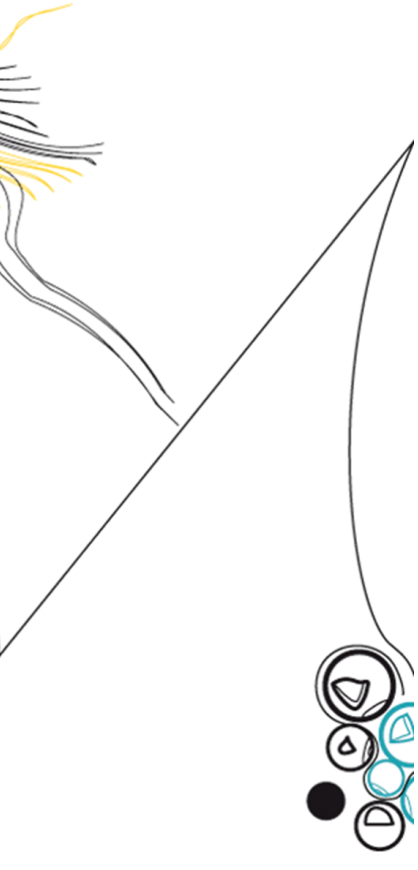
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Analysis and Design of a Dependability Manager for Self-Aware System-on-Chips

Stephen A. Geerlings

M.Sc. Thesis
August 2018



Supervisors:
dr.ir. Hans G. Kerkhoff
dr. Ahmed M.Y. Ibrahim
ir. Jan Scholten

Telecommunication Engineering Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

This thesis describes the analysis, design, implementation and validation of a Dependability Manager Core for Self-Aware System-on-Chips. This hardware is supported by a redesigned software toolchain to allow reuse of IEEE 1687 (IJTAG) networks. The Dependability Manager's purpose is to be integrated into a large scale System-on-Chip. It will increase dependability of the system using software-based dependability applications. The application reuses IJTAG boundary scan test networks during lifetime to increase dependability of system.

Many hardware and software solutions have been developed to improve dependability of System-on-Chips. The implementation of these solutions ranges from high to low level and design reuse is constrained in general. As complexity rises and manufacturing processes decrease over years, the industry must keep pace to guarantee correct functionality of their computer products. Hardware design is a long process where reliability is a trade-off between risk and cost. Long lead times and high prices of wafer manufacturing threatens the design cycle, where multiple iterations improve the final product. A time-proven and universal platform for system reliability will help future hardware designers to minimize design cost and time while providing the same amount of reliability. This is the motivation behind the work in this research; creating a standardized programmable core that has can manage reliability of a large scale system on chip.

This masters thesis constitutes the work for final graduation of the programme Embedded Systems at University of Twente, the Netherlands. This research follows a methodology for the design of an application specific instruction set processor. It has been adapted to suit the purpose of this research. It consists of analysing the application, creating a high level design and implementing and validating the hardware and software separately. After which the two parts are combined and tested. Previous research along with the IJTAG standard provide a background of the application of a Dependability Manager. After the background information is presented, this thesis moves to create a high level design for the hardware and software. These sections are developed; the first as an RISC-V processor with memory mapped de-

vices for retargeting and interrupt management, the latter as a custom toolchain and system drivers. The toolchain features a redesigned PDL compiler, similar to previous work, that creates an IJTAG framework to access the instruments in the network. Reuse of available tools such as the C compiler, Antlr, and VHDL memory IP speeds up the work. A Retargeting Engine for access to test networks and IJTAG interrupt manager are implemented and incorporated into the processor. Drivers along with a memory mapped IO system bus create a hardware abstraction layer that supports portability. The software toolchain and hardware implementation are validated by testing and eventually combined for a full integration test and quantification.

Hardware is emulated on an FPGA to show operation of a compiled program while benchmark networks from the Bastion project are used to show the correct operation of the PDL compiler. The retargeting engine is shown to operate on all Bastion Basic benchmark networks. The interrupt manager is tested with a special implementation of interrupt enabled network structures in a simple tree-network.

Although not perfect, the design of the dependability manager yields new ideas and entices the urge to develop a usable and matured IP core for dependability. The reuse of PDL scripts to generate a high level software framework makes them portable between different architectures and devices. It is a huge step compared to previous research into executing PDL in an embedded context. Practical issues have been found with converting PDL to static C which can be overcome with further research. The retargeting engine and interrupt manager have been validated to operate within the context of this research. The implementation of both hardware devices are tested and have been proven successful. The performance of the retargeting engine is adequate but can be improved with caching. The interrupt manager operates according to its design and decreases the time for interrupt localisation tremendously. The dependability manager processor is simplistic without pipelining or caching. The dependability applications interface well with the generated PDL software frameworks. The hardware abstraction layer makes the interrupt manager and retargeting engine easily usable.

Stephen Geerlings
s.a.geerlings@alumnus.utwente.nl
s0201111

Contents

Glossary

`iApply`	Applies the group of write, read or scan operations that has accummulated since the last `iApply` command.
`iCall`	Calls a PDL procedure defined by `iProc` from another procedure.
`iProc`	Defines a procedure that can be called. A procedure is a collection of commands.
`iRead`	Read a value from a register in the test network during the next `iApply` command. The command also can compare the read value to an expected value.
`iReset`	Enforces that the reset port becomes asserted and a target is set to its ICL defined reset state.
`iWrite`	Write a value to a register in the test network during the next `iApply` command. Arguments are the register or port as defined in the ICL specification and the value that should be written.
AR-Stack	Access Request Stack, a data structure of the retargeting engine that tracks the access requests and adds a new request on the stack if it is necessary to access a register or SCB.
AV	Access Vector, a bit string that is shifted into the IJTAG network.
C	The C programming language.
CSU	Capture-Shift-Update, a process that captures data into a TDR, shifts it out and updates the instruments with data shifted in through the TAP.
DM	Dependability Manager.

ESIB Extension of the SIB to store extra information as bit flags, in this research an interrupt flag which signals that the interrupt originated in the underlying segment.

ESIB_L A leaf node in the ESIB tree, performs similarly but does not open the segment below as those elements are not 'active' due to the Select signal being low.

FPGA Field Programmable Gate Array, a reconfigurable hardware capable of emulating digital integrated circuits.

gcc GNU C Compiler, the open source compiler for C that is ported to an impressive range of targets.

H-Array Hierarchy Array, a data structure that shows the ICL dependencies. It was introduced along with the traverse and generate algorithm for creating access vectors for IJTAG test networks.

HAL Hardware Abstraction Layer.

ICL Instrument Connectivity Language.

IJTAG Internal Joint Test Action Group develops the IEEE 1687 standard which is the successor of the JTAG standard.

IM H-Array Interrupt Management Hierarchy Array, a data structure that shows the network structure to locate interrupt sources.

IMU Interrupt Management Unit, used for managing interrupts coming from the extended IJTAG network.

IP Intellectual Property, a design for a component which can be licensed and reused for implementation.

ISA Instruction Set Architecture, the collection of operations a processor can perform and how they should be encoded.

ISR Interrupt Service Routine, a function or piece of software that is executed when an interrupt is triggered by an external event.

IVT Interrupt Vector Table, a structure that stores Interrupt Vectors. These consist of an Interrupt Service Routine pointer and metadata.

JTAG Joint Test Action Group, the group responsible for the IEEE 1149 JTAG standard that allows test access of chips via boundary scan.

LSU Load Store Unit, responsible for loading and storing data between registers and the memory.

MIPS32 Instruction set based on the MIPS I and MIPS II instruction set and it also added some features of III, IV and V. It was released next to a 64 bit version in 1999. It was also used as a base in the previous research.

MMIO Memory Mapped Input/Output.

PDL Procedural Description Language, scripting language defined by the IJTAG standard.

RE Retargeting Engine, a hardware device used for accessing the IJTAG network.

RISC Reduced Instruction Set Computer.

RISC-V RISC-V is an open instruction set architecture developed in academia and research and aimed at general purpose computing.

RSN Reconfigurable Scan Network, e.g. a IJTAG test network.

RV32I Base subset of the RISC-V instruction set architecture that features 32-bit integer addition and logic, conditional branching and jumping. It can be extended with floating point operations, multiplication and subtraction and other extensions.

ScanMux This network component of IJTAG acts as a switch. It is controlled by a scanmux control bit (SCB) and has at least two serial inputs and one output. It is a multiplexer of the scan chain of the test network.

SCB The ScanMux Control Bit is the register that asserts or deasserts a SIB.

SIB The Segment Insertion Bit is a network component of IJTAG to insert or skip a segment into the network, as defined in standard IEEE-1687, it is also an entry type in the H-Array.

SoC System-on-Chip.

TAP Test Access Port, an interface between the JTAG/IJTAG connection and the network.

Tcl Tool Command Language.

TDR Test Data Register, a component of the test network that stores data.

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.

Chapter 1

Introduction and Problem Statement

Scientific development has pushed the boundaries of chip production technology up to its physical limits. The quest for higher clock speeds, lower power consumption, better production yield and increasingly complex designs created a danger zone where lifetime of chips is shortened and reliability is compromised. Dependability of hardware, especially chips, is continuously threatened during lifetime as component failures occur randomly. This thesis is a step towards new ideas for chip reliability by managing chips during lifetime with software. By employing state of the art dependability procedures running on an embedded processor and reusing novel IJTAG test networks the chip can be monitored and dependability improved. Such a Dependability Manager (DM) combined with a PDL cross compiler has been created for PDL based dependability procedures [1]. This work will extend the design with a functional retargeting engine (RE) [2], support for interrupt servicing [3] and high level language based dependability procedures.

Dependability procedures aim to improve the dependability of a system during lifetime. This system can be anything from a multi-core server processor to a complex mobile System on Chip (SoC). Dependability procedures exist on many layers of a system from hardware to operating system. Their goal is making a system more dependable but the procedures differ greatly in their methodology. Obviously, dependability procedures need interaction with the system to operate. Data about the system comes from sensors and other on-chip instruments and the output of the procedure must be applied to the system. Dependability applications are currently designed in an ad-hoc manner, often for scientific research. Reuse of these applications in consumer or industrial products is problematic. Reuse is a significant money saver for chip designers and is encouraged by industry. Furthermore, dependability procedures implemented in a high level language could be used among different processors.

A new standard for internal instrument access is available since 2014. It improves on the tried and tested JTAG standard which was first named 'Boundary Scan' in the IEEE 1149.1-1990 standard [4]. This boundary scan architecture meant to test the interconnects of a chip but over time has grown to test access in general and it became synonymous with JTAG. The new standard called IJTAG [5] employs a reconfigurable scan chain to access internal instruments via a test access port. Its design goals are reusability and scalability meaning that instruments can be easily added to the network during chip design. A test host can use the network to access registers and instruments on the chip. However, usage of IJTAG is not limited to testing, the network can be used during lifetime to access instruments and safeguard correct functioning. An embedded processor would run a dependability procedure and use a retargeting engine to access this special network.

Not all chips will benefit from a dependability manager as its addition, along with the IJTAG network, incurs an overhead. The aim of the dependability manager is to govern heterogeneous complex SoCs. Such a system would most likely be managed by an operating system. The paradigm of a dependability manager hinges on the division of tasks in layers; functionality and dependability. If the functional layer, along with its operating system, undergoes some kind of fault then the dependability layer will notice and mitigate the problem. This dependability layer may be certified for use in vehicles or aircraft. Meanwhile the functional layer could undergo multiple revisions and can still rely on the same level of dependability. Making the DM feasible for as many chips as possible is a matter of costs. The overhead of a DM must be small. On the other hand, the DM could become a separate packaged chip. The design of scan networks enables the user to chain IJTAG devices on a printed circuit board together and a single DM could manage them all. In theory, the DM can be adopted to act as a separate IJTAG-programmer, i.e. a host-interface, to access or program devices during development. However, this thesis focusses on the first example; the DM as part of a large and complex SoC as in Fig. 1.1.

1.1 Contributions

The Dependability Manager (DM) is an embedded processor. It is designed for its specific application and operation conditions. It functions to access the on-chip instruments via IJTAG, service interrupt requests from the on-chip instruments and improve dependability through software. The embedded processor is an IP that could be added to any IJTAG enabled processor design. Combined with a collection of dependability procedures makes it a platform usable for industry to improve their

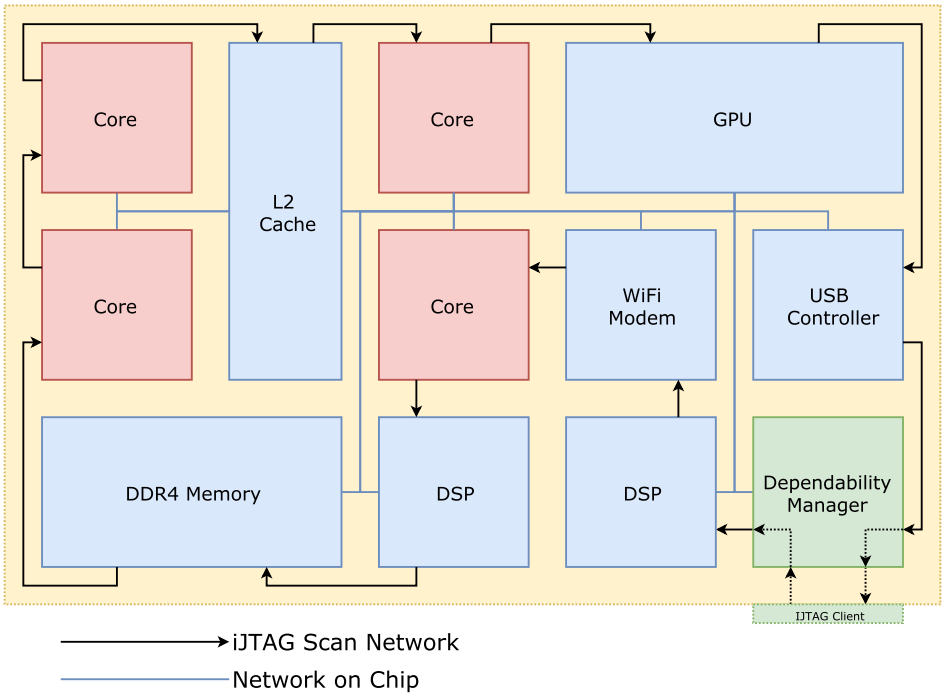


Figure 1.1: Example System on Chip incorporating a DM reusing the IJTAG scan network.

processors with small costs.

The product created in this research is the Dependability Manager IP along with supporting software and drivers. The contributions of this research can be divided into the following.

- Implementation of a RISC-V RV32I instruction set processor.
- Realisation of a PDL to C-framework compiler.
- Implementation of a reusable retargeting engine IP device for IJTAG networks.
- Implementation of an interrupt management unit for the RISC-V processor.
- Implementation of interrupt enabled IJTAG SIB and localisation algorithm in the interrupt management unit.

1.2 Methodology

Development of an embedded system is a complex task. To manage the progress, a design methodology [6] for Application Specific Instruction set Processor (ASIP) is adopted which can be seen in Fig 1.2. Although the structure is beneficial, the

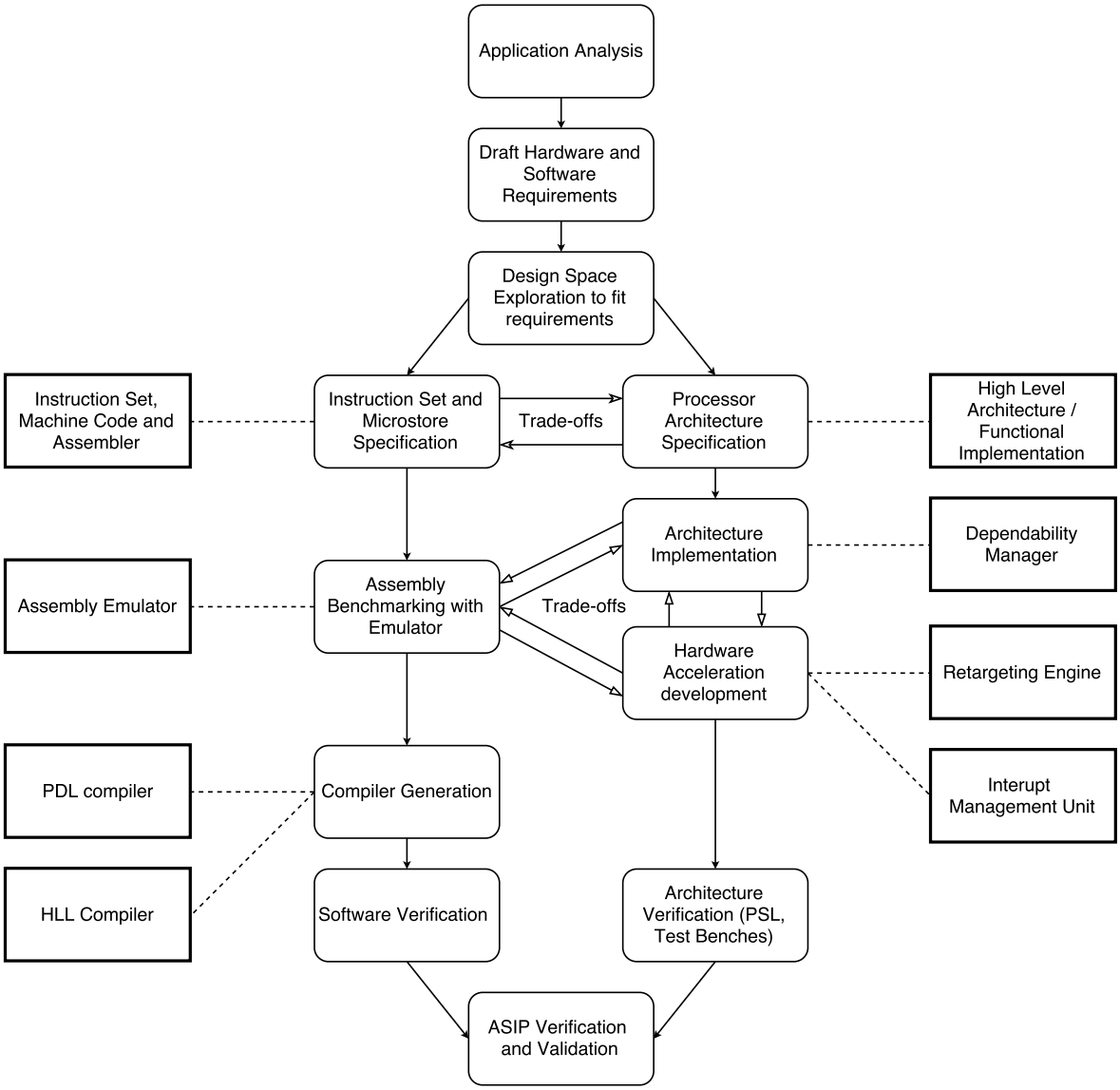


Figure 1.2: Design methodology and the products it yields, adapted for this project [6].

framework is originally exemplified by developing an Digital Signal Processor (DSP). The methodology has also been used in the previous work toward the DM [1]. The intention of this research is to extend a general purpose microprocessor with the retargeting engine and IJTAG IMU. Instead of creating a new application specific instruction set, an RISC instruction set is carefully selected for the application, i.e. the DM. In general, some steps in the methodology are unnecessary and skipped while others are slightly modified for this project's purposes.

The methodology starts with an analysis of the application. Not necessarily the software that will be running on the core but also environment and reason behind using the processor. What is its intended purpose? That will give an idea of what

its abilities should be. This is formulated as a list of requirements. Design Space Exploration (DSE) optimises the processor but takes time as developing and testing a new design is arduous. Maintaining a 'Harvard' architecture for the processor and focussing on the retargeting engine should yield a better design in the time frame.

Selection of the instruction set is entangled with the processor architecture. Trade-offs between them will bring them together. Development of an assembly emulator is not a goal of this project and relates more to ASIP feature testing. If the instruction set provides an assembly emulator it may be used to check the final design or test programs beforehand. High Level Language Compiler generation/development is a feature for ASIP production. The chosen instruction set architecture should provide one for this project to keep everything in the time budget. PDL Compiler is required to generate code that operates the Retargeting Engine. The HLL compiler and PDL compiler need to be verified, this can for example be done with the aforementioned assembly emulator to test the output.

The hardware development starts with designing an architecture around the chosen instruction set. Incorporation of the other system components is also part of the architecture. According to the methodology, a functional implementation can be made in a high level language (HLL) before switching to a hardware description language. This is similar to development of an emulator and as mentioned before not a step toward the goal. It is beneficial for ASIPs to be emulated and checked before further development [6]. For our architecture, it suffices to say that the algorithms behind the Retargeting Engine (RE) and the Interupt Management Unit (IMU) are already verified [2], [3]. The architecture implementation will create a base which will be extended in the hardware acceleration development step. This will yield a RE and IMU for IJTAG. The compiled programs, the base architecture and the extensions will need to operate together; trade-offs will define their relationship.

Test benches are created for the different hardware components to verify their behaviour. Annotation with Property Specification Language (PSL) is possible to verify the design methodologically and formally. The final product is tested by executing self-test programs in a VHDL simulator and an FPGA core. The retargeting engine is applied to benchmark networks. The interrupt management unit is tested and verified with an example network.

1.3 Outline

This thesis provides a literature study to gain in-depth knowledge about the field of study. The state of the art in dependability applications is discussed in Chapter 2. Retargeting and IJTAG is elaborated as further background for the reader. The structure of this thesis follows the methodology. It starts with an early design stage in Chapter 3. The purpose and application of a embedded processor is discussed and dependability applications are analysed to gather knowledge about features needed for their operation. This yields requirements and a discussion is held about architectures, ISAs, programming languages and incorporation of PDL. This creates a preliminary high level design of hardware and software that needs to be realised. The work on the PDL toolchain and supporting software for the DM is described in chapter 4. The work proceeds with the implementations of hardware in Chapter 5. The operation of the DM is quantified in Chapter 6. The whole project gets a conclusion in Chapter 7 where the designs are evaluated and omissions in this research are considered 'future work'.

Chapter 2

Related Work

This chapter attempts to give an overview of the state of the art in dependability management in embedded systems. The scope is limited regarding dependability as the subject is immense and generally found in many fields of engineering. This chapter handles software-based dependability applications and hardware architectures to support dependability applications. A background of boundary scan architecture and test networks is provided as a fall-back for readers unfamiliar to the subject.

2.1 Dependability Applications

Dependability of a system is defined by its measure of reliability, availability, safety, maintainability and integrity [7]. Dependability applications form a collection of software algorithms made to increase these facets. Dependability applications can be incorporated into operating systems or added to an embedded program. A crude example is the screen saver feature in many desktop operating systems. The moving images presented by the screen saver avoided 'burn-in' in old monitors. Current versions of operating systems feature methods to dim or deactivate screens to save energy and prolong battery life.

These small examples are easily understood but the field of increasing dependability is interesting to say the least. Boundaries in engineering are pushed to get the most out of hardware. This hardware is not always created equal since it is a physical process [8]. Currently, in a process called 'binning' the different tiers of chips are separated and they are sold according to their quality. The way chips are made creates a variability between devices. This is the first example of a dependability application that will be presented in this work. The variability of devices can be measured to adapt the operation of software on the device [9]. This increases the yield of a silicon wafer but also increases the confidence that software can run

on less-than-perfect devices.

The second example of dependability applications focusses on the reliability and availability of systems. Dynamic Reliability Management continuously estimates the future reliability of a chip based on its current state [10]. For example, a server is bought and desired to run for at least 10 years. The amount of system degradation can be calculated based on physical properties. Processors suffer under high temperatures, voltages and current and a mean time to failure can be calculated based on these numbers. If the probability that it will fail before 10 years has passed becomes too high the system needs to adapt and decrease temperature for example. However, the system must take into account that it has a certain workload, and that it will need to maintain a certain frequency to perform its job. Dynamic Reliability Management in general steers the processor based on a model of confidence but also needs to keep its workload in mind [11], [12].

2.2 Dependability Management

Dependable systems are ubiquitous. Bridges are 'over-engineered' to withstand the test of time and the design of the internet is primarily redundant to ensure its functionality. Some special electronic systems are also designed to be dependable. Examples of this are computer memory for servers with error correcting code. Other examples are the RAD6000 [13] and LEON [14] processor designs which feature ionizing radiation resistance for use in outer space. Rigorous testing under set circumstances tries to measure dependability of a device and to further improve the design. When it has undergone all possible tests and gets certified it cannot be changed easily. A new hardware revision takes months to produce new chips which need to be recertified before it may be used in cars or planes. Avoiding the uncertainty of new hardware has been a strategy in the Space Shuttle program at NASA, who were still searching for 8086 chips in 2002, two decades after their first release [15].

The term 'Dependability Management' is shorthand for a larger paradigm in embedded systems or rather computer systems in general. Dependability management does not only cover reliability of hardware but also the mitigation of degradation, faults or failures. Hardware can be divided into layers; a functional layer for executing the intended purpose and a *dependability layer* that will enhance reliability and safety and guarantees correct operation of the functional layer [16]. This last layer will be controlled with a *dependability manager*, a simple resilient programmable core that executes a Dependability Application. The dependability layer can be cer-

tified to give certainty to the developers of an application. And applications can be tested to document their performance. A better design for an application could quickly be tested and improved, thus functional revisions to the dependability layer can be made more often.

A dependability manager for a Dependable Reconfigurable Many-Core Processor uses the Network-on-Chip interface to access cores in the chip [17]. Test vectors are generated for the core under test and the response is evaluated. Both JTAG and IJTAG can be used for communication through the dependability layer as it provides a standardized access method for instruments. These test networks are also present on many devices to find interconnect faults or to program the device. Reusing them during lifetime is a cost-effective method to add a dependability layer to an existing device [18].

An automated test coprocessor for JTAG was created for a MicroBlaze 32-bit processor [19]. The coprocessor has a simple microprogrammed control path to execute tests on the larger system through the boundary scan interface. The JTAG operations on the network where analysed and a minimal command set was created to support the test. Accessing JTAG from an embedded context has been done to execute tests during lifetime. For this a JTAG controller has been built that executes based on a FIFO queue [20]. This device could be reused to connect a DM to a JTAG network.

The management of hardware within hardware has also taken a step with the introduction of Intel's Active Management Technology (AMT) which allows users to manage systems remotely [21]. Computer systems can be activated or deactivated with this daemon process within the hardware. The degradation and performance of systems was also presented to the user to let him repair components where needed. Among Intel's line of products, the Intel Management Engine became well known due to its far reaching access to the hardware of consumers [**?**]. The separate embedded processor featured in many chips would be active even when a computer was shut down [**?**]. Connecting to a central platform would be a great feature for any Dependability Manager. It would allow companies to gain insight into the degradation of devices, similarly to Intel AMT. Considering the rise of Internet-of-Things (IoT) devices, this could become a reality.

This research is a continuation towards the concept of online IJTAG-based dependability management [1]. The previous project created a compiler for PDL and a MIPS32-based DM. The compiler is based on the Antlr4 framework for processing programming languages [22]. The IJTAG standard specifies the syntax of ICL and

PDL in a format, i.e. a grammar specification, supported by the Antlr4 parser generator. The compiler compiles a modified version of PDL along with Tcl to MIPS32 assembly. The DM featured an empty shell for the retargeting engine and did not support the full MIPS32 instruction set. The future work suggested by the work of Zakiy includes: 'test for different processor, adding remaining PDL commands, cross compilation to other machines, and use of C libraries'. This research will take advantage of the lessons learned.

2.3 The IJTAG Standard

IEEE 1687-2014 Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device [5], colloquially referred to as IJTAG is a new standard developed by the Internal Joint Test Action Group. It provides methodology to access instruments within devices by reusing the Test Access Port (TAP), TAP Controller and Boundary Scan Architecture defined in the earlier JTAG standard [4]. It introduces a new version of the Procedure Description Language (PDL) along with the Instrument Connectivity Language (ICL). These technologies will be explained in this section to provide a background for the reader. Originally, the application of boundary scan architecture was to detect errors during manufacturing. For example, the scan cells can be used to probe signals to detect stuck-at faults on a printed circuit board. Since then, the use of JTAG has grown beyond testing. Among its uses are programming devices, accessing internal registers and controlling the chip during debugging. The role of JTAG has also grown in software design tools which can automatically load and execute a program on a device via JTAG. The rise in functionality, along with the rising hardware complexity prompted the development of the new standard. Its goals were, amongst others, to tackle the size of test networks and to support instruments with more functionality. The first was accomplished by adding network structures that add hierarchy and compartmentalises the network along with a description language that advocates reuse of network modules. The second goal is handled with PDL, a programming language targeted at electronic design automation and testing via the test network.

The philosophy behind the IJTAG standard is more descriptive rather than prescriptive to allow different architectures to conform [5]. It's scope is limited to discuss the access to instruments without being specific about the instruments themselves. It contains abstract notions of the test network which need to be realised by users of the standard. Therefore, some gaps will need to be filled to operate IJTAG as intended. An example of this can be found in IJTAG's description of retargeting, without giving a hint of how one should go about it.

Listing 2.1: SIB module defined in ICL [23].

```
Module SIB_mux_pre {
    ScanInPort SI;
    CaptureEnPort CE;
    ShiftEnPort SE;
    UpdateEnPort UE;
    SelectPort SEL;
    ResetPort RST;
    TCKPort TCK;
    ScanOutPort SO {
        Source SR;
    }
    ScanInterface client {
        Port SI; Port CE; Port SE; Port UE;
        Port SEL; Port RST; Port TCK; Port SO;
    }

    ScanInPort fromSO;
    ToCaptureEnPort toCE;
    ToShiftEnPort toSE;
    ToUpdateEnPort toUE;
    ToSelectPort toSEL;
    ToResetPort toRST;
    ToTCKPort toTCK;
    ScanOutPort toSI {
        Source SI;
    }
    ScanInterface host {
        Port fromSO; Port toCE; Port toSE; Port toUE;
        Port toSEL; Port toRST; Port toTCK; Port toSI;
    }

    ScanRegister SR {
        ScanInSource SIBmux; CaptureSource SR; ResetValue 1'b0;
    }
    ScanMux SIBmux SelectedBy SR {
        1'b0  SI;
        1'b1  fromSO;
    }
}
```

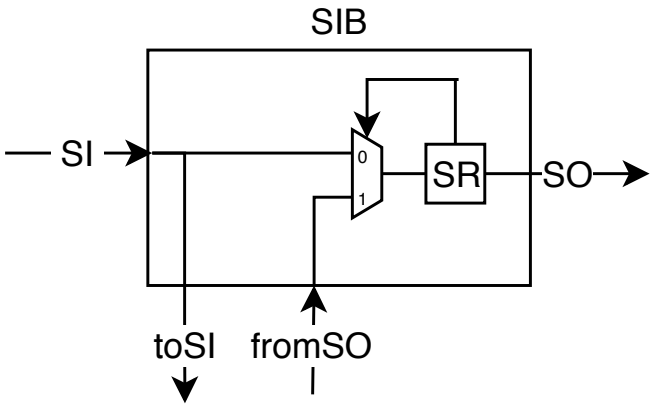


Figure 2.1: Abstract representation of a SIB, not all Scan Interface signals are shown.

Table 2.1: Signals of a Scan Interface according to the IJTAG standard [5].

Signal Name	Functionality
TCK (Clk)	Clock signal for the test network. Elements operaate on the rising edge, except for the update operation which is falling edge.
Reset	Reset signal for the test network, active high.
CaptureEn	Signals that data must be captured from an instrument into the TDR.
ShiftEn	Signals that data must be shifted through the network.
UpdateEn	Signals that all active TDRs must write their value to the instrument.
ScanIn (SI)	Serial data into the test network.
ScanOut (SO)	Serial data from the test network.
Select (Sel)	Select signal that activates a Scan Interface and subsequent interfaces.

2.3.1 Reconfigurable Network Structures

There are different structures within the test network that are defined in the standard [5]. In general a Reconfigurable Scan Network (RSN) starts at the 'client interface' which connects to a 'host interface'. This network interface, referred to as the Test Access Port (TAP) consists of signals which can be seen in Table 2.1. The Instrument Connectivity Language (ICL) specification of a network or an instrument contains modules and instances. A module can be seen as a box with signals going in and out. Behaviour of the box and predetermined signals are specified with ICL keywords. The module specification for the SIB in Fig. 2.1 is shown in Listing 2.1. A module can be instantiated within the network specification.

The signals into the network are driven by the host interface. This done by the 'Test Access Port'-controller for the top-level Scan Interface. The signals are propagated through the modules, this can be seen in Listing2.1. A SIB has input and output signals, which are connected to Scan Interfaces 'below' the SIB. The DM's TAP-controller will be connected to the top-level Scan Interface in this research. TAP-controllers operate as state machines, normally controlled by the 'Test Mode

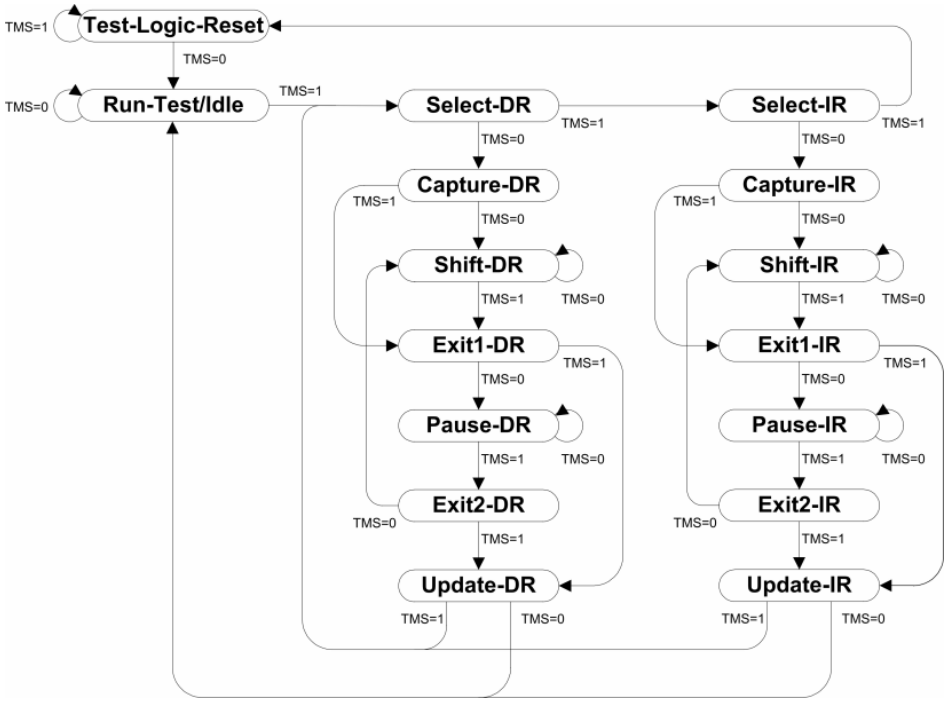


Figure 2.2: Test Access Port controller state machine, figure taken from the IJTAG standard [5].

Select' signal. This state machine is shown in Fig 2.2. Its behaviour will be emulated by the DM's hardware. The state machine executes cycles of Capture-Shift-Update (CSU-cycle) on the network to read and write data to the network. Most important to remember, is the sequence from the *Select-DR* state. After parts of the network are activated by selecting them, the subsequent *Capture-DR* state will read data into the internal Test Data Register (TDR) of an instrument. The length and representation of this data is dependant on the instrument and must be managed by the retargeting tool. After capturing, the data will be shifted through the scan chain and new data will be shifted in to the network at the same time. Of course, this happens during the *Shift-DR* stage. This new data is then written to an instrument during the Update cycle. There is always a Capture and Update stage, but the duration of the Shift cycle differs based on the length of active scan path. This length defines the size of the Access Vector (AV); the string of bits written to and read from the network.

Among the list of standard-defined network structures there are Test Data Registers (TDR), Segment Insertion Bit (SIB), ScanMux Control Bit (SCB) and Scan Multiplexers (ScanMuxes). TDRs are included or excluded in the network based on the configuration of the network. This configuration happens in the SIBs and in the ScanMuxes of the network. SIBs feature their own SCB before or after the segment that they control. A ScanMux relies on an external signal which is often connected

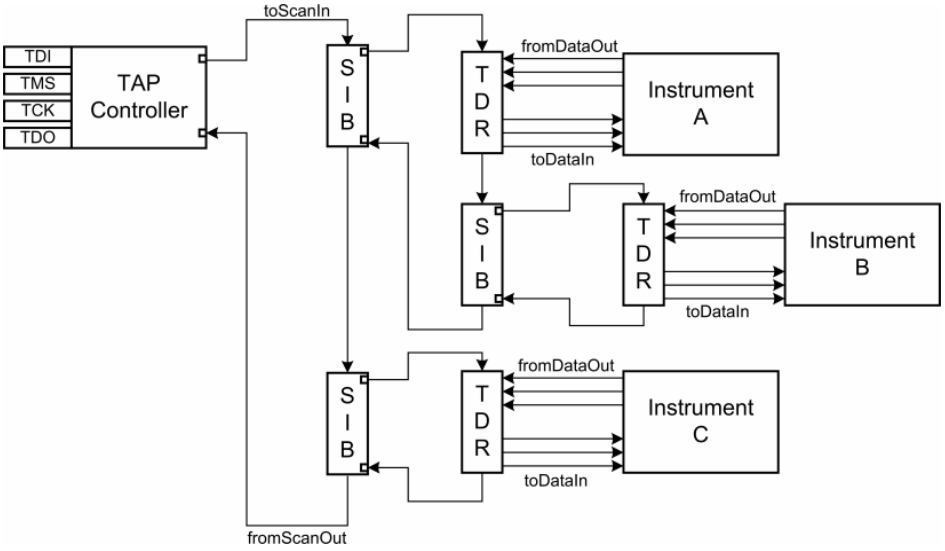


Figure 2.3: Hierarchical IJTAG Scan Network using SIBs, taken from the IJTAG standard [5].

to a SCB somewhere in the network. These structures offer the designer of a test network more versatility, optimise the design for fast access and provide hierarchy where needed. If no SIBs where placed in the hierarchical network of Fig 2.3 the shortest access vector would always be $3 * TDR.length$. But with these structures the access vector size is $TDR.length + 2$ for the TDR connected to Instrument C.

The data in access vector needs to have a certain order to wind up at the right place in the network. The data for the last element of the chain needs to be shifted in first. A nibble-sized example TDR is shown in Fig 2.4. The figure also contains a timing diagram of a CSU cycle along with the data in values (S0, S1..) and the data out values (D0, D1..).

ICL features more than is discussed here but this will suffice as a base for the remainder of this thesis. The design of ICL allows for many network configurations to be created. The remainder of this research will rely on the Hierarchy Array (H-Array) as a network specification. This data structure will be explained shortly.

2.3.2 Procedural Description Language

PDL is a language to be used in combination with IJTAG. The standard defines two levels of PDL which are syntactically compatible with the Tcl language. Since most Electronic Design Automation tools already support Tcl, it would reduce the effort needed to adopt the new language. The purpose of PDL is to provide a standardized language for manufacturers of IJTAG instruments. A smart sensor may need to

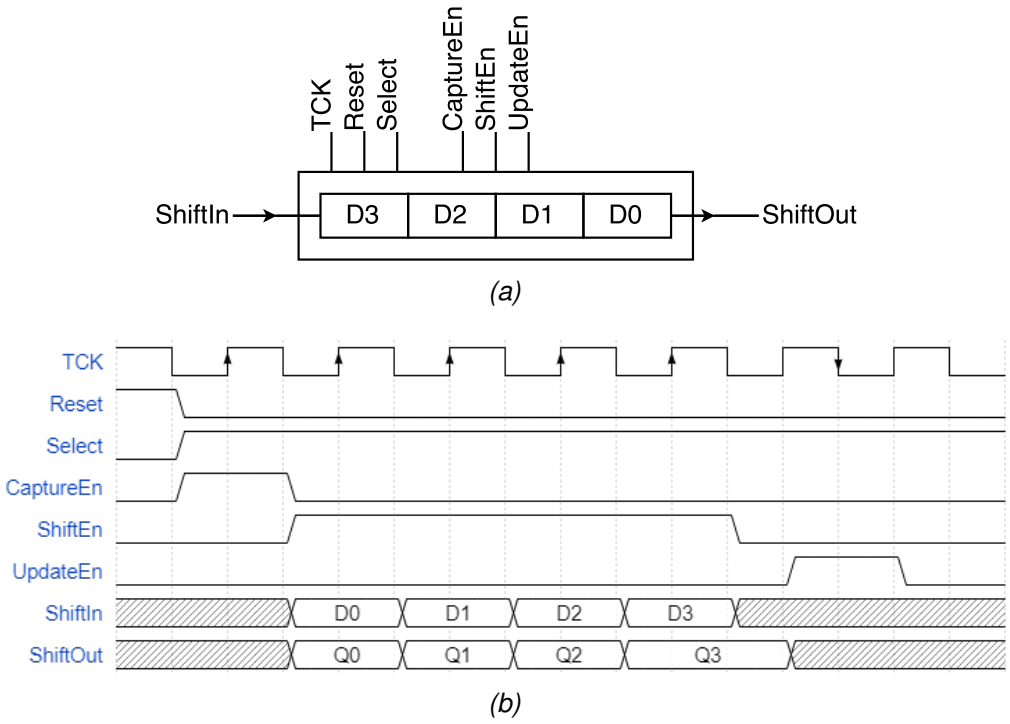


Figure 2.4: Timing diagram of a CSU cycle on a TDR Scan Interface.

be initialised to operate. It could also feature a self-test. Writing the code for those operations is error prone and tiring. An instrument designer can use PDL to write driver software to operate the device. The user of the device can then reuse the instrument and the code to quickly get up and running.

There are two levels of this language, namely PDL level-0 and PDL level-1. PDL level-0 is static in nature while PDL level-1 is dynamic. PDL level-1 incorporates Tcl which adds control flow to the language. Actually, when a file is deemed PDL level-1 the tool may expect Tcl statements along the code. This incorporates high level language components such as conditional branching in PDL. The base language is shown in Table 2.2 while the extra features are shown in Table 2.3. The operation of each command is discussed in Chapter 4.

PDL level-0 operates on the network with the 'scan commands', the set of commands that issue reads and writes on the network. `iWrite` is responsible for writing to a register and `iRead` retrieves a value from a register. All scan commands are queued in an `iApply` group until they are applied to the network. The subset of the language is also responsible for procedures (`iProc` and `iCall`) which house these operations.

PDL level-1 forms a bridge between PDL and Tcl by providing access to read

Listing 2.2: ICL along with PDL level-0 and PDL level-1 commands. Taken from the IJTAG standard [5].

```
ICL:
    ScanRegister RegA [5:0] { ... } ;
    ScanRegister RegB [5:0] { ... } ;
    Alias lbits = RegA[1:0], RegB[1:0] ;

PDL:
    iRead RegA ;
    iRead RegB(3:0) 0xf ;
    iApply ;

Tcl:
    iGetReadData RegA -hex ;     # returns a string representing
                                 # a hexadecimal number

    iGetReadData RegB(3:1) ;     # returns a string of 3 bits in binary

    iGetReadData lbits -bin ;    # returns a string of 4 bits in binary,
                                 # none of which are 'x', since all bits
                                 # this alias refers to were iRead
```

data from a device and to support a conditional flow through a test. An example of this can be found in Listing 2.2. The read data from a device can be used in Tcl after retrieving it. The test may go a different way based on the returned data from the network. Since this work will incorporate PDL in another high level programming language, the Tcl interface and support seems superfluous. More on this can be found in Chapters 3 and 4.

2.3.3 Structured Pattern Generation and Retargeting

The IJTAG networks are inherently master-slave and they need an entity to configure and control them. Specifically, IJTAG networks require a process called 'retargeting' that reconfigures the active scan path to access an instrument [5]. This process can be compared with the operation of a railway-network; The railway's switches need to be set appropriately to get a train from point a to point b. This sounds simple, but that is not all. In this case the order in which the switches are set matters; only the train driver can set the switches; and he can only do it when he passed the switch. At this point the analogy gets a bit vague. This subsection will try to explain the problems surrounding retargeting and discuss different approaches to solve them.

Resolving the necessary values for SCBs and SIBs, given a target instrument, is an active research subject since the arrival of the IJTAG standard. It is clear to see which SIBs should be opened to access a register in a hierarchical configu-

Table 2.2: The set of PDL level-0 commands.

Command	Example
iPDLLevel	iPDLLevel 0 version STD_1687_2014;
iPrefix	iPrefix core2.cpu;
iRead	iRead temp_register 0b1234;
iWrite	iWrite temp_register 0b1234;
iApply	iApply;
iReset	iReset;
iScan	iScan foo 4 -si 0b0101 -so 0b1x0x;
iOverrideScanInterface	iOverrideScanInterface core2.cpu -capture off; iOverrideScanInterface core2.cpu -update off;
iClock	iClock foo;
iClockOverride	iClockOverride foo -freqMultiplier 5;
iRunLoop	iRunLoop 1000 sck foo;
iProcsForModule	iProcsForModule foo_namespace::core2_module;

iProc	iProc foo { arg1 {arg2 1234}} {
	... ;
	}

iUseProcNameSpace	iUseProcNameSpace foo_namespace;
iCall	iCall foo 12 34;
iMerge	iMerge -begin; iCall proc1; iCall proc2; iMerge -end;
iNote	iNote -status "Important Message"; iNote -comment "Hello World";
iTake	iTake temp_register;
iRelease	iRelease temp_register;
iState	iState temp_register 0b1101110011; iState SIB1 0b1;

Table 2.3: The set of PDL level-1 commands.

Command	Examples
iGetReadData	iGetReadData temp_register hex ;
iGetMiscompares	iGetMiscompares temp_register hex;
iGetStatus	iGetStatus clear; iCall myProc; set num_fails [iGetStatus]
iSetFail	iSetFail "Error: unexpected result" -quit

ration, e.g. Fig. 2.3. For larger and more complex networks the problem forms a large Boolean equation. However, the order in which the SIBs and SCBs are set may matter in solving the equation. SAT solvers have been applied to perform retargeting [24]. When it is clear which SCBs need to be set, the next step will be to determine how to get them in the active scan path. Solving this problem efficiently is crucial for automatic access in increasingly larger scan networks [2], [5], [16], [24], [25].

The approach using a SAT solver is applied in the research of Baranowski et al. [24]. The work focusses on reduction of test time and robustness of a tool to formally verify a network's layout. The automatic pattern generation has been applied to benchmark networks and different experiments were held, for example to reduce access time or to merge multiple concurrent access requests. The research suggests that optimal pattern generation also implies taking care of closing the segments that are not needed any more.

The algorithm by Ibrahim for retargeting is based on an Access Request stack and a Hierarchy Array [2]. The H-Array shows the dependency relation between SIBs or ScanMuxes and their underlying instruments. This is done through the creation of a Selection Dependency Graph based on the ICL specification. The graph representation is then translated to the H-Array [2], [26]. To start retargeting the network, the access request stack is filled with one or multiple entries for instruments. Basically, an access request is created for every `iWrite` and `iRead` in the `iApply` group. If an SCB needs to be changed for access to an instrument, it is pushed on the the stack of that instrument. If a SCB is on the active scan path, its new value will be put in the access vector. This opens the necessary segments of the IJTAG network and the instruments can be accessed. The work does not present a method to close a segment of the network. This project will focus on the work of Ibrahim as its "execution model" is applicable in the embedded setting of the DM [2], [16], [25].

2.3.4 IJTAG Tools

Industry is currently working on adopting IJTAG in their products but many still rely on regular JTAG. Fortunately, some development tools such as JTAG debuggers are usable with IJTAG as they both use the same TAP control logic. At the moment of writing, no chips can be found at suppliers like RS and Farnell boasting the new IJTAG standard. This seems a *Catch-22* where the development of tools and software relies on people using IJTAG processors, while the usage of new technology relies on the support of tools. Among the available resources for IJTAG are software

tools and benchmark networks (in PDL, ICL and VHDL) made available by the Bastion project [23], [27]. These benchmarks contain ICL, VHDL, PDL and figures of different IJTAG networks. Some of the networks are based on the popular ITC'02 benchmark collection [28]. One of the networks, the Mingle network, is a versatile example that features all kinds of structures. This network has been used to test the automatic generation of the H-Array from the Selection Dependency Graph [26]. The benchmark networks can be used to test all the operation of the Retargeting Engine and PDL compiler.

Other tools that are available are aimed at the design of IJTAG compliant IP. Automatic generation of ICL and PDL based on hardware and automated testing of developed IP are among the suites provided by different companies [29]–[31].

As mentioned, the previous work by Zakiy [1] offers a PDL compiler to MIPS32 assembly code. This compiler could be refurbished to fit the needs of this project. However, the supported syntax of the compiler is an adapted version of PDL and Tcl. Due to the nature of compiler generation with Antlr, it may be more feasible to start from scratch.

2.4 Discussion

The surface of dependability management and dependability applications is barely scratched by this brief deliberation of the subject but it gives an insight into what should be expected. The information gained can be used in the next chapters to design the Dependability Manager. A background of IJTAG and boundary scan architecture is given to the reader which should suffice for understanding the remainder of this report. Of course, all information concerning the subject can be found in the IEEE standard [5].

Chapter 3

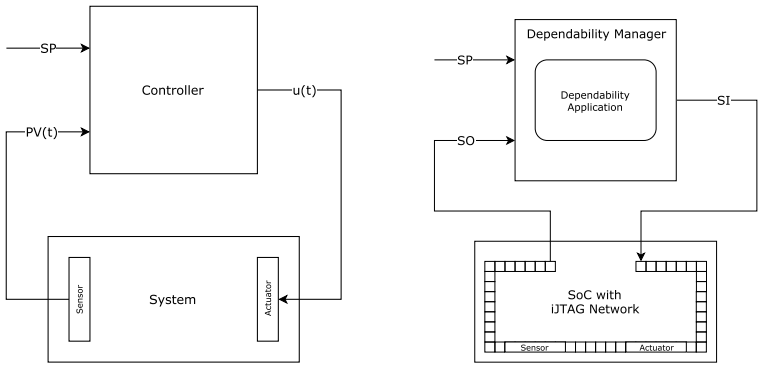
Early Hardware/Software Codesign

This chapter will move through the first steps of the design methodology presented in the work of Jain et al. [32], see Fig 1.2. The methodology begins with a design phase analysing the problem, then combining hardware and software into a single solution. After that it moves to separate the tracks for implementation. This chapter explores the design space and creates a complete picture of the DM before moving further with implementation.

3.1 Function of the Dependability Manager

The adopted methodology starts with an application analysis to gather information about the domain of Dependability Management. This domain is characterized by the dependability applications and the operation environment. There are different dependability applications and some generalisations will be made to know what the DM shall execute. The operation environment provides a setting in which the DM will exist and hardware needs to be designed accordingly.

In Section 2.1 some examples of dependability applications are given. These algorithms to manage dependability all need a form of input from sensors. To actively manage a system they also need an actuator. This is the definition of a control system as shown in Fig. 3.1. A control system consists of a controller, sensors, and actuators, see Fig 3.1. The system calculates the error between the setpoint and the measured value (PV). This error value is then used to steer the actuators. Many control algorithms exist for different purposes. Control systems are used in many fields of engineering; one of the first examples is the use of a so-called *Governor* on steam engines. It maintains a steady speed by managing the amount of steam let into the engine [33], [34]. With the rise of digital systems came the control algorithms. These are used to measure and actuate within milliseconds and they are the



(a) Control System *(b)* Using IJTAG

Figure 3.1: Similarity of a Control System and Dependability Manager using IJTAG.

driving force behind drones, rockets and industrial plants.

The dependability manager will also operate as a control system using sensors and actuators in the IJTAG network. This is also shown in Fig 3.1. The setpoint is stored in the DM or calculated as part of the dependability application. The input and output of sensor and steering values is done by shifting values through the IJTAG scan chain. The actuation rate of most dependability algorithms is low [10], as seconds pass between updates. This makes the use of IJTAG as a means of control system possible, at least for dependability applications. The clock speed for JTAG and IJTAG is affected by multiple factors. Normally, JTAG programmers are connected via a cable to the printed circuit board. The layout of the test access network may cause a clock speed limitation due to signal propagation delay. Typical clock speeds for JTAG programmers is 1 to 50 MHz, the higher the speed the more expensive the unit. To guarantee operation of control algorithms on the IJTAG network, the following assumptions about it are made.

- Proper scan path optimisation to the required sensors and actuators.
- High throughput communication with the retargeting engine.
- A high availability of the IJTAG network.
- An average to high clock speed for the IJTAG network.

The usage of dependability applications is also bound by the information stored in the network and the IJTAG-accessible actuators. A dependability application cannot do anything without an actuator in the network. The IJTAG standard describes a number of instruments which might be implemented into a core by chip manufacturers. Among that list are: PLL circuits, clock gate circuits, BIST engines, and

instruments to support the process of binning processors. The research into embedded instruments for IJTAG is ongoing and different instruments are adopted to be used within the network. Temperature, current and potential difference (voltage) sensors have been wrapped by an interface layer for IJTAG [1], [35]–[37]. The same approach for sensors can be taken for actuators. Thus the usage of dependability applications is limited by the *need* of these actuators in the test network.

Dependability is a measure of a system's availability, reliability, maintainability, integrity and security. A dependability application is defined as a software solution to improve dependability of a system. A dependable system implies maintainability of the system and by definition a dependability application needs to be maintainable to ensure its purpose. To create some scope of the domain in which the DM will operate different forms of dependability applications are discussed.

Variability Management Creating a processor in silicon is a physical process and not all chips are made the same. Multiple units are made on one wafer and subsequently cut into separate pieces, called a die. Each unit is then tested to finalize the manufacturing process. This test will check if all cores are functional or what the maximum sustained clock speed is. The chips are then sorted into separate bins [8], [38]. These bins give the practice its name; *binning* increases the yield from a waver as partially working cores can still be used.

The variability in the produced chips is a key issue in modern many-core processors. Binning is one of the options to mitigate the differences among processors and improve yield. The processors are tested after manufacturing and are clocked and configured accordingly. The variability among processors is not only noticeable after manufacturing but also during lifetime as the degradation of chips is also a physical process [9], [39]. The battery of a mobile phone or notebook computer will not keep the same capacity over time and the amount of power it can deliver at a given time will decrease. This has led to Apple under-clocking certain smart phones with a software update as the battery could not supply the phone with power [40].

The degradation process also holds for processors which suffer performance loss of their components over time. Eventually the component will not perform any longer and this must be mitigated to avoid unwelcome surprises, i.e. *graceful degradation*. The addition of embedded instruments to the processor enables Dynamic Variability Management [9]. Current and voltage sensors can be used to measure degradation of a signal path. The configuration of a system can be altered based on measured data by the dependability application. The application could perform the same measurements that the chip receives after production. The chip can also

be tested by executing benchmarks to check its operation [8]. The dependability application then proceeds to (de)activate components that have degraded or it decreases clock speeds to allow correct operation of affected areas. This could hurt performance of the system in general and some form of communication with the functional layer could be beneficial [16].

Reliability & Availability Management Where the subject of the previous paragraph simply accepted the reality of failing components, the field of Dynamic Reliability Management (DRM) tries to take it a step further. The physical process of degradation has been mapped and analysed to provide models of the environmental effects on degradation. These models are used to estimate the lifetime of a component, i.e. the mean time to failure. The goal of Dynamic Reliability Management is to give certainty that the component does not fail before reaching its lifetime goal while executing a certain workload.

The DRM algorithm proposes to manage a SoC's different failure modes by employing reliability models [10]. Temperature and voltage are the main causes of oxide breakdown, a reaction where the gate oxide of a MOSFET fails. This, among other failure modes of the semiconductors, are taken into consideration in the DRM algorithm. The algorithm calculates total reliability by analysing the different blocks on the chip. The workload of the device is also monitored and a gaussian distribution is fitted to it. A confidence function is used to create an estimator for the remainder of the work that needs to be executed in the chips lifetime. Knowing the processor's utilisation and its intended lifetime lets the algorithm calculate the likelihood of chip survival in those environmental conditions. The algorithm then steers the clock frequency and voltages applied to the device to lower temperature and voltage. Although this may reduce performance, the survival of the chip is guaranteed with a higher certainty.

The algorithm's calculations are extensive as the failure models, probability functions, estimators feature summations over all the blocks and double integrals for block reliability models. Although the research mostly focuses on the Oxide Breakdown Model, Electron Migration and Thermal Cycling are also included in calculating the systems reliability. The research employs look-up-tables which store the values of degradation models for certain parameters. The OBM model is based on voltage and temperature which determine those parameters. These tables can be calculated beforehand. The remaining calculations for the workload estimation happen on chip. Lastly, the dynamic frequency and voltage scaling module used to actuate employs a proportional-integral-derivative (PID) controller.

The algorithm is optimised with the look-up-table approach but features extensive arithmetic to make proper decisions. The algorithm was validated with a simulated approach but also was implemented and tested for 24 hours. The update interval of the algorithm was 10 minutes as this should be enough on the long term. The calculations for expected workload etc. should easily be performed within that time.

Fault Management Faults can occur at any time in a device. They are technically defined as the cause of errors. If an error reaches the service interface and alters the service it will result in a failure. Managing faults will limit the potential ensuing damage of a failure. For example, a fault in the clock signal of a data bus may lead to erroneously read values. These values may change the system behaviour and this becomes a failure. An *intermittent resistive fault* instrument used to detect timing issues can notify the DM that a fault occurred [35]. This notification can also happen with interrupt enabled IJTAG [3] which triggers a service routine to manage this occurrence. The interrupt enabled IJTAG architecture allows automatic scan path reorganisation to localize the interrupt. The test network configuration is not affected after the interrupt. There are multiple strategies in fault handling for dependability [7] based on the faults origin. Having a reactive system such as the interrupt manager will aid the end user to manage notifications of the instrument without continuous polling.

This concludes the application analysis and the research can move forward with the information gained. Dependability applications improve reliability and availabity amongst other features. The environment of the DM includes a instrument network that provides information to a control algorithm. The network also contains self test instruments and actuatrs. The algorithms range from simple to highly complex although their execution is intermittent and they focus on the long term. In principle the dependability layer will need to outlive the functional layer and it cannot continuously execute. The speed and configuration of the IJTAG network also limits the sample frequency of sensors in the network. Faults can occur at any time so an interrupt driven approach for handling them makes sense. This leaves a simple design for a microprocessor with interrupt management which should be able to perform most, if not all, dependability applications.

3.2 Architectural Design Exploration

The general concept of a dependability application is sketched in the previous section of this chapter. This section will continue with this input to explore the architectural design space [32]. Questions about the high level language compiler, instruction set, and other architectural features needs to be answered. At the end of the section, the interaction between software and hardware should be clear. This will allow for independent implementation of the two according to the methodology.

The proposed DM is always part of a larger system, it has no use on its own. To increase the cost-benefit ratio of adding a DM to a SoC the cost of the DM needs to be kept low. This means a small sized processor for executing dependability applications. This is a starting point for the design exploration. A small processor means a relatively simple instruction set, and less features will also ease implementation. The choice of instruction set affects the choice for a high level language as not all instruction sets are supported by all high level language compilers. Generally speaking, gcc/g++ is most suitable as their target support is unrivalled for now. Although our choice is limited multiple languages are evaluated for research purposes. The selected language will need to interface with the Retargeting Engine and execute PDL programs. PDL was translated to with custom coprocessor instructions in the previous research [1]. This was done by a custom compiler. The design of the processor and the retargeting engine needs to be considered when incorporating PDL. The dependencies between al facets of the design space is showcased in Fig. 3.2. The processor design and instruction set influence each other, e.g. the amount of registers available to the ISA will need to be realised in the processor. The ISA and high level language are coupled as a compiler will need to be available. The high level language and PDL are coupled since the PDL will need to be executed from the context of a dependability application. The high level language and processor design do not affect each other due to the abstraction provided by the instruction set and compiler. PDL incorporation and instruction set were coupled in previous research as custom instructions were used to execute scan operations. This limited the ability of the PDL compiler to target other machines. This research will try to decouple the execution of scan operations from the instruction set to increase portability of dependability applications. This will simplify reuse of the PDL compiler as well.

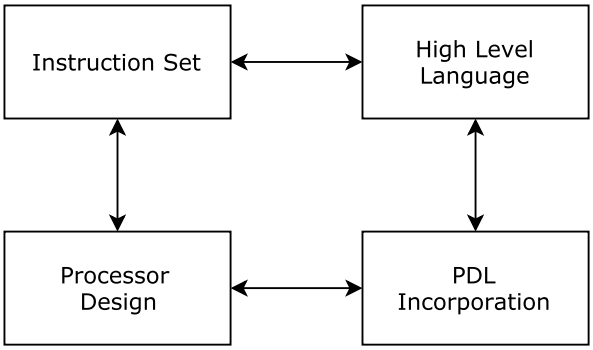


Figure 3.2: Design choices for the DM affect each other.

3.2.1 Instruction Set Architecture

A massive amount of Instruction Set Architectures (ISA) have been designed and implemented by processor manufacturers around the world. They are often reused or extended over the years, and some are backward-compatible. Instruction sets are classified, e.g. RISC, CISC, VLIW, based on features, size, functionality and purpose. A suitable instruction set is needed for the DM that fits its particular functionality, size and purpose. The instruction set is also required to feature an assembler and compiler for a high level language. The decisions for high level language and instruction set rely heavily on each other as can be seen in Fig. 3.2.

The work of Zakiy argues the use of MIPS (Microprocessor without Interlocked Pipeline Stages) as instruction set in its dependability manager [1]. The MIPS series architecture is chosen because the original MIPS I/R2000 ISA was published as an openly available architecture by its creator John L. Hennesy. MIPS I is a RISC ISA and the design features support for moving data to and from coprocessors. MIPS Technologies has made multiple iterations of the MIPS architecture. The MIPS I has been incorporated in the MIPS32 standard [41]. This ISA contains the full MIPS32 ISA which consists of multiple extensions, e.g. MIPS I, MIPS II, and this standard remains under intellectual property protection.

The coprocessor-specific instructions, which are employed by the work of Zakiy [1] to operate the retargeting engine coprocessor, are introduced by the MIPS32 standard. Mixing these instruction sets (a subset of MIPS I and the coprocessor specific from MIPS32) is a bad practice as high level language support becomes compromised. Adding arbitrary instructions from one instruction set into the other creates a non-standard instruction set. Compiler support for such an instruction set is non-existent. However, since the project had a custom compiler, this was not an issue [1].

These custom instructions also raise the question on how these they can be executed from the context of a high level language. The PDL-compiler automatically converted `iRead`, `iWrite`, and `iApply` to coprocessor instructions [1]. This is not an option as this new version will reuse an existing high level language compiler, such as gcc. The coprocessor-specific instructions should be usable from a high level language with inline assembly programming as shown in Fig. 3.3. This feature allows a programmer to incorporate assembly for increased performance and direct control of hardware. Some instruction sets like MIPS32 and RISC-V are extendible for application specific purposes, and a RISC computer can be extended to become an ASIP. This approach has the downside that the retargeting engine IP core will become less portable, and thus usable in other projects as the design is tied to the custom instruction set.

The approach of Zakiy's design [1], i.e. the coprocessor-specific instructions for the retargeting engine, affects the use of high level language. Not all high level languages allow for inline assembly programming or linking with custom machine code. A different approach than the ASIP solution is found in the Memory-Mapped I/O (MMIO) architecture and dedicated drivers for the retargeting engine and other DM subsystems. In general, drivers connect high level software with low level hardware. The MMIO architecture is further explained in subsection 3.3.

This leaves the choice for an ISA wide open. The results of the dependability application analysis in subsection 3.1 reveal that a RISC instruction set should suffice. High level language support is required for the ISA along with interrupt service routine features. Most dependability applications are some form of control algorithm so arithmetic must be supported. The high level language should also offer some mathematical libraries for computations. Relevant candidate ISAs are reviewed to find a suitable option. All discussed instruction sets are supported by gcc, so the project can fall back on C/C++ as a high level language. The requirements for the instruction set for the DM are drafted accordingly:

- The ISA must have high level language support, specifically a compiler for a preferred programming language.
- The ISA must have arithmetic and control logic that supports the execution of a compiled high level language program.
- The ISA must have memory operation support to enable the MMIO operation of DM subsystems.

- The ISA must have interrupt support to enable the execution of interrupt service routines.
- The ISA should be considered RISC as this group features small and simple hardware designs.
- The ISA should be available for use in this academical project without license fees.

The amount of instructions in the ISA directly corresponds to the effort needed to implement the processor. More instructions mean more work as design, implementation and validation of the processor is needed. However, if an architecture offers extendible implementations for this project it may be worthwhile to consider integrating it in this project.

The exploratory search into RISC architectures yielded the following candidates which are briefly discussed and compared in Table 3.1. All the ISAs in the table feature high level language support, arithmetic and control logic and memory operations, so this is left out of the table. The table shows the word size in bits of the architectures, which typically corresponds to the register size for these ISAs. The third column contains the amount of instructions in the instruction set. This is an important factor for implementation and complexity of the hardware. Next to that are the amount of general purpose registers. Finally a classification is given for the supporting software. This column is denoted as 'Toolchain' and more stars means the supporting resources are better. It is a classification based on amount of resources, software, emulators and other useful information surrounding the instruction set. The table shows whether interrupts are supported and if the ISA is 'Open', i.e. not protected against use in this project.

RISC-V, pronounced 'risk-five', is an instruction set constructed from different base ISAs, e.g. rv64i for 64-bit or rv32i for 32-bit integer operations. The base set is designed by incorporating the best features among different ISAs [42], [43]. RISC-V is modularly designed with application specific extensions to the base instruction set. The development is continuous and many of these extensions are not yet complete. The default configuration for RISC-V is 'RV64G' where the 'G' stands for the 'IMAFD' extensions together and it feature set is comparable to MIPS32 [43]. The Rocket Chip Generator project offers a high level implementation in Scala of RISC-V in a reconfigurable core [44]. The Scala files are then parsed by the Chisel3 compiler [45] to generate a Verilog implementation of the desired core. All these tools are openly available for use but deemed too complex in the case of the DM as the

```
// Some Interrupt Function
void foo_isr()
{
     ...
     ...
     // Return with uret
     asm("uret;");
}
```

Figure 3.3: Example of inline assembly programming in C.

Table 3.1: Comparison of instruction set architectures.

ISA	Bits	Instr.	Regi.	Toolchain	Interrupts	Open	
RV32I	32	50	32	★★★★	✓	✓	[42]–[44]
MIPS I	32	112	64	★★★	✓		[46]–[48]
MIPS32	32	177	64	★★★	✓		[41], [47], [48]
OpenRISC	32	48	32	★★	✓	✓	[49], [50]
OpenSPARC	64	140	640	★★★★	✓	✓	[51]–[53]
MICO32	32	61	32	★★★	✓	✓	[54], [55]
MMIX	64	134	256	★		✓	[56]–[58]

author is unfamiliar with Verilog and Scala.

The requirements for this project are covered by the 'RV32EN' extension, where 'E' stands for the 'embedded' base instruction set. This reduces the number of general purpose registers from 32 to 16. It also drops the need for operations that access the control status registers of the core. The 'N' extension covers the ability to use user-level interrupts but both extensions are still in development. The workaround here is implementing all the general purpose registers and only use parts of the 'N' extension. This means supporting the RV32I base instruction set. Surprisingly it is possible to employ 'N'-operations such as `uret` from C if we rely on inline assembly programming as can be seen in Fig 3.3.

MIPS I and MIPS32 are taken into consideration as it enables reusing previous work into the Dependability Manager [1]. Toolchain support is also extensive as MIPS is used in industry. However, implementing a MIPS32 compatible processor is an arduous task considering the amount of instructions. It is possible to eliminate part of that work by omitting the floating point coprocessor and using gcc with corresponding flags. Still Table 3.1 shows the amount of instructions when hardware floating point operations are disabled. Adding that to the fact that MIPS is proprietary technology makes it a bad choice.

The OpenRISC project aims to deliver an open and free instruction set architecture together with tools and hardware. The OpenRISC has created a RISC ISA,

accompanying simulator, hardware implementations and SoC design that feature the OpenRISC 1000 (or1k) processor [49]. They have ported Linux to their ISA and offer many tools to use the architecture [59]. Their ISA is also modular, the base instruction set is called ORBIS32 and features the same benefits as RISC-V.

The OpenSPARC architecture is available for use but it is not suitable for this project. It has a large amount of instructions and its 640 64-bit registers are certainly too much. There are more suitable ISAs. SPARC stands for Scalable Processor Architecture and is designed for data centers. OpenSPARC T1 was released as an open version of the UltraSPARC T1 for educational purpose [51]. This architecture complies with ARMv9 Level 1 and its features are described in Table 3.1. Presumably it features an extensive toolchain as the UltraSPARC is used in industry and is compatible with different operating systems. However, the amount of operations and the required general purpose registers makes it unsuitable for an small embedded environment.

The MICO32 RISC architecture has been made for use on Lattice Semiconductor FPGAs. The architecture software and hardware is available as source code and features interrupts and external device support [54]. The available soft core is implemented in Verilog. The device is feature rich and is ready for incorporation into this project. However, unfamiliarity with Verilog is the main reason to avoid using this architecture implementation.

The MMIX architecture was developed by Donald Knuth for educational purposes. Support for the project by the community seems limited and only a hardware implementation in Verilog is available. However, the incredible amount of registers for an embedded system, lack of interrupt support and scarcity of tools and documentation make this a poor choice for this project.

The choice for an ISA affects the whole project. This section discussed a handful of suitable open alternatives in a sea of ISAs. After careful consideration the RISC-V architecture is the preferred option to implement. Most architectures are not available for use due to intellectual property protection. The ones that are available have been compared. RISC-V stands out among the candidates due to its modular design and incorporation of common features among different ISAs [43]. It's supporting software toolchain consists of GNU C and C++ tools [60], [61], a processor emulator [44] and a configurable hardware generator based on Scala [44], [45].

3.2.2 High Level Language

This research is a continuation of the work towards the dependability manager [1]. The previous project had no support for a high level language. Proper integration of a high level language compiler is an important feature in this work. The choice for a high level language support for an embedded system is dictated by the instruction set. Since this choice was already made we rely on the tools available. For the sake of completeness this subsection discusses high level languages that can be used to program dependability applications. Next to the compatibility of the programming language with the instruction set, the suitability and supporting libraries of the language are discussed. Rust, C/C++, Python and Java are evaluated as high level languages regarding their support and limitations for running on targets.

Rust is a new concept programming language where memory and thread safety is guaranteed [62]. It is syntactically similar to C++ but adds new mechanics to ensure its intended goal. All values in the system have an implicit owner upon creation and are immutable unless otherwise specified. Current fully-supported targets for rust are the x86 and x64 architectures while other targets are supported with limitations [63]. There currently is no support for RISC-V, nonetheless, Rust would be a great language for dependability applications considering its safety features.

C and C++ are often named in the same breath and are considered the industry standard for programming embedded systems. C++'s object-oriented-programming characteristic is an added layer of higher level functionality to C. The language is considered mid-level as it lacks ease-of-use features present in modern high level languages such as garbage collection. C is compatible with many targets due to the design of its open source compiler that eases effort required for creating a cross compiler. The RISC-V architecture also features a cross compiler [60] to create bare metal code for the DM. Applications written in C are able to access memory locations directly which enables the MMIO operation of the DM's system devices. In general, function attributes can be used to declare interrupt service routines although this is not yet supported for RISC-V. Maintainability and readability of C applications require an effort on the programmer's part but adhering to code style conventions mitigates this problem [64], [65].

Embedded Python has been employed to program system such as the DM by running a minimal Python interpreter on the target [66]. This interpreter requires a significant amount of hardware resources, i.e. the hardware available on a Raspberry Pi [67] or a Beaglebone Black [68]. This CPython interpreter is a C application that runs the Python code. Running it on the DM means compiling this interpreter for

the chosen ISA. The Cython compiler translates Python to C code and then compiles to a Python module usable from the Python environment or from a C application. A dependability application can be made in Python and compiled to C and then compiled with a target-gcc [69].

Java is a popular programming language and supports many devices [70]. Java is compiled to Java byte code and subsequently executed by the Java virtual machine. Any target that features a virtual machine is able to execute Java programs. To support RISC-V an openly available virtual machine needs to be cross compiled, such as Avian [71]. Java was not designed to allow MMIO partly due to the virtual nature but some virtual machines feature direct access to memory nowadays [72].

The choice of ISA affect the high level language support tremendously. Python and Java feature the same method of operation, albeit a virtual machine or an interpreter, and both languages will require effort to run on the RISC-V ISA. Rust's benefits are enabled for certain targets, e.g. x86 and x64, but not (yet) supported for the RISC-V architecture. This leaves the GNU gcc toolchain supplied by the RISC-V developers. Using C for this project is beneficial as it is an industry-proven language with a long successful history. It can run bare metal which reduces the code size and improves the processing speed.

3.2.3 PDL Incorporation

Now the discussion between ISA and high level language support is completed a design is needed to incorporate the PDL programs. First an introduction to and previous work on the PDL compiler will be given as it is a starting point for how the programmer could use the DM. After a design is made to effectively use PDL specifications from a high level point of view.

PDL, Procedure Description Language, is a programming language that enables manufacturers of IJTAG products to offer usable programs with their instruments or components. IJTAG tries to mitigate scalability issues of test networks by reducing the workload on programmers when they want to operate the network as processors may incorporate tens to hundreds IJTAG instruments. Creating custom code to configure and access each instrument is a cumbersome and error-prone method. A PDL program provides tried and tested code bound to a specific instrument. The retargeting tool interprets the programs and executes the read and writes to the instruments.

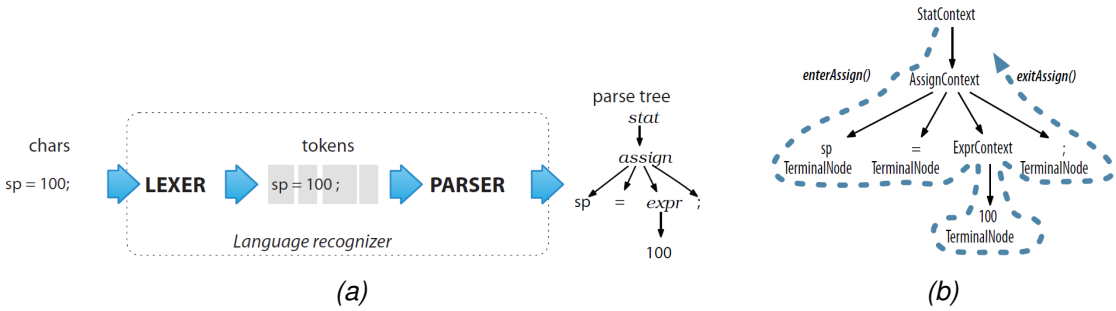


Figure 3.4: Language recognizer generated from a grammar changes input file to tree structure which is then walked. Custom code is called to perform the necessary actions at each node. Figures taken from the Antlr4 Definitive Reference [73].

The work of Zakiy introduced a PDL compiler from one complete PDL/Tcl script to an assembly program [1]. This compiler is based on the Antlr project which turns a BNF syntax grammar specification into a usable Lexer, Parser and Walker for an Abstract Syntax Tree (AST) [22], [73]. The Lexer will read a program and turn it into a stream of tokens. This tokenstream is parsed and stored as a tree data structure. This AST can then be walked by an observable TreeWalker object which calls functions in an observer class upon entering and exiting a tree node. The process is shown in Fig 3.4.

Zakiy's syntax grammar specification incorporates Tcl next to the PDL syntax in align with the IJTAG standard that defines that PDL level-1 programs may feature Tcl. However, Tcl is not a standardized language (reserved keywords may be redefined) which introduces the problem that no grammar was available and had to be reverse engineered. The IJTAG standard features an Antlr grammar for PDL (and also one for ICL) language and was modified to parse a subset of Tcl [1]. Unfortunately during this process many concessions where made and the resulting grammar could only be used to compile custom PDL code, `iProc` and `iCall` where completely ignored. It was not able to compile BASTION benchmark files [27].

The intention of this research is to incorporate PDL in a usable method in a high level language. A custom subset of PDL is not an option, it eliminates the benefits that PDL has to offer. It was made to be compliant with Tcl interpreters and it could incorporate Tcl commands if needed. Going forward we have two options as the previous grammar does not suffice. The first option is to create a compiler for the original PDL specification, i.e. use the non-modified grammar from the IJTAG stan-

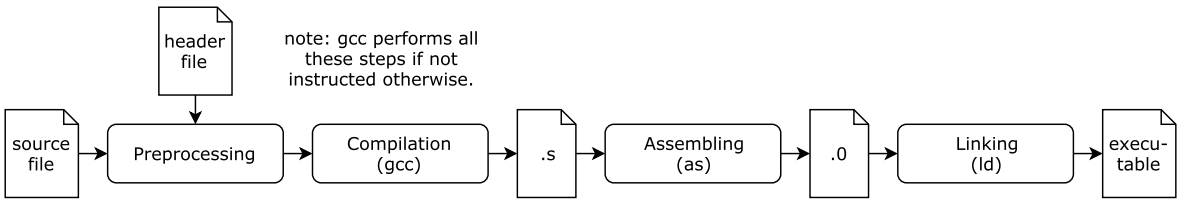


Figure 3.5: Steps of the GNU C compiler.

dard. The second option is to compile a Tcl interpreter for the DM and let it run PDL files directly on the target. The normal Tcl interpreter (tclsh) program is too large to fit on most embedded systems. A minimalistic interpreter such as Partcl [74], [75] could be used, it is extendible with custom Tcl commands which would be the PDL definitions and has been tested on an embedded system. The custom commands would control the retargeting engine and the interpreter would be configured for PDL. However the project is considered a 'toy project' by its creator, would require the whole dependability application to be written in Tcl and would be a far reach from the previous work. This leaves the PDL compiler as remaining option.

Among the choices for compiler output are high level language, assembly, and machine code. gcc uses these intermediate formats, see Fig. 3.5, and using these formats eases the integration of PDL and C. Some form of interfacing is required for using the PDL from the dependability application. So, basing the design on what tools are available is a reasonable choice. Assembly (.s) and machine code (.o) offer a complete control over the machine. However, compiling to assembly come with challenges. Storing and loading of variables needs to be managed, including the locations in memory. All administration of the machine needs to be done. The PDL compiler itself will also be responsible for managing the symbols (function and variable names) encountered in the PDL. Calling procedures in assembly must be correct. And the output of the compiler must adhere to the standard for linker files [76]. This is a tremendous amount of work compared to outputting high level language code. Fig. 3.6 shows the ease of converting the PDL syntax to C. Although this approach also has its challenges, the gcc compiler can manage the symbols, calling structures, and system management. This approach will only require correct parsing and symbol management of the PDL.

As discussed, Zakiy's grammar differs tremendously from the actual PDL syntax [1]. Zakiy's PDL compiler also outputs MIPS32 assembly while this research will use RISC-V as an open ISA. Rewriting the previous compiler is a hard task and due to the wrong grammar and nature of Antlr4, i.e. generating the framework and providing method stubs, also a fruitless task. Implementing the compiler from scratch,

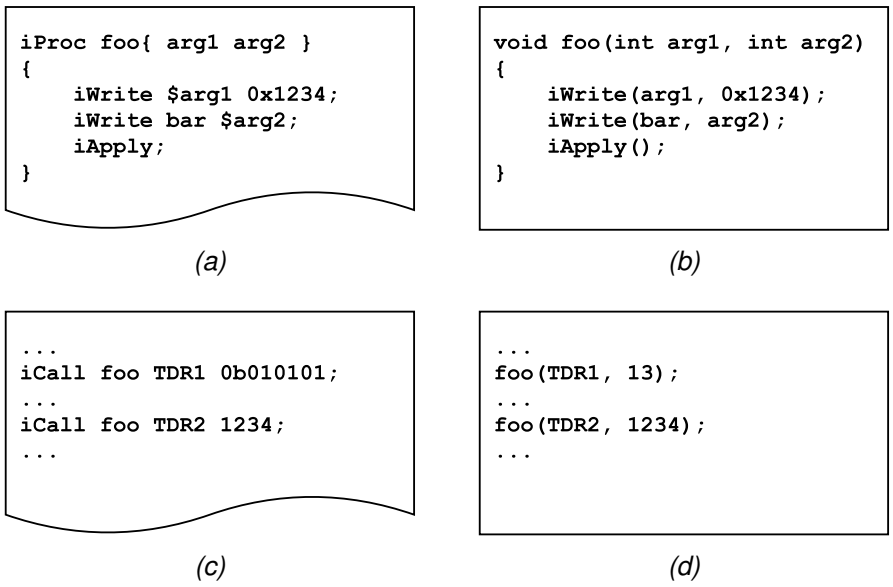


Figure 3.6: Two examples of how PDL can be converted to C.

by generating a new lexer and parser and filling the stubs, is necessary considering the grammar differences. The ISA used in the previous work is hard-coded into the compiler. The syntax specification is readily available from the IJTAG standard [5], so generating a new compiler with Antlr is trivial.

The previous work mentions portability as an option for future work [1]. In order to support any machine, and provide a hardware abstraction layer (HAL), a paradigm shift is needed for the compiler. This HAL consists of two parts: the instrument PDL library/framework and the retargeting engine drivers. To use the hardware effectively from software a layer of 'driver' software is created. This can also be seen in Fig 3.6; `iWrite` and `iRead` are functions in the retargeting engine driver. If the hardware changes, only the drivers will need to be updated to ensure the software executes. The other part, the instrument PDL library, will be a translation of the PDL file accompanying a instrument. The code in such a file is always contained in `iProc` procedures and these are an excellent collection of methods or a framework for the dependability application. Primarily because the procedures should contain relevant tests or methods to extract data from an IJTAG device. The driver software will be discussed in Section 4.3. The PDL instrument framework is elaborated here and in Section 4.1.

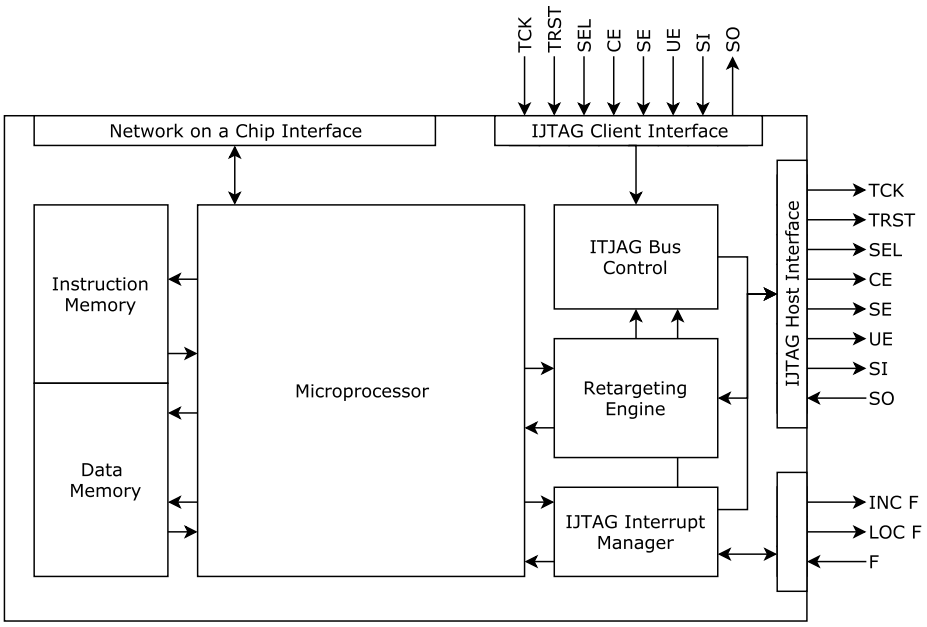


Figure 3.7: High level system block diagram.

3.3 Hardware design

A high level design for the dependability manager is discussed in this section. The application analysis showed no need for a specifically fast architecture. Recall that updates in the control algorithm happen with seconds in between. The calculations performed by the DM will be intermittent. Simplicity is at the core of the DM, to reduce implementation time and increase maintainability. Construction of performance increasing technologies such as pipelining and branch prediction will remain out of the scope of this project. The microprocessor will need multiple clock cycles per instruction as it moves through its execution stages. The focus during implementation will be on the retargeting engine and interrupt management unit. These will be necessary components to cope with the IJTAG networks.

An block diagram of the DM can be viewed in Fig. 3.7. The core component is a microprocessor. Research into usable ISAs has yielded RISC-V as base instruction set for this microprocessor. The ISA is comparatively small and can be extended in future versions if needed. The microprocessor will use memory to store data and instructions. A Harvard architecture would be better than a Von Neumann architecture as it protects the instruction space. If the execution of instructions would be pipelined, Harvard also allows the separate fetching of instructions while the data bus is busy. A brief research into bus protocols will help the implementation. Reusing an existing popular protocol will aid in the adoption of the RE by others.

The previous work used MIPS32 coprocessors to house the retargeting engine. Direct communication and specialized instructions were used to operate the RE. The register space in the coprocessor is limited to 32 words which was used to store TDR id and value. This research will implement the RE as a separate entity with a standardized communication scheme, based on the system bus protocol. The two parts of this strategy is the MMIO and driver paradigm. The registers of the RE will exist in the memory space of the DM at fixed positions. The drivers will know where to write the access requests and will control the RE with a dedicated register.

The IJTAG interrupt management unit will also be implemented in this research and an interface with the microprocessor will be designed. A complicated architecture will be avoided due to time constraints. The processor will need a jump address and a signal for when it needs to jump. Returning from an interrupt will be handled by the RISC-V `uret` instruction, although it is not strictly part of the base ISA.

The interrupt management unit and the retargeting engine will both communicate with the IJTAG network. A dedicated TAP controller is needed to make sure the protocol is adhered to. This bus controller will synchronize the network, tap state (see Fig. 2.2) and IJTAG master devices.

Communication with the functional layer may be a beneficial feature of the DM. It allows error reporting to the operating system of the SoC. The implementation of this communication will remain out of the scope of this research. The same holds for routing external IJTAG signals through the DM to the instrument network, see Fig.3.7. Thankfully, these components can be added to the hardware as an extension due to the nature of the DM's design, i.e. the system bus and memory mapped I/O devices.

3.3.1 Retargeting Engine

The retargeting engine is a hardware acceleration device in the DM responsible for communication with the IJTAG network. It will reconfigure the network and read and write values from the TDRs. Its design is based on an algorithm and data structure for retargeting [2]. The operation of the retargeting engine is described here.

The retargeting engine receives access requests (AR) from the processor. These requests contain an instrument id, a read/write flag and a value if it needs to be written to the instrument. The retargeting engine stores these requests in a list of stacks, each stack featuring the original AR as bottom element. These stacks are used dur-

ing the retargeting process where the Hierarchy Array (H-Array) is traversed. The H-Array contains a structured list of the instruments in the IJTAG network. As the reader might recall, these networks consists of Segment Insertion Bits, ScanMuxes and Test Data Registers and these are the types encountered in the H-Array.

The network state or configuration is stored in the State Vector (SV). This is a bit string containing current values of the ScanMux Control Bits (SCB), the registers in the network controlling the ScanMuxes. The configuration forms the *active scan path*, the current path through the scan network. In the H-Array, the segments inserted by a ScanMux have headers (I0 or I1) which are a representation of the state of that ScanMux. This also holds for the SIB, which skips or includes a segment. The segment headers store a reference to their SCB and store the amount of elements in the H-Array segment. The SCBs in turn store a reference to a position in the State Vector. The retargeting engine uses this data to execute the traverse and generate algorithm. An example of a network along with its H-Array is shown in Fig 3.8.

The goal of the algorithm is to reconfigure the network so that the requested instruments in the Access Request stack become part of the active scan path. The algorithm generates the access vector during the traversal. Instruments are identified by their position in the H-Array, this index is the instrument-id. When it encounters a segment header, the state vector is checked based on the SCB that accompanies the SIB or ScanMux. If the segment is in the active scan path, the algorithm enters the segment and otherwise it will skip. When an non-active segment header is encountered during traversal a check is performed to see if a requested instrument lies in the segment. This is achieved by checking if the instrument-id lies between the current H-Array-pointer and the end of the segment. If so, the algorithm then adds an access request to the stack of the requested instrument. This access request is a write to the corresponding SCB and the value is based on the segment header. ScanMuxes can have a ’0’ or ’1’ as desired value, while SIBs always require a ’1’ to open.

When an SCB or TDR is encountered in the traversal algorithm, it checks to see if the top of any stack contains that SCB- or instrument-id. If so, the access request is popped of the stack and the desired value is added to the scan vector. For read operations an extraction request is created and later used to extract the requested value out of the incoming shifted bits.

After traversing the H-Array and generating the access vector the Retargeting

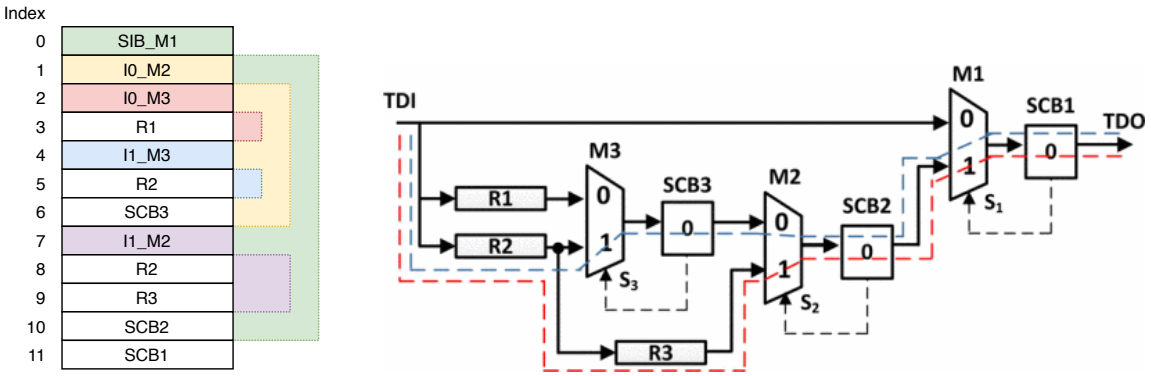


Figure 3.8: Example IJTAG network with H-Array [2].

Engine moves to the next state where the communication with then network happens. A CSU cycle is initiated and the access vector is shifted into the network. The incoming bits are stored and shifted into a read buffer. After shifting the retargeting process is started again. The retargeting engine is done when no access requests are left on the stacks.

An example network can be seen in Fig. 3.8 along with its H-Array. The headers are coloured along with their corresponding network segments. M1 can be seen as a SIB and M2 and M3 are ScanMuxes. The current configuration (all '0' in the SCBs) means the SIB is closed. There are different paths through the network as indicated by the blue and red dotted lines. In order to configure the active scan path to the blue line (access R2) multiple shifts are needed. First a '1' is shifted into SCB1 and this opens M1. In the second cycle, a '1' is again shifted into SCB1 but SCB2, SCB3 and R1 have become part of the scan chain. M2 is already configured correctly and the retargeting process therefore encounters the headers of M3. An access request for SCB3 is added as the I1_M3 needs to be opened. The algorithm encounters SCB3 later in the H-Array and the value is set in the access vector. To retarget the network from this state to access R3 another cycle is needed which shifts a '1' into SCB2.

The implementation of the retargeting engine algorithm can be realised with a state machine. The retargeting engine can be idle, traversing and shifting. The retargeting engine should also be designed as IP, so that reuse is simplified. The control of the retargeting engine will happen by 'starting' it and monitoring its progress. This can be done with a control and status register where specific bit fields are coupled to a hardware function similarly to the instruction register of the processor controller. Communicating the access requests to the retargeting engine will be done via some

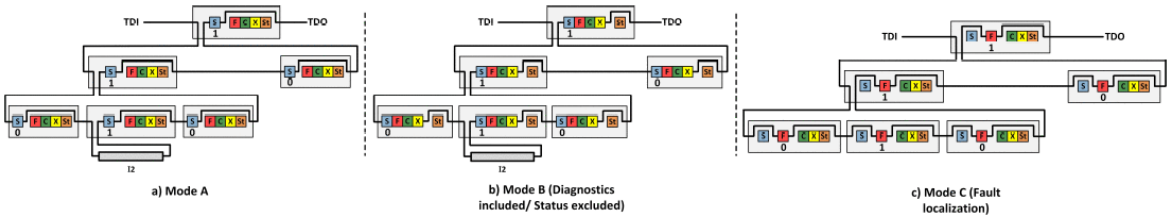


Figure 3.9: Different modes in the extended SIB, taken from [3].

buffer. The drivers could also access the AR-Stack directly and put the request in the right place on the stacks. It is also clear that storing the H-Array, Access Request stack, Read Requests and the shift buffers will take a certain amount of memory. Making these items configurable in size allows for an easy design space exploration and optimised hardware configuration.

3.3.2 Interrupt Manager

The dependability manager will support interrupt features in its design. Normally, an interrupt management unit (IMU) will handle all interrupts in a system, i.e. from external signals, memory/arithmetic traps and peripheral interrupts. Interrupt managers may feature interconnect fabric and masking that couples sources to service routines. These devices are highly configurable to speed up the servicing of interrupts.

The interrupt manager of the dependability manager will be simplistic; it's main goal will be localising and managing interrupts originating from the IJTAG network. This is a new concept and not part of the IEEE IJTAG standard [3]. It adopts the SIB and redesigns it to have three distinct modes, see Fig. 3.9. The first (Mode A) is normal operation as described in Section 2.3. The second (Mode B) allows for inclusion of novel flag registers into the scan chain at the retargeting tool's discretion. The last (Mode C), also referred to as 'localisation mode', discards the SCBs and TDRs from the scan chain and only includes the flag registers. There can many different flags that all have a system-wide *include* and *localisation* signal. The first allows for Mode B operations while the latter enables Mode C.

This research will focus on the implementations of the network structures and the interrupt manager for only one type of flag. The interrupt manager is notified of an interrupt with a flag propagation network and will localise the source of a flag with a special inverted version of the H-Array, the IM H-Array. The network layout plays a substantial role in the localisation process. Extended SIBs in a hierarchical tree network are the backbone for the algorithm. An example network is shown in

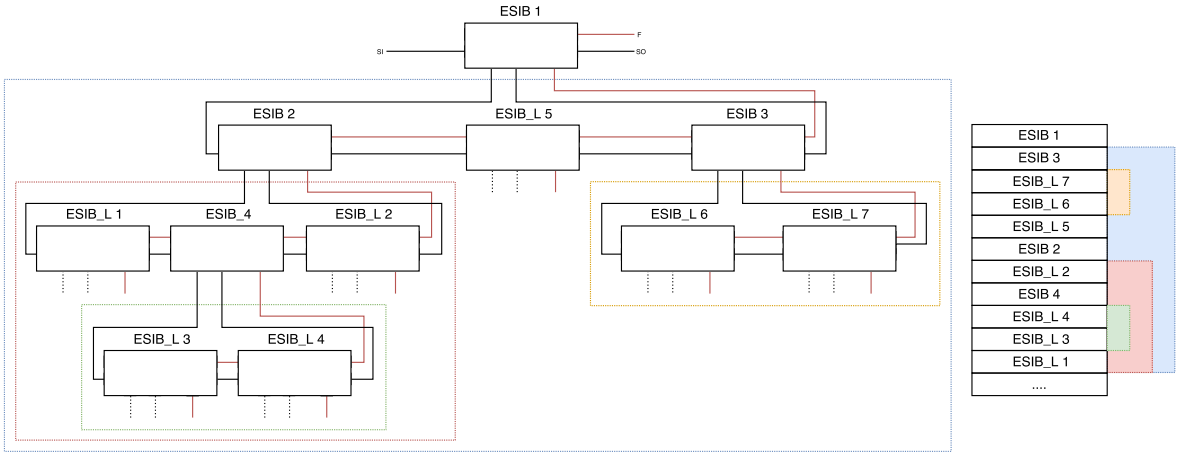


Figure 3.10: An interrupt enabled IJTAG network. The flag propagation network is shown in red. The colours of the IM H-Array correspond to segments in the network. The use of ESIBs and ESIB_L is discussed in Section 5.3

.

Fig. 3.10 along with its IM H-Array. The segments are coloured in the network and in the IM H-Array representation. The flag is propagated through the red line. Each extended SIB takes an input from the left and from below, these inputs are *ORred* and propagated to the next SIB [3]. This starts the localisation algorithm which is discussed in the next section.

3.3.3 Extending IJTAG for Interrupts

A part of this research is implementing the interrupt-enabled IJTAG SIB [3]. This research will reuse the VHDL network structures provided by the BASTION project [23], [27]. These will be adapted to operate according to the research into IJTAG fault localisation.

BASTION's benchmark set feature 24 different networks with varying degrees of ICL and VHDL specifications, example PDL files and structure diagrams. The network VHDL implementations reuse a set of entities that represent different IJTAG scan chain structures or instruments such as SIBs, multiplexers and SCBs or TDRs. An interrupt enabled network consist of a flag propagation network, interrupt-enabled SIBs and interrupt-generating TDRs. The hierarchical tree-layout that is needed for fast localisation consists of SIBs with the instruments as leaves [3]. The SIB implementation is extended with logic and control signals.

The control signals are used to configure the mode of the ESIB. It can be in normal operation mode (Mode A) where it behaves as a regular SIB and skips the extra

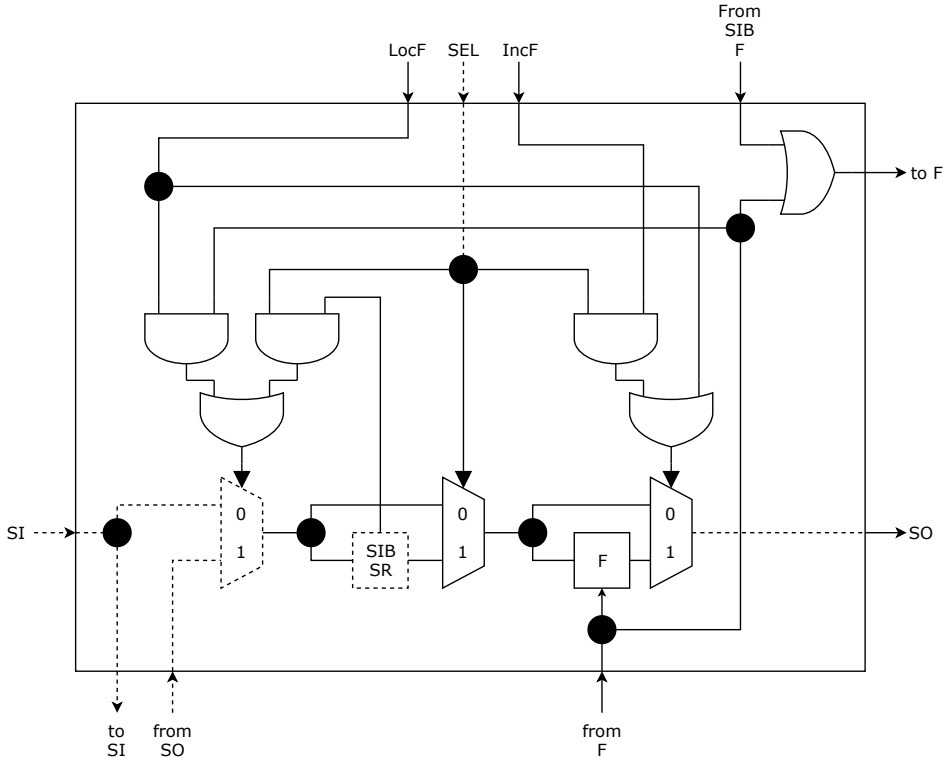


Figure 3.11: Structural design of an Extended SIB.

flag, it can be configured to include some flags for diagnostics (Mode B) or it can be used to localise the source of a flag (Mode C). This research focusses on the latter mode for interrupt localisation while the same principles can be applied to add other flags [3]. A detailed design diagram of a ESIB can be found in Fig. 3.11. The design shows the ESIB's segment control bit (SIB SR) and its flag register (F). The flag register's value is set during the capture state of the CSU cycle. An interrupt is propagated through an or-gate network to the DM which configures the ESIBs to mode C by setting the 'Select' signal to logical low and the 'Loc F' to logical high. This means that the SIB SR register is omitted while the 'F' register is added to the scan chain. The 'F' flag (note **not** the 'F' register) controls whether the underlying network is added to the scan chain, this also means an instrument must keep its interrupt signal high until it has been serviced by the controller. Otherwise, localisation will fail.

Localisation is done by performing a CSU cycle, and as mentioned, the capture phase stores the interrupt flag in a scan chain register. During shifting the values of the ESIBs are received and used to traverse an interrupt based hierarchy array. This IM H-Array is different from the one used in the retargeting engine. The entries of this IM H-Array are ordered correspondingly to bits shifted out of the scan chain. Reading a logical '0' means that the ESIB can be skipped, reading a '1' means that we must 'enter' the segment beneath the ESIB. The IM H-Array is traversed in this

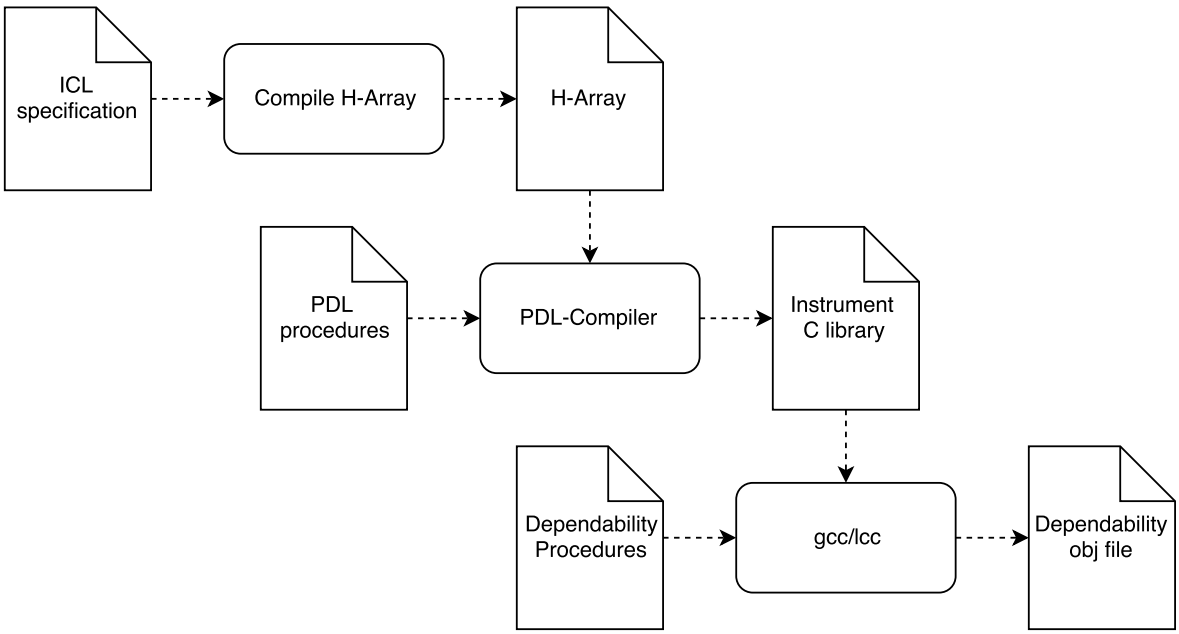


Figure 3.12: Compilation of a dependability application.

manner until a TDR is encountered which will be the interrupt source. However, during implementation it proved useful to add a leaf ESIB definition to the network and IM H-Array. Because TDRs are not active if not selected, the regular ESIB would propagate undriven signals. The leaf ESIB is considered as the interrupt source for the underlying segment, which would be a single smart instrument.

3.4 Software design

So far, the supporting software for the project consists of a compiler for the RISC-V instruction set. Incorporation of PDL will happen by transforming the scripts into a framework which can be called upon. This instrument C library, or framework, will consist of the `iProc` procedures from the PDL file. This generated code will can then be used as a C library for a specific instrument network. The dependability application will be linked to the framework, which in turn will incorporate a set of drivers for the hardware of the dependability manager. The set of drivers and the framework will be referred to as the dependability managers library.

The compilation flow is shown in Fig. 3.12. The generation of the H-Array is the first step as it is required for the generation of the library. The instruments in the network need to be known at compile time. The H-Array provides a structured mapping between instruments, their location in the network, and the ID in an Access

Request. The H-Array and its creation from a Selection Dependency Graph is part of the research into the retargeting engine [2], [26]. The generation of a Selection Dependency Graph by parsing the ICL has not yet been attempted and will not be part of this research. The assumption is that a H-Array specification is available and that it will be used as an input for the framework generation process. For example, the names of the instruments can be replaced by their indexes. These indexes can be used to issue `iRead`s and `iWrite`s.

The PDL compiler shown in Fig. 3.12 will take the instrument level PDL procedures provided by the instrument manufacturer and turn it into a library in the chosen high level language. The purpose of the framework is to provide the developer of dependability procedures/applications an easy entry point for operating the IJTAG network. The developer of the application can configure a device through the generated PDL library. This is based on the assumption that the manufacturer has created such procedures in the provided PDL. Examples of that are `iProc`s for build-in-self-test or initial configuration. The developer may also use the retargeting engine driver to issue `iRead`s and `iWrite` directly.

The software framework is joined with the DM specific drivers and compiled along with the dependability procedure into a executable file for the DM. This file must then be loaded into the DM simulator or synthesis tool to be used during the lifetime of the system. Simple scripts can be made to achieve this.

3.5 Discussion

This chapter has discussed the preliminary design of the dependability manager entirely. It starts with an analysis of some different dependability applications and the suitability of IJTAG as a means to sense and actuate. It then works through some design considerations of the previous work, how they apply in this next iteration of the dependability manager and then addresses them accordingly. This works aims to integrate PDL functionally in a high level language while previous work only focussed on executing PDL programs on an embedded system. Finally a top-level design is presented in the form of a block diagram (see Fig. 3.7) which separates the Retargeting Engine and Interrupt Manager as separate system bus devices. These devices will use an interface which enables reuse in different processor designs. The software will consist of a toolchain converting the PDL to a high level language that enables the same portability.

This chapter has outlined a plan for the separate Software and Hardware tracks

in the proposed methodology and provides details for the cooperation between the two parts. This plan for the dependability manager coprocessor will be elaborated in the next chapters.

Chapter 4

Software: Building the Toolchain

This chapter contains the software development part of this hardware-software codesign project. The methodology used in this research states that this must yield some form of compiler for the ASIP in development. It was established that gcc will be the main-compiler for the DM. However, incorporating PDL is one of the additions of this research. A toolchain for the DM will be developed in this section for incorporating PDL. It contains the PDL to C framework compiler. Along with the compiler comes a set of drivers to create a HAL between the dependability application and the dependability manager.

As discussed, the previous approach [1] to compile from PDL to MIPS32 assembly directly was practical but made the compiler too rigid for reuse. For example, the operation of loading data in the retargeting engine was specific for the MIPS32 machine. Furthermore, the grammar used to generated the compiler did not comply to the IJTAG standard. This research uses the PDL grammar provided by the standard [5]. C was chosen as an intermediary language so that the generated PDL framework can be compiled to other targets which makes it reusable in the future.

The chapter starts with a research into the finer details of PDL operation. The framework needs to use the H-Array data structure to identify instruments. Resolution of the right instrument is done during runtime and this creates problems in a static framework. Such concepts of the PDL language need to be converted to C to reach a working concept of a framework compiler. This chapter then proceeds with connecting the operations in the generated framework to the system components using drivers. These drivers are implemented according to the design of the hardware. All operations happen via Memory Mapped IO and drivers mostly revolve around setting the right bit in the right place to start retargeting or enable interrupts.

4.1 PDL to C Framework Compiler

There is a need to incorporate PDL into the dependability application. Instruments in the network can be configured for different testing purposes, e.g. a sensor may have gain or resolution settings. Instrument manufacturers may supply PDL programs along with their product. Since PDL is the de facto language for IJTAG instruments it saves time and money to enable incorporation of these available scripts. In Chapter 3 a discussion about different methods of PDL incorporation is held. Fig.3.12 shows the preliminary design of the toolchain. This section picks up the pace and implements the PDL2C framework compiler. The resulting compiler will transform PDL files to C files in order to be used with the dependability application.

PDL is a language which defines procedures for ICL modules. PDL is divided in two sets; level 0 and level 1. All PDL level-0 commands exist within `iProc`s except for the version declaration (`iPDLLevel`) and the `iProcsForModule` annotation. No dedicated `iProc` is the entry point for the program and this is left for the creator of the retargeting tool. The `iProc`s contain the commands for reading and writing (scan operations) and can call other procedures with `iCall`. PDL level-1 commands are used by the Tcl testing tool to access values; all commands in the PDL level-1 set return some kind of String datatype. PDL is syntactically compatible with Tcl and PDL level-1 files may also contain any Tcl operation. This research will drop the PDL level-1 commands and will not compile Tcl commands as was done in Zakiy's work [1].

4.1.1 Namespacing in PDL and ICL

The nature of IJTAG lets developers be more concise in module reuse both in hardware specification and software. ICL and PDL offer namespacing to match software with a specific instrument in the network. This is due to the close interaction of both languages but also the need for managing large scan networks. Although the compiler is designed with the H-Array in mind as network specification this subsection will discuss namespaces in PDL and offer a design of how it should be when H-Array generation becomes fully automatic.

A network starts with a single top level module, how this should be handled is left to the user according to the IJTAG standard. Modules exist in a namespace and if none is defined they exist in the 'root' namespace. It is presumed that an instrument in ICL is defined as a module. Instances always exist within a top-level module and this creates a hierarchy. The entry point of a PDL program should be the `iProc`

defined for a top-level module. Subsequent `iCall` and `iPrefix` commands operate within this module and influence the effective prefix. An `iCall` can target an instance within a module. The procedure that is called should be defined for the module which is that instance. This is annotated with the `iProcsForModule` command. An example of this can be seen in Listing. 4.1 where Mingle is the top-level module. The `iCall` targets a different instrument (`WI1...WI7`) while calling the same `iProc`.

The scan operations, e.g. `iWrite`, `iRead`, `iScan`, always target a register, port or alias for one of those. The effective prefix determines the specific instrument in the network. The same procedure can be called for different instances of a module. This means that the same piece of PDL code is used for different instruments. The intended instrument can only be found by following the calling sequence. This creates difficulties when creating a static PDL framework and makes mapping the scan operations to the H-Array harder.

4.1.2 Namespacing Tree Solution

It is necessary to keep in mind that the framework uses the H-Array as a structural model of the network. This abstraction contains TDR entries as addressable instruments. During the creation of this linear array, the sense of namespacing defined in the previous subsection is lost and no effort is made to allow the use of prefixes. In this subsection a suitable approach for this problem is discussed.

The Mingle network is used as an example during this deliberation. A schematic figure and the H-Array of this network is available in Appendix C. An entity diagram is show in Fig 4.1 which gives insight in the namespacing and hierarchy of instruments. The Mingle network contains eight instances of the `WrappedInstr` module, which is a generic read-write instrument. There are also some SIBs and SCBs which have addressable 1-bit `ScanRegisters` to configure the network. The last structure in the network is a `BypassRegister` without any function. Consult Fig. C.1 in Appendix C to view the complete network structure.

As mentioned, the effective prefix for an `iProc` can be set with an `iCall` and extended by an `iPrefix` command. This dynamic movement through `iCall` namespacing creates challenges for generating a static framework. When you call an `iProc` for a specific instance of a module, you have to know which specific instance you were addressing. A work-around is to require every scan command to originate from the root-prefix but this hinders the scalability of PDL. The dynamic namespacing enables reuse of `iProc`s for multiple instances of the same module. Such an `iProc`

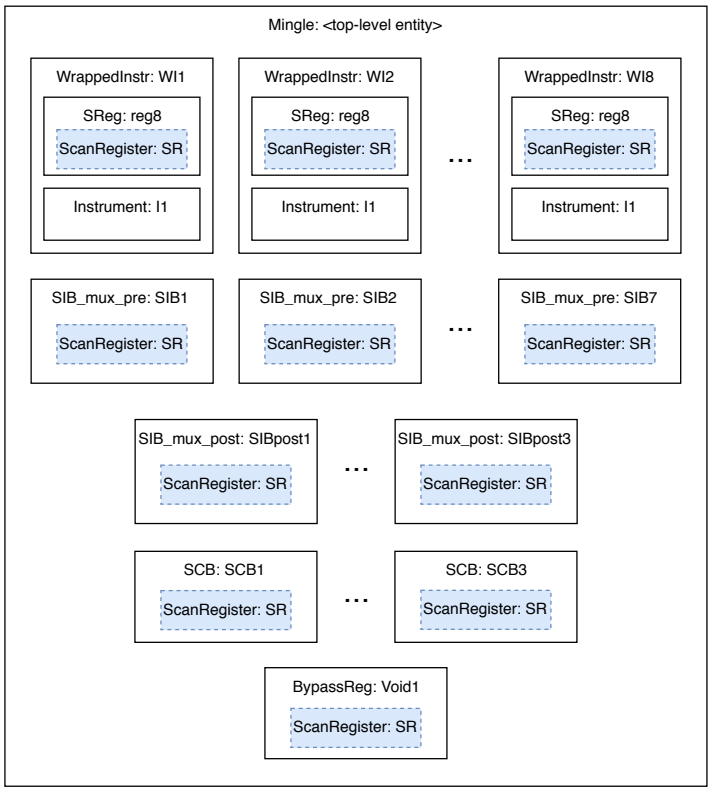


Figure 4.1: The hierarchy of entities in the Mingle module, similar instances of the same module are omitted, a full entity tree can be seen in Fig C.2 in Appendix C.

can be seen in Listing 4.1 for the Mingle network. The same procedure is called for every instrument in the network. Without PDL's ability to change the prefix, i.e. the work-around, there would need to be a `iProc` for every instance of the module which is not a good practice.

At closer scrutiny, namespacing in a test network is a tree structure with the root-prefix as top-level parent node. The program starts at the top-level entity and an `iCall` selects an entity by descending the tree and finding the right child. During the execution of the `iProc` all subsequent `iCall`s are based on the tree node that was selected. `iPrefix` may be used to descend further down the tree but will never go up, i.e. towards a parent or the top-level node. The tree for the Mingle Network is shown in Fig. 4.2. Returning from an `iCall` restores the prefix from before the `iCall`; the correct tree node must be reselected. The flow of the program in Listing 4.1 is shown in the hierarchy tree representation of Fig.4.2. The order of `iCall`s of the program in Listing 4.1 is also added to illustrate the difference in operation. The jumps are also denoted in the comments of Listing4.1. Generation of this hierarchy tree structure can be done from ICL but will sadly be out of the scope of this research. It is best to design these mechanisms in a later stage where the H-Array generation will also be implemented further.

This leaves the work-around as viable alternative for this project. The hierarchy diagram in Fig. 4.1 shows the actual addressable items in blue; they are `ScanRegister` ICL definitions. The PDL extract in listing 4.2 shows these registers as target for the write operations. As it is, determining the target of a scan command is straightforward as the command is issued from the root-prefix; the entire path to the ScanRegister is given. Requiring the effective prefix to remain empty allows the conversion of PDL to a static framework as the calling sequence does not influence the instrument path any more. In order to map the scan command target to the H-Array, the full path must be stored for every TDR.

This resolves a difficult part of the compiler. The solution that is employed is not the best but it works. A tree structure to store the instrument hierarchy along with their H-Array indices would be best for the generation of the framework. This structure would also be stored on the DM to allow retargeting for scalable networks and reuse of provided PDL. However, as parsing ICL and H-Array generation remains future work, so will the generation of this hierarchy tree.

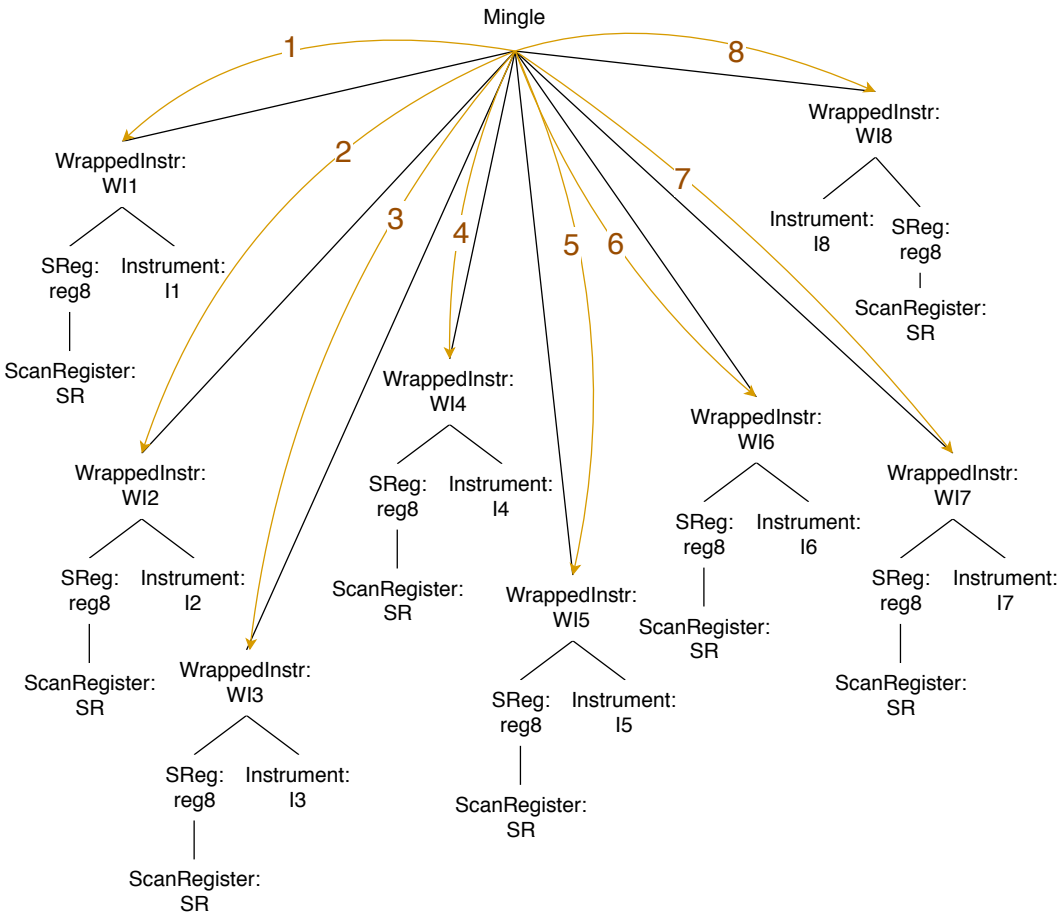


Figure 4.2: Mingle's hierarchy tree, the SCB and SIBs are omitted. The flow through the hierarchy tree of the PDL program in Listing 4.1 is shown in order. A full hierarchy tree can be viewed in Appendix C, Fig. C.2.

Listing 4.1: PDL file for the Mingle network showing the use of prefixes during procedure calling.

```
iPDLLevel 0 -version STD_1687_2014;

iProcsForModule root::Mingle;

iProc program_entry_point{}
{
    # Start at the top level entity
    # Refer to Fig. 4.2 for the jumps in the comments below

    iCall WI1.write_to_wrapped_instrument;  # Jump to child 1
    iCall WI2.write_to_wrapped_instrument;  # Jump 2
    iCall WI3.write_to_wrapped_instrument;  # Jump 3
    iCall WI4.write_to_wrapped_instrument;  # Jump 4
    iCall WI5.write_to_wrapped_instrument;  # Jump 5
    iCall WI6.write_to_wrapped_instrument;  # Jump 6
    iCall WI7.write_to_wrapped_instrument;  # Jump 7
    iCall WI8.write_to_wrapped_instrument;  # Jump 8

    # End of program
}

iProcsForModule WrappedInstr;

iProc write_to_wrapped_instrument{}
{
    # Find scan-entity 'SR' below child 'reg8'

    iWrite reg8.SR 0bx11110000;
    iWrite reg8.SR 0bx00001111;
    iApply;

    # Return to the tree node at the moment of iCall
    # In this case the top level entity, i.e. the root.
}
```

Listing 4.2: Extract of the Mingle PDL file of the Bastion benchmark set [27]. All `iWrite` commands originate from the root-prefix.

```
iPDLLevel 0 -version STD_1687_2014;

iProcsForModule root::Mingle;

iProc all_scanregistes_in_one_iApply {} {
    iWrite SIBpost1.SR 0b1;
    iWrite WI1.reg8.SR 0bx10001110110111101100101100011010;
    iWrite WI6.reg8.SR 0bx11101000011010011111011000011111;
    iWrite SIBpost2.SR 0b1;
    iWrite SIBpost3.SR 0b1;
    iWrite Void1.SR 0b1;
    iWrite WI5.reg8.SR 0bx11011111111100001000011100011110;
    iWrite SIB7.SR 0b1;
    iWrite WI4.reg8.SR 0bx10010111010110000011000000011101;
    iWrite SIB6.SR 0b1;
    iWrite SIB4.SR 0b1;
    iWrite SIB5.SR 0b1;
    iWrite SIB3.SR 0b1;
    iWrite SIB2.SR 0b1;
    iWrite WI2.reg8.SR 0bx10001100100011001111010000011011;
    iWrite SIB1.SR 0b1;
    iWrite WI7.reg8.SR 0bx10000111100101100101001000000;
    iWrite WI3.reg8.SR 0bx10110111011001111010100100011100;
    iWrite WI8.reg8.SR 0bx10011100101111011101010000100001;
    iWrite SCB1.SR 0b1;
    iWrite SCB2.SR 0b1;
    iWrite SCB3.SR 0b1;
    iApply;
    ...
```

4.1.3 Data Types

The scan operations within PDL intend to get bits in the right place. These bits are handled as numbers and are expressed with decimal, hexacdecimal or binary notation. As PDL is created to be executed by a Tcl interpreter it is reasonable to assume that their data types are similar. The IJTAG standard defines all PDL numbers as 'unsigned integers' [5], expressed as decimal, binary or hexadecimal numbers. This raises the question what the size of the integers should be. Tcl, on the other side, stores most variables as Strings but integer computations were performed with the C datatype *long int* up until Tcl version 8.5 [1]. This data type is 32 bits and this size can be handled for accessing instruments. This means that `iRead` and `iWrite` can provide a value with maximum size of 32 bits. If necessary, larger TDRs could be split into multiple pieces to receive data. This would result in sequential entries in the H-Array. For simplicity, this research determines that the maximum value size for TDRs size will be 32 bits.

4.1.4 Handling PDL Code Annotations

This subsection contains a research into the language annotations provided by PDL for the compiler. The standard does not differ between annotations and commands but our proposed PDL will need to handle them differently. This subsection discusses the operations that do not directly affect the test network or the program flow. The purpose of each command is evaluated and a proper solution in the framework is discussed. This will yield a base for the drivers of the Retargeting Engine as they will need to facilitate the behaviour of the commands.

`iPDLLevel` Every PDL file starts with this declaration to inform the retargeting tool which version of PDL it is parsing. Because the original JTAG standard also defines a set of commands which is dissimilar to the IJTAG standard. The compiler should check the version of PDL and warn the user.

`iProcsForModule` This annotates the module for which the subsequent `iProc`s are valid. The effective prefix of commands issued in those procedures will need to target a instance of that specific module. Due to the work-around in namespacing, only `iProc`s for the top-level entity should be allowed. This also implies proper addressing when parsing the instrument paths of scan commands.

[1] https://www.tcl.tk/man/tcl/TclCmd/expr.htm

`iPrefix` This command adds to the effective prefix of the `iProc`. The prefix builds a hierarchical path to target the instruments and registers in the network. When calling a `iProc` the path is build with the `iCall` command. The `iPrefix` can add to the path within the `iProc` and is the selected entity in the hierarchy tree is used as base for the following scan commands. This command can be used within the work-around as it just prepends paths to the scan commands. When converting scan commands and resolving the instrument the latest `iPrefix`-path can be added and the right instrument found. However, to be consistend with the desing of the work-around, `iPrefix` will not be used and a warning will be generated.

`iUseProcNameSpace` This annotation defines the namespace to be used for subsequent `iCall`s in an `iProc`. This is necessary as `iProc`s may share the same name in different namespaces. The namespace for an `iProc` can be defined as an option for the `iProcsForModule` command. This option avoids name-conficts when using instruments from different manufacturers who are unaware of eachother's `iProc` names.

`iNote` This command is used to provide feedback to the user of the system. It displays a String. This command is not suitable in an embedded setting for dependability management and will be skipped by the compiler. Off-course, in later revisions it may be coupled with a `printf` command that can have its own defined output steam.

`iClock` This command defines a system clock that exists somewhere in the network as a port. The clock needs to be in the ICL specification. This system clock needs to be based on an available clock, such as a TAP's regular clock signal. As the parsing of ICL remains future work, this commands will be skipped by the compiler.

`iClockOverride` This command is used to override characteristics of a system clock. The clock must be defined with the `iClock` command. This PDL function enables the user to change the clock's multiplier or divider compared to the source clock, or the source clock itself can be replaced by a different clock port. Again, it can be skipped to reach some state of a working compiler.

`iMerge` Merging of `iCall`s is possible with this command, effectively flattening the program. Sections are marked by this command with a begin and end argument. In

essence, the application of scan commands is delayed and a larger `iApply` group is formed. The standard allows for `iMerge` to be ignored, thus executing the `iCall`s subsequently. As this behaviour implemented by default in the compiler, the command will be ignored; possibly still generating a warning.

`iTake` Takes control of a network instrument, i.e. a target of a scan command. After taking control, one `iWrite` can be issued and applied before the instrument is unavailable. Storing the ownership of a resource can be done in the H-Array or the hierarchy tree structure.

`iRelease` Releases a network instrument that is taken by `iTake`.

`iState` The retargeting tool can use this statement to document the current state of the test network. The tool may output a list of `iState` commands in a textfile which can then be read by another tool. After reading the complete list, this second tool should have a complete overview of the configuration of the network. It is safe to assume that the compiler will not encounter such a command in PDL programs.

4.1.5 Converting PDL Commands

This part of the thesis describes the commands in PDL that operate directly on the network and are of most interest to the correct functioning of the framework and the Retargeting Engine.

`iProc` Almost all PDL commands exist within the definition of an `iProc` with the `iPDLLevel` and `iProcsForModule` being the only exceptions. An `iProc` is defined for an ICL module with the `iProcsForModule` command. However, the PDL grammar does not require this definition for any `iProc`. An `iProc` is a wrapper for multiple PDL commands and acts as a 'method' for instances of a module. These procedures can be executed by calling them with the `iCall` command. An `iProc` has a unique name within its namespace which can be defined with the `iProcsForModule` command. The method may feature parameters, which could have default values. The command structures PDL into methods available to the programmer to control the instrument network.

An entry point for the PDL program is not defined by the IJTAG standard and is left for the implementer of the retargeting tool. The entry point of the dependability

application will naturally be the *main* method. An entry point in the PDL is not required for this project as the framework will be compiled as a static library. It will be called upon by the dependability application, and the programmer decides where to start in the framework.

The structure of PDL programs provided by the `iProc`s will be copied to the framework. This makes the usage of the framework more natural for the end user, he or she can decide what framework method to call from the dependability application. This means a structured approach is needed to convert a `iProc` to a C function. This is pretty straight forward as Fig. 3.6 in Chapter 3 has shown. There are a few hurdles that need crossing; the first is that `iProc`s can be called on a module instance or the top level instance, the second is that an `iProc` may feature default values for its parameters.

This project uses a work-around for the first issue. All `iProc` will be defined for the top-level module. The second issue is solved by automatically filling the parameters in the C framework if they are omitted in the PDL. The compiler stores the `iProc` name and its parameter names and default values. These are then available to complete the `iCall` and for the function prototype in the framework header file. The `iProc` will become a C function with a void return type and as much int-typed parameters as it originally had.

`iCall` The relation between `iCall` and `iProc` is a familiar paradigm for any programmer. `iCall`s are used to invoke execution of a `iProc`. If the procedure is defined for a specific module, the `iCall` must provide a path to an instance of that module. In this case, no path is needed due to the namespacing-workaround.

The `iProc` could have parameters that need to be supplied when it is called. These parameters could have default values that are automatically supplied when not provided by the call. The `iCall` will be converted into a regular C function call. The compiler will provide missing default values where possible or will report an error otherwise. Figure 3.6 illustrates the conversion of `iCall` and `iProc` better.

`iRead` This command takes two parameters, an instrument and an expected value. The latter is used to compare with the result of the read operation. The number of *miscompares* is stored by the retargeting tool and can be accessed from PDL level-1. This is also the case for the read value. The `iRead` is stored within the `iApply`-group and executed when an `iApply` operation is encountered.

`iWrite` This command, like `iRead` also takes two parameters, an instrument and a value that needs to be written to the instrument. The value is, like all numbers in PDL, a unsigned integer. The last written value also needs to be stored. An `iWrite` is added to the `iApply`-group and issues when the operations are applied to the network.

`iApply` All operation that have accumulated are applied when an `iApply` is encountered. Commands between `iApply`s form a group. The order of executing the `iApply`-group is not predetermined and left for the retargeting tool. When multiple write commands are issued for an instrument the last one will be applied.

`iReset` The IJTAG client interface has an asynchronous reset port and this command uses that port. This resets the network, which also means its configuration changes. The retargeting engine needs to cope with that change.

`iScan` The `iScan` operation can be used to shift data in and out of a *black box* instrument in the network. Its parameters are the scan interface and the length; optionally the in- and output vector can be defined in the command. `iScan`s are also part of the `iApply` group. The problem with the `iScan` is that it defines the length of the access vector instead of relying on the TDR entry in the H-Array. `iWrite` and `iRead` target a register or port in the network while the `iScan` targets a Scan Interface, as seen in Table 2.1. There is no support for this in the H-Array.

`iOverrideScanInterface` This command can be used on scan interfaces which support modification of the update and capture behaviour. The module's ICL specification can define signals that can gate the control signals into the module. This is definitely out of the scope of the current PDL compiler.

`iRunLoop` This commands issues a certain number of clock cycles to the network without scanning data in or out. This can be used to wait on an instrument that needs to configure itself. `iRunLoop` uses the TAP's regular clock signal as reference by default but can also use a previously defined system clock. Support for this function in the hardware is not implemented, it can be added later but it is not a goal of this research. It is therefore also not present in the drivers.

4.2 Compiler Implementation

The compiler is based on the Antlr4 project as discussed in Chapter 3. Antlr4 generates a parser for an the input file according to a grammar file [22], [73]. The input is converted into a ParseTree which can be 'walked' to operate on the data. A regular TreeWalker as defined by Antlr4 will walk the tree in a certain order; first the child-nodes and then the node itself. Listeners that observe the TreeWalker will have user-defined enter and exit methods. These are called when a certain node is encountered. A visitor class can also walk the ParseTree but the user will have control over the order in which the children and node are visited. The visitor can also define the type that is returned visiting the children In general, they offer more control to the user.

It is also possible to override the basic TreeNode class with a user defined class. The TreeNode contains context for the current node. Overriding enables easy access to important user-defined objects such as the H-Array table and the `iProc` table. The class that is used to construct the Tree is called PDLParseRuleContext and it features access to the template engine, the H-Array and the encountered `iProc`s. A Class Diagram is shown in Fig. 4.4.

The compiler consist of a main class that reads from a file and instantiates the ParseTree along with a listener and a visitor, as can be seen in Fig. 4.3. The listener is implemented to perform semantic checks on the input. The visitor class extends the necessary methods in the base visitor class to generate the output of the compiler. They access the stored information, or context, in the PDLParseRuleContext tree node. To illustrate: the responsibilities of checkers are handling the annotations in Subsection 4.1.4 and the semantics of the commands, while the generator is responsible for the operations described in Subsection 4.1.5, although there is no strict border.

4.2.1 Checking the Input

Checkers are systems that can be employed to enforce the semantics of PDL. The setup is designed to walk the tree with a standard TreeWalker object which is observed by different Listener objects. A Checker is such a listener and its methods are called when a certain node is entered or exited [73]. Multiple listeners can be added in the same pass so each checker can have its own responsibility.

The compiler currently features one 'checker' instance to manage the declaration

and calling of iProcs. Next to checking, the system also provides the generator with necessary information such as the amount of parameters and default values of the procedure. This is stored in the IProcTable which keeps a list of IProcEntry objects. The table in turn is stored statically in the context tree nodes (PDLParseRuleContext). When an iCall is encountered, the checker makes sure that the target procedure exists, and adds any necessary default arguments to the call. The checker warns the user when a procedure is unknown or when the call is erroneous, i.e. wrong amount of arguments.

4.2.2 Generating the Output

The generator is responsible for generating the framework code based on the input program. After the ParseTree is walked and decorated by the checkers, the generator takes this annotated tree and uses the information stored in it to create the output [73]. The generator is an extension of the PDLVisitor class. A visitor is used to override the order in which the tree is traversed. The use of a visitor has as downside that all code must be in the same class. This may lead to a single object responsible for all the operations of the generator, but for now the code quality remains manageable.

The class consist of several visit-methods as can be seen in Fig. 4.3 which override methods in the base-class. For every ParseRule in the PDL grammar a visit<Rule_Name> method is automatically generated by Antlr4 in the Visitor base-class. The grammar has been altered to extract important information from the input stream among which the function names for example. This information is stored in the ParseTree context nodes along with the information provided by the checkers. Visit-functions gather this data and load a code-template with the template engine. The template is then filled in and returned at the end of the function. Visit-functions can visit the children of the node and in doing so receive the returned data of their visit-function. In this manner, the Pdl_source root node will receive the templates from its Iproc_def children, which in turn gets the rendered code from visiting any PDL command. This could be an Iwrite_def, which needs to parse the arguments that are given to it. It does so by visiting its children which are an instrument path and a PDL number. The instrument is searched in the H-Array based on the full path text (remember the namespacing workaround) and the PDL number is parsed and stored as an integer. This are then stored in the tree node, read by the Iwrite_def, put into a template and returned.

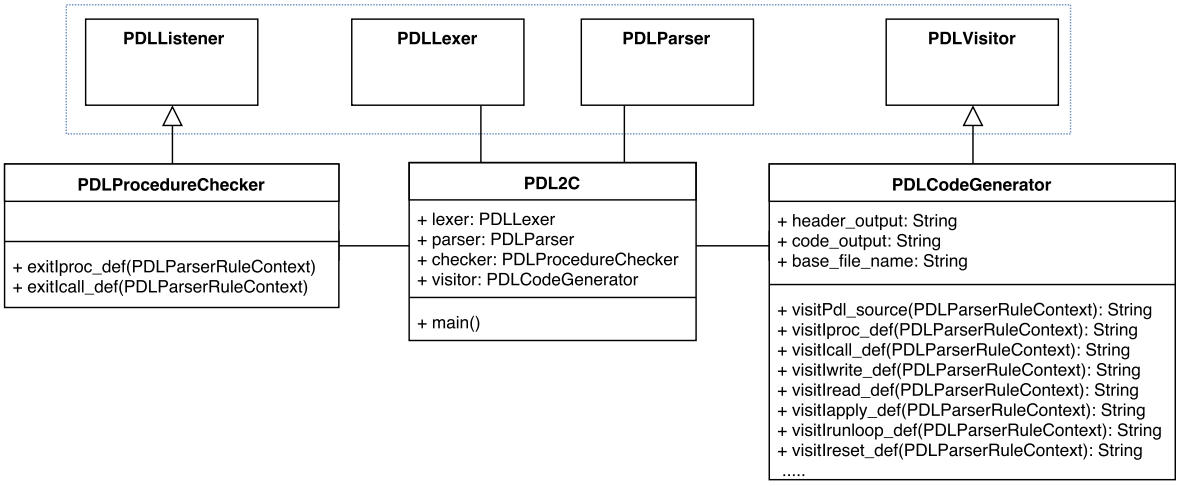


Figure 4.3: Antlr4 generated classes (in the blue arced box) are extended by a checker and generator classes.

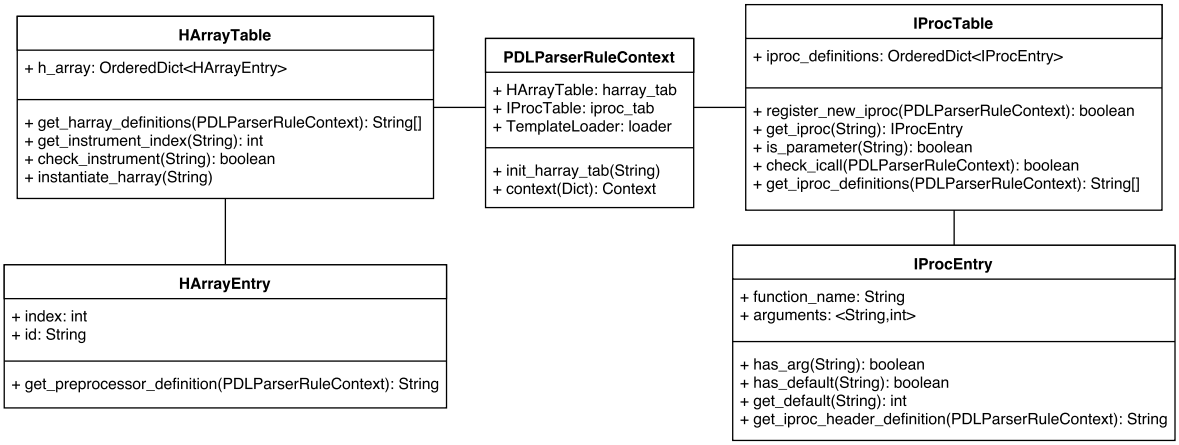


Figure 4.4: PDLParserRuleContext contains the instrument and procedure symbol tables and is the base class for all tree nodes.

4.2.3 Template Engine

The code generator class accesses the template engine via the PDLParseRuleContext tree nodes. Using a template engine for any code generation is a good idea to preserve syntax and layout of the output target. A template consists of a standard piece of code with place-holders that need to be 'filled' by the engine. This process is called *rendering* a template. A header and code template in C is made for every PDL command in the compiler. The generated code is returned through the tree and rendered in the `Pdl_source` root node to the output files.

4.2.4 Validation of the Compiler

As stated, this research uses the grammar that is part of the IJTAG standard [5]. This grammar is correct since it is part of the standard. Before proceeding, that assumption was validated by generating a parser with Antlr and feeding this some PDL files from the Bastion benchmark set [23], [27]. The generated syntax tree was analysed by hand for correctness.

The development and addition of listeners to the parser was also checked. The same benchmark PDL file was used for this purpose. The listeners store relevant information for the compiler in the context class. The listeners also determine if an instrument exists in the H-Array. This process was also checked by hand for the benchmark file. Errors that are detected by the listeners where also checked by removing instruments from the H-Array and removing `iProc`s from the file.

Using the template engine simplifies the validation of the output; every template can be individually tested. The templates were validated checking them syntactically with stubs as render context. These renders where checked by hand and compiled. The templates are connected to the generator class. The whole code generation process is then executed to see if the PDL was correctly translated to the framework. This framework includes the original PDL as comment for easy reference and manual testing. The framework is compiled and applied on the DM with success.

4.3 Drivers

The driver software form a Hardware Abstraction Layer between the DM and the generated software framework. This layer allows for reuse of both the hardware and software. It is a collection of software that is used by the dependability application and the framework to gain access to the retargeting engine and the interrupt

manager. The DM itself also has some drivers to operate the LEDs, discussed in subsection 5.5.6, from the application. Normally drivers exist between an operating system and hardware. They form a HAL between the machine and the OS. They would be compliant with the operating system and specific for the device. They register the device but also allow communication between the OS and the device. In this project the dependability application runs bare-metal on the DM and no operating system is used. The HAL and drivers are added to increase maintainability and reusability.

4.3.1 Operating the Retargeting Engine

The retargeting engine features four registers and a large addressable memory module. The memory contains the H-Array, Access Request Stack, the Returned Values and also the read extraction request stack, the latter being unimportant to the dependability application. The registers consist of a Control register and a Status register. The other registers are used to store the length of the H-Array and their functionality can be expanded if needed.

The Control and Status registers are used to communicate with the Retargeting Engine. The Status register is read only and has specific bit-fields mapped to a certain status of the retargeting engine. If the engine is busy with communicating to the network then a field reads a '1', if it is done, that field reads a zero. The Control register works in the same way but is only writeable. If a '1' is written to a certain field it will start the retargeting process.

The memory stores information necessary for retargeting such as the H-Array and the access request stack. The whole memory is addressable as this eases implementation. Before retargeting can happen the H-Array needs to be initialized. The driver library has functions that automatically encodes an entry based on given parameters. The instrument ids are available in the header file of the PDL framework as preprocessor definitions and can be used to encode the H-Array which is accessible as an array pointed to the right memory address.

The driver program has methods for issuing `iWrite`s and `iRead`s to the retargeting engine. They access the base of any Access Request stack and modify the pointer of that stack. They also automatically switch to the next stack to store the next Access Request.

After the `iRead` is applied to the network, the read values are also stored in the

memory. PDL level-0 has no specific way to give access to those values so a simple system is designed. The read values from the network are stored in an array with the same size as the H-Array. The id of the instrument in the H-Array determines the position of the value. The dependability application can read from the instrument by reading from its position in the read values array.

4.3.2 Operating the Interrupt Manager

The interrupt manager is also connected to the system bus and uses the same memory mapped IO strategy to communicate. Driver software is made to easily configure and operate the interrupt manager.

The interrupt manager features two accessible registers along with the interrupt vector table (IVT). The registers comprise of a Active and Busy register although the Busy register will not be very interesting from a software point of view. When an interrupt is being serviced, the Busy register will feature a one on the ESIB_L's position, based on its id in the IM H-Array. If one function handles multiple interrupt sources, it may identify the instrument through this.

The active register handles which interrupt vectors are valid and primed for execution. An interrupt service routine is attached to an instrument but may be deactivated for example when the normal program enters a critical point in the code. The driver contain a function to activate/deactivate a single IVT entry or all of them at once. An attached ISR must be activated before it can operate.

To register a function for a certain interrupt source, the driver has a function called `attachInterrupt`. It takes the ESIB_L-id in the IM H-Array and a handle of a user defined function. The handle is stored in the Interrupt Vector Table in the Interrupt Manager based on the entry pointed to in the IM H-Array. It is the user's responsibility to make sure that the interrupt function end with a `uret` command, displayed in Fig 3.3. This instruction notifies the hardware to return from the interrupt elevation, restore the program counter and registers and resume normal operation. The interrupt function may not feature parameters or a return type other than `void` as well.

The design of the registers of the interrupt manager is lacking in scalability. For example, the hierarchy array employed in the interrupt manager is not yet configurable in software, while it stores the connection between interrupt sources in the network and their corresponding IVT entry. The amount of interrupt sources will

probably be substantially larger than the 32-bit active/busy registers so their flagging strategy will not work. Resolving the IVT entry in hardware is easy due to the fact that the IM H-Array stores the pointer to the IVT.

4.3.3 Operating the Dependability Manager

The remainder of the Dependability Manager is managed by a set of supporting software. A library is made that contains a small self-test of the processor going through all the instructions of RISC-V. The supporting software also contain methods to drive the states of output LEDs, discussed in subsection 5.5.6. They are not useful for dependability management but very helpful to provide output during testing. The dependability manager library also offers a method to a success state and a failure state. These can be used to stop the program when the self-test fails. They both show a distinctive pattern on the LEDs to show the programmer what happened. To aid the creation of the H-Array in the retargeting engine some functions are made to encode entries. The driver for the DM is a collection of handy functions aiding the programmer.

4.4 Toolchain

The steps to compile a dependability application are shown in Fig. 4.5. The H-Array and PDL files are entered into the framework compiler which results into a collection of methods available to the dependability application. The application contains the entry point for the program and needs to include the header of the generated framework. The framework in turn includes the drivers and they are cross compiled and linked together. The object file created from the dependability application can then be converted to a hexadecimal format for loading into the simulation and synthesis tool. The RISC-V compiler is available online [60] and this project includes a simple script for automatically executing the steps.

4.5 Discussion

The design, implementation and verification of the PDL2C compiler has been discussed in the relevant subsection. Automatic testing of the Antlr created parser is difficult as it can only parse programs according to the specification. Antlr4 provides a manual testing tool [73]. Errors in parsing are hard to simulate and check. Unit testing of the compiler is not possible as creating token streams or ParseTrees is a

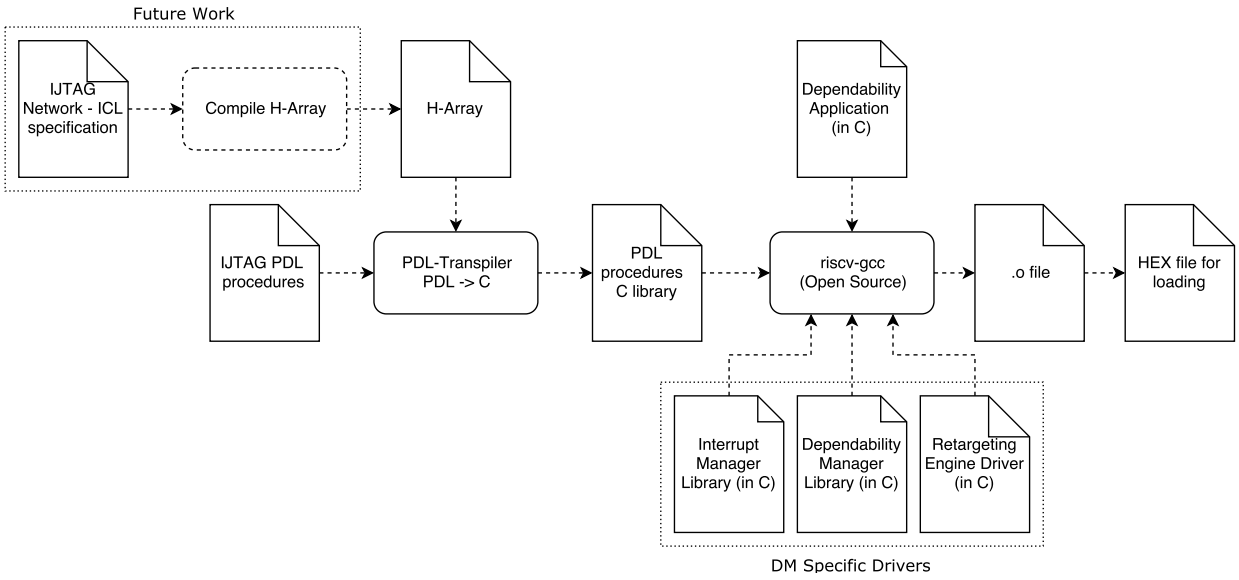


Figure 4.5: Compilation of a dependability application with drivers.

cumbersome and task without much to gain. Manually checking the input versus the output is not ideal or gives any guarantees but it suffices for this research.

The remainder of the chapter focusses on the driver implementation which are in itself quite simple. They need to write a certain bit to a certain address to operate the devices on the system bus, see section5.1.2. When the whole processor is simulated it can be checked to see if the addresses are correct and the driver behaves as it should.

A discussion must be held whether the compiler is now complete, good enough or lacking in features. The research set out to automatically create a library from a PDL file and it must be determined whether it was successful. This research has omitted the incorporation of PDL level-1 commands and the support for Tcl. The reasons for this are that the PDL level-1 commands are not suitable within an embedded context as most deliver some kind of text information to the retargeting tool. This information can be used along with the Tcl language to control the program flow. The most applicable command of the language subset is `iGetReadData` which extracts the last read or written value of to an instrument. Tcl incorporation has been done but at the cost of supporting original version of PDL [1].

It is unwise to start drawing lines of what is supported and what not per command as this would create confusion. It is better to state that this research supports PDL level-0 commands which is concise and clear as specification. This step also eliminates Tcl support. The need for program flow control and such elements is already handled by the C programming language. This will create problems for developers

using PDL level-1 files. It should be addressed in future research by combining this work (creating a C framework) and Zakiy's work [1] (supporting Tcl) without breaking the syntax requirements from the IJTAG standard.

Table 4.1: Realisation of PDL features in the compiler

Subject	**Function**	
Grammar	Parse and compile PDL level-0	★
	Parse and compile PDL level-1 and Tcl	-
	Parse H-Array and map instrument indexes	★
	Parse H-Array metadata to automatically generate initialisation code	-
	Parse and merge multiple files	-
Namespacing	Root-level namespacing of all instruments	★
	Dynamic namespacing based on Hierarchy Tree	-
Checking	Check that instrument exists in H-Array.	★
	Check that `iProc` is defined.	★
	Check parameters of `iProc` at `iCall`	★
	Check that `iProcsForModule` matches entity	-
`iProc`	Conversion to framework function	★
	Save `iProc` namespace	-
	Mapping to relevant modules in Hierarchy Tree	-
`iCall`	Calling the `iProc` framework function	★
	Parsing and passing the parameters	★
	Substitute default parameters of the `iProc`	★
	Apply the `iUseProcNameSpace` to calls	-
`iWrite`	Implemented in driver	★
	Call from framework to driver	★
	Identifying relevant H-Array instrument	★
	Parsing and passing the value as integer	★
`iRead`	Implemented in driver	★
	Call from framework to driver	★
	Identifying relevant H-Array instrument	★
	Parsing and passing the value as integer	★
`iScan`	Implemented in driver	-
	Call from framework to driver	-
`iApply`	Implemented in driver	★
	Call from framework to driver	★
`iRunLoop`	*Not implemented in driver / RE*	-
	Call from framework to driver	★
	Function stub added in driver	★
`iReset`	Call from framework to driver	-
	Function implemented in driver	★
	Implemented in Retargeting Engine	★

To answer whether the compiler is done, its goal is discussed per feature. Table 4.1 shows the features of the compiler that it could have and which are now realised. It shows features that are explored in this research, that should be explored in the future research and the elements that are starred are featured in the compiler. Some elements are not ready to be implemented as the H-Array specification falls behind and leaves out important information about the network. The Hierarchy Tree that shows which entities exist within other entities is among that

necessary information. Some elements of PDL can be easily ignored as they do not affect the Retargeting Engine currently.

Table 4.1 does not discuss the fate of the PDL commands that are not yet supported or are ignored by the compiler. `iClock` and `iClockOverride` are ignored as there is no base in the H-Array for these operations. The compiler is unaware of any clocks in the network other than the TAP clock. The `iNote` is ignored as there is no output on the DM which can support a String representation. `iMerge`, `iTake` and `iRelease` are removed in the same breath as the Hierarchy Tree is not available to store the state of the instrument, no other threads are on the DM to access them and the standard allows the merge of `iCall`s to be ignored. Lastly, the `iState` command allows the sharing of the network state among different retargeting tools which will not be the case for the DM.

The toolchain in Fig 4.5 shows the steps from PDL file to the executable. The DM specific drivers can be replaced by drivers for any other target and the PDL library would be still as useful. In future work the conversion from ICL to the necessary data types will need to be handled. The generation of the H-Array [26] from a graph representation misses relevant data such as segment length, although this should not be hard to add. Parsing ICL to produce the selection dependency graph may be a nice challenge for a group of graduate students. The hierarchy tree must also be generated so dynamic namespacing can be supported.

Chapter 5

Hardware: Creating the Dependability Manager

This chapter will discuss the realisation of the Dependability Manager. The design will consist of a simple processor for the RISC-V RV32I instruction set. Some instructions that are barely used or are difficult to realise will be omitted (`ebreak`, `ecall`, `csrread`). An instruction from the RISC-V N-extension is borrowed to support interrupts [42].

The processor features a controller and datapath structure with data registers and an Arithmetic Logic Unit (ALU) which are configured by a controlling instance. It is designed as a classic *Von Neumann* architecture where instructions and data exist in the same memory space. The Load/Store Unit, instructed by the controller, is responsible for loading data into the registers from the memory and vice versa. Therefore it is also the System Bus master and enables communication with the devices on the bus. The System Bus is extendible and this allows for easy adoption of multiple devices. The default configuration of the DM sports a retargeting engine and a interrupt manager. For testing purposes a LED driver instance is added to provide easy feedback to the developer.

5.1 Processor

The processor consists of a 'datapath' that houses the Registers, ALU and the Load-/Store Unit. The Registers and ALU perform al computations while the LSU is responsible for fetching and storing the data between memory and registers. Any device can be connected to the LSU with the system bus. The system bus is designed to use the Wishbone protocol [77]. The device will need to be assigned an address space and need to operate with the protocol. The LSU only supports Wishbone's

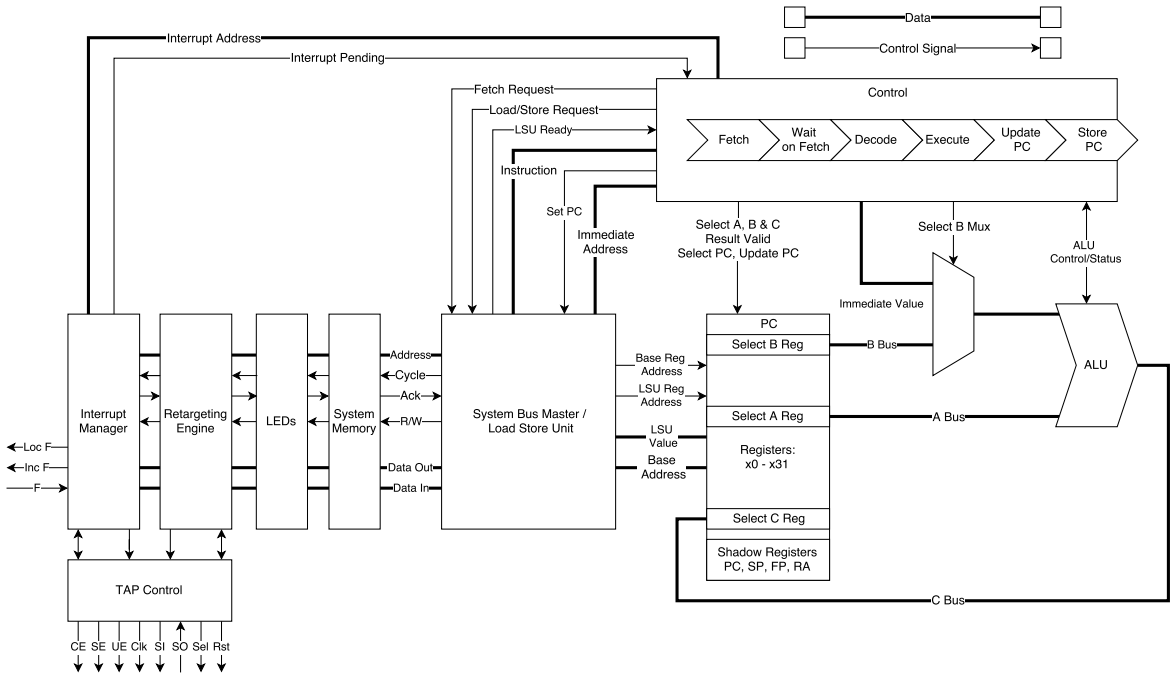


Figure 5.1: Detailed system block diagram.

Single Read and Single Write operations, which suffice in the context of this simple processor. A block diagram of all the systems components can be viewed in Fig. 5.1.

The processor starts executing any program at the start address stored in hardware. This start-address is configured in the compiler's linker-script and this file has been amended to begin at 0x0074. The program is stored in the dependability manager's main memory, which also serves as its random access memory. Although, if realised within a SoC, it will need to become non-volatile for program storage.

5.1.1 Controller

The controller initiates all the actions of the devices within the processor. The ALU is instructed what operation to perform and when, the LSU is requested to load and store data and the registers are signalled to store results or provide their contents. The controller takes the instruction that needs to be executed and decodes it to execute the program one step at the time. It is kept simplistic to lessen the implementation time, meaning no out-of-order execution or pipelining of instructions. The controller moves through the states described below to start and finish every instruction correctly. The PC is calculated with the ALU, which definitely slows the processor down. Accessing the instruction memory with the system bus is also a limitation to the design as it takes several cycles to retrieve a word from memory. Improvements to the architecture will be discussed at the end of this chapter. The

cycles per instructions are discussed in section 6.3.

Fetch - Issue the retrieval of the next instruction.
Wait On Fetch - Wait on LSU to provide next codeword and signal ready.
Decode - Decode the instruction and configure the datapath.
Execute - Result of the ALU is available and needs to be stored.
Write - Result stored, prepare the update of the PC and manage branching.
UpdatePC - Calculate the PC and signal that it can be stored.
StorePC - Store the PC and go to the Fetch state.

The controller is responsible for decoding the instruction that it receives after the fetch-state. The fetch of the next instruction is initiated by the controller and the LSU signals when it is available on the instruction bus. The RISC-V instructions supported by the processors can be seen in Fig 5.2. The instructions are listed in distinct columns which will be discussed below. They will be familiar to the reader due to the design of the ISA itself, it features the common operations among many noteworthy architectures.

The processor has 32 general purpose registers with the register at address zero tied to '0', which is quite common in processor design. These registers are accessible to the ALU which operates on two inputs to produce a 32-bit output. The operations it supports are in the leftmost two columns of Fig 5.2, namely 'Register' and 'Immediate'. The register operations encode two input and one output operation. The immediate operations replaces the second input register with an immediate value. For this, the second input of the ALU is multiplexed with the immediate bus coming from the controller. This bus is also used by other instructions, most notably LUI and AUIPC.

Unlike textbook instruction sets (such as MIPS I), RISC-V's memory operations combine a base address register and an immediate offset to specify a memory location. The ISA also allows for direct manipulation of the Program Counter (PC) through `AUIPC`. This needed to be accommodated in the design, the PC is added to the registers to be directly accessible by the ALU. This was done instead of a separate PC unit, in this way the datapath remains uniform. The LSU and registers are also directly interfaced with the registers having 4 output buses (base address, store-value, a-bus and b-bus) and two input buses (load-value and c-bus). The load and store operations are byte-addressed and specify the word length. It also distinguishes between loading unsigned and signed bytes or half-words. The LSU is responsible for sign-extending the values when dictated by the memory operation.

Since loading and store does not use the ALU, it is used freely to update the PC while the processor waits for the memory operation to finish.

Among the control flow operations in the base instruction set are conditional branches and jump statements. The branch operations load two register values into the ALU to compare (signed or unsigned) and the resulting ALU status notifies the controller to jump or not. The regular jump-and-link instructions always updates the PC to the specified address and stores the next PC value in a register. The JALR operation does the same but computes the jump address based on a base register and an immediate value.

Two other operations introduce immediate values into the datapath. Load Upper Immediate (`LUI`) uses the similarly named bus from the controller to store a value in a register via the ALU. The Add Unsigned Integer and PC `AUIPC` instruction can be used to create relative addresses based on the Program Counter, it adds an immediate value to the PC and stores it in a register.

The arced column 'System' is not (yet) implemented in the controller. The reasoning behind this is that the Control and Status Register (`CSRxxx`) instructions are needed to address the processors internal registers, which the base RV32I set only has 3 of, and the RV32e does not require. Note that the latter is not yet supported by gcc and is designed to be used in embedded systems. The Fence (`FENCE`) instructions make sure memory operations are performed in order, while our processor will always execute them in order. The External Call (`ECALL`) and External Break (`EBREAK`) instructions are not clearly specified as they operate with an, as of yet, unknown Application Binary Interface[1]. This interface allows two binary programs to call to each other. This can be used to call operations in the operating systems (such as starting threads, getting the system time or requesting memory space). The RISC-V specification does not elaborate on this subject [42]. Only when a program ends (main function returns) an `ECALL` is produced and this halts our implementation of the processor.

The last instruction to discus, which is also arced, comes from the RV32n extension and is the User Interrupt Return `URET` instruction. This is used to signal that the controller must return from an user-level interrupt function. It restores the stack pointer and frame pointer from before the interrupt. A stack pointer is a common feature used in memory addressing. The stack pointer may change within a function as more variables need to be stored. The frame pointer helps as it remains stable

[1] https://en.wikipedia.org/wiki/Application_binary_interface

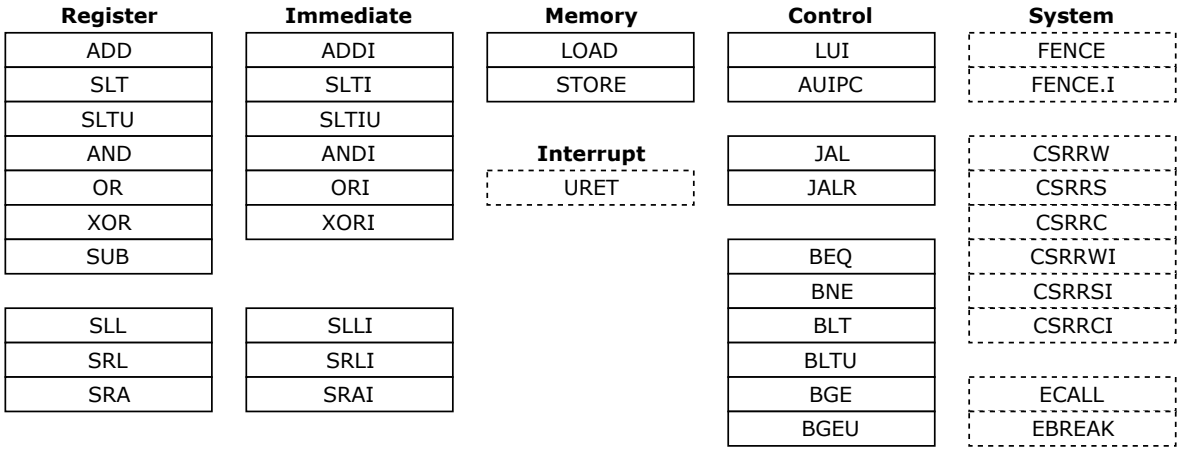


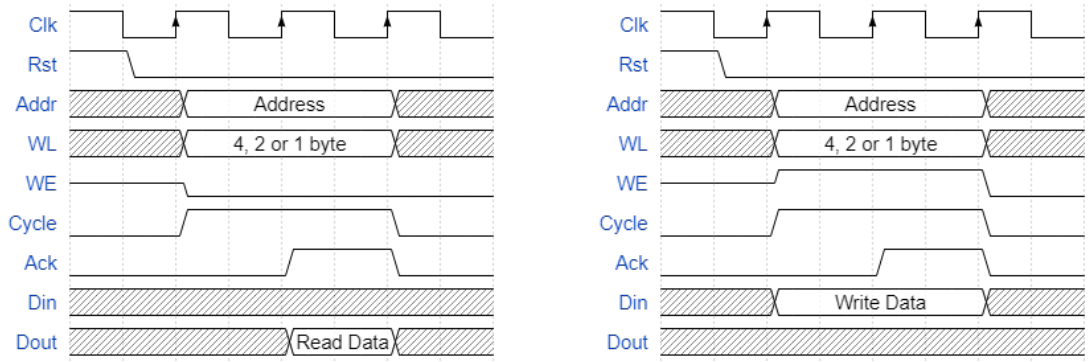
Figure 5.2: The RISC-V RV32I set and the RISC-V N extension for interrupts.

in the function. This makes addressing in assembly easier, it is just as important as the stack pointer. Returning from an interrupt also restores the program counter from the shadow registers and lets the normal program flow from the point where the interrupt occurred. It is imperative that an ISR function returns with the `URET` statement shown in Fig 3.3 for the program to behave correctly.

5.1.2 System Bus

The system bus is implemented to allow simple memory access, no burst mode or caching of data. The Wishbone protocol specifies Single Read and Single Write operations, the signal timing diagrams for these are shown in Fig. 5.3. It is basically a hand-shake protocol where a client device needs to acknowledge the write or read operation. Devices are daisy-chained along the bus and they operate a 'relay or talk' protocol based on the current address on the bus. If it is within their address space they will need to answer to the master, otherwise they will relay the information along the line. The master has complete control of the system bus signals but receives requests from the controller to operate on the bus.

The processor architecture combines all the system components implemented in VHDL. The memory, the LED driver, the retargeting engine and the interrupt manager are instantiated and linked together. The address space of each component is given at their instantiation. For example, the memory slave device is instantiated and connected at the 'upstream' side to the signals coming from the master. The device also has a 'downstream' bus that connects to the next device (LED driver). The address space of the system bus devices is shown in Table 5.1.



(a) Read *(b)* Write

Figure 5.3: Single Cycle Read and Write operations according to the Wishbone B4 specification [77].

Table 5.1: Address space within the DM.

System Bus Device	Start	End
Main Memory	0x0000_0000	0x0001_0000
LED driver	0x000A_0000	0x000A_0004
Retargeting Engine	0x000B_0000	0x000B_2000
Interrupt Manager	0x000C_0000	0x000C_0100

The main memory slave entity houses the main memory which is filled with program data for simulation and FPGA emulation. The memory uses a generated volatile SD-RAM IP from Altera to employ the dedicated memory cells in the used FPGA (Cyclone IV). This yields a large program space for simulation and emulation, the same IP is also employed for storage within the Retargeting Engine only smaller. The DM is configured with 32KB of storage space for programs, comparable to the flash memory of an Arduino Uno. However, this space is also used as Random Access Memory (RAM). This memory system will need to be replaced when the dependability manager is realised in silicon as part of a SoC.

5.2 Retargeting Engine

The retargeting engine is responsible for the configuration of the IJTAG test network. The RE's 'execution model' [2], [16], its method of operation, is implemented in VHDL. Previous work by Zakiy featured a shell to incorporate the retargeting engine and devised a method to communicate with and control it [1]. This research's implementation has been added within this shell to validate Zakiy's work. However, due to reasons described in Chapter 3, Zakiy's DM implementation has been dropped. The retargeting engine was added as a system bus device to the processor. Implementation of the retargeting engine is focussed on the realisation on

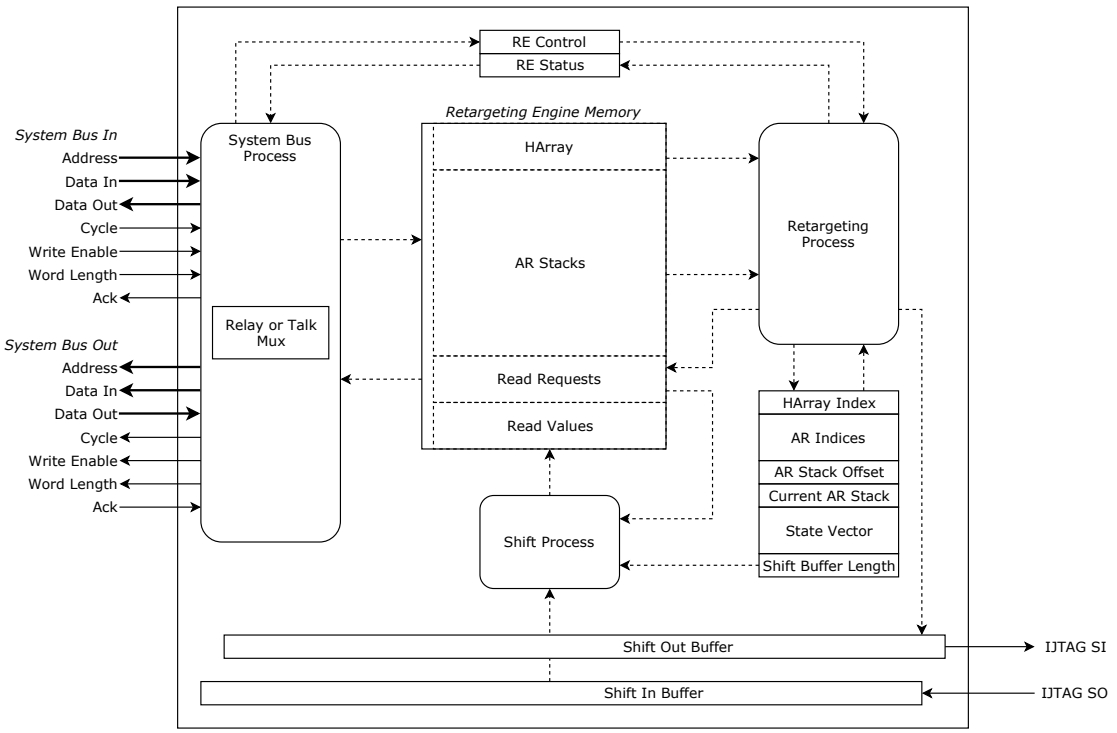


Figure 5.4: Detailed block diagram of the Retargeting Engine.

an Altera FPGA. It's design is modular, see Fig5.4, to allow replacement of certain components. This will ease the work for realisation on another FPGA or in Silicon. This section will describe the construction of the retargeting engine IP.

A block diagram of the RE IP is shown in Fig. 5.4. It shows the registers, the memory and the shift buffers. The processes in the architecture are communication according to the system bus protocol, the retargeting process that traverses the H-Array and generates the access vector (AV), and the shift process that shifts the access vector into the test network.

The System Bus communication process is used to respond to the master when requested to. MMIO allows a program to write a value directly to a register or memory cell within the retargeting engine. The control and status registers are used by the retargeting engine driver software to initiate retargeting and communication with the network. The retargeting engine memory entity is also mapped and is used to store access requests and to load read values from the network. It is a generated Altera SD-RAM IP similar to the main memory, it has an extra access port compared to main memory to allow simultaneous communication for the System Bus process and the Retargeting process. Using this memory IP takes advantage of the features of the FPGA. However, it will need to be replaced by a different memory system when the DM is realised. A map of the memory and its usage is shown in Fig.B.1 in

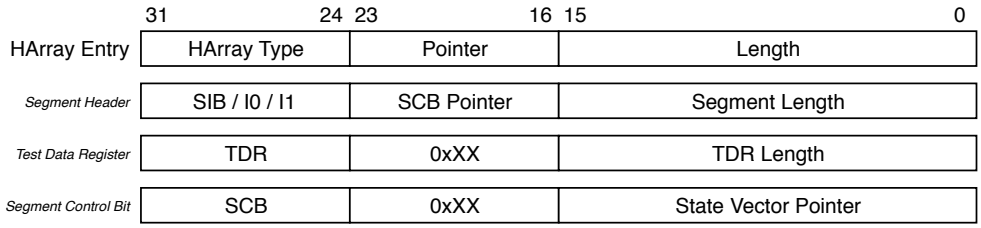
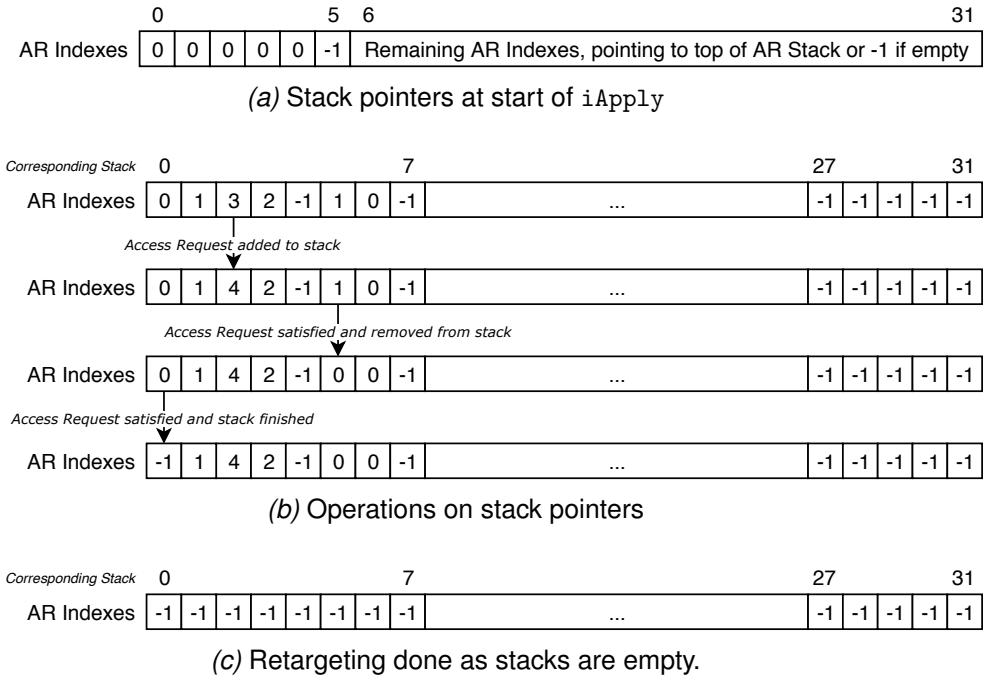


Figure 5.5: Encoding of H-Array entries.

Appendix B.

The Retargeting process uses the H-Array stored in the memory of the RE. Note that this is a different memory device than the previously described main memory. The H-Array contains references to all the test network structures and test data registers. An encoding is made for the entries in the H-Array. The most significant byte in a H-Array entry is reserved for the entry type, either SIB, I0, I1, SCB or TDR. It is trivial that these types can be encoded with 3 bits, so using 8 is a waste. This encoding has a practical nature to allow simple debugging in simulation and driver creation. An argument can be made that an I1 and SIB operate similarly (a SIB is a ScanMux with an null-segment) and that an encoding of 2 bits should suffice. The encoding of the entries is shown in Fig. 5.5. The segment header entries (SIB, I0, I1) store a pointer to the segment control bit entry and the length of the segment. The SCB stores a pointer to the State Vector, a register that stores the last written value of a SIB or SCB. This is the network's state. A TDR stores its length, necessary to correctly generate the access vector.

The next piece of crucial information for retargeting is the Access Request Stack (AR-Stack). This structure contains the original Access Request at the base of the stack, position zero. The stacks can be observed in Fig B.1 in Appendix B in the memory of the retargeting engine. The size and amount of stacks is adjustable with parameters which is why not the full amount of memory is used by the retargeting engine. When a segment header in the H-Array is encountered the state of the segment is retrieved from the State Vector. The State Vector is implemented as a register with the same length as the maximum H-Array size. Then it is determined if there is an Access Request at the top of any stack that needs to access this segment. This can be determined easily for every Access Request by checking if the instrument ID of the request is between the current H-Array index and the end of the segment. If so, a request is added to the top of the stack with the relevant SCB as its target.



(a) Stack pointers at start of `iApply`

(b) Operations on stack pointers

(c) Retargeting done as stacks are empty.

Figure 5.6: Stack Pointers used to keep track of Access Request Stacks.

If an SCB or TDR is encountered during the traversal of the H-Array it means that it is in the active scan path. Again, all the tops of the stacks are checked to see if they contain that instrument, i.e. the current H-Array index. If so, the value in the access request is shifted into the shift buffer in case of a write, and otherwise a read request is added to the memory to later extract data from the incoming bit stream.

Stack pointers keep track of the Access Request stacks. These are vital when passing through all the tops of the stacks. A '-1' shows that a stack is empty, otherwise the pointer points to the top of the respective stack. Fig. 5.6 showcases the operations on the stack pointers to influence the actual Access Request stacks. The retargeting process repeats itself until all stack pointers are showing '-1'. A stack pointer is an integer ranging from -1 to `max_stack_size`, which default is configured as 7. This configuration requires 4 bits per stack.

The state vector is accessed during the traverse and generate process. The state vector is displayed in Fig 5.7. This process creates the next access vector by going through the H-Array which is guided going in or skipping the segment based on their header entries. The state vector keeps track of the last written values to the SCBs, which control a header. Those H-Array entries keep a pointer to a place in the state vector. During the traverse and generate function new values will be shifted to the SCBs which are stored in the New State Vector. When a CSU cycle occurs, the New State Vector is copied to the current State Vector. Old, unchanged values

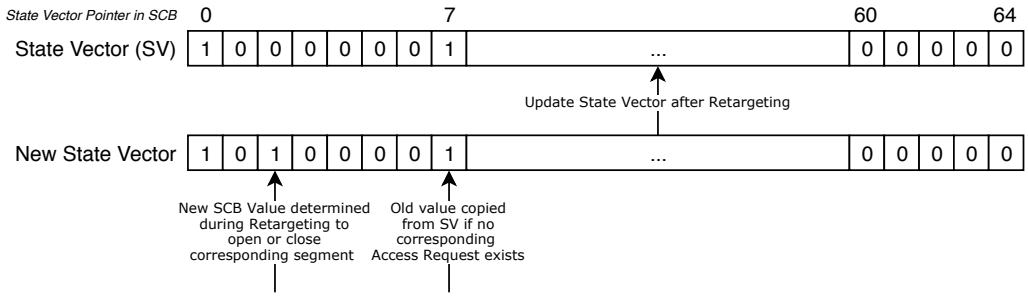


Figure 5.7: State Vector, the current state is updated when the new values are shifted into the network.

are preserved. This bit-vector is then used for the next traversal of the H-Array.

To better describe the flow of the algorithm that accesses the network to read and write data to instruments a flow chart is created in Fig. 5.8. From the Idle state the RE receives a signal that in needs to start retargeting. The H-Array is then traversed from the first element until the last based on the H-Array length register. It loads an entry from the network representation from the memory and checks its type. In all cases it will start loading the relevant tops of the stacks from memory based on the stack pointers. In the Process AR state it may store a new one at the top of a stack or pop one of the stack if it is performed on the network, see Fig. 5.6. If the algorithm reaches the end of the H-Array it will signal to the shift process to start interaction with the network. After shifting the values it will loop through any Read Requests present in the memory. These are then extracted and the values stored at the right memory place based on the index of the instrument in the H-Array. If the stacks are not empty at that point, the retargeting engines starts a new iteration of the process until all access requests are satisfied.

Fig. 5.9 shows a nice feature of the retargeting engine concerning the traverse and generate process. The structure and ordering of the H-Array allows simplified generation of the outgoing shift buffer. Note that in the figure, the shift out buffer will be send to the SI TAP pin and shift in buffer is filled with data from the SO tap pin. Since the last element of the network needs to be filled by the bit shifted into the network at the first clock cycle, the buffer can be created by going through the H-Array and shifting the buffer to the left. The final element of the example H-Array will take the rightmost place in the buffer. This place is shifted at the first clock cycle into the network if the buffer is shifted out to the right.

The read requests are a different kind of request. They are generated by the

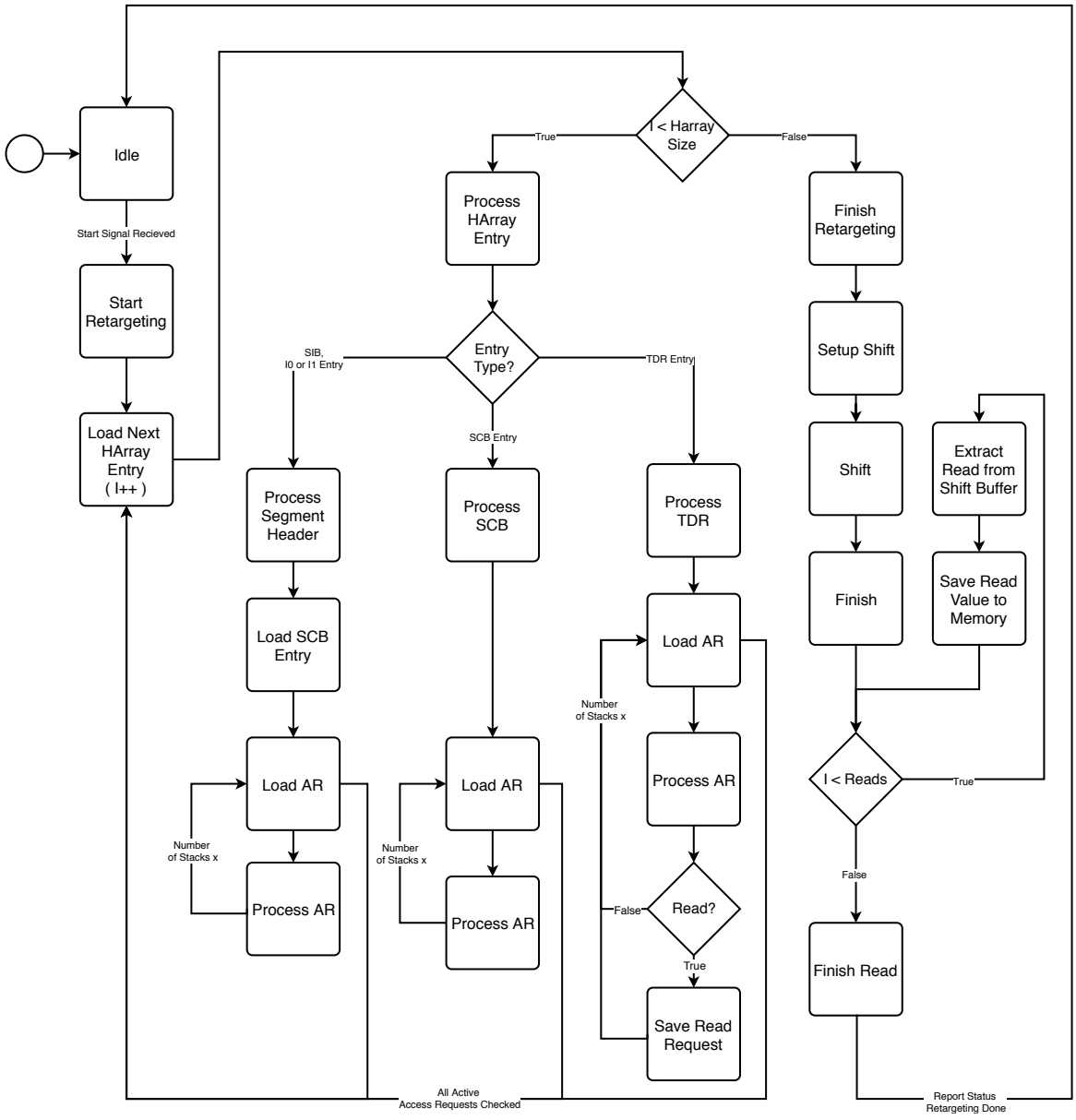


Figure 5.8: Flow Chart describing the steps taken by the retargeting engine for generating an Access Vector, Shifting it into the network and extracting the Read Values.

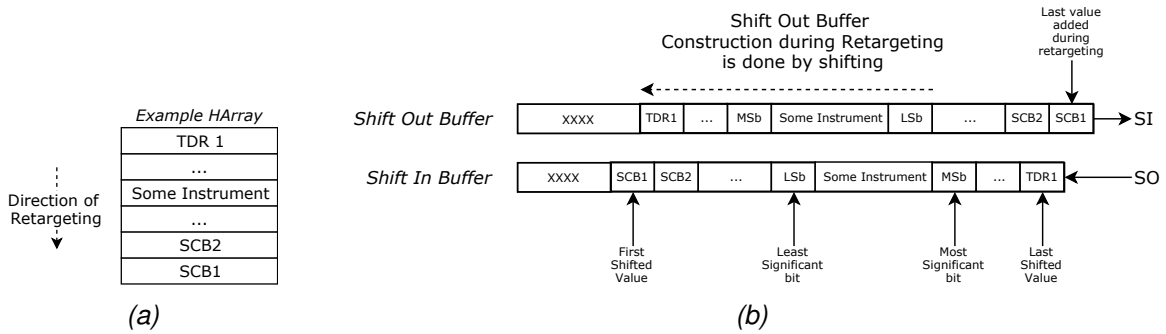


Figure 5.9: Traverse and Generate the Shift Out Buffer, the order in which values arrive affect the extraction of Read Values.

retargeting engine to extract a word from the incoming bit stream. The shift in buffer is filled with the last element of the network which arrives at the first clock cycle. By shifting to the left every clock cycle the received data from an TDR is in reversed order. This is solved during the handling of the read requests. The read request stores the place where a TDR is in the incoming bit stream. The length of the TDR is also stored in the read request. As can be seen in Fig.5.9, the order in which the values arrive matters. The value is extracted and put in the correct order in the read values memory space. The encoding of a read request can be seen in Fig. B.1 in Appendix B.

This section described the implementation of the retargeting engine. Using the memory and state machine led to a very rigid design that can access large IJTAG networks. The retargeting engine can be configured with parameters, so testing a different design was simple, see section 6.4.

5.3 Interrupt Manager

This section is not a research into interrupt managers but rather an elaboration of the implementation of the IJTAG extension for interrupt enabled instruments along with a subsystem of the DM to locate and issue interrupts for the processor. This extension for IJTAG is designed by Ibrahim [3] and realised and tested in this work. However, there are other methods in research to enable interrupts in IJTAG networks which are not covered in this work.

5.3.1 Locating Interrupts

Specialized test networks are needed to locate interrupt sources to avoid polling-based localisation. Polling would be an inefficient strategy as it would occupy network resources and takes large amounts of access time depending on the network

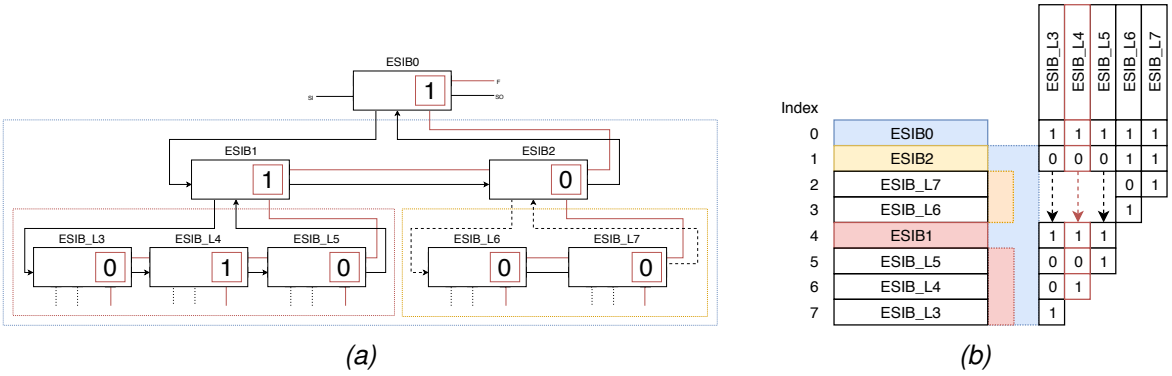


Figure 5.10: Simple Interrupt Network along with its IM H-Array. The columns show the scan vectors when an interrupt originates from the leaf ESIB.

topology. The work by Ibrahim designed specialized Extended SIBs and validated a purely hierarchical tree layout of the network to provide optimal access time [3]. The design of the ESIB is explained in subsection 3.3.3, and is shown in Fig. 3.11. In short, it is a SIB that contains one or more flag registers and has three modes of operation; as a normal SIB, in diagnostic mode and in localisation mode. The latter includes the flag registers and opens the underlying segment according to the flag value. This research implemented the ESIB based on the network structures provided by the Bastion benchmark set [23].

The ESIB were added to a network based on the design in Fig. 3.10. This design is not part of the Bastion set [23] and will be referred to as the Simple Interrupt Network or Simple network. During development it was found that leaf nodes in the network had a difficult position. During normal operation all the network structures are activated by the Select signal of the TAP. The network structures in the work of Ibrahim [3] rely on an inactive Select signal to function. Although not present in the Simple Network, unaltered network structures would prevent the extended interrupt network from functioning. This created the situation where normal TDRs would ruin the efforts of the extended SIB by not shifting the bits through. Later, it was discovered that a purely hierarchical network implied that only ESIBs would be accessed. The idea of leaf nodes stuck due to the fact that they do not open their segment when a flag is active. This mitigated the problem that unselected/inactive TDRs introduced.

An encoding was made for the entries of the IM H-Array in the interrupt management unit. The encoding is shown in Fig. 5.11. The network data structure used for localising interrupts contains ESIBs, ESIB_L and TDR. Compared to the previous work of the interrupt management unit, the ESIB_L takes the place of the TDR

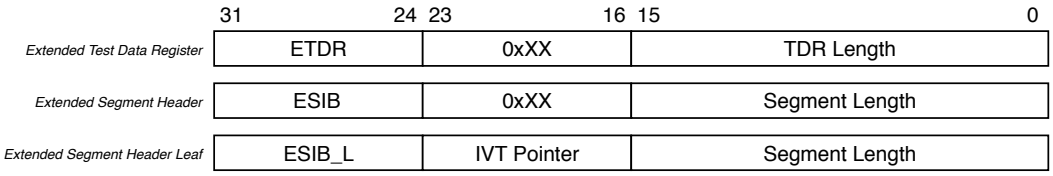


Figure 5.11: Encoding of IM H-Array entries.

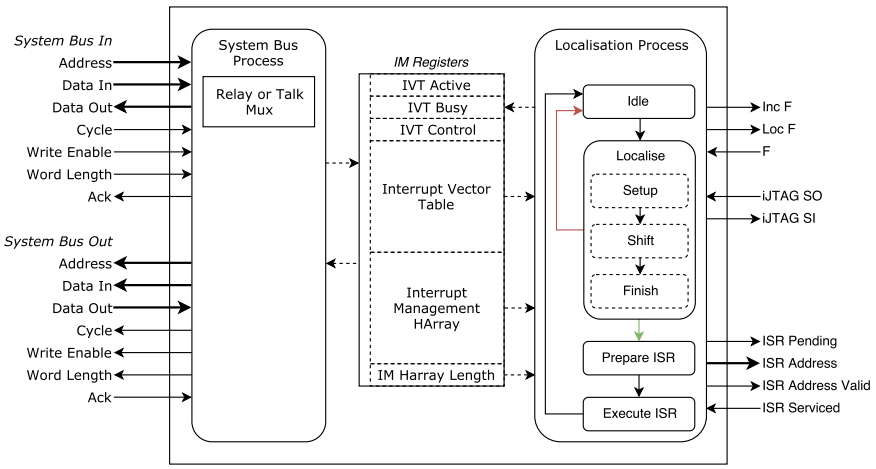


Figure 5.12: Detailed block diagram of the Interrupt Manager.

as the work states that when a TDR is encountered the interrupt is localised. The localisation is done by moving the network to localisation mode using the extending signals for interrupt management. When the whole network is in that mode it stores the interrupt flag it receives from its children in a register which is included in the scan path. In fact, it is the only included register. The interrupt manager then moves to shift out the flags in these registers. Fig.5.10 shows such a network and the values of the registers of the ESIBs. The figure also shows the relation between the received vector and the instrument that created the interrupt. If a segment header reads a '0' the segment is skipped as the interrupt did not originate there. If a ESIB leaf instance read a '1' then it has created the interrupt, otherwise its brother on the same level or an segment below. The interrupt manager for IJTAG networks is shown in Fig.5.12. It features registers for controlling the device and a IM H-Array and an Interrupt Vector Table. The first is used during localisation and the latter stores the function pointers of the interrupt service routines.

5.3.2 Servicing Interrupts

The art of servicing interrupts is well researched and many excellent schemes exist to manage multiple source levels, pre-emptive schedulers and masking interrupt vectors to quickly get from the signal to execution of the intended interrupt service

routine. The strategy of this research is simplistic in nature as implementing new ways to manage interrupts is not part of the scope. However, due to the state of development tools for the RISC-V architecture, a smart solution is needed to incorporate interrupt service routines without full compiler support.

This limited support means that an interrupt service routine is defined as a regular function in C without return type and parameters. The compiler takes into account that a function needs room on the stack to execute. The stack-pointer, frame-pointer, return address, program counter, thread pointer and global pointer are used by the compiler to manage the program flow. Shortly after a function is called the stack-pointer is amended to reserve space for any return value and the local variables. This is fine when calling a function as it creates the stack space and removes the space before returning. This also means that many nested functions can be called, at least until the stack limit is reached. However, the processor must be notified when an ISR returns. Using the regular return instruction (Jump And Link using the return address general purpose register), would break because that return register would not be filled by entering an ISR. It would also prevent nested function calls in the ISR.

To mitigate the issue the `URET` instruction is borrowed from the RV32n extension. This instruction notifies the processor that the program returns from an interrupt. The processor is designed with special shadow-registers for the management-registers (Stack Pointer, Frame Pointer, Program Counter etc.) that are used by the compiler. When an ISR is entered their values are copied into the shadow registers. An ISR finishes with the `URET` instruction. The `URET` signals that the original program flow must be restored and all values are copied from the shadow into the actual registers. For this to go well, an interrupt service routine can only be initiated in the *fetch* stage of the controller. Otherwise an instruction runs the risk of being executed twice. Although this solution works it is a shame that it is not very expendable. When an interrupt is being serviced, no other interrupt (with possibly a higher priority) can be serviced. When the RISC-V compiler will support the interrupt extension better solutions can be found, for example storing the management-register-values on the stack space. This will enable pre-emptive interrupt servicing.

5.4 TAP Control

The IJTAG interrupt manager and the retargeting engine operate on the same test network simultaneously. Since IJTAG networks follow the master-slave principle in hardware engineering, it is a good practice to have a dedicated master for the net-

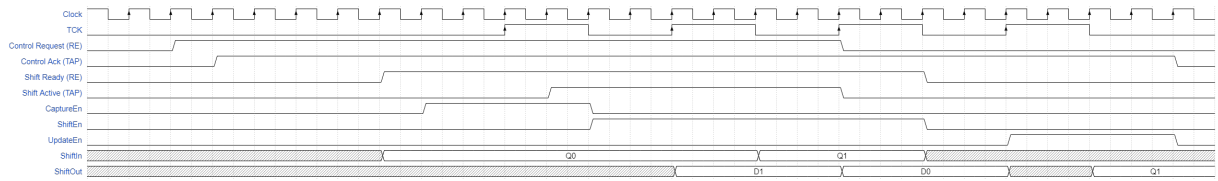


Figure 5.13: Timing diagram of TAP negotiation.

work. It cannot be that both the IMU and the RE try to control the network at the same time. It is also not allowed to digress from the Test Access Port state diagram shown in 2.2. When a CSU cycle is started it must be finished. It is also necessary that the network state, maintained by the RE, is accurate. When a CSU is aborted while the RE believes it has been applied, an incorrect network state is the result. This needs to be managed, the simplest approach is finishing every CSU that is started. The design of the ESIB allows switching between mode A and mode C, see Fig. **??**. Managing this will remain out of the scope of this research. Finally, it is also important that the shifting of bits happens synchronously across all the devices, meaning that the first bit read is also intended as the first bit from the network.

An extra device is added to the design of the DM that has complete control of the Test Access Port. The TAP Control entity in the dependability manager receives a request from the RE or the IMU that they want to shift data through the scan network. A handshake protocol acknowledges the request and the TAP Control moves to the Capture state. At the same time the requesting devices gets a signal that the Shift will happen starting at the next rising edge of the clock. The SI and SO signals from the requesting device is connected to the SI and SO of the Test Access Port. The end of the access vector is also signalled to the TAP control by the requesting device. This lets the controller move to its next internal state and issue the Update state accordingly.

This handshake protocol is visible in Fig 5.13. The timing diagram shows the system clock and the TCK generated based on the system clock. The TAP Controller contains a clock divider to manage the clock at which the IJTAG network operates. As it is not entirely clear what the speed criteria will be for IJTAG, this divider allows the engineer to change this. A device that wants to access the TAP will request this from the controller. After which an acknowlegdement will be sent. When the requesting device is ready to shift, this is signalled to the controller. The controller will move through the TAP FSM, see Fig.2.2, and signal when the 'SHIFT-DR' state is reached. The TAP controller and device will be synchronised based on the TCK. When the device is finished with the IJTAG network, it will deactivate its request.

The TAP Control maintains a priority between the devices. It can also be extended to manage the external IJTAG signals hinted at in the high level design (Fig 1.1 and Fig. 3.7). However, this creates problems as the RE and the external retargeting tool both store the network state. This can be solved by giving external access to the state vector. The state vector can be part of the IJTAG addressable registers within the dependability manager. A protocol can then be defined that states the length of the access vector when a switch occurs and what the bits in the state vector mean. The same protocol opens the underlying scan network previously controlled by the RE. At the end the external tool can alter the state vector based on its alterations and relinquish control back to the DM. Another approach is to reset the scan network when control is handed over. This will revert the network state to its original setting but this may also affect instruments and their TDRs within the network.

5.5 Validation of the Dependability Manager

The validation section of the Software chapter focussed on the correct implementation of the compiler, the drivers and the operation of the toolchain. Most of this is checked by hand along with tools such as `objdump`. This section will validate the last section of the dependability manager; the hardware. Test benches aid in development of several components of the dependability manager as they provide a meaningful . Not all entities were subjected to a unit test. These simple entities would take too much time and effort and this section would become long-winded and boring. Four major systems of the dependability manager were The test benches provided necessary input for the unit-under-test, but the output was checked manually.

5.5.1 Controller Test Bench

The controller was constructed as one of the first parts of the processor. It is responsible for steering all the components and a correct implementation would be a solid base for the remainder of the work on the datapath. The test bench contains the unit-under-test and a clock mechanism. When the controller requests the fetch of a new instruction, the next one is loaded on the instruction bus. The test bench contains one instruction of each column in Fig 5.2 which covers most operations of the controller. To clarify, loading two registers for an addition is handled in the same manner as loading two registers for a subtraction, only the ALU operation code differs. The waveform of the test can be found in Fig. 5.14. After debugging, the test

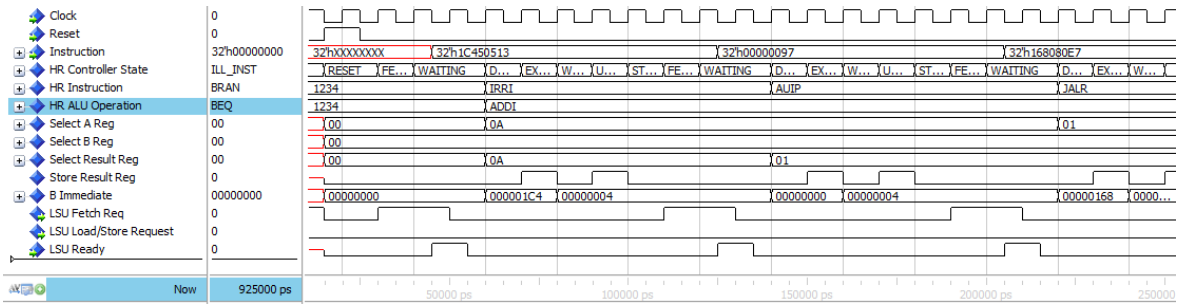


Figure 5.14: Controller Test Bench.

bench showed that the controller moved through all its designed states and outputs the right control signals for the datapath.

5.5.2 System Bus Test Bench

To develop and check the Wishbone bus implementation, a test bench was created that housed the System Bus Master entity along with a System Bus Slave entities. The slave devices had a memory range assigned two them and the master was requested to read and write to those addresses by the test bench. It was validated with simulation of the test bench that the Single Read and Single Write operations of the Wishbone protocol operated correctly. The waveform can be found in Fig. 5.15. The memory request is accompanied by an immediate address and a offset register value. The system bus master calculates the resulting address and operates the Wishbone Bus. The test bench switches between reading and writing so both operations can be checked. It can be seen in the waveform that the *Wishbone Cycle* is high for 3 cycles instead of 2 (see Fig 5.3). The system bus is created to be flexible and this is incorporated in the slave device. It delays its response by a clock cycle and this behaviour was also needed for the system bus devices with the Altera Memory IP. These devices took extra time to interface with the memory which takes at least two clock cycles. The system bus slave process is later incorporated into the Retargeting Engine, Interrupt Management unit and the other two System Bus devices.

5.5.3 Retargeting Engine Test Bench

The retargeting engine was in development as a separate component and had an accompanying test bench. The test bench created a clock but nothing further and had the Retargeting Engine as a unit-under-test. Its correct operation was verified with ModelSim while the initial Access Request stack, H-Array and relevant param-

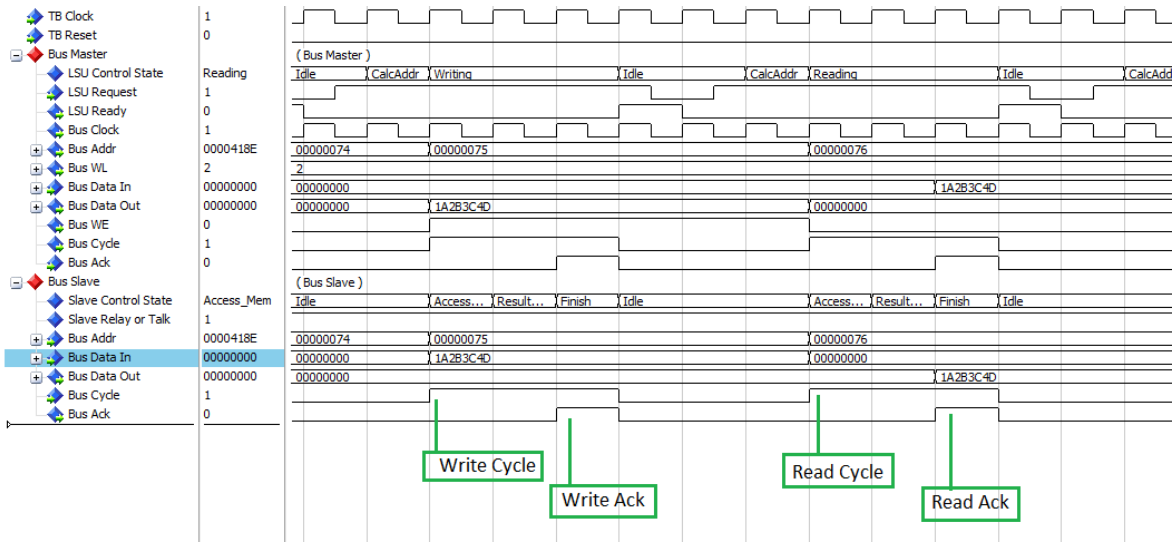


Figure 5.15: System Bus Test Bench.

eters hard-coded into the architecture. The retargeting engine relies on the H-Array VHDL-package, which contains functions to encode and decode the data-types and structures. This package also stores different test network H-Arrays, among which is the minimal test network used in the paper of Ibrahim [2]. The stand-alone test bench can be viewed in Fig. D.1. Notice that *UpdateEn* signal from the TAP has a small error which was resolved. The access request stacks can also be observed in the waveform.

The test bench was used during debugging until the retargeting engine operated correctly. ModelSim allows access to all signals in a VHDL design. The generation of access vectors was checked along with the amending of the state vector, managing the AR-Stacks and the stack pointers and filling and shifting of the shift buffer. A test vector was shifted as an input into the device to test the extraction of data from the shift buffer. The retargeting engine has also been added to the coprocessor-wrapper in the design of Zakiy [1]. The access request buffer that the wrapper has was connected to the retargeting engine. The scan commands from the program are loaded into the retargeting engine one by one. The test program was altered to address actual TDRs in the network. The network was the small example network from Ibrahim [2], and the scan vectors were checked manually. The network was hard-coded into the VHDL design, but the access request and scan command are loaded dynamically. The program ran successfully as can be seen in Fig D.2.

The final version of the retargeting engine (with memory and TAP handshake) was tested via a full system test, described later on in section6.1. This was easier than creating a test bench that interfaced with the retargeting engine via the system

bus. Since the system bus and processor were reliable, the retargeting engine could be fed commands and data. The memory block, state machines and access request stacks could be observed along with output that was generated.

5.5.4 Interrupt Manager Test Bench

The interrupt manager is tested with a standalone test bench containing the Simple network shown in Fig. 5.10. The test bench generated an interrupt for ESIB 3 to 7, the leaf nodes of the network. The Interrupt Manager is observed during this test bench to make see if it resolves the right interrupt source, and locates the proper interrupt vector. The waveform for the 5 localisation operations is in Fig.D.3. The IM H-Array of the Simple network is hard-coded into the interrupt manager in the current hardware revision. After locating the source, the interrupt manager either signals an interrupt to the controller or it dismisses it based if the interrupt vector is not activated. This means that only the localisation is tested, since none of the interrupt vectors are activated. This also validated the ESIB implementation as a full implementation of the Simple network was used to create the localisation vector.

5.5.5 Dependability Manager Simulation

Before the FPGA emulation is done, a simulation is performed of the entire dependability manager processor. During simulation, all facets of the coprocessor can be observed; at least all signals in and between the entities. ModelSim is used to simulate the processor, which is in a wrapper entity along with a IJTAG test network and a clock generation circuit. The controller entity also outputs the instruction and the states in a human-readable form. The program counter, registers and LEDs also help with under-standing the flow of the program. Test programs are loaded from the *loaders* folder. The dependability manager can be started with different *run* architectures. These VHDL architectures configure the loader-file and other parameters. The execution of a test application can be followed closely with this approach, for an example see Fig. 6.4 in section 6.3. The disassembly, created with `objdump`, creates a clear image of the program being tested.

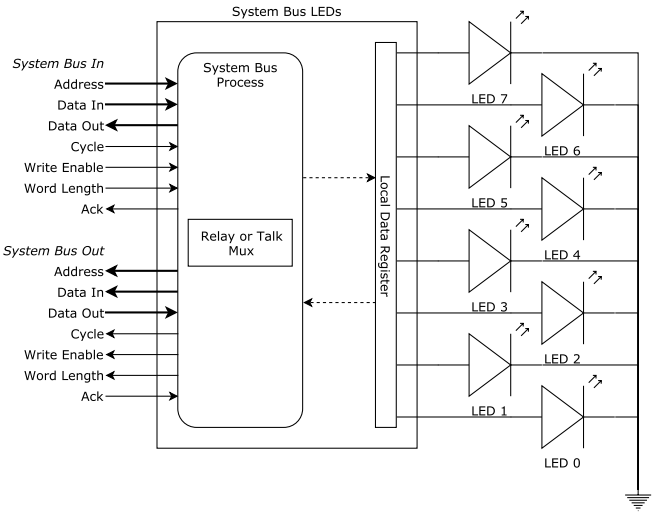


Figure 5.16: Detailed block diagram of the LEDs used for output of information.

5.5.6 Dependability Manager Emulation

The dependability manager was emulated with the Mingle network on an FPGA. Some form of output was needed to display the status of the test. The board offers many GPIO pins and eight LEDs which where connected to the last system bus device to be discussed; the LED driver. A block diagram of the device can be seen in Fig 5.16. It shows a simple device containing one read-write register connected to the system bus. The register maps its values directly to the output pins of the FPGA, which in turn are connected to the LEDs of the DE0-Nano FPGA development board, shown in 5.17. The DE0-Nano needs to be used along with Quartus to compile the design and program the board. The programming is not persistent by default, meaning that it needs to be re-uploaded when power is removed from the board.

The LED driver system bus device was also validated during the ModelSim simulation described in the previous section. The LEDs offer a 256 different combinations as output to the user. They are memory mapped and a driver is created to write an integer or to turn a specific on or off. The first test of the dependability manager was a simple *blink* project. It is an executabel that does not use the retargeting engine or the interrupt management unit. It is a program that executes all instructions in the instruction set and reports its status via the LEDs. After the complete system test a Fibonacci sequence will be calculated recursively and shown on the LEDs. After that it moves to a success state which shows a decorative sequence on the LEDs.

Chapter6 will continue with the quantification of the retargeting engine and the interrupt manager. It also discusses the use of a logic analyser to check the output

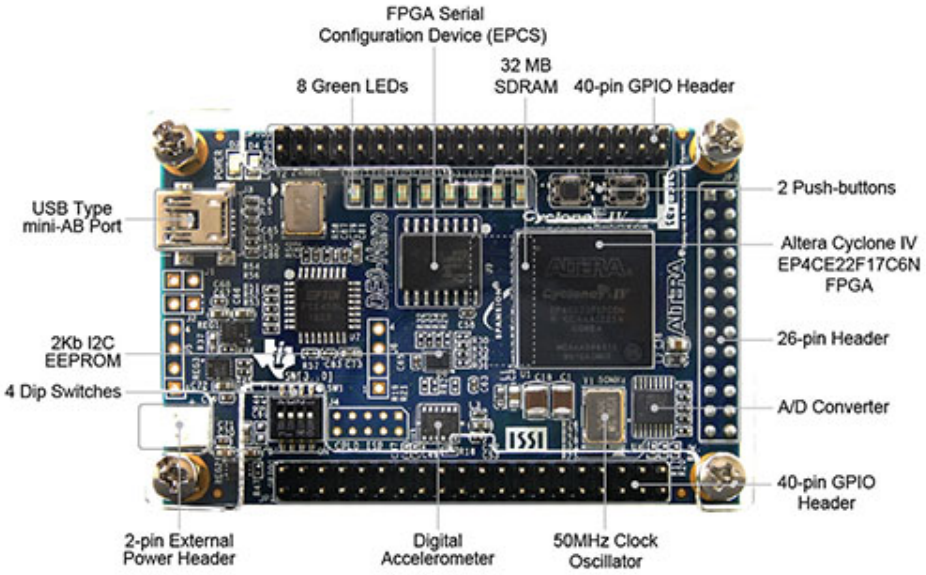


Figure 5.17: DE0-Nano FPGA development board, picture from Terasic [78].

of the retargeting engine.

5.6 Discussion

The DM is based on a processor that is implemented for this research partly based on another implementation of the RISC-V architecture [79]. It was a challenge but a worthwhile experience. However, during writing of this thesis the existence of Qsys, also know as the Platform Designer, by Altera [80] came to light. Initially, I believed the use of these tools to be too expensive but Altera provides many of them for reduces prices for academic purposes. This tool creates processors with customizable properties regarding size and complexity. The processors are based on the Nios2 ISA and could have saved much time and effort. The RISC-V processor that was created as a base for the DM is still a viable option. The current DM architecture allows modification, customisation and optimisation of the hardware to work towards a smaller and lighter processor where a Qsys generated core would be less flexible. However, a Nios2 core would have much more features available such as instruction caching, interrupt management and a programmer available via USB. It could have been worthwhile to research the suitability of this architecture instead of dismissing it too soon.

The implemented core provides no caching, pipelining or out of order execution, and this affects efficiency. It is almost a textbook example of a RISC processor

with some exceptions. The performance per clock cycle can be improved greatly by adding more complexity to the design. One of the options is to separate the instruction memory and RAM memory; basically moving from a Von Neumann to a Harvard architecture. This allows caching and pre-fetching of the next instruction even when the system bus is occupied with writing data. The next step after that would be pipelining as it is almost a guarantee that the clock could be reduced while maintaining performance. A lower clock speed could aid the lifetime of the DM and in essence make it more likely to outlive the functional layer. For pipelining to occur, the datapath elements must be more separate. The implementation reuses the ALU to calculate the next Program Counter value and this also prevents pipelining. Chapter6 will further discuss the efficiency and possible improvements.

The design of the retargeting engine moved through different versions. The addition of a memory IP block was prompted by the emulation on an FPGA. Using registers to store the H-Array, AR-Stack, and read values would not have fit on the Cyclone IV. Checking the Access Requests needed to be done one-by-one to reduce the design size. Also the extraction of data from a large shift buffer increased the design size past its boundaries. Adding the memory, and doing operations sequentially made the retargeting engine slower but feasible. Some optimisations are made to save clock cycles; for example not fetching empty stacks. Another speed improvement would be to access memory while the retargeting process continues. The retargeting engine has been proven in Zakiy's DM [1] as well as our iteration of the DM. If the dependability manager's architecture is changed again, it should not be difficult to adapt the retargeting engine with another bus protocol.

The retargeting engine uses its memory heavily. When a TDR is encountered in the H-Array, every non-empty AR-Stack is checked. This means loading the access requests, one-by-one, from the top of the stacks. An improvement would be to have a cache for the top access requests, or at least the instrument-id that they target. This would make checking the AR-Stack faster and the H-Array can be traversed in less time. Another improvement would be to separate the generation of the shift buffer and the shifting. The next access vector could be created while the current one is being shifted into the network.

The interrupt manager was implemented but only tested on one network. The improved network structures are also proven by testing on the network. Unfortunately, the network size is smaller than those used in previous research [3]. Implementing a larger network would have cost time and effort, and frankly, should be automated. The interrupt manager has been shown to work and it resolved all different interrupt

sources. Chapter6 tests the interrupt manager further.

The validation of the design was a success. The test benches of the separate entities helped tremendously as a kind of test-driven development. The test benches were only created with a test pattern generator. They lacked a response evaluator, which could have enable automated testing. Creating these test benches would have been incredibly cumbersome. ModelSim is a valuable tool to troubleshoot the implementation, especially when the design grows and becomes more complex. Emulating the design on an FPGA was a rewarding experience that has proven the capability of the dependability manager.

Chapter 6

Experimental Results

The final part of the methodology is a verification and validation of the product. The dependability manager is a processor with a retargeting engine and a interrupt management unit. The operation of the retargeting engine and interrupt manager will be quantified separately. The processor itself will also be analysed. This will happen as a complete system test; the test program is loaded into the dependability manager which will be connected to a Bastion benchmark network. The 'Basic' set of networks are chosen as their VHDL implementation is available. To test the interrupt manager, this research will rely on the simple network described in Fig 5.10 in the previous chapter. The retargeting engine is tested for single reads and writes and for `iApply` groups of multiple reads or writes. This approach has been taken before for simulating the principle of the retargeting engine [**?**]. The clock cycles needed for the traverse and generate algorithm are counted, as well as the cycles needed for shifting the access vector. The extraction cycles are counted for the read commands. The target instrument for the read and write operations are chosen randomly.

After the results of the retargeting engine are discussed, the focus is shifted to the interrupt manager. It is tested to quantify the time it takes to load an ISR after an interrupt flag is present. Two tests are performed on the simple network, one where there is no other activity on the IJTAG bus, and one where random single writes are performed. The cycles needed to gain access to the IJTAG bus and to localise the interrupt are amongst the results, along with the time from flag to ISR request, and the time until the processor returns from the ISR.

Finally some general statistics of the processor are shown and some system components are discussed during simulation to see how they operate. The statistics can be used to determine how to increase the efficiency of the processor. The processor is also emulated on an FPGA and a logic analyser is used to verify its operation.

Table 6.1: Parameters for Retargeting Tests.

Network	H-Array	TDRs	Test I			Test II			Test III		
			iApply	iWrites	iReads	iApply	iWrites	iReads	iApply	iWrites	iReads
Mingle	38	8 / 4	200	100	100	200	400	400	200	400	400
BasicSCB	41	5 / 3	200	100	100	200	300	300	200	300	300
TreeFlat	49	22 / 10	200	100	100	200	1000	1000	200	1000	1000
TreeFlatEx	218	63 / 10	1000	500	500	200	1000	1000	200	1000	1000
TreeBalanced	161	45 / 10	1000	500	500	200	1000	1000	200	1000	1000
TreeUnbalanced	96	35 / 10	1000	500	500	200	1000	1000	200	1000	1000

TDRs in total / TDRs per iApply group for Test II and III.

6.1 Performance of the Retargeting Engine

The retargeting engine is implemented according to the flowchart in Fig. 5.8. It consists of three main phases; the retargeting, shifting and extracting. The retargeting is also known as 'traverse and generate' [2]. The retargeting engine was connected to the test networks listed in Table 6.1. The H-Array definitions for these networks where made by hand according to the ICL definitions. They were implemented using the developed libraries and included in the test program. Each network was subjugated to three test: random single reads and writes, random read and write-groups and random read and write groups while resetting the network. Resetting the network reverts all the SIBs and SCBs to their initial value, which means the segments were closed.

Table 6.1 shows the amount of reads and writes performed in each test. The reads and writes were divided equally over the iApply groups. The table shows the length of the H-Array, the amount of Test Data Registers and the amount of registers that were randomly selected for each iApply group. The dependability manager was clocked at 50MHz and the TAP controller was used to generate a TCK of 0.96MHz. This error was discovered when all the tests were finished; it was supposed to be 1MHz. This error has slightly affected the test results in the 'Shifting' columns. It should not matter for this discussion as the time needed for shifting grows linear with the length of the Access Vector (AV), which is also added to the tables. The tests on the 'TreeUnbalanced' network were performed with a faster clock speed of 5MHz. This range in frequency is assumed to be used in industry. Using IJTAG networks at higher clock speeds may require more power. The maximum clock speed is limited to the physical speed of the longest combinational path in the network. In this chapter it is assumed that the networks are built to accommodate the speed of an off-chip IJTAG controller. The on-chip retargeting engine may perform at higher speeds than an IJTAG programmer, if the IJTAG network supports these speeds.

The Clock Cycles (CC) displayed in Tables 6.3, 6.4 and 6.5 are based on the 50MHz system clock. Using different TCK speeds explores the time needed for

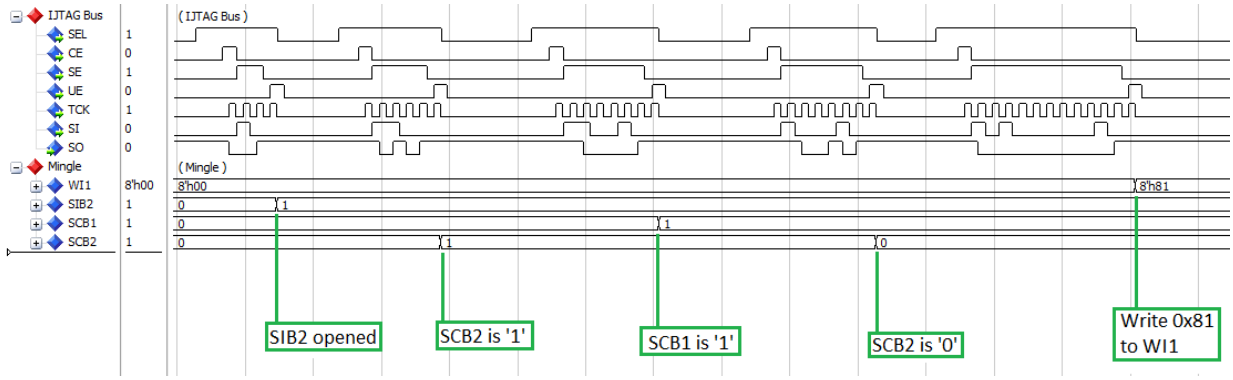


Figure 6.1: Writing `0x81` to WI1, see Table 6.2.

shifting versus retargeting. This time is linear to the length of the access vector; every bit in the access vector takes 50 Clock Cycles when the TCK speed is 1Mhz (52 at 0.96MHz). The TAP control synchronisation takes 1 to 50 cycles. The Capture state and Update state at the beginning and end of each CSU again take 50 cycles.

The benchmark networks used for this research differ in structure in multiple ways. Mingle is a collection of SIBs and ScanMuxes with 8-bit TDRs. BasicSCB consists only of ScanMuxes and also has 8-bit TDRs. TreeFlat is a tree structure of SIBs where almost all of them are on the same level. It has a ScanMux to bypass the tree and have all the 8-bit instruments as long scan chain. The next network starts to get more serious with larger and differently sized TDRs. The TreeFlatEx network reused modules from the ITC'02 benchmarks in a flat tree structure. All these modules exist on the same level in the tree. TreeBalanced is again a tree structure but with more depth. The tree is balanced as its name suggests, meaning that the ITC'02 modules are spread across a branching tree. Most of the nodes in the tree have 2 or 3 child nodes. The last network contains huge TDRs with lengths of up to 2625 bits. The tree structure is lopsided; having many large modules at one side of the tree whilst having almost none on the other side. Drawings of all the networks are available online [27], the Mingle network is displayed in Fig C.1. The use of ScanMuxes or SIBs and the structure influences the length of the access vector for reaching an instrument. As described in section 5.2, SIBs are not closed by the retargeting engine unless it is instructed to. This results in segments in the active scan chain that might not be necessary.

It is interesting to look at an example of the operation of the retargeting engine before the test results are discussed. A simulation waveform created with Modelsim is shown in Fig. 6.1. The waveform shows vital variables within the retargeting engine such as its state machine and AR-Stack pointers. It also shows the operation

Table 6.2: Retargeting of one `iWrite` and one `iRead` of WI1 on the Mingle network.

		Retargeting (CC)	Shifting (CC)	Extracting (CC)	AV	Description
`iApply` 1						
	CSU 1	158	186	-	2	Open SIB2 to access segment of WI1
	CSU 2	293	327	-	4	Put '1' in SCB2 to access SCB1
	CSU 3	401	427	-	6	Put '1' in SCB1
	CSU 4	402	426	-	6	Close SCB2 to access WI1
	CSU 5	285	699	-	11	`iWrite` to WI1
`iApply` 2						
	CSU 1	286	674	5	11	`iRead` from WI1

Refer to the waveform in Fig. 6.1 and the Mingle network in Fig C.1.

of the TAP controller and IJTAG bus activity. There are six CSU cycles present in the figure which correspond to the rows in Table 6.2. The waveform also shows the network structures (TDRs, SIBs and SCBs) of the Mingle network. The retargeting engine is instructed to perform two `iApply`s during the test, one to write to the instrument and one to read from it. The device retargets the network by writing to the SCBs and SIBs as shown in Table 6.2. The instrument remains in the scan chain after the write is performed, that is why the read requires less retargeting of the network. The table show clock cycles needed for retargeting and shifting, and it shows the length of the access vector for each CSU cycle. The time needed for retargeting is dependant on the amount of segments that need to be traversed. The shifting time grows linearly with the access vector, and it is influenced by the chosen IJTAG clock. The execution of the read request takes 5 clock cycles. These are needed to check for read requests (1), to load the read request from memory (2) and to store the read value back in the memory (2).

As mentioned, three test were performed on the 'Basic' set of benchmark networks. Before the tests could be done, the retargeting engine needed validation. After all, the H-Arrays of the networks were created by hand from the ICL files and drawings online [27]. Furthermore, the VHDL versions of the networks were used which might differ from the ICL files. To validate both the H-Array and the VHDL implementation provided by Bastion, all instruments in the network underwent a read after write test. All tests were performed using Modelsim. The WrappedInstruments, as they are called, were added to the ModelSim waveform. The WrappedInstrument entity has a ScanRegister connected to the scan chain. The instrument copies the value from the ScanRegister if its most significant bit is '1' during a CSU update. This value persists and can be captured. Either the ScanRegister or the internal register of the WrappedInstruments were monitored. The created H-Array and used VHDL performed well and the quantification tests could be performed.

The first test consists of single writes and single reads at randomly chosen in-

Table 6.3: Retargeting Test I, random reads and writes, no reset.

Network	CSU	Access Vector			Retargeting CC			Shifting CC			Extraction CC	
	$\pm$	$\pm$	$\bigtriangledown$	$\bigtriangleup$	$\pm$	$\bigtriangledown$	$\bigtriangleup$	$\pm$	$\bigtriangledown$	$\bigtriangleup$	$\pm$	η
Mingle	1.51	43.29	2	51	654.90	158	750	2358.75	212	2785	5.00	100
BasicSCB	1.57	34.37	7	37	738.55	406	804	1896.81	464	2057	5.00	100
TreeFlat	1.53	88.71	2	101	748.92	164	1032	4719.54	191	5385	5.00	100
TreeFlatEx	1.12	4748.60	14	5102	5555.86	1094	5876	247035.35	847	265437	5.00	500
TreeBalanced	1.16	4964.04	1	5220	3956.49	80	4176	258237.69	150	271573	5.00	500
TreeUnbalanced	1.03	40470.69	2	41887	2645.05	158	2750	485675.88	50	502677	5.00	500

Average number of CSU Cycles needed per iApply, the Access Vector length, Retargeting cycles, Shifting cycles and cycles needed for Extracting read values.

struments. Table 6.3 shows the test results on the six networks. The average ($\pm$), minimum ($\bigtriangledown$) and maximum ($\bigtriangleup$) are displayed to give an intuitive image of what is happening. Retargeting, Shifting and Extracting are measured as the clock cycles needed for 'Traverse and Generate', the shifting of the access vector and the retrieval of read values from the incoming shifted bits. The average number of CSU per `iApply` is in the second column, when this value approaches 1 it means that the target instrument is in the active scan chain more often. During the tests, the retargeting engine is never instructed to close a SIB and eventually they are all opened. This means eventually the RE only needs one CSU to access any instrument. This behaviour is also visible in Table6.2: 5 CSU cycles to open access the instrument, and 1 when it is already in the scan path. The networks that contain mostly SIBs, i.e. TreeBalanced, TreeUnbalanced and TreeFlatEx, show this behaviour even more. Although less CSU cycles may seem favourable, the opened segments increase the time needed for retargeting and shifting. Less segments in the H-Array can be skipped during retargeting as they are open. An open segment increases the length of the access vector which is exacerbated by the relatively slow TCK frequency. The extraction of read values is consistent. A single read value extraction takes 5 cycles, just like the example discussed earlier. The extraction clock edges are counted for access vectors that contain required values from instruments, so not every CSU cycle.

The second test uses `iApply` groups of multiple access requests. The size of these groups can be found in Table 6.1. The performance of the retargeting engine is shown in Table 6.4. Grouping of scan operations allows the retargeting engine to apply multiple commands at the same time. The order in which this happens is not relevant. More CSU cycles are used per `iApply` to retarget the network to access all the instruments in the group. The length of the access vector grows to its maximum size similarly to the first test as no SIBs are closed. This reflects in its average and in the statistics for the shifting of the access vector. The cycles needed to traverse and generate the access vector is similar to the first test. This may look strange as more reads and writes are applied. There are two reasons that may explain this.

Table 6.4: Retargeting Test II, random read groups, write groups, no reset.

Network	CSU	Access Vector			Retargeting CC			Shifting CC			Extraction CC			
	$\pm$	$\pm$	$\triangledown$	$\triangle$	$\pm$	$\triangledown$	$\triangle$	$\pm$	$\triangledown$	$\triangle$	$\pm$	$\triangledown$	$\triangle$	η
Mingle	2.23	43.43	2	51	699.17	185	861	2364.71	190	2785	9.74	5	17	183
BasicSCB	2.75	34.88	7	37	761.13	448	871	1922.50	463	2057	8.69	5	13	156
TreeFlat	2.75	93.99	2	101	896.44	245	1469	4995.85	204	5385	21.14	5	37	200
TreeFlatEx	1.70	5058.89	14	5102	6487.17	1607	7769	263167.52	828	265437	30.94	5	41	668
TreeBalanced	1.49	4989.50	1	5220	4621.55	125	5471	259562.22	139	271573	41.00	41	41	100
TreeUnbalanced	1.47	40439.47	2	41887	3051.72	239	3619	404418.02	48	418898	41.00	41	41	100

Average number of CSU Cycles needed per iApply, the Access Vector length, Retargeting cycles, Shifting cycles and cycles needed for Extracting read values.

Table 6.5: Retargeting Test III, random read groups, write groups, with reset.

Network	CSU	Access Vector			Retargeting CC			Shifting CC			Extraction CC			
	$\pm$	$\pm$	$\triangledown$	$\triangle$	$\pm$	$\triangledown$	$\triangle$	$\pm$	$\triangledown$	$\triangle$	$\pm$	$\triangledown$	$\triangle$	η
Mingle	6.81	16.47	2	51	553.48	185	845	963.06	186	2771	6.93	5	17	270
BasicSCB	6.77	13.09	7	30	675.32	406	882	786.15	446	1693	7.32	5	13	190
TreeFlat	4.68	42.40	2	89	805.01	245	1447	2312.26	186	4761	21.10	5	41	199
TreeFlatEx	4.62	752.44	14	2831	3487.23	1607	5343	37725.17	779	141663	18.62	5	41	227
TreeBalanced	8.00	814.42	1	3921	1654.10	125	3793	42459.49	134	203974	12.20	5	29	363
TreeUnbalanced	8.67	9204.11	2	28789	1712.28	239	2704	92064.43	40	287916	10.43	5	37	424

Average number of CSU Cycles needed per iApply, the Access Vector length, Retargeting cycles, Shifting cycles and cycles needed for Extracting read values.

The first is that most segments are opened in both tests and need to be traversed. The second is that there are more CSU cycles performed in the second test and the measured clock cycles are averaged over the amount of CSU cycles. The minimum and maximum for retargeting are higher in the second test, especially for the larger iApply groups. This is because more AR-Stacks need to be checked when an instrument entry is encountered during retargeting. The extraction of read values is not as consistent as multiple read requests can be performed per `iApply`. The CSU cycles that accessed an instrument with read requests is shown in the last column (denoted by η). All accessed instruments were in the scan chain for TreeBalanced and TreeUnbalanced. This resulted in all 10 read requests being handled in the same CSU cycles; 1 clock cycle to check and 4 cycles per read request. This is likely since the instruments selected by the ScanMux of TreeBalanced (Module 5) where not used for the `iApply` groups. TreeUnbalanced has no ScanMuxes at all and all its SIBs were opened.

The third test is the same as the second test but the network is reset after every `iApply`. The results of the third test can be found in Table 6.5. Resetting the network closes SIBs and also resets the state vector. Compared to the second test, this decreases the average length of the access vector, but it increases the amount of CSU cycles. The extra CSU cycles are needed to re-open the required segments of the network. The retargeting time is also less per CSU as most segments are closed and can be skipped. Effectively, the difference between test II and III is similar to the two `iApply` operations displayed in the example; Table 6.2. Test III starts

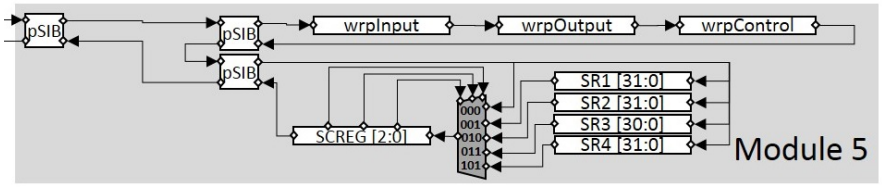


Figure 6.2: 3-Bit ScanMux present in TreeFlatEx and TreeBalanced.

with a closed network (`iApply` 1) while test II can retarget from the current network state (`iApply` 2). The performance is somewhat better compared to test II although it is heavily influenced by the network structure and the length of the access vector. The time needed for retargeting per CSU is lower but more CSU cycles are needed. More cycles are used per `iApply` for retargeting. The same thing happens when read requests are handled; more CSU cycles but less time used per CSU.

Some difficulties were encountered during the tests. Modelling an n-bit ScanMux as multiple 1-bit ScanMuxes in the H-Array was suggested when the retargeting engine was first introduced [2]. Unfortunately, this creates a new problem for the retargeting engine. Module 5 in the TreeBalanced and TreeFlatEx networks contains such a ScanMux, see Fig. 6.2. The division of the SCB into multiple 1-bit registers created temporal issues between them. A temporal conflict occurs when two instruments are on the AR-Stack that need conflicting configuration of two or more SCBs. The SCB in Fig. 6.2 is modelled as three SCBs; scb0, scb1 and scb2. To access SR1 and SR2 simultaneously different things need to happen. First scb0 is '0', so the retargeting engine enters the segment and adds write '1' to scb2 to access SR1. But the retargeting engine also wants to open the other segment to access SR2 and a write '1' to scb1 is added. These two writes are popped of their AR-Stacks, the CSU cycle applies the changes and the retargeting happens again. Now both the instruments are still not in the active scan chain. The network state shows '011' and the retargeting engine adds writes (one to create '010' and the other to create '001') to the AR-Stack. The resulting network state is '000' and the problem iterates. These temporal issues are mitigated in new research into the retargeting engine using SCB chaining trees [25]. During tests II and III, only one of the scan registers of Module 5 was added to an `iApply` group to prevent these occurrences.

Another issue with the Bastion scan networks and the realisation of the retargeting engine is the size of the TDRs. A maximum size was chosen for test data based on the description of PDL level-1 in the IJTAG standard [5]. The design of the H-Array, AR-Stack and the retargeting engine is based on the assumption that TDRs are 32 bits or less. Most of the TDRs in the used benchmark network had

TDRs with a length more than that. The retargeting engine was modified for testing purposes to pad the value with '0'. The extraction of read values was also adapted to manage these larger values and only the least significant part was extracted to be stored in the memory. This is not desirable. A fix would be to use multiple TDR entries in the H-Array to emulate a larger TDR, but this is not scalable. To use large TDRs, the retargeting engine needs to change. An access request should at least store a pointer to memory, along with more information such as the length and index within the TDR.

Optimal scan pattern generation remains a difficult problem. The access vector needs to be minimised and the creation of the access vector must not take long. The retargeting engine traverses and generates the access vector quickly, but makes no effort to shorten the access vector. The retargeting engine can be used by the user to close segments but this does not happen automatically. The access vector is observed to continuously grow in a SIB based network until the maximum length is reached. No algorithms have been proposed to close SIBs. The temporal issue has been mitigated in research [25] but has not been implemented in this work.

6.2 Performance of the Interrupt Manager

The interrupt manager for IJTAG networks was developed in this research. Normally IJTAG networks do not support interrupts and the instruments need to be polled continuously. There are multiple strategies, currently in research, to add interrupts to the scan chain architecture. This research implemented one of them; an interrupt manager and an interrupt enabled SIB [3]. This section will test the effectiveness of the chosen architecture, and determine the time needed to handle an interrupt flag. The interrupt manager is designed to share the IJTAG bus with the retargeting engine through the TAP controller. This creates a small overhead and delay when handling an interrupt. The test, ran on the Simple network, is designed to show the behaviour of the interrupt enabled IJTAG network.

The test is executed on the Simple network, which was introduced in section5.3. The hierarchical tree network contains 5 interrupt sources at its leaves. The network is smaller than those used in previous research [3]. The goal is to ensure the correct operation of the interrupt manager, and to check the assumptions made about the localisation time. All the measurements are expressed in system clock cycles, and the IJTAG TCK is again set to 1MHz. The network could operate at higher frequencies. The interrupt manager first waits for bus control, which is measured. The

Table 6.6: Interrupt Manager Test I.

Interrupt Source	Waiting (CC)			Localisation (TCK)	Flag to ISR Request (CC)			ISR Request to Return (CC)		
Source	±	▽	△	±	±	▽	△	±	▽	△
ESIB_L3	1.00	1	1	8.0	406.50	382	431	54.06	49	59
ESIB_L4	1.00	1	1	7.0	356.47	332	381	53.99	49	59
ESIB_L5	1.00	1	1	6.0	306.63	282	331	54.02	49	59
ESIB_L6	1.00	1	1	6.0	306.68	282	331	53.90	49	59
ESIB_L7	1.00	1	1	5.0	256.79	232	281	54.00	49	59

localisation through the IJTAG bus is the next step. The last two measurements are total time from flag to ISR request, and total time to finalize the ISR request. During the test, instruments in the network will generate an interrupt at their leaf node. The localisation will occur and the next interrupt will be generated at a different instrument. The time of the interrupts is randomized and only one interrupt is active at the same time. The measurements are collected of 250 interrupts per instruments. A different ISR, from the test program, is coupled to interrupt vector. The routines are similar, empty functions that only contain a return-from-interrupt command. The retargeting engine is inactive during the test. The instruments in the Simple network are 8-bit sized TDRs. Along with the SIBs, this creates a maximum access vector length of 48 bits. However, with the IJTAG enabled network structures, the longest access vector during localisation is 6 bits.

The results of the test are shown in Table 6.6. The test shows almost no overhead for waiting on IJTAG TAP control. The localisation happens as fast as the IJTAG bus clock speed allows. The length of the localisation vector is based on the position of the specific instrument in the network. These vectors can be found in Fig 5.10 in section 5.3. The localisation cycles correspond to the length of the vector for each instrument. The servicing of the ISR takes roughly the same time as the first test, and for each interrupt source. The time needed is roughly 55 cycles. During the tests, the DM executes an 'empty' interrupt service routine. It contains a minimum of 4 instructions; 3 to manage the stack pointer, present at each function, and 1 to return from the interrupt.

This concludes the analysis of the interrupt manager. The localisation of interrupts works as expected. The layout of the network decides the length of the access vector. This is directly coupled to the shifting time, which is determined by the TCK speed. Localisation occurs between 250 and 400 cycles while a polling strategy would require at least 2500 clock cycles to check every instrument in the Simple network. A higher TCK frequency would reduce the cycles needed but the length of the access vector would not change. The TAP controller handles request from the retargeting engine and interrupt manager well.

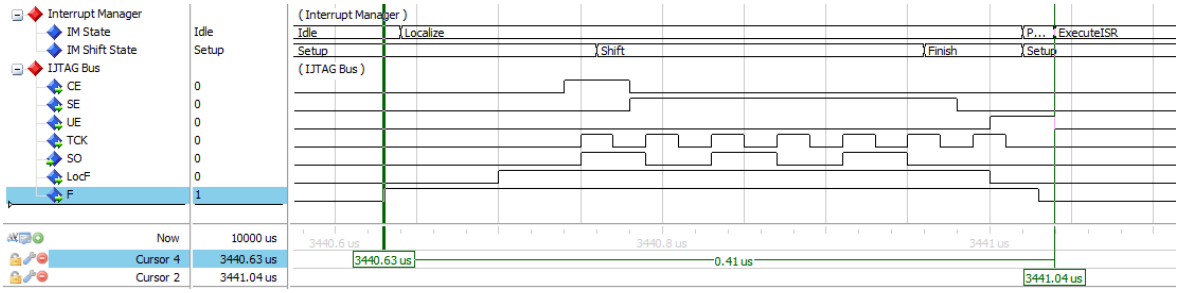


Figure 6.3: Localisation of ESIB_L4 with TCK at 25MHz.

6.3 Performance of the Dependability Manager

This section is dedicated to a brief analysis of the performance of the dependability manager. The system consists of the core processor, which is connected to the retargeting engine and interrupt manager via the system bus. The dependability manager has no caching strategies, and the Wishbone bus implementation only supports single reads and writes. Table 6.7 shows the states of the controller and the percentage of time spend in them. The controller moves through these RISC-like states to execute the instructions. The results were gathered during Test II of the interrupt manager in the previous section. The last column shows the cycles needed per instruction. Normally any register-to-register calculation takes 12 cycles, but load and store operations wait for the value to be stored and take a cycle longer.

Table 6.7: Percentage of cycles spend in controller states.

Fetch	Start ISR	Wait on Fetch	Decode IR	Execute	Write Back	Update PC	Store PC	CC per Instruction
7.67%	0.04%	47.18%	7.67%	7.67%	7.67%	14.43%	7.67%	12.93

The lack of advanced memory architectures reduces the efficiency of the dependability manager. The fetching of instructions from the main memory takes 47.18% of the time. The system bus takes time to communicate and in addition to that, the memory IP block also takes 2 clock cycles to produce output. The controller waveform is shown in Fig 6.4 along with the system bus master. The fetch delay be improved by cache prefetching of the instructions [81]. Most of the instructions will not affect the program counter, so the next instruction could already be loaded. The single read/write operations of the Wishbone bus can be improved with burst reads/writes [77]. This will facilitate the prefetching of instructions. The program counter is stored in the datapath and uses the ALU to increment itself. This takes

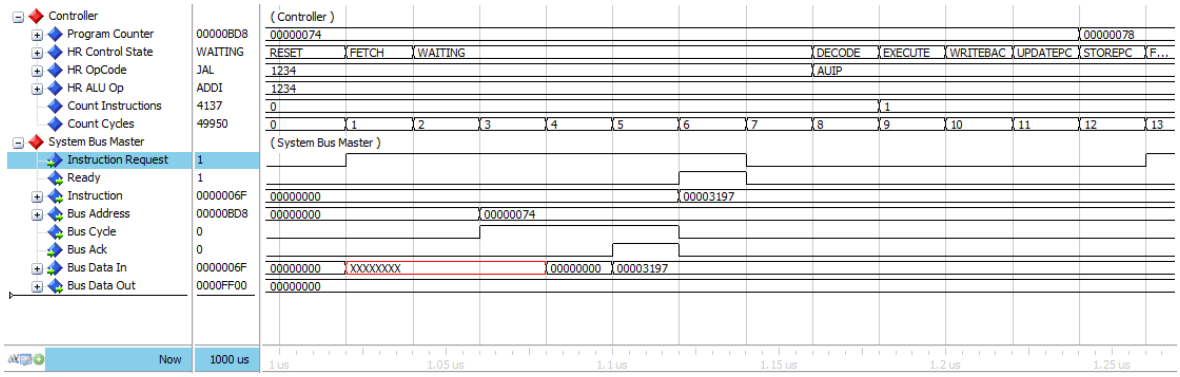


Figure 6.4: Operation of the controller and system bus master.

an extra state compared to traditional RISC architecture [81]. A dedicated program counter unit could aid, although some instructions directly calculate on the program counter. The final addition to this collection of improvements would be to implement the controller as a pipelined architecture. This would require more storage elements between the datapath components but would also dramatically improve the Cycles per Instruction (on average).

The design of the dependability manager was simplistic. The results in this section are not unexpected. There are more improvements that could be made to the processor. The impact of these improvements must be measured, which would decide whether they have the intended effect. It is clear that the Dependability Manager correctly executes high level language programs. These programs are executed within a reasonable time frame. Improving the speed of the processor may be necessary for calculation-heavy dependability applications.

6.4 FPGA Resource Usage

Quartus Prime Lite Edition 17.1 has been used to compile the designs for emulation on a DE0-Nano development board. This board contains a Cyclone IV FPGA, and is shown in Fig.5.17. The design passed the timing analysis of Quartus and was latch-free. After compilation, Quartus was used to program the board, along with the contents of the Main Memory. This is provided by reading the contents of an *ihex* file, named dm_system_memory_xxxxx.hex. The LEDs are used during tests on the development board to provide feedback. Just like the simulations, the Dependability Manager processor is instantiated in a wrapper entity, but this time with a clock divider process. This process is added to reduce the 50 MHz clock frequency of the DE0-Nano to other lower frequencies. It was unnecessary since the timing analysis

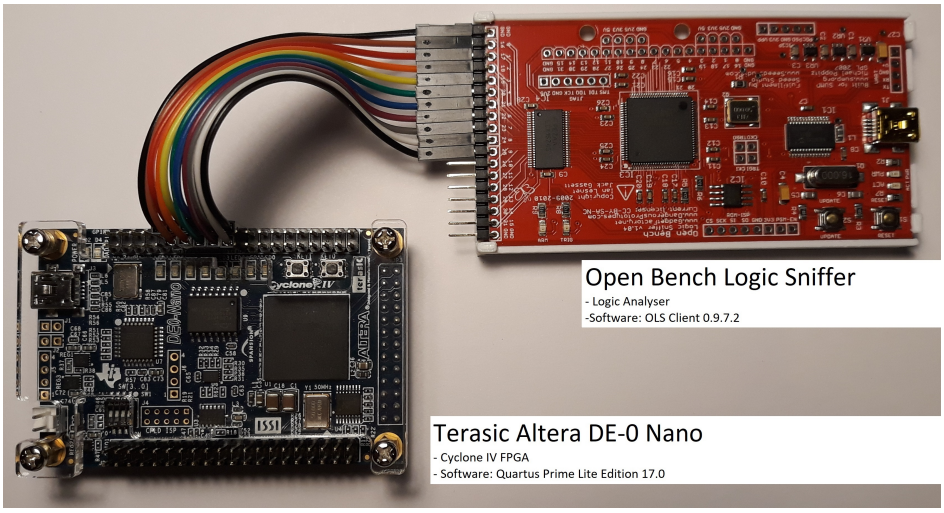


Figure 6.5: Setup used to gather emulation test.

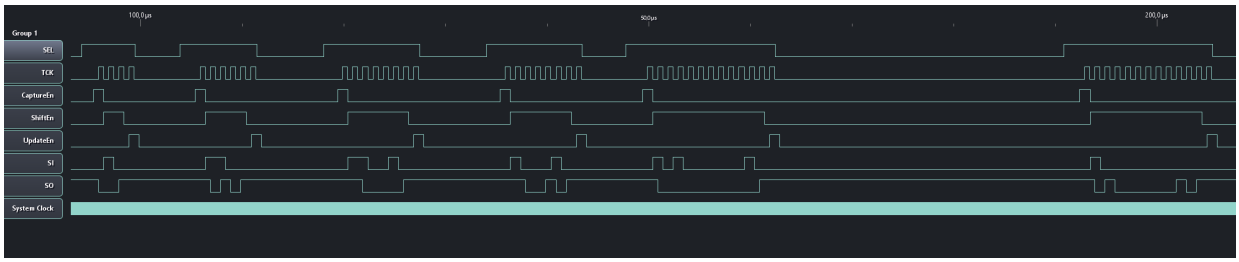


Figure 6.6: Data gathered with logic analyser.

was successful. The wrapper entity has a clock input, two inputs for buttons and 8 output 'pins' for the LEDs. Later on, the GPIO header was also included and the TAP connections have been mapped to it. A test setup was created by connecting a logic analyser to the GPIO header of the DE0-Nano. A waveform was created with the logic analyser, see Fig. 6.6. The example from Table 6.2 was continuously ran on the Mingle network. The wrapper entity of the emulation also contains one of the IJTAG network entities to communicate with the retargeting engine or with the interrupt management unit. This was, off-course, based on which test was executed.

The compilation results of Quartus are shown in Table 6.8. It shows the different design entities of the dependability manager along with their respective resource usage. The resources are expressed as Logic Cells (LC), Dedicated Logic Registers (DLR) and Memory Bits. The basic building block of Altera FPGAs are Logic Elements. They can be configured in *normal mode* or *arithmetic mode*. The memory bits can be used with the Altera Memory IP. The LC and DLR are probably related to the modes of the logic elements but no more information given about them [82]. The FPGA was an Altera Cyclone IV EP4CE22F17C6N; it contains 22000 logic ele-

Table 6.8: FPGA resource usage of Dependability Manager in Quartus.

Entity			LC	DLR	Memory Bits
fpga_dm_main			12.556	3.722	294.912
	dm_main_processor:unit_under_test		12.283	3.502	294.912
		dm_main_control:control	393	129	-
		dm_system_bus_master:memory_manager	809	177	-
		dm_main_registers:registers	2.179	1.056	-
		dm_main_alu:alu	645	33	-
		dm_main_alu_in_mux:b_mux	677	0	-
		dm_system_bus_slave:slave_ram_memory	198	76	262.144
		dm_system_bus_leds:slave_leds	17	12	-
		dm_retargeting_engine:retargeting_engine	6.790	1.352	32.768
		dm_main_interrupt_manager:interrupt_manager	811	612	-
		dm_tap_control:ijtag_tap_control	42	21	-
	Mingle:mingle		272	220	-
	Simple:simple		196	144	-

ments and 594Kb of memory. It is hard shed a light on the resource usage. Devices from different manufacturers cannot be compared easily with each other [83]. The numbers here are also not reflective of the eventual design size in silicon. However, it is good to take a look and see where some improvements can be made in the design, and to explore different parameters.

It becomes clear from Table 6.8 that the entities which store data require most logic cells to be emulated. This concerns the retargeting engine, registers, and the interrupt manager. This is also the reason that the Altera Memory IP is used as the ram memory. It is unclear why the *b_mux* takes such an amount of resources: it is only a 32-bit asynchronous multiplexer. The interrupt manager also requires storage for the interrupt hierarchy array and interrupt vector table. The retargeting engine is the largest entity in the design. The memory bits it uses are because of the Memory IP that is used to store the H-Array, AR-Stack and read values. Without this the design would not fit the FPGA. A brief design exploration is held to determine the factors that affect its resource usage. The retargeting engine is designed with 4 parameters that affect the design. They are shown in Table6.9 along with the resource usage measured. The default instantiation of the device is: 32 AR-stacks with 8 entries each, 256-bit shift buffer, and a H-Array with 64 entries. The shift buffer has the most influence on the design size. Shifting and extracting the values takes a large amount of resources. There are actually two buffers; one to create the access vector during traverse-and-generate, which is copied to the actual buffer for shifting. This was intended to parallelise generating and shifting the access vector, which was never implemented unfortunately. Also, the shift buffer operates at a slower clock speed (TCK) than the rest of the system. Hypothetically, this allows the buffer to be stored in memory. The next value to be shifted could be loaded well

Table 6.9: FPGA resource usage of RE entity in Quartus.

	LC	DLR	LC	DLR	LC	DLR	LC	DLR
Shift Buffer Size:	128		256		512		1024	
	4.364	966	6.790	1.352	12.514	2.122	22.636	3.660
AR-Stacks:	8		16		32		64	
	6.445	1.248	6.605	1.284	6.790	1.352	7.212	1.484
AR-Stack Size:	8		12		16		24	
	6.790	1.352	6.873	1.391	6.865	1.385	6.967	1.424
H-Array Size:	32		64		128		256	
	6.672	1.283	6.790	1.352	7.027	1.485	7.438	1.746

before it is needed, but extracting read values would take more time. The amount of stacks is the next most influential factor. This is likely due to amount of stack pointers needed, which are integers bound by the stack size (-1 to stack size). Increasing the other parameters does not affect the design as much, mostly because the data is stored in the memory block. The size of this memory block is not altered for the tests.

The FPGA resource statistics give some insight into the size and complexity of each component. The numbers are not reflective of the actual hardware size if the dependability manager is made into silicon. Unfortunately, the processor was not compiled to a silicon design. This was not possible due to the use of the Altera IP. Because of this, there is also no information available about the power usage or maximum clock speed. The design exploration shows that the retargeting engine could be optimised to allow larger shift buffers. The current design fits the relatively small FPGA and it can operate at the 50MHz clock speed. The usage of Memory IP allowed the large high-level programs to be stored on the FPGA.

6.5 Conclusion

This chapter contains the test results of the retargeting engine and interrupt manager. It also discusses the efficiency of the dependability manager's processor in terms of speed and size. The results of the simulations are as expected. The retargeting happens quickly and the created access vectors are correct. The retargeting engine is tested on six benchmark networks. Their structure and size also affect the time needed to retarget. The major issues with the retargeting engine are that it does not automatically close SIBs. This increases the average access vector length. The second issue is that it cannot resolve temporal conflicts between SCBs. Furthermore, its design is not able to handle TDRs larger than 32 bit. However, these issues were known at the time of implementation and ongoing research will hopefully resolve them.

The interrupt manager localises the interrupt sources as fast as possible. The interrupt flag architecture of the ESIB creates a small, efficient scan chain for localisation. The slower clock speed of the TCK determines the time at which the network can be accessed, so shorter, optimized access vectors help tremendously. Sharing the IJTAG bus with the retargeting engine introduces long wait times for interrupt localisation. This cannot be mitigated as CSU cycles cannot be aborted according to the IJTAG specification.

The dependability manager was designed to be simple and maintainable. It features no pipelined architecture and its controller is based on the classic RISC state machine. The processor takes almost half of its time to load the next instruction. This can definitely be improved by instruction cache prefetching. Pipelining the dependability manager may be necessary for calculation intensive dependability procedures.

The dependability manager was successfully emulated on an Altera Cyclone IV FPGA along with the Mingle and Simple IJTAG network. The resources were predominantly used by the entities that store data. Different parameters were tried for the retargeting engine, and this gave some insight in the determining factors. The length of the shift buffer influences the design size the most.

The picture becomes pretty clear after these measurements. The retargeting engine and interrupt manager operate as expected and this reflects in the quantification. The dependability manager works well but there are many improvements that could be made. These advanced improvements were not feasible within the scope of this research project.

Chapter 7

Conclusions & Future Work

This work started with a goal to create a dependability manager coprocessor for IJTAG enabled System-on-Chips. The DM needed to execute self-awareness dependability applications which aim to manage the hardware degradation, allow dynamic reliability management or handle faults in hardware. Any software that increases dependability of a System-on-Chip or a Processor is considered a dependability application. The hardware that is used or reused to enable dependability management is part of a dependability layer, separate from the functional layer. The role of the DM is to provide a reusable platform that supports a large library of dependability applications, along with a uniform method of access to instruments in the dependability layer of a System-on-Chip or processor. The dependability layer uses the IJTAG network to write and read data to and from the instruments. A processor is made that support communication with this network.

After the background of the research field, the high level design of the DM is presented. This design starts with an application analysis discussing several dependability applications. The analysis shows that in general sampling frequency is very low, applications act as software based controllers and that a generic processor design can support the applications. The hardware design of the processor should be as small as possible to reduce the overhead of incorporating a DM in a SoC. The use of PDL to access instruments happens at such a speed that an application specific instruction set is not necessary. The extension of the instruction set, as in the previous dependability manager [1], can be replaced with a Memory Mapped IO interface to the retargeting engine. The choice of instruction set is redone and RISC-V is selected due to its size, extensions and the fact that MIPS32 is not open for use. Among the base instruction sets of RISC-V is also a reduced instruction set aimed at embedded systems which could be used in the future.

There are multiple ways to incorporate PDL into high level languages, for exam-

ple using a PDL interpreter or conversion to another programming language. The latter is implemented and the choice for a high level language, namely C, is made to allow cross compilation to many different targets. The previous work only targets MIPS32 assembly and is based on a modified version of PDL [1]. In essence, the compiler in this research converts the procedure files to a C framework, this is possible as almost all PDL level-0 code exists within the confines of a `iProc` procedure. As high level languages will be used for dependability applications, the need to execute Tcl and PDL level-1 programs fades. However, PDL level-1 files, along with Tcl, may need to be converted in future work. The new PDL compiler is extendible through the template engine and the HAL. Procedure files in PDL level-0 suffice to issue scan commands to the network and to retrieve values from the instruments.

To ease the job of the programmer, this work provides a software toolchain aimed at generating the PDL framework, compiling drivers for the hardware, linking dependability software to the framework and drivers and compiling the entire application to a format ready to use in simulation and FPGA emulation. The framework compiler is based on the Antlr4 project which allows automatic generation of parsers and interpreters. The Antlr4 tool needs a grammar specification of the PDL language and this is provided by the IJTAG standard. The compiler, build in Python, parses the PDL program and uses a template engine to generate C code. An overview is given in section 4.5 of the features of PDL supported by the implementation. Not all features are implemented due to the limited time available in this project. Among those features are the namespacing that is present in PDL programs. A tree-design is given as to how future work could handle this, as ICL to H-Array parsing is still a missing link in the IJTAG toolchain. The compiler was successfully tested with Bastion benchmark PDL files [23] together with a generated H-Array specification [26].

The framework relies on driver files for the Retargeting Engine. This driver software forms a Hardware Abstraction Layer that allows easy reuse of the Retargeting Engine IP on a different platform, using a different Retargeting Tools or cross compilation of the framework to other targets. The driver supports assisted generation of the H-Array and has method stubs for `iRead`, `iWrite` and `iApply`, the operations supported by the Retargeting Engine. The driver keeps track of the data structures suchs as the H-Array and the AR-Stack. It also features a memory mapped mapped return values array that can be used to extract the last read value of an instrument.

The base for the dependability manager is a RISC-V processor designed in VHDL. It is a simple Von Neumann-design with no instruction pipeline or caching. Hardware devices can be added as slaves of the system bus via Memory Mapped

IO. The processor features the Retargeting Engine designed by Ibrahim [2] to allow communication with the instrument network. This retargeting engine generates access vectors based on the current configuration of the network. It keeps track of access requests in multiple stacks to execute in the scan commands in parallel. The retargeting engine uses the hierarchy array as a network representation which easily shows the instruments in the active scan path. The retargeting process is implemented as a large state machine.

The processor also features a novel interrupt manager for IJTAG networks [3]. This research has also modified the SIB structure provided by the Bastion project to enable interrupt propagation in the scan network. The interrupt manager operates on strictly hierarchical tree networks and a simple example network is implemented in VHDL to test the interrupt localisation and servicing. The device is also memory mapped and a driver is created to attach interrupt service routines to a specific instrument. The interrupt manager allows the activation and deactivation of interrupt service routines to temporarily disable interrupts and this is also supported by the drivers.

The final step of the methodology used in the research asks that the complete system needs validation. Before that, the separate components of the DM are put to the test. The compiler was validated separately by using benchmark PDL files and parsing them to the C framework. This framework was compiled successfully and manually checked that it provided the necessary functionality. The drivers where also created, checked manually with GCC tools, and used in a complete system integration test to make sure that they operate on the system bus devices correctly. The processor and system devices are bench tested separately and finally the whole processor design was simulated with Modelsim and later emulated on an FPGA. The system ran a full instruction set test succesfully and three different programs where made to test the system. One to show output of the LEDs and check the operation of the Memory Mapped IO, one that uses the Mingle network from the benchmark set to check reading and writing to the network. It also executes the converted PDL test file of the benchmark set. The last uses the interrupt enabled simple test network to show output to the LEDs.

7.1 Future Work

After these integration tests the operation of the DM and its software has been proven. However, there are still some goals to be met to create a successful De-

pendability Management Platform.

A concious effort to optimize the size and design of the DM. At the beginning of this thesis it is mentioned that the incorporation of the DM is only viable if its design size is small compared to the functional layer. This research has made little attempt to create as small a design as possible. Optimizing the design should be focussed on the realisation in Silicon. This research took a best-effort approach regarding processor hardware emulation on an FPGA. The design of the retargeting engine is already minimized by using a state machine with multiple clock cycles for loading and storing data in a dedicated memory IP. Furthermore, the DM features no build-in-self-test or error correcting hardware. These techniques could improve its resilience and reliability. In order for the DM be effective, it should be more robust than the hardware it is supposed to be checking.

Implement the remainder of the PDL language features into the PDL2C compiler. The compiler design is described clearly in this report so that it can be reused in future work. The use of templates and separate listeners/generator classes provides a usable base to continue the framework compiler. As discussed, an important subset of PDL level-0 functions is implemented but to reach a level of maturity, the full language must be usable. This also includes full support of the namespacing of instruments as a Hierarchy Tree combined with the H-Array.

Converting the ICL specification to a usable H-Array and Hierarchy Tree. The process of converting a network specification to a H-Array is cumbersome and error-prone. It is also a large omission in the toolchain, that currently is based on a H-Array file without any meta-data. The Hierarchy Tree specification can also be tackled in the same process and it can store references to instruments in the H-Array. An effort is already made [2], [26] but the conversion of ICL to the selection dependency graph is not yet realised.

To conclude, the idea behind the Dependability Manager is a goal worth reaching. Reusing the already available IJTAG test network on a chip for life time reliability management and continuous testing will increase system dependability at a low price. A large openly available collection of tried-and-tested dependability applications could become usable by many engineers for a single platform. System reliability and dependability in general may become further standardised and more transparent to the end-user.

Bibliography

[1] M. F. Zakiy, “Hw-sw co-design of an on-chip ijtag dependability processor,” August 2016. [Online]. Available: http://essay.utwente.nl/70623/

[2] A. Ibrahim and H. G. Kerkhoff, “Analysis and design of an on-chip retargeting engine for ieee 1687 networks,” in *2016 21th IEEE European Test Symposium (ETS)*, May 2016, pp. 1–6.

[3] ——, “Efficient utilization of hierarchical ijtag networks for interrupts management,” in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Sept 2016, pp. 97–102.

[4] “IEEE standard for test access port and boundary-scan architecture,” IEEE, Tech. Rep., May 2013.

[5] “IEEE standard for access and control of instrumentation embedded within a semiconductor device,” IEEE, Tech. Rep., Dec 2014.

[6] D. Liu, *Embedded DSP Processor Design: Application Specific Instruction Set Processors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[7] A. A. Ucla, A. Avizienis, J. claude Laprie, and B. Randell, “Fundamental concepts of dependability,” 2001.

[8] D. Cheng and S. K. Gupta, “Ppb: Partially-working processors binning for maximizing wafer utilization,” in *2015 IEEE 33rd VLSI Test Symposium (VTS)*, April 2015, pp. 1–6.

[9] P. Mercati, F. Paterna, A. Bartolini, L. Benini, and T. S. Rosing, “Dynamic variability management in mobile multicore processors under lifetime constraints,” in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct 2014, pp. 448–455.

[10] C. Zhuo, D. Sylvester, and D. Blaauw, “Process variation and temperature-aware reliability management,” in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 580–585.

[11] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, June 2004, pp. 276–287.

[12] P. Mercati, A. Bartolini, F. Paterna, T. S. Rosing, and L. Benini, "Workload and user experience-aware dynamic reliability management in multicore processors," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–6.

[13] R. Berger, L. Burcin, D. Hutcheson, J. Koehler, M. Lassa, M. Milliser, D. Moser, D. Stanley, R. Zeger, B. Blalock, and M. Hale, "The rad6000mc system-on-chip microcontroller for spacecraft avionics and instrument control," in *2008 IEEE Aerospace Conference*, March 2008, pp. 1–14.

[14] J. Andersson, M. Hjorth, F. Johansson, and S. Habinc, "Leon processor devices for space missions: First 20 years of leon in space," in *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, Sept 2017, pp. 136–141.

[15] W. J. Broad, "For parts, nasa boldly goes . . . on ebay - the new york times," https://www.nytimes.com/2002/05/12/us/for-parts-nasa-boldly-goes-on-ebay.html, (Accessed on 08/19/2018).

[16] A. Ibrahim, "Test standards reuse for structured and cost-efficient dependability management of system-on-chips," Ph.D. dissertation, University of Twente, 4 2018, iDS Ph.D Thesis Series No. 18-461 ISSN: 2589-4730.

[17] H. G. Kerkhoff and X. Zhang, "Design of an infrastructural ip dependability manager for a dependable reconfigurable many-core processor," in *2010 Fifth IEEE International Symposium on Electronic Design, Test Applications*, Jan 2010, pp. 270–275.

[18] A. Ibrahim and H. G. Kerkhoff, "A cost-efficient dependability management framework for self-aware system-on-chips based on ieee 1687," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, July 2017, pp. 1–2.

[19] U. I. Gebremeskel and J. M. M. Ferreira, "A microprogrammed control path architecture for an embedded ieee 1149.1 test coprocessor," in *Design of Circuits and Integrated Systems*, Nov 2014, pp. 1–6.

[20] C. J. Cabral, "Design and implementation of an ieee 1149.7-compliant cjtag controller for debug and trace probe," December 2012. [Online]. Available: https://repositories.lib.utexas.edu/handle/2152/19988?show=full

[21] “Intel active management technology,” https://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html, (Accessed on 08/19/2018).

[22] T. Parr, “Antlr,” http://www.antlr.org/, (Accessed on 03/22/2018).

[23] A. Tertov, A. Jutman, S. Devadze, M. S. Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath, “A suite of ieee 1687 benchmark networks,” in *2016 IEEE International Test Conference (ITC)*, Nov 2016, pp. 1–10.

[24] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, “Reconfigurable scan networks: Modeling, verification, and optimal pattern generation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 2, pp. 30:1–30:27, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699863

[25] A. Ibrahim and H. Kerkhoff, “Structured scan patterns retargeting for dynamic instruments access,” in *2017 IEEE 35th VLSI Test Symposium (VTS)*. IEEE, 4 2017.

[26] F. Mansvelder, “Graph based automatic generation of an on-chip model for ijtag networks,” 2017.

[27] “Bastion - board and soc test instrumentation for ageing and no failure found,” http://fp7-bastion.eu/index.php, (Accessed on 03/16/2018).

[28] E. J. Marinissen, V. Iyengar, and K. Chakrabarty, “A set of benchmarks for modular testing of socs,” in *Proceedings. International Test Conference*, 2002, pp. 519–528.

[29] “Scanworks ijtag test — asset intertech,” https://www.asset-intertech.com/products/ijtag-test, (Accessed on 05/28/2018).

[30] “Tessent ijtag - mentor graphics,” https://www.mentor.com/products/silicon-yield/products/ijtag, (Accessed on 05/28/2018).

[31] “Siliconaid ijtag tool suite,” http://www.siliconaid.com/IJTAG_services.html, (Accessed on 05/28/2018).

[32] M. K. Jain, M. Balakrishnan, and A. Kumar, “Asip design methodologies: survey and issues,” in *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, 2001, pp. 76–81.

[33] “Centrifugal governor - wikipedia,” https://en.wikipedia.org/wiki/Centrifugal_governor, (Accessed on 08/20/2018).

[34] J. C. Maxwell *et al.*, “I. on governors,” *Proceedings of the Royal Society of London*, vol. 16, pp. 270–283, 1868.

[35] H. G. Kerkhoff, G. Ali, H. Ebrahimi, and A. Ibrahim, “An automotive mp-soc featuring an advanced embedded instrument infrastructure for high dependability,” *2017 International Test Conference in Asia (ITC-Asia)*, pp. 65–70, 2017.

[36] H. G. Kerkhoff, G. Ali, J. Wan, A. Ibrahim, and J. Pathrose, “Applying ijtag-compatible embedded instruments for lifetime enhancement of analog front-ends of cyber-physical systems,” in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2017, pp. 1–6.

[37] A. Ibrahim and H. G. Kerkhoff, “ijtag integration of complex digital embedded instruments,” in *2014 9th International Design and Test Symposium (IDT)*, Dec 2014, pp. 18–23.

[38] L. Zhang, Y. Han, Q. Xu, and X.-W. Li, “Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology,” pp. 891–896, 03 2008.

[39] J. R. Black, “Electromigration: A brief survey and some recent results,” *IEEE Transactions on Electron Devices*, vol. 16, no. 4, pp. 338–347, Apr 1969.

[40] “iphone slow and batteries: What’s going on with apple’s batteries, and how to get them replaced — the independent,” https://www.independent.co.uk/infact/iphone-slow-apple-battery-replacement-cost-price-how-to-explained-latest-a8159826.html, (Accessed on 08/20/2018).

[41] Imagination, “Mips architecture for programmers volume ii-a: The mips32 instruction set manual,” https://www.mips.com/products/architectures/mips32/, (Accessed on 03/14/2017).

[42] “Specifications - risc-v foundation,” https://riscv.org/specifications/, (Accessed on 03/14/2018).

[43] “Risc-v genealogy - risc-v foundation,” https://riscv.org/risc-v-genealogy/, (Accessed on 03/14/2018).

[44] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The rocket chip generator,” EECS Department, University of California,

Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[45] “freechipsproject/chisel3: Chisel 3,” https://github.com/freechipsproject/chisel3, (Accessed on 03/14/2018).

[46] “Befehlssatz.pdf,” http://ti.ira.uka.de/TI-2/Mips/Befehlssatz.pdf, (Accessed on 03/14/2018).

[47] “Mips32 architecture for programmers volume ii: The mips32 instruction set,” https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf, (Accessed on 03/14/2018).

[48] “Using the gnu compiler collection (gcc): Mips options,” https://gcc.gnu.org/onlinedocs/gcc/MIPS-Options.html, (Accessed on 03/14/2018).

[49] “Openrisc - openrisc,” https://openrisc.io/, (Accessed on 03/14/2018).

[50] “Openrisc 1200 soft processor — realtime embedded,” http://www.rte.se/blog/blogg-modesty-corex/openrisc-1200-soft-processor, (Accessed on 03/14/2018).

[51] “Overview of opensparc resources,” http://www.oracle.com/technetwork/systems/opensparc/index.html, (Accessed on 03/12/2018).

[52] “Opensparc t1,” http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html, (Accessed on 03/14/2018).

[53] “Sparcv9,” https://cr.yp.to/2005-590/sparcv9.pdf, (Accessed on 03/14/2018).

[54] L. Semiconductor, “Latticemico32 processor - lattice semiconductor,” http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx, (Accessed on 03/14/2018).

[55] “wyvernsemi/mico32: Latticemico32 instruction set simulator project,” https://github.com/wyvernSemi/mico32, (Accessed on 03/14/2018).

[56] D. E. Knuth, “Knuth: Mmix,” https://www-cs-faculty.stanford.edu/~knuth/mmix.html, (Accessed on 03/14/2018).

[57] ——, “fasc1.pdf,” http://mmix.cs.hm.edu/doc/fasc1.pdf, (Accessed on 03/14/2018).

[58] “Using the gnu compiler collection (gcc): Mmix options,” https://gcc.gnu.org/onlinedocs/gcc/MMIX-Options.html, (Accessed on 03/14/2018).

[59] “openrisc,” https://github.com/openrisc, (Accessed on 03/14/2018).

[60] “riscv/riscv-gcc,” https://github.com/riscv/riscv-gcc, (Accessed on 03/16/2018).

[61] “Binutils - gnu project - free software foundation,” https://www.gnu.org/software/binutils/, (Accessed on 03/12/2018).

[62] “The rust programming language,” https://www.rust-lang.org/en-US/, (Accessed on 03/16/2018).

[63] “Rust platform support the rust programming language,” https://forge.rust-lang.org/platform-support.html, (Accessed on 03/16/2018).

[64] “Misra - the motor industry software reliability association,” https://www.misra.org.uk/, (Accessed on 03/16/2018).

[65] “Jpl institutional coding standard,” https://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf, (Accessed on 03/16/2018).

[66] “Embeddedpython - python wiki,” https://wiki.python.org/moin/EmbeddedPython, (Accessed on 03/16/2018).

[67] “Raspberry pi teach, learn, and make with raspberry pi,” https://www.raspberrypi.org/, (Accessed on 08/21/2018).

[68] “Beagleboard.org - black,” https://beagleboard.org/black/, (Accessed on 08/21/2018).

[69] “Interfacing with external c code cython 0.28 documentation,” http://cython.readthedocs.io/en/latest/src/userguide/external_C_code.html, (Accessed on 03/16/2018).

[70] “Java se development kit 8 - downloads,” http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html, (Accessed on 03/16/2018).

[71] “Readytalk/avian: Avian is a lightweight virtual machine and class library designed to provide a useful subset of java’s features, suitable for building self-contained applications.” https://github.com/ReadyTalk/avian, (Accessed on 03/16/2018).

[72] “Java write directly to memory,” https://www.mkyong.com/java/java-write-directly-to-memory/, (Accessed on 03/16/2018).

[73] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[74] “Partcl - a tiny command language,” https://zserge.com/blog/tcl-interpreter.html, (Accessed on 03/22/2018).

[75] “zserge/partcl: Partcl - a micro tcl implementation,” https://github.com/zserge/partcl, (Accessed on 03/22/2018).

[76] “Gcc, the gnu compiler collection - gnu project - free software foundation (fsf),” https://gcc.gnu.org/, (Accessed on 03/16/2017).

[77] “Wishbone :: Opencores,” https://opencores.org/howto/wishbone, (Accessed on 04/12/2018).

[78] “Terasic - all fpga main boards - cyclone iv - de0-nano development and education board,” https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=593&PartNo=3, (Accessed on 05/30/2018).

[79] D. M. Merten, “maikmerten/riscv-tomthumb: A small risc-v rv32i core written in vhdl, intended as testbed for my personal vhdl learning,” https://github.com/maikmerten/riscv-tomthumb, (Accessed on 03/16/2018).

[80] “Introduction to qsys,” https://www.altera.com/support/training/course/oqsys1000.html, (Accessed on 05/31/2018).

[81] M. Murdocca and V. Heuring, *Computer Architecture and Organization: An Integrated Approach*. Wiley, 2007. [Online]. Available: https://books.google.nl/books?id=yUQ_AQAAIAAJ

[82] “Logic elements and logic array blocks in cyclone iv devices, cyclone iv device handbook volume 1, chapter 2.” https://www.altera.com/en_US/pdfs/literature/hb/cyclone-iv/cyiv-51002.pdf, (Accessed on 06/03/2018).

[83] “Fpga logic cells comparison,” http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf, (Accessed on 06/03/2018).

Appendix A

CSU Timing Diagram

This appendix contains some reference for the IJTAG scan elements. The first is a functional circuit diagram in Fig. A.1. The second is a timing diagram of a CSU cycle in Fig. A.2. Together they allow scanning of data through the boundary scan interface.

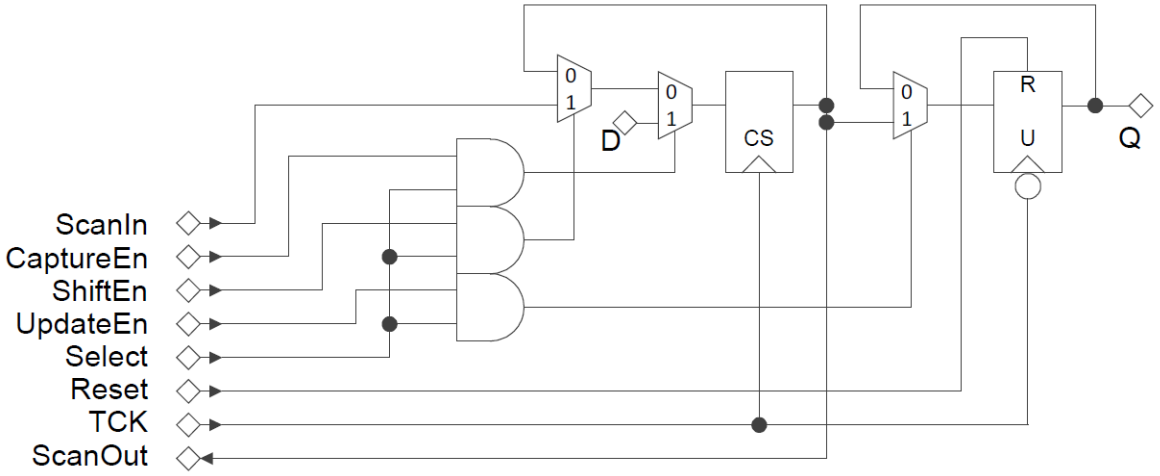


Figure A.1: The logic within a scan register taken from the IJTAG standard [5].

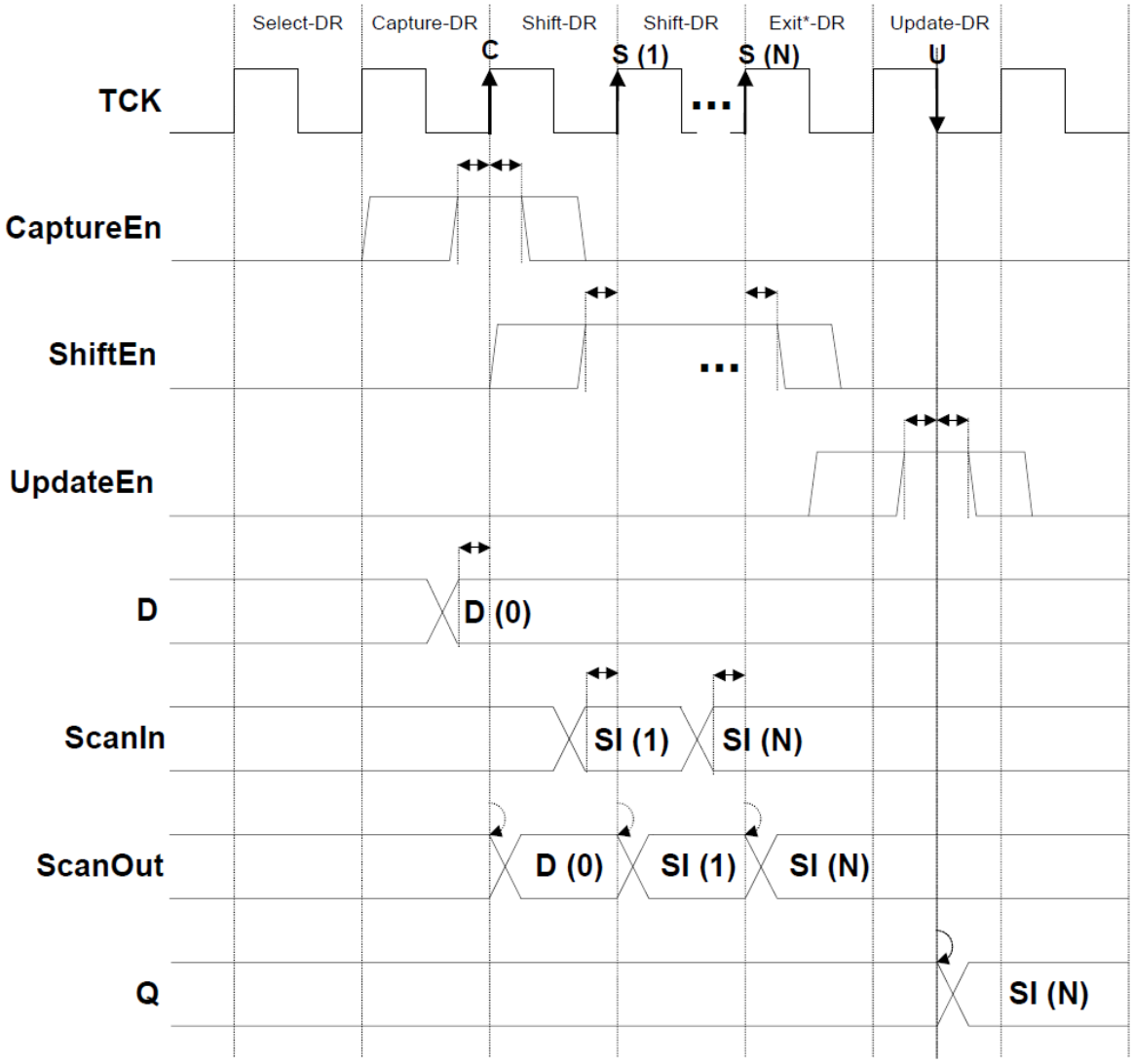


Figure A.2: CSU timing diagram taken from the IJTAG standard [5].

Appendix B

Retargeting Engine Memory

This appendix contains an overview of the memory used by the Retargeting Engine, see Fig B.1. The memory is an IP block from Altera that enables use of the dedicated memory cells of the FPGA. Its size is 1024 words of 32 bit, of which roughly half is used by the retargeting engine. The data-structures of the retargeting engine are parametrised to allow different sizes for testing purposes. The data types are always the size of a data-word so access is to the data is easier. The retargeting engine data is memory mapped with byte addresses to the system bus. The memory contains the H-Array with 64 entries, 32 AR-Stacks which hold 8 spaces, 64 Read Requests spaces and Read Values at the same size of the H-Array.

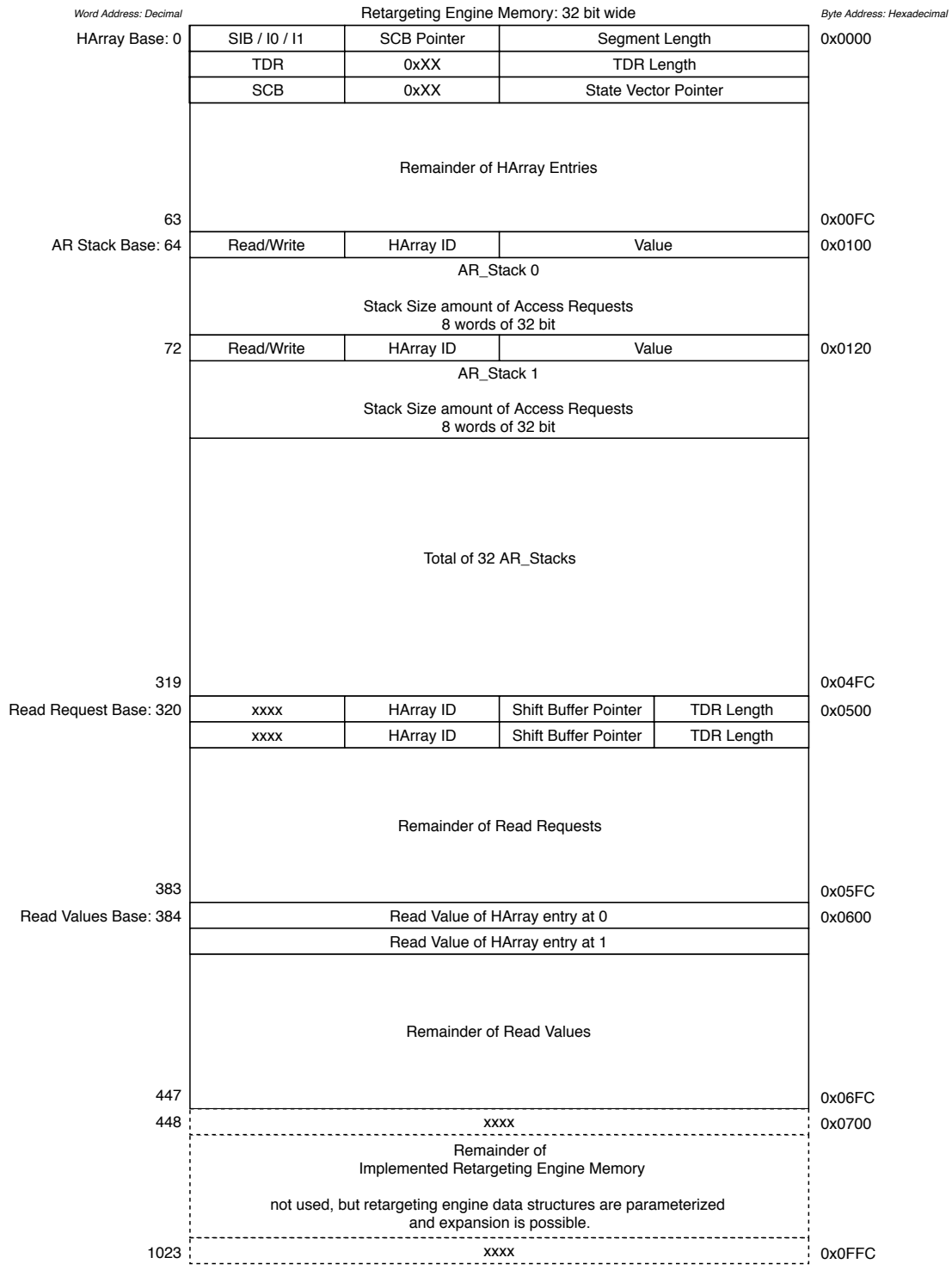


Figure B.1: Memory used by the Retargeting Engine, accessible via the System Bus.

Appendix C

The Mingle Network

This appendix contains resources related to the Mingle network from the Bastion benchmark test networks available online [23], [27].

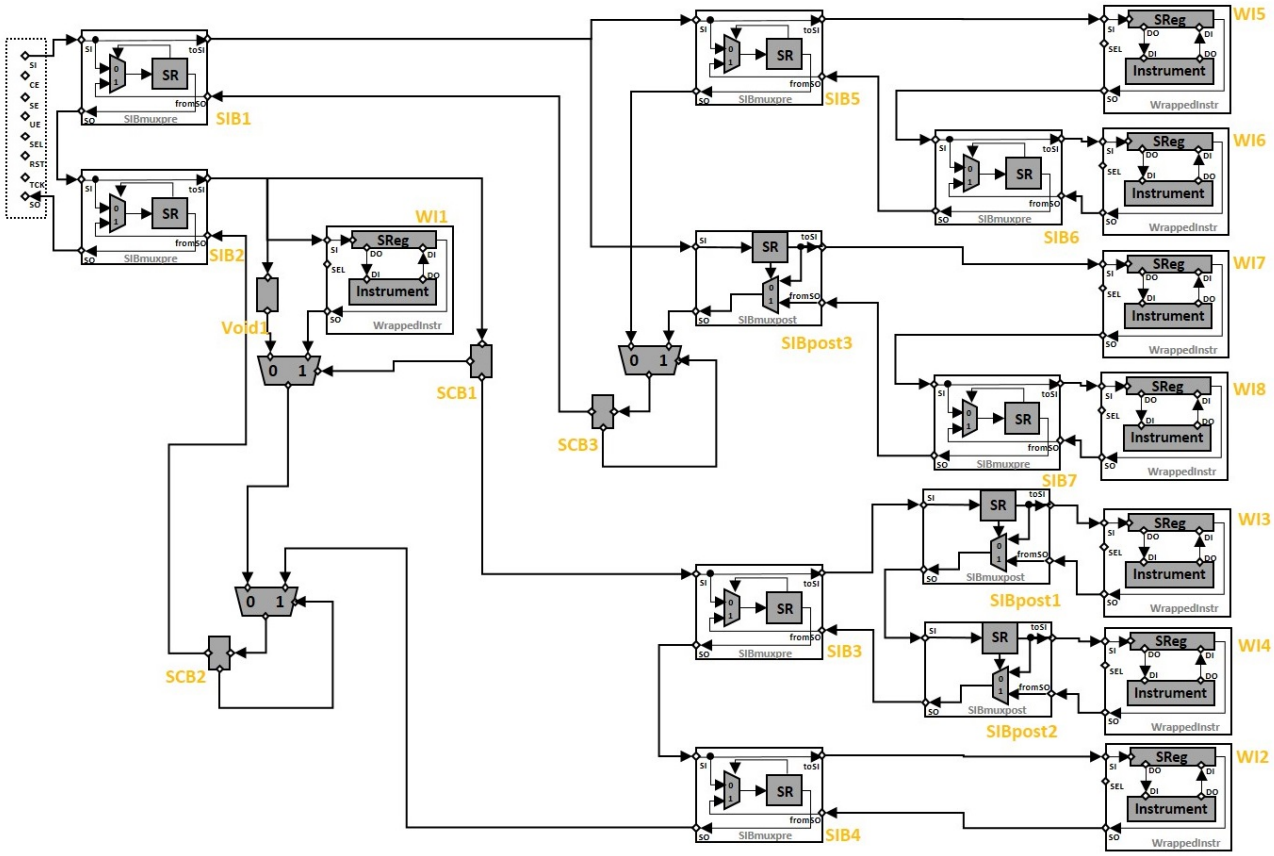


Figure C.1: A schematic overview of the network layout taken from the Bastion benchmark set [27].

Listing C.1: Newly created PDL file for the Mingle network.

```
# PDL file for the Mingle Bastion Benchmark network.
# Stephen Geerlings

iPDLLevel 0 -version STD_1687_2014;

iProcsForModule root::Mingle;


iProc iwrite_to_instrument { write_instrument value} {
    iWrite $write_instrument $value;
    iApply;
}

iProc iread_from_instrument { read_instrument } {
    iRead $read_instrument ;
    iApply;
}

iProc start_up { } {
    iWrite WI_1 11;
    iWrite WI_6 22;
    iWrite WI_5 33;
    iWrite WI_4 44;
    iWrite WI_2 55;
    iWrite WI_7 66;
    iWrite WI_3 77;
    iWrite WI_8 88;
    iApply;

}

iProc check_start_up { } {
    iRead WI_1 11;
    iRead WI_2 22;
    iRead WI_3 33;
    iRead WI_4 44;
    iRead WI_5 55;
    iRead WI_6 66;
    iRead WI_7 77;
    iRead WI_8 88;
    iApply;
}


iProc read_write_test { instrument} {
    iCall iwrite_to_instrument $instrument 0xAA;
    iCall iread_from_instrument $instrument;
}
```

Listing C.2: Generated header file for the framework.

```
#ifndef PDL_HARRAY_DEFINITIONS
#define PDL_HARRAY_DEFINITIONS
/**
*   This is a generated header file and library for executing PDL on the Dependability
    ↪ Manager Core
*   University of Twente, 2018
*   Stephen Geerlings
*   s.a.geerlings@alumnus.utwente.nl
*/

#define SIB_1 (0)
#define IO_SCB3 (1)
#define SIB_5 (2)
#define WI_5 (3)
#define SIB_6 (4)
#define WI_6 (5)
#define SCB_6 (6)
#define SCB_5 (7)
#define I1_SCB3 (8)
#define SCB_POST3 (9)
#define SIB_POST3 (10)
#define WI_7 (11)
#define SIB_7 (12)
#define WI_8 (13)
#define SCB_7 (14)
#define SCB_SCB3 (15)
#define SCB_1 (16)
#define SIB_2 (17)
#define IO_SCB2 (18)
#define IO_SCB1 (19)
#define VOID_1 (20)
#define I1_SCB1 (21)
#define WI_1 (22)
#define I1_SCB2 (23)
#define SCB_SCB1 (24)
#define SIB_3 (25)
#define SCB_POST1 (26)
#define SIB_POST1 (27)
#define WI_3 (28)
#define SCB_POST2 (29)
#define SIB_POST2 (30)
#define WI_4 (31)
#define SCB_3 (32)
#define SIB_4 (33)
#define WI_2 (34)
#define SCB_4 (35)
#define SCB_SCB2 (36)
#define SCB_2 (37)

void iwrite_to_instrument(int, int);
void iread_from_instrument(int);
void start_up();
void check_start_up();
void read_write_test(int);

/**
*
*   End of header
*   Enjoy!
*
*/
#endif
```

Listing C.3: Generated code file based on Listing C.1 (1/2).

```
/**
*   This is a generated library for executing PDL on the Dependability Manager IP Core
*   University of Twente, 2018
*   Stephen Geerlings
*   s.a.geerlings@alumnus.utwente.nl
*/

#include "../dm_lib/dm_re_lib.h"
#include "mingle_pdl.h"

/**
*   Autogenerated function
**/
void iwrite_to_instrument(int write_instrument, int value)
{
    /** iWrite $write_instrument $value **/
    iWrite(write_instrument, value);
    /** iApply **/
    iApply();
}

/**
*   Autogenerated function
**/
void iread_from_instrument(int read_instrument)
{
    /** iRead $read_instrument  **/
    iRead( read_instrument );
    /** iApply **/
    iApply();
}

/**
*   Autogenerated function
**/
void start_up()
{
    /** iWrite WI_1 11 **/
    iWrite(22, 11);
    /** iWrite WI_6 22 **/
    iWrite(5, 22);
    /** iWrite WI_5 33 **/
    iWrite(3, 33);
    /** iWrite WI_4 44 **/
    iWrite(31, 44);
    /** iWrite WI_2 55 **/
    iWrite(34, 55);
    /** iWrite WI_7 66 **/
    iWrite(11, 66);
    /** iWrite WI_3 77 **/
    iWrite(28, 77);
    /** iWrite WI_8 88 **/
    iWrite(13, 88);
    /** iApply **/
    iApply();
}
```

Listing C.4: Generated code file continued (2/2).

```
/**
*   Autogenerated function
**/
void check_start_up()
{
    /** iRead WI_1 11 **/
    iRead( 22 );
    /** iRead WI_2 22 **/
    iRead( 34 );
    /** iRead WI_3 33 **/
    iRead( 28 );
    /** iRead WI_4 44 **/
    iRead( 31 );
    /** iRead WI_5 55 **/
    iRead( 3 );
    /** iRead WI_6 66 **/
    iRead( 5 );
    /** iRead WI_7 77 **/
    iRead( 11 );
    /** iRead WI_8 88 **/
    iRead( 13 );
    /** iApply **/
    iApply();

}

/**
*   Autogenerated function
**/
void read_write_test(int instrument)
{
    iwrite_to_instrument( instrument,  170 );
    iread_from_instrument( instrument );
}

/**
*   End of library
*   Enjoy!
*
*/
```

Listing C.5: PDL file for the Mingle network from the Bastion project [23], [27].

```
# Date: 08_01_2018 17:58_51
# Version: 1_10_17
# Generated by Testonica Lab tools

# Modified to allow integration with identifiers in HArray
# Stephen Geerlings
#
# For example:
# ICL Path -> HArray ID
# WI1.reg8.SR -> WI_1
# SIBpost1.SR -> SCB_post1
# SIB7.SR -> SCB_7
#

iPDLLevel 0 -version STD_1687_2014;

iProcsForModule root::Mingle;

iProc all_scanregistes_in_one_iApply {} {
    iWrite SCB_post1 0b1;
    iWrite WI_1 0bx10001110110111101100101100011010;
    iWrite WI_6 0bx11101000011010011110110000011111;
    iWrite SCB_post2 0b1;
    iWrite SCB_post3 0b1;
    iWrite Void_1 0b1;
    iWrite WI_5 0bx11011111111100001000011100011110;
    iWrite SCB_7 0b1;
    iWrite WI_4 0bx10010111010110000011000000011101;
    iWrite SCB_6 0b1;
    iWrite SCB_4 0b1;
    iWrite SCB_5 0b1;
    iWrite SCB_3 0b1;
    iWrite SCB_2 0b1;
    iWrite WI_2 0bx10001100100011001111010000011011;
    iWrite SCB_1 0b1;
    iWrite WI_7 0bx10000111100101100101001000000;
    iWrite WI_3 0bx10110111011001111010100100011100;
    iWrite WI_8 0bx10011100101111011101010000100001;
    iWrite SCB_scb1 0b1;
    iWrite SCB_scb2 0b1;
    iWrite SCB_scb3 0b1;
    iApply;
    iRead SCB_post1 0b1;
    iRead WI_1 0bx10001110110111101100101100011010;
    iRead WI_6 0bx11101000011010011110110000011111;
    iRead SCB_post2 0b1;
    iRead SCB_post3 0b1;
    iRead Void_1 0b1;
    iRead WI_5 0bx11011111111100001000011100011110;
    iRead SCB_7 0b1;
    iRead WI_4 0bx10010111010110000011000000011101;
    iRead SCB_6 0b1;
    iRead SCB_4 0b1;
    iRead SCB_5 0b1;
    iRead SCB_3 0b1;
    iRead SCB_2 0b1;
    iRead WI_2 0bx10001100100011001111010000011011;
    iRead SCB_1 0b1;
    iRead WI_7 0bx10000111100101100101001000000;
    ...
```

Listing C.6: PDL file continued (2/3).

```
    iRead WI_3 0bx10110111011001111010100100011100;
    iRead WI_8 0bx10011100101111011101010000100001;
    iRead SCB_scb1 0b1;
    iRead SCB_scb2 0b1;
    iRead SCB_scb3 0b1;
    iApply;
}

iProc one_scanregister_per_iApply {} {
    iWrite SCB_post1 0b1;
    iApply;
    iRead SCB_post1 0b1;
    iApply;
    iWrite WI_1 0bx10001110110111101100101100011010;
    iApply;
    iRead WI_1 0bx10001110110111101100101100011010;
    iApply;
    iWrite WI_6 0bx11101000011010011110110000011111;
    iApply;
    iRead WI_6 0bx11101000011010011110110000011111;
    iApply;
    iWrite SCB_post2 0b1;
    iApply;
    iRead SCB_post2 0b1;
    iApply;
    iWrite SCB_post3 0b1;
    iApply;
    iRead SCB_post3 0b1;
    iApply;
    iWrite Void_1 0b1;
    iApply;
    iRead Void_1 0b1;
    iApply;
    iWrite WI_5 0bx11011111111100001000011100011110;
    iApply;
    iRead WI_5 0bx11011111111100001000011100011110;
    iApply;
    iWrite SCB_7 0b1;
    iApply;
    iRead SCB_7 0b1;
    iApply;
    iWrite WI_4 0bx10010111010110000001100000011101;
    iApply;
    iRead WI_4 0bx10010111010110000001100000011101;
    iApply;
    iWrite SCB_6 0b1;
    iApply;
    iRead SCB_6 0b1;
    iApply;
    iWrite SCB_4 0b1;
    iApply;
    iRead SCB_4 0b1;
    iApply;
    iWrite SCB_5 0b1;
    iApply;
    iRead SCB_5 0b1;
    iApply;
    iWrite SCB_3 0b1;
    iApply;
    iRead SCB_3 0b1;
    ...
```

Listing C.7: PDL file continued (3/3).

```
    iApply;
    iWrite SCB_2 0b1;
    iApply;
    iRead SCB_2 0b1;
    iApply;
    iWrite WI_2 0bx1000110010001100111010000011011;
    iApply;
    iRead WI_2 0bx1000110010001100111010000011011;
    iApply;
    iWrite SCB_1 0b1;
    iApply;
    iRead SCB_1 0b1;
    iApply;
    iWrite WI_7 0bx1000011110010110010100100000;
    iApply;
    iRead WI_7 0bx1000011110010110010100100000;
    iApply;
    iWrite WI_3 0bx1011011101100111101010010011100;
    iApply;
    iRead WI_3 0bx1011011101100111101010010011100;
    iApply;
    iWrite WI_8 0bx1001110010111101110101000100001;
    iApply;
    iRead WI_8 0bx1001110010111101110101000100001;
    iApply;
    iWrite SCB_scb1 0b1;
    iApply;
    iRead SCB_scb1 0b1;
    iApply;
    iWrite SCB_scb2 0b1;
    iApply;
    iRead SCB_scb2 0b1;
    iApply;
    iWrite SCB_scb3 0b1;
    iApply;
    iRead SCB_scb3 0b1;
    iApply;
}

iProc start_test {} {
    iCall all_scanregistes_in_one_iApply;
    iCall one_scanregister_per_iApply;
}
```

Listing C.8: Generated header file for the Mingle PDL.

```
#ifndef PDL_HARRAY_DEFINITIONS
#define PDL_HARRAY_DEFINITIONS
/**
*   This is a generated header file and library for executing PDL on the Dependability
    ↪ Manager Core
*   University of Twente, 2018
*   Stephen Geerlings
*   s.a.geerlings@alumnus.utwente.nl
*/

#define SIB_1 (0)
#define IO_SCB3 (1)
#define SIB_5 (2)
#define WI_5 (3)
#define SIB_6 (4)
#define WI_6 (5)
#define SCB_6 (6)
#define SCB_5 (7)
#define I1_SCB3 (8)
#define SCB_POST3 (9)
#define SIB_POST3 (10)
#define WI_7 (11)
#define SIB_7 (12)
#define WI_8 (13)
#define SCB_7 (14)
#define SCB_SCB3 (15)
#define SCB_1 (16)
#define SIB_2 (17)
#define IO_SCB2 (18)
#define IO_SCB1 (19)
#define VOID_1 (20)
#define I1_SCB1 (21)
#define WI_1 (22)
#define I1_SCB2 (23)
#define SCB_SCB1 (24)
#define SIB_3 (25)
#define SCB_POST1 (26)
#define SIB_POST1 (27)
#define WI_3 (28)
#define SCB_POST2 (29)
#define SIB_POST2 (30)
#define WI_4 (31)
#define SCB_3 (32)
#define SIB_4 (33)
#define WI_2 (34)
#define SCB_4 (35)
#define SCB_SCB2 (36)
#define SCB_2 (37)

void all_scanregistes_in_one_iApply();
void one_scanregister_per_iApply();
void start_test();

/**
*
*   End of header
*   Enjoy!
*
*/
#endif
```

Listing C.9: Generated code file based on Listing C.5 (1/6).

```
/**
*   This is a generated library for executing PDL on the Dependability Manager IP Core
*   University of Twente, 2018
*   Stephen Geerlings
*   s.a.geerlings@alumnus.utwente.nl
*/

#include "../dm_lib/dm_re_lib.h"
#include "mingle_pdl.h"


/**
*   Autogenerated function
*
**/
void all_scanregistes_in_one_iApply()
{
    /** iWrite SCB_post1 0b1 **/
    iWrite(26, 1);
    /** iWrite WI_1 0bx10001110110111101100101100011010 **/
    iWrite(22, 2396965658);
    /** iWrite WI_6 0bx11101000011010011111011000011111 **/
    iWrite(5, 1949627935);
    /** iWrite SCB_post2 0b1 **/
    iWrite(29, 1);
    /** iWrite SCB_post3 0b1 **/
    iWrite(9, 1);
    /** iWrite Void_1 0b1 **/
    iWrite(20, 1);
    /** iWrite WI_5 0bx11011111111100001000011100011110 **/
    iWrite(3, 3757082398);
    /** iWrite SCB_7 0b1 **/
    iWrite(14, 1);
    /** iWrite WI_4 0bx10010111010110000011000000011101 **/
    iWrite(31, 1269569565);
    /** iWrite SCB_6 0b1 **/
    iWrite(6, 1);
    /** iWrite SCB_4 0b1 **/
    iWrite(35, 1);
    /** iWrite SCB_5 0b1 **/
    iWrite(7, 1);
    /** iWrite SCB_3 0b1 **/
    iWrite(32, 1);
    /** iWrite SCB_2 0b1 **/
    iWrite(37, 1);
    /** iWrite WI_2 0bx10001100100011001111010000011011 **/
    iWrite(34, 589511195);
    /** iWrite SCB_1 0b1 **/
    iWrite(16, 1);
    /** iWrite WI_7 0bx10000111100101100101001000000 **/
    iWrite(11, 142173472);
    /** iWrite WI_3 0bx10110111011001111010100100011100 **/
    iWrite(28, 3077024028);
    /** iWrite WI_8 0bx10011100101111011101010000100001 **/
    iWrite(13, 2629686305);
    /** iWrite SCB_scb1 0b1 **/
    iWrite(24, 1);
    /** iWrite SCB_scb2 0b1 **/
    iWrite(36, 1);
    ...
```

Listing C.10: Generated code file continued (2/6).

```
    /** iWrite SCB_scb3 0b1 **/
    iWrite(15, 1);
    /** iApply **/
    iApply();
    /** iRead SCB_post1 0b1 **/
    iRead( 26 );
    /** iRead WI_1 0bx10001110110111101100101100011010 **/
    iRead( 22 );
    /** iRead WI_6 0bx11101000011010011111011000011111 **/
    iRead( 5 );
    /** iRead SCB_post2 0b1 **/
    iRead( 29 );
    /** iRead SCB_post3 0b1 **/
    iRead( 9 );
    /** iRead Void_1 0b1 **/
    iRead( 20 );
    /** iRead WI_5 0bx11011111111100001000011100011110 **/
    iRead( 3 );
    /** iRead SCB_7 0b1 **/
    iRead( 14 );
    /** iRead WI_4 0bx10010111010110000011000000011101 **/
    iRead( 31 );
    /** iRead SCB_6 0b1 **/
    iRead( 6 );
    /** iRead SCB_4 0b1 **/
    iRead( 35 );
    /** iRead SCB_5 0b1 **/
    iRead( 7 );
    /** iRead SCB_3 0b1 **/
    iRead( 32 );
    /** iRead SCB_2 0b1 **/
    iRead( 37 );
    /** iRead WI_2 0bx10001100100011001110100011011 **/
    iRead( 34 );
    /** iRead SCB_1 0b1 **/
    iRead( 16 );
    /** iRead WI_7 0bx10000111100101100101001000000 **/
    iRead( 11 );
    /** iRead WI_3 0bx10110111011001111010100100011100 **/
    iRead( 28 );
    /** iRead WI_8 0bx10011100101111011101010000100001 **/
    iRead( 13 );
    /** iRead SCB_scb1 0b1 **/
    iRead( 24 );
    /** iRead SCB_scb2 0b1 **/
    iRead( 36 );
    /** iRead SCB_scb3 0b1 **/
    iRead( 15 );
    /** iApply **/
    iApply();
}


/**
*   Autogenerated function
*
**/
void one_scanregister_per_iApply()
{
    /** iWrite SCB_post1 0b1 **/
    iWrite(26, 1);
    ...
```

Listing C.11: Generated code file continued (3/6).

```
/** iApply **/
iApply();
/** iRead SCB_post1 0b1 **/
iRead( 26 );
/** iApply **/
iApply();
/** iWrite WI_1 0bx10001110110111101100101100011010 **/
iWrite(22, 2396965658);
/** iApply **/
iApply();
/** iRead WI_1 0bx10001110110111101100101100011010 **/
iRead( 22 );
/** iApply **/
iApply();
/** iWrite WI_6 0bx11101000011010011111011000011111 **/
iWrite(5, 1949627935);
/** iApply **/
iApply();
/** iRead WI_6 0bx11101000011010011111011000011111 **/
iRead( 5 );
/** iApply **/
iApply();
/** iWrite SCB_post2 0b1 **/
iWrite(29, 1);
/** iApply **/
iApply();
/** iRead SCB_post2 0b1 **/
iRead( 29 );
/** iApply **/
iApply();
/** iWrite SCB_post3 0b1 **/
iWrite(9, 1);
/** iApply **/
iApply();
/** iRead SCB_post3 0b1 **/
iRead( 9 );
/** iApply **/
iApply();
/** iWrite Void_1 0b1 **/
iWrite(20, 1);
/** iApply **/
iApply();
/** iRead Void_1 0b1 **/
iRead( 20 );
/** iApply **/
iApply();
/** iWrite WI_5 0bx11011111111100001000011100011110 **/
iWrite(3, 3757082398);
/** iApply **/
iApply();
/** iRead WI_5 0bx11011111111100001000011100011110 **/
iRead( 3 );
/** iApply **/
iApply();
/** iWrite SCB_7 0b1 **/
iWrite(14, 1);
/** iApply **/
iApply();
/** iRead SCB_7 0b1 **/
iRead( 14 );
...
```

Listing C.12: Generated code file continued (4/6).

```
/** iApply **/
iApply();
/** iWrite WI_4 0bx10010111010110000011000000011101 **/
iWrite(31, 1269569565);
/** iApply **/
iApply();
/** iRead WI_4 0bx10010111010110000011000000011101 **/
iRead( 31 );
/** iApply **/
iApply();
/** iWrite SCB_6 0b1 **/
iWrite(6, 1);
/** iApply **/
iApply();
/** iRead SCB_6 0b1 **/
iRead( 6 );
/** iApply **/
iApply();
/** iWrite SCB_4 0b1 **/
iWrite(35, 1);
/** iApply **/
iApply();
/** iRead SCB_4 0b1 **/
iRead( 35 );
/** iApply **/
iApply();
/** iWrite SCB_5 0b1 **/
iWrite(7, 1);
/** iApply **/
iApply();
/** iRead SCB_5 0b1 **/
iRead( 7 );
/** iApply **/
iApply();
/** iWrite SCB_3 0b1 **/
iWrite(32, 1);
/** iApply **/
iApply();
/** iRead SCB_3 0b1 **/
iRead( 32 );
/** iApply **/
iApply();
/** iWrite SCB_2 0b1 **/
iWrite(37, 1);
/** iApply **/
iApply();
/** iRead SCB_2 0b1 **/
iRead( 37 );
/** iApply **/
iApply();
/** iWrite WI_2 0bx10001100100011001110100011011 **/
iWrite(34, 589511195);
/** iApply **/
iApply();
/** iRead WI_2 0bx10001100100011001110100011011 **/
iRead( 34 );
/** iApply **/
iApply();
/** iWrite SCB_1 0b1 **/
iWrite(16, 1);
...
```

Listing C.13: Generated code file continued (5/6).

```
    /** iApply **/
    iApply();
    /** iRead SCB_1 0b1 **/
    iRead( 16 );
    /** iApply **/
    iApply();
    /** iWrite WI_7 0bx10000111100101100101001000000 **/
    iWrite(11, 142173472);
    /** iApply **/
    iApply();
    /** iRead WI_7 0bx10000111100101100101001000000 **/
    iRead( 11 );
    /** iApply **/
    iApply();
    /** iWrite WI_3 0bx10110111011001111010100100011100 **/
    iWrite(28, 3077024028);
    /** iApply **/
    iApply();
    /** iRead WI_3 0bx10110111011001111010100100011100 **/
    iRead( 28 );
    /** iApply **/
    iApply();
    /** iWrite WI_8 0bx10011100101111011101010000100001 **/
    iWrite(13, 2629686305);
    /** iApply **/
    iApply();
    /** iRead WI_8 0bx10011100101111011101010000100001 **/
    iRead( 13 );
    /** iApply **/
    iApply();
    /** iWrite SCB_scb1 0b1 **/
    iWrite(24, 1);
    /** iApply **/
    iApply();
    /** iRead SCB_scb1 0b1 **/
    iRead( 24 );
    /** iApply **/
    iApply();
    /** iWrite SCB_scb2 0b1 **/
    iWrite(36, 1);
    /** iApply **/
    iApply();
    /** iRead SCB_scb2 0b1 **/
    iRead( 36 );
    /** iApply **/
    iApply();
    /** iWrite SCB_scb3 0b1 **/
    iWrite(15, 1);
    /** iApply **/
    iApply();
    /** iRead SCB_scb3 0b1 **/
    iRead( 15 );
    /** iApply **/
    iApply();
}
```

Listing C.14: Generated code file continued (6/6).

```
/**
*   Autogenerated function
*
**/
void start_test()
{
    all_scanregistes_in_one_iApply(  );
    one_scanregister_per_iApply(  );

}

/**
*
*   End of library
*   Enjoy!
*
*/
```

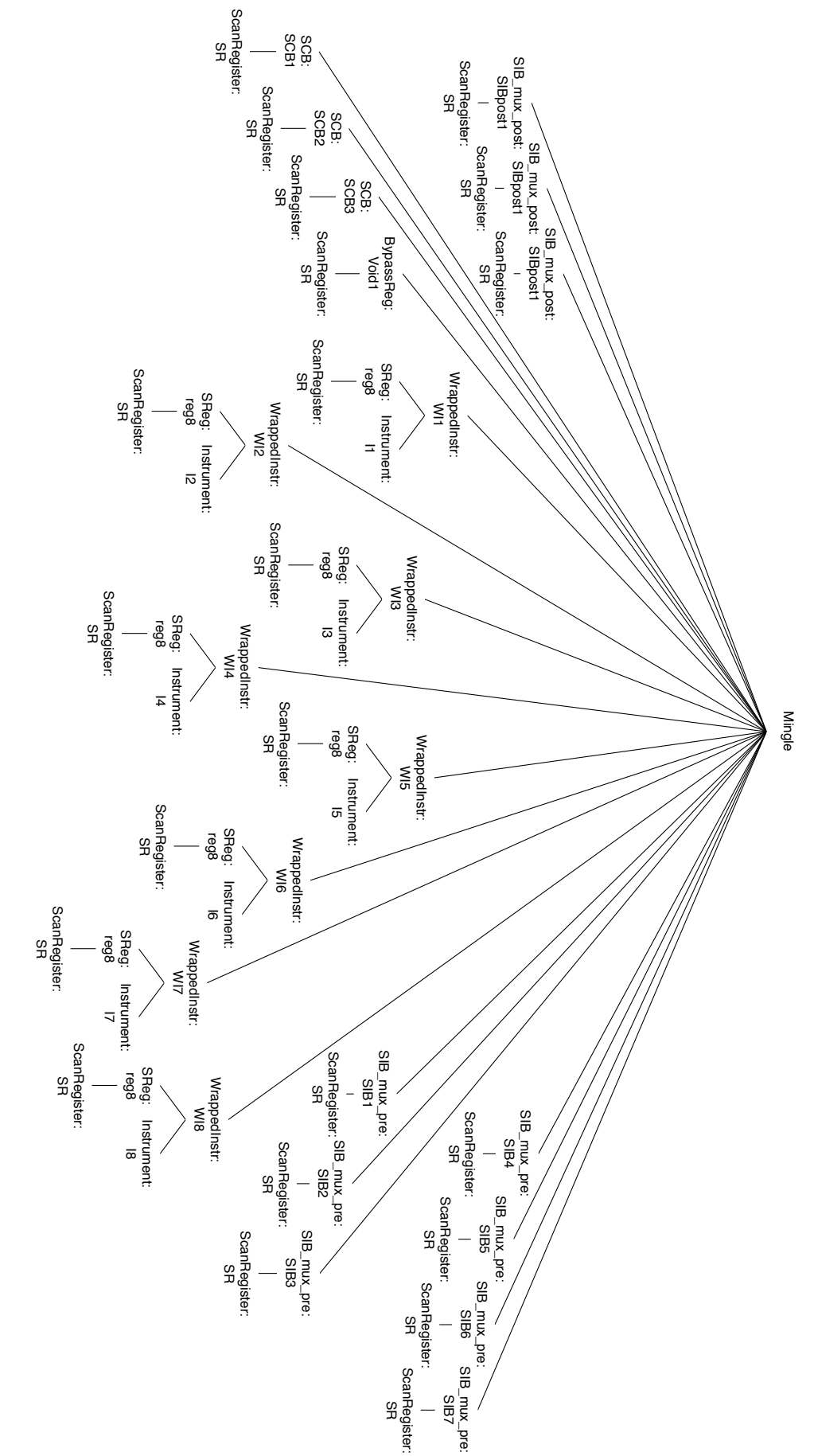


Figure C.2: The full hierarchy tree of the Mingle network.

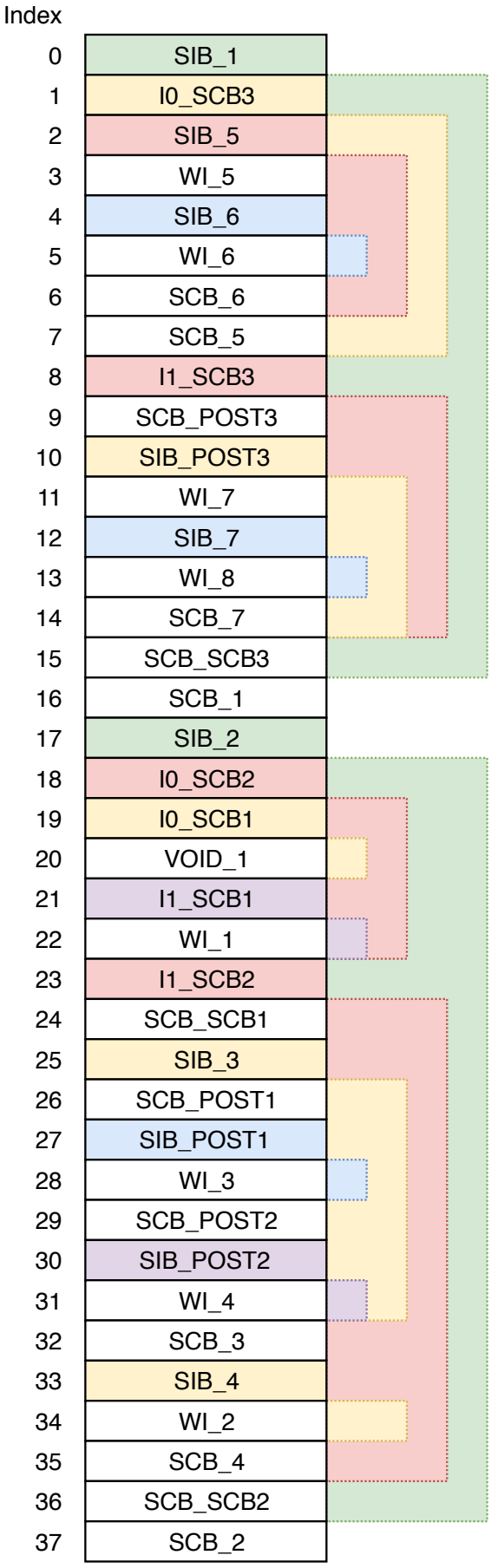


Figure C.3: The H-Array generated for the Mingle network [26].

Appendix D

Detailed Modelsim Testbenches

This chapter contains some detailed figures of Modelsim test benches discussed in section 5.5.

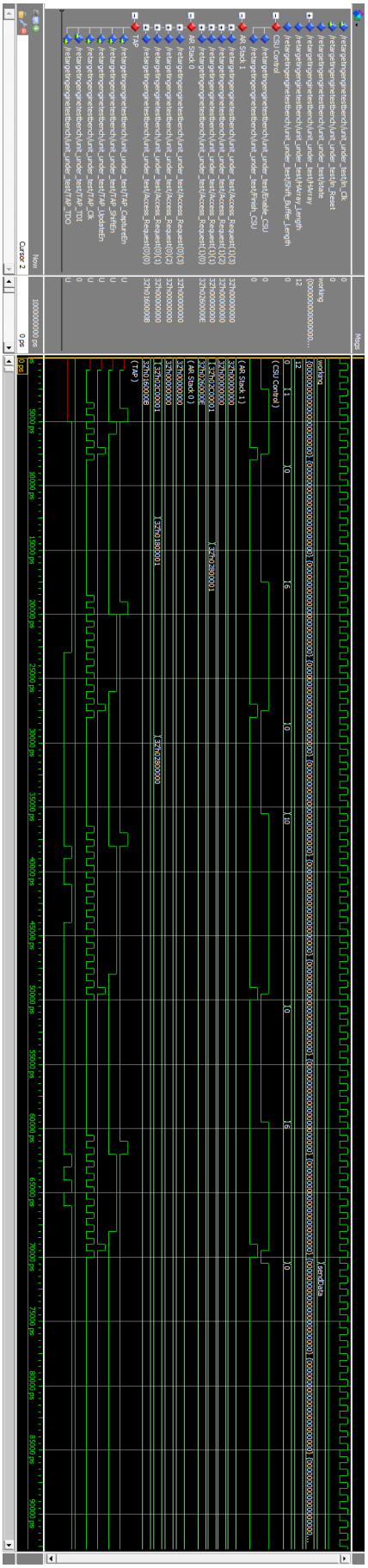


Figure D.1: Test Bench of the initial retargeting engine implementation.

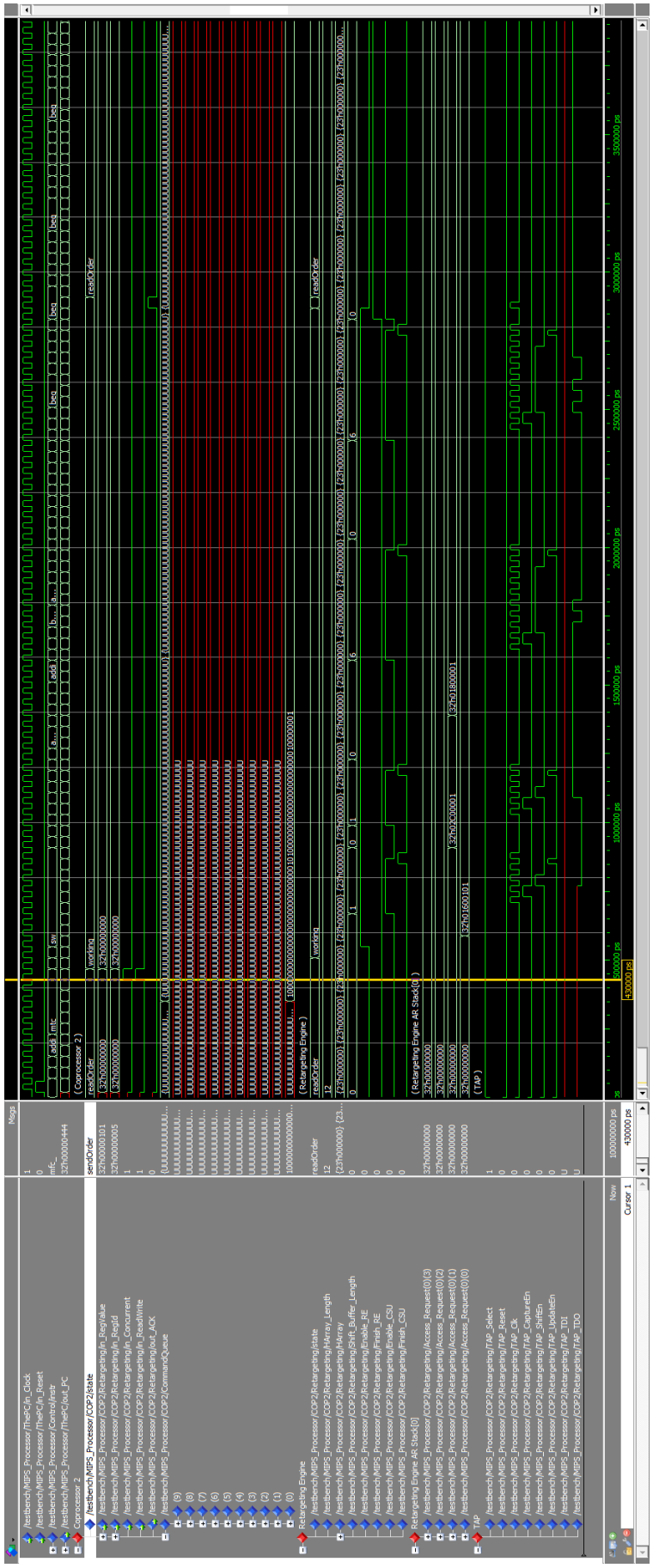


Figure D.2: Test Bench of the initial retargeting engine incorporated into the coprocessor-wrapper as designed by Zakiy [1].

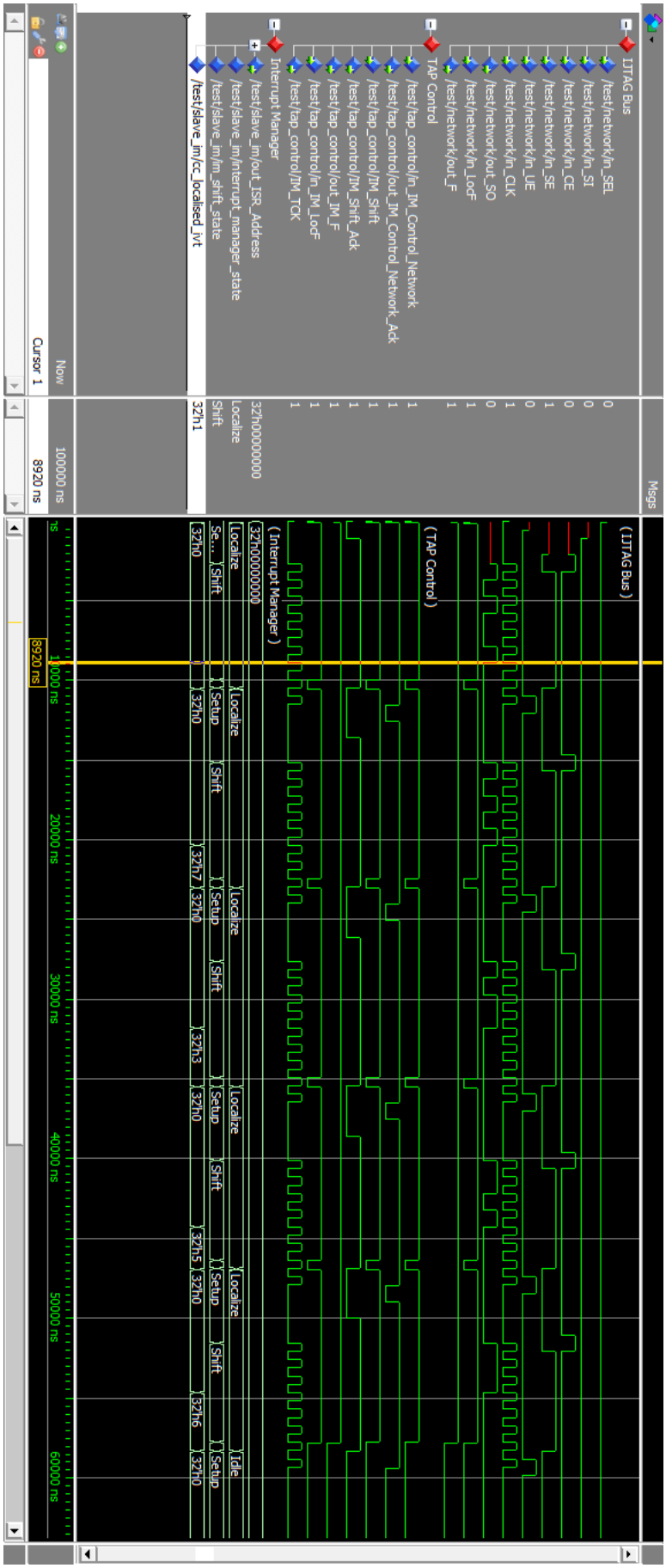


Figure D.3: Test Bench of the interrupt manager with the Simple network.

Appendix E

How to

This part of the Appendix tries to help anyone trying to reuse parts of this work. The tools needed encompass Altera Quartus, Modelsim by Mentor Graphics, Python3, GCC, Github and maybe Antlr4.

E.1 Access the IJTAG PDL Compiler Source?

The code of the PDL2C compiler is stored in a private repository at Github. Access can be requested by email or through Github. The link below points to the repository's wiki. The compilers directory has multiple folders, input programs can be stored in the *input* folder to be parsed. Note that the compiler relies on some Python3 packages which will need to be installed locally through Pythons package manager. The requirements can be found in the aptly named *pip_requirements* file.

`https://github.com/stephengeerlings/IJTAG-PDL-Compiler/wiki`

E.2 Use the PDL2C framework compiler?

After you are sure that you have Python3 installed along with all the packages in the requirements file, you are ready to start compiling your own framework. As input you must provide a PDL file along with a H-Array file with the same name. Examples are present in the *input* folder. The compiler, PDL2C.py, can then be run using Python along with the name of the file, e.g. `mingle_pdl`, you want to compile.

E.3 Access the Dependability Manager Source?

The hardware design files of the dependability manager are also stored in a private repository at Github. Access can be requested in the same manner, email or via Github. The link below points to the repository's wiki. The repository also contains the Retargeting Engine IP and Interrupt Manager code. The *software* folder contains the projects used for validation and the *synthesis* contains images ready to be programmed to the DE0-Nano FPGA.

`https://github.com/stephengeerlings/IJTAG-Dependability-Manager/wiki`

E.4 Use the RISC-V Compiler?

Go to the excellent tutorial made by the RISC-V project. The url is stated below. It can be used to build and install the GCC compiler for RISC-V. On Linux distributions this is easily accomplished, on Windows 10 the *Bash on Ubuntu for Windows* is a nice tool to have a Linux shell on your PC.

`https://github.com/riscv/riscv-gnu-toolchain`

E.5 Simulate the DM in ModelSim?

To successfully compile and run the Dependability Manager on Modelsim the user must create a project in ModelSim. The VHDL in the folder *design* contains all the files that together create a DM. The list of files needed is shown in Listing E.1. To load a program into the memory it is important that you move the `dm_system_memory.hex` that you have created into the ModelSim project folder. After that it should be possible to simulate the `tb_dm_main` entity.

E.6 Compile your own Dependability Application?

After you have installed the RISC-V GCC and are sure that Python3 is on the path, the `compile.sh` file can be used to create a dependability application loader file.

Listing E.1: List of the files that are needed in the Modelsim and Quartus projects.

```
./design/DM_Main_ALU.vhd
./design/DM_Main_Processor.vhd
./design/DM_Main_Registers.vhd
./design/dependability_manager.vhd
./design/dm_harray_pkg.vhd
./design/dm_instruction_memory.vhd
./design/dm_interrupt_manager.vhd
./design/dm_interrupt_manager_pkg.vhd
./design/dm_main_alu_in_mux.vhd
./design/dm_main_alu_out_mux.vhd
./design/dm_main_alu_pkg.vhd
./design/dm_main_base_pkg.vhd
./design/dm_main_control.vhd
./design/dm_main_control_pkg.vhd
./design/dm_main_ir_decode.vhd
./design/dm_retargeting_engine.vhd
./design/dm_retargeting_engine_improved_arch.vhd
./design/dm_retargeting_engine_memory.vhd
./design/dm_system_bus_leds.vhd
./design/dm_system_bus_master.vhd
./design/dm_system_bus_pkg.vhd
./design/dm_system_bus_slave.vhd
./design/dm_system_memory.vhd
./design/dm_tap_control.vhd
./design/ijtag/Instruments.vhd
./design/ijtag/Interrupt/ExtendedInstruments.vhd
./design/ijtag/Interrupt/ExtendedSIB.vhd
./design/ijtag/Interrupt/Simple/simple.vhd
./design/ijtag/NetworkStructs.vhd
./design/ijtag/Primitives.vhd
./design/runs/run_basicscb_re1.vhd
./design/runs/run_basicscb_re2.vhd
./design/runs/run_basicscb_re3.vhd
./design/runs/run_blink.vhd
./design/runs/run_mingle_demo.vhd
./design/runs/run_mingle_re1.vhd
./design/runs/run_mingle_re2.vhd
./design/runs/run_mingle_re3.vhd
./design/runs/run_simple_difficult.vhd
./design/runs/run_simple_dm.vhd
./design/runs/run_simple_interrupt.vhd
./design/runs/run_treebalanced_re1.vhd
./design/runs/run_treebalanced_re2.vhd
./design/runs/run_treebalanced_re3.vhd
./design/runs/run_treeflat_ex_re1.vhd
./design/runs/run_treeflat_ex_re2.vhd
./design/runs/run_treeflat_ex_re3.vhd
./design/runs/run_treeflat_re1.vhd
./design/runs/run_treeflat_re2.vhd
./design/runs/run_treeflat_re3.vhd
./design/runs/run_treeunbalanced_re1.vhd
./design/runs/run_treeunbalanced_re2.vhd
./design/runs/run_treeunbalanced_re3.vhd
```

E.7 Emulate the DM with Quartus?

The instructions for the simulation of the DM also apply to the Altera Quartus environment. A new project can be started and the code of the project can be imported into the project. The program that will run on the DM must be moved the `dm_system_memory.hex` into the Quartus project folder. After providing the pinout to the Clock, Reset and other pins that it the `fpga_dm_main` needs it should be possible to compile and emulate the entity.