure 4.1: Different orthogonal paths or approaches to solve the main problem. The block reconfiguration aims on partial reconfiguration of a running FPGA. Off-line debug is simulation and FPGA register register data analysis. The online debug path handles debugging, while the firmware is running on the FPGA.

The *block reconfiguration* path is not considered in this project. The reason for using block reconfiguration is to possibly reduce the synthesis and build time during development. The method is to split the design into smaller components. These components or subsystems can be synthesised and build independently. During the development phase changes are made to these subsystems when correcting errors. The synthesis for a smaller design part is in theory faster than for the entire design. The individual subsystems can be connected, when building the final bit-stream.

If a design partition is defined then next the partition content must be defined. This content is the subsystems to be located in a certain partition. This is followed by setting the signal constraints of these partition border logic. This logic is the edge of the partition and communicates with both the internal and external part of the partition.

The Xilinx design flow states that partitions requiring the most resources must be fitted first. The reason for this approach is to make sure that all future partial designs will fit in the partition. The following step is to test whether all constraints can be met. Expected is that the Xilinx tooling will fit the partial designs in the partition based on the constraints set in the first design. This is partially true, since the design is synthesised and fitted in a smaller timespan. The problem is that the last step in the build process is rechecking all constraints. This checking makes the process even slower than without partial reconfiguration. This tooling limitation makes the application of partial reconfiguration in this research not applicable. There is another use for partial reconfiguration, but that is described in chapter 8 as future work.

The off-line and online debug paths are described in the next sections. This includes a chronological order in which the different template components are introduced. Evaluation of the template is not included in this chapter but these results are grouped in chapter 5.

The debugging methodology is based on a communication centric approach. In order to achieve this the template requires a generic communication infrastructure. This communication infrastructure is described in Section 4.1. The monitoring unit that enables observability in this infrastructure is shown in Section 4.2. In Section 4.3 a unit to inject or overwrite values is introduced. The notion of breakpoints is explained in Section 4.4. The breakpoints increase the controllability of the template. The template can be inspected at hardware level when a breakpoint is active. This inspection is explained in Section 4.5. The interface of the template is shown in Section 4.6.

4.1 Generic communication structure

In an FPGA design different subsystems perform functional tasks. These FPGA designs consist of strongly coupled subsystems. This coupling makes it difficult to determine when communication between subsystems occurs. The difficulty is that registers are updated immediately, when information is available. The information is used by other subsystems as soon as the information is required. During synthesis the compiler can optimize the design making the design use fewer resources, but more difficult to observe. Different net-names sharing the same functionality are merged into a new nets, which are more difficult to identify when probing with ChipScope or Xilinx debug probes. In order to decouple systems a new communication structure is proposed and introduced. This communication structure is shown in Figure 4.2.

There are different levels of abstraction possible, when grouping processes into subsystems. In Figure 4.2 these abstraction levels are shown. There are processes which are grouped into process groups. When these groups are able to independently fulfill a task they are referred to as a subsystem. A task is difficult to define, but examples of tasks are: handling external communication, calculate an electrical current loop, etc. The more subsystems are introduced, the more communication overhead, but also the more observability and controllability. This choice is therefore very application specific.

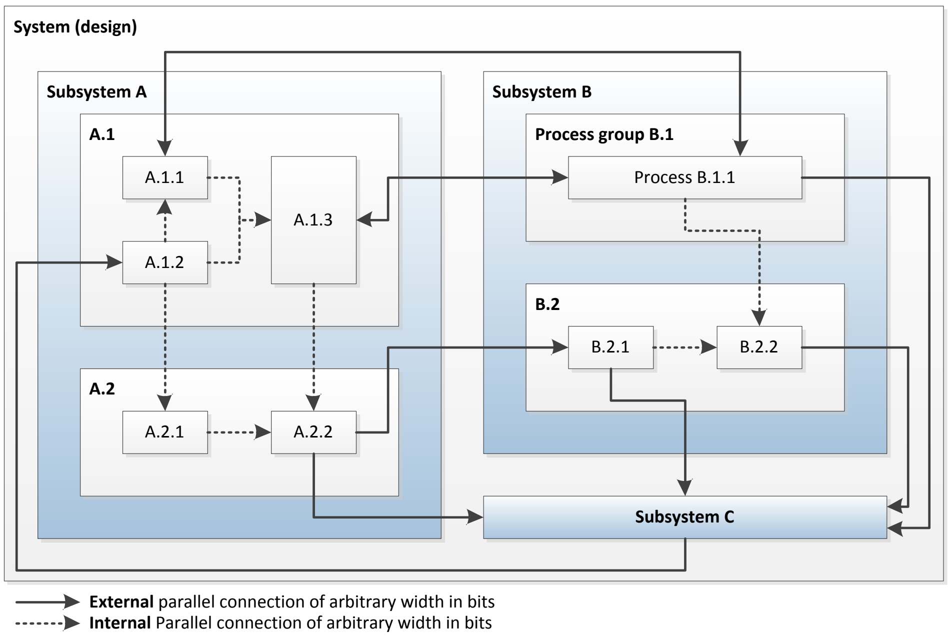


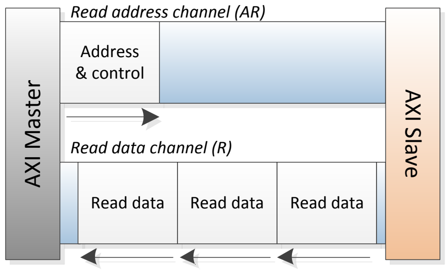
Figure 4.2: A tightly coupled system with direct connected parallel connections. There are different levels of abstraction possible in this existing design. The chosen subsystems are A, B and C. One of the chosen process groups is B.1, while one of the processes is B.1.1.

The subsystems are coupled too tight making it difficult to verify them when they are synthesised into the final design. The target of this design step is to decouple systems while minimizing the system's total overhead. The decoupling also makes it easier to determine the systems *internal (execution) state*. The internal state of a subsystem or process combined with the input generates the next state. This state information is helpful to determine what is computed at a given moment in time. The state of processor registers is stored when debugging software applications. Since an FPGA does not have these type of registers the state needs to be determined in another way. The state is required when design needs to continue after a part of the design is stopped. Stopping a design can be because of a break condition which is introduced in section 4.4.

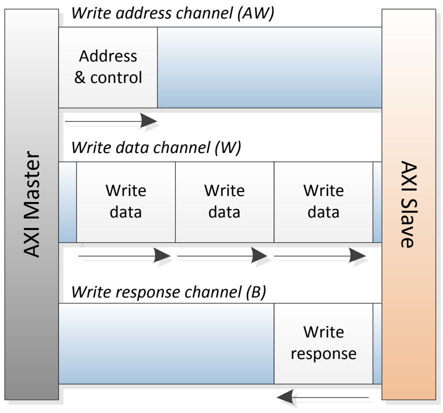
There are different standard interconnection bus systems available to be used in FPGA designs. Examples of these interconnections systems are Avalon [35], On-Chip Peripheral Bus (OPB) [36], Processor Local Bus (PLB) [37], AMBA AXI [38] and Open Core Protocol (OCP-IP) [39]. The AMBA AXI4 interconnection system is chosen to be the interconnection system to use. The ARM AMBA AXI4 interconnect is adopted by Xilinx and Altera as interface for their Intellectual Property (IP) blocks. The interconnection will be referred to as AXI bus for the remainder of the document. The reason for choosing this interconnection type is to ensure compatibility with future off-the-shelf building blocks of the FPGA and other vendors. This reduces development time and improves flexibility. By introducing this AXI bus in designs it is possible to create and use, fully independent building blocks.

The AXI bus architecture is designed by ARM and adopted by Xilinx for future logic cores [40]. The AXI interconnect system is implemented as an intelligent crossbar. The AXI bus specification makes a clear distinction between AXI4, AXI4-Lite and AXI4-Stream. The lite interface is used for single register reads and writes while the full AXI4 interconnect is able to perform memory mapped burst transfers. The light interface has a subset of the signals provided by the full interface. The last type is the stream interface designed for the use with memory and direct memory access (DMA) controllers. For this project all slaves, masters and interconnections are implemented for the full AXI4 system and not for the stream and lite interface. The lite interface can be derived from the full AXI4 system by removing the additional signals and setting burst size to one.

The AXI topology is very flexible, since it supports multiple masters and slaves. The routing is handled by the interconnection system making the master and slave implementations easy and independent. The AXI bus comprises of five individual communication processes or channels as shown in Figure 4.3. Two of these processes are responsible for handling bus read commands while the other three handle bus writes. The two read processes are independent of each other. The first process handles the address and control sequence, while the second process delivers the read data to the master. The three write processes have a similar setup only with an additional write response process. The write response process signals the master whether all data has been successfully received and processed.



(a) AMBA AXI4 Read

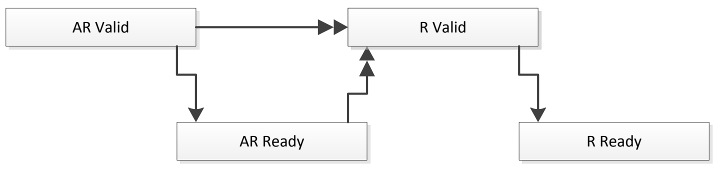


(b) AMBA AXI4 Write

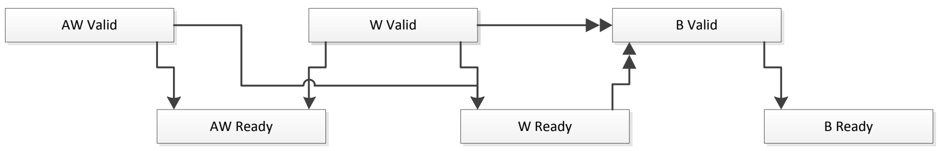
Figure 4.3: Showing the five independent AXI communication processes or channels. Two read channels and three write channels.

The AXI bus uses independent handshaking on all five separate communication channels. The handshaking works by means of *valid* and *ready* signals. The receiving side asserts a ready signal to show it is ready to receive data. The transmitting side puts its data on the bus and asserts the valid signal. Only when both the ready and valid signal are asserted communication occurs. The receiving side is at this point allowed to copy the data to internal registers. The specification of the AXI bus states that as soon as a communication is initiated that it should be completed, even when errors occur during communication.

The AXI communication protocol defines the handshake behaviour of the different communication processes. The handshake behaviour is graphically represented in Figure 4.4. There are two types of dependencies. The light dependencies are indicated as a single headed arrow that denotes a signal can be asserted both before and after the previous signal is asserted. The double headed arrows denote a strong dependency meaning that the previous signal must be asserted.



(a) AXI read handshake dependencies



(b) AXI write handshake dependencies

Figure 4.4: Showing the handshaking constraints set by the AXI specification. The single arrows are light dependencies showing signals that can assert before the previous signal is asserted. The double arrows are strong dependencies which cannot be asserted before the previous signal.

The communication structure comprises of different components. In order to make the template quickly adoptable to the existing design flow, the slave interfaces map their parallel connections into a register interface. Parallel connections that exist between processes within a subsystem can be mapped to the AXI slave but this is not required. If the information is relevant for debugging then this is advised. Using this method it is possible to monitor internal signal values, other than only the input and output of a subsystem. The AXI setup of the design in Figure 4.2 is shown in Figure 4.5.

The AXI bus works with an AXI master initiating all communication sequences and AXI slaves providing data. The control loop firmware is a constantly looping sequence with *measure*, *compare/-calculate* and *act* actions. The AXI master initiates all inter-subsystem communication. Due to this looping behaviour an off-line calculated Time Division Multiplexed (TDM) schedule is created. This schedule is fed into a bus arbitration unit which connects to the AXI master. The bus arbitration unit, interconnect and slaves are a completely independent system. The schedule of the AXI bus can be simulated in Modelsim [4] or implemented in the physical FPGA for qualification. The design blocks can be added to this design by attaching the parallel registers to the register interface.

The arbiter is responsible for the execution of the communication schedule. This schedule is stored in blockram memory attached to the arbiter. The internal state-machine of the arbiter shown in

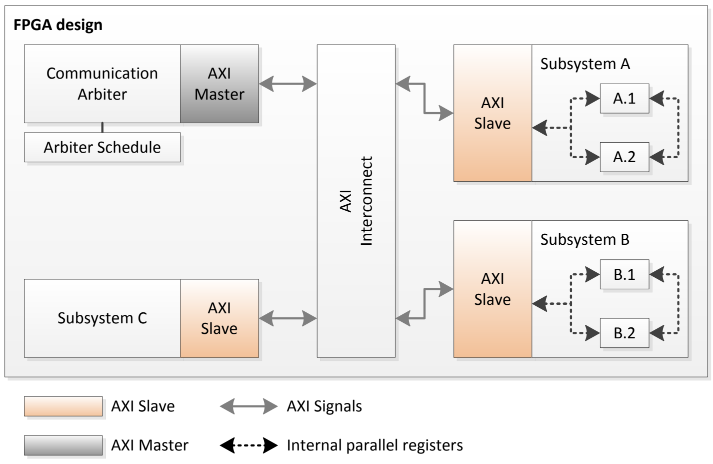


Figure 4.5: The original design of Figure 4.2 converted to a system using an AXI memory mapped interconnection bus.

Figure 4.6 handles this. The schedule consists of starting a read with a slave and as soon as data is available writing this data to another slave. Since AXI interconnect is memory mapped only addresses and sizes are used to initiate connections. In the *Read x* state the configuration memory is read to determine which address and size should be read. In the *Write x* state the next line in the configuration memory is read to determine the destination.

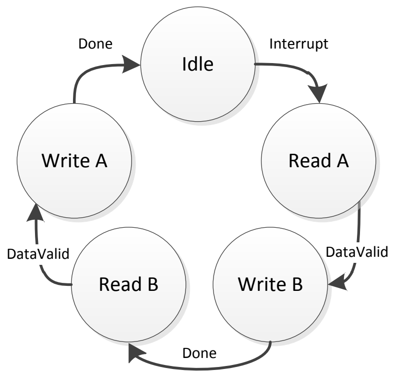


Figure 4.6: Communication arbiter state machine copying data between subsystem A and B.

In this project a custom implementation of the interconnect is made, because the Xilinx generated interconnect does not yet support simulation of their interconnect [41]. Therefore the design could not be validated off-line which significantly delays the implementation and validation process. The interconnect supports two AXI slaves and a single master. In the future this is easy to expand using the Xilinx Core Generator to generate a fully configurable AXI interconnect.

Next to the interconnect a generic AXI slave is implemented. This slave implements the complete AXI required logic for communication. This block works completely independent and can be attached to the AXI interconnect on one side and to a register interface on the other. Writing to a slave with a disconnected register interface has no effect. Reading from a slave with disconnected register interface returns all zeroes. This slave can be attached to a register interface that maps parallel registers to a location in the register interface. When this connection is made the slave will return the values on the matching location in the register interface. This slave can be attached to existing building blocks in order to insert them in an AXI design. The work required to do this is mapping parallel register into the register interface and calculate whether the TDM scheduler is fast enough.

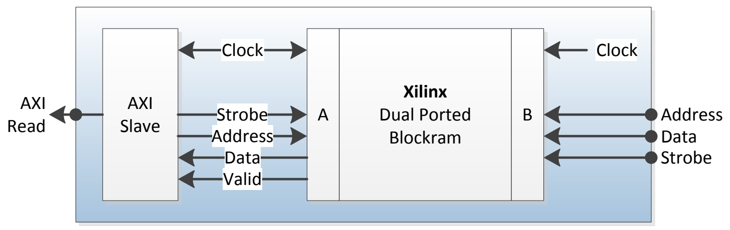


Figure 4.7: AXI acquisition blockram with a low overhead interface for filling, and a standard memory mapped AXI interface for reading

In order to enable diagnostics an additional AXI slave is designed, which is shown in Figure 4.7. This slave incorporates a dual ported blockram which is filled using the Xilinx blockram interface [7]. This interface enables a low latency and low overhead method for writing diagnostics data memory. The Xilinx blockram interface accepts new address and data signals every clock-cycle. The AXI interface attached to the other blockram port enables reading the data from the blockram via the standard AXI interface.

The functionality of the AXI interconnect is validated via simulation and by running the design physically on the FPGA. The design is inserted into an existing design of an electrical current controller. The TDM schedule and the input to output latencies have been checked both off-line and online. The results of the evaluation are grouped in chapter 5.

4.2 Bus-level spy unit (AXI monitor)

The individual subsystems complete a function or task. This output of this function is based on its current input and/or in combination with the current state of the subsystem. This states that the output of the subsystem is generated either from new unrelated information, or with history taken into account. The history, current input and current state are valuable when determining the root-cause of a problem. This problem can be noticed at the output. Using input and state information the problem situation can be reconstructed.

The more relevant information that can be extracted from the system. The easier it becomes to determine the root-cause of an problem [26]. The design is transformed into a structured network passing information at fixed moments in time. The next step is to determine a method to extract this information passed in the network without interfering. This form of non-intrusive debugging is used to determine the sequence that leads to a problem or event.

The AXI master is responsible for all communication. The master is controlled by the arbitration unit which executes the communication schedule. The master's AXI signals connect via the interconnect and to the slaves, as shown in Figure 4.5. This single connection can be spied by multiplexing it into an observer or bus spy. Spying the communication at this location introduces the least overhead. The other options would be to spy inside the interconnect. The disadvantage of this solution is that the interconnect cannot be generated with external tooling like Xilinx Core Generator [42]. Spying the slave interfaces requires a spy at each individual slave. This would increase the logic required for spying with the number of slaves. The used logic remains the same when spying at each the master.

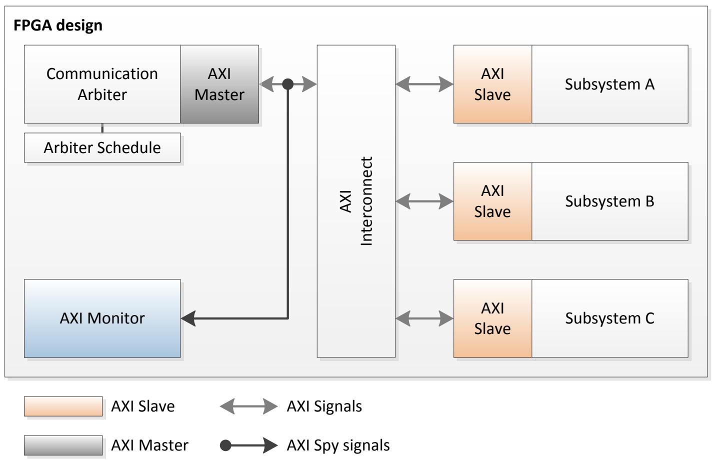


Figure 4.8: AXI spy interface monitoring the AXI master signals.

The signals that need to be monitored depend on the available resources and required information. The minimal required signals to monitor are the handshaking signals for each AXI process. The AXI processes are *address read (ar)*, *read data (r)*, *address write (aw)*, *write data (w)* and *write response (b)*. The handshake signals of the interface that requires monitoring need to be present in the monitor. The monitor can only accept data when both the valid and ready signals of the according data lines are asserted [38]. The monitor or spy unit designed during this project monitors the handshake, address and data signals for all AXI processes.

The monitor is able to monitor all address and data signals of the AXI bus. The monitor is also able to trigger on a certain address and data values. Using different masks and comparison types it is possible to create different trigger conditions e.g. equal, not equal, smaller and larger than. The match unit is depicted for a single AXI process in Figure 4.9.

The AXI monitor starts storing the communicated data into the internal blockrams when a trigger is generated. The AXI slave shown in Figure 4.7 is used to store the acquisition data. The standard blockram interface is used to fill the blockram since it has no burst size limitations and requires minimal communication overhead. Reading the blockram is a non-time-critical process and can be done via the AXI interface. This creates a standard slave interface for reading and a fast and low overhead interface for writing.

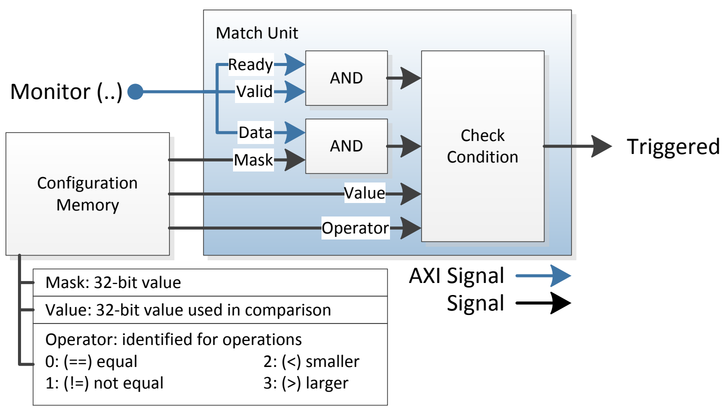


Figure 4.9: AXI match unit is used to determine whether a valid trigger condition occurs. When the trigger occurs the monitor starts acquisition of the communicated data.

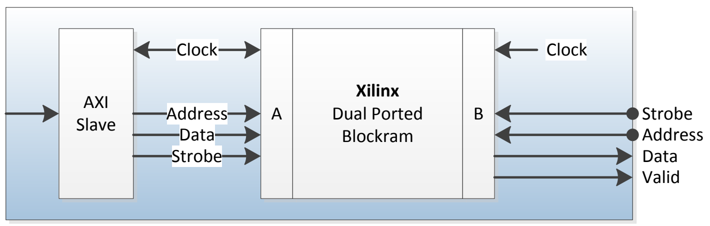


Figure 4.10: AXI memory which can be used for the configuration of subsystems

The configuration and control of the monitor unit is handled via control signals and a configuration memory. The monitor includes an $8\mathrm{bit}$ configuration register. This $8\mathrm{bit}$ register is directly controlling the state-machines inside the monitoring unit. This interface is similar to a General Purpose Input and Output (GPIO) register and is shown in Table 4.1. The real-time status of the monitoring unit is stored in another $8\mathrm{bit}$ register. This status register shows the state ID values of the different internal state-machines. The additional configuration of the monitoring unit is done via the configuration memory. This additional configuration includes setting the trigger conditions, mask values and compare types. This memory is a blockram that is filled via the AXI interface while the monitoring unit reads it via the standard blockram interface.

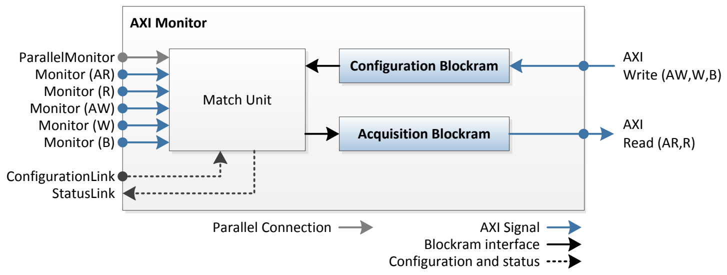
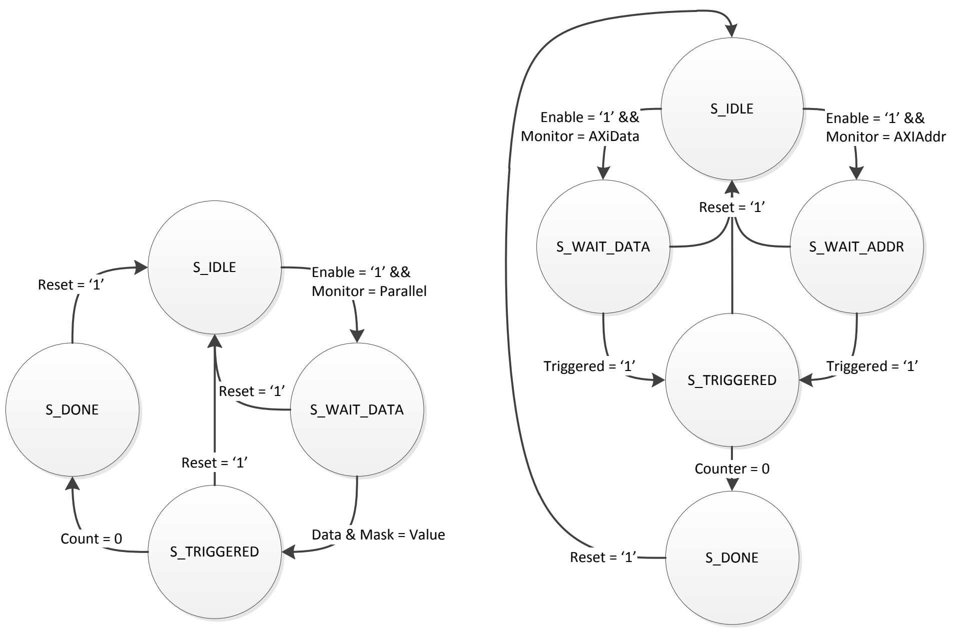


Figure 4.11: The configuration and acquisition blockram implementations are used in the AXI monitor.

Register	Mnemonic	Bit	Remark
Configuration	MonitorType	`[1:0]`	`00`, nothing `01`, AXI `10`, parallel `11`, reserved
	ReadConfig	2	Read the configuration memory
	Reset	3	Reset the state-machine responsible for acquisition
	Enable	4	Enable the monitor selected with `MonitorType`
	TriggerCapture	5	Enable the capture primitive
	Break	6	Request break condition
	Reserved	7	-
Status	ConfigState	`[1:0]`	`00`: IDLE `01`: WAIT `10`: STORE `11`: DONE
	ParallelMonitorState	`[3:2]`	`00`: IDLE `01`: WAIT_DATA `10`: TRIGGERED `11`: DONE
	AXIMonitorState	`[6:4]`	`000`: IDLE `001`: WAIT_ADDRESS `010`: WAIT_DATA `011`: TRIGGERED `100`: DONE
	Reserved	7	-

Table 4.1: The $8\mathrm{bit}$ status and control register of the monitor unit. The state codes refer to the states shown in Figure 4.12

The monitor unit is extended with an additional $32\mathrm{bit}$ register which can also be triggered on. The $32\mathrm{bit}$ length is chosen arbitrarily. Different signals can be combined into this registers. This register can be used to probe the internal design. Changing the probes means that the design must be rebuilt entirely. This is an extra feature to be able to acquire parallel register values directly into a buffer based on a trigger condition. This parallel register is faster to acquire, but less flexible since changes require the design to be rebuilt. The two parallel acquisition state-machines are depicted in Figure 4.12. Changing the register does require a complete rebuild of the FPGA image.



(a) Parallel register acquisition (b) AXI monitor acquisition

Figure 4.12: The two parallel state-machines responsible for acquiring data when a trigger condition is met.

4.3 Bus-level injection unit

The AXI monitor is able to monitor and store events that are communicated via the interface. Since the AXI bus signals are all uni-directional it is not possible to provide subsystems with alternative input data. This alternative input data can be used to inject events into a subsystem or to qualify the behaviour for a range of input values. This injection is helpful when qualifying or validating subsystems. The qualification of a subsystem is checking whether its behaviour is according to the specification for all specified situations. Since simulation abstracts from real hardware behaviour running the design on the FPGA is required in the validation process.

It is desirable to introduce a possibility to provide subsystems alternative input data. This can be at runtime via the register interface. This input cannot be given via the monitor interface since it is purely a read-only system able to monitor transactions.

The memory mapped bus interface is used to inject this data. Every blockram or register interface that is connected to the AXI interconnect can behave like another slave. The only change required is to the communication schedule. The arbitration unit has a number of *from* and *to* addresses which are used to copy data between slaves. Changing the *from* address to a blockram results in supplying alternative data to a subsystem under test. This subsystem performs its computations and the monitor unit can monitor the output generated with the alternative input. The subsystem used to mimic other subsystems is shown in Figure 4.13.

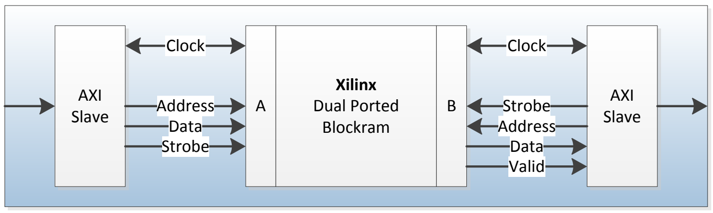


Figure 4.13: The AXI blockram can be filled and read simultaneously via different AXI interfaces. The blockram is connect to two different individual AXI interconnection networks and masters.

There are different methods for maintaining the alternative data stored in blockrams such as Figure 4.13. The first option is to create a *filler* process that is fast enough to overwrite the data in the blockram before it is used. This is not desired since timing is a critical component here. When the data is not updated fast enough the subsystem re-uses old input for computations. Another option is to store all data in the blockram at once in sequential blocks as depicted in Figure 4.14. This requires the arbitration unit to be updated with a new address table fast enough, or the arbitration unit must know how many addresses to skip for each new data transfer. The last option is chosen, because it introduces the least amount of impact.

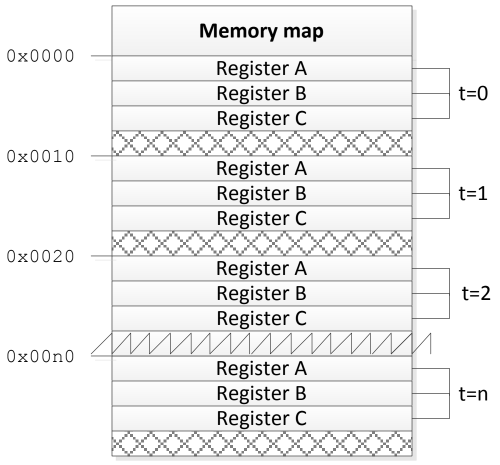


Figure 4.14: Memory map showing sequential data used for replacing subsystem with stub, the step-size between copy actions is $0x10$.

The arbitration unit is provided with a configuration memory and checks this memory to determine which data needs to be copied. Reading this memory is a part of the TDM schedule that is designed for the arbitration unit. The configuration AXI slave shown in Figure 4.10 is used for this application. The AXI interface makes it easy to configure the arbitration dynamically, while the arbitration unit can read from the memory with 1 cycle latency.

The arbitration unit is supplied with an alternative configuration in order to inject data into the external registers of subsystems. The injection method works via a communication centric approach. This means injection is handled via communicated events and not via internal register writes.

4.4 Break conditions

The events that happen on the communication bus can be monitored and data can be injected when the alternative data schedule are applied at the right moment in time. This moment in time is difficult to determine when the FPGA is running e.g. ranging between 40MHz and 200MHz.

In software development it is common to set a breakpoint [43] for when a condition occurs or a certain line of code is executed. A graphical representation of a breakpoint is given in Figure 4.15. These breakpoints are not available in Xilinx and Altera FPGA tools. Breakpoints are a powerful method to determine when new data needs to be supplied to the slave. If the design stops just before supplying new data then the alternative data memories can be filled. The execution continues and the output is checked for correctness by reading the output registers.

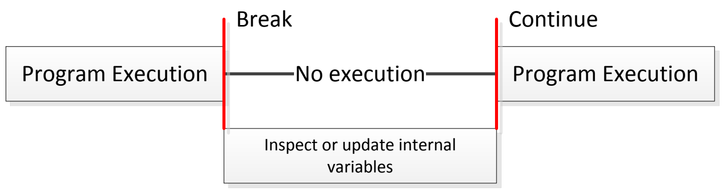


Figure 4.15: A breakpoint in typical software debugging. The program executes until a certain line of code. At this line the execution is stopped and it is possible to update and view internal variables. After viewing or modifying the variables it is possible to continue execution.

It is desirable to introduce the notion of breakpoints. This enables stopping a design or parts of a design when a condition occurs. The state of the design must be preserved to ensure a continue after breaking is possible. This continue means that the design continues to execute starting where it was stopped taking injected values into account.

Stopping an FPGA design means that all signals and registers are preserved from the moment the *stop* is triggered. The designs to which this applies are all clock synchronous designs on the rising edge of the clock. This means that when the clock remains high that all data is preserved. Starting and stopping a single subsystem with no relation to other systems is relatively easy. Stopping only requires a technique to physically stop the clock when it is high. If the subsystem has a relation with another subsystems, then there are different options. The subsystems that continues executing is informed that the data from the stopped subsystem is invalid. Another option is to stop both subsystems. How this is handled is a conceptual choice based on the type of application.

In order to stop on more conditions than only an external trigger the debugger monitors the 32bit parallel register. This register is used for break conditions. This register can be attached to parallel signals in the design. The monitor unit can, via the configuration interface, be configured to trigger on matching conditions on this interface.

The introduction of breakpoints or conditions is split into two independent problems that require their own approach. The first problem is the FPGA hardware technique required stop design parts. Only parts of the design are stopped to still have the possibility to read register values of AXI slaves. This problem is addressed in subsection 4.4.1. The other problem is the subsystem state and how related subsystem states are handled. The stating problem is described in subsection 4.4.2.

4.4.1 Technique for stopping the clock (clock gating)

The clocking architecture of the Virtex-6 FPGA [44] used in this project is similar to that of other Xilinx FPGAs. This research aims on this series of Xilinx FPGAs. Other vendors as Altera [45] also support clock gating.

It is desirables to determine a technique that can be used to stop individual subsystems while other parts of the design remain operational.

The FPGA is build-up from a different number of individual clock-tiles. These clock tiles are split by the clock-spine in the vertical direction and horizontal clock-buffers in horizontal direction. A single clock tile is divided into a number of clock regions. These regions are the smallest granularity in which the design can be supplied with an individual clock. These different clock regions are shown in Figure 4.16. The figure shows only four clock tiles, while the Virtex-6 FPGA contains a total number of twelve clock tiles, six on the left and six on the right.

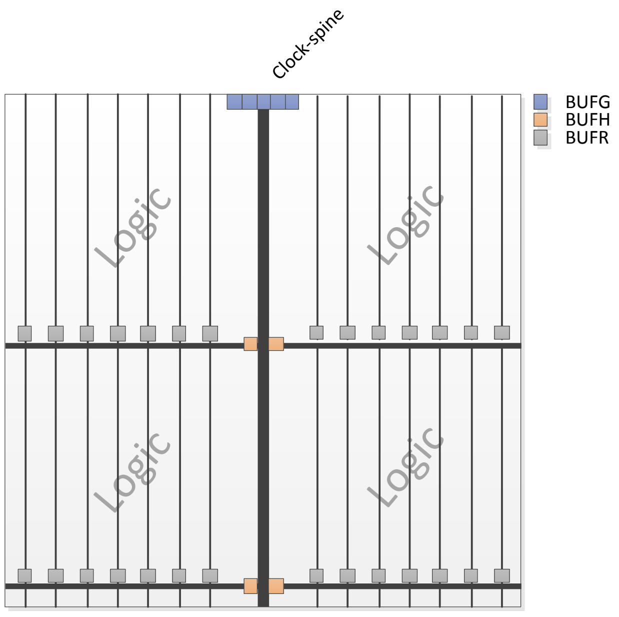


Figure 4.16: Different clock routing components of the Xilinx Virtex-6 FPGA. Clock gating is applied at the BUFR components. The BUFR is a regional clock buffer, BUFH is an horizontal clock buffer and the BUFG is a global clock buffer.

The clocks of the FPGA are usually generated by the MMCM (Mixed Mode Clock Manager) blocks. It is also possible to directly accept an external clock bypassing the MMCMs. The MMCM blocks route the clock signals to all other components. The regions shown in Figure 4.16 are *BUFG* which are Global Clock Buffers. These buffers run from the center clock-spine to all individual tiles. The tiles get their clock from this clock-spine via the *BUFH* components. These Horizontal Clock Buffers splits the main clock-spine into branches. The *BUFR* of Regional Clock Buffers route the clock to the clocked logic.

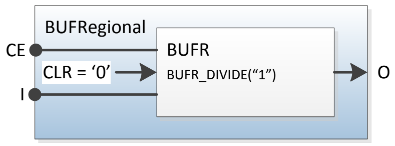


Figure 4.17: FPGA BUFR component, when CE is high the clock I is passed to the output O. If the divider is used the then clock is divided with this factor.

The clock controlling logic is chosen at the lowest level, e.g. BUFR [44]. The regional clock buffers are given an extra enable signal. The BUFR component is shown in Figure 4.17. The `BUFR_DIVIDE` setting allows to specify a clock divider at this level. The clock divider must be set to `1` since the register interface and function clock must be equal. The `CLR` or *clear* signal is ignored since the divider is set to `1`. The wrapper created around the BUFR primitive routes the `I` input clock, `CE` clock enable and output clock `O`.

4.4.2 Subsystem behaviour on a break condition

It is desirable to stop a subsystem for inspection. Stopping a subsystem or parts of the design means that this subsystem stops interacting with the rest of the design. There are different problems with stopping a functional part of the design which require attention. The input provided to the stopped subsystem is not used, since the functional code is not running. The output of the system which still exists in the output registers cannot be trusted, since it is unknown whether these registers are up-to-date. In order to avoid these problems all subsystems are stopped at the same time. The only active part of the design is the interconnection structure including the master and slaves. This structure remains operational to be able to read all registers and buffers from the moment the design is stopped. The architecture of this system is shown in Figure 4.18. The clock is deactivated for the light blue parts. The dark blue communication arbiter stops after its current copy action and the interconnect, AXI slaves and register interfaces remain operational.

The clock to the slaves is stopped immediately, when the communication arbiter finishes its last transaction. Finishing the current transaction is required, because the same communication bus is used to retrieve the state of the subsystem. The AXI specification states that all initiated transactions must be finished even when an error occurs. The state of a subsystem represents all information or data which is used determine the next state of the subsystem.

The state needs to be stored for all individual subsystems that require inspection. The designer of a subsystem is aware of which information is used in the state determination. The state is stored in ring-buffers which are similar to the acquisition AXI slave Figure 4.19. The difference between the AXI acquisition slave and AXI state acquisition slave is the number of blockrams. This state acquisition blockram has internal logic and procedures making it easier to insert it into a subsystem.

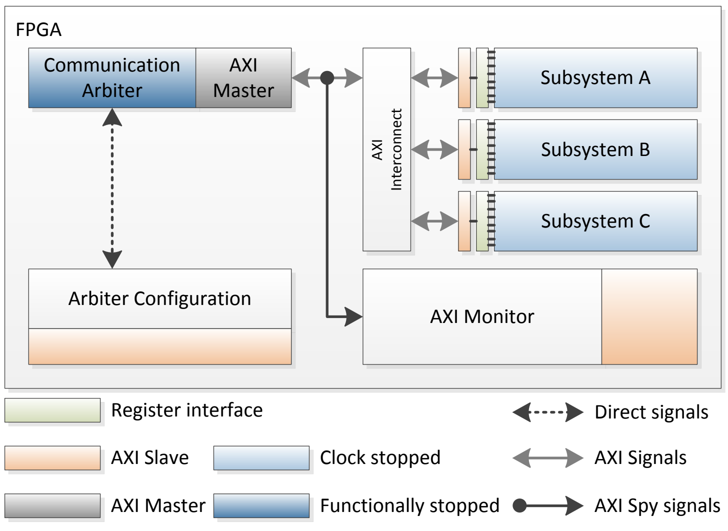


Figure 4.18: Top-level architecture of the FQF system showing the which components stop and which remain operational during a break condition.

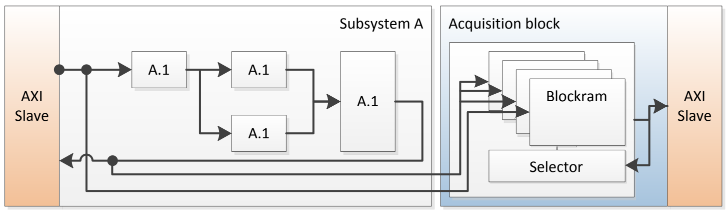


Figure 4.19: State acquisition blockram contains multiple ring-buffer blockrams which stores parallel acquisition traces while only being a single AXI slave on AXI side.

The state acquisition blockram is able to store the internal state of the subsystem. This state is stored in internal blockram ring-buffers. In Figure 4.19 this state acquisition blockram is attached to a subsystem. The detailed version of this system is shown in Figure 4.20. The designer of the system determines which acquisition points e.g. `Data0`, `Data1`, `Data2`, `Data3`, etc. are relevant for state information. The trigger determines at which moment acquisition must be done. Triggering on the clock makes sure every clock-cycle the state is captured. Every 32bit register is assigned to its own blockram. This implementation makes sure that it is possible to acquire data every clock-cycle. The acquisition occurs in parallel and the blockram interface requires only a single clock-cycle for writing.

When the subsystems clock is stopped, due to a break condition, the data of that clock-cycle is the last record in the ring-buffer. The ring-buffer is acquired via the AXI read interface. The blockram selector and address translator, translates the AXI read command to the correct internal blockram. Since the blockram is a ring-buffer the AXI read addresses do not match the internal blockram addresses as shown in Figure 4.21. In order to solve this difference an address translator is implemented. The ring-buffer loops at the end of the buffer and each cycle represents a trigger.

The information read from the state acquisition subsystem ring-buffers, can be imported or loaded

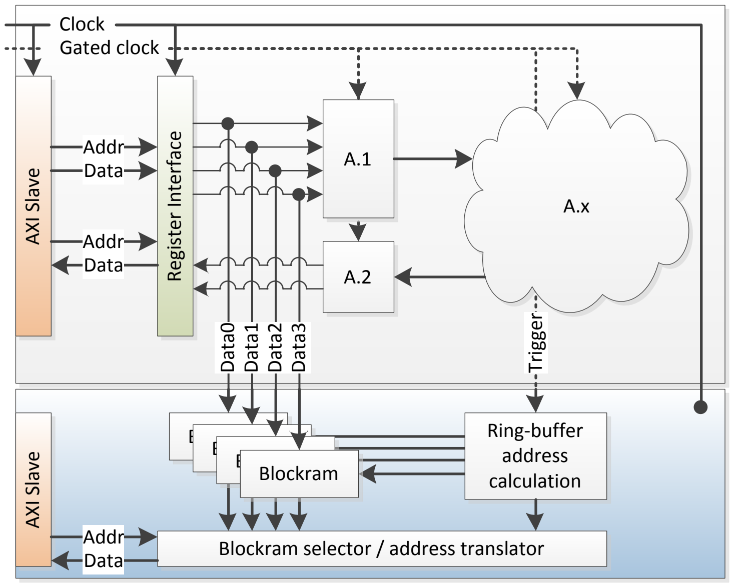


Figure 4.20: Detailed figure of the state acquisition subsystem shown in Figure 4.19.

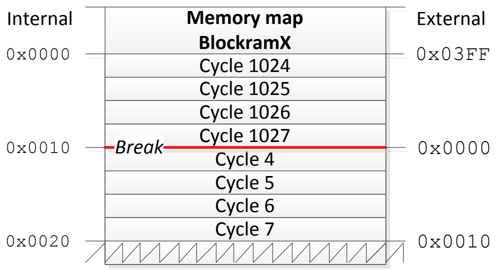


Figure 4.21: Blockram internal address is translated to the external address used by the AXI interface

into the Modelsim test-bench. Providing this information to Modelsim enables replaying the sequence that leads to the break condition with full behavioural observability. If the output signals are stored as well, unlike in Figure 4.20, then it is also possible to validate the simulations behaviour with the real FPGA behaviour. This can show problems that exist between the FPGA implementation and the way the simulator handles this. The simulator abstract from low-level hardware behaviour and signal timing behaviour and could therefore behave differently.

The test-bench is equipped with a procedure that reads a line in the file that contains the trace on every clock cycle. This means that every clock cycle the same data is provided to the subsystem under test as it was during execution. The timing diagram shows the internal signal values calculated by the simulator. This provides additional observability.

In order to continue the execution of the design, until a new break condition occurs sometimes re-initialization must be done. If the design is stopped, and the current transfer is finished then this data must be discarded or marked invalid. The data that has been marked bad must be re-transfered, when the subsystem restarts. There are different moments in time at which a break condition can

Phase	Description	Action
$\#2 - \#0$	Idle	No data requires to be discarded since all communication already completed successfully. The arbiter jumps to a wait state until execution is continued.
$\#0 - \#1$	After initialization of the read sequence	Discard the data read from the subsystem's register. The read is already initiated thus cannot be stopped. The data should not be written to the receiving slave. This is referred to as discarding the read data. When execution continues it restarts at the point just before initialize read. This restart the read process.
$\#1 - \#2$	After initialization of the write sequence	The read has already been started and cannot be stopped. This is the same as for condition $\#0 - \#1$, but in this case parts of the read data have already been written to the destination subsystem. If possible the destination subsystem must be notified that incorrect data has been written. This is referred to marking data invalid. It is also possible to disallow breakpoints during this phase and postpone them until $\#2 - \#0$ or $\#0 - \#1$.

Table 4.2: Different phases of the arbiters communication schedule, see Figure 4.21 for a timeline reference.

occur. The amount of data that needs to be discarded differs per situation. These distinct situations are depicted in Figure 4.22 and the according actions are shown in Table 4.2.

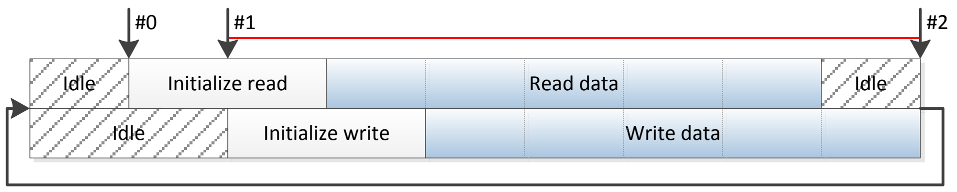


Figure 4.22: A break conditions can occur in different phases of the arbiters communication schedule, see Table 4.2 for the impact and actions.

In Table 4.2 is explained that it is impractical to stop the design during an active bus transaction, e.g. interval $\#1 - \#2$. The slave has already received parts of the input data and immediately starts computing with this data. Due to these immediate calculations some of the internal register integrity cannot be guaranteed. It is too resource intensive to keep shadow registers of the entire register interface. This introduces latency, because the entire register interface must be written in the shadow register. This register is copied to the real register in a single clock-cycle. The time it takes to fill the shadow register is added. Since additionally introduced delay is intrusive, and limits the design possibilities it is chosen to delay the break situation until the bus transaction is complete. The moment in time where the break is generated is stored in the state acquisition buffer in order, to identify the trigger location in the Modelsim simulation.

During a break condition in the red area of Figure 4.23 and Figure 4.22 the arbiter is notified about the break request. This request will be accepted after the write transaction completing at clock cycle 15 and 30 in the example. The arbiter jumps to the breakpoint state where it constantly reads the configuration memory and processes those transactions. At the end of each transaction the arbiter is allowed to jump back to the point where it came from. The extended state diagram of the arbiter can be found in Figure 4.24.

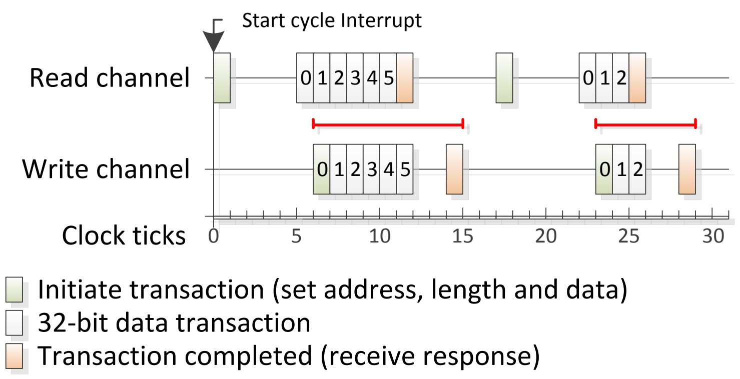


Figure 4.23: Detailed clock schedule, break requests in the red region are postponed, until the end of the region.

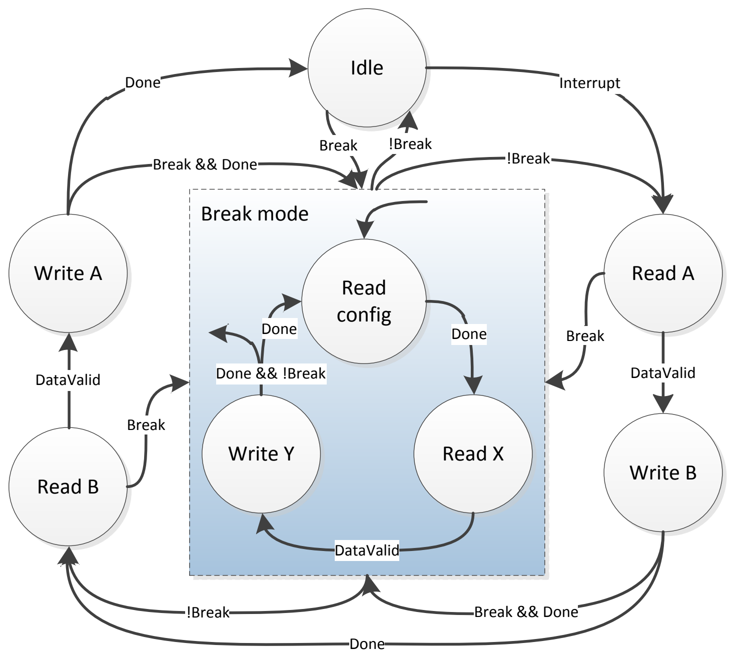


Figure 4.24: The arbiter of Figure 4.6 is extended with a break mode. The break mode can be entered from within the read state, after the write states or in the idle state. The break mode returns to the state it came from.

4.5 Hardware-level inspection

When debugging a problem the more relevant information is available the easier it is to localize the root-cause of a problem. When a design is stopped the interconnection system remains operational to be able to copy information from the subsystem under test. This information is used to determine the next steps in the debug process. A distinction is made between different types of stop conditions or breakpoints. The soft breakpoints are requested and applied when all communication is finished.

The second type of breakpoints are the hard breakpoints, these are introduced for hardware-level inspection and applied immediately.

Since there is a certain level of abstraction, when defining subsystem boundaries there are still process groups that communicate internally within a subsystem. These process groups share information, which is not available in the register interface. This information can be valuable, even when the creation of an individual subsystem would introduce too much communication overhead. Debugging these type of systems is done in two steps. The first step is by using the communication spy to locate in which area the problem lies. When the subsystem with the problem is found more observability at a known moment in time can be valuable. Internal blockrams can be filled with information during the entire run-time of the FPGA making this information not present in the state acquiring ring-buffers. During the hard breakpoints it is not possible to use the interconnection structure. The clock of the communication arbiter and the interconnection structure are both stopped as well.

It is desirable to find a way to introduce observability, while a hard breakpoint or condition is active. This can be an intrusive method, since the nature of a hard break condition is already intrusive.

In Xilinx FPGAs there exists a method to capture and read-back the state of the FPGA. This method cannot be used with Xilinx tooling and is only briefly described in the Virtex documentation and datasheets [13]. The read-back and capture method consists of different independent steps. The first step is capturing the active state of the FPGA. This means that the state of all switching logic e.g. LUTs and flip-flops is stored in the configuration registers. This state contains, thus the active signals at a particular moment in time. This capture technique is described in subsection 4.5.1. The second step is reading the data in these configuration registers. This read-back is performed by addressing all logic in the FPGA and reading it. The captured logic has a slightly different approach for read-back of blockrams, this is explained in subsection 4.5.2.

4.5.1 Capture FPGA state

The FPGA is configured by writing the binary data of the bit-stream in the configuration memory. As soon as the FPGA is released from shutdown this memory is read and the FPGA is initialized. When reading this memory without a capture unit the boot time configuration is returned. This information is used by the Xilinx configuration tools to verify the program action before releasing from shutdown. When the FPGA is released from shutdown with capture enabled then the configuration is constantly changing. Every clock-cycle the current state of each internal FPGA component is stored in the configuration registers.

In order to be able to capture the FPGA state vendor Xilinx provides a primitive or component called *CAPTURE_VIRTEX6* shown in Figure 4.25. If this block is supplied with a clock and an enable signal (CAP) then the FPGA its registers (flip-flops and latches) are captured into the configuration memory. The Look-Up Table (LUT) RAM, Shift Lookup Registers (SLR) and blockram blocks are not captured. Blockram memory only has two ports and cannot be accessed any other way and therefore can be read-back but not be captured. An asserted high CAP signal indicates that registers in the device

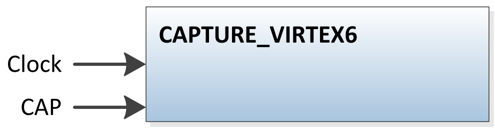


Figure 4.25: CAPTURE_VIRTEX6 primitive, when clock is supplied and CAP is high this module stores the active FPGA configuration in the configuration memory. As soon as CAP is low the last configuration remains in these registers.

Device	Configuration	Device total	Frame length	Overhead
LX240T	28464 frames 2305584 words	28488 frames	81 words	583 words

Table 4.3: Virtex-6 LX240T total number of configuration frames and sizes.

are captured at the next low to high clock transition.

The capture technique is used to be able to store state information of low-level internal logic at an exact moment in time. Reading this state information is not part of the capture technique and explained in subsection 4.5.2.

4.5.2 Read-back FPGA state

The configuration registers of the FPGA can be read using different interfaces like JTAG, Internal Configuration Access Port (ICAP) and SelectMAP. SelectMAP is the external implementation of ICAP. The interfaces for accessing the configuration memory in a similar way using configuration messages.

The Virtex-6 configuration memory is arranged in tiles consisting of frames [13]. These frames are the smallest addressable segments of the FPGA configuration. The configuration must always be done on whole configuration frames. The configuration of the Virtex-6 FPGA used in this project is shown in Table 4.3.

The configuration of the FPGA is done by sequentially writing or reading the FPGA bit-stream the configuration registers. The FPGA binary consists of a number of commands that need to be executed. These commands are grouped in two different packets. These packets are required in the read-back sequence since the same configuration registers are used. The type 2 packet must always follow a type 1 packet and has therefore no address. The lay-out of these header packets is shown in Table 4.4. The relevant subset of register address is included in the table. The Virtex-6 FPGA support more configuration addresses.

Using Xilinx tooling it is possible to generate a read-back command sequence file. This file contains a list of commands to read-back the FPGA configuration. The read-back data can be masked with a mask file also generated by the tooling. These files are used by the Xilinx tooling to verify the FPGA configuration in the device. The mask file hides all configuration fields subjected to change during execution of the code on the FPGA.

	Name	Bit	Remark
Type 1	Word Count	[10:0]	xxxxxxxxxxx, number of words to read or write
	Reserved	[12:11]	00
	Register Address	[26:13]	000000000xxxxx 00000000000001, Frame Address Register (FAR) 00000000000010, Frame Data Register In (FDRI), write only 00000000000011, Frame Data Register Out (FDRO), read only 00000000000100, Command Register (CMD),
	Opcode		00, NOOP 01, Read 10, Write 11, Reserved
	Header Type	[31:29]	001, for type 1 packet
Type 2	Word Count	[26:0]	xxxxxxxxxxxxxxxxxxxxxxxxxxx, commands
	Opcode	[28:27]	00, Reserved
	Header Type	[31:29]	010, for type 2 packet

Table 4.4: Type 1 and 2 packet header used for addressing the configuration registers of a Virtex-6 FPGA.

This mask file cannot be used in the research, since the goal is to read-back the variable part and not the static part. The read-back sequence file can not be used either, because it assumes the read-back commands are issued externally of the FPGA. The FPGA is put in shutdown first and in this sequence CRC checking is also enabled. Both methods corrupt the design since in this template the MicroBlaze and AXI interconnection bus must remain operational. The CRC cannot be used together with the capture primitive because the CRC is computed over the original configuration data. This original configuration CRC will not be equal to the constantly changing variable part. The command sequence used for read-back is shown in Table 4.6.

The original read-back sequence is altered in order to cope with these differences. This design also doesn't perform a full read-back of the FPGA, but only relevant parts for that moment in time. The addressing of the read-back logic is set via the FAR address. This frame address register is used to address all individual configuration frames in the FPGA. The layout of this register is shown in Table 4.5. The FAR address register is set via the messages shown in Table 4.4.

Name	Bit	Remark
Minor address	[6:0]	Select frame in a column
Column address	[14:7]	Select major column (e.g. a column of CLBs). Start at address 0 on the left and increase to the right
Row address	[19:15]	Select the current row. Address is increased from center to top, reset then center to bottom
TOP_B	20	0, top half of the FPGA 1, bottom half of the FPGA
Block type	[23:21]	000, Configurable Logic Blocks (CLBs) 001, Blockram contents 010, CLB configuration

Table 4.5: Description of the Frame Address Register (FAR) used for read-back

The FAR address can be used in two different ways for read-back. The first way is to supply the FAR address with the first frame address e.g. `0x0000 0000` and set the Frame Data Read Out (FDRO) register to the number of configuration frames available. Using this method the entire FPGA is read. The ICAP handles the internal incrementation of the register addresses. The second method is to determine in which column logic is located and write that address to the FAR register. The FDRO register should contain the number of frames in this column type plus a dummy frame. The second method is used to read-back values. The determination of locations is shown in Figure 4.26.

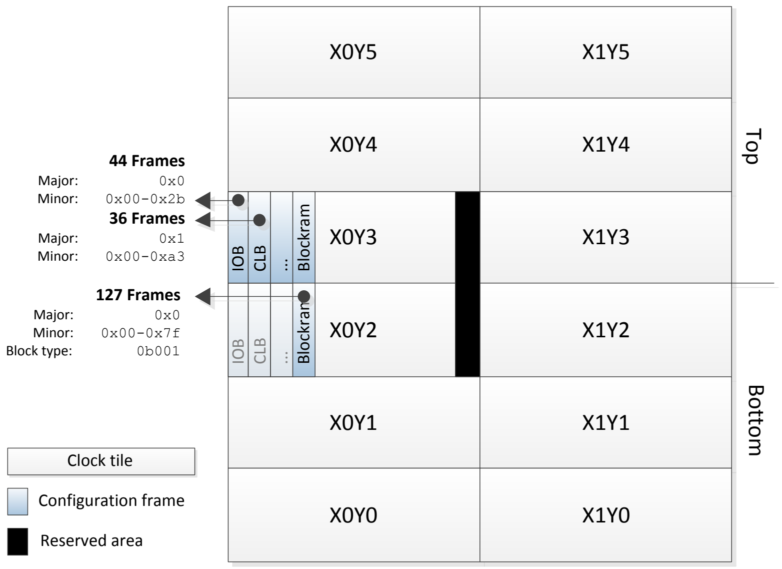


Figure 4.26: Example of how logic is addressed in the FPGA during a read-back.

The layout in Figure 4.26 also shows that there are different type of columns. These columns contain a different number of frames. The Input Output Block (IOB) columns contain 44 frames, while the Configurable Logic Block (CLB) columns have only 36 valid configuration frames. The addressing of blockram columns differs from other logic. If a blockram column is addressed with *block type* set to `0b000` only the interconnection configuration is read-back. In order to address the blockram data the *block type* must be set to `0b001`. The addressing of these columns also differ, since this type only consists of blockram columns.

The Xilinx logic allocation file (`.ll`) contains the relation between the physical location of logic in the FPGA and location in the read-back data stream. A part of this allocation file is shown in Table 4.7. The information from the allocation file must be manually parsed to find the location of a register. In chapter 8, future work suggestions are given to automate this process to increase usability. The NIFD [23] system shown in chapter 3 also parses this information.

Direction	Command	Value	Remark
Write	Packet::Dummy	`0xffffffff`	
	Packet::Sync	`0x000000bb`	
	Packet::Detect	`0x11220044`	
	2x Packet::Noop	`0x20000000`	
	Packet::Type1Header	`0x30008001`	Type 1, Write, CMD register, 1 word
	Packet::Rcrc	`0x00000007`	Register CRC, CRC configuration
	2x Packet::Noop	`0x20000000`	
	Packet::Type1Header	`0x30008001`	Type 1, Write, CMD register, 1 word
	Packet::Rcfg	`0x00000004`	
	3x Packet::Noop	`0x20000000`	
	Packet::Type1Header	`0x30002001`	Type 1, Write, FAR register, 1 word
	Packet::FAR	-	Table 4.5
	Packet::Type1Header	`0x28006000`	Type 1, Read, FDRO, 0 words
	Packet::Type2Header	`0x48000000`	$+((81+1) * FramesPerColumn)$
	32x Packet::Noop	`0x20000000`	Wait for the FPGA to become ready
Read	-		Read number of requested words
Write	Packet::Desync	`0x0000000d`	

Table 4.6: Read-back command sequence for reading a single FPGA column without shutdown sequence and disabled CRC.

Part of the Logic Allocation file of the design

```
Bit 40929826 0x0010941e 2146 Block=SLICE_X63Y73 Latch=AQ Net=LEDs_8Bits_TRI_O_1_OBUF
Bit 40929832 0x0010941e 2152 Block=SLICE_X62Y73 Latch=AMUX Net=MonitorLinkOut<3>
Bit 40929879 0x0010941e 2199 Block=SLICE_X63Y73 Latch=DMUX Net=MonitorLinkOut<4>
```

	Component	Value	Remark
Location	Offset	40929826 bit	
	Frame address	`0x0010941`	
	Frame offset	2146	
Information	Block	`SLICE_X63Y73`	Block in memory location
	Latch	AQ	Latch in memory location
	Net	`LEDs_8Bits_TRI_O_1_OBUF`	User net name

Table 4.7: Logic allocation file generated during FPGA binary generation contains the physical location of signals in logical blocks.

4.6 MicroBlaze interface

The debug and qualification template requires an interface to configure all individual components. During the pre-study phase [22] it is chosen to use an internal MicroBlaze processor running Embedded Linux. This processor and operating system enables the use of standard drivers for ICAP port and AXI slaves. The processor still requires user interface tooling that makes use of these device drivers. The MicroBlaze enables re-use of existing components and the entire solution can be tested within a single FPGA.

The supporting tools for this template are incorporated in the framework and implemented inside the FPGA. The base of the template is chosen in a way that the internal processor can be replaced by an external debug processor in the future. This processor can in theory connect to the debug framework via for example external AXI, JTAG or a another protocol e.g. UART. This project determines whether the debug applications can be used. The usability suggestion of how the system can be improved are added as future work chapter 8.

The interface system of the qualification and debug framework consists of two different parts both working together. The MicroBlaze system is a fully configurable soft-core processor. The design of this system is referred to as the MicroBlaze hardware and is described in subsection 4.6.1. The Linux kernel, included drivers and application software is considered software and explained in subsection 4.6.2.

4.6.1 MicroBlaze hardware

The MicroBlaze hardware consists of a base processor core which is configured to work with an AXI interconnect structure. The slaves that are connected to this interconnect can be found in Figure 4.27. The *Ethernet Slave* [46] and *UART Slave* [47] are used as user-interface. The *ICAP Slave* [48] is required to translate the read-back commands from the internal ICAP primitive to MicroBlaze memory. The *GPIO Slave* [49] functions as a 8-bit two way control and status interface with the monitor. The *External Slave* [50] components are blocks to attach AXI slaves which are not part of the Xilinx MicroBlaze project. This component takes care of leading the AXI signal to the external interface of the MicroBlaze.

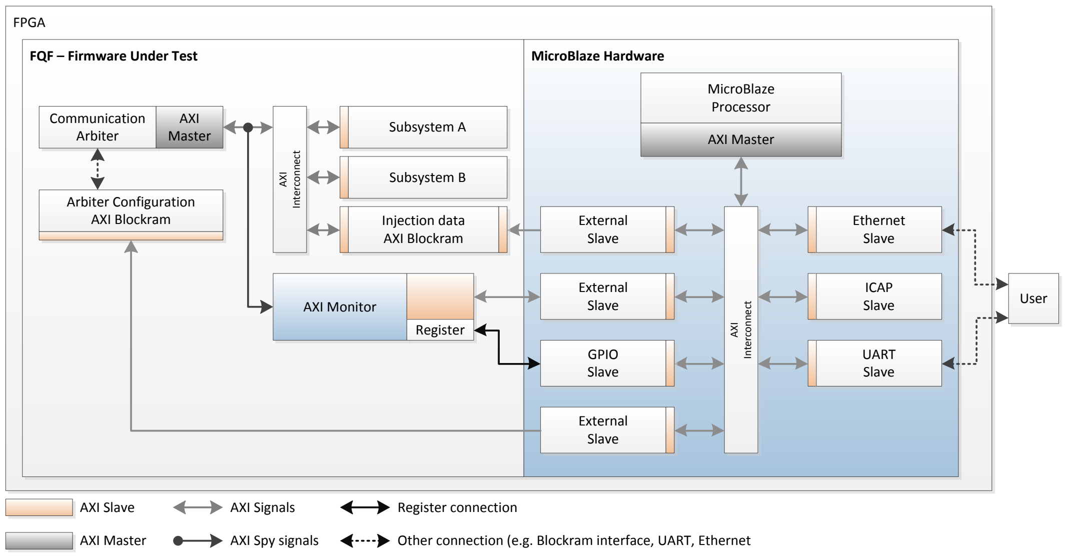


Figure 4.27: FFQF template design with all individual library components instantiated.

The external slave modules which are connected to the MicroBlaze design are explained in the sections above. The *Arbiter Configuration* slave is a block memory that contains the arbiter schedule. This configuration is read at the start of each communication cycle. This configuration can be overwritten by the MicroBlaze, to for instance enable an alternative schedule which copies data from an*Injection Unit* slave. This slave provides alternative data to be used by one of the subsystems.

The AXI Monitor is an AXI slave and AXI spy. The side connected to the MicroBlaze is the AXI

Tool	Description
ICAP interface	The ICAP application performs *read* and *write* commands on the ICAP file descriptor. This descriptor is obtained by opening the $/dev/icap0$ device node. This is a direct link to the internal driver [51]. The write commands are the commands given in Table 4.6.
Monitor Control	The monitor control application handles the AXI writes to the configuration blockram based in its input. The status of the monitor unit is printed using internal GPIO reads.

Table 4.8: Linux interface tools used to communicate with the qualification and debug template.

slave. This slave write the internal configuration memory on writes and returns the acquisition data on reads. The register connection is the bidirectional 8bit connection handled by the 8bit GPIO slave.

The MicroBlaze processor can run any frequency. The entire interconnection system and all attached slaves both internal and external run at 80MHz. The standard Xilinx interconnection system is able to handle clock region crossing but this is not required and not tested.

4.6.2 MicroBlaze software

The software running on the MicroBlaze processor is an Embedded Linux version especially compiled for the MicroBlaze. The code is supplied by Xilinx via `http://git.xilinx.com`. This package includes cross-compilers, a base Linux configuration and drivers code.

The Xilinx supplied drivers do not support the Xilinx Virtex-6 processor. The drivers support up-to the Virtex-5 series. The driver supports Virtex-6 devices after modifying the configuration register lay-out according to the datasheet. The memory map drivers are able to handle memory mapped read and write commands to AXI slaves. Single memory reads and writes can be used to access the status and control registers via the GPIO driver. The custom tools which are required for this template are a user-space applications which make use of the Linux drivers shown in Table 4.8.

Chapter 5

Template evaluation

The introduction of the FFQF template improves observability and controllability. These benefits come at a price of area overhead, increasing latencies and less throughput. A model is defined in this evaluation which quantizes the overhead. The accepted overhead of this system cannot be determined in general. It depends on the project or design on which the template is applied. The requirements for the template are modularity and minimal area usage. The power consumption of the template is not taken into account and outside the scope of this research. The assumption can be made that if the area increase is minimal the power increase will be minimal as well, but research has not been done here.

The evaluation of the MicroBlaze infrastructure is given in section 5.1. The communication infrastructure that scales with the number of slaves is evaluated in section 5.2. The modules or units which make use of the template its infrastructure are analysed in section 5.3. The conclusion of this chapter is given in section 5.4. The area overhead is expressed in logic cells, these cells are explained in section 2.1. The area measurements of the template are performed with *keep hierarchy* enabled during synthesis. This disables optimizations between hierarchical blocks and gives worst-case results. The area usage is extracted from the map report.

5.1 MicroBlaze infrastructure

The MicroBlaze processor is used as an interface to the system. The part of the design that is considered as MicroBlaze processor is shown in Figure 5.1. The resources used by this processor are shown in Table 5.1. To indicate the impact of this interfacing processor the number of available resources is also added to this table. The MicroBlaze processor is a module in the system and can be replaced by for example a Zynq ARM core or another FPGA. This module is not required to be physically in the same FPGA. Removing the interface processor from the template is suggested as future work in chapter 8.

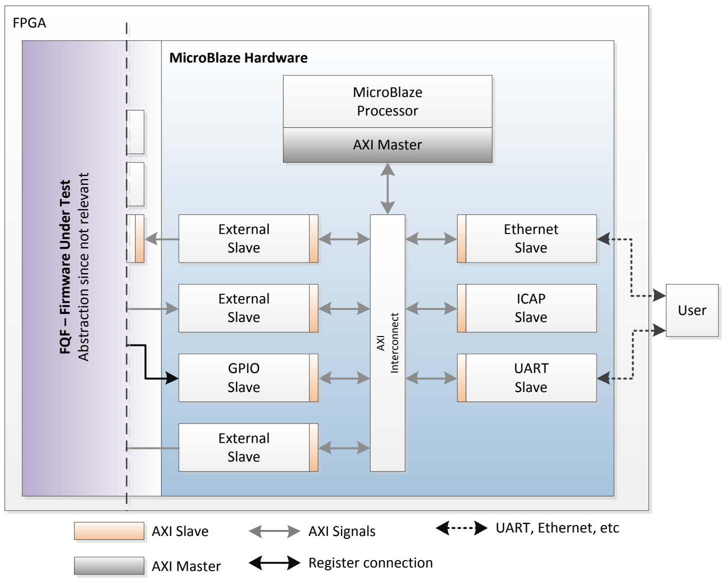


Figure 5.1: Graphical representation of the resource usage analysed in Table 5.1, this figure is derived from Figure 4.27.

		Virtex-6 LX240T		
Logic cell		**Used**	**Available**	**Usage**
Slice registers		18.830	301.440	6%
Slice Look-up tables	Total	17.745	150.720	11%
	Logic	14.142		9%
	Memory	2.555	58.400	4%
Slices		7.813	37.680	20%
Blockram	36	33	416	7%
	18	2	832	1%
DSP48E1		3	768	1%

Table 5.1: Analysis of the resource usage of the MicroBlaze processor. The processor is used to interface with the debug system. This interface can also be implemented on an external device.

Table 5.1 depicts the resource usage of the MicroBlaze. This shows that the MicroBlaze covers a large part of the FPGA resources. The processor uses about $\frac{1}{5}$ of the slices of the FPGA. The area occupied by the MicroBlaze is constant. The focus of this project was not on the MicroBlaze design and therefore no optimizations have been made to this processor. The processor behaves purely as an interface to the debug and qualification system.

5.2 Communication structure

The AXI communication structure is a $32\mathrm{bit}$ interface that replaces direct parallel connections. The introduction of the communication interface introduces overhead. The interconnection structure consisting of a large crossbar, a number of slaves, and a communication arbiter as shown in Figure 5.2.

The functionality is qualified using test-benches and firmware execution on the FPGA. This qualification is not part of the evaluation process. The communication structure evaluation measures the footprint of the communication system and comparing this with the area of direct connections. The resources required for the communication structure are shown in Table 5.2. A multiple master configuration is not taken into account since the interconnect implementation at this moment does not support more than a single master. The AXI protocol supports multiple masters with data prioritization.

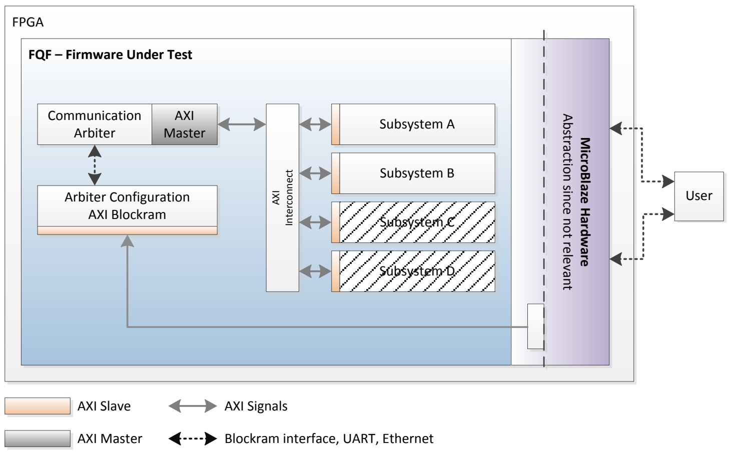


Figure 5.2: Graphical representation of the resource usage analysed in Table 5.3, this figures is derived from Figure 4.27

The AXI slave implementation is generic. This slave is instantiated for each subsystem that is added to the communication system. The required resources for this slave are equal for each instantiation. The subsystem itself requires some logic to connect the parallel connections to either a blockram or custom register interface. This custom register interface is implemented in distributed logic.

The AXI arbiter and master implementation are also generic. The arbiter copies data from slave to slave. Since the communication schedule is stored in a blockram the resource usage will not increase with more slaves. The schedule is stored in the 32bit blockram registers. The first line in the memory contains a 24bit from address and 8bit length to copy. The second line holds a 24bit destination address and 8bit length. This continues until `0xDEADBEEF` is read. The value `0xDEADBEEF` is arbitrarily chosen.

The information in Table 5.2 only shows logic required by the generic components. The interconnection system is the basis of the template and required for the methodology to apply. The scalability of the entire interconnection system is shown in Table 5.3. The equations derived from this data are shown in Equation (5.1).

The minimal area introduced with the FFQF template can be calculated using Equation (5.1). This is only the area introduced by the communication architecture. These numbers are determined with the data of Table 5.3 and Table 5.2. Due to optimizations in the synthesis process the equations will never give an exact result but more an approximation. The slices are approximated with a 2^{nd} order

Logic cell	AXI Slave	AXI Blockram R+W Slave	AXI Master	Arbiter
Slices	59	$45 + 17 = 62$	31	43
Slice registers	148	$103 + 65 = 168$	91	142
LUTs	98	$65 + 38 = 103$	21	31
LUT RAM	0	$0 + 0 = 0$	0	0
BRAM	0	$1 + 1 = 2$	0	1
DSP48E1	0	$0 + 0 = 0$	0	0

Table 5.2: Area usage of the generic AXI master and slave implementations. This does not include the register interface which translates parallel registers to an AXI memory map. The AXI blockram R slave is a consumer and W slave is a producer.

Logic cell	Usage for 1 Master and 2 Slaves	4 Slaves	6 Slaves	8 Slaves
Slices	243	284	418	667
Slice registers	482	812	1140	1468
LUTs	374	633	945	1190
LUT RAM	0	0	0	0
BRAM	4	8	12	16
DSP48E1	0	0	0	0

Table 5.3: The area usage of the interconnection system with different configurations. All components except for the slices scale linear.

polynomial function in MATLAB. The rest of the area has a linear relation.

$$\begin{aligned}
A_s &= \text{Area\_in\_slices} \\
A_s(slaves) &= 13 * slaves^2 - 59.7 * slaves + 311.5 \quad &, O(slaves^2) \\
A_r &= \text{Area\_in\_slice\_registers} \\
A_r(slaves) &= 164.5 * slaves + 153 &, O(slaves) \\
A_l &= \text{Area\_in\_LUTs} \\
A_l(slaves) &= 136 * slaves + 62 &, O(slaves) \\
A_b &= \text{Area\_in\_blockrams} \\
A_b(slaves) &= slaves * 2 &, O(slaves)
\end{aligned} \tag{5.1}$$

Besides area overhead introduced by the interconnect there is also communication overhead. This overhead is the time it takes to transfer a single value between subsystems. When using direct connections, data is ready for the receiver in the cycle it is computed. This cycle the information can be used by the receiving side. In this AXI design the data is provided on request of the master. The arbiter initiates a data request on a certain address via the master. This address is via the AXI interconnect routed to the correct slave. The slave copies its internal data to the master via the interconnect. The master supplies the data to the arbiter which copies this to another slave. These signals are again routed via the AXI interconnect. The latency of the interconnection system is shown in Section 5.2.1. The throughput is given in Section 5.2.2.

5.2.1 Latency

The latency of the designs without with direct connections is a single clock cycle for each connection. The data is set and can be read the next clock cycle by the receiving side. The introduction of the bus architecture increases the latency for data communicated between subsystems. In the latency calculation the assumption is taken into account that all data is copied from a source subsystem to a destination subsystem. The latency depends on the transaction burst length.

The AXI copy transactions initiated by the arbiter consist of reading data at *slave a* and writing it to *slave b*. The AXI read transaction starts with an initialization. During initialization the address and burst length are supplied. On acceptance of the read the data is provided. During the setup up of a write command the first 32bit word is supplied in parallel with the initialization. The write starts after receiving the first 32bit word from *slave a*.

The latency is the combination of initialization time for a transaction, the time it takes to transfer a single burst, and the time it takes to receive the status response. There is a minimum and maximum latency since one cycle delay is accepted by de slave between data ready and valid. The latency figures are given in Table 5.4.

Transfer type	**Latency Initialization+burst_size+finish**
Minimal latency	
Read	$6 + n + 1$ cycles
Write	$4 + n + 1$ cycles
Copy (read + write pipelined)	$14 + n + 1$ cycles
Maximum latency, slave waits a cycle with accepting	
Read	$7 + n + 1$ cycles
Write	$5 + n + 1$ cycles
Copy (read + write pipelined)	$16 + n + 1$ cycles

Table 5.4: The latencies introduced by the AXI interconnection system, where n is the burst_size. This holds for the communication initiated by the communication arbiter.

5.2.2 Throughput

The throughput of the AXI interconnection system is limited. The throughput is maximized by the chosen data width and the burst size. This choice is based on design decisions and the AXI specification [38]. The data width chosen for this template is 32bit, the Xilinx components also support sizes of 64bit and 128bit. The AXI standard supports burst transfer widths to a maximum of 1024bit. The maximum burst size allowed by the AXI specification is 16. The maximum amount of data that can be copied in a single burst is $16 * 32\text{bit} = 512\text{bit} = 128\text{byte}$. The required burst length depend on the number of registers available in each subsystem. The clock frequency of the bus can be chosen arbitrarily. In order to avoid clock domain crossings the interconnect can be clocked at the same frequency as the rest of the design. In the throughput calculations a bus frequency of 80MHz is assumed. This frequency is arbitrarily chosen based on existing designs.

The throughput is calculated by taking the communication time of the master into account. Since the communication channel is dedicated for all internal communication the maximum throughput is equal to the achieved throughput. The bus transactions are never interrupted. The copy action is pipelined since the write can be initiated when the first item is read. The read and write channels are completely independent.

The throughput figures are gathered from a behavioural simulation of the design. The calculated throughput is given in Equation (5.2). The corresponding equations are shown in Table 5.5. The design has also been tested on the actual hardware to makes sure no timing problems exist.

Burst size	**Burst length**	**Throughput**
Read		
1	$7+1$ cycles	$\frac{8}{80\text{MHz}} * 32\text{bit} = 3.2\text{Mbps}$
2	$8+1$ cycles	$\frac{9}{80\text{MHz}} * 64\text{bit} = 7.2\text{Mbps}$
4	$10+1$ cycles	$\frac{11}{80\text{MHz}} * 128\text{bit} = 17.6\text{Mbps}$
8	$14+1$ cycles	$\frac{15}{80\text{MHz}} * 256\text{bit} = 48\text{Mbps}$
16	$22+1$ cycles	$\frac{23}{80\text{MHz}} * 512\text{bit} = 147.2\text{Mbps}$
Write		
1	$5+1$ cycles	$\frac{6}{80\text{MHz}} * 32\text{bit} = 2.4\text{Mbps}$
2	$6+1$ cycles	$\frac{7}{80\text{MHz}} * 64\text{bit} = 5.6\text{Mbps}$
4	$8+1$ cycles	$\frac{9}{80\text{MHz}} * 128\text{bit} = 14.4\text{Mbps}$
8	$12+1$ cycles	$\frac{13}{80\text{MHz}} * 256\text{bit} = 41.6\text{Mbps}$
16	$20+1$ cycles	$\frac{21}{80\text{MHz}} * 512\text{bit} = 134.4\text{Mbps}$
Copy (read + write pipelined)		
1	$14+1$ cycles	$\frac{15}{80\text{MHz}} * 32\text{bit} = 6.0\text{Mbps}$
2	$15+1$ cycles	$\frac{16}{80\text{MHz}} * 64\text{bit} = 12.8\text{Mbps}$
4	$17+1$ cycles	$\frac{18}{80\text{MHz}} * 128\text{bit} = 28.8\text{Mbps}$
8	$21+1$ cycles	$\frac{22}{80\text{MHz}} * 256\text{bit} = 70.4\text{Mbps}$
16	$29+1$ cycles	$\frac{30}{80\text{MHz}} * 512\text{bit} = 192.0\text{Mbps}$

Table 5.5: Maximum throughput of the AXI interconnection system with a static schedule hard-coded into VHDL.

$$
\begin{aligned}
n &= \text{burst\_size} \\
\sigma &= \text{throughput} \\
\sigma(n) &= \frac{t_{start} + n + t_{end}}{f_{bus}} \\
\sigma_{read\_max}(n) &= \frac{6 + n + 1}{80\text{MHz}} \\
\sigma_{write\_max}(n) &= \frac{4 + n + 1}{80\text{MHz}} \\
\sigma_{copy\_max}(n) &= \frac{13 + n + 1}{80\text{MHz}}
\end{aligned}
\tag{5.2}
$$

5.3 AXI monitor and injection

The AXI monitor block is the debugging core embedded into the design. This debugger has constant area requirements which do not change with the number of slaves. If multiple masters need to be investigated multiple monitor components need to be instantiated and connected. Each master requires its own monitor unit. This research only takes a design with a single master into account. Figure 5.3 shows the design part which includes the monitor unit.

The area occupied by the acquisition blockram depends on the number of registers which are monitored. The acquisition blockram is used to store traces of data communicated between subsystems. The more registers need to be traced the more blockram blocks are required. The number of blockram blocks scales linear with the number of registers.

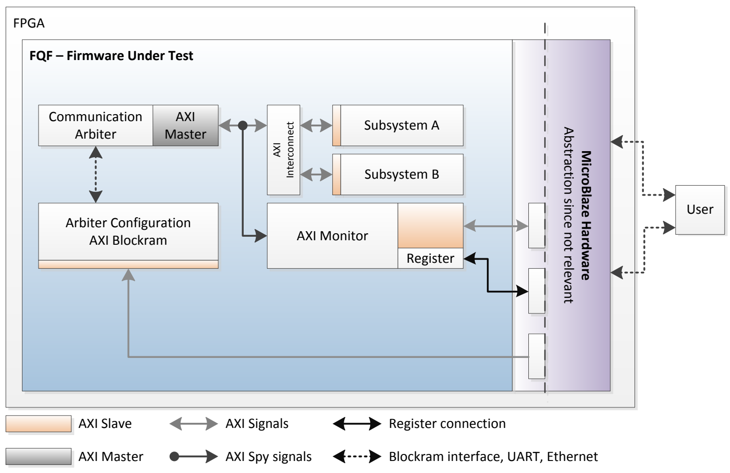


Figure 5.3: Graphical representation of the resource usage analysed in Table 5.6, this figures is derived from Figure 4.27

Logic cell	AXI Monitor	4 Acquisition blockrams
Slices	191	47
Slice registers	376	113
LUTs	288	110
LUT RAM	0	0
BRAM	2	0
DSP48E1	0	0
CAPTURE_V6	1	0

Table 5.6: Area usage of the monitor and debugging blocks.

5.4 Conclusion

The base of the template is the AXI interconnection system. This system introduces additional area overhead compared to the original designs with direct connections. The overhead is only limited compared to the area available in the FPGA as shown in Equation (5.3). The calculation only holds for the Virtex-6 LX240T FPGA.

$$\begin{aligned}
0.00658\% &= \frac{A_s(4)}{\text{Available\_slices}} &&= \frac{284}{37680} \\
0.01770\% &= \frac{A_s(8)}{\text{Available\_slices}} &&= \frac{667}{37680} \\
0.00269\% &= \frac{A_r(4)}{\text{Available\_slice\_registers}} &&= \frac{812}{301440} \\
0.00487\% &= \frac{A_r(8)}{\text{Available\_slice\_registers}} &&= \frac{1468}{301440} \\
0.00420\% &= \frac{A_l(4)}{\text{Available\_LUTs}} &&= \frac{633}{150720} \\
0.00790\% &= \frac{A_l(8)}{\text{Available\_LUTs}} &&= \frac{1190}{150720}
\end{aligned} \tag{5.3}$$

The monitor blocks can be added during qualification of the design and remain in the design since they are non-intrusive. When disabling the breakpoints the system has no effect on the design existing in the FPGA. If the framework is used in an area critical design the area impact of the MicroBlaze might be too much. The processor is used in this framework as a proof-of-concept to enable reuse of existing software components.

Chapter 6

Case-study

The template is applied to an existing design. This case-study is used to check the expected impact for correctness. The designs of an electrical current controller is used as a base. This design consists of three subsystems. The *communication interface*, the *current controller*, and the *module controller*. The communication interface is responsible for periodically communicating with an external host. The status information of the current controller is provided by the device. The device receives desired set-point information and other settings from the external host. The current controller measures current and adjusts the amplifier outputs accordingly. The module controller is responsible for internal status and error handling. The design setup and requirements are denoted in section 6.1. The template is applied to this design in section 6.2, the impact is expressed in latency increase, area overhead and communication throughput differences. The usage of the framework is shown in section 6.3.

6.1 Electrical current controller

The electrical current controller is responsible for controlling a motor via an amplifier. The desired electrical current set-point is received via the communication interface. This set-point is used as input for the controller. The controller measures the actual current flowing through the system and compares this with the desired set-point. The output PWM signal to the amplifier is adjusted according to the difference in electrical current. The effect on the actual electrical current is measured and this process is repeated while the system is running. Figure 6.1 shows the setup of the system and design.

The requirements which are relevant for the application of the template are denoted in Table 6.1. The requirements have impact on the communication latency. The relevant communication latency is between the communication interface and the current controller. The communication interface is supplied with data updated cyclically every 10kHz. The data consists of real-time data and periodic data. The system is required to respond quickly to changes initiated by the host system. The other

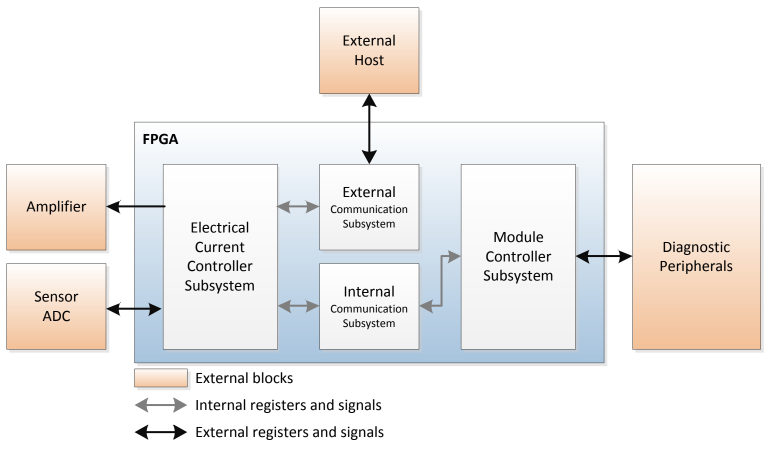

External Host
FPGA
Amplifier
Electrical Current Controller Subsystem
External Communication Subsystem
Module Controller Subsystem
Diagnostic Peripherals
Sensor ADC
Internal Communication Subsystem
External blocks
Internal registers and signals
External registers and signals

Figure 6.1: Overview of the electrical current controller is on which the template is applied, can also be found in Figure 1.5.

requirements have effect on the speed of the controller.

Requirement	**Value**
The time between receiving real-time data from the host and sending it to the current controller subsystem.	$< 500\text{ns}$
The frequency of the current control loop	375kHz
Communication packet frequency	10kHz
The system clock frequency	80MHz

Table 6.1: The requirements that can have an effect on the system, when the infrastructure is changed into the FFQF debugging template.

The current controller and communication interface subsystems are connected directly. These parallel connections are investigated and grouped in Table 6.2 and Table 6.3. These values need to be transferred via the AXI bus, when the template is applied to this design.

Record	**Size**	**Remark**
ControllerFlags	41bit	Enable, data valid signals and desired set-point value
ControllerParameters	80bit	Parameters setting for example controller gain
CommunicationStatus	11bit	General error and timeout flags
TestParameters	2bit	Disable warnings during production tests
Total	134bit	Sum all of the above

Table 6.2: Data communicated from the communication interface to the current controller.

Record	**Size**	**Remark**
ControllerStatus	25bit	Current state, output current, etc.
EepromData	76bit	Serial numbers, version information
MonitorData	48bit	Current power, voltage info
Total	149bit	Sum all of the above

Table 6.3: Data communicated from the current controller to the host interface.

The area occupied by this design is shown in Table 6.4. This area is expressed in logic cells. The logic cells are explained in section 2.1. The logic increase in this base design is the overhead introduced by the FFQF template.

Logic cell	**Module Controller**	**Communication Interface**	**Current Controller**	**Total**
Slices	227	741	690	1658
Slice registers	655	1631	1961	4247
LUTs	488	1468	995	2951
LUT RAM	1	3	12	16
BRAM	0	4	1	5
DSP48E1	1	0	15	16

Table 6.4: The area of the current controller design is used by the communication interface and controller process.

6.2 FFQF electrical current controller

The original design consists of hierarchical processes communicating via parallel registers. The first step is to determine at what level subsystems, process groups and processes are located. The *current controller* and *communication interface* are both completing an independent high-level function. The *current controller* and *communication interface* are therefore chosen as subsystems. The processes at a lower hierarchical level are automatically elected process groups.

There are two processes communicating with each other. The system input given by the user is transfered via the *host interface* to the *current controller*. The status information of the *current controller* is sent to the user via the *host interface*.

The direct connections in Table 6.2 and Table 6.3 are replaced with an address based register interface. In order to minimize communication latency the parallel connections are combined into single 32bit lines. The less 32bit registers the smaller the communication bursts. The communication interface and current controller both have input and output registers. The registers are either readable or writeable. The register interface of the FFQF enabled current controller is shown in Table 6.5.

Since there is only a limited number of 32bit registers the AXI slave with custom register interface is used. This slave requires a custom implementation converting parallel signals to registers.

The data from the communication interface to the current controller is split in order to meet the $< 500\text{ns}$ requirement. The AXI communication arbiter is triggered by the ready signal of the real-time data. This signal is provided by the communication interface. This trigger initiates a copy action with a burst length of 1, copying only the real-time data. The worst-case latency is $16 + 1 + 1 = 18\text{cycles}$. With a bus frequency of 80MHz the time to communicate this data is $\frac{18}{80\text{MHz}} = 225\text{ns}$. This is fast enough to meet the requirement of 500ns. The entire communication schedule is completed in $3 * (16 + 1) + 1 + 4 + 6 = 62\text{cycles}$. This takes a total time of $\frac{62}{80\text{MHz}} = 775\text{ns}$. This schedule fits in the $10\text{kHz} = 200\mu\text{s}$ communication cycle and the $375\text{kHz} = 2.67\mu\text{s}$ control loop cycle.

Mnemonic	**Offset**	**Bit**	**Name**	**Remark**
RealtimeData	`0x0000`	`27:0`	SetPoint	Data needs to be transferred through the system in $< 500\text{ns}$
		28	EnableDebugSetPoint	
		29	ResetAxis	
		30	EnableAxis	
		31	Reserved	
CurrentControlB	`0x0004`	`15:0`	CurrentControlB0	
		`31:16`	CurrentControlB1	
Addition fields	`0x0008-` `0x0014`	-	-	CurrentControlGain, LinkStatus, TestParameters
	`0x1000` `0x1018`	-	-	SerialNumber, IOut, StateInformation, etc.

Table 6.5: The register interface between the current controller and the communication interface. The `0x000` range is used for read registers while the `0x1000` range is used for writeable registers.

Logic cell	**Module Controller**	**Communication Interface**	**Current Controller**	**Total**	**Increase**
Slices	322	$765 + 93 = 858$	$712 + 125 = 837$	2017	$\frac{2017}{1658} \approx 1.22$
Slice registers	682	$1582 + 327 = 1909$	$2034 + 285 = 2319$	4919	$\frac{4919}{4247} \approx 1.16$
LUTs	635	$1481 + 124 = 1605$	$1007 + 281 = 1288$	3528	$\frac{3528}{2951} \approx 1.20$
LUT RAM	1	3	12	16	equal
BRAM	0	4	1	5	equal
DSP48E1	1	0	15	17	equal

Table 6.6: The area of the current controller design is used by the communication interface and controller process. The increase in area is compared to the design without FFQF in Table 6.4.

6.3 FFQF usage

The added value of additional observability and controllability is difficult quantize. This section shows how the techniques can be used. A designer or tester is the biggest factor in the search for a problem. The tools try to make his work easier.

The use of breakpoints is shown in Figure 6.2. The static communication schedule is executed by the arbiter. The arbiter receives a break requires, completes it current ongoing copy action and accepts the break. During the break condition the alternative communication schedule is executed. This is also the moment where the buffers can be read for off-line analysis. Next the execution is continued again by releasing the break request. The alternative communication action is completed and normal execution is continued.

The ring-buffers storing the execution trace to until the break condition are retrieved by reading the blockrams in the MicroBlaze. There is a kernel module which handles this read feature via the AXI interface. Using the command `insmod blockram.ko Addr=0xc8040000 Len=0x1000 R=1 > buffer0.trace`. The buffer located at `0xc8040000` is stored into a text-file. This text-file can be processed by the offline test-bench.

The hardware registers can be inspected if it is not important to continue execution. The Linux ICAP

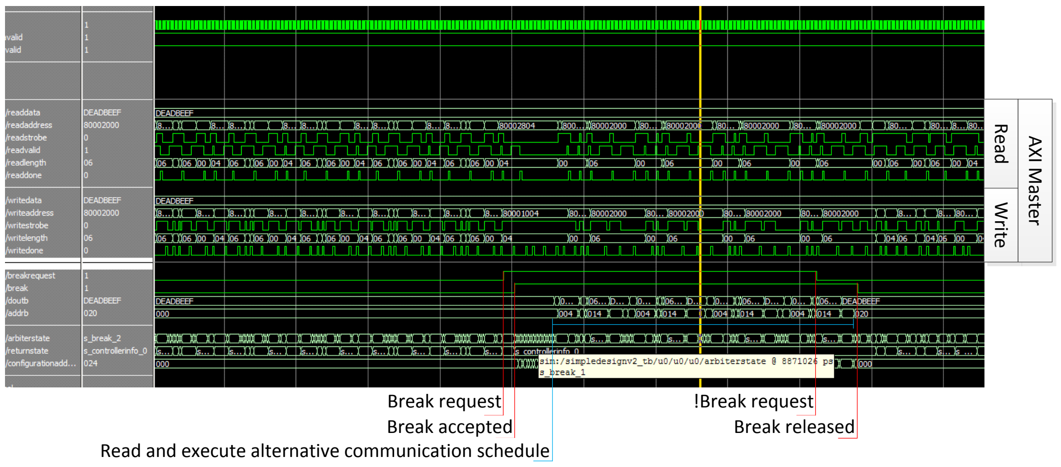


Figure 6.2: The template behaviour on a break request. During the break the alternative communication schedule is executed.

driver is accessed by using the `ICAP_interface` tool. This requires Xilinx Plan Ahead to determine the exact location of a signal. The this location is provided to the `ICAP_interface 'Block' 'Top' 'Row' 'Major' 'Size' > readback` application. Using the `.ll` Logic Allocation file the exact location is value of the signal is determined. It is not possible to continue from this breakpoint since the state of the other subsystems is unknown. It is possible that they have continued for an addition number of clock-cycles. If all subsystems and the interconnection structure are attached to the gated clock it is possible to continue. This choice is left to the engineer using the template.

Chapter 7

Conclusion

The **F**PGA **F**irmware **Q**ualification **F**ramework (FFQF) is presented in this thesis. The framework is a modular set of design blocks combined into a template. This template allows to instantiate its library components. The provided methodology requires a base design that can be setup using the following steps:

- Decouple the design into individual subsystems at an arbitrary level. The level at which subsystem boundaries are defined affects the communication overhead and observability. Higher observability introduces more communication overhead.
- Attach a generic AXI slave from the template to each subsystem. These slaves connect via a custom register interface to the subsystems. This adapter between the slave and register interface depends on the subsystem's parallel signal connections. The parallel signals are combined into a memory mapped register interface.
- The AXI slaves need to be connected to the AXI interconnect available in the template. The AXI interconnect is connected to the generic AXI master and communication arbiter.
- The communication schedule can be programmed into the communication arbiter provided by the template. The arbiter processes each line in the configuration memory in sequential order.
- Connect the debug slaves to a separate AXI interconnection system. This system is used to interface the debugging blocks. The MicroBlaze processor provided in this template library can be used.

When the design conforms to this base design it is possible to increase the observability and controllability. The communication centric observability is enabled with an AXI bus monitor. This module is able to trigger acquisition on data or address matches. The data is stored in acquisition buffers which can be read via another AXI master. The interface provided by the template is a MicroBlaze processor embedded into the FPGA.

Besides observing the communicated data with the AXI monitor it is also possible to store execution traces. The execution traces are stored in ring-buffers connected to the input and output registers in the register interface. The execution traces can be used to off-line repeat the sequence leading to a condition. The buffers store each register value on every clock cycle.

The controllability is introduced with breakpoints. There are different types of breakpoints. The communication centric breakpoints and the hard breakpoints. The communication centric breakpoints stop the clock of a subsystem but leave the interconnect system operational. The interconnection system can be used to read the values from the stopped slave. These type of breakpoints are postponed until active communication is completed. This enables to continue execution afterwards. The hard breakpoints stop the slave immediately. This enables the highest possible observability at hardware register level. That is viewing the values of all flip-flops, latches and blockram contents in the FPGA.

The injection of test vectors in a subsystem is done by attaching a buffer slave to the interconnection structure. This buffer can be filled externally via the AXI interface of the MicroBlaze processor. The arbiter transfers the test vectors in each communication cycle to the slave. The monitor unit is able to monitor the output of the subsystem.

The basic interconnection system requires less than 0.1% of the resources of an LX240T FPGA. This system consists of two AXI slaves, the AXI interconnect, an AXI master and a communication arbiter. The interconnection system scales linear for all resources except slices with the number of slaves. The slices scale quadratic with the number of slaves. The case-study shows that the impact on relatively small subsystems is large. The area required increase with the interconnection system is approximately 20%.

The tendency in the Xilinx FPGA roadmap shows that FPGA area is getting cheaper [52]. The available logic inside the FPGA increases and designs get more complex. The area overhead of debugging frameworks will become less important in the future.

The FFQF methodology and framework is usable but still requires a lot of handcraft. The framework lacks tools that increase usability. The framework misses a easy-to-use graphical user interface to configure the monitoring unit. The status information of the monitoring unit has to be read back manually. The parsing of read-back information at hardware register level state must be manually parsed. A parser for this information is proposed in chapter 8. The resources used by the MicroBlaze interface processor are also high. This processor currently requires 20% of the LX240T its slices. This processor can in theory be placed in a separate debugging module, this is also proposed in chapter 8.

Chapter 8

Future work

The debug framework which is designed during this research can be used to debug a design that setup according to the specification. There are still tools missing to increase usability. These tools can be for instance an intelligent parser that analyses the reports and files generated during synthesis. These reports contain the locations of all logic in the FPGA. It would be nice to provide this parser with a net-name and the synthesis files that it returns the command sequence to read-back this net. The same holds for reading the contents of blockram blocks. The two-way effect would introduce the possibility to read-back an FPGA location, and then give the net or logic block on that location. This can be helpful when comparing two read-backs with a single clock-cycle in between. This would shown the signals changed during this clock-cycle. This tool would significantly increase the usability. The current available work that is studied is able to find nets in a full FPGA read-back. This sequence is illustrated in Figure 8.1.

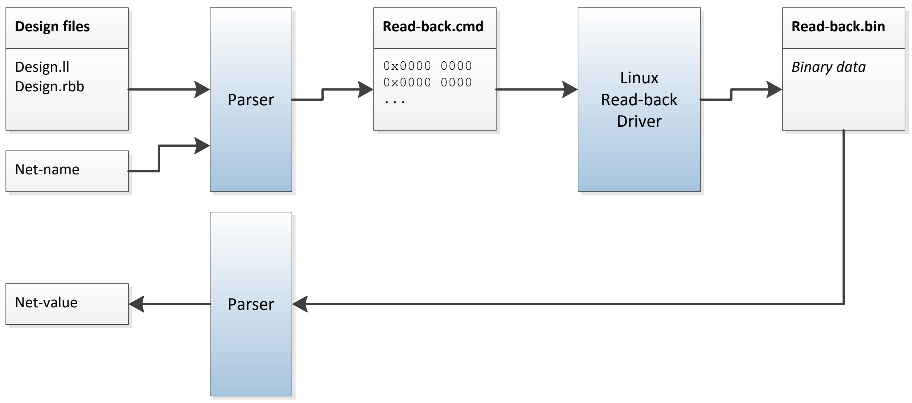
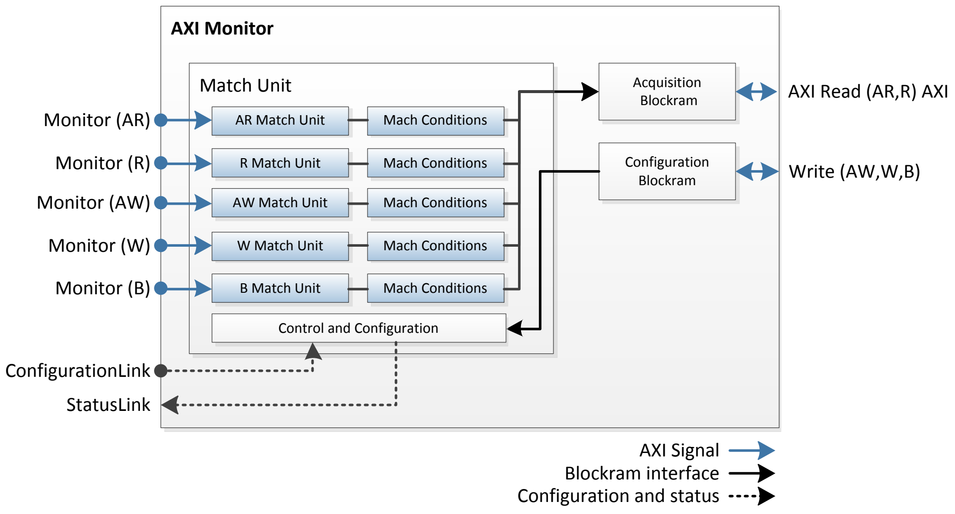


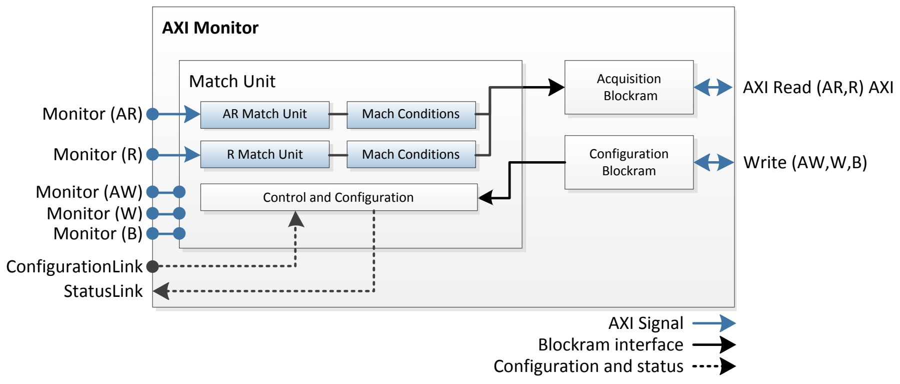
Figure 8.1: A two way data parser which can determine logic locations and determine current values based on read-back data

During the pre-study phase the possibility to reconfigure parts of the is investigated. This path is not researched any further since it does not introduce a profit in design time due to tooling limitations.

This reconfiguration could be used to make the monitor unit more flexible. Currently a number of signals enter the monitoring unit. The design of this unit get quite large. It implements monitors on each different interface. When picking a smaller footprint and programming this monitor only with sub monitors on the required signals the overhead of the system decreases. The system is only able to monitor and therefore this cannot have a negative effect on the rest of the design. In the future this gets more easy to use because partial reconfiguration techniques are improved with every iteration.



(a) Full Monitor



(b) Reconfigurable Monitor

Figure 8.2: The full monitor has area overhead when only monitoring limited signals, a reconfigurable monitor with smaller footprint can fix this

In order to make the system easier to introduce in future designs a lower resource consuming interface can be designed. The MicroBlaze system is very resource intensive. The MicroBlaze design can be implemented on an external FPGA. This external FPGA communicates via for example a `UART` interface with the internal blocks for configuration and data extraction. The monitor is required to remain inside the *system under test* design due to pin requirements. This system is denoted in Figure 8.3. The AXI interconnection system requires already 64 pins for only the read and write data channels in a $32\mathrm{bit}$ interface. This does not include any AXI control signals, when adding these signal it reaches approximately 178 pins.

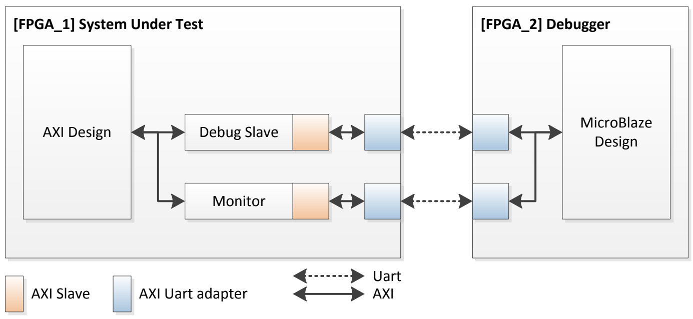


Figure 8.3: A type of communication abstraction when placing the MicroBlaze on another FPGA. This reduces the area requirements of the design in system under test. The downside is that this slows down the communication due to limited bandwidth.

List of Figures

List of Tables

Bibliography

[1] X. Inc., *Virtex-6 FPGAs Product Table*, 2010.

[2] iABG, *V-Modell 97 - komplett im Format Word*, 2006. http://v-modell.iabg.de.

[3] Lattice, "Behavioral modeling in vhdl simulations." Conference Presentation, February 1999.

[4] M. Graphics, *ModelSim® User's Manual Software Version 6.6d*, 2010.

[5] K. Arshak, E. Jafer, and C. Ibala, "Testing fpga based digital system using xilinx chipscope logic analyzer," in *Electronics Technology, 2006. ISSE '06. 29th International Spring Seminar on*, pp. 355 – 360, May 2006.

[6] X. Inc., *Virtex-6 FPGA Configurable Logic Block*, 2009. UG364.

[7] X. Inc., *Virtex-6 FPGA Memory Resources*, 2011. UG363.

[8] X. Inc., *Virtex-6 FPGA DSP48E1 Slice User Guide*, 2011.

[9] X. Inc., *Virtex-6 FPGA GTX Transceivers User Guide*, 2011.

[10] X. Inc., *Virtex-6 FPGA GTH Transceivers User Guide*, 2011.

[11] X. Inc., *Virtex-6 FPGA Integrated Block for PCI Express User Guide*, 2012.

[12] X. Inc., *Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC User Guide*, 2011.

[13] X. Inc., *Virtex-6 FPGA Configuration User Guide*, 2011. UG360.

[14] X. Inc., *7 Series FPGAs Overview*, 2012.

[15] X. Inc., *Zynq-7000 Extensible Processing Platform Summary*, 2011.

[16] X. Inc., *Partial Reconfiguration User Guide*, 2011.

[17] W. Gao, A. Kugel, R. Manner, N. Abel, N. Meier, and U. Kebschull, "Dpr in high energy physics," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 39 – 44, april 2009.

[18] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, “Dynamic and partial fpga exploitation,” *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438 – 452, 2007.

[19] N. Abel, S. Manz, F. Grull, and U. Kebschull, “Increasing design changeability using dynamic partial reconfiguration,” *Nuclear Science, IEEE Transactions on*, vol. 57, pp. 602 – 609, april 2010.

[20] S. Mitra, P. P. Shirvani, and E. J. Mccluskey, “Fault location in fpga-based reconfigurable systems,” in *IEEE Intl. High Level Design Validation and Test Workshop*, pp. 12 – 14, A, 1998.

[21] T. S. S.A., *On-Chip Instrumentation and Debug Tool*, 2005.

[22] L. Lemmen, *Master’s Thesis Preparation, FPGA Firmware Qualification Framework*, 2011.

[23] H. Angepat, G. Eads, C. Craik, and D. Chiou, “Nifd: Non-intrusive fpga debugger – debugging fpga ’threads’ for rapid hw/sw systems prototyping,” in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 356 – 359, 2010.

[24] R. Stallman, R. Pesch, S. Shebs, and et al., *Debugging with GDB, The gnu Source-Level Debugger.* GNU.org, ninth edition, for gdb version 6.8.50.20080705 ed.

[25] C. Ciordas, A. Hansson, K. Goossens, and T. Basten, “A monitoring-aware network-on-chip design flow,” *J. Syst. Archit.*, vol. 54, pp. 397 – 410, March 2008.

[26] P. S. Graham, *Logical Hardware Debuggers for FPGA-Based Systems*. PhD thesis, Brigham Young University, December 2001.

[27] P. Bellows and B. Hutchings, “Jhdl-an hdl for reconfigurable systems,” in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 175 – 184, apr 1998.

[28] T.-Y. Lee, C.-C. Hu, L.-W. Lai, and C.-C. Tsai, “Hardware context-switch methodology for dynamically partially reconfigurable systems,” *J. Inf. Sci. Eng.*, vol. 26, no. 4, pp. 1289 – 1305, 2010.

[29] K. S. Hemmert and B. Hutchings, “Issues in debugging highly parallel fpga-based applications derived from source code,” in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC ’03, (New York, NY, USA), pp. 483 – 488, ACM, 2003.

[30] K. Denolf, A. Chirila-Rus, P. Schumacher, R. Turney, K. Vissers, D. Verkest, and H. Corporaal, “A systematic approach to design low-power video codec cores,” *EURASIP J. Embedded Syst.*, vol. 2007, pp. 42 – 42, January 2007.

[31] L. Lagadec and D. Picard, “Software-like debugging methodology for reconfigurable platforms,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 – 4, may 2009.

[32] M. Khan, R. Pittman, and A. Forin, “gnosis: A board-level debugging and verification tool,” in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pp. 43 – 48, 2010.

[33] B. Vermeulen and K. Goossens, “Debugging multi-core systems on chip,” in *Multi-Core Embedded Systems* (G. Kornaros, ed.), ch. 5, pp. 153 – 198, CRC Press/Taylor & Francis Group, 2010.

[34] B. Vermeulen and K. Goossens, “A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs,” in *VLSI Design, Automation and Test, 2009. VLSI-DAT ’09. International Symposium on*, pp. 183 – 186, april 2009.

[35] Altera, “Avalon interface specifications,” May 2011.

[36] IBM, “On-chip peripheral bus architecture specifications,” April 2001. Version 2.1.

[37] IBM, “128-bit processor local bus architecture specifications,” May 2007. Version 4.7.

[38] ARM, “Amba axi protocol specification,” March 2010. Version 2.0.

[39] OPC, “Open core protocol specification,” 2005. Version 2.1 Revision 1.0.

[40] X. Inc., *Xilinx AXI Reference Guide*, 2010. UG761.

[41] X. Inc., *LogiCORE AXI Interconnect IP (v1.01.a)*, 2010. DS768.

[42] X. Inc., *CORE Generator Guide*, 2000. coregen.

[43] R. Stallman, R. Pesch, S. Shebs, and et al., *Debugging with GDB: The GNU Source-Level Debugger*, ninth edition ed., January 2002. for GDB version 5.1.1.

[44] X. Inc., *Virtex-6 FPGA Clocking Resources*, 2011. ug362.

[45] Altera, *Quartus II Handbook Version 11.1*, 2011.

[46] X. Inc., *LogiCORE IP AXI Ethernet Lite MAC (v1.00a)*, 2011.

[47] X. Inc., *LogiCORE IP AXI UART Lite (v1.01a)*, 2010.

[48] X. Inc., *LogiCORE IP AXI HWICAP (v2.00a)*, 2010.

[49] X. Inc., *LogiCORE IP AXI GPIO (v1.00a)*, 2010.

[50] X. Inc., *LogiCORE. IP AXI External Slave Connector (v1.00.a)*, 2012.

[51] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O’Reilly Media, Inc., 2005.

[52] X. Inc., *Product Brief: 7 SERIES FPGAS*, 2011.