Reconfigurable Instrument Access Network with a Functional Port Interface

PRATHAMESH MURALI
GANI KUMISBEK

MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Reconfigurable Instrument Access Network with a Functional Port Interface

Prathamesh Murali
pr3062mu-s@student.lu.se
Gani Kumisbek
soc15gku@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Erik Larsson

Examiner: Pietro Andreani

May 3, 2019

Abstract

The ever-increasing need for higher performance and more complex functionality pushes the electronics industry to find a faster and more efficient way to test and debug an Integrated Circuit (IC). Currently, the IEEE Std. 1149.1, known as Joint Test Action Group (JTAG) is considered as state of the art by the industry. JTAG is used to perform debugging and testing through Test Access Port (TAP). However, the IEEE Std. 1149.1 standard has three major drawbacks, such as:

- Lack of flexibility of hardware and scalability in scheduling the access to the instruments;
- Boundary Scan Definition Language (BSDL), which is part of the JTAG standard, is insufficient to describe the myriad types of instruments present in an IC;
- Absence of a language to ascertain the operation of an instrument independently of its position, configuration or utilization within an IC.

Therefore, the IEEE Std. 1687, also known as Internal JTAG (IJTAG), was developed to mitigate these drawbacks by offering additional features, namely:

- The Segment Insertion Bit (SIB) and ScanMux control bit are introduced for dynamic reconfiguration of a boundary scan path;
- Procedural Description Language (PDL) and Instrument Connectivity Language (ICL) are used to fulfil the need for interfacing and description of a on-chip instrument of variable complexity.

In this thesis, we proposed Universal Asynchronous Receiver Transmitter, known as UART, as a functional port to access embedded instruments and designed IJTAG network on Xilinx Field-Programmable Gate Array (FPGA) followed by implementation of the re-targeting tool in Python programming language. Our main objective was to determine if the TAP and the associated controller can be replaced by an UART port interface, while maintaining the same functionality. Additionally, we used data transfer as a performance metric to determine the feasibility of the UART.

We explored 4 different design alternatives by building a narrative from a pure software solution to full-featured hardware solution, consequently adding new components to efficiently interpret re-targeting commands, thereby optimizing data transfer and FPGA resource utilization.

Finally, we made recommendations based on results obtained, as to inclusion or exclusion of the different components.

Popular Science Summary

Electronic devices are typically composed of various components known as Integrated Circuits (IC) or chips that operate together to fulfill the functions of the device. ICs are in turn composed of transistors, the basic element of an electronic circuit. Rapid advances in manufacturing technology has allowed for a substantial reduction in the size of these transistors. This means that designers can afford to use more transistors to produce complex ICs. However, this level of complexity has increased the effect of errors (or bugs) and faults during the design and manufacturing process. It has also made the ICs more sensitive to external environmental factors and natural processes like aging from wear and tear.

Given these challenges, facilitating constant monitoring of the various components of an electronic device for failures has become a crucial task for designers. In order to achieve this in a non-intrusive manner, a solution was developed to embed the monitoring, testing and debugging (finding and removing bugs) components into the ICs during the manufacturing process. These embedded components are referred to as on-chip instruments.

Since the instruments are embedded inside a chip, additional infrastructure is required to make them accessible to the environment. This is known as the instrument access network or network in short. Work has been done to design networks that allow for non-intrusive, re-configurable instrument access in an efficient manner. However, these networks require their own dedicated interface to the environment (known as a port) that cannot be used for any other purpose. This could potentially prove problematic for designers who have access to a limited amount of resources.

In this thesis, we attempt to mitigate this issue by eliminating the need for a special port by connecting the network to already existing interfaces (or a functional port) that are used to facilitate the functioning of the chip. Additionally, we also develop a control scheme based on the functional port to maintain the efficiency in terms of data exchanged between the chip and environment (known as data overhead). This means that the same functionality is maintained without the need for a dedicated port. Finally, we determine if any improvements have been made by comparing the data overhead costs between implementations with the dedicated and functional ports.

Acknowledgment

We would like to express our sincere gratitude to our supervisor Prof. Erik Larsson, who sparked and supervised the project.

Also we would like to thank Ericsson supervisors Magnus Osterholm, Shkelqim Lahi, Steffan Nilsson and Umer Hasnain for a rare opportunity to test our solution on the field and providing us with lab equipment and consultation.

We would like to express our gratitude to Bolashak Scholarship programme for funding graduate studies. Special thanks to Gulmira Baikhozhayeva, Ayaulym Erbolkyzy and the committee members.

Special thanks to our International Master's Coordinator - Helene von Wachenfelt, for her patience and administrative support.

Last but not least, we would like to express our gratitude to our families. Without their support and motivation, we wouldn't be able to study at Lund University.

Table of Contents

1 Introduction **1**
1.1 Thesis motivation . . . 1
1.2 Planned results . . . 2
1.3 Thesis organization . . . 2

2 Theory **3**
2.1 UART standard . . . 3
2.2 IEEE Std 1149.1 (JTAG) . . . 4
2.3 IEEE Std 1687 (IJTAG) . . . 6
2.4 Segment Insertion Bit (SIB) . . . 6
2.5 Description languages . . . 8
2.6 Re-targeting . . . 8
2.7 iApply groups . . . 9
2.8 Discussion on data overhead . . . 9
2.9 Summary . . . 10

3 Methodology **11**
3.1 Technical setup and workflow . . . 11
3.2 Bit-banging case . . . 12
3.3 Naive case . . . 14
3.4 Case without Dummy Data Handling . . . 21
3.5 Full-featured case . . . 29

4 Results **35**
4.1 Testing methodology . . . 35
4.2 Bit-banging case . . . 37
4.3 Naive case . . . 40
4.4 Case without Dummy Data Handling . . . 42
4.5 Case without Instrument Length Memory . . . 42
4.6 Full-featured case . . . 42
4.7 Inferences . . . 46

5 Conclusion **51**
5.1 Introduction and theory . . . 51

5.2 Methodology 51
5.3 Results 51
5.4 Future work 52

References **53**

List of Figures

2.1 Break down of UART transmission. *Courtesy: Soliton Technologies* [9]. 4
2.2 JTAG TAP controller circuitry 5
2.3 TAP controller state diagram.*Courtesy: Embecosm Limited* [10] . . . 5
2.4 Flat 1687 Network containing two instruments 6
2.5 Conceptual diagram of SIB implementation 7
2.6 Detailed SIB schematic. 7
2.7 Three instrument network. 8

3.1 Digilent Nexys 4 DDR FPGA. 11
3.2 Conceptual diagram of bit-banging case. 12
3.3 Retargeting flow for read operation of bit-banging case. 13
3.4 Retargeting flow for write operation of bit-banging case. 13
3.5 Conceptual diagram of the Naive case. 15
3.6 ASMD diagram of the main FSM component for Naive case. 16
3.7 ASMD diagram of UART interpreter component for Naive case. . . . 17
3.8 Bit-wise description of *iRead* setup command for the Naive case. . . 19
3.9 Bit-wise description of *iWrite* setup command for the Naive case. . . 21
3.10 Outputs of apply groups. 21
3.11 Conceptual diagram of the case without dummy data handling. . . . 22
3.12 ASMD diagram of main FSM component of the case without dummy data handling. 23
3.13 ASMD diagram of main FSM component of the case without dummy data handling. 24
3.14 ASMD diagram of output FSM component of the case without dummy data handling. 26
3.15 Bit-wise description of *iRead* setup command for the case without Dummy Data Handling. 27
3.16 Bit-wise description of *iWrite* setup command for the case without Dummy Data Handling. 29
3.17 Outputs of apply groups for the case without DDH. 31
3.18 Conceptual diagram of full-featured case. 31
3.19 UART transceiver FSM module. 32
3.20 Result of *iRead* instrument 1 apply command for the full-featured case. 34

4.1 Data overhead comparison between full-featured and without DDH cases. 47
4.2 Data overhead comparison between full-featured and without ILM cases. 48
4.3 Data overhead comparison across all cases. 49
4.4 Resource utilization scaling with respect to data overhead. 50

List of Tables

4.1 Resource consumption for all cases in terms of logic blocks. 38
4.2 Test results for bit-banging case. 39
4.3 Test results for Naive case. 41
4.4 Test results for no dummy data handling case. 43
4.5 Test results for no instruction length memory case. 44
4.6 Test results for full-featured case. 45

Acronyms

AMBA	The ARM Advanced Microcontroller Bus Architecture (AMBA)
ASMD	Algorithmic State Machine with Data-path
BPS	Bits per second
BSDL	Boundary Scan Definition Language
CLB	Configurable Logic Block
DDH	Dummy Data Handling
EDA	Electronic design automation
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
I^{2}C	Inter-Integrated Circuit
IC	Integrated Circuit
ICL	Instrument Connectivity Language
IEEE	Institute of Electrical and Electronics Engineers
IJTAG	Internal JTAG
ILM	Instrument Length Memory
IR	Instruction Register
JTAG	Joint Test Action Group
LB	Logic Block
MSB	Most significant bit
ODU	Output Discard Unit
PDL	Procedural Description Language

SCR	SIB Control Register
SIB	Segment Insertion Bit
SPI	Serial Peripheral Interface
TAP	Test Access Port
TCK	Test Clock
TDI	Test Data In
TDO	Test Data Out
TDR	Test Data Registers
TMS	Test Mode Select
UART	Universal Asynchronous Receiver-Transmitter

Listings

3.1 PDL of the apply group used in this chapter 14
4.1 PDL for the BASTION benchmark 36

List of Algorithms

1 Setup commands for the Naive case 18
2 Action command for the Naive case 20
3 Setup commands for case without dummy data handling 28
4 Action command for case without dummy data handling 30
5 Setup commands for full-featured case 33
6 Action command for full-featured case 34

Chapter 1

Introduction

By the 1980's, traditional circuit board testing methodologies such as bed-of-nails testing were becoming increasingly cumbersome and expensive. The IEEE 1149.1 standard, known by the acronym JTAG, was developed to circumvent this problem [1, 2]. JTAG uses boundary-scan testing to perform debugging and testing on a design through a Test Access Port (TAP). Boundary-scan testing is facilitated by including shift register cells adjacent to each component pin so that the peripheral signals can be controlled and observed during testing [2].

In recent times, however, Integrated Circuit (IC) are being manufactured with smaller and faster transistors. While massively advantageous, this also makes the ICs more prone to in-field malfunctioning due to phenomena such as soft errors, intermittent faults and aging [3]. These problems have led to modern ICs being equipped with increasingly complex features for testing, debugging, configuration and control [1, 4]. These embedded on-chip features, known as instruments [4], have to be accessed in a low-cost, non-intrusive manner for reliable testing and debugging of the chip. In this context, the JTAG standard has proven to be insufficient due to lack of flexibility and scaling. The IEEE Std 1687 was introduced to address these problems. It introduces scalable and automatable on-chip instrument access methods while using the framework provided by the JTAG standard, namely the TAP controller and its constituent components.

1.1 Thesis motivation

While work has been done to optimize the IJTAG architecture [4] and potential applications have been explored [4, 5, 6, 7, 8], these implementations and proposed designs still use the TAP and its associated architecture. The objective of our thesis is to evaluate the feasibility of UART port as replacement for the TAP proposed in the JTAG standard while maintaining the same functionality. This is done for the following reasons:

- The requirement of a dedicated testing port with at least 4 pins might prove to be expensive decision to make for designers with hardware constraints.
- The ubiquity of functional ports such as UART, SPI, I^2C etc. and the fact that they can be used for other purposes while not performing testing

operations makes it simpler and cheaper to access on-chip instruments while maintaining flexibility.

Among the various functional ports available, UART was chosen because it offers full duplex communication with requirements for only two pins. I^2C requires additional features such as interrupts that might add to the efficiency metrics such as data overhead and SPI requires 4 pins to operate just like the TAP, thereby defeating the purpose of the thesis.

1.2 Planned results

The goal of the thesis is to replace the existing TAP and its associated controller with a UART port interface and determine if any tangible improvements can be detected. To that end, hardware overhead in the form of data and area overhead has been selected as the performance metric. In our thesis, *the data overhead is defined as the total sum of all bits used in establishing access to on-chip instruments through UART port interface that cannot be categorized as useful data bits.* The overhead is presented for networks with an UART port and one with a TAP and comparisons will be made. Additionally, the impact of the various components that make up the UART controller is also measured in terms of data overhead and compared with the full-featured implementation. Commentary on the feasibility is made based on the analysis of the aforementioned comparisons.

1.3 Thesis organization

The rest of the thesis is organized in the following manner:

In Chapter 2, the basic concepts upon which the implementation is built are explored and discussed. Namely, the architecture of TAP and TAP controller are briefly presented. The state machine of the TAP controller is discussed in more detail since the UART controller is based on it. The basic architecture of the 1687 network and associated concepts such as ICL, PDL, and Apply Groups etc. are discussed. The hardware overhead is explored in this chapter. A brief overview of the UART standard is also presented.

In Chapter 3, the UART protocol developed to facilitate communication is introduced and explored. Furthermore, the state machines for the various versions of the UART controller that were implemented are presented and explained.

In Chapter 4, the results of the measurements made are presented in the form of tables and charts. Brief remarks are also made about the `BASTION` benchmark that are used to make measurements of our overhead metric. Analysis is also done of the measurements made.

In Chapter 5, inferences and recommendations are made based on analysis of the data gathered. Concluding remarks are also presented here and potential improvements and future work are briefly discussed.

Chapter 2
Theory

In this chapter, the concepts used for the implementation of our UART controller and 1687 network are discussed. We begin with an overview of the UART standard used to facilitate communication between the network and the user. We move on to take a brief look at the 1149.1 TAP controller based on which the UART controller was designed. Finally, the architecture of the 1687 network and related concepts such as PDL, ICL and Apply Groups are elaborated upon. A brief look at hardware overhead is also made.

2.1 UART standard

The Universal Asynchronous Receiver-Transmitter (UART) is an asynchronous serial communication protocol that allows for variable data formats and transmission speeds. The standard includes a transmitter and a receiver. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and then reassembles the data. The serial line is '1' when it is idle. Transmission starts with a start bit, which is '0', followed by 5-8 data bits, an optional parity bit and ends with a stop bit, which is '1'. The optional parity bit is used for error detection. For odd parity, it is set to '0' when the data bits have an odd number of 1s. For even parity, it is set to '0' when data bits have an even number of 1s. The number of stop bits can be 1, 1.5 or 2.

Figure 2.1 shows a transmission with 8 data bits, 1 parity bit and 1 stop bit. No clock information is conveyed through the serial line.

Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate, specifying the number of bits transmitted per second (BPS), number of data bits and stop bits and the use of the parity bit. Commonly used baud rates are 2400, 4800, 9600, 19200 and 115200 BPS.

Since no clock information is conveyed by the transmitter, the receiver can retrieve the data bits only by using predetermined parameters. An oversampling scheme is used to overcome this problem. As per this scheme, the middle point of the transmitted bits is estimated, and the receiver polls the channel at these points. The oversampling rate is determined by the baud rate and the clock period of the receiver. The most commonly used oversampling rate is around 16 times

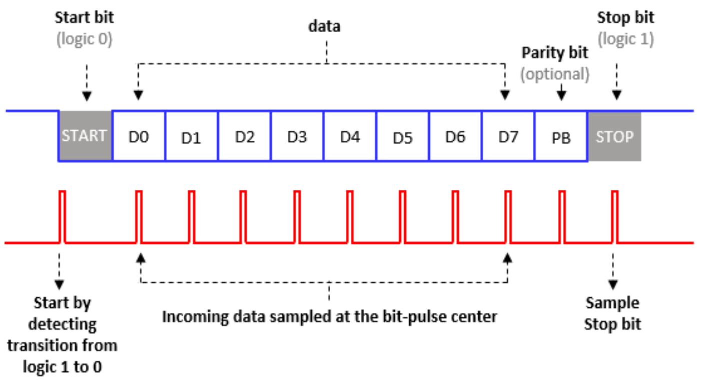


Figure 2.1: Break down of UART transmission. *Courtesy: Soliton Technologies* [9].

the baud rate, meaning each transmitted bit is sampled 16 times.

Our implementation uses a UART standard with 8 data bits, no parity bits, 1 stop bit and a baud rate of 115200 BPS.

2.2 IEEE Std 1149.1 (JTAG)

A discussion on the TAP controller architecture and state machine is made in this section. Figure 2.2 shows a conceptual view of the JTAG circuitry in a chip. Two Test Data Registers (TDR)s are mandatory, namely Boundary Scan Register and Bypass Register. Design specific TDRs can also be accommodated based on the design in the chip. JTAG uses a serial protocol and either Instruction Register (IR) or one of the TDRs is accessible serially. The interface for this circuitry is the Test Access Port (TAP). The port includes four mandatory ports, namely, Test Data In (TDI), Test Data Out (TDO), Test Mode Select (TMS) and Test Clock (TCK) [2].

The TMS signal is decoded by the state machine to generate control signal capture, shift and update operations on the IR and TDRs. The state diagram is depicted in Figure 2.3. The Capture operation is defined as parallel loading of a value into the IR or TDR. Update operation is defined as transferring logic values from the shift register stage of IR or TDR to their latched parallel outputs. The Shift operation is defined as shifting the data serially in and out of the IR or TDR one bit per clock cycle.

In the following, it is explained how the state machine generates the control signals and used to transfer data to and from IR or the TDRs. The state machine has two similar branches, the IR branch for operations on the IR and the DR branch for operations on the currently selected TDR. The select signals for the TDRs come from the IR decoder, which are sued to activate a TDR based on the instruction in the IR. Input vectors are serially shifted into the currently selected TDR when the controller is in `Shift_DR` state. The input vectors can be fully shifted in by keeping the TMS signal at logic ’0’. Moving to `Update_DR` loads

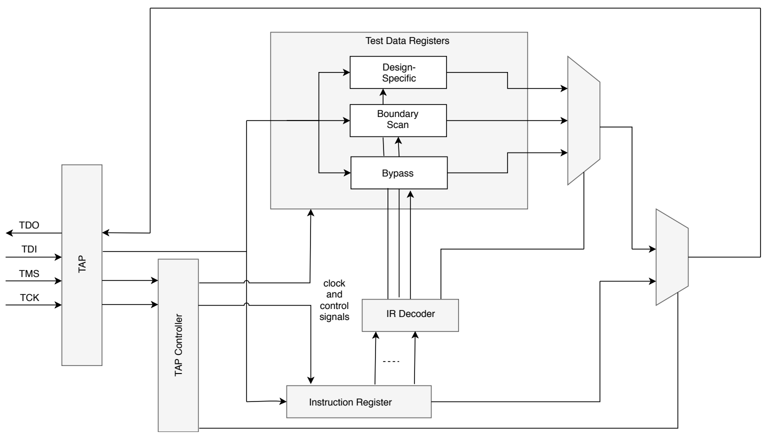


Figure 2.2: JTAG TAP controller circuitry

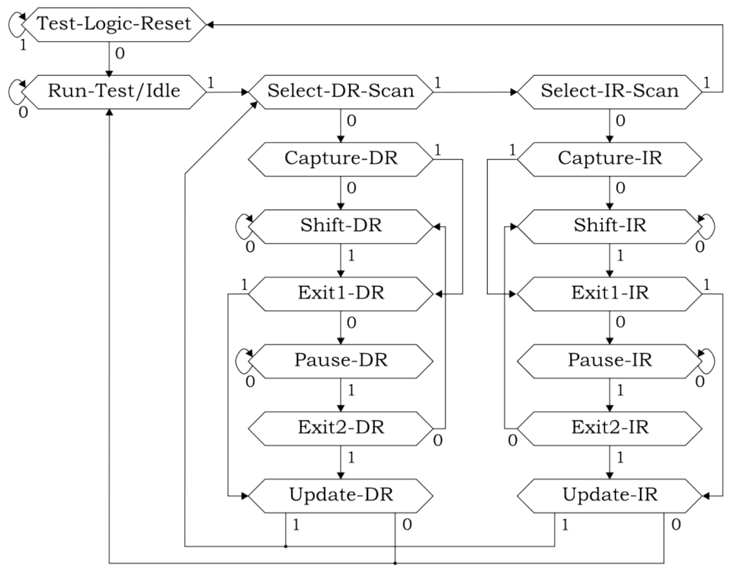


Figure 2.3: TAP controller state diagram.*Courtesy: Embecosm Limited* [10]

the input vector to parallel outputs of the selected TDR. The output vectors are loaded onto the selected TDR during the `Capture_DR` state.

2.3 IEEE Std 1687 (IJTAG)

The IJTAG standard [11], describes a methodology for accessing on-chip instrumentation through a dynamically re-configurable network. This allows for greater flexibility and scalability than that offered by the 1149.1 standard. To enable dynamic reconfiguration in 1687 networks, multiplexers are used on the scan path, hereby referred to as Scan MUXes. These MUXes are configured via a control bit, which is a shift-update register that can be placed anywhere on the scan path. An example network consisting of two instruments is shown in Figure 2.4.

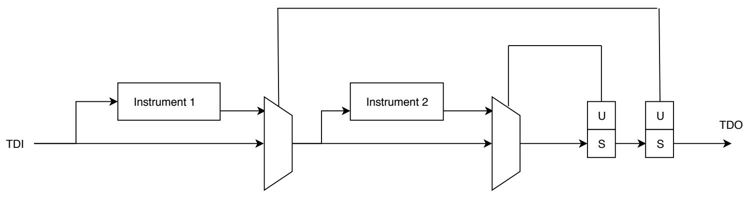


Figure 2.4: Flat 1687 Network containing two instruments

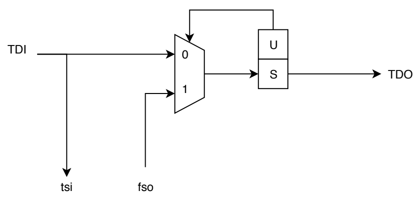
To configure the control bit, the desired value is placed in the shift cell, denoted by `S` during the Shift phase of the state machine. This is followed by the Update phase during which the value in `S` is loaded into its parallel latch, denoted by `U`. Configuring control bits or applying input vectors to the network ID done by cycling through `Capture`, `Shift` and `Update` phases of the state machine. This cycle is referred to as a `CSU` operation.

It should be noted that different network design approaches are possible such as Flat networks, Hierarchical networks, Multiple Networks, Daisy-chained Networks etc. [4]. However, due to its relative simplicity, an UART interface for a flat network is explored in this thesis.

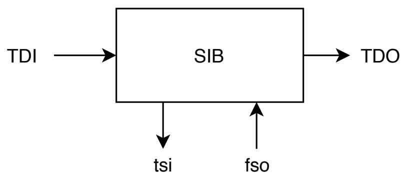
2.4 Segment Insertion Bit (SIB)

In order to allow for dynamically re-configurable instrument access networks, a combination of two input Scan MUXes and control bit was introduced in the last section. This combination is referred to as the Segment Insertion Bit abbreviated as SIB. Figure 2.5 shows the conceptual diagram of a typical SIB implementation.

The SIB has shift flip-flop, denoted by `S`, an update flip-flop, denoted by `U` and a two input Scan MUX. A SIB is programmed by shifting a bit into their `S` flip-flop and loading it into its parallel latch `U` flip-flop. If the latched bit is '0', the SIB is closed, and the scan path is from `Test Data In (TDI)` to `Test Data Out (TDO)` and it bypasses the segment between `TSI` (to scan-in) and `FSO` (from scan-out) terminals. These terminals are called the host port. This segment



(a) Simplified SIB schematic.



(b) Symbol representing a SIB.

Figure 2.5: Conceptual diagram of SIB implementation

contains the instrument and an associated shift register. If the latched bit is '1', the SIB is opened, and the scan path includes the instrument and shift register. A detailed schematic of our SIB implementation is shown in Figure 2.6. The control

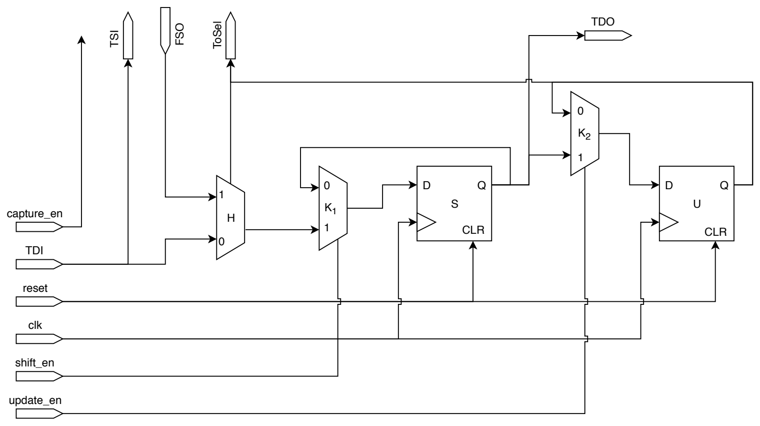


Figure 2.6: Detailed SIB schematic.

signals `shift_en`, `update_en` and `capture_en` are generated by the state machine developed for the implementation. When the state machine is in its shift phase, `shift_en` is set to '1' and new values are shifted into the S flip-flop via `H` (host port MUX) and `K1` (Keeper MUX 1). While the state machine is not in shift phase, `shift_en` is set to '0' and the `S` flip-flop retains its value through feedback from `K1`. During update phase of the state machine, `update_en` is set to '1' and the `U` flip-flop obtains the value stored in the `S` flip-flop via `K2` (keeper MUX 2). While not in update phase, `update_en` is set to '0' and the `U` flip-flop retains its value through feedback from `K2`. If the `U` flip-flop stores a value of '0', `H` receives values directly from `TDI`. The segment between `TSI` and `FSO` containing the instrument is also deselected by the ToSel terminal. In this case data is shifted directly from `TDI` to `TDO`. In cases where the `U` flip-flop stores a value of '1', input 1 of `H` is selected and it receives values from the `FSO` terminal. The segment between `TSI` and `FSO` is also included in the scan path by the `ToSel` terminal. In this project, this process is called activating the instrument. Therefore, in these cases, data is shifted in via the `TDI` port, through the host port terminals and out of the `TDO` port.

It should be noted that only one of the control signals is active at any given time. This is because different phases of the state machine activate different control signals and deactivates the others. It should also be noted that `capture_en` and `update_en` are used to facilitate parallel loading of data between instrument and shift register and shift register and instrument respectively.

2.5 Description languages

The IEEE 1687 standard introduces two description languages, Instrument Connectivity Language (ICL) and Procedural Description Language (PDL). PDL is used to describe the operation of instruments at their terminal. Our project implements an UART protocol that allows for read and write operations on the instrument shift registers and configurable components. The purpose of ICL is to describe the characteristics of the instruments such as data length and position within the network and the requirements for interfacing to them [1]. In our project, the network can accommodate a maximum of 1000 instruments due to hardware constraints. Since we are attempting to establish a protocol to replace the TAP, simple inverters with variable data lengths were chosen as instruments. The data lengths of the inverters vary between 8 bits, 16 bits or 32 bits based on their position in the network. The first instrument, i.e. the instrument closest to the input port `TDI`, denoted by `i0`, has a data length of 8 bits. `i1` has a data length of 16 bits and `i2` has a data length of 32 bits (see fig. 2.7). This sequence of data lengths continues in a loop for all subsequent instruments.

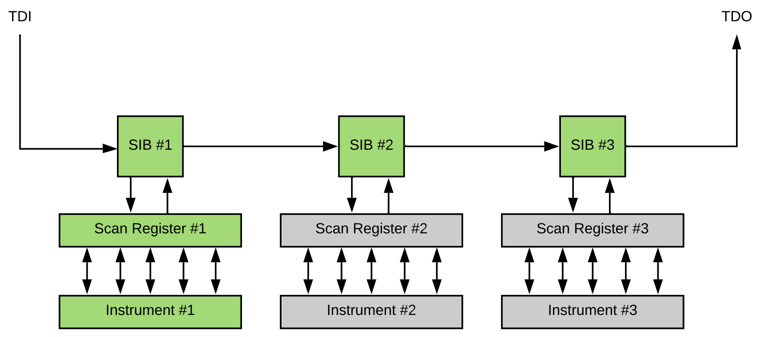


Figure 2.7: Three instrument network.

2.6 Re-targeting

The PDL and ICL describe the instruments in a network and commands to be executed on them. However, they do not generate the input vectors that need to be shifted into the network to configure the instruments. A re-targeting tool is used for this purpose. The tool is designed to create the input vector and transport them to the instrument shift registers. This means that the designer can simply input the necessary command and let the tool automatically generate the input vector and deliver it to the shift registers of the instruments. In our project, the re-targeting tool to interpret the PDL is written is software using `Python 3`. It transmits control commands through the UART channel to controller module in the hardware implementation of the 1687 network which generates the input vectors to be shifted in. The controller contains components called SIB Control Register (SCR) and Instrument Length Memory (ILM) to facilitate this. The SCR stores information about which instruments are part of the active scan path and what command is to be executed on them. The ILM holds information about the data lengths of the instruments in the network and the position of each instrument. The values in the SCR and ILM are transmitted by the software re-targeting tool through the UART channel. Additionally, since one of the objectives of the project is to minimize data overhead, an Output Discard Unit (ODU) was introduced to allow the controller to discard garbage bits that are outputted by the network during the CSU cycles.

2.7 iApply groups

The PDL has a set of commands to specify how to operate on an instrument such as reading from or writing to its terminals. These are called *Level-0 PDL commands*. Level-0 commands consist of two types, namely, *setup commands* and *action commands*. Setup commands such as *iRead* and *iWrite* are queued and only take effect when the first subsequent action command is executes. *iApply* is an example of an action command. For example, multiple *iRead* and *iWrite* commands addressed to various instruments in a network can be queued up and executed simultaneously when the first *iApply* command is run. This format with multiple setup command followed by an *iApply* action command is called an *iApply* group.

In our project, *iApply* groups are interpreted by logic in the re-targeting tool in the software and converted into a set of addresses and control commands. The addresses specify which instrument are to be activated and the control commands determine which operation is to be executed on the activated instruments. This data is then transmitted to the hardware controller through the UART protocol developed for this purpose. The protocol is elaborated upon in Chapter 3. These UART bytes are deciphered by the controller and stored in the SCR and ILM mentioned in the previous sub-chapter. Finally, the input vectors to be shifted into the network are generated by the controller based on the SCR and ILM.

2.8 Discussion on data overhead

As mentioned in previous subchapters, the controller contains components such as SCR, ILM and ODU to assist it with interpreting the UART protocol and generate input vectors to be shifted into the 1687 network. One of the aims of the project is to determine the impact of these components on the data overhead. The overall data overhead is divided into the following sub divisions in order to facilitate this:

- SIB overhead, which specifies the overhead needed to activate the instruments by modifying the control bit in the SIBs. The addresses would fall into this category.

- PDL overhead, which specifies the overhead cost of transmitting action commands through the UART channel that trigger the controller to generate input vectors.
- Output overhead, which specifies overhead generated by the network during various CSU cycles. For example, outputs are generated by the network when instruments are activated. These are not useful and would fall into this category.

To further elaborate on the necessity of these subdivisions, we consider the impact of the ODU as an example. Because the ODU discards all output bits that are not useful, the output overhead is eliminated in implementations where the ODU is present. Therefore, it becomes easy to categorize the impact of the ODU in order to analyze it and make recommendations.

2.9 Summary

Based on the concepts discussed in this chapter, it is possible to specify exactly how the implementation will be done and what results will be obtained. The implementation will be divided into the following cases:

- Full-Featured case, where all the components (SCR, ILM and ODU) are active.
- No Dummy Data Handling (DDH) case, in which only the ODU is inactive.
- No ILM case, in which only the ILM is inactive.
- Naïve case, in which none of the components are active.
- Bit banging case, which simulates the overhead for a TAP interface. Note that this case does not actually implement a TAP and TAP controller. It uses the UART channel to simulate the behavior of a 1687 network with a TAP interface.

The various cases and the corresponding implementation are explained in more detail in the following chapter and the measurements taken are discussed in the subsequent chapters.

Chapter 3 Methodology

3.1 Technical setup and workflow

In this chapter we explain the overall methodology of our work. As a part of this work, we explored four design alternatives, namely:

- the *bit-banging* case;
- the *naive* case;
- a case when no *Dummy Data Handling (DDH)* is present;
- *full-featured* case.

We do this by consequently adding new components to look how it affects the data overhead and resource utilization of a FPGA board.

We chose Digilent Nexys 4 DDR, shown in Figure 3.1, as our main platform to develop IEEE 1687 network and test out different design alternatives [12]. The *Xilinx Vivado 2018.1* EDA tool was used for synthesis and implementation.

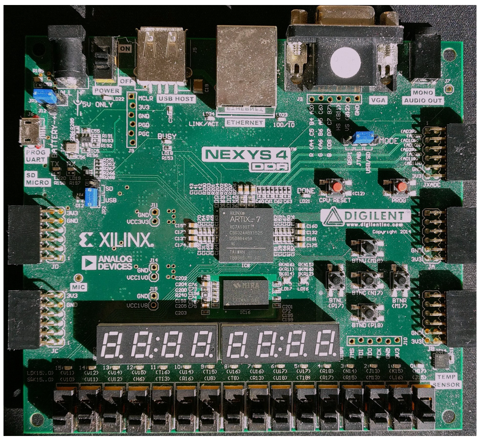


Figure 3.1: Digilent Nexys 4 DDR FPGA.

For re-targeting tool, we chose Python programming language due to extensive number of available libraries and rapid development cycle. The *pyserial* module was used to communicate with the FPGA board via serial port [13]. The first extreme design alternative we explored was bit-banging case.

3.2 Bit-banging case

Bit-banging is a special case when interaction with the IEEE 1687 instrument network is done solely on re-targeting tool. In Figure 3.2 the conceptual block diagram is shown. The TAP controller is replaced by a UART transceiver and re-targeting is performed by sending and receiving 2 byte per each clock cycle in order to transmit or receive one bit. Note that there is no UART protocol present in this case, that is because when a byte is sent to the network it gets directly applied into 8 input signals, namely `TDI`, `shift_enable`, `capture_enable`, *update_ enable*, `1687_clk`, `1687_reset`, `select` and `TDO`.

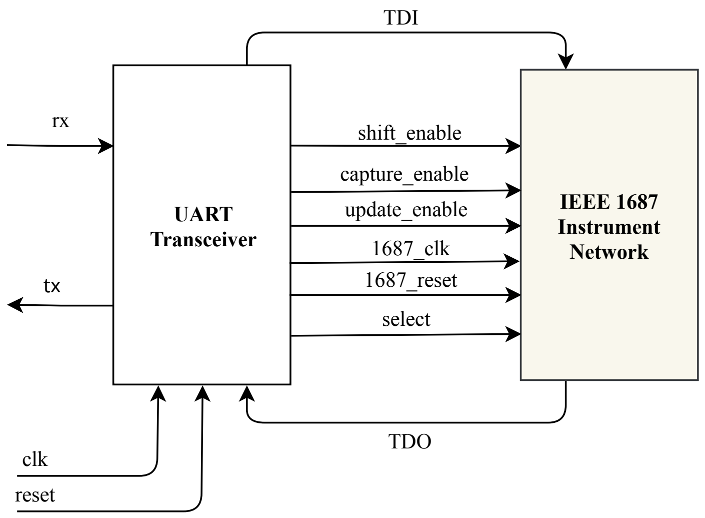


Figure 3.2: Conceptual diagram of bit-banging case.

In case of *iWrite* control command (shown in fig. 3.4), we shift number of configuration bits equal to the number of instruments in the network which applies to *iRead* control command. Simultaneously, output bits generated by the network, should be transmitted back to re-targeting tool. During the update phase, we send 2 bytes in order to update the network. However no output bits are received, since no shifting is happening. Next, during data shift phase configuration bits alongside of data bits are transmitted in order to push the required bits into the registers. Additionally, an second update is required to simultaneously load the data bits into the instruments.

As for *iRead* control command (shown in fig. 3.3), configuration and update phases follow the same procedure as in *iWrite* command, followed by capture phase in order to push the required bits out of instruments into the shift register. These are then shifted out of the network in the following phase.

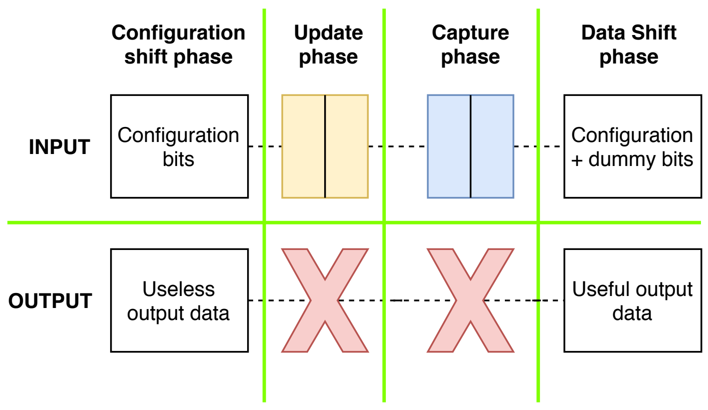


Figure 3.3: Retargeting flow for read operation of bit-banging case.

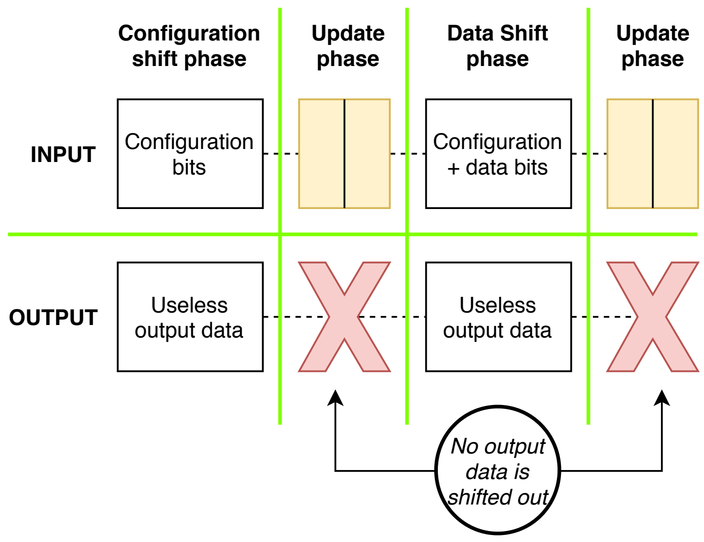


Figure 3.4: Retargeting flow for write operation of bit-banging case.

Major flaw of this solution lies within a large data consumption, because each time a data bit is sent, we have to send and receive 2 bytes in exchange. Furthermore, bit-banging method is prone with data miss and miscommunication errors due to difference in the device clock speed and re-targeting tool clock speed and therefore in order to mitigate this problem additional hardware is required to correctly interpret re-targeting commands as well as reduce data overhead.

3.3 Naive case

For sake of simplicity for design alternatives presented in this chapter, we make following assumptions:

- three instruments are present inside the network of which only first instrument lies in active scan path, i.e. activated;
- the first instrument has a data length of 8 bits (1 byte);
- *apply group commands* are sent separately.

Furthermore, we use two action commands, i.e. iWrite and iRead, for which PDL is shown in listing 3.1.

```
/* Write to instrument 1 */
iWrite(1, "10101010"); //"10101010" = 0xAA
iApply;

/* Read from instrument 1 */
iRead(1, "01010101"); //"01010101" = 0x55
iApply;
```

Listing 3.1: PDL of the apply group used in this chapter

For the Naive case, we introduce our first component - *1687 Finite State Machine* (1687 FSM for short), which serves a purpose of basic interpretation of commands received from re-targeting tool, shown in Figure 3.5 The FSM controller controls the shift, update and capture functions of the 1687 network. It is a replacement for the TAP controller defined by the 1149.1 standard. The operation of the FSM is divided into two phases, the *configuration phase* and the *data phase*. In the configuration phase, a shift-in sequence is generated to either activate a SIB or leave it inactive based on the PDL interpreted by the re-targeting tool. Note that no data is shifted into the network during this phase. The shift-in sequence merely grants access to the shift registers of active instruments.

3.3.1 Hardware description of the case

The states in this FSM (see fig. 3.6) perform the same functions as the ones specified in the full-featured case. However, the control signals that trigger state transitions are different in this case because the UART protocol is different here. The control signals are called `control_bytes` and `data_bytes`. `control_bytes` specifies how many configuration bytes, i.e. bytes needed to activate the necessary SIBs and pull the associated shift registers into the active scan path. `data_bytes`

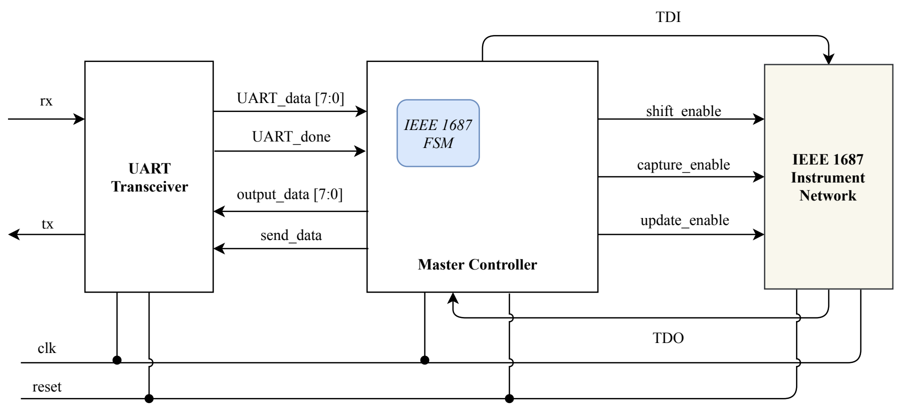


Figure 3.5: Conceptual diagram of the Naive case.

specifies the size of the shift-in sequence, i.e., bytes that contain the data to be stored in the instruments in the case of *iWrite* and dummy bits to push out the useful data in case of *iRead*. This is dependent on the number of active instruments.

A new interpreter FSM was created (see fig. 3.7) for the Naive case since the UART protocol changes by a large degree with respect to the one in the full-featured case. In accordance with the protocol, the first 3 bytes signify the transmission of configuration bytes and are processed in the `REC_BYTES1` and `REC_BYTES2` states. The first UART byte specifies the number of configuration bytes that will be transmitted. This value is stored in control signal, `control_bytes`. The next two bytes specify the number of bytes required for the shift-in sequence that are to be transmitted to the network. These are stored in control signal, `data_bytes`.

The FSM transitions to `REC_CONTROL` state where the configuration bits are received and shifted into the network. When the number of bytes received reaches the number stored in `control_bytes`, the FSM transitions to `REC_DATA` state signifying that all configuration bytes have been received. The same logic is followed here. The FSM remains in this state until the number of bytes received matches the number stored in `data_bytes`. The FSM then transitions to IDLE state where it will wait for the next iteration to begin.

The output FSM, behaves in the same fashion as the one in case without dummy data handling (which will discuss in the next section) since the Output Discard Unit is absent here as well. The only difference here is that 8 bits are transmitted out through the UART channel every time a byte of data is received instead of when data bytes are received.

3.3.2 Re-targeting flow for the case

We start the re-targeting flow with setup commands, shown in Algorithm 1. First, we set the padding for 3 instrument network (5 bits of padding are required in our case). Then, we set configuration bits according to which instrument is currently active in scan path. Also, we set dummy bits for read setup command based on

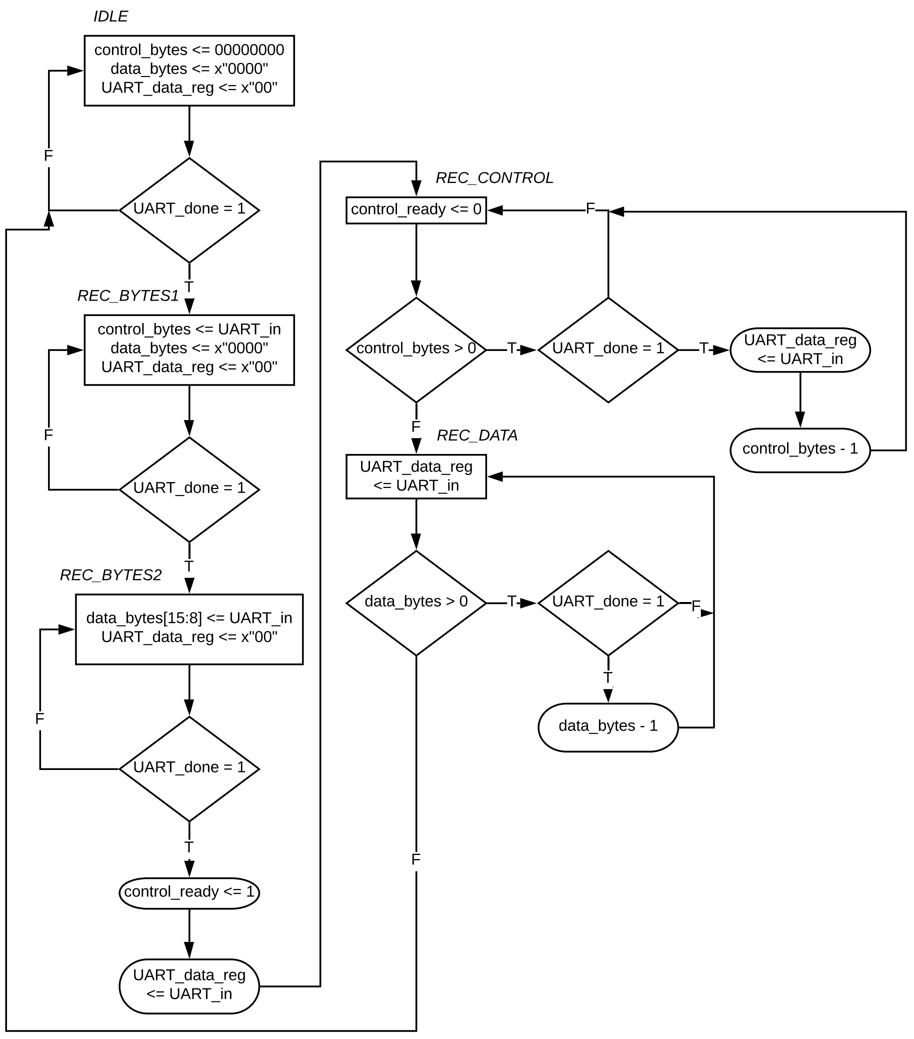


Figure 3.6: ASMD diagram of the main FSM component for Naive case.

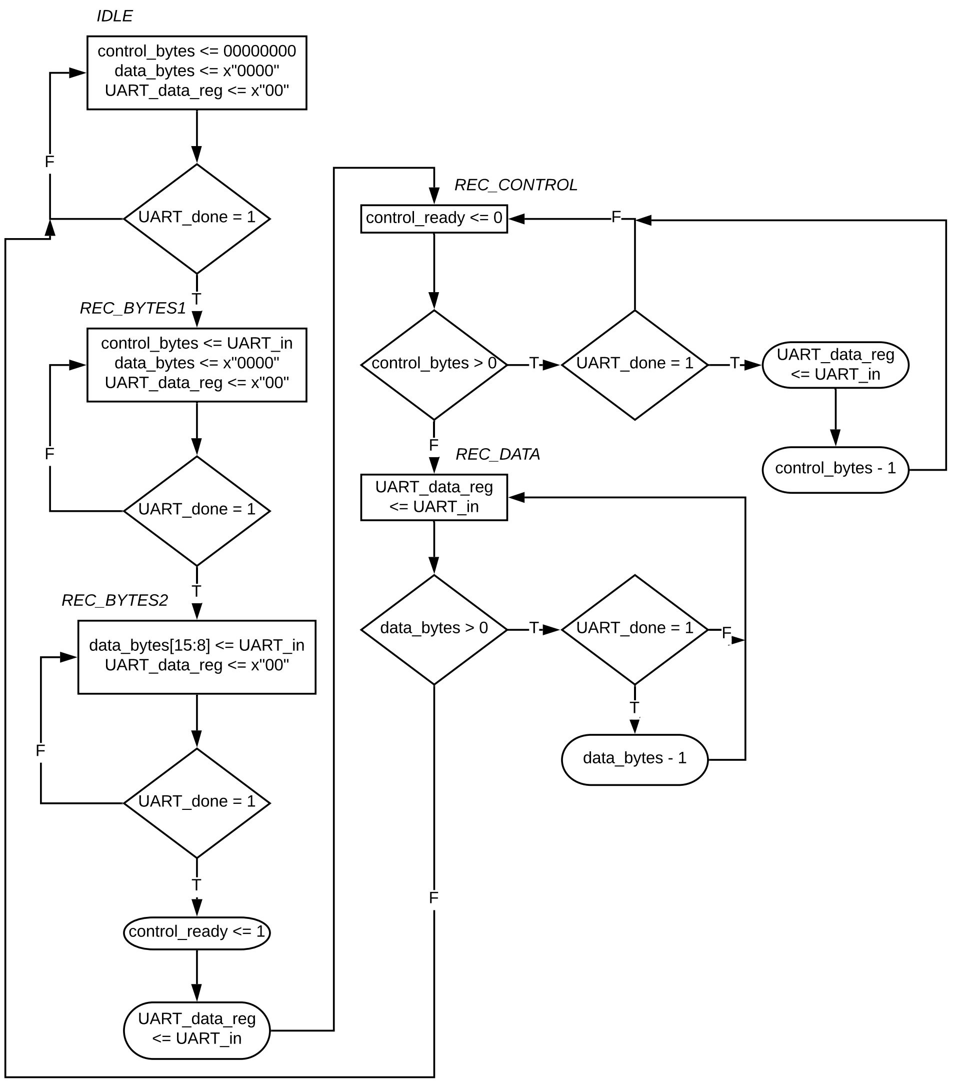


Figure 3.7: ASMD diagram of UART interpreter component for Naive case.

Algorithm 1 Setup commands for the Naive case

Require: Instrument address and instrument data
Ensure: Configuration and dummy/data bits

```
1: $totalInstruments \leftarrow 3$    ▷ Network with 3 instruments
2: $padding \leftarrow \mathrm{NND}(totalInstruments)$    ▷ Nearest number divisable by 8 for padding
3: $paddingBits \leftarrow padding + totalInstruments - 1$

4: function iRead($instrumentAddress$)
5:     $configBits[paddingBits - instrumentAddress] \leftarrow 1$    ▷ Activate the instrument
6:     if $instrumentAddress \bmod 3 = 0$ then    ▷ generate dummy bits
7:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 7)$    ▷ 1 byte
8:     else if $instrumentAddress \bmod 3 = 1$ then
9:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 15)$    ▷ 2 bytes
10:    else
11:        $dummyBits[instrumentAddress] \leftarrow (0 \ll 31)$    ▷ 4 bytes
12:    end if
13:    return $configBits, dummyBits$    ▷ Return configuration and dummy bits to action command for further processing
14: end function

15: function iWrite($instrumentAddress$, $instrumentData$)
16:    $configBits[paddingBits - instrumentAddress] \leftarrow 1$
17:    if $instrumentAddress \bmod 3 = 0$ then    ▷ translate into data bits
18:        $dataBits[instrumentAddress] \leftarrow instrumentData$    ▷ 1 byte
19:    else if $instrumentAddress \bmod 3 = 1$ then
20:        $dataBits[instrumentAddress] \leftarrow instrumentData$    ▷ 2 bytes
21:    else
22:        $dataBits[instrumentAddress] \leftarrow instrumentData$    ▷ 4 bytes
23:    end if
24:    return $configBits, dataBits$    ▷ Return configuration and data bits
25: end function
```

Instrument Length Memory (ILM) of current instrument, in our case $ILM = 8$ bits or 1 byte. Same procedure applies to write setup command, but with slight difference. Since write command requires an input from the user, data bits are received during setup phase of PDL description. In Figure 3.8 we can observe in detail the operation of read command PDL.

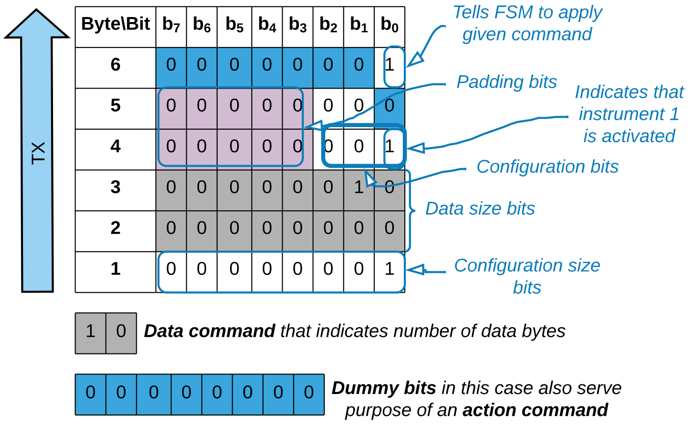

TX
Byte\Bit
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
6: 0 0 0 0 0 0 0 1
5: 0 0 0 0 0 0 0 0
4: 0 0 0 0 0 0 0 1
3: 0 0 0 0 0 0 1 0
2: 0 0 0 0 0 0 0 0
1: 0 0 0 0 0 0 0 1
Tells FSM to apply given command
Padding bits
Indicates that instrument 1 is activated
Configuration bits
Data size bits
Configuration size bits
1 0 **Data command** that indicates number of data bytes
0 0 0 0 0 0 0 0 **Dummy bits** in this case also serve purpose of an **action command**

Figure 3.8: Bit-wise description of *iRead* setup command for the Naive case.

The first byte tells the size of configuration bits for an FSM required to set the first SIBs inside the network. Second and third bytes tell an FSM the size of incoming dummy bits, in our case the size of incoming data is equal to 2 bytes. Next, actual configuration bits are shifted inside the network to tell which SIB to activate followed by the actual dummy bits. The PDL sequence then ends with last bit set to 1, to tell FSM to apply given commands and shift out the data stored inside of requested instrument.

The *iWrite* setup command, shown in fig 3.9 follows the same principle with an exception that instead of shifting dummy bits to shift out data from instrument memory, we shift in data bits to setup first instrument. Notice that in this particular case, apply and setup commands are merged into single bit-stream.

After configuration bits are generated and dummy or data bits are set, we move towards action command sequence described in Algorithm 2. The purpose of the apply function is to use UART interface to shift setup and data commands into the network and capture an output back. It also verifies if received data matching expected data, that is if we expect to receive $0x55$ it should match with received value.

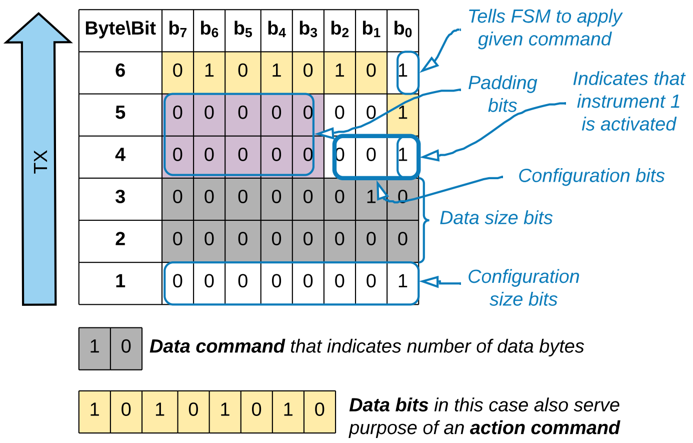
Finally, we receive an output for both *iRead* and *iWrite* setup commands, shown in figure 3.10. Note that when we read from an instrument, bits represented in yellow (see fig. 3.10(a)), are considered as a useful data bits and bits marked in red are considered as an overhead bits. When it comes to *iWrite* (see fig. 3.10(b)) everything that is received back is considered as an overhead, because during the

Algorithm 2 Action command for the Naive case

Require: Configuration and dummy/data bits

Ensure: Shift respective lengths of configuration and data/dummy bits into the network and send setup and data commands

```
1: $expectedData = "01010101"$    ▷ Expected instrument data, used to verify read command
2: $instrumentLength = 8$    ▷ Instrument length of the activated instrument (in bits)
3: $bufferSize = 1024$    ▷ 1KB sized buffer to store incoming UART data

4: function iApply($command$)
5:     if $command = iWrite$ then
6:         $configSize \leftarrow (0 \ll 7) \lor (\text{LENGTH}(configBits) \div 8)$    ▷ Calculate length of configuration and data bits
7:         $dataSize \leftarrow (0 \ll 15) \lor (\text{LENGTH}(dataBits) \div 8)$
8:         serialShift($configSize$)    ▷ and shift them into the network
9:         serialShift($dataSize$)
10:        serialShift($configBits$)    ▷ Shift setup
11:        serialShift($dataBits$)    ▷ and data bits into the network
12:    else if $command = iRead$ then
13:        $configSize \leftarrow (0 \ll 7) \lor (\text{LENGTH}(configBits) \div 8)$
14:        $dummySize \leftarrow (0 \ll 15) \lor (\text{LENGTH}(dummyBits) \div 8)$
15:        serialShift($configSize$)
16:        serialShift($dummySize$)
17:        serialShift($configBits$)    ▷ Shift setup
18:        serialShift($dummyBits$)    ▷ and dummy bits into the network
19:        $incomingData = \text{serialCapture}(bufferSize)$    ▷ Receive incoming data
20:        compare($incomingData, expectedData, instrumentLength$)    ▷ and verify its correctness
21:    end if
22: end function
```


TX
Tells FSM to apply given command
Padding bits
Indicates that instrument 1 is activated
Configuration bits
Data size bits
Configuration size bits

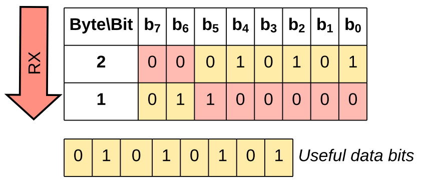
Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
6	0	1	0	1	0	1	0	1
5	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1

1	0	***Data command*** *that indicates number of data bytes*

1	0	1	0	1	0	1	0	***Data bits*** *in this case also serve purpose of an* ***action command***

Figure 3.9: Bit-wise description of *iWrite* setup command for the Naive case.

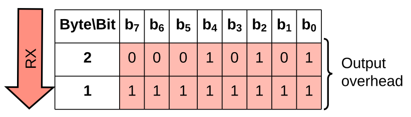
write sequence we don't expect any data back.



Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
2	0	0	0	1	0	1	0	1
1	0	1	1	0	0	0	0	0

0	1	0	1	0	1	0	1	*Useful data bits*

(a) Captured result of *iRead* command.



Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
2	0	0	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1

(b) Captured result of *iWrite* command.

Figure 3.10: Outputs of apply groups.

Lastly, the biggest issue with this method is that we still transmit large data overhead in particular when we use multiple apply groups, which we will discuss in the next chapter.

3.4 Case without Dummy Data Handling

In order to further reduce the SIB and PDL overhead, we upgrade the system with new components, like Instrument Length Memory and SIB Control Register. In Figure 3.11 the conceptual diagram of the case is shown.

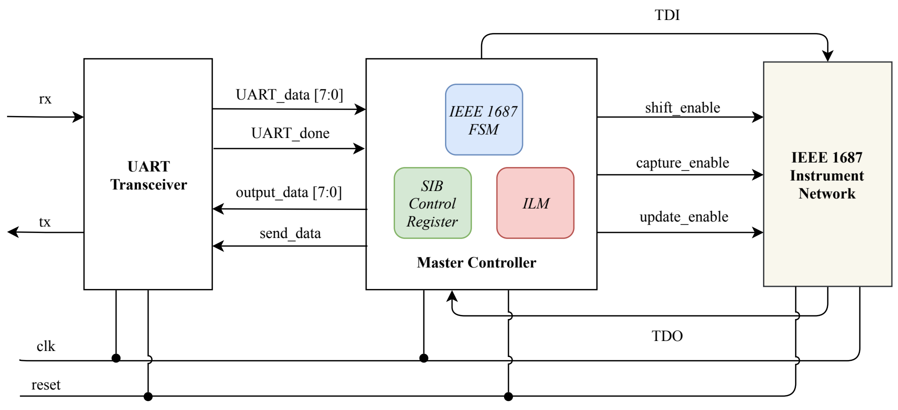


Figure 3.11: Conceptual diagram of the case without dummy data handling.

3.4.1 Hardware description of the case

Previously, we discussed the operation phase of the main FSM component. Let us clarify the functionality in more detail. `SHIFT_CONTROL`, `UPDATE_CONTROL` and `CAPTURE` fall under the configuration phase (see fig. 3.12). As the names imply, the bit sequence is shifted into the network during the `SHIFT_CONTROL` state after which he FSM transitions to `UPDATE_CONTROL`. An update happens in this state which activates the required SIBs and pulls the associated shift registers into the active scan path. A capture happens in the `CAPTURE` state which loads the shift registers with the data stored in the instruments. This will be useful data in case of *iRead* and discarded as garbage bits in case of *iWrite*s.

The FSM then transitions to the `IDLE` state and waits for the first byte of data to be transmitted by the re-targeting tool. Note that this is only the case for *iWrite*, where data must be shifted into the network. In case of *iRead*, dummy bits are shifted in instead to push out the useful data. Once the data has arrived, a shift-in sequence is generated to shift the data or dummy bits into the correct activated instruments. This means that additional dummy bits are required for those SIBs that are not active along with the actual data.

`SHIFT_DATA` and `UPDATE_DATA` are states that fall under the data phase. In the `SHIFT_DATA` state, the shift-in sequence is generated and shifted into the network. The FSM then transitions to the `UPDATE_DATA` state where an update occurs. This prompts the parallel loading of data from the shift registers to the instruments. The FSM finally transitions to the `RESET_STATE`, wherein all the counters and states of other FSM are reset and ready for a new iteration of operation.

The purpose of FSM, shown in Figure 3.13, is to interpret the protocol developed for UART communication and assist the main FSM in generating the shift-in sequence by the way of control signals. Note that this does not perform the function of the re-targeting tool. That happens in software.

The default state of the FSM is `IDLE`, from which it transitions when it receives a byte of data through the UART channel. This is signified by the `UART_done`

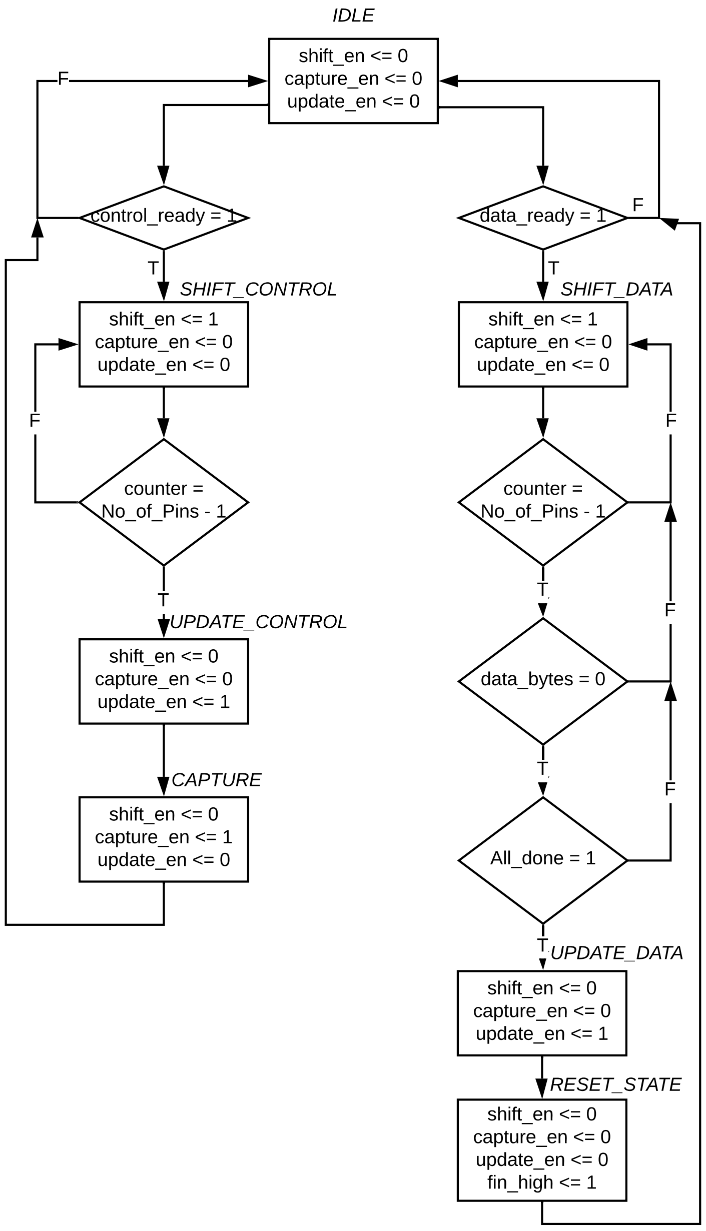


Figure 3.12: ASMD diagram of main FSM component of the case without dummy data handling.

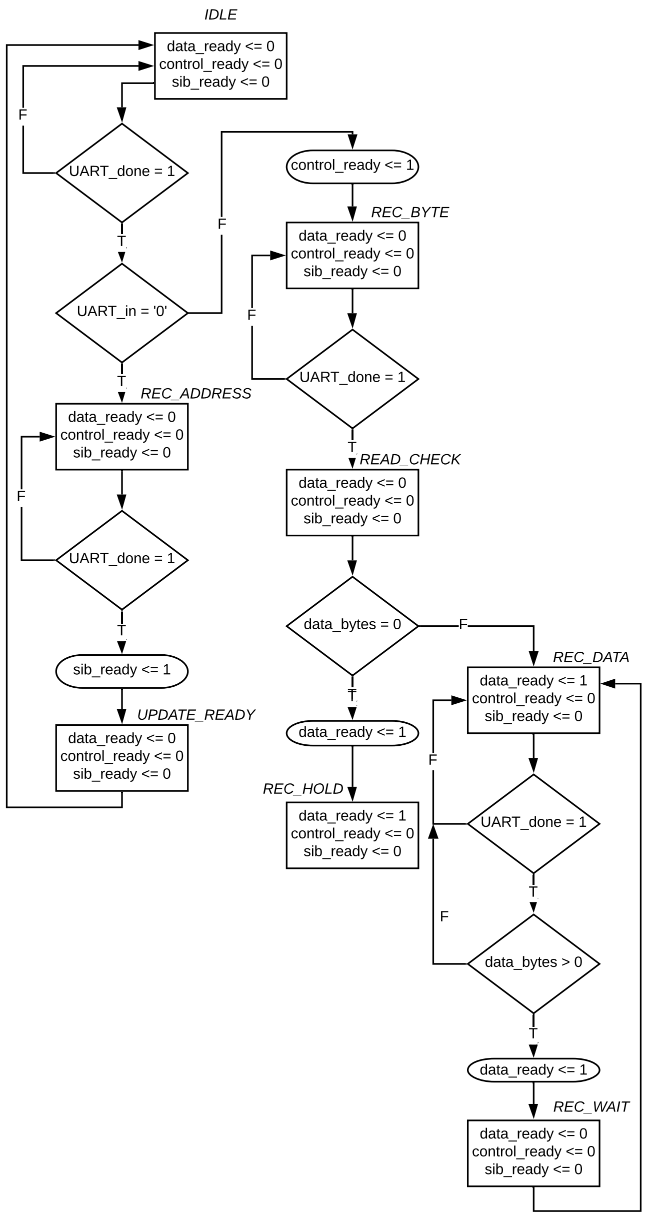


Figure 3.13: ASMD diagram of main FSM component of the case without dummy data handling.

signal. As per the UART protocol, the received byte with MSB of $'0'$ specify the address of the SIB and instrument to be activated and which command it must execute. The addresses are deciphered and the `SIB_control_reg` is set in the `REC_ADDRESS` and `UPDATE_ADDRESS` states.

Also, as per the protocol, an MSB of $'1'$ on a received UART bytes indicates that the re-targeting tool has sent the addresses for all the SIBs that are to be activated and the 1687 FSM can begin operation and generate the shift-in sequence. This is achieved by toggling the control signal `control_ready`. The FSM transitions to `REC_BYTE` and `READ_CHECK` states to determine if the number data bytes that are not addresses for instruments is zero. This is because, for *iRead* commands, no data bytes must be transmitted. Therefore, if the number is zero at this stage, it can be safely assumed that only read commands are to be executed and the main FSM can start shifting in the bit sequence without waiting for data from the re-targeting tool. The FSM toggles control signal, `data_ready` and transitions to `REC_HOLD` state.

On the other hand, if one or more commands were *iWrite*, the number of data bytes would have a non-zero value. In such cases, the FSM transitions to `REC_DATA`, where it waits for data bytes to arrive before toggling control signal, `data_ready`. If multiple data bytes are expected, the control signal is toggled multiple times.

Because the Output Discard Unit (ODU) does not exist in this case, all outputs from the 1687 network must be processed alongside the useful ones (see fig. 3.14). Since the UART standard can transmit a maximum of 8 bits and is much slower than hardware, additional logic is required to store incoming bits and stall the network while the outputs are being transmitted. This FSM fulfills that purpose.

The default state of the FSM is *IDLE*, where it waits for the control commands to be transmitted by the re-targeting tool. Once the instruments to be activated and the associated control commands have been deciphered, the interpreter FSM toggles the `control_ready` signal which triggers a state transition to `RECIEVE_OP` in this FSM. In this state, the FSM is ready to capture output bits from the network and store them in a register. Once 8 bits, the maximum capacity of the UART standard, have been captured, the FSM transitions to `UART_DATA`. In this state, the FSM stalls the main FSM in charge of shifting bits into the network and the stored byte is transmitted to the re-targeting tool through the UART channel. Once acknowledgment has been received that the byte has been transmitted, the FSM transitions back to `RECIEVE_OP` and restarts the main FSM. This continues till all the output bits have been transmitted and the main FSM completes operation.

Note that there is no logic in this FSM to decipher the output bits it stores, meaning it cannot differentiate between useful outputs and garbage bits. That falls under the purview of the re-targeting tool.

The model presented above is enough for *iRead* control command where there are no data bytes transmitted by the re-targeting tool. However, for *iWrite* commands, data bytes will be transmitted to the hardware after the configuration bytes with the addresses. This could lead to scenarios in which the data is lost because the output FSM stalled the main FSM while transmitting output bits. In order to account for this eventuality, the `WRITE_WAIT` state was introduced.

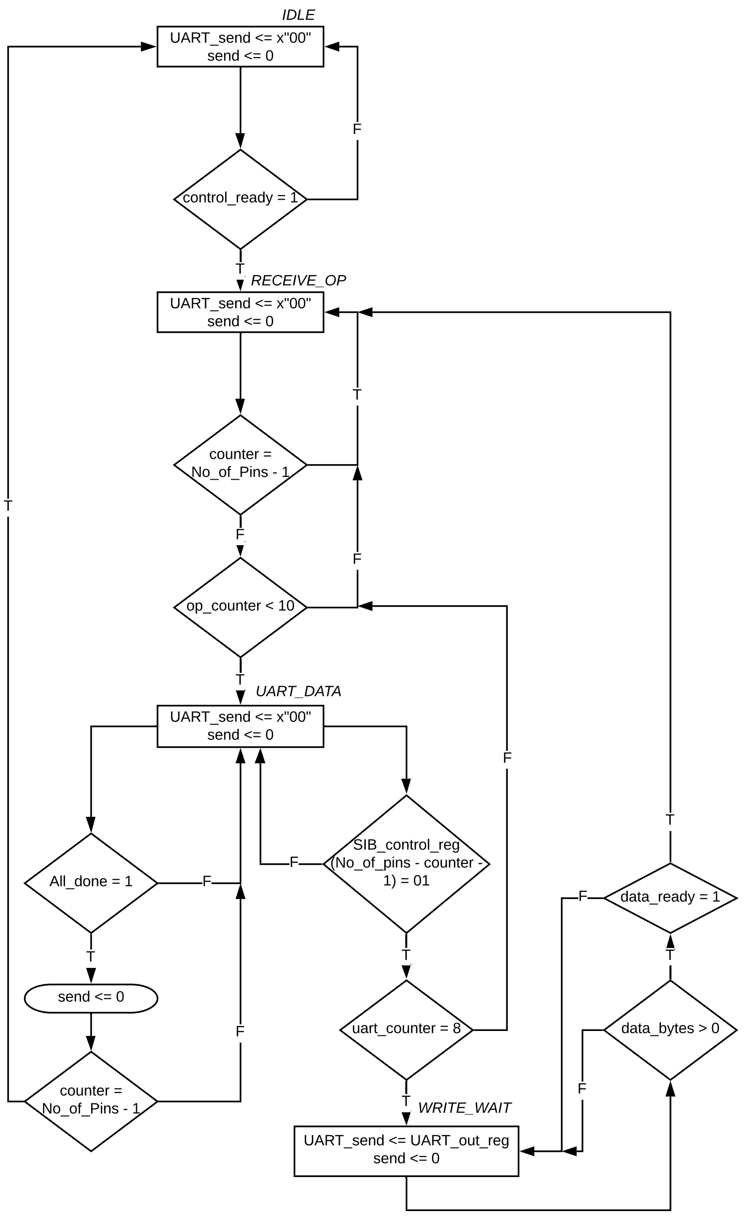


Figure 3.14: ASMD diagram of output FSM component of the case without dummy data handling.

In cases where the re-targeting tool is still transmitting a byte of data, the FSM transitions from `RECIEVE_OP` to `WRITE_WAIT` directly. This means that nothing is shifted out of the network making it impossible to lose data. After data is successfully received, denoted by control signal, `data_ready`, the FSM transition back to `RECIEVE_OP` to continue receiving output bits.

3.4.2 Re-targeting flow for the case

Compared to the previous case, due to the presence of additional components such as SIB Control Register and SIB Control Register we can further optimize our algorithm (see Algorithm 3) by removing configuration and padding bits, since those are already being taken care of by improved FSM. Our main focus here is to reduce PDL and SIB overheads.

Again, depending on which setup command is being entered we can decide whether to generate dummy bits or translate `instrumentData` into string of command bits, which in our case is called `setupBits`. Let us take closer look to the bit-stream, presented in Figure 3.15.

Byte\Bit	**b$_7$**	**b$_6$**	**b$_5$**	**b$_4$**	**b$_3$**	**b$_2$**	**b$_1$**	**b$_0$**
4	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

0	0	***Setup command*** *which specifies an* ***iRead*** *command*
1	0	***Apply command*** *which specifies an* ***iApply*** *command*

Figure 3.15: Bit-wise description of *iRead* setup command for the case without Dummy Data Handling.

In comparison to the previous re-targeting flow (the Naive case) we have a clear distinction between setup command sequence and action command sequence. Here for example, first 2 bytes are representing *iRead* command, starting with 2 bits set to $'00'$ which according to our protocol indicates the read operation of instrument 1. Since we do not shifting any data bits to the system, we set LSB to $'0'$ for the action command sequence. This in turn tells an FSM, that no data is being shifted in to the network. Notice how drastically was reduced the data overhead, since re-targeting tool is no longer needs to manually perform shift and capture operations for FSM.

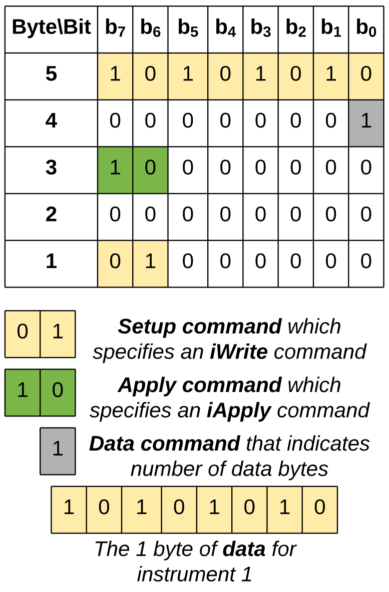
For write sequence, PDL is translated into stream of bits, shown in Figure 3.16, which starts with leading $'01'$ indicating that write setup command has been applied. However, when we enter an *iApply* command, we have to state how many bits of data is being sent to the network. This is done by setting LSB to the size of incoming data bits, which in our case is equal to 1 byte or $0b00000001$ in binary notation, also denoted as *data command*. After the size of data bits are sent, we

Algorithm 3 Setup commands for case without dummy data handling

Require: Instrument address and instrument data
Ensure: Setup and dummy/data bits

```
1: $totalInstruments \leftarrow 3$                                  ▷ Network with 3 instruments

2: function iRead($instrumentAddress$)
3:     $setupBits \leftarrow (0 \ll 15) \lor instrumentAddress$
4:     if $instrumentAddress \bmod 3 = 0$ then                   ▷ generate dummy bits
5:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 7)$              ▷ 1 byte
6:     else if $instrumentAddress \bmod 3 = 1$ then
7:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 15)$             ▷ 2 bytes
8:     else
9:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 31)$             ▷ 4 bytes
10:    end if
11:    return $setupBits, dummyBits$  ▷ return setup and dummy bits to action command for further processing
12: end function

13: function iWrite($instrumentAddress$, $instrumentData$)
14:    $setupBits \leftarrow (1 \ll 14) \lor instrumentAddress$
15:    if $instrumentAddress \bmod 3 = 0$ then                ▷ translate into data bits
16:        $dataBits[instrumentAddress] \leftarrow instrumentData$          ▷ 1 byte
17:    else if $instrumentAddress \bmod 3 = 1$ then
18:        $dataBits[instrumentAddress] \leftarrow instrumentData$          ▷ 2 bytes
19:    else
20:        $dataBits[instrumentAddress] \leftarrow instrumentData$          ▷ 4 bytes
21:    end if
22:    return $setupBits, dataBits$      ▷ return configuration and data bits
23: end function
```

send actual data bits, which sets instrument 1 with value $0xAA$.



Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
5	1	0	1	0	1	0	1	0
4	0	0	0	0	0	0	0	1
3	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0

0 1
Setup command *which specifies an **iWrite** command*
1 0
Apply command *which specifies an **iApply** command*
1
Data command *that indicates number of data bytes*
1 0 1 0 1 0 1 0
*The 1 byte of **data** for instrument 1*

Figure 3.16: Bit-wise description of *iWrite* setup command for the case without Dummy Data Handling.

Later, when all setup commands are sent, we finish the transmission with *iApply* action command, which is shown in Algorithm 4.

Since, output data is not refined by *discard unit* we do the same search and extract sequence as we did for the Naive case. The output result is presented in Figure 3.17.

The output for current case is the identical to the Naive case, therefore we still face a lot of output data overhead, In the next section we describe how to circumvent this issue by introducing Output Discard Unit component to our design.

3.5 Full-featured case

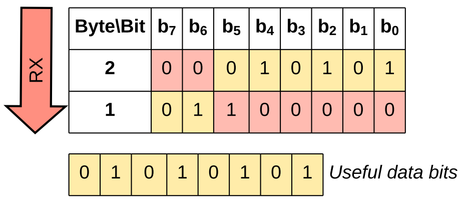
Full-featured case (see fig. 3.18) has all the necessary components to efficiently interact with the 1687 network. In full-featured case we introduced *Discard Unit* component (or Output Discard Unit (ODU)), which filters out any irrelevant data during the output phase.

3.5.1 Hardware description of the case

The main FSM and UART interpreter are absolutely identical to the case without DDH with an exception of presence of Output Discard Unit component, which discards an output overhead and produces only useful bits requested by re-targeting tool.

Algorithm 4 Action command for case without dummy data handling

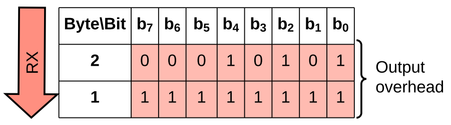
```
Require: Setup and dummy/data bits
Ensure: Shift respective lengths of setup and data/dummy bits into the
    network and send setup and data commands

 1: $expectedData = ”01010101”$        ▷ Expected instrument data, used to
    verify read command
 2: $instrumentLength = 8$             ▷ Instrument length of the activated
    instrument (in bits)
 3: $bufferSize = 1024$   ▷ 1KB sized buffer to store incoming UART data

 4: function iApply($command$)
 5:     if $command = iWrite$ then
 6:         $dataSize \leftarrow (1 \ll 15) \vee$ (length($dataBits$) $\div 8$)
 7:         serialShift($setupBits$)                  ▷ Shift setup,
 8:         serialShift($dataSize$)          ▷ data, action command
 9:         serialShift($dataBits$)     ▷ and data bits into the network
10:     else if $command = iRead$ then
11:         $dummySize \leftarrow (1 \ll 15)$
12:         serialShift($setupBits$)                  ▷ Shift setup,
13:         serialShift($dummySize$)         ▷ data, action command
14:         serialShift($dummyBits$) ▷ and dummy bits into the network
15:         $incomingData$ = serialCapture($bufferSize$)       ▷ Receive
    incoming data
16:         compare($incomingData, expectedData, instrumentLength$) ▷
    and verify its correctness
17:     end if
18: end function
```



Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
2	0	0	0	1	0	1	0	1
1	0	1	1	0	0	0	0	0

0	1	0	1	0	1	0	1	*Useful data bits*

(a) Captured result of *iRead* command.



Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
2	0	0	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1

RX
Output overhead

(b) Captured result of *iWrite* command.

Figure 3.17: Outputs of apply groups for the case without DDH.

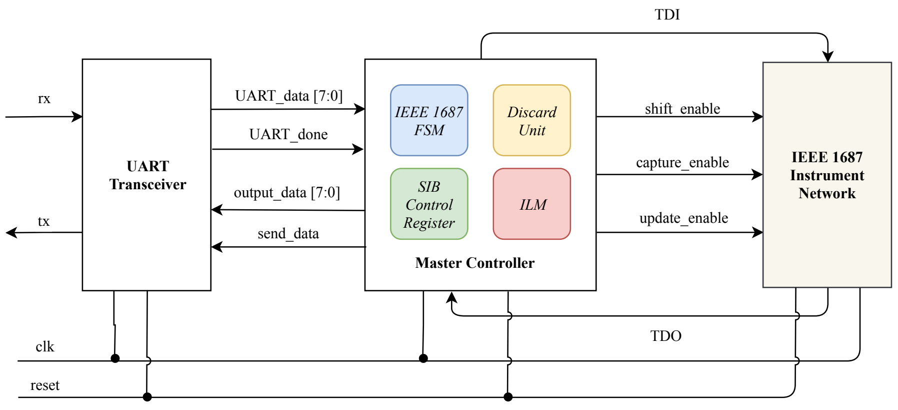


Figure 3.18: Conceptual diagram of full-featured case.

To conclude all cases, we will discuss UART transceiver FSM, shown in Figure 3.19, which is part of all discussed cases. The FSM stores incoming UART bits by polling the `rx` line at intervals specified by the `SAMPLING_RATE` constant. Note that this FSM only stores incoming data into byte sized registers and transmits them to the controller for processing. It does not interpret the bytes in any fashion.

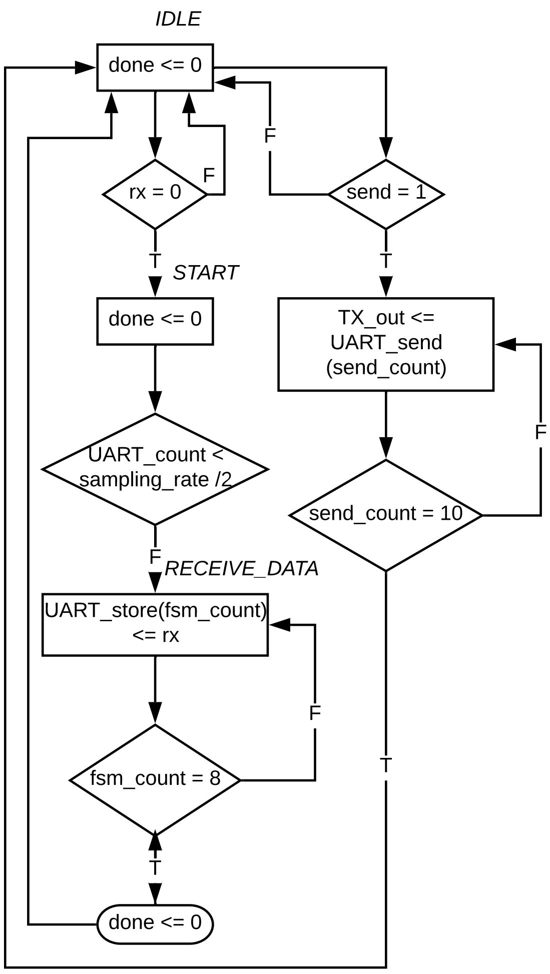


Figure 3.19: UART transceiver FSM module.

The FSM remains in its default state of `IDLE` until `rx` is toggled to ′0′, signifying imminent transmission of data. The FSM transitions to the `START` state where it confirms that a transmission is occurring.

In `RECIEVE_DATA` state, the FSM stores the incoming bits to form a byte as per the UART standard. Once a full byte has been stored, the FSM toggles control signal, `done` to inform the controller that a byte of UART data has been successfully received and that the controller can begin processing it. The FSM then transitions back to `IDLE` to wait for the next byte to arrive.

While transmitting data through `TX_out`, the FSM transitions to `SEND_DATA` when the controller toggles control signal, `send` that signifies that a byte is ready to be transmitted back to the re-targeting tool. Once the byte is transmitted the FSM transitions back to `IDLE`.

3.5.2 Re-targeting flow for the case

By introducing ODU we managed to size down output overhead to zero. Final result is shown in Algorithm 5 and Algorithm 6. The setup and action commands are same as in the case without Dummy Data Handling.

Algorithm 5 Setup commands for full-featured case

```
Require: Instrument address and instrument data
Ensure: Setup and dummy/data bits

 1: $totalInstruments \leftarrow 3$                                  ▷ Network with 3 instruments

 2: function iRead($instrumentAddress$)
 3:     $setupBits \leftarrow (0 \ll 15) \vee instrumentAddress$
 4:     if $instrumentAddress \bmod 3 = 0$ then                    ▷ generate dummy bits
 5:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 7)$              ▷ 1 byte
 6:     else if $instrumentAddress \bmod 3 = 1$ then
 7:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 15)$             ▷ 2 bytes
 8:     else
 9:         $dummyBits[instrumentAddress] \leftarrow (0 \ll 31)$             ▷ 4 bytes
10:     end if
11:     return $setupBits, dummyBits$   ▷ return setup and dummy bits to action command for further processing
12: end function

13: function iWrite($instrumentAddress$, $instrumentData$)
14:     $setupBits \leftarrow (1 \ll 14) \vee instrumentAddress$
15:     if $instrumentAddress \bmod 3 = 0$ then                 ▷ translate into data bits
16:         $dataBits[instrumentAddress] \leftarrow instrumentData$          ▷ 1 byte
17:     else if $instrumentAddress \bmod 3 = 1$ then
18:         $dataBits[instrumentAddress] \leftarrow instrumentData$          ▷ 2 bytes
19:     else
20:         $dataBits[instrumentAddress] \leftarrow instrumentData$          ▷ 4 bytes
21:     end if
22:     return $setupBits, dataBits$        ▷ return configuration and data bits
23: end function
```

Algorithm 6 Action command for full-featured case

Require: Setup and dummy/data bits
Ensure: Shift respective lengths of setup and data/dummy bits into the network and send setup and data commands

```
 1: $bufferSize = 1024$    ▷ 1KB sized buffer to store incoming UART data

 2: function iApply($command$)
 3:     if $command = iWrite$ then
 4:         $dataSize \leftarrow (1 \ll 15) \vee (\text{LENGTH}(dataBits) \div 8)$
 5:         serialShift($setupBits$)                    ▷ Shift setup,
 6:         serialShift($dataSize$)                     ▷ data, action command
 7:         serialShift($dataBits$)                     ▷ and data bits into the network
 8:     else if $command = iRead$ then
 9:         $dummySize \leftarrow (1 \ll 15)$
10:         serialShift($setupBits$)                    ▷ Shift setup,
11:         serialShift($dummySize$)                    ▷ data, action command
12:         serialShift($dummyBits$)                    ▷ and dummy bits into the network
13:         $incomingData = \text{serialCapture}(bufferSize)$    ▷ Receive incoming data
14:     end if
15: end function
```

In the end, we receive output only during a read sequence, shown in Figure 3.20:

Byte\Bit	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	0	1	0	1	0	1

Figure 3.20: Result of *iRead* instrument 1 apply command for the full-featured case.

Chapter 4 Results

In this chapter, we present the experiments performed to compare proposed design alternatives. The narrative will be build from bit-banging (purely software) case to full-featured (dominantly hardware) case, as we did in the previous chapter. We also discuss testing methodology and make inferences based on the results obtained.

4.1 Testing methodology

For the experiments, we calculated three data overhead components, namely:

- *SIB overhead* - overhead bits required to configure SIBs.
- *PDL overhead* - overhead bits required to perform setup commands, generated by the re-targeting tool.
- *Output overhead* - unwanted garbage bits, shifted out of the network that are not considered as data bits.

Summation of the listed overhead components produces total data overhead metric:

$$\begin{aligned} Total\ overhead = &\sum_{i=1}^{N} SIB\ overhead\ + \\ &+\sum_{i=1}^{N} PDL\ overhead\ + \\ &+\sum_{i=1}^{N} Output\ overhead \end{aligned} \tag{4.1}$$

where N is total number of instruments present in the IEEE 1687 instrument network.

Furthermore, in order to calculate useful data percentage, i.e. actual data bits, we use total bits exchanged metric, which indicates the total amount of bits transmitted through the function port:

$$Total\ bits\ exchanged = Total\ overhead + \sum_{i=1}^{N} Useful\ bits \tag{4.2}$$

$$Useful\ data\ percentage = \frac{Useful\ bits}{Total\ bits\ exchanged} \times 100 \tag{4.3}$$

Next, we have 3 instrument network sizes: $N = 50$, 100 and 150 to check scalability of the IEEE 1687 instrument network.

Additionally, we have tested five PDL apply groups, specifically:

- Read from the first instrument (denoted as *iRead 1*);
- Write to the first instrument (denoted as *iWrite 1*);
- Read from all instruments inside the network (denoted as *Read all*);
- Write to all instruments inside the network (denoted as *Write all*);
- `BASTION` benchmark for flat network (denoted as *BASTION*), shown in Listing 4.1 [14].

```
/* Access all instruments in single iApply group */
iProc all_instruments_in_one_iApply {}{
     iWrite(totalInstruments, "10101010");
     ...
     iWrite(0, "10101010");
     iApply;

     iRead(totalInstruments, "01010101");
     ...
     iRead(0, "01010101");
     iApply;
}
/* Access instruments in individual manner */
iProc one_instrument_per_iApply {}{
     iWrite(totalInstruments, "10101010");
     iApply;
     iRead(totalInstruments, "01010101");
     iApply;
     ...
     iWrite(0, "10101010");
     iApply;
     iRead(0, "01010101");
     iApply;
}
```

Listing 4.1: PDL for the `BASTION` benchmark

`BASTION` benchmark was developed by European collaborative research project BASTION in 2014 to investigate currently unknown issues and study aging of embedded electronic instruments [15]. Since we have implemented a flat network, we decided to test `TreeFlat` benchmark, which basically performs read and write operations in a single apply group and same sequence in separate apply groups [14].

As for area utilization of proposed designs, we use resource consumption metric of an FPGA extracted from resource utilization report, generated by the Xilinx EDA. Each FPGA manufacturer uses different techniques to report resource utilization, but in our case we use Configurable Logic Block (or logic block, LB for short) which consists of two logic slices (Slice M and Slice L) and represents fundamental building block of the Xilinx FPGA fabric [16]. The equation for calculating LB is shown below:

$$LB = 8 \times LUT + 16 \times FF \tag{4.4}$$

The results of resource report presented in Table 4.1.

4.2 Bit-banging case

The testing for bit-banging case is done as a thought experiment. Let us consider read operation for one instrument. In Figure 3.3 read action command is performed. During shift configuration phase, 2 bytes are required to toggle clock line, 1 byte is required to send 1 bit of data, 2 bytes of data are required to send configuration bits and 2 bytes of useless output bits are generated by the network. Next, during update phase, we send 2 bytes to update the SIBs. After that, we send capture signal, which also requires 2 bytes. During Shift-Update, no output data is shifted out for read operation. The data overhead equation for read operation is shown below:

$$\begin{aligned} Total\ overhead = {} & 2 \times 8 \times 2 \times (2 \times Configuration\ bits) + \\ & + 2 \times 8 \times (Dummy\ bits) + \\ & + 2 \times 8 \times (UC) \end{aligned} \tag{4.5}$$

where UC denotes an update-capture phase.

For write operation, during the configuration shift phase, we 1 byte of configuration bits and receive back useless output bits, then during update phase we shift in 2 bytes to update contents of instruments, but no output is received back. Next, during data shift phase we shift in both configuration bits followed by data bits and shift out useless output data. Needless to say, that each time we send configuration bit the network generates corresponding output bit. Lastly, we finish the flow by shifting 2 bytes for update. The data overhead equation for write operation is shown below:

$$\begin{aligned} Total\ overhead = {} & 2 \times 8 \times 2 \times (U) + \\ & + 2 \times 8 \times 2 \times (2 \times Configuration\ bits + Data\ bits) \end{aligned} \tag{4.6}$$

where U denotes update bits.

Table 4.2 presents the results for the experiments in details.

Table 4.1: Resource consumption for all cases in terms of logic blocks.

	Resource consumption														
Instruments	**1687 network**			**w/o DDH**			**w/o ILM**			**Naive**			**Full-featured**		
	FF	LUT	LB	FF	LUT	LB	FF	LUT	LB	FF	LUT	LB	FF	LUT	LB
50	1946	2006	369	400	852	103	400	739	96	193	330	45	400	801	100
100	3918	4036	742	500	1060	129	500	881	118	193	330	45	502	959	123
150	5906	6082	1118	600	1158	147	600	1046	140	193	330	45	600	1109	144

Table 4.2: Test results for bit-banging case.

Number of instruments	Apply Group	Total bits exchanged	Number of useful bits	Useful data percentage
50	iRead 1	1776	8	0.50%
	iWrite 1	1776	8	0.50%
	Write all	16368	920	5.60%
	Read all	16368	920	5.60%
	BASTION	222128	3680	1.66%
100	iRead 1	3376	8	0.24%
	iWrite 1	3376	8	0.24%
	Write all	32944	1856	5.60%
	Read all	32944	1856	5.60%
	BASTION	765232	7424	0.97%
150	iRead 1	4976	8	0.16%
	iWrite 1	4976	8	0.16%
	Write all	49648	2800	5.60%
	Read all	49648	2800	5.60%
	BASTION	1628848	11200	0.69%

4.3 Naive case

In previous chapter, we discussed in detail functionality of the Naive case. In this section, we will discuss calculation of data overhead components. The *PDL overhead* is equal to 24 bits (or 3 bytes). According to UART protocol, configuration size is limited to 8 bits and data size is limited to 16 bits for single apply group, therefore:

$$PDL\ overhead = Configuration\ size + Data\ size = 24\ bits \tag{4.7}$$

The value of *SIB overhead* directly depends on which instruments are currently active, their respective ILM sizes and number of useful bits (for *iWrite*) or dummy bits (for *iRead*). The general equation for calculating *SIB overhead* is shown below:

$$SIB\ overhead = \sum_{i=0}^{M} Configuration\ bits + \sum_{i=0}^{M} Data\ bits \tag{4.8}$$

Note that when *iWrite* setup command is applied, data bits must be subtracted from the equation, because those bits are considered as useful. Hence, we get following equation for *iWrite* apply group:

$$PDL\ overhead = \sum_{i=0}^{M} Configuration\ bits + \sum_{i=0}^{M} Data\ bits - Data\ size \tag{4.9}$$

In the end, after PDL commands are shifted in, we receive output bits. If write operation is performed, all incoming bits are considered as an *Output overhead*. On the other hand, if read operation is performed, re-targeting tool filters out useful data bits and discards remaining bits as an overhead.

Finally, we get following results, shown in Table 4.3:

Table 4.3: Test results for Naive case.

Number of instruments	Apply Group	PDL overhead	SIB overhead	Output overhead	Total overhead	Number of useful bits	Useful data percentage
50	iRead 1	24	120	111	255	8	3.00%
	iWrite 1	24	112	118	254	8	3.10%
	Write all	24	112	1031	1167	920	44.10%
	Read all	24	1032	226	1282	920	41.80%
	BASTION	2448	13152	13702	29302	3680	11.20%
100	iRead 1	24	216	205	445	8	1.20%
	iWrite 1	24	208	212	444	8	1.80%
	Write all	24	208	2063	2295	1856	44.70%
	Read all	24	2064	439	2527	1856	42.30%
	BASTION	4848	45520	45692	96060	7424	7.20%
150	iRead 1	24	312	349	685	8	1.20%
	iWrite 1	24	304	356	684	8	1.20%
	Write all	24	304	3103	3431	2800	44.90%
	Read all	24	3104	1245	4373	2800	39.00%
	BASTION	7248	97104	96758	201110	11200	5.30%

4.4 Case without Dummy Data Handling

In this section, we will report the test results for case without dummy data handling. This is the case, when we introduced first crucial components to improve re-targeting data efficiency, i.e SIB Control Register and Instrument Length Memory.

By introducing above mentioned components, SIB overhead and PDL overhead bits get dramatically reduced. For instance, $SIB\ overhead$ for both $iRead$ and $iWrite$ setup commands is 16 bits and $PDL\ overhead$ is equal to 16 bits for each action command. The output overhead however is identical to the Naive's case. That is because ODU is absent, therefore re-targeting tool has to discard an output independently. Results for this case is presented in Table 4.4.

4.5 Case without Instrument Length Memory

This case is quite identical in many respects to full-featured case, that is ODU is present and output overhead is discarded by the master controller. The only difference in this case is absence of ILM component. $PDL\ overhead = 16$ bits, which is the same as both in full-featured case and case without DDH, however since ILM serves a purpose of storing length of the instrument (because in our proposed architecture instruments have variable ILMs), re-targeting tool must also send additional byte describing the length of the instrument accessed. Therefore, $SIB\ overhead = 24$ bits for each setup command. Notwithstanding, output overhead is equal to zero. The results for this case is shown in Table 4.5.

4.6 Full-featured case

The full-featured case is most revised case with all components present. $PDL\ overhead$ and $SIB\ overhead$ for each setup and action commands are equal to 16 bits, which the same as in no DDH case. Also, since ODU is present no output overhead is produced. Thus, we get following results, presented in Table 4.6.

Table 4.4: Test results for no dummy data handling case.

Number of instruments	**Apply Group**	**PDL overhead**	**SIB overhead**	**Output overhead**	**Total overhead**	**Number of useful bits**	**Useful data percentage**
50	iRead 1	16	16	111	143	8	5.30%
	iWrite 1	16	16	118	150	8	5.30%
	Write all	16	800	974	1790	920	33.90%
	Read all	16	800	170	986	920	48.30%
	BASTION	1632	3200	2290	7122	3680	34.10%
100	iRead 1	16	16	205	237	8	3.30%
	iWrite 1	16	16	212	244	8	3.20%
	Write all	16	1600	1958	3574	1856	34.20%
	Read all	16	1600	335	1951	1856	48.80%
	BASTION	3232	6400	4586	14218	7424	34.30%
150	iRead 1	16	16	299	331	8	2.40%
	iWrite 1	16	16	306	338	8	2.30%
	Write all	16	2400	2950	5366	2800	34.30%
	Read all	16	2400	501	2917	2800	49.00%
	BASTION	4832	9600	6904	21336	11200	34.40%

Table 4.5: Test results for no instruction length memory case.

Number of instruments	**Apply Group**	**PDL overhead**	**SIB overhead**	**Output overhead**	**Total overhead**	**Number of useful bits**	**Useful data percentage**
50	iRead 1	16	24	0	40	8	16.70%
	iWrite 1	16	24	0	40	8	16.70%
	Write all	16	1200	0	1216	920	43.10%
	Read all	16	1200	0	1216	920	43.10%
	BASTION	1632	4800	0	6432	3680	36.40%
100	iRead 1	16	24	0	40	8	16.70%
	iWrite 1	16	24	0	40	8	16.70%
	Write all	16	2400	0	2416	1856	43.40%
	Read all	16	2400	0	2416	1856	43.40%
	BASTION	3232	9600	0	12832	7424	36.70%
150	iRead 1	16	24	0	40	8	16.70%
	iWrite 1	16	24	0	40	8	16.70%
	Write all	16	3600	0	3616	2800	43.60%
	Read all	16	3600	0	3616	2800	43.60%
	BASTION	4832	14400	0	19232	11200	36.80%

Table 4.6: Test results for full-featured case.

Number of instruments	Apply Group	PDL overhead	SIB overhead	Output overhead	Total overhead	Number of useful bits	Useful data percentage
50	iRead 1	16	16	0	32	8	20%
	iWrite 1	16	16	0	32	8	20%
	Write all	16	800	0	816	920	53%
	Read all	16	800	0	816	920	53%
	BASTION	1632	3200	0	4832	3680	43%
100	iRead 1	16	16	0	32	8	20%
	iWrite 1	16	16	0	32	8	20%
	Write all	16	1600	0	1616	1856	53.50%
	Read all	16	1600	0	1616	1856	53.50%
	BASTION	3232	6400	0	9632	7424	44%
150	iRead 1	16	16	0	32	8	20%
	iWrite 1	16	16	0	32	8	20%
	Write all	16	2400	0	2416	2800	53.70%
	Read all	16	2400	0	2416	2800	53.70%
	BASTION	4832	9600	0	14432	11200	44%

4.7 Inferences

4.7.1 Full-featured case versus case without DDH

Looking at Figure 4.1, we observe that output overhead parameter is absent in full-featured case.

For a relatively low area consumption, we could infer that including ODU proves to be a net benefit. Based on results obtained in Tables 4.4 and 4.6, the impact of ODU on the write setup commands is greater compared to read commands. That is because in case of write setup command, none of the output bits are useful. By looking at `BASTION` apply group, we can conclude that useful data percentage remains constant with increase of number of instruments.

4.7.2 Full-featured case versus case without ILM

Looking at Figure 4.2 we can conclude that for same area cost, we get higher returns in data overhead as number of instruments increases.

This proves that the impact of ILM scales well and including the components improves data transfer efficiency.

4.7.3 Full-featured case versus case without ILM

Looking at Figure 4.3 it is evident, that the total overhead falls sharply as we move from bit-banging to full-featured case. Hence, we can prove that full-featured implementation provides the best efficiency in terms of data overhead. It is especially beneficial as the number of instrument increases.

As a final remark, we can say that embedded solution, which splits the retargeting functionality between hardware and software components, is the most beneficial when UART port interface is used.

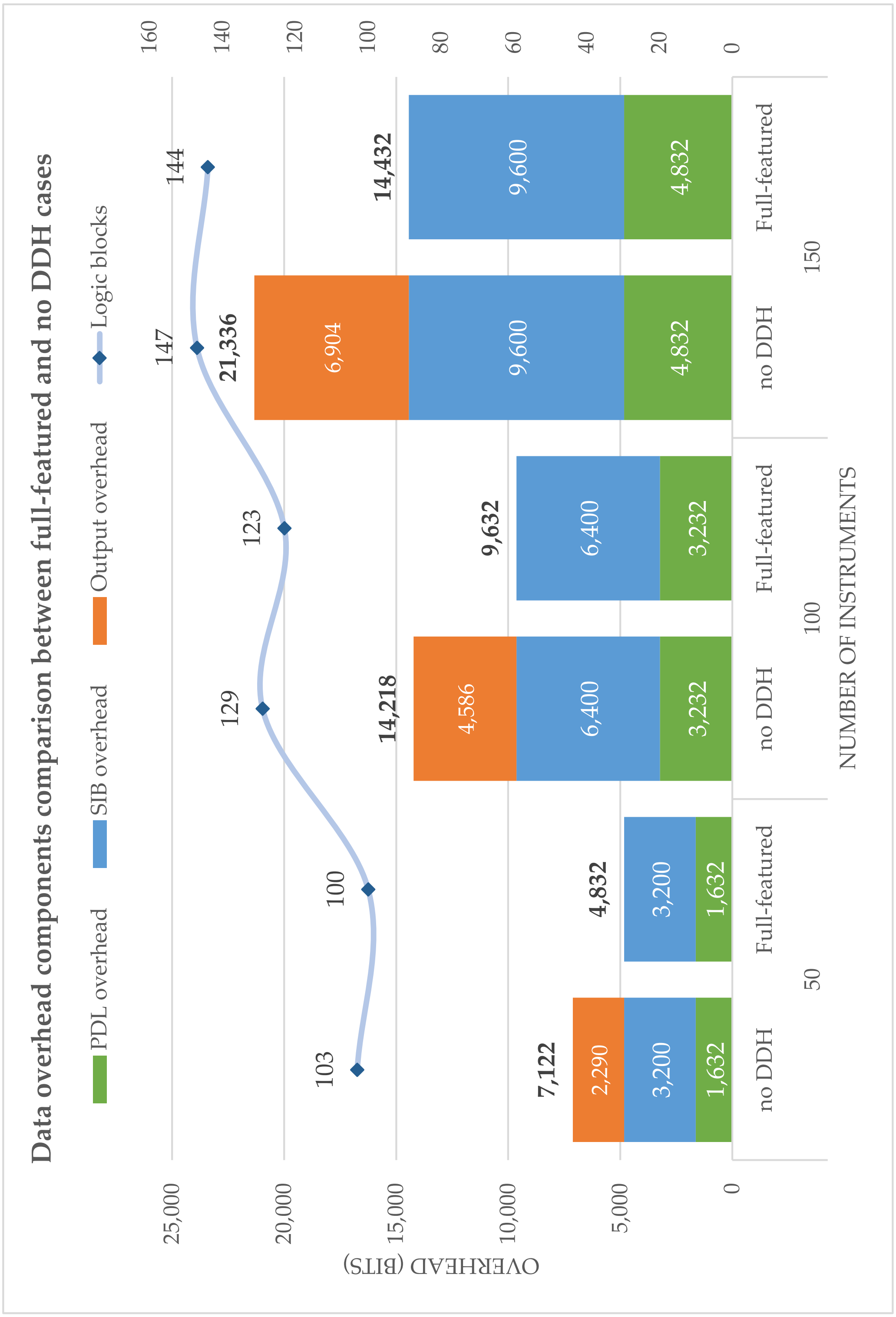


Figure 4.1: Data overhead comparison between full-featured and without DDH cases.

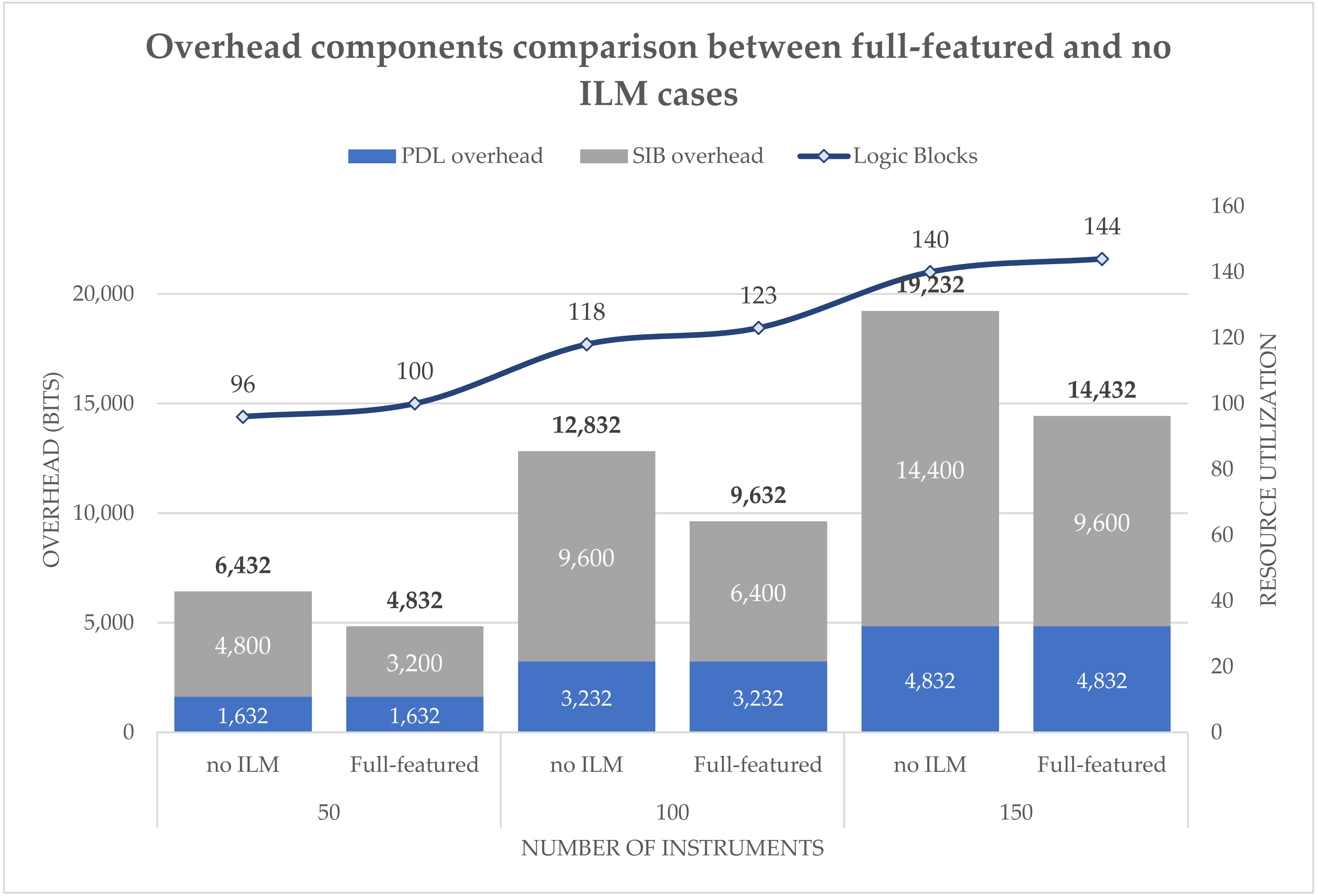


Figure 4.2: Data overhead comparison between full-featured and without ILM cases.

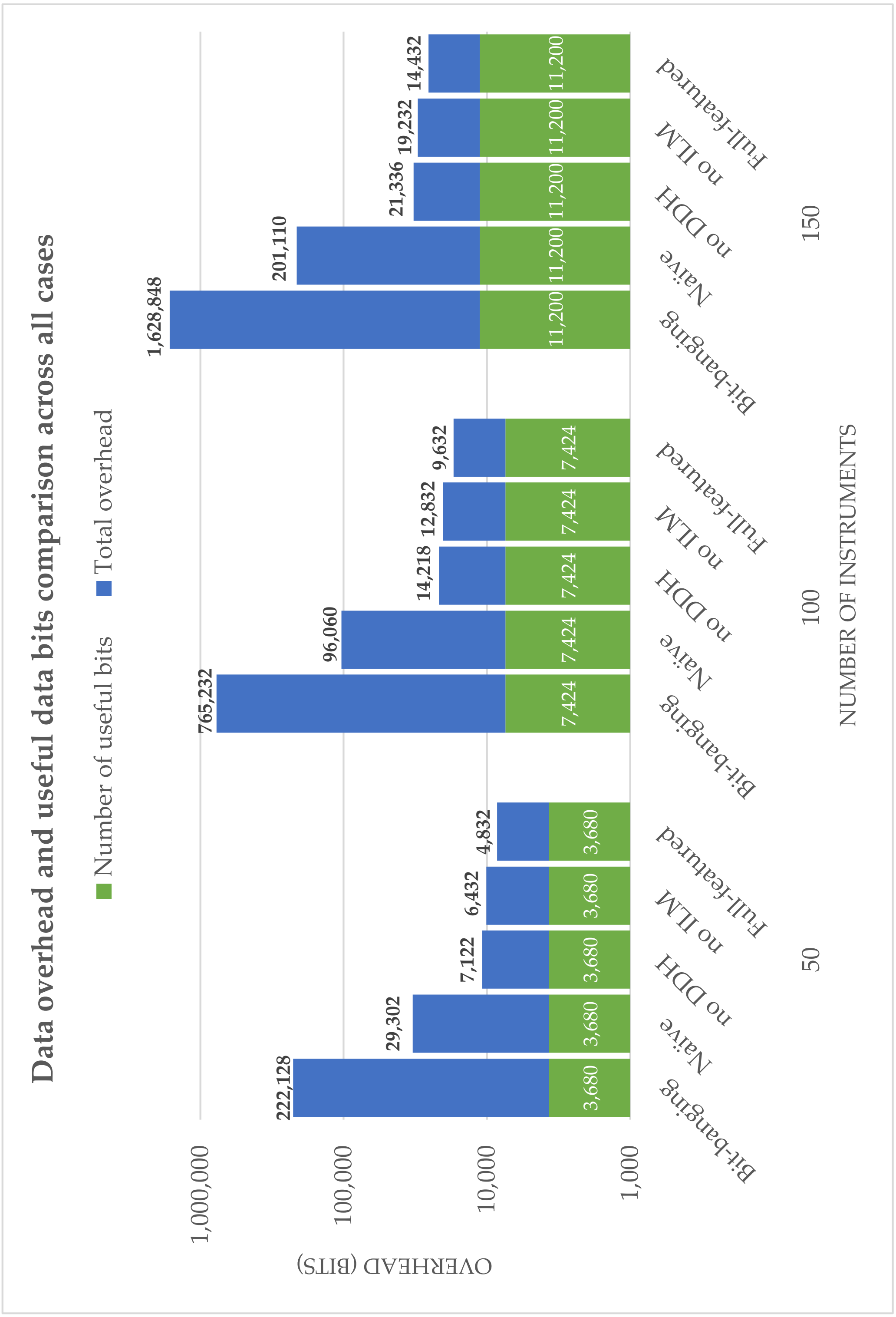


Figure 4.3: Data overhead comparison across all cases.

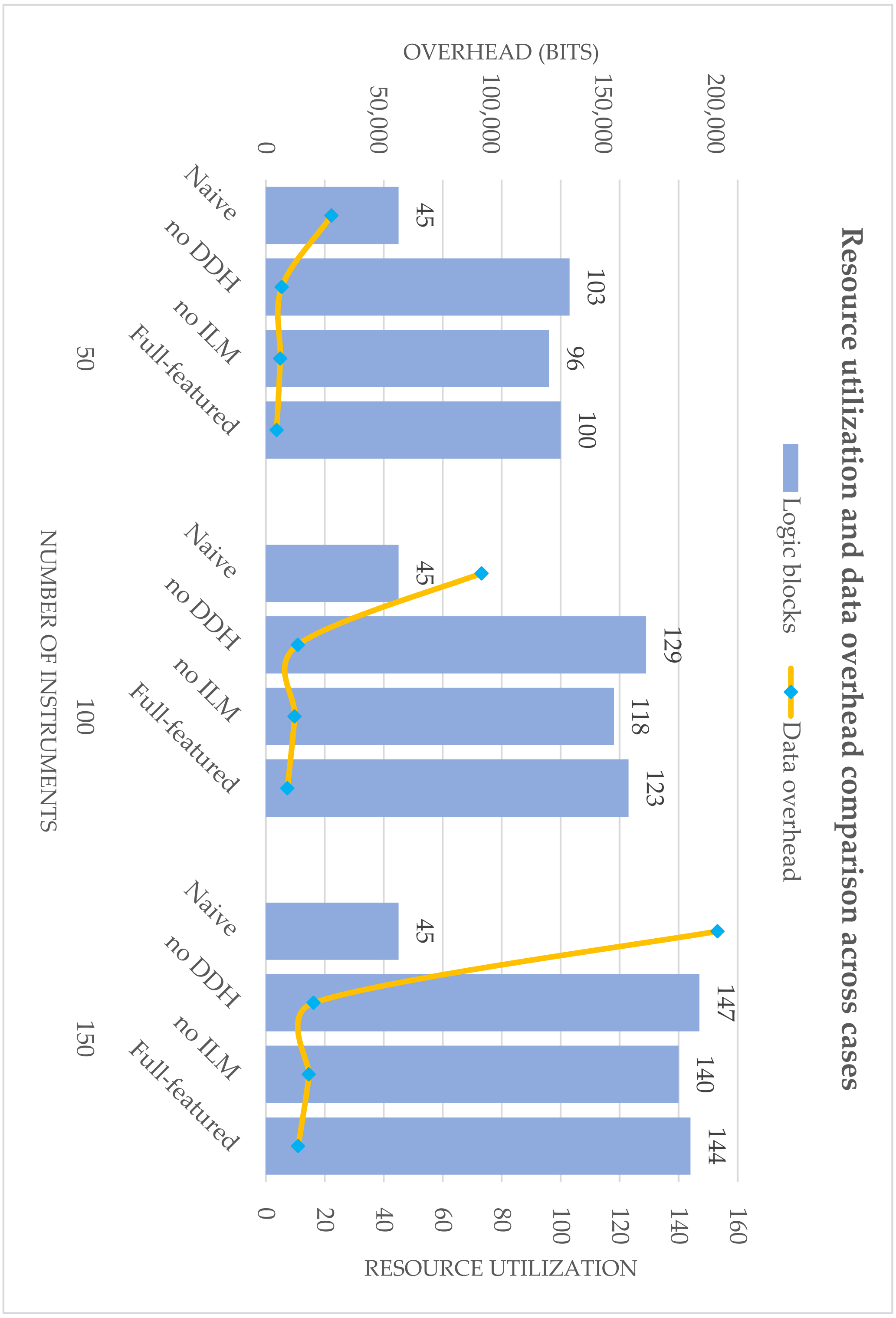


Figure 4.4: Resource utilization scaling with respect to data overhead.

Chapter 5

Conclusion

In this chapter, we present summary for each chapter and suggest future improvement.

5.1 Introduction and theory

In these chapters, we did literature review for IEEE 1687 network and were exploring possible design alternatives. We established an objective and elaborated on the results we expected to obtain.

We discussed the UART standard and defined its constituent components. Next, we explained TAP and TAP controller architecture and state machine. Furthermore, we reviewed IEEE Std. 1687 IJTAG and various associated concepts.

Finally, we quantified performance the metrics we would be measuring and established various design alternatives.

5.2 Methodology

In this chapter, we explained our implementation starting from bit-banging case, which simulated TAP interface. We then progressively added each of the components, proposed in previous chapter, i.e. SCR, ILM and ODU and explored re-targeting flow as well as UART protocol for each consecutive case.

Additionally, we presented ASMD diagrams, pseudo-codes, bit-wise diagrams and block diagrams associated with each case.

5.3 Results

In this chapter, we provided brief overview of PDL specified by `BASTION` benchmark. Next, we presented results for various cases in form of tables, charts and equations. Furthermore, we made inferences based on results we obtained and made appropriate remarks.

5.4 Future work

In this thesis, we made implementation based only on flat instrument access network, however our work could be expanded to accommodate other design approaches, namely, hierarchical networks, multiple networks, daisy-chained network, etc. Moreover, our work could be used as a frame of reference for implementations that use other functional port interfaces such as I^2C, SPI, AMBA and others.

Additionally, our work could also be further expanded for fault detection and management applications.

References

[1] Erik Larsson and Farrokh Ghani Zadegan. "Accessing embedded DfT instruments with IEEE P1687". In: *2012 IEEE 21st Asian Test Symposium*. IEEE. 2012, pp. 71–76.

[2] Colin M Maunder and Rodham Tulloss. *The test access port and boundary-scan architecture*. IEEE Computer Society Press, 1990.

[3] Farrokh Ghani Zadegan. "Reconfigurable On-Chip Instrument Access Networks: Analysis, Design, Operation, and Application". In: (2017).

[4] Farrokh Ghani Zadegan et al. "Design, verification, and application of IEEE 1687". In: *2014 IEEE 23rd Asian Test Symposium*. IEEE. 2014, pp. 93–100.

[5] Artur Jutman, Sergei Devadze, and Konstantin Shibin. "Effective scalable IEEE 1687 instrumentation network for fault management". In: *IEEE Design & Test* 30.5 (2013), pp. 26–35.

[6] MARTIN Keim et al. "Automated Test Creation For Mixed Signal IP Using IJTAG". In: *white paper Mentor Graphics* (2012).

[7] K Shibin et al. "IEEE P1687 IJTAG demonstrator on FPGA". In: ().

[8] Hans Martin von Staudt and Alexios Spyronasios. "Using IJTAG digital islands in analogue circuits to perform trim and test functions". In: *2015 IEEE 20th International Mixed-Signals Testing Workshop (IMSTW)*. IEEE. 2015, pp. 1–5.

[9] Soliton Technologies. *UART PROTOCOL VALIDATION SERVICE*. 2018. URL: `https://www.solitontech.com/uart-protocol-validation-service/` (visited on 12/15/2018).

[10] Embecosm Limited. *JTAG Chip Architecture*. 2009. URL: `https://www.embecosm.com/appnotes/ean5/html/ch02s01s02.html` (visited on 12/20/2018).

[11] "IEEE Standard Test Access Port and Boundary Scan Architecture". In: *IEEE Std 1149.1-2001* (2001), pp. 1–212. DOI: `10.1109/IEEESTD.2001.92950`.

[12] Digilent Inc. *Nexys 4 DDR Reference Manual*. 2018. URL: `https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual` (visited on 11/15/2018).

[13] Chris Liechti. *`pySerial`'s documentation*. 2018. URL: `https://pythonhosted.org/pyserial/` (visited on 11/01/2018).

[14] European collaborative research project BASTION. *PDL for flat tree network*. 2018. URL: `http://fp7-bastion.eu/files/TreeFlat.pdl` (visited on 11/25/2018).

[15] European collaborative research project BASTION. *About `BASTION`*. 2018. URL: `http://fp7-bastion.eu/index.php?page=1` (visited on 11/25/2018).

[16] Digilent Inc. Nate Eastland. *FPGA – Configurable Logic Block*. 2018. URL: `https://blog.digilentinc.com/fpga-configurable-logic-block/` (visited on 11/20/2018).

Printed by Tryckeriet i E-huset, Lund 2019

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-692
http://www.eit.lth.se