

Kristoffer Venæs Monsen

Challenges and solutions in creating a RISC-V computing platform

Master's thesis in Computer Science

Supervisor: Magnus Själander

June 2019

Kristoffer Venæs Monsen

Challenges and solutions in creating a RISC-V computing platform

Master's thesis in Computer Science
Supervisor: Magnus Sjölander
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

 **NTNU**
Norwegian University of
Science and Technology

*Dedicated to everyone who contribute countless hours to open software and hardware,
for no other gain than the betterment of the modern world.*

Summary

In this thesis, we detail the implementation of a RISC-V ISA centered computing system. By reusing existing IP cores in Cobham Gaisler's open-source GRLIB IP core library and its sample board designs, we successfully construct a computing platform with the existing PicoRV RISC-V processor, implemented on an FPGA on the Xilinx ZC702 development kit.

The GRLIB IP core library is originally intended for the SPARC V8 ISA, which creates some architectural challenges. We show how to solve the main architectural issues, namely system bus interfacing, interrupt handling, and mixed-endian hardware. We also show how to verify and test the system during development, and during final prototyping. To drive the computing system, we have derived firmware for the target processor based on existing works. The firmware can initialize GRLIB IP peripherals on the system, and communicate with the user through the system's UART module. For a user to communicate with the system, we develop a JTAG application, which uses the ZC702's JTAG scan-chain in order to communicate with the RISC-V system.

Finally, we discuss the shortcomings we identified in the process. Great care must be taken when designing the verification process and corresponding verification methods, as failing to do so will scale poorly as the design complexity increases. This is especially true when the open-source hardware development software situation is found to be rather lacking, which posed challenges in our development process. We believe that this tooling issue will improve over time, as we see a growing interest in open-source hardware.

Oppsummering

I denne masteroppgaven beskriver vi implementasjonen av et datasystem som baserer seg på RISC-V instruksjonssettet. Ved å gjenbruke eksisterende maskinvaremoduler fra Cobham Gaislers åpent tilgjengelige *GRLIB IP core library* og dets eksempeldatasystemer, klarer vi å konstruere en plattform med den eksisterende PicoRV RISC-V prosessoren, implementert på en FPGA på en Xilinx ZC702 utviklingsplattform.

GRLIB er egentlig ment for SPARC V8 instruksjonssettet, som skaper noen utfordringer. Vi viser hvordan man løser hovedutfordringene på datamaskinarkitektursiden, spesifikt oppkobling mot systembussen, håndtering av systemvarsler ("interrupts"), og hvordan man korrekt håndterer variasjon i hvordan data lagres i hovedminnet ("endianness"). Vi viser også hvordan man kan teste og verifisere systemet under utvikling, og når man skal teste den endelige versjonen når alt kjører. For å vise hva datasystemet kan gjøre, har vi utviklet systemprogramvare for prosessoren basert på eksisterende verk. Programvaren kan initialisere GRLIB moduler i systemet, og kommunisere med brukeren gjennom systemets UART modul. For at en bruker skal kunne kommunisere med systemet, har vi også utviklet en JTAG applikasjon, som bruker utviklingsplattformens JTAG søkekjede for å kunne kommunisere med RISC-V systemet.

Til slutt diskuterer vi mangler og forbedringer som kan gjøres i prosessen vår. Man bør tenke grundig gjennom verifiseringsprosessen og dens metoder, da mangler på dette fører til en utviklingsprosess som skalerer dårlig med økt kompleksitet i maskinvaren man skaper. Dette er enda viktigere siden mengden åpne og offentlig tilgjengelige maskinvareutviklingsapplikasjoner er ganske liten, som fører til ekstra utfordringer i utviklingsprosessen. Vi tror at mengden applikasjoner vil forbedre seg over tid, nå som åpen og allmenn tilgjengelig maskinvare blir stadig mer aktuelt.

Acknowledgements

Thanks to Geir Kulia for valuable support and resources.

I'm also very grateful for valuable insight, thoughts, comments, input, and the original problem statement from my supervisor Magnus Sjalander. Thanks for lending me the Xilinx ZC702 so that an actual implementation became very possible, which was a huge motivating factor.

Many thanks to Edgar Vedvik for explaining the most important points in relation to bare-metal programming, and sharing his sources.

Thanks to Finn Inderhaug and Trude Telle for proof-reading the manuscript.

Much credit is due to the original works this work depends upon: The GNU software toolchain, Cobham Gaisler's GRLIB IP core library, Revanth Kamaraj's FreeAHB, Clifford Wolf's PicoRV core and Stephen William's Icarus Verilog. We wouldn't be able to get as far without these works being shared openly.

Finally, thanks to all family and friends throughout all the years for all the support and getting me to this point.

Table of Contents

Summary	i
Oppsummering	ii
Acknowledgements	iii
Table of Contents	vii
Abbreviations	viii
1 Introduction	1
1.1 Motivation	2
1.2 The need for open-source computer architecture	3
1.3 Our approach	4
1.4 Thesis structure	5
2 Background	7
2.1 The hardware development flow	8
2.2 The state of available hardware development tools	10
2.3 Hardware porting challenges	12
2.4 RISC-V basics	15
2.5 GRLIB overview	20
2.6 A survey of other major, existing open source ISAs	24
2.7 State of the art RISC-V platforms	26
3 Methodology	29
3.1 Work structure	30
3.2 Goal	30
3.3 A note on measurements and reporting	31
3.4 Development methodology	31
3.5 Development setup	32

3.6	Custom notation	35
3.7	Utilized software	36
4	Implementation	39
4.1	Choosing a RISC-V core and attaching an AMBA AHB master to it . . .	40
4.2	Studying the AMBA AHB bus protocol and deriving a Finite State Machine	48
4.3	Finding an AMBA AHB master	59
4.4	Fusing together PicoRV and FreeAHB	62
4.5	Verifying the design so far	64
4.6	Implementing a base GRLIB design on an FPGA	74
4.7	Adding the RISC-V toplevel to the design	81
4.8	Implementation debugging	84
4.9	Creating a JTAG application for UART communication	93
4.10	Writing dedicated firmware	98
5	Results	111
5.1	Final design	111
5.2	Memory organization	113
5.3	Building and running the prototype	114
6	Discussion	117
6.1	Evaluating the porting process	117
6.2	The importance of development tools for productivity	120
6.3	The importance of a good verification infrastructure	122
6.4	Future works	129
7	Conclusion	131
	Appendices	141
A	A review of software licences	143
A.1	Copyright versus copyleft	143
A.2	Permissive, copyleft and hybrid approaches	143
A.3	The licences involved in this project	144
B	Application software porting issues	145
B.1	Software layer organization	145
B.2	Software porting techniques	146
C	Original version of the FreeAHB master	149
D	The PicoRV to FreeAHB adapter	157
E	UartMonitor	163
F	RISC-V Verilog toplevel to GRLIB system VHDL Wrapper	173

Abbreviations

ABI	=	Application Binary Interface
AMBA	=	Advanced Microcontroller Bus Architecture
AMBA AHB	=	AMBA Advanced High-performance Bus
AMBA APB	=	AMBA Advanced Peripheral Bus
AMBA AXI	=	AMBA Advanced Extensible Interface
API	=	Application Programming Interface
APSOC	=	Application Programmable SoC
ASCII	=	American Standard Code for Information Interchange
ASIC	=	Application-Specific Integrated Circuit
AST	=	Abstract Syntax Tree
BAR	=	Bank Address Register
BASH	=	Bourne Again SHell
BCC	=	Bare-C Compiler
CHISEL	=	Constructing Hardware In a Scala Embedded Language
CMS	=	Code-Morphing System
CP	=	CoProcessor
CPU	=	Central Processing Unit
CTS	=	Clear To Send
DAP	=	Debug Access Port
DDR	=	Double Data Rate
DIP	=	Dual Inline Package
DP	=	Double-Point (regarding decimal precision)
DR	=	Data Register
DSP	=	Digital Signal Processing
DSU	=	Debug Support Unit
DUV	=	Device Under Verification
EBNF	=	Extended Backus-Naur Form
EDA	=	Electronic Design Automation
FIFO	=	First In, First Out
FIRRTL	=	Flexible Intermediate Representation for RTL
FP	=	Floating Point
FPGA	=	Field-Programmable Grid Array
FPU	=	Floating Point Unit
FSM	=	Finite State Machine
GCC	=	GNU Compiler Collection
GDB	=	The GNU project debugger
GNU	=	GNU's Not Unix
GNU GPL	=	GNU General Public Licence
GPIO	=	General Purpose Input (and) Output

HAL	=	Hardware Abstraction Layer
Hart	=	Hardware thread
HDL	=	Hardware Definition Language
IDE	=	Integrated Development Environment
IE	=	Interrupt Enable
IEEE	=	Institute of Electrical and Electronics Engineers
IoT	=	Internet of Things
IP (when discussing interrupts)	=	Interrupt Pending
IP (when discussing works)	=	Intellectual Property
IR	=	Instruction Register
IRQ	=	Interrupt request
ISA	=	Instruction Set Architecture
IU	=	Integer Unit
I/O	=	Input/Output
JTAG	=	Joint Test Action Group
LED	=	Light-Emitting Diode
LTS	=	Long-Term Support
LUT	=	LookUp Table
L1	=	Level 1 (cache)
L2	=	Level 2 (cache)
MMU	=	Memory Management Unit
MPI	=	Message Passing Interface
MPL	=	Mozilla Public Licence
OCI	=	On-Chip Interconnect
OoO	=	Out-of-Order
OS	=	Operating System
PCI	=	Peripheral Component Interconnect
PLIC	=	Platform-Level Interrupt Controller
PMD	=	Personal Mobile Device
PPA	=	Power, Performance and Area
PS	=	Processing System
QP	=	Quadruple-Point (regarding decimal precision)
RAM	=	Random-Access Memory
RCG	=	Rocket Chip Generator
RISC	=	Reduced Instruction Set Computer
RoCC	=	Rocket Custom Coprocessor
ROM	=	Read-Only Memory
RTL	=	Resistor-Transistor Logic
RTS	=	Request To Send
SDK	=	Software Development Kit
SMP	=	Symmetric MultiProcessing
SoC	=	System on a Chip
SP	=	Single-Point (regarding decimal precision)
SPARC	=	Scalable Processor ARChitecture
SRMMU	=	SPARC Reference MMU

SW	=	Software (depending on context)
SW	=	Switch (depending on context)
TAP	=	Test Access Port
TDI	=	Test Data Input
TDO	=	Test Data Output
TDP	=	Thermal Design Power
TMS	=	Test Mode Select
TSO	=	Total Store Ordering
UART	=	Universal Asynchronous Receiver/Transmitter
UI	=	User Interface
USB	=	Universal Serial Bus
VCD	=	Value Change Dump
VHDL	=	VHSIC Hardware Description Language
VHSIC	=	Very High Speed Integrated Circuit
VLSI	=	Very Large Scale Integration
VM	=	Virtual Machine
WSC	=	Warehouse-Scale Computer

Introduction

RISC-V is, without a doubt, gaining traction. From the growing amount of taped-out RISC-V based designs [1] [2], industry-wide interest (and dissent [3]), to a rising number of new actors such as Shakti [4], SiFive [5], and lowRISC [6] pushing forward for an open source architecture to the mainstream, it becomes clear that the RISC-V ISA might be a real potential alternative in today's market. In a computing world which is almost entirely dominated by x86, and where other large actors such as Arm push for their ISA as the next market dominator in all levels of computing [7], it is fair to ask the question of whether RISC-V is fit for fight.

A processor is arguably useless without any computing platform to run on. Arm and x86 have gained large followings throughout the years, and thus much development has gone into compatible computing platforms and software designed with these ISAs in mind. Arm has, up until the time of writing, mainly dominated the mobile market with energy efficient CPU designs – over 9 billion ARMv7-based chips in 2013 [8, ch.2, p.152]. Manufacturers licence Arm IP to aid in their design of specialized applications [8, ch.3, p.346]. x86 is dominant in both the server and desktop market with an estimated 350M chips produced in 2013 [8, ch.2, p.152], thanks to a strong focus on backwards compatibility, reducing software costs, and benefiting greatly from the value and size of its ecosystem [8].

RISC-V [9], on the other hand, is a fairly new ISA, which is just beginning to gain traction. Much effort is placed in innovations such as a new HDL language (CHISEL [10]), a parameterizable chip generator (Rocket Chip Generator [2]), and a new open system bus protocol (TileLink [11]). Still, the RISC-V ecosystem suffers from the lack of open tools and components required in order to construct a platform. We believe it would be significantly beneficial for both RISC-V, and perhaps even for computer architecture as a whole, to further improve and democratize the hardware development process. We seek to answer:

1. Can one easily derive a RISC-V driven platform by using existing works?
2. How can the development process be improved?

1.1 Motivation

We wish to strengthen primarily open-source architecture, and consequentially the entire architecture environment as a whole.

The vast performance improvements we have seen in the past decades in computing are starting to decline [12, p.3]. This is due to a magnitude of factors, including the memory wall [13], the collapse of Moore's law (ultimately due to the decreased gains of Dennardian scaling and the closely related power wall [13]), and increasing amounts of dark silicon on a chip [14]. This in turn means more efforts are required in computer architecture in order to address the stagnating performance increases. Facing a steadily energy-efficient and more heterogenous computing future [13], we believe that computer architecture as a whole will thrive in the face of increased competition between multiple architecture vendors, pushing the actors to constantly innovate in order to gain traction, instead of resting on its market dominance.

In addition, we believe that all aspects of computing should be open – in order to easily allow new actors with minor resources into existing segments of computing, and to benefit from existing works. The authors believe that, historically, science and technology has progressed by standing on the shoulders of giants, not by constantly reinventing the wheel. Previously, the scientific community has urged for more accurate and modular simulators to enable research [15], but we believe it is important to also ease the following real-world implementation.

1.2 The need for open-source computer architecture

Further, we believe an open-source architecture is needed for three reasons:

- **Auditability.** Very recently the first series of SHAKTI processors [4], sponsored by the government of India, have surfaced. Based on RISC-V [16], the project states a fear of security backdoors in proprietary processors to be one of the main motivations for strengthening open-source ISAs [4]. Security aside, audability and customizability is also a great asset when designing high performance systems in which all aspects of the design needs to be known. The European Processor Initiative (EPI) plans to build a exascale computer within the next decade [17], and it sees the RISC-V ISA as a potential candidate for being a core technology for accelerators within such a computer due to its open and customazible nature [18].
- **Lower economic barrier of entry.** Proprietary ISAs are expensive to license, usually forces the application to conform to the inflexible ISA, and is arguably ill-suited in an increasingly specialized computing world, where overall cost to market is a constant driving factor. The founders of RISC-V argue that *“the case is even clearer for an open ISA than for an open OS, as ISAs change very slowly”* [19]. Furthermore, the founders of RISC-V state the risk of the proprietary ISA company going bankrupt, the rise of CPUs on almost all SoCs, the rise of the internet of things (IoT), increasing complexity of mainstream ISAs (often due to backwards compatibility support in the case of x86 [8, ch.2]), and the licensing troubles associated with proprietary solutions, as reasons for why the time is right for a true, major open-source alternative [19].
- **Enabling physical computer architecture prototypes.** While the trend within computer architecture in recent years have moved towards utilizing simulators in order to propose new designs and techniques [15], we believe that physical prototyping, measuring and experimentation will be of greater significance in the future. While simulators aid in testing computer architecture concepts with idealistic clock cycles and perfect signal propagation, physical prototypes aid in getting a feel of issues “in the field of battle” (clock propagation, voltage leakage, routing constraints, available area, and other electrical engineering phenomena), which we believe can be just as important to address if one wishes to take a successful simulated concept into the real world.

Critics of RISC-V and other non-dominant ISAs raise some heavily debated concerns, which we nevertheless believe are important to address. Arm once argued against designing a custom SoC with RISC-V, citing cost, lack of maturity in the market, lack of thorough testing through widespread use, and the lack of good validation tools as arguments [3]. Others such as Linux creator Linus Torvalds argue from the software perspective that Arm and other architectures will never succeed due to issues with cross compilation, and due to the sheer magnitude of the x86 ecosystem [20].

1.3 Our approach

We wish to take an existing, arbitrary RISC-V core, and construct an entire RISC-V powered computing platform out of it by porting Cobham Gaisler's IP core library [21], which is originally targeting the SPARC V8 architecture.

This requires addressing multiple issues on the architectural level, namely endianness, address space, driver software translation, interrupt handling, and dealing with the fact that system modules (timers, memory controllers, UARTs) also make certain assumptions about the connected processors. We will also be dealing with issues such as mixed language on the HDL level, which presents additional challenges.

1.4 Thesis structure

This thesis presents the fundamental **background** needed to understand the challenges of adapting a set of hardware cores intended for one architecture over to a new architecture, issues with architectural differences in general, and looks at the main characteristics of RISC-V, SPARC V8, and GRLIB.

We then present our work **methodology**, and the tools used in the process.

Then, we chronologically present our **implementation** of a RISC-V driven platform, by porting the GRLIB IP core library [21] to RISC-V.

In the **results** section, we present our final proof of concept computing platform.

For the **discussion**, we reflect over our implementation process and also consider relevant literature.

Finally, the **conclusion** answers our research questions briefly, and laying out a way forward.

Information and findings that are not strictly relevant for the thesis, but can be nice to be aware of, will be placed in the **appendices** in order to keep the main content as concise as possible. Here you will also find longer pieces of code which is referred to in the text, in order to provide an independent read.

Background

In this chapter we present the extended theoretical background needed to work with the issue of porting IP from one architecture to another. We first discuss immediate issues that arise at developing at the hardware level, before looking at RISC-V. Then we will look at the IP core library we are to utilize - GRLIB. Then we briefly present other open-source ISAs, and notably SPARC V8, which GRLIB is originally developed for.

We assume that the reader has a grasp of the basic concepts within computer architecture. We strongly recommend *Computer Architecture: A Quantitative Approach* [12] as a primer, or as a refresher on specific, base concepts. We will build upon those concepts in the rest of the thesis.

In addition, we assume some familiarity with VHDL and Verilog. We can recommend *Circuit Design and Simulation with VHDL* [22] as an extensive reference on VHDL, and *Digital Design and Verilog HDL Fundamentals* [23] for Verilog.

The authors believe that a grasp of licencing issues (Appendix A) and application software porting issues (Appendix B) is important to keep in mind when constructing a platform, but is not deemed vital for understanding our implementation. We have therefore placed these topics in their own appendicies, and recommend those for the interested reader, for a wider exploration of the porting issue as a whole.

2.1 The hardware development flow

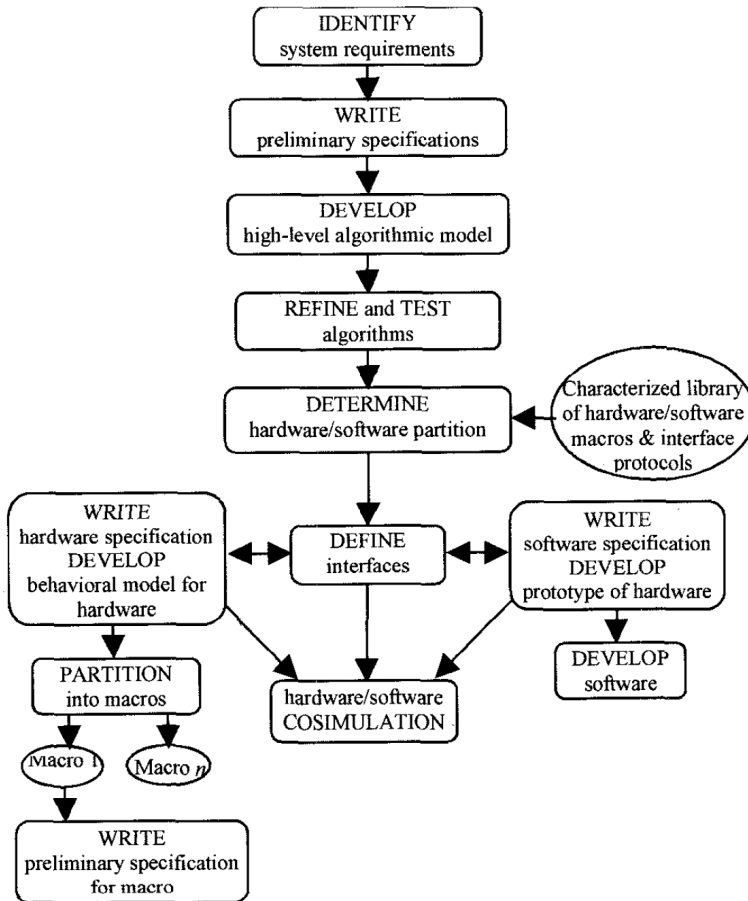


Figure 2.1: “The system-on-a-chip design process”. Like in much of software development, we must initially have a firm idea of what to design before we design it. For the purposes of this port, we barely touch upon the first five steps. We wish to run simple RISC-V compiled software on a SoC with a RISC-V core, deriving from a sample GRLIB design. The last steps are highly relevant. We must identify which cores we wish to use, how we interface them, and then ultimately write/modify/adapt them in order to fit into the design. Figure originally from a paper on IP reuse for SoC design [24].

In broad strokes, the entire process consists of specifying a chip design through hardware design languages (HDLs) such as Verilog or VHDL, running the designs on simulators which can simulate some hundred or thousand cycles of the chip in reasonable time, until one can start prototyping on chips with field-programmable grid arrays (FPGAs) or similar via the use of a synthesis tool to produce gate-level RTL, and an implementation tool for the target FPGA technology. Not before extensive prototyping is done is it

typically feasible to start producing dedicated test chips, using trace-and-route tools to carefully place components on the chip, and wiring (routing) them together and producing a tape-out which is sent to production.

Hardware development resemble software development in many ways. While software programmers utilize encapsulation through usages of objects, functions, and a development cycle (be it waterfall, SCRUM, KANBAN, or any other flavor) for efficient coordination and delegation of work units, hardware has similar concepts. Dividing the design into modules and submodules until you reach the basic blocks of the design is a common form of hardware abstraction, which produces work units that can easily be delegated to dedicated teams.

The hardware development flow for an entire, industry-grade chip, is depicted in figure 2.1, while the typical development cycle for an individual IP core or sub-block of a design, is given by figure 2.2

The formal design process has quite a few elements, and is originally intended for large teams working on a design from scratch. This process can be heavily simplified since we will be working on extending an existing, verified SoC design from GRLIB.

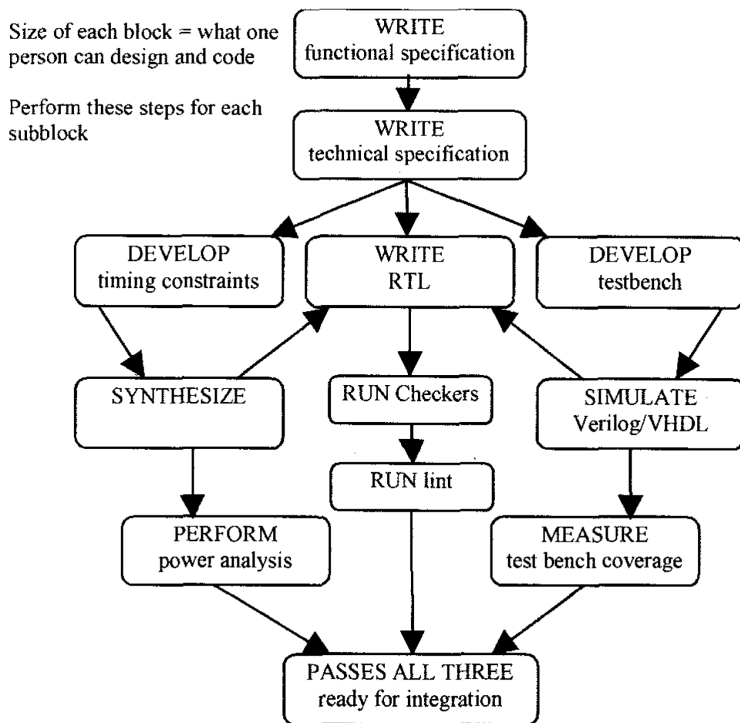


Figure 2.2: “Sub-blocks Design Flow”. This flow is quite formal, but illustrates nicely the inherent complexity of hardware design. Figure originally from a paper on IP reuse for SoC design [24].

2.2 The state of available hardware development tools

Perhaps even more than software development, hardware development strongly requires sophisticated tools in order to support productive development. This includes a good HDL coding environment, IDEs, linters, testbenches, accurate software simulators, possibly development boards for prototyping of implementation, and then finally resulting in an Application-Specific Integrated Circuit (ASIC). The closer we move towards the ASIC, the more time consuming and costly a potential bug becomes, since one often has to walk through all the previous steps again. Trying to test code by going straight to development board or ASICs is at best time consuming, at worst it takes more time and resources than what is available.

Hardware definition languages and tools

On the HDL side, the creators of RISC-V have responded with CHISEL (Constructing Hardware In a Scala Embedded Language) [10] – a language designed with hardware synthesis and high level programming abstractions in mind. It is designed for enabling modular, reusable, parameterizable hardware designs. This is in contrast to the traditional Verilog and VHDL, which are “*hardware simulation languages, (...) later adopted as a basis for hardware synthesis*” [10]. Verilog and VHDL today have many of those features, but the CHISEL authors argue that their language is cleaner since those features were part of the design from the beginning.

The modifiable, open-source Atom editor [25] can be extended with multiple plugins for extended hardware definition support, such as a Verilog linter which utilizes Icarus Verilog for linting.

Simulators

For simulating on the ISA level, the open source RISC-V toolchain [26] does include an ISA simulator called Spike. Some RISC-V projects such as the Rocket Chip generator [2] include a C emulator that can dump VCD files (*wavedumps*) which can be displayed in waveform viewers such as GTKWave [27]. For simulating whole designs, one can utilize the free Icarus Verilog v0.9 [28] simulator if the design entirely consists of Verilog. It can also be used as a Verilog to VHDL generator, and has an experimental VHDL to Verilog preprocessor. On the VHDL side you also have the GNU Compiler Collection (GCC) based GHDL simulator [29], currently on version 0,36.

Implementation tools

When one is ready to move from simulation to physical environments, it is typical to use *Field Programmable Grid Arrays* (FPGAs) for prototyping, before producing dedicated chips. For implementation, a few open-source variants exist, such as QFlow [30].

State of the art: Utilizing vendor design suites

While we have now listed some possible open-source flavors, the authors argue that the industry standard flow is to utilize commercial hardware design suites from the target technology vendor, such as Xilinx Vivado [31] and Intel Quartus Prime [32] to synthesize designs, and then complement them with specialized tools such as ModelSim [33] for extensive simulation, or other tools which provide entirely new functionality. From a short literature review, there is no clear competitor or de-facto open-source alternatives to utilize. You could assemble a fairly complete simulation toolchain, but for implementation, you will be dependent on existing, proprietary vendors.

Tool complexity and cost: a commercialized example

To give an idea of the complexity of such toolsuites, we look at Xilinx's state-of-the-art Vivado design suite. Made for designing and prototyping FPGAs on Xilinx SoCs, it includes the entire tool flow, from top-level system design, to mixed-language simulation, synthesis, implementation (translating, mapping, and place and routing the gate-level RTL from synthesis to a Xilinx target technology), and generating a bitfile which is then used to program the target Xilinx FPGA. This industry standard suite cost a reported "(...) 1000 person-year (\$200million) (...)" [34].

2.3 Hardware porting challenges

2.3.1 Mixed language

Mixed language in terms of hardware development is usually referring to using the two major HDLs, *Verilog* and *VHDL*, together. Support for this is rare in available simulators, and this issue arises in the case of porting GRLIB to RISC-V, since GRLIB is a library written in VHDL, while most RISC-V cores are specified using CHISEL or Verilog. In addition, by aiming for mixing HDLs, you have access to a much larger collection of IP. The two approaches to working with mixed language code bases are to either convert one of the code bases over to the other language, or to use a simulator supporting both languages simultaneously, or to use multiple language-specific simulators (*co-simulation*) which simulate in lockstep and communicate state changes between them [35]. An example is given in figure 2.3.

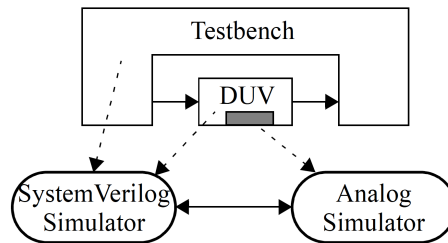


Figure 2.3: An example of a co-simulation environment, in which two simulators meant for two different types of simulation work together in order to provide a complete test environment. Figure from Bergeron 2006 [35].

Translating between HDLs

The approach of converting from one language to another was suggested by the FPGA community [36], by utilizing an openly available translation tool to perform most of the translation work. There are VHDL to Verilog generators and vice-versa, but most require that certain constructs need some manual correction in order for the translation to become complete, such as the commercial SynaptiCAD *Verilog2VHDL* and *VHDL2Verilog* tools [37]. A possible issue here is human error if the amount of manual translation is significant, and becomes even worse when the underlying code base is large. There is also the issue of how to check for correctness when the translation is done. Overall, it requires that the designer is quite familiar with the designs they are trying to translate.

Mixed language supported simulator

Mixed language supported simulators usually provide the designer with the option of writing a *wrapper* that instantiates a VHDL library inside a Verilog library, or the other way around. This feature is usually commercialized, found on only some popular simulator and synthesis tools. Most open source simulators come only in single language variants. The only open solution we found, is Icarus Verilog's VHDL preprocessor, which is in an experimental stage.

2.3.2 Interrupts

Usually, one refers to a *trap* as a software generated exception, while an *interrupt* denotes an asynchronous event happening outside of the processors, but various derivatives of the terms are used in different architectures. A common technique to handle interrupts is to include an *interrupt controller* on the processor, or as a unit on the system bus. Usually, such a controller takes one or several interrupt signals from the platform as input, and then routes them to one or several outputs. How these interrupts are generated, forwarded, forced, and masked, is usually highly architecture specific, and possibly also implementation specific if the architecture allows it. It is therefore yet another concern an architecture porting process must keep in mind.

2.3.3 Atomic instructions

An architecture usually defines several *atomic instructions*, instructions guaranteed to be run in sequence without disruption. Atomics are widely used in order to ensure thread safety for concurrent programs executed by the processor. A potential issue that can arise when changing architecture, is that the atomic instruction on one architecture might not exist on the other. The architect must then take great care in making sure that this atomic instruction is somehow mapped onto the atomic instructions available on the target ISA, and still be atomic (atomic instructions are not *composable* – just running two atomic instructions in sequence does not guarantee that the combination is atomic [38]).

2.3.4 Device drivers

Device driver porting also causes issues when porting devices from one architecture to another. We consider the actual driver porting out of scope for this thesis, but mention the issue regardless. Examples include drivers used for using some of the GRLIB IP cores such as *GRETH_GBIT* (Gigabit Ethernet Media Access Controller) and *GRSPW* (SpaceWire coded with AHB Host Interface and RMAP Target) [21].

Device drivers are quite low level in nature, usually providing an OS with a means to utilize some underlying hardware. This subsequently means that drivers are also very architecture dependent. Although one can argue that device drivers are software, they are so close to the hardware that the software porting approaches we mentioned earlier are not effective, and usually they require manual translation. Oracle [39] provides insight into issues when porting drivers between platforms. Here we list a few:

- **Function call and return syntax might differ.** How arguments, return addresses, and stack space are specified for called functions is highly architecture specific, and is usually defined by a *calling convention* defined in a ABI document for the architecture or similar.
- **Endianness between architectures might differ.** Obviously, normal operation fails if we wrongfully assume instructions or data to have a memory endianness which does not match reality, as the order in which data is read directly impacts its value in a specific endianness.
- **Word size, data types, and signedness for specific datatypes used in the driver might differ between platforms.**
- **Existing drivers might use non-obvious functions or depend on OS-specific behavior.** In other words, there might be behaviors in the legacy operating system which is assumed by the driver, but not explicitly stated. If one only has the driver code available, but no documentation of these assumptions, porting might be more problematic.

2.4 RISC-V basics

The RISC-V architects predicted the following three future trends in computing, as a guideline for how to shape their ISA [19]:

1. Internet of Things (IoT).
2. Personal Mobile Devices (PMDs).
3. Warehouse-Scale Computers (WSCs).

From this, the architects devised four key requirements for RISC-V:

1. **Base-plus-extension ISA:** In order to meet the requirements of the three different areas of computing, the ISA should have a minimal base set of instructions to ensure that all programs for RISC-V can be compiled to run on the base set. In addition, RISC-V provides application-specific extensions to the ISA.
2. **Compact instruction set encoding:** IoT pushes for small code sizes, and strict memory requirements.
3. **Single-Precision (SP), Dual-Precision (DP), and Quadruple-Precision (QP) floating point support:** Scientific applications emphasize accuracy in calculations.
4. **32-bit, 64-bit and 128-bit addressing:** In order to secure a large enough address space for WSCs, which are steadily increasing in size.

From this, they have drafted two architecture volumes of the RISC-V instruction set manual, one for user-mode architecture [40] and one for privileged architecture [16]. We will briefly mention the available RISC-extensions that are fleshed out in the user-mode volume, and then take a closer look at the privileged ISA to take a closer look at the more platform-related concerns. Both volumes are quite extensive, so the reader is referred to the source material for detailed reading.

2.4.1 RISC-V ISA bases and extensions

As part of their design goals, the RISC-V ISA has ended up with numerous variants of its ISA, achieved by defining a set of base ISAs, and a set of ISA extensions which can be added on top of a base RISC-V ISA.

RISC-V Base	Description
RV32I	RISC-V 32-bit addressing with the base integer set.
RV32E	RISC-V 32-bit addressing, a subset of RV32I, aimed at embedded computing.
RV64I	RISC-V 64-bit with the base integer set.
RV128I	RISC-V 128-bit with the base integer set.

Table 2.1: List of base RISC-V instruction sets. Compiled information from [40].

While the base set is designed to be entirely frozen and mandatory in all processor implementations, RISC-V is made very extensible by providing a set of standardized extensions, and also allowing for custom extensions in order to suit all applications.

RISC-V Extension	Description
I	The mandatory base integer ISA.
M	Multiplication and division extension.
A	Atomic instructions extension.
F	Single precision floating point.
D	Double-precision floating point.
G	A combination of I, M, A, F and G.
C	Compressed instructions.

Table 2.2: Select RISC-V extensions. Compiled information from [40].

RISC-V operates with little-endian instead of big-endian, stating simplicity of hardware design as a motivation, combined with it being the standard within dominant ISAs. Non-standard versions can change the endianness.

2.4.2 Platform-Level Interrupt Controller (PLIC)

The interrupt controller and subsequent interrupt handling is strictly defined in the privileged architecture. A fundamental concept in RISC-V is that of a *hart*, synonymous to the term *hardware thread*. There are two types of interrupts, *global interrupts* from external sources (system peripherals and the like), and *local interrupts*, which is the traditional trap (generated by software conditions). However, local interrupts are not handled by the PLIC, but directly by harts using a level-based scheme [16].

For global interrupts, the PLIC comes into play. The interrupt gateways convert IRQs to a standardized IRQ format [16]. From here, it is up to the implementation how interrupts are routed, and how they are prioritized. A specific IRQ can reach a single output hart context, or many. PLIC also supports level-based priorities, edge-triggered priorities, or message-signalled interrupts as mechanisms for forwarding interrupts. The gateway assigns a unique ID to each IRQ, and if two IRQs with the same priority go to the same target at the same time, the IRQ with the lowest unique ID wins. PLIC also supports thresholding of priority levels directed to each target. Meaning that lower-priority interrupts can essentially be filtered out if need be.

While the PLIC is the standardized interrupt controller, RISC-V emphasizes customizability. We have seen implementations in the field such as the PicoRV core [41] and the ORCA core [42], which provide simplified controller schemes by utilizing the custom instructions provided by RISC-V.

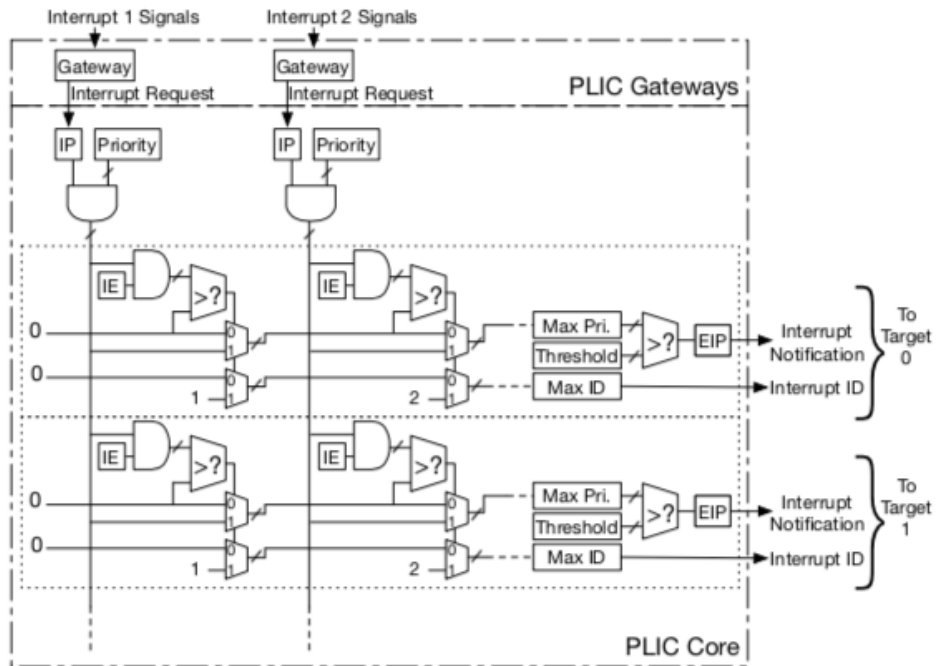


Figure 2.4: The internal structure of the RISC-V Platform-Level Interrupt Controller (PLIC). Each target hart (hardware thread) has an Interrupt Enable (IE) vector, which is used to mask interrupts. The Interrupt Pending (IP) vector indicates whether a specific interrupt with a specific global ID is currently pending to be handled. The pending interrupt with the highest priority, which exceeds the priority threshold of the target, and which is interrupt enabled by the target, gets treated by the PLIC by having the PLIC send the target hart a notification alongside the interrupt ID. The target must notify the interrupt gateway when handling is complete for the corresponding IP vector bit to clear. Figure from [16].

2.4.3 Existing RISC-V cores

We present a small selection of available RISC-V cores and their main characteristics. The RISC-V Foundation maintains an even more updated overview on their website [43].

The Rocket core [2] | *BSD licence* [44] | *RV32G/RV64G* | *CHISEL*

It is highly customizable, and is also usable in the Rocket Chip Generator. It has a MMU with paged virtual memory function. Can support extensions M, A, F and D as well.

The ORCA core [42] | *BSD licence* [44] | *RV32I[M]* | *VHDL*

It targets a proprietary matrix processor. Added support for Xilinx FPGAs. Scraps the standard PLIC in favor of a simpler interrupt controller.

BOOM [1] | *BSD licence* [44] | *RV64G* | *CHISEL*

The Berkeley Out-of-Order machine is a 10-stage OoO-processor. Also usable in the Rocket Chip Generator. A true proof-of-concept that has been tested commercially. Shares many components with the Rocket core.

mriscv [45] | *MIT licence* [46] | *RV32I* | *Verilog*

Targets an in-house microcontroller. Has a APB-bus on-chip, and a SPI-interface. Not updated in over a year, stating new features "coming soon".

SHAKTI series [4] | *MIT licence* [46] | *N/A* | *BlueSpec*
SystemVerilog

The SHAKTI [4] series are processors from India, targetted for governmental applications amongst other things. Implements a variety of RISC-V sets, depending on type. Currently boasts two classes: C-class, a high-performance in-order core, and E-Class for embedded applications.

PicoRV [41] | *ISC licence* [47] | *RV32I[M][C]* | *Verilog*

PicoRV is a very simple, very minimal RISC-V core. It is designed for minimal space usage on FPGAs, at the cost of performance. It also provides a very simple valid-ready memory interface for instruction and data fetches.

2.5 GRLIB overview

Recall that we wish to build a computing platform with a RISC-V core and GRLIB IP cores. The IP cores are well tested and quite diverse, and it is believed that using these will save time on implementing a complete SoC, for the price of solving some architectural challenges. What follows is a summary of the GRLIB IP core library as described in its manuals [48, 21].

2.5.1 GRLIB system

GRLIB [48] is a hardware IP core library, created by *Cobham Gaisler*, a multinational semiconductor company which focuses especially on embedded systems for the aerospace industry. One of the library's more notable IPs is the *LEON3 processor* [49], utilizing the open-source *SPARC V8 ISA* [50]. The LEON3 has 32-bit addressing and a 7-stage pipeline.

Fault tolerance is ensured in their products by using radiation resistant FPGAs from *Actel* and *Xilinx* [51], and by adding fault tolerant features in the processor pipeline. Fault tolerance is a commercialized feature from Gaisler, but nothing prevents the hardware developer from introducing their own fault tolerant features to a design, including using alternate means such as adding redundant cores which execute the same instructions, going for a MISD-architecture per *Flynn's Taxonomy*.

The GRLIB IP core library consists of several vendor libraries. Each vendor's library is organized into a uniquely named VHDL library, which may include one or several IP cores. The goal, according to Gaisler, is to “*hide unnecessary implementation details from the user*”, presenting only the IP interfaces when necessary. GRLIB also supports mapping designs to a variety of FPGAs out of the box.

In a GRLIB configuration, IP cores are connected together via a *system bus*. The creators chose Arm's AMBA AHB bus [52], stating market dominance, documentation, and licence-free usage as the factors speaking in favor of the bus choice. We describe more details in the implementation chapter.

The GRLIB IP library manual [48] provides more technical details on the overall platform.

Minimal LEON/GRLIB system

A minimal LEON system, as given by its documentation [49], includes the following IP cores:

- CLKGEN - A clock generator.
- RSTGEN - A reset signal generator.
- AHBCTRL - Implementing the system bus arbiter and controller.
- APBCTRL - AHB/APB bridge.
- LEON3 - processor core.
- IRQMP - the multiprocessor interrupt controller.
- GPTIMER - General Purpose Timer Unit.
- MEMCTRL - Controller for ROM and/or RAM access.

2.5.2 GRLIB IP cores

GRLIB documents approximately 156 different IP cores in a variety of categories, including:

- Processor and support - this includes their LEON3 and the new LEON4 processor (both SPARC V8 32-bit), and associated modules, like interrupt controllers, clock generators, and debug support units (DSUs).
- Memory control modules.
- AMBA bus modules.
- PCI interface.
- On-chip memory functions.
- Serial communications.
- Encryption.

In short, it should be clear that GRLIB is a rich library of diverse hardware IP cores, which would make the creation of a fully embedded RISC-V system much more feasible.

LEON3

Cobham Gaisler's LEON3 processor is a 7-stage, SPARC V8 based processor, which is highly configurable. Extra features include optional *SPARC Reference Memory Management Unit* (SRMMU), AMBA AHB system bus interface, symmetric multiprocessing (SMP) and fault tolerant features (under a commercial licence).

The processor is released under the *GNU GPL* licence [53] for large portions of the design, with some additional features requiring a commercial licence.

The AMBA AHB interface of the LEON3 is described in [49, p.1194]. It reads to and from the on-chip cache, and has mechanisms in place to handle burst transmission, and for handling transmission faults reported by the slave.

It also has an interrupt interface which follows the SPARC V8 convention (15 asynchronous interrupts), and can both generate and acknowledge interrupts in order to implement the behavior as set forth by the GRLIB Interrupt Controller, IRQMP, described shortly.

IRQMP: The multiprocessor interrupt controller

The IRQMP IP core is the GRLIB system's interrupt controller. All bus units on the AHB bus (including possible APB-bridges) are connected together via interrupt lines, forming an interrupt bus. The bus supports 32 interrupts, while the IRQMP is by default configured to support 15 interrupt levels – which matches the number of supported interrupt levels on a LEON3 processor. It can be extended to cover all the interrupt signals, but the ones not originally included will then have to be mapped onto the existing 15 interrupt levels in order to be compatible with the processor.

By making the processors acknowledge forwarded interrupts, the IRQMP can easily serve multiple processors. The high level overview is given by figure 2.5. The internals are given by figure 2.6. Of note, the interrupt pending register assigns each interrupt a level of 0 or 1, where 1 has higher priority. Within this main priority scheme, the interrupts are further prioritized by interrupt number, where 1 is the lowest, and 15 the highest. When the controller passes a new interrupt to an available processor, it looks for a nonmasked interrupt or a forced interrupt with the highest priority to pass on, and waits on an acknowledgement from the processor before the controller clears the relevant bit in the IRQ pending register.

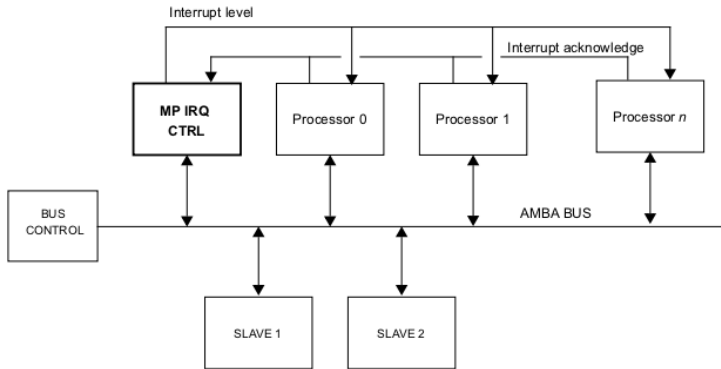


Figure 2.5: Example configuration with the IRQMP. Interrupts generated by the GRLIB based system can be shared with several, individual bus master processors. Figure from GRLIB IP Core User's Manual [21].

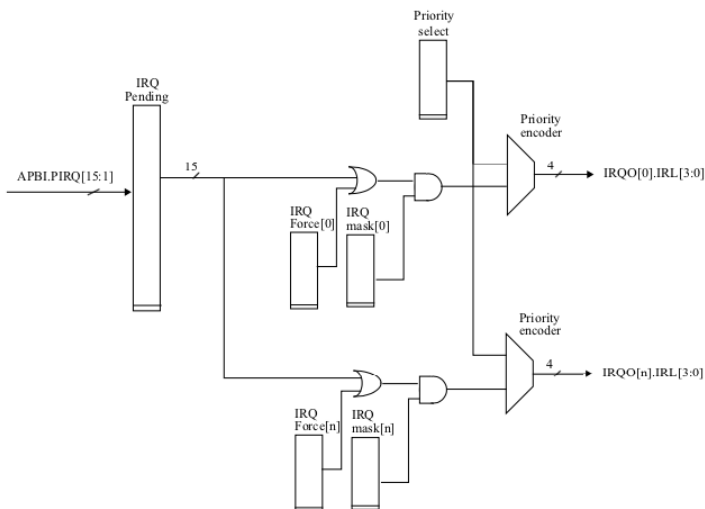


Figure 2.6: The internals of the IRQMP. The way GRLIB is intended to work is to let any bus participant read and set 32 shared interrupt lines. The IRQMP, out-of-the-box, only reads the first 15 lines. In the controller, the IRQ pending register keeps track of which of the 15 interrupts have been raised but not handled. Based on the interrupt's assigned priority, the processor's interrupt mask register (IRQ Mask), and the interrupt force register on the processor (IRQ Force), the controller forwards relevant interrupts to specific processors, who then acknowledge the interrupt in order to clear it in the controller. Figure from GRLIB IP Core User's Manual [21].

2.6 A survey of other major, existing open source ISAs

The authors of RISC-V argues that one of its strengths is the ability to start anew, and learning from the mistakes from its open-source predecessors [19]. Therefore we survey the two major recent predecessors – *OpenRISC* and *SPARC V8*. We will be much more detailed in our survey of *SPARC V8* than *OpenRISC*, since we will have to get familiar with this ISA in order to successfully port to RISC-V.

2.6.1 OpenRISC

OpenRISC is an open ISA project which took flight in year 2000, with a goal of making a free ISA focused on embedded systems. The current and only architecture defined by the project is *OpenRISC 1000*. The architecture is a RISC-architecture with the following main features [54]:

- 32-bit and 64-bit address space.
- Uniform length instructions.
- ORBIS32/64 is the name of the base instruction set.
- Digital Signal Processing (DSP) and floating point (FP) extensions available.

Two processors based on the OpenRISC 1000 specification exist: the *OR1200* [55], and *mor1kx* [56]. OpenRISC 1000 can also be run on the QEMU simulator [57], it has a linux port, and an OpenRISC port for GCC [58], so there is some software available to aid system development for this ISA in mind.

2.6.2 SPARC V8

The SPARC V8 architecture is described in its architecture document [50]. Here we offer a short summary.

Scalable Processor ARChitecture (SPARC) is a RISC-based ISA, originally drafted by *Sun Microsystems* in 1985 in response to the increasing popularity of object oriented languages, leading to modern features at the time such as register windows for fast context switching whenever needed. SPARC version 8 was released in 1992.

SPARC V8 allows designers to use any MMU of choice, including none. It uses *Total Store Ordering* (TSO) as a memory model, which requires that “*stores, FLUSHes, and atomic load-stores of all processors are executed by memory serially in an order that conforms to the order in which the instructions were issued by processors*” [50]. Some notable features include:

- All instructions are 32 bits wide, and align on 32 bit boundaries.
- Three basic instruction formats.
- Register relative addressing and offset relative addressing supported.
- Big-endian memory ordering.
- Harvard architecture for processor cache.

Processor structure

The processor is divided into an integer unit (IU), a floating point unit (FPU), and an optional coprocessor (CP). The integer unit has the conventional general registers, controls execution, and calculates memory addresses. The number of registers available, and register windows, is implementation dependent.

The FPU has 32 32-bit floating point registers, holding either 32 single-precision, 16 double-precision, or 8 quad-precision FP values.

The coprocessor is an implementation-dependent option which enables designers to add a coprocessor of their choosing in order to speed up some computation.

Traps and interrupts

The SPARC V8 architecture defines *precise traps*, *deferred traps*, and *interrupt traps*. Precise traps and deferred traps are generated by software, where the former requires that the trap handler be called before any program state occurs. Interrupt traps are traps generated by hardware, and is of greater interest to our porting means. The architecture defines a standardized default trap model, requiring all traps to be precise, except deferrable FPU and CP exceptions, interrupting trap, and some "machine-check" exceptions. The architecture also defines an enhanced trap model in order to enable user applications to handle certain traps.

For handling interrupts, it uses a priority-level scheme, where a level of 15 indicates an unmaskable interrupt. On an incoming interrupt request, the processor checks the IRQ level compared to the processor's set interrupt level. The IRQ trap is called if the IRQ level is greater than the processor's. Handling behavior is implementation dependent.

SPARC V9

The successor to SPARC V8, SPARC V9, was drafted back in 1993. It introduced, amongst other things, a 64-bit address space, and it went from being entirely open to requiring a licence. Most active development is reportedly seen on SPARC V9, so development on SPARC V8 becomes increasingly more difficult as time passes and more efforts and software toolchain support are focused on SPARC V9.

2.7 State of the art RISC-V platforms

Here, we present some state-of-the-art platforms within the RISC-V realm. Other open-source platforms surely exist, but have not been looked into for this project.

2.7.1 SiFive

SiFive [59] is the company founded by some of the creators behind RISC-V, and is one of the main contributors on the Rocket Chip generator [2].

SiFive offers a wide span of custom-made RISC-V cores, covering the power, performance and area (PPA) spectrum. They also provide reference boards, such as the Freedom Everywhere HiFive1, a RISC-V take on an Arduino board, and the Linux-capable, multi-core HiFive Unleashed board.

Recently, they announced an upcoming chip designer, which will enable designers to easily create an SoC by using template designs, SiFive DesignShare licenced IPs [60], and the designer's own IP. However, not much information was available at the time of writing, but it is still worth mentioning.

2.7.2 SHAKTI

SHAKTI [4] appears to be a serious, state-sponsored actor creating innovative RISC-V cores within various categories. They aim to provide reference SoC designs for various computer types. For now they have demonstrated designs for the E-series (embedded) and C-series (desktop computing). They also state a mission goal of sharing hardware components and tools. Their intent is to make everything 3-clause BSD licenced [44]. One important limiting factor from other approaches is that they utilize the commercial Bluespec SystemVerilog [61] instead of a free and open alternative.

2.7.3 lowRISC

Although it is not yet fully released, lowRISC appears to be closely related to our project, as they aim to create “*a fully open-sourced, Linux-capable, System-on-a-chip*” [6]. They wish to focus on the entire development process from design to manufacture, and on sharing all tools and code created in the process. Early code is already available on GitHub [62] under a permissive licence, but they are yet to release their first platform.

2.7.4 How they differ from our approach

Our proposition is to further democratize computing platform development. SiFive seems to try and generate momentum for the RISC-V ISA by providing configurable reference designs, but not necessarily a free platform in and of itself. Their upcoming chip designer might be interesting once it releases (no date given at the time of writing), but as of now, it seems to focus on gaining momentum for the RISC-V ISA, and little else.

We argue that we must push even further and strive for an entirely open computer architecture which is deemed serious enough for research and commercial purposes. Here we see lowRISC and SHAKTI as promising initiatives. When it comes to lowRISC we are still not able to discern whether or not this foundation will produce tools and designs which will touch upon the same issues as the ones we will be looking at – namely core interfacing, verification, and bare-metal programming. We suspect that this area will be too specific for lowRISC to begin with, and we argue that this project therefore complements their efforts nicely.

Chapter 3

Methodology

In this chapter we briefly present our work structure, goal, development methodology, setup, and the tools and notation that we utilized in this project.

3.1 Work structure

The master's thesis has a time limit of 20 consecutive weeks. It also accounts for 30 course credits, or 100% of allotted study time for the semester. We assume a full workday to be 7.5 hours, and a work week to consist of five workdays. We will therefore be working 37.5 hours every week with the thesis.

To avoid going into the pitfall of forgetting results, we immediately report and document them in the the thesis when ready. This prevents using precious time and resources in remembering what was done, and enables constant review of the thesis contents. This is done as a result of reviewing the efficiency of the process of the master's research project.

Supervisor meetings occurred on a roughly biweekly basis, in order to effectively synchronize often and get valuable feedback.

3.2 Goal

Our goal is to prove that it is possible to construct a RISC-V computing platform by utilizing existing IP and tools. We will use the GRLIB IP core library for most of the components, as it already comes with readily implemented FPGA board designs out of the box, which saves significant amounts of effort in order to bring up a design. On the RISC-V side, we will use an existing processor, and then try to reuse works whenever possible in order to fuse it together with the GRLIB system.

This goal will allow us to answer two research questions: The first is to find out the level of difficulty in constructing a RISC-V computer, the second is to see if there are improvements that can be made in the design process to further empower computer designers in efficiently creating complete, field-tested designs.

3.3 A note on measurements and reporting

In traditional computer architecture papers, one usually performs quantitative benchmarks on some simulator in order to compare a proposed solution to a baseline solution. In our case, this becomes a bit difficult, as porting does not necessarily aim to increase performance or similar. It would be beneficial, of course, to measure execution times on a LEON3 version of a GRLIB system, and then measure execution times of a similar RISC-V GRLIB system to see if we can gain useful insight. Our main initial focus will however be on achieving an "accurate" port. We define accurate as the fact that a RISC-V variant should run similarly to the GRLIB variant (handle interrupts and other runtime behavior in a similar manner), and should produce the same output. Determining this accuracy may not be straightforward, depending on the modules we use in our GRLIB system and what behavior we wish to monitor.

Most importantly, our purpose in our implementation is to gain experience with extending a RISC-V processor with existing IP cores, and derive methods for how to do so on a general basis. While performance should always be taken into consideration, we will prioritize simplicity and reaching a proof-of-concept over performance. When the proof of concepts work, we can then try and work upwards to higher performing solutions if there is remaining time.

3.4 Development methodology

In this project, we will follow a somewhat simplified hardware development structure as compared to what was presented in chapter 2.

Namely, we loosely follow a *design* \rightarrow *validate* \rightarrow *implement* \rightarrow *test* cycle. In reality, we will be jumping back and forth between these constantly. Traditional hardware development projects usually dedicate entire teams to each of these phases, using weeks or months just on planning. We do only have half a man-year available, and a good chunk of that already goes to report work. Therefore, we most likely will have to do simplifications, but try to do so wisely. As echoed in the halls of programming labs: "*Hours of coding can save you minutes of planning*", but at some point we will have to stop planning, and try to reach our goal within the allotted time.

We will try to explain our development decisions, thought processes, and considered alternatives throughout the implementation chapter. Readability will always be prioritized over chronological order. So while the end thought processes might look very clean, there was usually several rounds of trial and error before them, but we omit these to present a more orderly read.

3.5 Development setup

For any type of research, reproducing the results is vital. We ensure this by providing the complete code on GitHub, with complete installation and run instructions located there as well.

3.5.1 GitHub code repository

Here we assume that you are familiar with code versioning, and the related Git tool [63]. GitHub (one of many online Git repository vendors) has extensive documentation if you need a quick primer on forks, branches, commits, submodules, and version trees.

We mention in our approach that we initially want to try fusing together a RISC-V core and GRLIB. This leads to the issue of how we should organize existing code libraries. Also, will updates of one or the other lead to incompatibility issues? What is the most realistic organization? How do we incorporate our code?

We start by creating a GitHub repository for our project, `riscvy-business` [64]. The intent is that one who wishes to reproduce all our results here can do so, by only consulting this repository. We strive to make it very clear which code is not our own, and to respect the original licences.



```
.
├── README.md
├── riscvy-grlib
│   ├── Makefile
│   ├── bin
│   ├── boards
│   ├── designs
│   ├── doc
│   ├── lib
│   └── software
├── uartMonitor
│   ├── Makefile
│   └── UartMonitor.cpp
```

Figure 3.1: We have created a public Github code repository, `riscvy-business` [64]. It contains two top directories, `uartMonitor` and `riscvy-grlib`. `uartMonitor` contains an application for communicating with the GRLIB system’s APBUART via the JTAG port. `riscvy-grlib` is a modified version of the 2018.3 GPL release of GRLIB. Of note, you will find most of the IPs, including our riscv addition, in `riscvy-grlib/lib`, and various default, synthesizable board designs in `riscvy-grlib/designs`.

GRLIB was, at least at the start of development, only available through the Cobham Gaisler website. They do periodic, official releases which are made readily available, but without the commit history. We have worked with `"grrlib-gpl-2018.3-b4226"`. 2018.3 is the year and quarter. b4226 refers to the build number. The `"gpl"` part indicates that this is the GPL-licenced version of GRLIB, in contrast to their commercial licence releases with fault-tolerant features and others. We have added this codebase as `riscvy-grlib/`

in our `riscvy-business/` repository to indicate that this is a modified version of GRLIB. Finally, we create a `riscv` library in `riscvy-grlib/lib/` for most of our contributions.

The other codebases we have used were available on GitHub at the time of writing. This includes FreeAHB [65], PicoRV [41], and the RISC-V variant of the GNU toolchain [66]. We have chosen to fork most of these to the same user as where the `riscvy-business` repository resides to clearly indicate the origins of the source code we have used, safekeep a copy of the code, and clearly indicate which specific commit of the code we are working with.

With the forks in place, we reference them inside our newly created `riscvy-grlib/lib/riscv/` library in the `riscvy-business` repository through the use of git submodules. See figure 3.1 for further details on the repository structure. The structure of the `riscv` library added to our copy of GRLIB is described in figure 3.2.

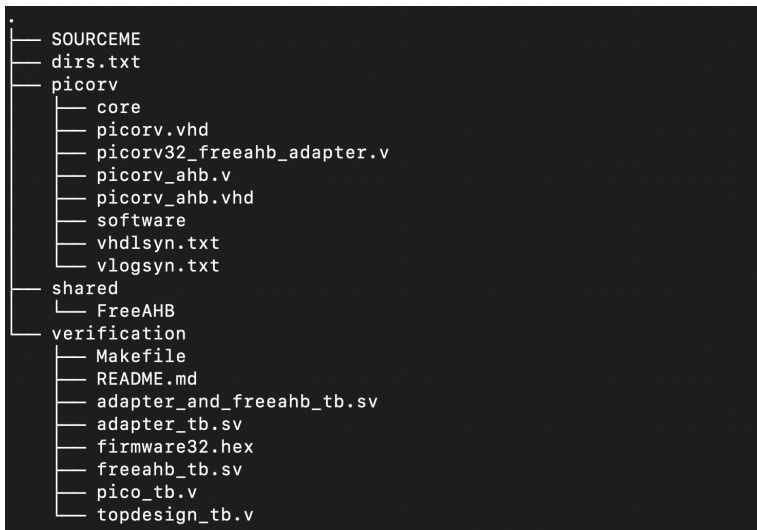


Figure 3.2: The `riscv` library file structure in `riscvy-grlib`, our modification of GRLIB, using the `tree` utility. The text (.txt) files are a part of the build process in GRLIB, which we will describe later. `picorv/core` is a git submodule (reference to another repository) of a fork (copy) of the official `picoRV32` (created by Clifford Wolf), for safekeeping. The various Verilog (.v) and VHDL (.vhd) is part of our porting to GRLIB, which we again will discuss later. In the `shared` folder, one will find common elements that are shared between various RISC-V cores. For now, it only contains FreeAHB (another git submodule), a AHB master originally written by Revanth Kamaraj, and with minor modifications on our part so that it follows the AMBA AHB spec. More on that later.

3.5.2 Choice of development infrastructure

Initially, we had a macOS laptop running Ubuntu on a virtual machine using Oracle VirtualBox 6, but this added significant overhead which made interaction with Vivado very sluggish, even when tuning the VM to optimal settings. Running Vivado on a remote server, while programming the bitfiles locally helped somewhat, but then we started having issues with USB passthrough and USB-drivers for the FPGA connection, which led to more time wasted on debugging caused by virtualization.

Having a dedicated x86 workstation with Ubuntu 18.04LTS ensures that the results are more reproducible, drivers work without any issues, and everything runs faster. GRLIB recommends Ubuntu over Windows with Cygwin.

3.6 Custom notation

3.6.1 Commands

We will assume that you use a terminal with the Bourne Again Shell (bash) in order to run system commands. Other shells may be possible, but might require some modification. To denote a command, we will start the line with a dollar sign:

```
$ echo "Hello world!"
```

3.6.2 Environment variables

Many of the involved systems rely on the use of environment variables, that is, variables that are defined in the current environment. The environment is typically contained within a shell process (terminal instance).

To set a variable, we will usually write it in all caps, like this:

```
WHAT_IS_YOUR_QUEST="To seek the holy grail"
```

To use the value of a variable, we write:

```
$ echo $WHAT_IS_YOUR_QUEST
```

3.6.3 Folders, files and elements of code

Folders, files, and variables from pieces of code, will be marked like `this`. The context should make it clear what it refers to.

3.7 Utilized software

Here we list all the major relevant pieces of software which we use in our project. For detailed installation and running of the final project, refer to the thesis' code repository [64].

3.7.1 Active-HDL: Student Edition

ACTEL ActiveHDL [67] is a mixed-language HDL simulator, which has a free student edition. It is used to simulate the entire final implementation.

3.7.2 BCC

The Bare-C compiler is a software toolchain made by Cobham Gaisler for compiling applications for their LEON3 and LEON4 processors on a GRLIB system. It includes firmware for interacting with most GRLIB peripherals. Available from the Cobham Gaisler website [68].

3.7.3 Cygwin

Cygwin [69] is a collection of open-source tools which are usually found on Linux, made available on Windows. It is used to provide tools required for running the Actel Active-HDL simulator with testbenches provided by GRLIB.

3.7.4 Digilent Adept

Digilent Adept is a software suite for Digilent Adept compatible devices. In our project, we will use a Digilent HS1 platform connection, which gives us access to a on-board JTAG scan chain. The Digilent Adept runtime provides us with the required cable drivers. The Adept utilities provide us with some useful tools, and the extensive Adept SDK provides us with libraries which can be used to develop our own applications for Digilent Adept targets. The SDK also includes all the programmer's manuals required. All downloads are available from the Digilent Adept webpage [70].

3.7.5 FreeAHB

FreeAHB is an openly available AHB master written in Verilog with SPLIT and burst capabilities, created by Revanth Kamaraj, available on GitHub [65].

3.7.6 Git

Git is, per its manual pages, the "stupid content tracker" [63]. It enables version control of code bases, and to distribute them in any manner to a variety of locations such as GitHub.

3.7.7 GRLIB

GRLIB is an open-source IP core library by Cobham Gaisler [48], aimed at embedded aerospace applications. They provide most of the IPs as open-source with a GPL licence, while some features come at a cost (error-correction, fault tolerancy). We describe GRLIB in greater detail in section 2.5.

3.7.8 GRMON

GRMON is the GRLIB monitor, a piece of software which can connect to a GRLIB system on an FPGA or a finished ASIC by using one of multiple available debug links such as UART and JTAG. It is intended for on-chip debugging of LEON3 and LEON4 software. Comes in a commercial and an evaluation version. Available from the Cobham Gaisler website [68].

3.7.9 Icarus Verilog

Icarus Verilog is an open Verilog simulator, which can compile and simulate designs written in Verilog, and produce wavedumps which can then be inspected in a waveform viewer. Available from the Icarus Verilog webpage [28], or via Ubuntu's package manager.

3.7.10 Newlib

Newlib is a library for organizing common syscalls used by C and C++ programs in such a fashion that it can easily be adapted for embedded applications. Available on their SourceForge page [71].

3.7.11 PicoRV

PicoRV is a minimal RISC-V core created by Clifford Wolf, available on GitHub [41].

3.7.12 RISC-V build tools

The RISC-V tools are readily available on GitHub [26]. They include the GNU toolchain for compiling applications for the RISC-V, the ISA-simulator Spike for running the built applications on the ISA, and various tests.

3.7.13 Ubuntu 18.04 LTS

We will utilize the latest Long-Term Support (LTS) version of Ubuntu for this project - Ubuntu 18.04.01 Bionic Beaver [72]. Ubuntu is a variant of the popular Debian distribution, based on Linux. We could choose a different Linux-flavored OS, but keeping it as similar as possible to what the original developers use can leave out the possibility of mismatching toolchain versions and similar, as different Linux-flavors have different policies of how updated, or "bleeding edge", their package repositories are. We also chose this OS due to its large following and great documentation.

3.7.14 Windows 10

We will need the Windows 10 operating system for running licenced software which are not built for Ubuntu. In our case, it is the ACTEL Active-HDL simulator we need to get running. Windows 10 requires a licence, but most universities usually provide academic licences.

3.7.15 Xilinx Vivado

Xilinx Vivado [31] is a very popular HDL design suite which can perform HDL synthesis and analysis on both Verilog, VHDL, or a mixture of the two. An industry-standard tool, which reportedly took 1000 man years to create. It is a commercial tool, but academic versions are available.

Chapter 4

Implementation

We wish to create a proof-of-concept to show that it is indeed feasible to construct a free RISC-V computing platform using the GRLIB IP Library. We discussed our intended approach in chapter 3. In this chapter, we will present the process. We will try to keep it an independent read, mostly high-level, with some code examples. Leaving the detailed code documentation and specifics to the code repository and the appendices.

4.1 Choosing a RISC-V core and attaching an AMBA AHB master to it

Before anything else, we need to start off with a RISC-V core to work with. We will also need to figure out how to interface a candidate core with the GRLIB's system bus (AMBA AHB) so that it can be served memory and control GRLIB peripherals in a complete design. During our survey of RISC-V cores, we discovered the Rocket Chip Generator (RCG) [2], a generator which can create a full RISC-V chip with a BOOM or a Rocket core as its CPU. We consider this to be rather state of the art, so we look into possibly using this first.

4.1.1 Locating a point of entry in the Rocket Chip Generator

The main characteristics of the *Rocket Chip Generator (RCG)* is described in detail in a paper by its authors [2]. A sample generated system is given in figure 4.1.

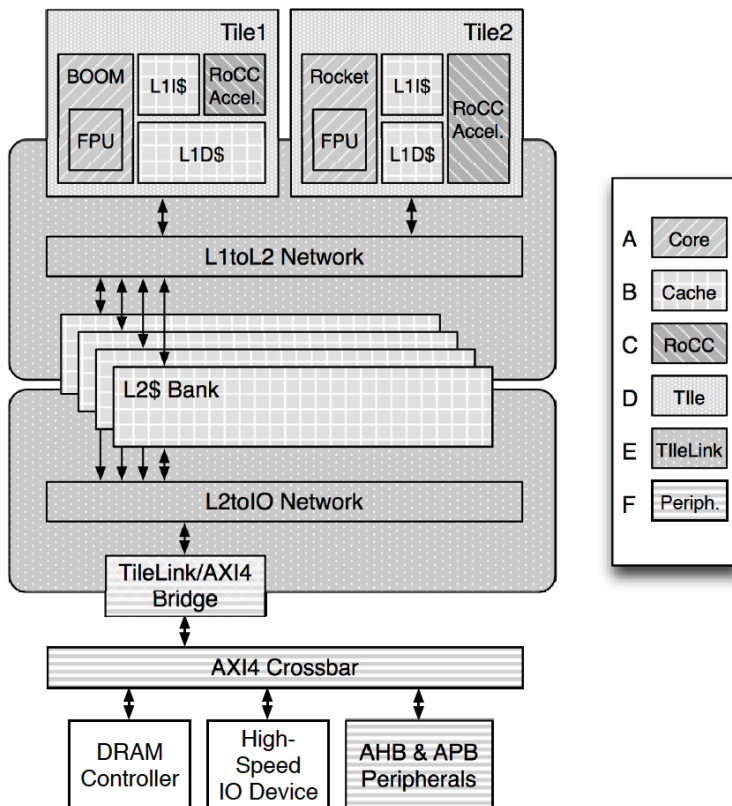


Figure 4.1: A sample design that is achievable with the RCG. The list on the right indicates the various generators which the RCG is composed of. Figure from the original Rocket Chip paper [2].

One of the on-chip interconnects (OCIs) that are available on the RCG is *TileLink*, an *AMBA AXI4* (current AMBA bus generation) alternative from SiFive, one of the main contributors to the RCG. The generator comes with TileLink adapters to some AMBA bus masters, namely *AXI4*, *AHB-Lite*, and *APB*. Unfortunately, there is no full AHB master available.

The question therefore becomes where and how to couple the RCG with the FreeAHB core. Initially, based on our limited previous experience, we proposed a simple memory-mapped I/O approach, where the signal lines of our full AHB master would be mapped to a part of the system address space, then a driver routine would constantly poll it and implement our proposed FSM based on those values. This would mean that parts of the system address space would have to be mapped onto some form of scratch pad or similar on the Rocket Chip, for startup and similar, while most of the functionality and main programs be accessible via AHB on the GRLIB system.

However, this approach makes the memory system awkward as it adds overhead to even simple memory transfers, and ideally one wishes to utilize AHB as a true system bus. Therefore, we were advised to look at a solution that was more tightly coupled with the processor. This is depicted in figure 4.2.

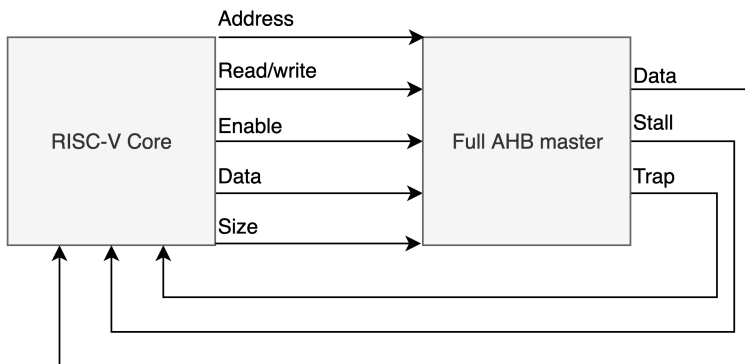


Figure 4.2: Suggested, conceptual interfacing between a RISC-V core and the AHB Master we are to create. The core should be able to tell the AHB master which address to start reading or writing to, what data to write for writes, how much data to read or write, and so on. Not all possible inputs to the master is given, such as protection bits and burst specifications. The AHB master should then in turn tell the processor to stall until the read or write has occurred. When we finish stalling the processor, it can read the data lines if this was a read operation. Finally, we may wish to raise a trap for certain conditions, such as an AHB slave responding with ERROR. Figure created in draw.io.

With this in mind, we dove further into the RCG documentation in the repository, which we found to be outdated and does not describe many of its components in much detail – seemingly aiming for a “*learning by doing*”-approach. The closest thing we found to how to add a new memory interface were three suggestions for how to couple accelerators with a RISC-V core. While an AHB master is not necessarily an accelerator and will most likely not be integrated in the same way, it could point us in the right direction. The suggestions given in the RCG documentation [2] are, in order from most tightly coupled to not-so-tightly coupled:

1. Extend the RISC-V pipeline.
2. Use the coprocessor interface, RoCC.
3. Latch onto TileLink with a custom tile.

We will now go over each approach in detail.

Extending the RISC-V pipeline

Consider the textbook CPU pipeline [12, Appendix C] containing *Instruction Fetch*, *Instruction Decode*, *Execute*, *Memory access* and *Writeback* stages. The proposition is to have a special RISC-V instruction that tells the processor that the instruction is to be run on a dedicated functional unit: the accelerator. For a memory interface, such as the AHB master, the more natural placement would be as part of the Instruction Fetch and Memory access stages of the pipeline, when the pipeline fetches instructions and data respectively.

This solution would be highly core-specific, but also very tightly coupled to the processor. For this approach, it would mean leaving out the RCG caches, which then slashes away the performance benefits gained from the memory hierarchy shaped by L1 and L2 caches.

The coprocessor approach

The coprocessor approach is perhaps the least suitable approach for a memory system (but very suitable for an actual accelerator), as it makes the coupling between the processor and the memory system unnecessarily complex, as every instruction fetch and data fetch has to go through the coprocessor to feed the main processor. We entertain the thought a bit here as a digression, but feel free to move on to the TileLink approach, which we consider to be the second realistic point of entry for a memory system.

The RoCC coprocessor mechanism makes use of the customizability of the RISC-V ISA to add some special RoCC-coprocessor instructions, which are used by the main processor to tell the coprocessor what to do. The Rocket Core pipeline will pass such a RoCC instruction over the RoCC processor once it has passed the commit stage of the pipeline. So instead of being part of the pipeline, it now resides right outside of it.

The RoCC interface allows a connected coprocessor to access the rest of the memory system, and to raise an interrupt directly to the processor, which will certainly be useful in our case.

One of our immediate concerns regarding the RoCC coprocessor approach is the problem of how to issue instructions to the coprocessor. If we consider figure 4.2, assume that

we keep the address space to 32-bits to match the GRLIB target, and perform unlocked transfers with 32-bit reads and writes only, we are looking at the following information we will need to feed the AHB master in order for it to initialize a transfer:

- **Transfer size:** 3 bits
- **Burst type:** 3 bits
- **Protection bits:** 4 bits
- **Read/write:** 1 bit
- **Address:** 32-bits (assuming 32-bit addressing)

The RoCC interface has the following instruction format:

CustomX	rd	rs1	rs2	funct
2-bit opcode, accelerator number	destination register	source register 1	source register 2	7-bit integer instruction identifier

To fit our AHB master in this scheme, a rather natural approach would be to utilize the registers as pointers to data in the address space (source and target), and then use the `rd` register to either hold the result, or point to an address containing the result. `rd`, `rs1` and `rs2` will be 32-bits on a 32-bit RISC-V core. However, the `funct` bit of the RoCC instruction remains a constant, and is a bit of a problem. We still have up to 11 bits (size, burst, prot, r/w) that we need in order to fully inform the AHB master about what transfer we want. So a single RoCC instruction will not do.

One possible solution is to make the processor issue several instructions to the RoCC connected AHB master to setup and run a transfer. It is for example possible to split the 7 `funct` bits into two parts: a 3-bit opcode and a 4-bit value. We envision something like the following:

Opcode (3 bits)	Value (4 bits)	Action
000	0000 or 0001	Specify transfer direction, HWRITE.
001	0xxx	Specify burst parameters, HBURST.
010	xxxx	Specify protection bits, HPROT.
011	Any value	Fetch GRLIB configuration table.
100	Any value	Start the specified transfer.
101	Any value	Reset the AHB master

On each instruction issued for a specific transfer, the `customX` bits, `rd`, `rs1`, and `rs2` bits should be kept constant. Since single `customX` instructions are being committed in the pipeline, the AHB master should then be able to verify that all the `customX` instructions refer to the same transfer by checking the `rd`, `rs1` and `rs2` against previously recorded values of the registers.

This solution has some obvious flaws. For one, extra logic is required in the AHB master to check that all commands refer to the same transfer. Secondly, a minimum of 4-5 instructions are required in order to initiate a transfer. Third, out of 24 bits in a RoCC instruction on a 32-bit system, 17 bits are repeated across all instructions, which isn't very efficient.

Create a separate TileLink tile for the AHB master

This approach would be to have a TileLink/AHB bridge, similar in placement as the TileLink/AXI4 bridge in figure 4.1. The significant advantage over the first likely alternative - extending the CPU pipeline - is that it includes the RCGs L1 and L2 caches. The drawback is that we will have to add TileLink and Diplomacy [73] to the list of technologies and techniques we need to get familiar with, before moving on to implementation.

Furthermore, our advisor suggested to look into modifying the existing TileLink to AHB-lite bridge into a Full AHB solution. The difference between AHB-lite and full AHB on the master's side is that the full version has extra signals and logic to support a multiple master setup and arbitration. Therefore, in theory, it should not be too much of a reach to try several approaches, such as:

1. Modify the TileLink AHB-lite source code, such as the AHB-lite state machine and outgoing ports so that it becomes a full AHB-master.
2. Synthesize the RCG with the existing AHB-lite module. Connect the AHB-lite ports going to the AHB bus to an AHB-lite to full AHB adapter written in Verilog, which then acts as the actual AHB-master on the bus.

Since the FreeAHB master is to be the sole external memory interface for our design, such a TileLink-AHB bridge would be the most natural approach. This would also allow us to use the extensive functionality of the AHB protocol with bursts, wraps, and protection bits, as a potential MMU serving the I- and D-caches of a RCG could specify the relevant control signals for a transfer.

4.1.2 Conclusions on the Rocket Chip Generator

We realize that the Rocket Chip Generator is starting to become too complex for our initial prototype. It is a very large, modular system with a language that abstracts away the Verilog. But when most other Verilog-side things are written in Verilog and not CHISEL, the fusion between the two becomes an issue. In addition, the highly abstracted, parameterized, and largely undocumented codebase makes it very difficult to work with for an unexperienced designer.

Notably, the Rocket Chip Generator is very centered around many new technologies, such as CHISEL3, TileLink, and Diplomacy [73], which adds on top of the rest of the tools, languages and techniques we already have to get familiar with. It adds complexity, when there already is plenty.

A major concern is that following this path can lead to extra potential issues, when what we initially want to test is simple (interface a RISC-V core with GRLIB via a AHB master, fix the architectural issues, then the performance issues later).

We do not have much experience with hardware design in general, and therefore we fear from experience in software design that having a huge codebase with many unknowns can cause many headaches early on. In addition, we observe that no matter the choice of core, there are still other independent problems that need to be solved, such as:

- Running software on a bare-metal RISC-V system.
- Verifying the RISC-V side of the entire design with simulator testbenches.
- Interfacing our verilog-based RISC-V design with the GRLIB system VHDL.
- Converting a regular AHB master to a GRLIB AHB master.
- Implementing a design on FPGA.
- Running software on the FPGA.
- On-chip debugging.

So no matter the choice of core, we will have to solve several other problems. In short, cores in a final solution should not be difficult to interchange, especially when the goal is a proof-of-concept, and not a design that has to be tailored for the most performance. Once one has constructed a RISC-V computing platform with GRLIB for one RISC-V core, it should not be far-fetched to apply the same methods, techniques and testbenches for another RISC-V core.

All in all, the biggest problem is that the Rocket Chip Generator is designed to be highly flexible in terms of generating designs from it, not necessarily in terms of modifying the Rocket Chip Generator. We believe that since we are after a very specific setup as a proof of concept, time is better spent on a specific, simple core, instead of making a highly general AMBA-AHB capable Rocket-Chip and working from there.

4.1.3 The PicoRV RISC-V core

After deciding that the barrier of entry (as a first-time VLSI project) was too high with the RCG, we searched for simpler cores. We found PicoRV32 by Clifford Wolf [41], a simple RISC-V 32I[M][C] written in Verilog, with simple, well-defined memory, interrupt, and trace interfaces.

It is minimal in the sense that it has been designed for minimal size on FPGAs, not performance [41]. This means, amongst other things, that there are no caches or scratchpads as on the LEON3 or on a Rocket Chip design. It exposes a very simple memory interface with a valid-ready cycle, depicted in figure 4.3.

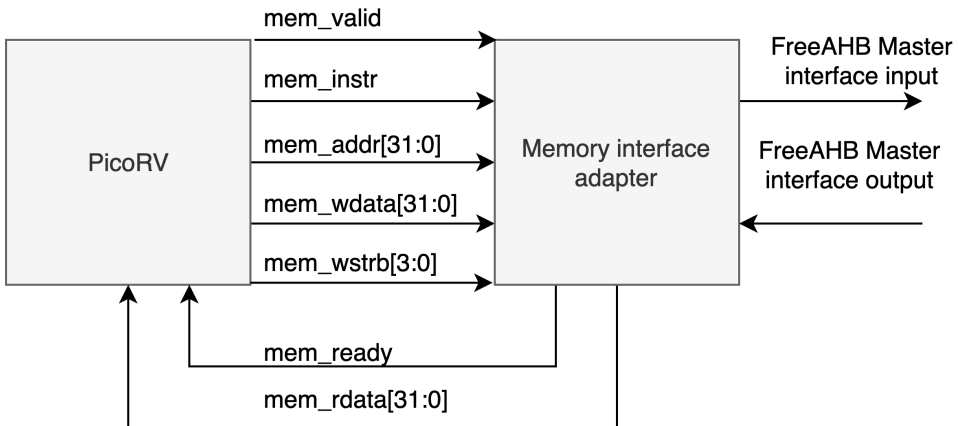


Figure 4.3: The memory interface exposed by PicoRV for connecting it to a memory system (this omits the interrupt, trace, and other signal ports from PicoRV). Out-of-the-box it comes with a sample AXI-interface adapter. By following a similar style, we can connect this valid-ready style interface with an AHB master interface as well, which we depict here as well. The adapter simply translates the PicoRV input and output signals into signals that conform to the FreeAHB interface. Figure produced with draw.io on the basis of the PicoRV documentation [41].

When PicoRV raises `MEM_VALID`, it will hold all the input signals to the adapter constant until the adapter raises `MEM_READY`. `MEM_INSTR` is asserted when fetching instructions, `MEM_ADDR` specifies the target address, `MEM_WDATA` is the write bus, and `MEM_WSTRB` are write strobes (enables) to specify which of the bytes to write in an addressed word. A `MEM_WSTRB` of 0 indicates a read. This is commonly used in the AXI4 protocol. [41]

It also has a memory lookahead interface so one can see the next requested transfer a cycle earlier, but we do not currently intend to utilize this feature.

4.2 Studying the AMBA AHB bus protocol and deriving a Finite State Machine

We have decided to use PicoRV as our core of choice, which presents a simple memory interface. We wish to connect PicoRV to a GRLIB system. GRLIB uses AMBA AHB as its system bus, so in order to successfully interface the two, we must get familiar with AMBA AHB first.

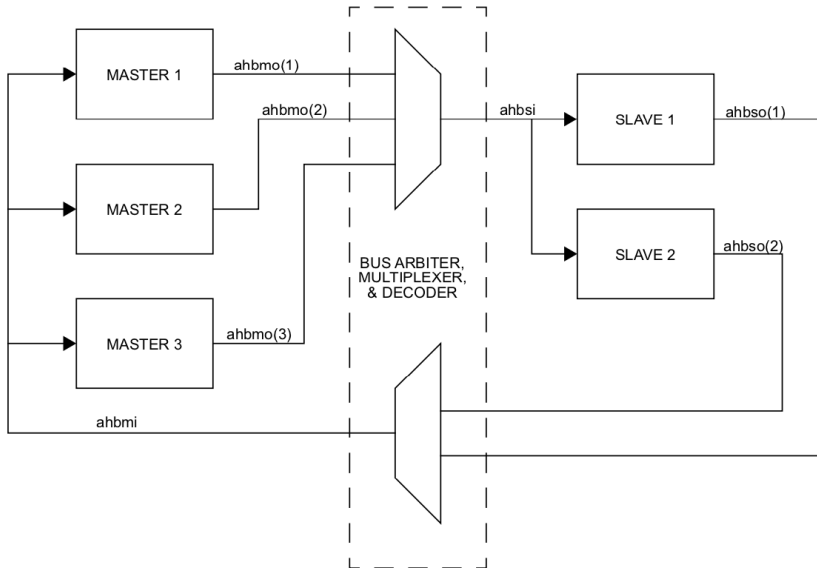


Figure 4.4: "AHB inter-connection view", from the GRLIB documentation, which is faithful to the original interconnection in most ways. The custom `ahbmi` VHDL primitive is the collection of all signals going to the AHB masters, while the `ahbmo` VHDL primitive is the collection of all signals going to the bus arbiter and bus slaves. Figure from the GRLIB user manual [48].

4.2.1 The AMBA AHB specification

The full *AMBA AHB* bus is defined in the *AMBA specification revision 2.0* document [52]. It is a multiple master, multiple slave, system bus protocol, which uses a centralized bus controller for address decoding and arbiting bus control. It must not be confused with *AHB Lite*, which is a simplified, single master multiple slave variant of the bus, without an intermediate arbiter.

A typical transfer consists of a bus request phase, an addressing/control phase which lasts for exactly one clock cycle, and one or several data transfer phases – depending on the characteristics of the transfer (burst size, transfer type, data bus width). Its operation is pipelined for increased performance. The address (and control information, if it is an entirely new burst) for the next data transfer *T* is given while data transfer *T-1* is taking place on the bus. This pipelining is illustrated in figure 4.5.

There are three main components in an AHB bus:

1. **Masters:** The ones who initiate transfers, by requesting the bus, and specifying address range and various control signals when granted, which sets the specifics of a transfer. Provides write data via the `HWDATA` bus, or receives data from the `HRDATA` bus.
2. **Slaves:** Units on the bus who send data to, or receive data from a master, depending on the transfer. In addition to obeying the master when needed, it also communicates back to the master the state of an individual transfer.
3. **Bus controller:** Sits conceptually between the masters and the slaves. It usually consists of an *address decoder* and an *arbiter*. The arbiter decides which master gets to utilize the bus, and does so by a simple arbitration algorithm such as round-robin, highest priority, or similar. The most suitable algorithm will depend on the nature of the target system, and the prevention of deadlock or starvation conditions must be taken into consideration. The address decoder takes the memory address given by the master, and determines which slave is to receive the data, and enables that slave for the duration of the transfer.

Each type of participant drives a subset of the bus signals, in order to fulfill their function. Changes occur on the rising edge of the bus clock, `HCLK`. Which signals are driven when is dictated by the protocol, while the precise timings (when in a clock cycle to consider a signal valid) is left up to implementation, as this may vary depending on the target technology [52]. We are going to refer to these signals time and again, and we therefore find it useful to present these in table 4.3.

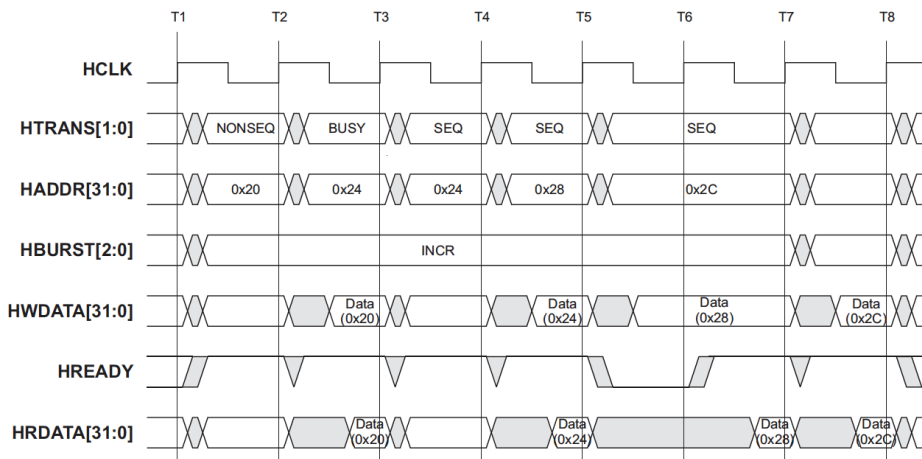


Figure 4.5: The AHB pipeline. We see that the address information for the next data transfer is provided at the same time as the current data transfer takes place. A "BUSY" HTRANS state indicates that the AHB master currently does not have the data to write available, or that it cannot receive more read data immediately. Source: AHB specification document [52].

AMBA AHB Signals		
AHB Master	Bus controller	Slave
<p>HADDR 32-bit system address bus, indicates the location of the next transfer.</p> <p>HTRANS Two bits Indicating the type of transfer. Is either <i>IDLE</i>, <i>BUSY</i>, <i>NONSEQUENTIAL</i> or <i>SEQUENTIAL</i>.</p> <p>HWRITE Indicates whether the transfer is a read or a write.</p> <p>HSIZE Two bits indicating the total amount of bits to transfer.</p> <p>HBURST Indicates single transfer, incrementing burst, or wrapping burst.</p> <p>HPROT Indicates whether the transfer is cacheable, bufferable, privileged, and/or a fetch for opcodes or data.</p> <p>HWDATA Default 32-bit bus used for writing from master to slave.</p> <p>HBUSREQx Raised by bus master nr x when it requires control of the bus.</p> <p>HLOCKx Signals that this transfer should not be interrupted.</p>	<p>HSELx Indicates the currently selected slave.</p> <p>HRESETn Active low. Indicates full bus reset.</p> <p>HGRANTx Indicates that bus master x has currently been granted the bus.</p> <p>HMASTER 4-bit signal indicating which master is currently conducting a transfer.</p> <p>HMASTLOCK Set when the current transfer is locked.</p> <p>HCLK The bus clock. Transitions occur on the rising edge.</p>	<p>HRDATA Default 32-bit bus used when master performs a read transfer with slave.</p> <p>HREADY Used to tell the master whether the current transfer is completed, or if more time is needed.</p> <p>HRESP Used to tell the master the status of the current transfer. Is either <i>OKAY</i>, <i>RETRY</i>, <i>ERROR</i> or <i>SPLIT</i>.</p> <p>HSPLIT 16-bit signal telling the arbiter which master should be allowed to continue a non-locked, split transfer.</p>

Table 4.3: AMBA AHB signals, categorized by the driving entity. Compiled information from the AHB specification document [52].

The AMBA AHB Master

It is natural for our chosen core to be an AHB master, as it will need to request memory from the bus, and it needs to control system peripherals and perform writes, which can only be done by bus masters.

The master is always the one who initiates a new transfer. It must know the address ranges of each slave in order to properly address them. It must also decide whether to perform a read or write, to lock the bus, whether to place protection on the transfer, what transfer size to use, and insert BUSY cycles if it currently can't transmit the next write data, or receive the newest read.

The specification [52] gives an interface-level overview of a theoretical master, which is depicted in figure 4.6.

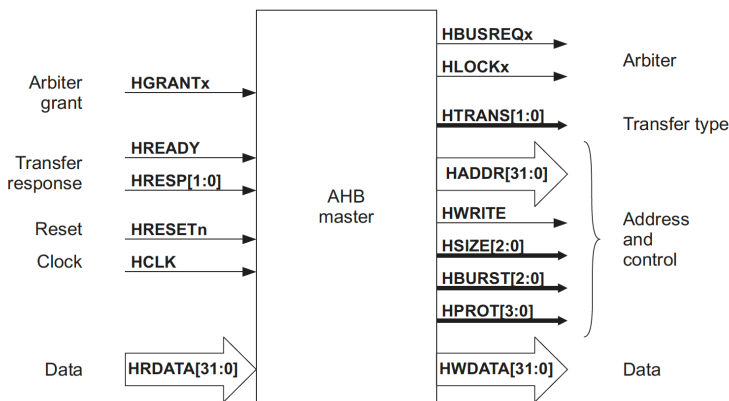


Figure 4.6: AHB Master interfacing. Shows ingoing and outgoing signals for a sample master. The control information in a burst (HSIZE, HWRITE, HBURST, HPROT) is held constant throughout the transfer (except on pipeline rollbacks, when HSIZE and HBURST needs to be recomputed), while the address and data change between each individual transfer that constitutes the burst. Source: AHB specification [52].

Given that the participating slaves support SPLIT transactions, master X can, if the nature of the transfer allows it, specify the transfer to not be locked by driving HLOCKx to LOW. This possibly leads to a higher-performing bus, since if a SPLIT-capable slave detects that producing data for transfer will take time, it can tell the arbiter to SPLIT the transfer, so that in the meantime another master can be granted the bus to perform their transfer.

However, since we in our prototype will only be utilizing one AHB-master (a single RISC-V core), we plan to only allow locked transfers. This is to simplify our FSM and to make the overall setup simpler. We depict an FSM with this behavior in figure 4.7.

Transfers can come in three forms: a *single* transfer, an *incrementing* burst, or a *wrapping* burst. The single transfer is either defined as a SINGLE transfer in the HBURST signal, or uses INCR for incremented burst of unspecified length, with only one transfer. The incremental burst accesses sequential portions of memory with either 4, 8 or 16 beats

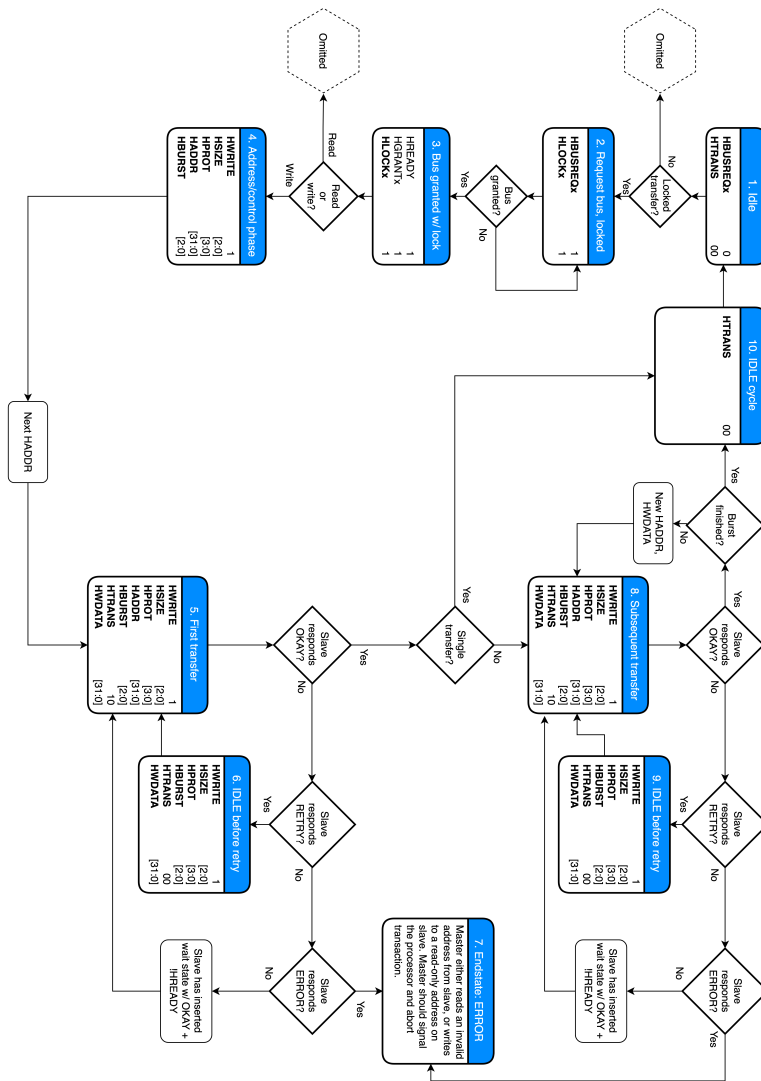


Figure 4.7: A proposed finite state machine for our AHB master. Signals driven by the AHB master is typed in boldface. Unlocked, interruptable transfers (via SPLIT or RETRY), is considered out of scope for our first version. We also omit read transactions from the FSM, as they are for the master pretty similar to a write, except for utilizing the read bus HRDATA. We also omit BUSY states, they are similar to a SEQ state, but its HWDATA is not valid, and it then transitions to a SEQ state when the write data is ready. Every visit to every state shown takes a clock cycle, unless repeated cycles are indicated. Note that at the end of a locked cycle, a IDLE cycle is required in order to allow for arbitration to other masters after a transfer is done. Figure created with information from the AHB specification [52].

(transfers), and each beat has a size determined by the HSIZE signal, which can be 8 bits, 16 bits, 32 bits, 64 bits, 128 bits, 256 bits, 512 bits, and 1024 bits, depending on the transfer and the data width of the implemented bus (the recommended default is 32 bits). An incremental transfer can, per the specification, not cross a 1kB boundary. A wrapping burst wraps accesses on an alignment boundary, set by the size of the entire series of burst, which is HSIZE times the number of beats.

Not depicted in the figure above is how the AHB master interfaces with what it serves on behalf of. For example, our PicoRV will need to connect with an AHB master somehow. Typically, one exposes a set of ports which serves as a user interface, which allows said PicoRV to tell the AHB master what transfers to perform, and where to read from or write to. Such a user interface is highly implementation specific, and is therefore not depicted in the AMBA AHB specification.

4.2.2 GRLIB-specific modifications of AMBA AHB

GRLIB has a *plug-and-play* mechanism added on top of the standard AMBA AHB specification in order to easily swap out IP cores in a chip configuration. It achieves this with three features: *identification* of units, *address mapping* for slaves, and *interrupt routing*. Each AHB participant (master or slave) must have a GRLIB-defined, VHDL primitive configuration - HCONFIG - which consist of eight 32-bit words (see figure 4.8). The first word, the identification register, is used for identification and interrupt routing. The following three words are user defined, and the last three are the so-called bank address registers, used for address mapping to the slaves.

During synthesis of the design, the HCONFIG primitives are assembled and placed in the AHBROM GRLIB IP, which is a simple ROM accessible to all on the AHB bus. By utilizing software routines for LEON3, or by utilizing GRMON – a system monitor geared towards GRLIB systems (or more specifically LEON3 on-chip application debugging), one can read the contents of the table and swiftly identify system components.

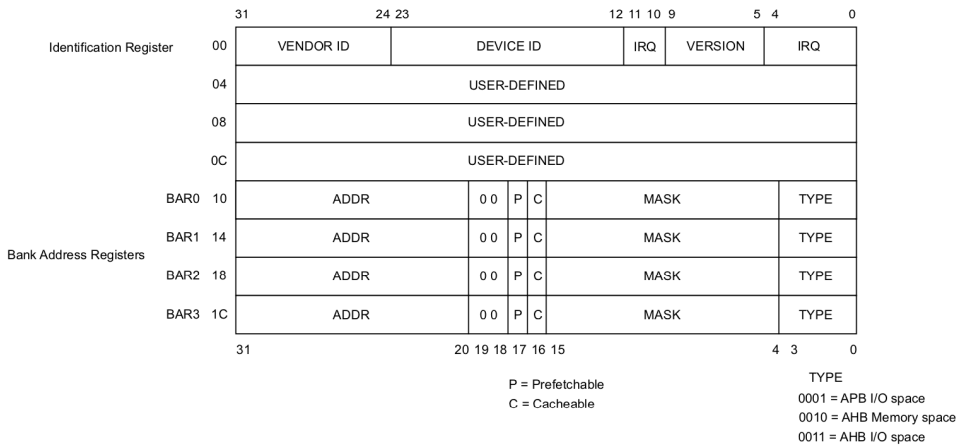


Figure 4.8: "AHB unit configuration", specific to GRLIB's plug and play protocol. The identification register holds some standard information, the following three words are user defined, and the last four words are Bank Address Registers (BAR) used for configuring AHB slaves. Figure from the GRLIB user manual [48].

4.2.3 Mixing endianness on a fixed-endian bus

From the AHB specification [52, p. 3-27]: “*Dynamic endianness is not supported, because in the majority of embedded systems, this would lead to a significant silicon overhead that is redundant*”. Only in a “*wide variety of applications*” do they recommend bi-endian modules, and GRLIB has gone for big-endian, complicating matters slightly.

Namely, a little-endian RISC-V system that is getting instructions and/or data from a big-endian AHB bus needs to be careful of two key aspects:

1. **Which byte lanes on the data busses are driven** for reads, and which to drive for writes. See figure 4.9.
2. **The endianness of received instructions and data**, be it from data registers in IP cores, or from sections of main memory.

Transfer size	Address offset	DATA [31:24]	DATA [23:16]	DATA [15:8]	DATA [7:0]
Word	0	✓	✓	✓	✓
Halfword	0	✓	✓	-	-
Halfword	2	-	-	✓	✓
Byte	0	✓	-	-	-
Byte	1	-	✓	-	-
Byte	2	-	-	✓	-
Byte	3	-	-	-	✓

Figure 4.9: “Active byte lanes for a 32-bit big-endian data bus”. On a little-endian bus, the order is reversed (where applicable). Accesses must be aligned on 32-bit (word) boundaries. Figure source: AMBA AHB specification document [52, p. 3-26].

The latter issue applies to both reads and writes. When reading little-endian data via a big-endian bus, received `HRDATA` will be presented in big-endian order, and must be rearranged to little-endian on the byte-order. Bit-order is assumed to be big-endian by convention. Likewise, when writing, the write must either be converted to a little-endian write over a big-endian bus write before transmitting over `HWDATA`, or be passed straight to the bus, in the case of writing big-endian data on a big-endian based bus.

It is vital that the entire chip orchestrates the storage and retrieval of data with differing endianness in an orderly fashion. One could for example use the vendor and device IDs in the GRLIB plug-and-play mechanism in order to look up its endianness (given it can only be configured with one endianness), or extend one of the user-defined registers in

`HCONFIG` for all IPs being used. As for main-memory, one could partition it into areas of given endiannesses – simply not mix data of differing endianness.

For our project, we will mostly deal with some big-endian data when performing writes to some bus slaves such as the UART registers, and deal with the problem of byte-lanes when reading little-endian instructions and data from main memory via the big-endian AHB bus. The RISC-V system will likewise perform most of the computing on little-endian instructions and data, avoiding endian-translation issues such as strings being presented in reverse, as demonstrated by Tanenbaum & Austin [74] in figure 4.10.

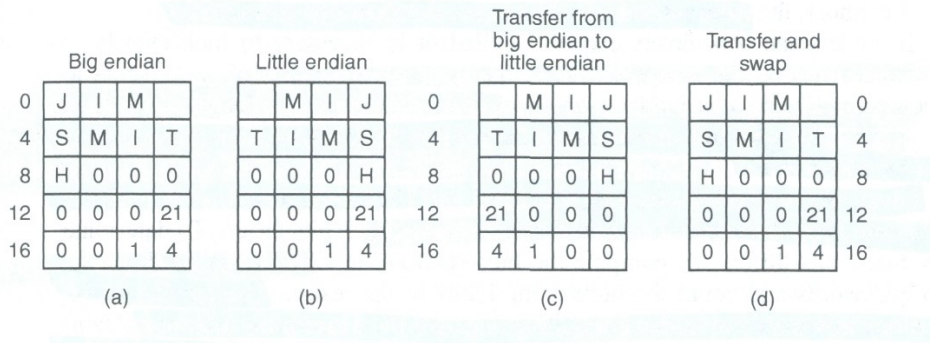


Figure 4.10: A simple example with a sample string and two integers, illustrating the issue of translating data between endiannesses. Original source: Structured Computer Organization [74, p. 77].

4.3 Finding an AMBA AHB master

With our AHB FSM in place, we felt prepared for constructing an AMBA AHB master module in Verilog. The reasoning behind using it over VHDL is that most RISC-V core designs we have encountered are written in Verilog. We wish to have a single, simple Verilog-to-VHDL wrapper between our RISC-V system and a sample GRLIB system design.

Before venturing out on writing an AHB master from scratch, we did a quick search for existing, open variants. We figured that if we do so when writing application software (using existing libraries and the like), there should be no reason to do differently in hardware. This resulted in finding *FreeAHB* [65], a capable AMBA AHB master, written by Revanth Kamaraj, and published on GitHub under the MIT licence.

On first inspection of the source code, three things became clear:

1. **FreeAHB supports unlocked and SPLIT transfers**, two simplifications we did in our AHB study. It makes the FSM more complex, since an unlocked transfer can be interrupted mid-transfer, and the AHB master must then perform a pipeline rollback, recompute the burst for the remaining data, and request (and be granted) the bus before continuing the transfer. It is possible to disable the SPLIT behavior of all AHB units in GRLIB, if this proves to be a problem.
2. **The attached testbench only tests some simple behavior**. The testbench simulates an AHB master, but instead of an arbiter, it immediately grants the AHB master the bus. Then, it writes random data to a simple slave 100 times, before exiting. We can therefore not know whether or not the SPLIT, RETRY and RETRY behavior works, or the master's reaction to the arbitration works correctly, given the wavedumps we can retrieve from the given testbench.
3. **The design is marked as experimental**. We therefore treat it as such.

From this first inspection, we decide that this is a nice starting point for getting familiar with writing FSMs in Verilog, and decide to extend/improve it rather than writing something new from scratch.

4.3.1 FreeAHB implementation specifics

AHB implementation

Before proposing changes, we believe it is informative to present the initial version of the code. You will find a complete original version in appendix C.

The code defines a single AMBA AHB module. The module has a set of I/O ports for connecting to a target AMBA AHB bus (“*AHB signals*”), and a set of I/O ports for communicating with the entity that submits memory requests to the AMBA AHB master (“*FreeAHB UI*”).

FreeAHB implements the AHB pipeline by utilizing two word memories for most of the protocol signals, in order to facilitate pipeline rollback, in case of a slave RESET or SPLIT response, as it must then revert to the previous transfer and recompute the burst – per the specification. The control signals (HWRITE, HSIZE, HPROT,) of the AHB protocol are constant throughout a burst, and thus only needs that state remembered.

The code then selects the appropriate memory words in the pipeline flipflops to drive the outgoing AHB bus signals. Most of the rest of the code follow the AHB specification, except for handling ERROR responses from the slave, and wrapping bursts as mentioned in the repository’s readme.

FreeAHB User Interface

The FreeAHB UI is the most implementation specific part of FreeAHB, which we have to concern ourselves with.

It has a `i_min_len` signal for setting the minimum guaranteed length of a burst. Ideally, we would like to replace this signal with more UI signals, so that the design that is connected to the master can specify all the control signals (HADDR, HSIZE, HBURST) by itself, and therefore always be certain how many beats there will be in a potential burst.

It also has a signal `o_addr` for indicating which address the read was done from. While this might be nice for debugging, we believe that the unit connected to the master should be able to remember what it previously has requested when we have verified the design in a simulator.

It also has a `i_dav` and a `o_dav` signal as a valid-ready style interface. The input data valid signal indicates to FreeAHB that the provided write data is valid, and can therefore continue with the transfer. The output valid indicates that the read data is valid, and that the current read beat is successful.

One important signal is `o_next`. This signal is driven whenever the bus has been granted, the HREADY signal is asserted, and when there are no pending splits in the pipeline. It indicates that the pipeline is ready to move on to the next pipeline step. A UI input of `!i_write` and `!i_read` indicate IDLE conditions, and `i_write & !i_valid` during a write burst indicates a BUSY condition to the FreeAHB master.

In normal operation, you start the FreeAHB master by performing the initial control/address phase. Asserting `i_read` if it is to be a read transfer, and `i_write` for a write transfer. The FreeAHB master will then request the bus with the control information. `o_next` is raised when it is granted the bus and observes HREADY on the AHB bus. The unit attached to the FreeAHB UI is expected to provide the next pipeline step (address T, and data T-1) on the next raising edge after it observes `o_next`.

4.3.2 Code review

We quickly discover that some signals are missing from the spec, namely HPROT, the protection signals, and HLOCK, the control signal from the master specifying whether or not the transfer should be locked. Therefore, we extend FreeAHB by adding `i_prot` and `i_lock` as input ports. We add corresponding pipeline flip-flops `hprot` and `hlock` for these two signals, which only needs one memory word since they are control phase signals which do not change during a burst.

While we would want to feature-complete the AMBA AHB master with proper wrapping burst and reactions to an ERROR response, we delay this until later, as we will assume for the time being that we will only be using SINGLE memory transfers which never cause ERROR conditions, and we will configure GRLIB slaves to not utilize the SPLIT capability.

4.4 Fusing together PicoRV and FreeAHB

Interfacing PicoRV with the FreeAHB consists of three components and related wiring: the PicoRV core, an adapter to translate signals from the PicoRV memory interface to the FreeAHB UI, and the FreeAHB master itself. Building on figure 4.3, we get figure 4.11.

The verilog code for the adapter can be found in appendix D.

`MEM_INSTR` is used to set the relevant `i_prot` bit, which again maps to the `HPROT` signal of the AMBA protocol. When `MEM_WSTRB` is 0000, it indicates a read, and thus this makes the adapter raise `i_read`. Else it raises `i_write`, and indicates which of the four bytes to write in an addressed word.

Translating the write strobes of the memory interface adds some complexity to our adapter. AHB does not have any write enable signals in the same style as AXI4. We must therefore construct a mapping between all 15 potential combinations of `MEM_WSTRB` and AHB compatible transfers.

Our initial proposal is using the AHB's `SINGLE` transfer mechanism. This might be the lowest-performing solution due to poor data bus utilization, but should be simpler to implement than other approaches. In short, we make the adapter do a single transfer of 8 bits for every write enabled byte, making sure to calculate the correct address offset for the byte. Performance hits will be taken whenever there are two or more sequential bytes that are write enabled, as performing a burst or a transfer with a larger beat size would be more efficient utilization of the AHB bus.

A potential improvement would be to handle sequences of write enabled bytes with the largest transfer possible. A problem is the case of three sequential bytes, as AHB does not support a `H_SIZE` of 24 bits. Here, one would have to do a 3-beat incremental burst with a `H_SIZE` of 8 bits or similar.

In addition, we add a module parameter for informing the adapter of which endianness the AHB has, and drives the correct byte lanes for reads and writes accordingly. This choice is finalized in synthesis, as unreachable code (such as big-endian AHB behaviors if synthing with `.BIG_ENDIAN_AHB(0)`) will be pruned away by the synthesizer.

Finally, one problem with our initial `MEM_WSTRB` handling is that in some applications, it might be crucial that a master gets to finish writing all the write enabled bytes in the addressed word, before another master is handed the bus. In our GRLIB prototype, it might not be all that relevant since we will most likely only have one master, but in a multi-master setup, this might be a potential memory coherence issue for the memory on the slave. Therefore, it would be advisable to assert `HLOCK` for the transfer in such environments to ensure that the master gets to finish writing its entire word.

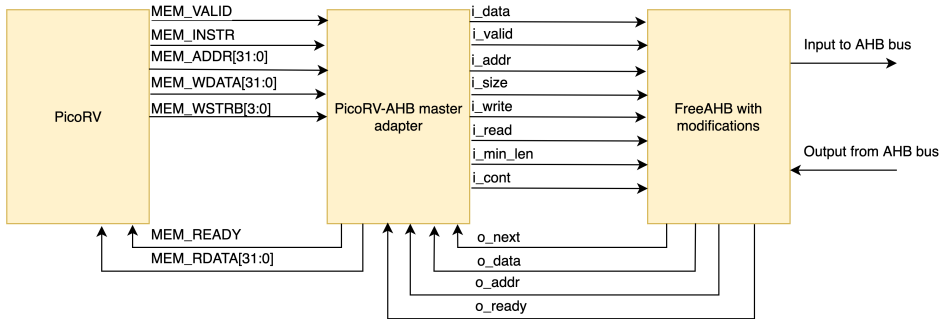


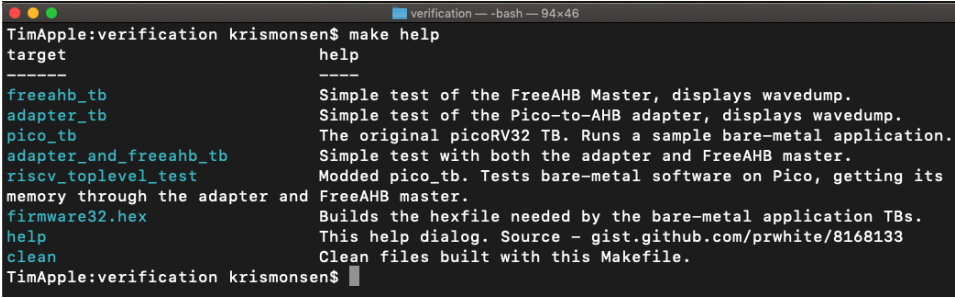
Figure 4.11: The schematic for connecting PicoRV to the FreeAHB master for memory transfers.

4.5 Verifying the design so far

With the RISC-V system in place, it is now suitable to run some simple tests in *Icarus Verilog*, an open-source Verilog simulator, in order to verify that the design works as intended. Referring to the thesis code repository structure in chapter 3, in `riscvy-gllib/lib/riscv`, we now create a new folder, `verification`. It consists of a set of simple testbenches of the various components.

We also provide a simple Makefile in `verification`, which compiles and runs specified testbenches on Icarus Verilog, and then presents the resulting wavedump using GTK-Wave.

In this section we describe the debug methods, software, and the behavior of our testbenches.



```
TimApple:verification krismonsens$ make help
target                                help
-----                                -----
freeahb_tb                            Simple test of the FreeAHB Master, displays wavedump.
adapter_tb                             Simple test of the Pico-to-AHB adapter, displays wavedump.
pico_tb                                The original picoRV32 TB. Runs a sample bare-metal application.
adapter_and_freeahb_tb                 Simple test with both the adapter and FreeAHB master.
riscv_toplevel_test                    Modded pico_tb. Tests bare-metal software on Pico, getting its
memory through the adapter and FreeAHB master.
firmware32.hex                          Builds the hexfile needed by the bare-metal application TBs.
help                                    This help dialog. Source - gist.github.com/prwhite/8168133
clean                                    Clean files built with this Makefile.
TimApple:verification krismonsens$
```

Figure 4.12: Running `$make help` in `RISCVY_GRLIB/lib/riscv/verification` presents the available testbench targets.

4.5.1 Simple test firmware

When testing early phases of design, we found it useful to have a test firmware available which the PicoRV can run. The PicoRV has a `scripts/cxxdemo` folder, which contained a full working example for running a simple C++ application, which prints out hello world, creates some objects, sorts some numbers, and prints them to the screen. It was simple enough to get into and adapt for our system so that we could start testing early. In a later phase of the implementation, we will dive a bit further into the firmware and extend it.

For now, we will only mention that this "hello world" firmware has been relocated to `riscvy-business/riscvy-glib/lib/riscv/picorv/software/sim`. It has been only slightly modified. Namely, we change the stack pointer so that it matches our target GRLIB system. We also change memory locations in the linker script in order to match our memory system. Finally, prints to standard output are forwarded to GRLIB AP-BUART on address `0x8000_0100` instead. Because of this, we also had to do some minor modifications to the original PicoRV testbench and how the simulated memory file (firmware32.hex) was generated. We defer from a discussion of this, as we consider it trivial.

4.5.2 Debugging methods

During this initial debugging, four debugging techniques were used:

Checking PicoRV assembly printouts

If one defines the `DEBUGASM` macro, PicoRV will print out the corresponding assembly that it is currently using. Defining `DEBUG` will also give you the value of memory read into and out of the memory interface. This, in combination with the `ram.srec` (a more readable version of the test firmware which can be made with `make ram.srec`), makes it very easy to check if memory accesses are performed correctly, at least for the startup code, which are almost only sequential reads from `0x4000_0000` to `0x4000_0090`.

A SREC file can consist of various types of S-records [75]. The firmware SREC mostly consists of S3-type lines, which are data meant for 32-bit address systems, and can look like the following:

```
S3 15 40000000 0B600006F1AE13000000130000000100 73
```

Spaces have been added for readability. An S-record always starts with an S, then the type of record. Then, the next hex-pair displays the amount of bytes that follow. In this example, it is $16 * 1 + 1 * 5 = 21$ bytes. Four bytes naturally go to the 32-bit address (third grouping), and one byte goes to the checksum (last group), leaving us with 16 bytes of data (16 hex pairs, the longest group). Having the SREC file at hand, one can easily cross-reference what the PicoRV states it got, and see if it matches that in the SREC file, or if it reads something too early, too late, or just returns garbage (in case of a bug in the memory system).

Utilize debug prints in FreeAHB and the adapter

FreeAHB comes with some debug prints for its beat and burst calculating functions. This is useful for checking to see if FreeAHB constructs the bursts you expect it to. Early in testing, we misunderstood the `i_min_len` FreeAHB UI port, which caused FreeAHB to calculate a different burst than what we expected. In the adapter, we would also place some debug prints, but we quickly found them to be redundant as most can be seen in wavedumps.

Inspecting wavedumps

Wavedumps are a complete recording of a simulator session. It allows you to see the state of internal registers in simulated modules, and the values of internal and outgoing signals as well, giving you a complete view of the state of the simulated state at any point in time. This is especially useful for checking, for example, the AHB protocol in FreeAHB, as there are quite a lot of signals involved. By looking at the wavedump, we can quickly compare the signals values we observe there with the examples we see in the AHB specification. If a design halts indefinitely, which happens on a faulty FSM such as in early versions of the adapter, we can see the module state which caused it to hang, and then derive why.

Performing code review

There were instances in which we became completely stuck, and could not immediately assess the error from wavedumps or debug prints. We then found it effective to walk through our code again, step by step, drawing out the execution on paper for each timestep. If required, we could also revisit specification documents or READMEs and clear up misunderstandings from first readings.

4.5.3 Testbenches

The FreeAHB testbench

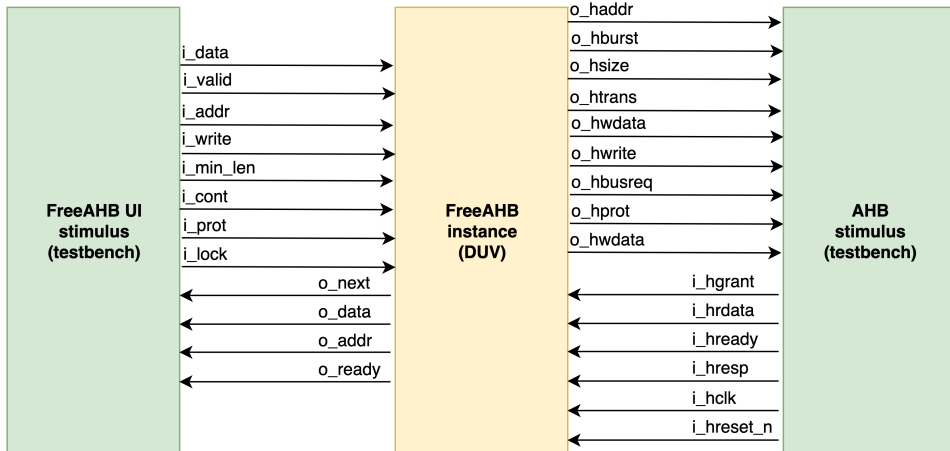


Figure 4.13: Schematic illustrating the DUV and how stimulus is fed into the design, and what signals go from the DUV to the testbench. Diagram constructed in *draw.io*.

The FreeAHB testbench is a SystemVerilog (a superset of Verilog) testbench, slightly modified from the original FreeAHB testbench found in its source repository [65]. The testbench instantiates the FreeAHB master, and provides stimuli for the FreeAHB UI, and ingoing AHB signals. We have pruned away a dummy AHB slave from the original testbench, seeing as it only performed functions a testbench could do (generating simple test stimulus).

The stimulus execute a single test. It immediately grants the FreeAHB master the bus, and the slave always responds with `OKAY`. Then, the FreeAHB master UI is driven to indicate that this is a transfer with at least 42 beats (composed by FreeAHB with a combination of `INCR16`, `INCR8`, and `INCR` transfers).

When FreeAHB asserts `o_next` to indicate that it is ready for the next AHB pipeline step, the testbench randomly sets `o_dav` to HIGH or LOW, and the data written is a count of how many writes is done so far.

It only writes when the write data, `i_data`, is valid (`i_dav`). On invalid data it uses a technique called *x-injection* to cause a high-impedance state on the write bus in simulation, which is done to distinguish between the data write value `0x0000_0000` and not driving a value on the bus, `0XXXXXXXX`.

X-injection is only applicable in simulation. It makes debugging via wavedump-inspection easier. Wavedumps result from running the testbench, and by using a waveform viewer such as *GTKWave*, one can view the state of all signals in the design at any point from the testbench run.

The testbench illustrates nicely how the FreeAHB functions in an ideal environment where it is always granted the bus, and never receives a RETRY, SPLIT, or ERROR response from a slave. However, it does not cover these responses, assert if the writes are correct, and if HLOCK and HPROT are handled correctly, to name a few. Coverage is therefore not great, but we will have to defer more work on this until later, as time is of the essence.

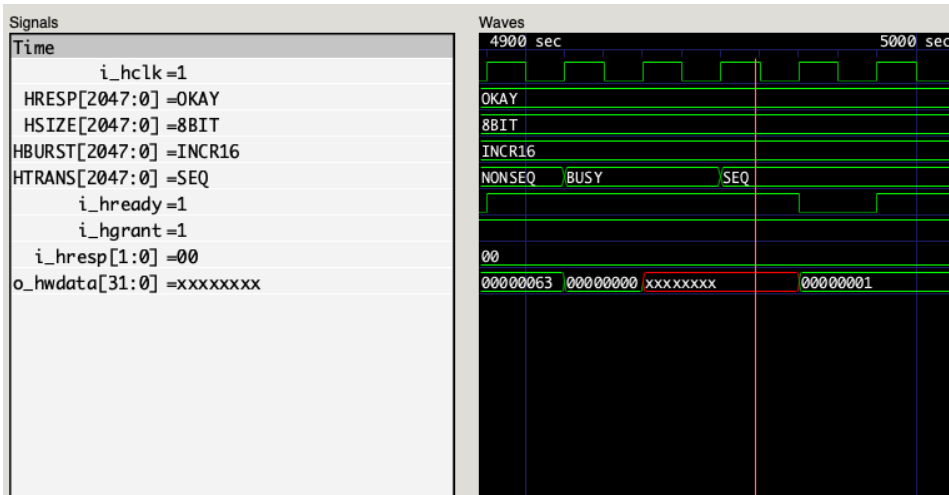


Figure 4.14: A screenshot from analysing wavedumps from the FreeAHB testbench, by utilizing GTKWave. HRESP, HSIZE, HBURST, and HTRANS are special debug registers which display the status of their corresponding signals in ASCII.

The PicoRV-to-FreeAHB adapter testbench

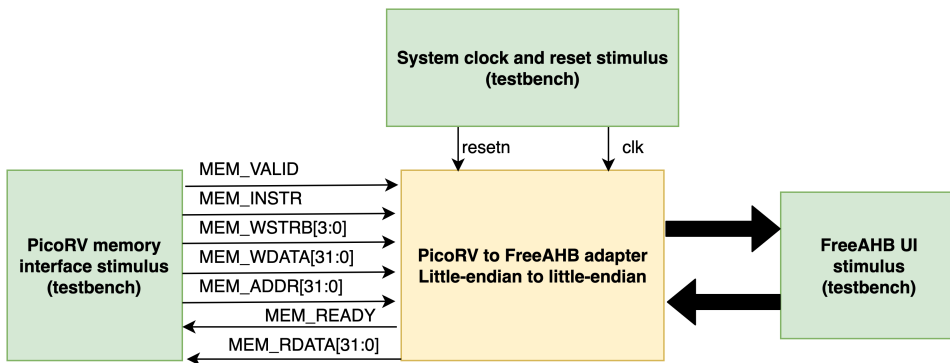


Figure 4.15: The adapter testbench. The adapter is configured for little-endian PicoRV to little-endian AHB. The FreeAHB UI signalling is the same as in figure 4.15, but the DUV now drives the signals going into the mocked FreeAHB, while the testbench provides stimuli for the outgoing. The PicoRV is now also mocked by the testbench, by having it provide valid memory interface signals. Produced in *draw.io*.

This testbench instantiates our custom adapter in little-endian PicoRV to little-endian AHB mode, which converts PicoRV memory interface signals into UI signals for the FreeAHB master. PicoRV-connections and FreeAHB UI connections are stimulated by the testbench.

The FreeAHB master is mocked, by having the testbench wait for the adapter to drive the UI signals from an IDLE to a running state. Then the testbench stimulates signals on the FreeAHB UI back to the adapter which gives idealistic conditions – the next signal is always raised, and writes and reads are always done on the same clock.

There are two tests. One read request from the mocked PicoRV (which is always a 32-bit read), and one write request with two of the byte lanes being written (the two highest-order bytes). The validity of the design is confirmed via the outgoing wavedump. The written and read data is not checked.

The PicoRV testbench

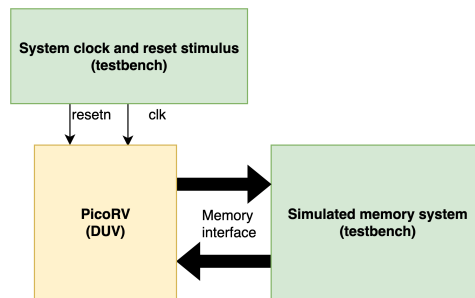


Figure 4.16: Block diagram of the PicoRV testbench. The memory interface signals are the same as before, but the memory system is entirely simulated by the testbench. Figure produced in *draw.io*.

The PicoRV testbench we utilize is a slightly modified version of the one from the original PicoRV source repository [41], and requires some scaffolding before initiating a testbench run (most of which also was already provided in the original repository, published under a permissive licence). Via a Makefile, the RISC-V toolchain alongside some software, provide a test routine which demonstrates printing hello world in C (`printf`), C++ (`std::out`), and pushing some integers on the stack, sorting them, and then printing out the sorted numbers.

We will dive into the software process later when describing bare-metal software development in section 4.10. For now, we focus on the testbench. The testbench starts by loading in a modified hex file of the generated program into a 4-hex sized memory. The modified software chain originally links the program to start at `0x4000_0000` (matching the system memory start in `GRLIB`), while the simulated memory starts at `0x0000_0000`, so the testbench cheats a little by taking the `MEM_ADDR` which is provided by the PicoRV memory interface, and subtracts a constant `MEMORY_OFFSET`, which is set to `0x4000_0000`. The FreeAHB master is tricked into thinking that it is communicating with the bus, but in reality, the testbench grants ideal conditions (`HREADY`, `OKAY` as `HRESP`), and performs transfers without any wait cycles.

When data is written to a special, predefined memory location, it will print that value to the simulator console.

The FreeAHB and adapter testbench

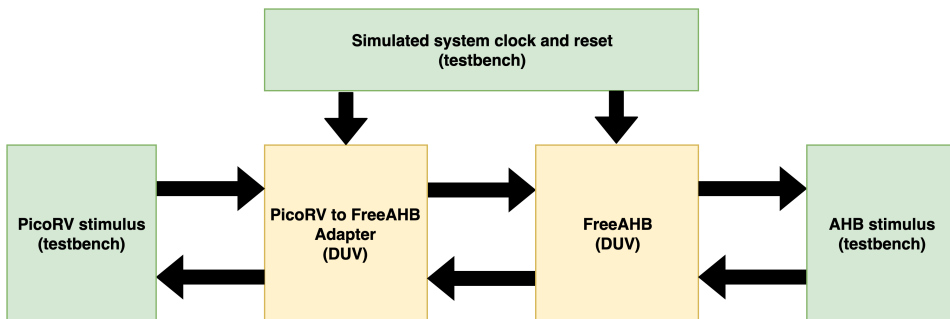


Figure 4.17: Block diagram of the testbench expanding the DUV to two modules: the adapter and FreeAHB. This is to capture the natural interaction between the two. Notice that the system clock and reset is wired directly to both. This is done to reduce signal propagation delay (instead of passing on the system clock through every intermediate module from the bus). Figure produced in *draw.io*.

The FreeAHB and adapter testbench extends the DUV to also include the adapter with the same configuration as the adapter testbench. The FreeAHB UI stimuli that has been provided in previous testbenches (see figure 4.13) is now driven by the DUV itself, but no assertions are done on those signals.

The testbench kicks off a write process by driving the memory interface signals to indicate a write to address `0x8000_0000`, data `0xF0FF_0FAA`, and write strobes set to only write the two highest-order bytes. When the FreeAHB master requests the bus, the testbench immediately grants it, and constantly drives HREADY and an OKAY response in HRESP. The testbench ends when `mem_ready` is raised, and it is intended that one checks the wavedumps afterwards for correctness.

The full RISC-V side testbench

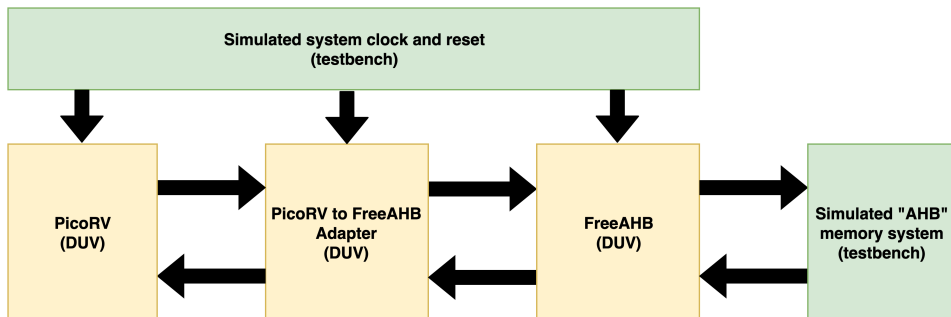


Figure 4.18: The entire RISC-V side toplevel. The only simulated behavior is what the FreeAHB sees on the "AHB bus", which gives us a pretty close look at how the toplevel design behaves in practice. Figure produced in *draw.io*.

This testbench is a derivative of the original PicoRV testbench, which we have seen earlier.

The full RISC-side testbench creates a toplevel design in which the PicoRV core, adapter, and FreeAHB master are all wired together. Now, only the AHB-signals and system memory is being mimicked by the testbench. The full testbench runs under the assumption of being little-endian, and bus conditions are - as always - simplified to immediately grant the bus and always respond with HRESP and write and read data to requested areas of memory.

It executes the same test program, but now all data has to go through the adapter and FreeAHB as well.

4.5.4 Initial simulation results and observations

One immediate observation is that we perhaps started with too small DUVs. Only looking at the FreeAHB master, for example, or only at the adapter, causes a lot of initial work, as one has to manually create stimuli on both sides, which could otherwise be provided by the actual module. In hindsight, a DUV at a complexity and size such as the toplevel testbench would be sufficient in our case, and we could then have placed assertions on the signaling between the instantiated designs, to verify that our assumptions about the design and its stimuli is correct.

By inspecting the resulting wavedumps from the testbenches, and printouts from the bare-metal programs running on the relevant testbenches, we conclude that the overall design seems to function as intended. But as we learn later in the on-chip implementation debugging, this is not necessarily the case.

4.6 Implementing a base GRLIB design on an FPGA

After some simple validation of the RISC-V side of the design, it is time to connect this design to the GRLIB system so that we can test it out in a real-world environment and perform more complex simulation and validation. First we need to have a base design running on the board, before we can extend it with our additions.

4.6.1 FPGA development concerns

On-chip verification with an FPGA can potentially require a whole lot more work, since many abstractions and simplifications we can do in simulation now actually have to be fleshed out in order to meet real-world, physical constraints. In addition, there is a potential mismatch between what we simulate, and what we meet in a real-world environment.

When developing for a simulator, you can easily introduce so-called stubs into the design. This can be clock stubs, memory stubs, interface stubs, and the like. A perhaps more illustrative word for them are "placeholders". They enable simulation, but when time comes to select a target FPGA, these stubs must be replaced with actual routing to various components. This is done extensively in GRLIB's sample design testbenches, where on-board devices are often only mimicked in the testbenches.

In addition, simulators can be cycle-accurate with idealistic clock cycles with 0 clock delay propagation to the entire design. However, when moving on to a target technology, concerns such as the speed-grade of the target technology, and clock-propagation potentially becomes a real issue. One would often want to synthesize the design, and then perform some post-synthesis simulation to check if the design meets the required timings.

More generally, place-and-routing of the design onto an FPGA requires that there are enough LUTs on the FPGA, that the design matches the available FPGA pinout, and that a design is correctly mapped onto various on-board memories correctly. We consider technology mapping out of scope, as this will be provided for us by GRLIB.

4.6.2 Finding a vendor and development board

Identifying these problems early on in the staging process, together with our supervisor, it was decided that a necessary simplification was to expand upon an existing GRLIB configuration for a target board. One significant benefit of the IP core library, is that it has support for a huge variety of boards for various vendors, with existing technology maps and simulators which accurately reflect the target environment.

Because of this wide board support, we were fortunate enough to have one such board available. The *Xilinx ZC702 Evaluation kit*, from Xilinx' *ZYNQ-7000 Application Programmable SoC* series. This also means that we are to utilize the accompanying Xilinx Vivado design suite, which comes for free in a *WebPACK edition*.

Out-of-the-box GRLIB setup on ZC702

We find the presentations of some existing figures to be very effective in order to present the system we are working with. A top-level view of the whole ZC702 board is given by 4.19, a detailed view of the ZC702's processing system (PS) is depicted in figure 4.20. Finally, the out-of-the-box GRLIB configuration for this board is shown in figure 4.21.

To build and program the ZC702, we must utilize the script generator that GRLIB provides. The script generator looks for available IP core libraries in `RISCVY_GRLIB/lib`, checks the configuration done in the graphical configuration tool, and then creates a tool-specific buildscript which can then be run in order to build the design. It is run by going into the desired design, in our case `RISCVY_GRLIB/designs/leon3-xilinx-zc702`, and then running `$make scripts`.

The script generator generates a script for Vivado 2013.4, and the board configuration depends on certain Xilinx IP from that version (such as that for the onboard Xilinx processing system). These IPs have changed somewhat in recent versions, and trying to do a run in newer Vivado versions will cause problems. Therefore, sticking to the original design suite version is strongly recommended in order to save time.

With the build script having been generated, we can now in the same folder run `$make vivado`. This will programatically run the synthesis and implementation phases of the Vivado design suite, which will take a while.

Eventually, it will produce a FPGA bitfile for the specified topdesign. The topdesign is usually called `leon3mp.vhd`, and the resulting bitfile is called `leon3mp.bit`. With bitfile in hand, we can connect to the ZC702 via the *Digilent Platform* JTAG USB connection. Making sure that the switches controlling which JTAG connection to use is set to Digilent Platform, we can switch the board on.

Running `$ make program-zc702` makes a script which is then fed to the Xilinx XMD tool, which in turn reads a *tcl* (tool command language) script which tells it what to run. It sources two files generated by the automated build process, then it programs the FPGA, sets up the Xilinx processing system, and then finally calls `init_user` to initialize some clocks for the FPGA. The problem is that the command is defunct, and we therefore replace the `init_user` call in the Makefile's XMD script creation with the more appropriate `ps7_post_config`, to make the system start normally.

We can then move on to start communicating with the FPGA design.

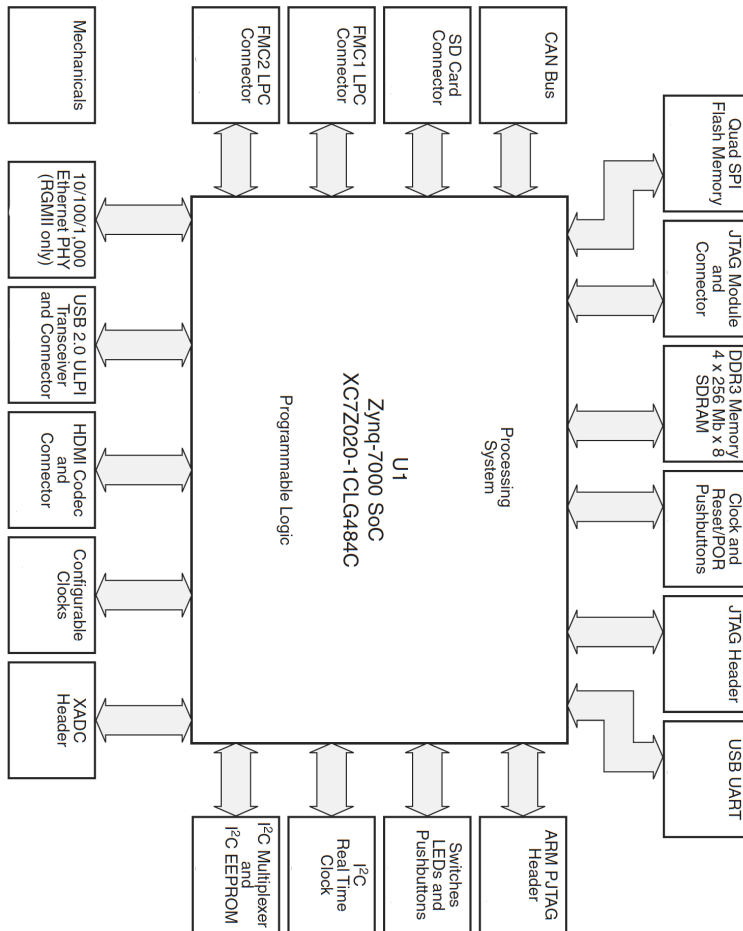


Figure 4.19: A toplevel view of the ZC702’s main components. Of note, there is DDR3 memory we can utilize for storing and executing programs. A JTAG module with a JTAG-to-USB platform which we can debug our design with, and an UART port for serial output from the board. The individual components are described more in detail in the ZC702 user guide [76]. Figure source: ZC702 user guide [76], modified by removing page numbers.

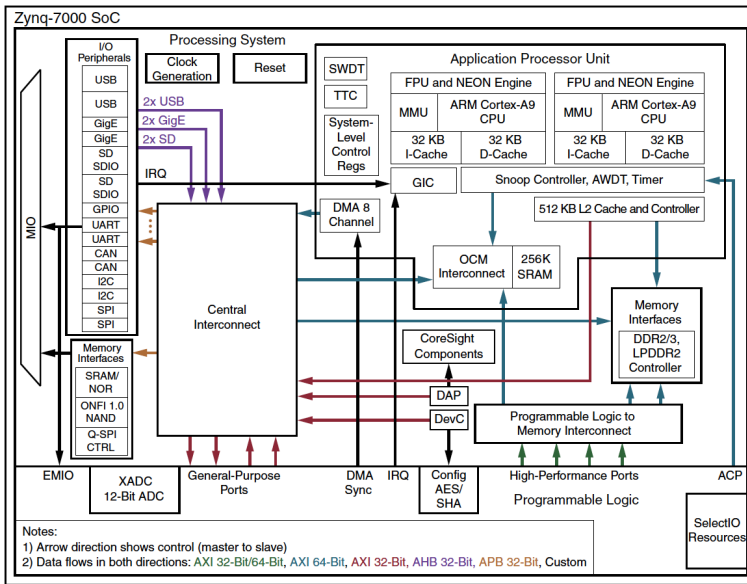


Figure 4.20: A detailed view of the Xilinx ZC702 SoC’s processing system (PS). The important takeaway is the interface between the programmable logic (the FPGA), and the rest of the processing system. The GRLIB out-of-the box design for the ZC702 uses an AHB-to-AXI module in order to interface with the high-performance ports which in turn go to the DDR memory controller. Source: ZC702 user guide [76].

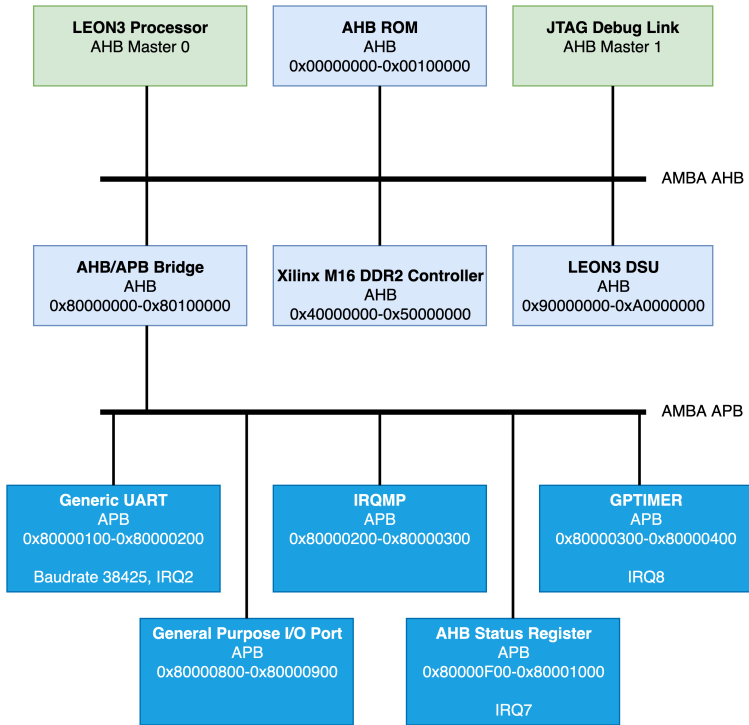


Figure 4.21: GRLIB’s default design for the LEON3 processing platform on the Xilinx ZC702 FPGA. It contains a single-core configuration of GRLIB’s LEON3 processor, a corresponding Debug Support Unit (DSU) and the AHB ROM which contains the GRLIB AMBA plug and play information. It also includes a JTAG debug link connected to the JTAG of the Xilinx ZC702, which in turn can be accessed externally via tools such as GRMON to communicate with the chip. Most of the system memory comes from a custom AHB-to-AXI4 Xilinx memory controller, which maps onto parts of the Xilinx RAM, for a total of 256MB (16^7 available addresses, each address contains one byte). A AHB/APB bus is utilized to get a generic UART, the interrupt controller for the LEON3, GRLIB’s timer unit, GPIO ports mapped to the Xilinx’s IO, and a status register for the AHB, in which HREADY is mapped onto one of the board’s debug LEDs.

4.6.3 Utilizing GRMON for JTAG-based debugging

To communicate with the GRLIB environment, the library's JTAG AHB Master (AHBJTAG [21, ch. 7]) module is in place. This JTAG module is connected to the FPGA's JTAG TAP (Test Access Port – more on this later). This TAP is part of a JTAG *scan chain* (where the other device is the on-board Arm core), which is accessible via the Digilent JTAG-to-USB platform module, which we in turn can access via our development machine.

As Cobham's documentation points out [77], a huge benefit with the JTAG master is that it also directly implements, in hardware, the JTAG protocol and the custom plug-and-play mechanic that GRLIB adds on top of the AMBA specification. This means that the JTAG module functions, even without a LEON-processor attached to the system, which is very practical in case we wish to replace it entirely with our RISC-V core.

Cobham Gaisler provides a debugging monitor to efficiently troubleshoot LEON3-based designs, named GRMON3 [77], which supports various JTAG-connections, UART connections, and various other debug links commonly found on development boards.

GRMON3 has an evaluation/academic licence. For non-academics, this licence lasts 30 days before having to purchase a GRMON3 pro licence.

Developing an application without having a debugger or communications line with your design available is arguably like driving blindfolded. It is essential for efficient application development on SoCs. We will later demonstrate how to create a debug application via JTAG, as we will find debugging a PicoRV application difficult with GRMON.

Initially when debugging our design on the ZC702, we will be running `$grmon` with the `-digilent` and the `-u` option. The first specifies to use the ZC702's Digilent JTAG platform connection, depending on the Digilent Adept 2 runtime for communications. The latter option specifies to also display the UART output. GRMON through the JTAG master sets the relevant control bit in the APBUART to enable FIFO debug mode [21], and then it utilizes UART's flow control mechanism to successfully poll all values during a LEON3 program run [77]. Using the `-u` option should render the actual UART-port unusable.

4.6.4 Running a sample program on the LEON3 processor

To compile software for a LEON-based system, Cobham Gaisler provides the *Leon3/4 Bare-C Cross Compiler* (BCC) [78]. It comes with some sample C programs which we will use to quickly test the setup. Here we demonstrate running the archetypical hello world.

First we compile the program located in `bcc/source/examples/hello` by running:

```
sparc-gaisler-elf-gcc -mcpu=leon3 -msoft=float -O2 -g hello.c -o hello
```

Then, connecting with GRMON to the sample GRLIB design flashed and running on the ZC702's FPGA, loading and running the compiled program, yields figure 4.22:

```
kris@MacGyver:/opt/bcc/src/examples/hello$ grmon -digilent -u
GRMON LEON debug monitor v3.0.15 64-bit eval version

Copyright (C) 2019 Cobham Gaisler - All rights reserved.
For latest updates, go to http://www.gaisler.com/
Comments or bug-reports to support@gaisler.com

This eval version will expire on 28/08/2019

JTAG chain (2): xc7x020 zynq7000_arm_dap
GRLIB build version: 4226
Detected frequency: 83.0 MHz

Component                               Vendor
LEON3 SPARC V8 Processor                 Cobham Gaisler
JTAG Debug Link                         Cobham Gaisler
Contributed core 1                      Various contributions
Generic AHB ROM                         Cobham Gaisler
AHB/APB Bridge                          Cobham Gaisler
LEON3 Debug Support Unit                 Cobham Gaisler
Xilinx MIG DDR2 Controller               Cobham Gaisler
Generic UART                             Cobham Gaisler
Multi-processor Interrupt Ctrl.         Cobham Gaisler
Modular Timer Unit                      Cobham Gaisler
General Purpose I/O port                Cobham Gaisler
AHB Status Register                     Cobham Gaisler

Use command 'info sys' to print a detailed report of attached cores

grmon3> load hello
40000000 .text                25.2kB / 25.2kB [=====] 100%
400064E0 .rodata              128B [=====] 100%
40006560 .data                 1.2kB / 1.2kB [=====] 100%
Total size: 26.50kB (529.48kbit/s)
Entry point 0x40000000
Image /opt/bcc/src/examples/hello/hello loaded

grmon3> run
hello, world

Program exited normally.
```

Figure 4.22: Compiling and running hello world on a LEON3-based GRLIB system running on the Xilinx ZC702.

Since the GRLIB design is standardized, and the BCC is designed with this in mind, there is very little we have to do manually to get a successful program. As the documentation mentions, it is only if we start creating custom memory interfaces, or wish to change the code located in the PROM used for initializing the LEON processor, that we need to take extra action [78].

4.7 Adding the RISC-V toplevel to the design

With the base design running nicely on the development board, we can now work towards extending the design.

4.7.1 Wrapping the RISC-V side Verilog to GRLIB's VHDL

The RISC-V part of the design is written with Verilog, while the GRLIB parts are written in VHDL. To interface the FreeAHB with the GRLIB plug'n'play AMBA bus, we have to language wrap the Verilog with VHDL, then wrap the VHDL-version of FreeAHB with the GRLIB interface.

Mixed-language support is highly vendor specific, therefore we have to follow the Xilinx Vivado flow in our case. Per an old Xilinx ISE manual [79], one declares a component with the same name as the Verilog module, and the same ports with the same name. Then, one initializes the module in VHDL using *named association* for the ports. We believe that by declaring a component, it tells the Vivado that “*something at some point will have this component name with these named ports*”. When Vivado preprocesses the Verilog that is added in the workfiles for the design, it will then end up with a module with the same name and ports that matches the VHDL component declaration. When Vivado finds no other VHDL entity with that name, we believe it resorts to resolving it with the Verilog module it found, and thus we get our Verilog into the VHDL design when synthesizing the RTL.

With a language-wrapped AHB master, wrapping that master to become a GRLIB plug-and-play AHB-master is a matter of following the wrapper template set by the GRLIB IP core library user manual [48, p.142], modifying the AMBA AHB slave example so that it suits our master instead.

The resulting wrapping code can be found in appendix F.

4.7.2 Adding the PicoRV IP to the GRLIB library files

Before utilizing the wrapped VHDL entity in other VHDL designs, we need to put it in a VHDL package. This is done in `riscvy-grlib/lib/riscv/picorv/picorv.vhd` :

```

library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.config_types.all;
use grlib.config.all;
use grlib.stdlib.all;

package picorv is
  component picorv_grlib_ahb_master is
    generic (
      master_index : integer := 0);
    port (
      rst      : in std_ulogic;
      clk      : in std_ulogic;
      enable   : in std_ulogic;
      ahbmi    : in ahb_mst_in_type;
      ahbmo    : out ahb_mst_out_type);
  end component;
end;

```

Per the GRLIB user manual [48], we need to perform some modifications to GRLIB in order to add the RISC-V IPs to the build script generators. To start, we have to create a new directory for our library. It can be anywhere on the filesystem due to indirect addressing, but a natural place is in `RISCVY_GRLIB/lib`, so we created `RISCVY_GRLIB/lib/riscv`. We add this directory to `RISCVY_GRLIB/lib/libs.txt`, which the script generator parses when running `make scripts`.

In `RISCVY_GRLIB/lib/riscv`, we need to have a `dirs.txt`, which specifies folders for VHDL packages that this library includes. Recall the folder structure for our library in section 3.5.1. We currently only have one package, PicoRV, so we put the `picorv` folder as the only line in `dirs.txt`.

In `GRLIB/lib/riscv/picorv`, we finally have to provide two files - one to specify all the VHDL files for synthesis, `vhdlsyn.txt`, and one for Verilog `verisyn.txt`. This involves our PicoRV VHDL package, the entity which includes it, the corresponding Verilog files, and the Verilog file for the FreeAHB master.

4.7.3 Initializing the wrapped PicoRV in the board design

Now, having added the library to the GRLIB script generator, and due to the very plug-and-play nature of AMBA AHB in general, aided extra with the GRLIB plug-and-play mechanics, it is very simple to add the PicoRV VHDL entity to the topdesign,

```
GRLIB/designs/leon3-xilinx-zc702/leon3mp.vhd .
```

We simply import our `riscv.picorv` VHDL package at the start of `leon3mp.vhd`, then we add the following lines to the architectural definition of `leon3mp`:

```
-- PicoRV instance
picorv0: picorv_grlib_ahb_master
  generic map (master_index => 2)
  port map(
    rst      => rstn,
    clk      => clk,
    enable   => always_enabled,
    ahbmi    => ahbmi,
    ahbmo    => ahbmo(2));
```

Finally, we have to update the `maxahbm` constant to the correct amount of AHB masters. This configuration parameter is used to drive all unused master signals to 0. We need to take care that we assign an unused master index to our RISC-V toplevel. Based on what we observed in the sample design run earlier, we know there are only two AHB masters – a LEON3 processor, and an AHB_JTAG module. Therefore, we can safely assign it as number 2 for the time being.

And that's it. We have wired our design onto the design's AMBA bus, which provides all necessary signals (clock, reset, and data) for our unit to function. We can now generate the system with `make vivado` and then program it as before. When running GRMON on the board now, we will see that the `CONTRIB_CORE1` successfully shows up in the system overview, so it has been correctly added to the plug-and-play information in the AHBROM at least. However, this does not indicate in any way that it functions correctly.

We currently have no software to run on the core, and we have not verified the entire design (RISC-V + GRLIB) in the GRLIB design testbench yet. We also have yet to connect the interrupt lines from the GRLIB to the Pico and the trace lines from the pico to anywhere useful. However, this proves that our entire proposed workflow is indeed possible.

Next up, we describe our rather lengthy full design verification process.

4.8 Implementation debugging

Tom Cargill at Bell Labs once humorously stated “*The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time*” [80]. Janick Bergeron at Synopsis further illuminates the verification problem: “*If you survey hardware design groups, you will learn that between 60% and 80% of their effort is dedicated to verification.*” [35]. We strongly found this to be the case.

4.8.1 Initial on-chip testing

Initially, with “hello world” firmware and GRLIB design in hand, we tried moving straight to the ZC702 to see if things ran successfully. It did not. The most immediate problem was that it was very difficult to verify whether or not a successful system run had taken place. The initial board design utilizes four of the onboard LEDs mapped to the DSU’s debug mode and error mode signals, the HREADY signal, and the UART1 TX signal (send). It also uses one button (aside from the full ZC702 reset button) connected to the HRESET of the GRLIB design’s AHB bus. The only debug link we have available is the AHBJTAG on the GRLIB system which allows AHB writes and reads. The GRLIB AHBUART is not connected to the ZC702’s UART according to the design’s README. Therefore, as the design is currently, it is very difficult to know what exactly is wrong on the chip when bugs occur. We have a black box, and the only debugging we really could do sometimes with the on-chip design was guesswork.

Faulty wiring

The first issue, which we found out by manually parsing the extensive Vivado build log (created by running `make vivado`), was that our RISC-V top-level had been pruned out during the implementation process, as we had connected the wrong `HGRANT` signal to FreeAHB, which essentially rendered the design inactive. We found it odd that no errors were raised by the build process.

With the right `HGRANT` index, the RISC-V top-level was now actually implemented and included in the FPGA bitfile, which could be verified by inspecting the design in the implementation table that Vivado provides in the build log. However, when the FPGA then got programmed with the bitfile, and we reset the design once (which we always do after flashing the design), the LED for HREADY immediately went LOW. In addition, one could not connect to the system with GRMON via the AHBJTAG master, which strongly suggested that the bus was being trashed. Since the bus had been working perfectly fine with only the LEON3 and AHBJTAG as AHB masters, there obviously had to be something wrong with our design additions.

After some guesses at what could be wrong, we landed on an observation we did when slightly modifying the PicoRV testbenches during design verification: if the test program was loaded incorrectly to memory (wrong offset), and there were no valid instructions on the PicoRV start address, the core would just perform sequential reads until it found a first instruction. Given that the RAM is not initially loaded with the “hello world” firmware SREC, it figures that the PicoRV would behave as we have seen before with invalid data.

The problem is that we need GRMON to be able to connect to AHBJTAG in order to write data to RAM.

We did in the design phase consider that a PicoRV fetching instructions non-stop could be a potential problem, but we did not believe that it would actually lock all other AHB masters out of the bus, as AHBCTRL is configured to grant the bus round robin style. Seeing if FreeAHB inserts the correct wait cycles for the arbiter to determine the end of transfer was one thing that could be validated. But we ended up prioritizing an enable button for the processor instead, as it would indirectly solve the entirely-blocked bus problem immediately. However, if the PicoRV is to execute lengthy programs while other AHB masters also utilize the bus, then checking whether or not FreeAHB inserts a wait cycle after an undefined length burst (which tells the arbiter that the unspecified length burst is done) will have merit again.

Utilizing on-board buttons and LEDs

To add a button to the design, we take one of the buttons from the GPIO module in `leon3mp.vhd`, and then connect that button to an enable signal instead. We extend our RISC-V toplevel to take in the enable signal, and we pass it all the way to the picorv-to-freeahb adapter. Here, extend the adapter FSM to do nothing (even on `MEM_VALID`) until the enable button is pushed. To disable again, the HRESET button must be pushed to reset the system. In addition, to get a bit more insight into what is going on, we map three of the four remaining debug LEDs to the value of the enable signal, and the HBUSREQ and HGRANT signals that are connected to the RISC-V top-level.

Building the reiterated design in Vivado and flashing it to the FPGA, things start to get better. The HREADY LED lights up, so it indicates that the bus is available while the system is idling. We can also connect with GRMON, which indicates that the AHBJTAG module can issue reads and writes to the AHB bus as well. We load the RISC-V program into RAM via GRMON. Then it should, in theory, be a press of the enable button, and we should see some UART printout in GRMON.

Instead, what we get is erratic behavior. Sometimes, HREADY goes and stays to LOW. On other runs the HREADY LED does not react at all when we enable the Pico to run the program.

Early on-chip debugging is too time-consuming

After some tedious trial and failure, we determine that this blackbox, on chip development approach is a no-go. Testing fixes to the design requires a rebuild of Vivado and a reprogram of the FPGA every time, and the Vivado build takes roughly 15-20 minutes on a quad-core laptop. It is a very inefficient and frustrating flow for testing the design. We had two approaches at this point: either build on-chip support units, or we could see if we could simulate the entire design on a mixed-language simulator.

The first approach was quickly disfavored. The idea was to provide some of the facilities that the LEON3 DSU provides, but for the PicoRV instead. This would include registers in which we could store traces, memory commands, and other relevant data that came from the PicoRV, and then address those debug memories via AHB to see if we could see what was wrong. The biggest issue we had with this idea was that it meant

adding more untested hardware to a system where we, at the time, were not sure where the problem resided.

It therefore had to be full system simulation. We had considered this approach in our pre-study before this master thesis, but when surveying the simulators supported by GRLIB - Active HDL, NCSim, Riviera Pro, ModelSim, GHDL, VCS - we did not detect that any of these had an academic version of their simulator with mixed-language capabilities. Some, such as ModelSim (GRLIBs default), have a student edition, but do not allow mixing the HDLs. This was a problem, seeing as we have our RISC-V top-level in Verilog, and GRLIB consists (mostly) of VHDL components.

However, since our only option at this point (except for excessive trial and error) was to acquire one (or start from scratch with a VHDL core and create a VHDL AHB master), we looked extra hard, and we had missed that ACTEL ActiveHDL [67] has a student edition with mixed language capabilities. The only tidbit is that it only runs on Windows (while most of our workflow has been centered around Ubuntu).

4.8.2 Full system simulation with Active HDL

ActiveHDL is Windows only, so we have to setup a skeleton workflow on a Windows machine so that we can run the testbench there. The GRLIB User Manual [48] describes how to utilize CYGWIN [69] on Windows in order to provide a huge collection of open-source tools usually found in Linux environments, which are required by the GRLIB's Makefiles in order to launch the full-system simulation.

In a CYGWIN shell, we go to `riscvy-grlib/designs/leon3-xilinx-zc702-SIM` and then run `$ make avhdl-launch`. This will setup a testbench which can run the design as specified in `leon3mp.vhd`, replacing ZC702 system components (RAM, Arm processors) with simulated variants. The original testbench runs some routines on the LEON before exiting the testbench by reporting a "failure" in the simulation. This behavior is seen at the end of `testbench.vhd`:

```
iuerr : process
begin
  wait for 5000 ns;
  wait on led(1);
  assert (led(1) = '1')
  report "*** IU in error mode, simulation halted ***"
  severity failure ;
end process;
```

To make sure that the testbench does not halt when the LEON3 routines are finished, we simply comment out the "report" line, which generates the simulation error. In addition, we edit the system's configuration, `config.vhd`, by changing the value of `CFG_DEFMST` (default bus master) to the AHB master index of the RISC-V master.

Finally, we swap out the original `ram.srec` with the `ram.srec` that can be generated by running GNU's RISC-V variant of the `objcopy` on the compiled "hello world" firmware that we have described earlier. It is entirely possible, as inspecting `leon3_zc702_stub.vhd` will reveal how the various system components are simulated. The memory system is initialized by parsing the `ram.srec` file until the end of the file, and it only inspects S3 records (32-bit addressing data records), and loads data into the specified locations. The original `ram.srec` includes startup and programs for the LEON. With this entirely replaced with the PicoRV firmware, and with the PicoRV as the default master, we have essentially modified the testbench to run our RISC-V program in the full design.

4.8.3 Errata and bugs

We made several errors in our original implementation of the RISC-V top-level. Throughout the text so far, we have provided you with the final, working versions for readability, but here we mention in short the errata and bugs we came across by combining runs in ActiveHDL on Windows, and with Icarus Verilog runs on Ubuntu. Critique of our process and possible improvements of the validation process take place in the discussion. The thorough reader can refer to the thesis repository's commit log to see the intermediate versions of the various components before the final versions.

Misunderstanding the FreeAHB UI

While the AHB study left us with a good idea of how the AHB bus functions, with its bus request phase, address phase, and data phase(s), we were initially somewhat uncertain about the mapping between this and the FreeAHB UI. This caused us to construct an incorrect FSM for the adapter in which the proper condition for a read finish (`o_dav`) and write finish (`o_next`) was not checked, causing incorrect operation. We also fed the AHB pipeline data T alongside address T (recall that the pipelined operation implies that the address information for transaction T is presented during data for transaction T-1), which causes errors on the writes (we only did single reads, so this bug did not affect it).

Finally, we also assumed in our first adapter prototype that FreeAHB initializes the next pipeline step when `o_next` is raised, and the FreeAHB then reacts when there is a change of FreeAHB UI values. A quick look at the source code (and its comments) illustrates that this is not the case. On `o_next`, FreeAHB expects that the next set of UI signals are presented immediately on the next clock. A `i_write` HIGH and `i_dav` HIGH or `i_read` HIGH (a write with valid data, or a read UI) input will indicate to the FreeAHB that this is the next address/data cycle. All other combinations cause the FreeAHB to either go IDLE or emit BUSY (if mid-burst).

This misunderstanding caused an odd bug in the ActiveHDL testbench. In our Verilog testbenches, all reads and writes completed without any wait cycles, but this was not the case in the ActiveHDL testbench, as reads could take several cycles. When a FreeAHB UI indicating the next transfer is committed to the FreeAHB pipeline, `o_next` is raised immediately, even before that read is complete (this is indicated separately by `o_dav`). Not reacting to the FreeAHB's request for the next transfer, it caused our adapter to request the same initial read to `0x4000_0000` two-three times, before the first read was returned to the adapter and subsequently the processor.

Problem is that when PicoRV then requests reads to `0x4000_0004` and `0x4000_0008`, there are already two `0x4000_0000` reads in the pipeline. So when the PicoRV requests a read from `0x4000_0004`, the adapter sets the FreeAHB UI again, and shortly thereafter also sees that `o_dav` is raised. Problem is that this is the data from `0x4000_0000` that was in the pipeline, not `0x4000_0004`, which is one or two pipeline steps behind.

This mismatch between what the processor requested, and what data the FreeAHB returned, results in incorrect operation from the start. This could have been prevented in the design verification phase, by utilizing the available `o_addr` signal which specifies which read address the data corresponds to, and placing an assertion on it. At the time, “we didn't see much point with the `o_addr` signal”. We did since.

Misaligning memory writes to the memory system

Another interpretation error was that of how byte-sized writes are done. In the AHB specification, it is stated:

“All transfers within a burst must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is $A[1:0] = 00$), halfword transfers must be aligned to halfword address boundaries (that is $A[0] = 0$).” [52]

When designing how the adapter should perform writes, our interpretation of this paragraph was that for byte-sized transfer, there was no need for alignment on the write bus. Therefore, we drove all write-bytes (A, A+1, A+2 and A+3) on the highest-order byte of WDATA, with its corresponding HADDR (A, A+1, A+2, A+3).

This assumption was not tested, and as it turned out by inspecting the wavedump in the ActiveHDL testbench, the AHB2AXI memory system assumes that all data is write aligned on word boundaries, and uses write strobes to select which bytes on an assumed-aligned HWDATA bus to write into memory. Therefore, for a little-endian PicoRV to a big-endian write bus, the AHB writes should actually be:

```
freeahb_wdata[31:24] <= mem_wdata[7:0];    \\ Byte A
freeahb_wdata[23:16] <= mem_wdata[15:8];  \\ Byte A+1
freeahb_wdata[15:8]  <= mem_wdata[23:16]; \\ Byte A+2
freeahb_wdata[7:0]  <= mem_wdata[31:24]; \\ Byte A+3
```

Timing errors in the adapter

Throughout development we saw many variants of this bug. The first example we encountered was that we failed to treat `MEM_VALID` going into the adapter properly. Recall that `MEM_VALID` is raised by the PicoRV to indicate a new memory request. In our initial FSM for the adapter, we checked to see if `MEM_VALID` was low, and if it was, we would put the adapter in an idle state one cycle later. However, the FSM was constructed so that it expected to be put into IDLE one cycle after `MEM_READY` was raised. The *Idle on !MEM_VALID*, however, takes two cycles before the adapter actually idles, not one. This caused the FSM to behave erroneously. The fix is, simply, to put the adapter to IDLE immediately in the cycle after `MEM_READY` is raised, as this is what PicoRV expects.

Running the processor before memory is properly initialized

We mentioned this problem earlier, but list it here too for completeness. A prime example of problems arising from mismatches between what we simulate, and what we actually run on. In simulation, the program SRECs were already loaded into the simulated memory before the program was run. On the ZC702, the programs are loaded into RAM (which just contains random values when powering up the board) after the FPGA is programmed. The run conditions differ. This causes the processor to request sequential addresses constantly, thrashing the bus, and this renders us unable to load the program to RAM via AHB system bus.

PicoRV UART writes cannot be observed

Recall that the APBUART is not connected to the ZC702's JTAG port, and that by running `$ grmon -digilent -u`, we put the APBUART in FIFO debug mode. When a LEON program is run via GRMON, it will constantly poll the APBUART during the program run. GRMON does not pull the APBUART otherwise, and GRMON is one of the closed-source components from Cobham Gaisler. Therefore, we do not currently have any method of polling APBUART (nor writing to it), so we currently have no means to communicate with our RISC-V centered system.

As a first effort to see that the PicoRV actually successfully writes, we modify the "hello world" firmware to write to a fixed byte of RAM instead of to the UART. It is not very effective, as we can only observe the last ASCII character being written (a 0x0A, the newline character) since the program runs in under a second on chip, but we can at least see that the program effectively runs.

4.8.4 State of the implementation

When the PicoRV master is the default bus master, the system works when mitigating the bugs discovered in the implementation debugging phase. However, it is still desirable to communicate with our system via the UART in debug mode, and we have to test the interrupt behavior of the system as well.

To facilitate on-chip communication with the PicoRV, we will now look into the ZC702's Digilent Adept JTAG port, and its connection to AHBJTAG, in order to create a software application which can set the APBUART in debug mode, and then receive a "hello world" from the PicoRV.

4.9 Creating a JTAG application for UART communication

4.9.1 The JTAG standard

Since we are unable to communicate with APBUART via the ZC702's UART, we have to use the debug link to the GRLIB design which we have available – the AHBJTAG AHB master.

The JTAG protocol is described in the IEEE 1149.1 standard [81], and we only mention the relevant parts here. It defines a standardized on-chip interface which can be used to debug entire designs, or specific components. A JTAG environment consists of one or several *Test Access Ports* (TAPs), which can be *daisy-chained* together and form a scan-chain. A TAP consists of an instruction register (IR), multiple data registers (DRs), and a standardized state machine for controlling its operation. All registers are FIFO shift registers.

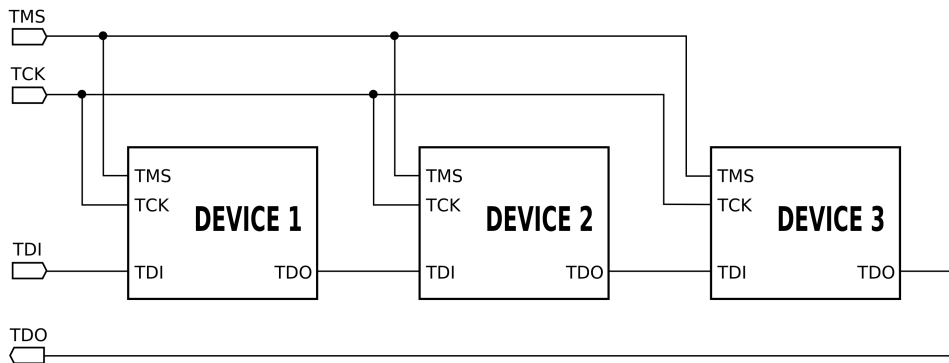


Figure 4.24: A sample JTAG scan chain. Figure made by Grzegorz Pietrzak [82].

The TAPs are connected together via two signals: *Test Data Input* (TDI) contains values shifted into a TAP, and *Test Data Output* (TDO) contains values shifted out of a TAP. The TDO of one TAP is connected to the TDI of the following TAP. The end result is a large chain of shift registers.

One bit of TDI is shifted in, and one bit of TDO is shifted out, on each clock of the JTAG clock, TCK. Which TAP register (instruction or data) that is connected to TDI and TDO depends on the controller state. The TAP controller state machine is controlled by the *Test Mode Select* (TMS) signal, which is also shared between all TAPs.

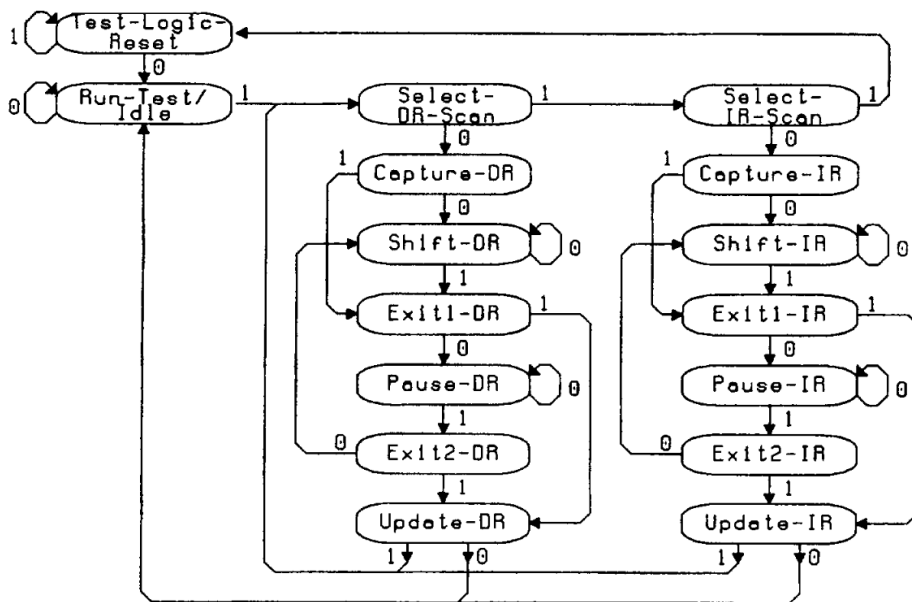


Figure 4.25: The state diagram for the JTAG TAP controller. The state machine moves into the next state on each rising edge of TCK, depending on the value of TMS. Figure from the IEEE 1149.1 standard [81].

Normal operation usually starts off with resetting the TAPs. This is done by holding the TMS line high for five rising edges of TCK. This causes the TAPs to load a default instruction into its IR registers. Usually, this is either the *BYPASS* instruction, which connects a single register with a '0' value between TDI and TDO, or the *IDCODE* instruction, which connects a 32-bit register identifying the TAP.

Then, the state machine is set to perform the desired function. Commonly, one puts it in the *RUN-TEST/IDLE* state when one does not wish to change the current state. If one wishes to change the current instruction, one goes to the *SHIFT-IR* state, shifts in the data, and then change state back to *RUN-TEST/IDLE*, passing the *UPDATE-IR* state on the way. Then, one can shift in (and/or out) data to the corresponding data register in a similar fashion.

4.9.2 The ZC702 JTAG scan chain

The Xilinx ZC702 comes with three ways to connect with on-board JTAG scan chain. We utilize the *Digilent HSI platform* USB connection, which allows us to connect from our development machine using the Digilent Adept 2 runtime and a microUSB cable.

The Digilent Adept 2 software family also has a utility package and an SDK package. We install both. One of the utilities, `djtgcfg` (the Digilent JTAG configuration utility), can be used to discover the available JTAG devices.

By running `djtgcfg enum`, we get a list of Adept-supported JTAG devices that is reachable from our system. If we have the ZC702 powered on, the on-board JTAG select DIP-switch (SW10) set to "Digilent", and connected a microUSB to it, the utility should identify it as *JtagSmt1*.

Then, we can run `djtgcfg init -d JtagSmt1`, and it will display the IDCODEs of the devices in the ZC702's scan chain. We get two device codes, *4BA0 0477* (Arm DAP (Arm calls it Debug Access Port)), and *2372 7093* (Xilinx TAP). While IDCODEs are great, we will still have to know the length of the instruction registers in order to utilize them. By searching around for manuals, we find IR lengths in the Zynq-7000 SoC Technical Reference Manual [83, ch. 27] (The ZC702 is an evaluation board in the Xilinx Zynq-7000 series). The Arm DAP's IR is 4 bits, while the Xilinx TAP is 6 bits.

The technical reference also lists four custom, implementable instructions (USER1, USER2, USER3, USER4) with corresponding data registers (where their width depends on the implementation, according to the closely related FPGA configuration manual [84]).

4.9.3 The GRLIB AHBJTAG module

AHBJTAG is a part of the GRLIB IP core library, and is instantiated in the base GRLIB design for the ZC702. It is described in the GRLIB IP core manual [21].

The module is an AHB bus master which can perform AHB reads and writes. It does so by implementing two JTAG instructions. One, it calls the *JTAG Debug Link Command/Address register*, and maps by default to the USER1 instruction of the Xilinx TAP. The accompanying data register is used to specify the AHB address for the transfer, the transfer size, and a bit for indicating whether it is a write or a read.

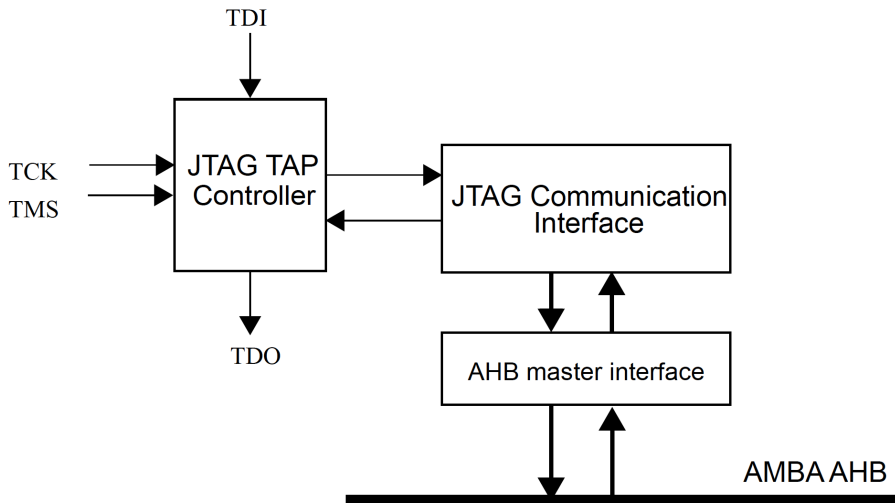


Figure 4.26: The AHBJTAG module. It allows a JTAG debugging routine to access the AHB system, by accepting instructions from a JTAG TAP, and performing AHB master bus accesses on behalf of it, and then returning the data. Figure from the GRLIB IP core manual [21].

The other instruction is the *JTAG Debug Link data register*, maps onto the USER2 Xilinx TAP instruction, which has a 32-bit data field and a SEQ bit.

When requesting a read via the command register, the access will start immediately when the TAP controller enters UPDATE-DR. Then, one is expected to go into the USER2 instruction, and shift out the data from the data register. If the SEQ bit is set, the access is finished and the data is valid. Else, one is expected to exit shifting, go to the IDLE state, and then retry reads to the data register until the SEQ bit is set. If one then wishes to read to the next consecutive address, one is expected to shift in a '1' to the SEQ position.

For writes, one specifies the control information in the USER1 instruction. Then one has to start the write by writing in the payload and setting the SEQ bit in the USER2 instruction. Afterwards, it is the same procedure with shifting out data until a SEQ '1' is seen to confirm that the transfer has finished.

The connection between the AHBJTAG and Xilinx TAP is more complex than depicted here, but thanks to the already provided technology mapping by GRLIB, we do not need to concern ourselves with this.

4.9.4 The JTAG application: UartMonitor

Knowing our environment better, we can start looking into the Digilent Adept 2 SDK. The SDK comes with programming manuals, APIs and code samples for a huge variety of debug links. We are only interested in DJTAG. The DJTAG API – which allows a programmer to manipulate TCK, TMS, and TDI, and to retrieve TDO bits – is described in the DJTAG Programming Manual [70]. The demo code runs a simple program, which enables a connection to the JTAG port, resets the scan chain, and then shifts out IDCODEs until it receives a word of zeroes. Finally, it prints out the IDCODEs in the correct order.

We expand upon this demo in order to create *UartMonitor*. The source code can be found in appendix E. First, we reset the TAP controllers yet again. Then we set the clock speed of TCK to 1MHz to match the default done by GRLIB when it connects via Digilent JTAG [77].

The expanded code includes functions which make it possible for UartMonitor to communicate with AHBJTAG via the Xilinx TAP, so that we can issue AHB reads and writes from the UartMonitor on our development machine. With these functions, we set the GRLIB APBUART's control register to FIFO debug mode and to raise an interrupt upon receiving data. FIFO debug mode disables the sending of transmitter data based on the standard UART CTS and RTS signals, and allows us to read transmitter FIFO data via the FIFO register, and write receiver FIFO data, which is useful when an actual UART connection is not actually in place.

Then, we constantly poll the status register of APBUART and check the transmitter empty (IE) bit. When the UartMonitor detects that it is not, it will perform a read to the FIFO debug register in order to read out the transmitter data. If it is the ASCII "ENQ" character, this signals that PicoRV requires input, and it will prompt the user for a character and send it back to APBUART. Else, the character received will be printed out to the user. In this manner, we can successfully communicate with the PicoRV.

4.10 Writing dedicated firmware

With UartMonitor, we can now communicate with the GRLIB system, and receive a "hello world!" if we want. Focus is now directed towards how to write extended bare-metal programs for the system, a stretch farther than "hello world".

4.10.1 What we wish to run

With the PicoRV32 successfully integrated into the GRLIB system, proven by running the simple "hello world" program with the PicoRV32's C++ demo code, it will be desirable for us to create and run a firmware to show that the system can actually do something. Namely, we want a firmware that can initialize IP cores in the GRLIB system, print out to some console, read from some standard input, run an interrupt routine such as a button press on the board, and react to a timer event. This is the bare minimum that shows that further work can be done, such as working towards adding hardware abstraction layers, privilege levels, and everything else required to get an embedded OS running on top.

4.10.2 Recompiling C programs for our target

At the onset, we are running on bare metal (no operating system). On bare-metal in a specialized hardware design such as our PicoRV32 GRLIB design, there are no prebuilt standard C libraries available (as they are highly OS and architecture specific [85]). Consider again the classical "Hello world!" from *The C Programming Language* [85]:

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Here, the `printf` call refers to a function in the `stdio` library, including system calls to `stdout`, which is highly vendor specific. What do we do when there is no out-of-the-box standard library we can utilize?

One approach, as suggested by technical blog *Freedom Embedded* [86], is to hardcode the printing directly to the UART:

```
volatile unsigned int * const UART0DR = (unsigned int *)0x101f1000;

void print_uart0(const char *s) {
    while(*s != '\0') { /* Loop until end of string */
        *UART0DR = (unsigned int)(*s); /* Transmit char */
        s++; /* Next char */
    }
}

void c_entry() {
    print_uart0("Hello world!\n");
}
```

`c_entry` would then be where our processor starts off after running an initial setup routine (to initialize CPU registers, the stack pointer, and other memories if they exist). Here we have no calls to any system libraries, so it only needs proper linking for its target system.

While this is a simple and illustrative example, trouble occurs if you wish to recompile existing programs, such as the Dhrystone benchmark. These programs depend on standard C libraries and system calls, and when trying to compile to your exotic target which does not have the system calls in place, the compiler will start complaining once unimplemented system calls appear. Since it is rather tedious and error-prone to try and rewrite existing programs, it is favorable to have a toolchain which can fix the library issues for you. This is where Newlib comes in.

Newlib

Newlib [71] is a C library which is designed for being used in embedded applications. It does so by exposing a small set of syscalls stubs to the application programmer. The system calls, and the expected minimal implementations, are described in the Newlib syscalls documentation [87].

Which syscalls need implementing depend on the target C/C++ application. These stubs require to at least be implemented with some dummy value. If the target system does not have a file system for example, then there is little need to implement calls to `_open` (open a file) or `_read` (read a file). For our prototype firmware, we do however need to implement the `_write` and `_read` system calls, as we wish to read from stdin and write to stdout.

4.10.3 The GNU Toolchain for RISC-V

To simplify matters further, the widely used GNU toolchain [66] is available for cross-compiling C and C++ programs from our x86 development machine to our target RISC-V32-IC ISA. In addition, it comes with Newlib included. One can either follow the build instructions in the GNU toolchain repository to build the desired architecture target (RV32I[M][C]), or one can utilize a practical Makefile in the original PicoRV repository,

```
riscv-glibc/lib/riscv/picorv/core , and run make build-tools . This will build all GNU toolchain combinations of RV32I[M][C], and place them in respective folders in /opt on your system. This requires some prerequisite build packages to be installed, see the README in the original repository [41].
```

4.10.4 Configuring GRLIB IP core peripherals

Before writing firmware that initializes system peripherals, we need to know how to do it. This process consists mostly of consulting the GRLIB IP core manual for the corresponding IP cores and determining how to configure them to our purpose. This also requires us to slightly modify the hardware design so that it suits our needs. We present our final configurations on both the software and the hardware side here.

GRGPIO - React to button events

GRGPIO has been configured in the ZC702 hardware design (`riscv-grlib/designs/leon3-xilinx-zc702/leon3mp.vhd`) to have access to only one button, SW7. The initial, out-of-the box design was configured to have eight switches and four buttons. Consulting the ZC702 user manual's user I/O section [76, p.44], we only have two user pushbuttons available (SW5 and SW7). There is also one DIP (Dual In-line Package) switch, SW12. There are some other pushbuttons and DIPs, but these are either related to the ZC702 board operation, or are connected to the board's PS (the on-board Arm core). We therefore wonder where all the other buttons and switches have gone.

We set the `iflagreg` generic of GRGPIO to 1. This implements GRGPIO with the *Interrupt Available* register (displays which GPIO lines can interrupt) and *Interrupt Flag* register (displays which GPIO line has generated an interrupt, if any, which then can be cleared by writing a 1 to the position) [48].

With these registers, we can now program the board. Then we connect to it with GRMON. First we write `0xFFFFFFFF` to the GRGPIO *Interrupt Mask* register (IMASK, enable interrupts from all GPIO lines), then we start pushing buttons, and flipping DIP-switches. The only buttons that trigger an interrupt and seem to be working are SW5 and SW7, the user pushbuttons.

So that is what we have to work with. SW5 is already tied to HRESET (AHB bus reset), so it is not really suitable for GPIO. That leaves SW7. SW7 is already used for enabling the PicoRV, but the PicoRV is only enabled once (it is permanently enabled on the first SW7 press until the AHB system bus is reset), so we can still use that button. We reconfigure GRGPIO again in the hardware definition, leading to the final configuration:

```

grgpio0 : grgpio
  generic map (pindex => 8, paddr => 8, imask => 16#0001#,
              nbits => 3, pirq => 6, irqgen => 1, iflagreg => 1)
  port map (rst => rstn, clk => clk, apbi => apbi, apbo => apbo(8),
           gpioi => gpioi, gpioo => gpioo);

enable_and_interrupt_button : inpad
  generic map (tech => padtech, level => cmos, voltage => x18v)
  port map (button(1), gpioi.din(0));

do_not_care1 : inpad
  generic map (tech => padtech, level => cmos, voltage => x18v)
  port map (button(2), gpioi.din(1));

do_not_care2 : inpad
  generic map (tech => padtech, level => cmos, voltage => x18v)
  port map (button(3), gpioi.din(2));

```

We only found one button we could use, but the Vivado implementation script raises an error that stops the build if the other buttons (which we have not found physically) are not instantiated, so we just instantiate them in the design to get past the error.

Therefore, the *imask* generic is only set to enable interrupts for the first button. *irqgen* is set to 1, which means it will only generate interrupts on a single interrupt line. The interrupt line is number 6.

That's all for the hardware. Now we move onto runtime configuration. There are three GRGPIO registers we must write to at runtime:

1. *IMASK* - The interrupt mask register (offset 0x0C). Mentioned earlier. Writing a 1 in a bit location will enable interrupts for the corresponding GPIO line. In this case we only have one (real) GPIO line, line 0, which is connected to SW7.
2. *IPOL* - The interrupt polarity register (offset 0x10). Writing a 1 in a bit position here indicates that the corresponding GPIO line is to interrupt on high voltage or on the rising edge (depending on whether it is a level interrupt or edge interrupt).
3. *IEDGE* - Interrupt edge register (offset 0x14). Asserting a GPIO line's bit here indicates that it is an edge interrupt. Leaving it at 0 indicates a level interrupt.

The firmware enables SW7's GPIO line, line0, by making sure that the interrupt is enabled in *IMASK*, specifying to *IPOL* that it has a high polarity, and finally write to *IEDGE* and indicate that it is an edge interrupt. The end result is that we get a pulse interrupt on the rising edge of the button press (when the button is pressed down).

IMASK is reset to 0 at HRESET, while *IPOL* and *IEDGE* keep their values. It does not really cause any problems for our firmware, as the firmware sets these values at every program run.

GPTIMER - Timer interrupts

GPTIMER is best illustrated with its diagram found in the GRLIB IP core manual [21, ch.31]:

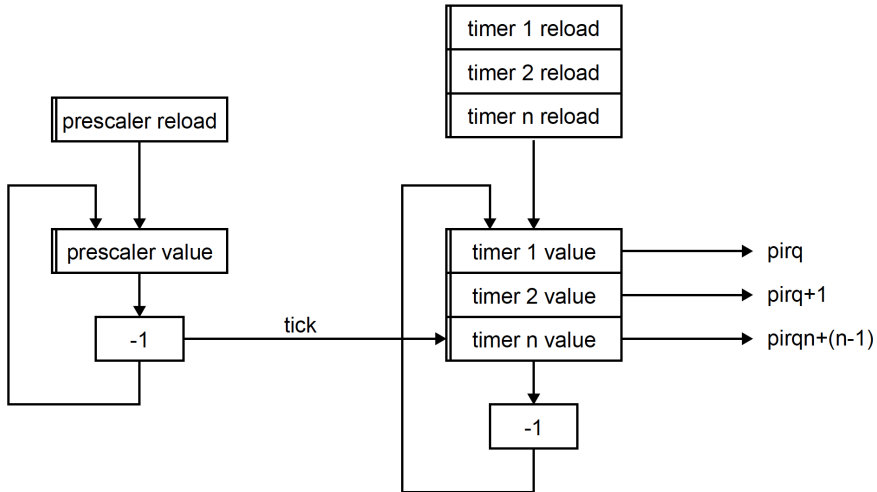


Figure 4.27: GPTIMER diagram from the GRLIB IP core manual [21].

GPTIMER is connected to the bus clock, which is 83MHz in the ZC702 design. The clock first goes into the prescaler. The prescaler ticks down a set number of times, once for each clock tick (at 1/83MHz), and when the prescaler underflows, it sends a tick to the timers, and then restarts from a set reload value.

GPTIMER can be configured to have up to seven timers. Each timer can have their own countdown value, and can be set to either share an interrupt line, or have their own. When the prescaler ticks, the values held in the timers are decremented once. When the timer(s) underflow, they can raise an interrupt [21, ch.31]. The timer is only restarted if the relevant control bits are set.

Our firmware is only meant as a proof of concept. We therefore wish to have only one timer which can keep track of seconds for us. This results in the following hardware definition (we have substituted configuration variables):

```
gpt : if 1 /= 0 generate
  timer0 : gptimer          -- timer unit
  generic map (pindex => 3, paddr => 3, pirq => 8,
    sepirq => 0, sbits => 8, ntimers => 1,
    nbits => 32, wdog => 0)
  port map (rstn, clkm, apbi, apbo(3), gpti, gpto);
  gpti <= gpti_dhalt_drive(dsuo.tstop);
end generate;
```

The *pirq*, the base interrupt that will be driven, is set to interrupt line 8. All timers will drive the same interrupt (*!sepirq*). The scaler value register is 8 bits (*sbits*), while the timer value register is 32 bits (*nbits*). There is only 1 timer (*ntimers*), and we do not utilize the watchdog timer in this design.

That is all for hardware, now we just need to configure relevant registers at runtime with our firmware. The relevant GPTIMER registers are:

- *SRELOAD* -Scaler reload value register (offset 0x04). This is the value that gets reloaded to the scaler every time it underflows. Defines the number of system clock ticks it takes to produce one timer tick.
- *TRLDVALI* - Timer 1 Reload Value Register (offset 0x14). The value that will be reloaded to timer 1 when the timer is enabled or restarted.
- *TCTRL1* - Timer 1 Control Register (offset 0x18). Has three relevant bits we wish to set when starting the timer: The interrupt enable bit (IE, index 3), the load reset value bit (LD, index 1), and finally, the enable timer bit (EN, index 0). We do not utilize the reset bit, as we only want the clock to interrupt once for our implementation.

Finally, we have to determine what values the prescaler and timer1 must use in order to get an interrupt at 1 second for a 1MHz bus. If we prescale as much as possible, with a value of 0xFF (255), we can utilize this simple formula, by inserting the amount of seconds, *x*, and then round to the nearest cycle to attain our timer value:

$$timervalue = \frac{x \cdot 83'000'000}{prescalervalue}$$

For one second, this should return a timer value of 324218, or 0x18BC62 in hex.

In our firmware, we set *SRELOAD* and *TRLDVALI* at startup, since this is never changed during the firmware. When the firmware demonstrates timer interrupts by counting, we simply have a loop that iterates five times. In each iteration, we set *TCTRL1* to load the reset value into the timer, interrupt enable, and finally enable the timer. We call `waitonirq` in the firmware. When the interrupt is called, the interrupt handler is empty (minus the if-checks done on the interrupt bit vector) and we print out the count in the firmware upon return from the interrupt handler. Then repeat the process until we are finished with counting.

APBUART - Requesting data

No hardware changes are needed for the APBUART (only setting the correct interrupt line).

We wish to receive some input from the user during the run of the program. We could request the input from the user, and then constantly poll the status register of APBUART and see if the receiver count changes. But we try to busy-wait as little as possible. Therefore, we set the receiver interrupt enable bit (index 2, RI), which triggers an interrupt when a character is received on the receiver.

A keen eye will notice that the firmware (presented shortly) sets not only the RI bit, but also all the other control bits that is supposed to be set by UartMonitor only (FIFO Debug mode, receiver enable, transmitter enable). The better way to do it would be to load the control register, OR in the one bit we wish to set, and then write the result back to the register.

4.10.5 The firmware

With the toolchain in place, knowing how to initialize GRLIB IP core system peripherals and with newlib as part of the mix, we can turn our attention to the final firmware. Luckily, Clifford Wolf (the PicoRV creator) has already made some complete and great examples in `PICORV/scripts/cxxdemo` and `PICORV/firmware` released into the public domain. The following software is built upon these examples, adapted to fit our hardware.

Initializing the processor: `start.s`

`start.S` is an assembly file, written in RISC-V assembly [88]. It first sets the interrupt mask register to all zeros (enable all interrupts), before jumping to the main startup routine. The main startup routine zero-initializes all of the PicoRV's registers, except the `x0` register, which is always 0. Then it sets the stack pointer, and pushes some zeroes on that as well, before jumping to the main program. Both the initial position of the stack pointer and the location of the main program is defined by the firmware linker script.

It is vital that this is always included in our bare-metal applications, as we can never assume the registers to be zero initially. Alternatively, they can still have register states from earlier runs, so zero initializing is always a good idea.

`start.S` also contains a wrapper which makes sure to store register state before entering the interrupt handler. One drawback in our current implementation is that we do not handle other memories/shared resources which might be overwritten. For instance, the memory area which is defined by Newlib for `stdin/stdout` will be overwritten by the interrupt handler (as some of the handlers write to `stdout`). So, if an interrupt occurs in the middle of a print, it is very likely that when the handler returns, the `stdin/stdout` buffer will now contain characters from the handler's prints, thus interrupting the normal flow.

Lastly, we add a small section which calls PicoRV's custom `waitirq` (wait for interrupt) instruction. This is used by the firmware in order to easily call this special assembly macro.

Configuring peripherals and running the main routine: `firmware.cc`

This is the main application we wish to run on the bare-metal system. The original variant first tests printing via C's `printf`, then via C++'s `std::cout`. It then tests utilizing the stack by creating two objects and calling some of their functions. Then it creates a vector of integers and sorts it, before printing the sorted numbers out to `std::cout`. It ends with an "all done". This is the behavior we saw previously in `verification/topdesign_tb` and `verification/pico_tb`.

For the final implementation, the firmware has been extended to utilize the available functionality in our target system (using what we know about the GRLIB peripherals), and then running a while loop at the end of the firmware which tests all three peripherals indefinitely.

The firmware code is located in appendix G.

The gateway to Newlib: `syscalls.c`

`syscalls.c` is our entrypoint to Newlib, in which we implement system calls per the recommendations in the Newlib syscalls documentation [87].

The minimal implementation was already given in the PicoRV repository [41]. We will only do some writing to stdout, and reading from stdin, therefore we have expanded upon the `_write` and `_read` system calls, which is for writing and reading files respectively. Let's first look at `_write`:

```
ssize_t _write(int file, const void *ptr, size_t len)
{
    volatile const void *eptr = ptr + len;
    volatile char characterToWrite;
    volatile int wordToWrite;
    while (ptr != eptr) {

        /* We require that UART transmitter FIFO must be empty (TE)-
        * before we write. This could be relaxed to when the transmitter
        * FIFO is not full (!TF) for better performance.
        *
        * Payload is reversed since APBUART registers are big-endian.
        */
        if (*(volatile int*)0x80000104 & (1 << 26)) {
            characterToWrite= *(char*)(ptr++);
            wordToWrite = 0x00000000
                + ((volatile int)characterToWrite << 24);
            *(volatile int*)0x80000100 = wordToWrite;
        }

        return len;
    }
}
```

The system call receives three arguments: a file number, a pointer to the start of the string to write, and the amount of characters to write. We ignore the file number, as we have not implemented any sort of file system or any file protections. If we were to make one, we could store a set of file descriptors somewhere, and then use the file number in order to look up the descriptors for the corresponding file.

The write routine waits for the APBUART status register's `TE` (transmitter empty) bit to be set, before it performs a write, then busy-waits by polling the status register until it indicates `TE` again.

There are some obvious downsides to this approach. First, we could relax the write condition to be `!TF` (transmitter not full), which is possible since this is a FIFO shift register, which can hold a couple of characters at a time.

The second problem is the busy-waiting, which causes extra strain on the system bus. A solution to this would be to enable bit 9, `TF`, of the APBUART control register. This enables a level interrupt from the transmitter FIFO, which triggers whenever the transmitter is less-than-half full [21, ch. 17]. If we receive this interrupt, we would need to mask it off in the interrupt handler (elsewise the interrupt handler will be called continuously). Then, we can perform a read to the status register and see the current data count in the transmitter FIFO (`TCNT` bits), and then write as many characters as we can without shifting out any untransmitted character from the transmitter FIFO. At the end of the write, we can check to see if the FIFO is full. If it is, we can unmask the APBUART interrupt line, and call `waitforirq`.

Then we have `_read`, which is called when we request a character in our firmware with `getchar()`:

```
ssize_t _read(int file, void *ptr, size_t len)
{
    /* Request the user to input character by sending ENQ character */
    *(volatile int*)0x80000100 = 0x05000000;

    /* Wait until we receive an interrupt (from the UART, hopefully) */
    WaitForInterrupt();

    /* Read out the received character. */
    volatile char receivedCharacter;
    volatile int receivedWord;
    receivedWord = *(volatile int*)0x80000100;
    receivedCharacter = (char)(receivedWord >> (8*3)) & 0xff;
    *(char*)(ptr) = receivedCharacter;

    return 1;
}
```

This is a very crude, but simple implementation. We looked up the ASCII table, and found that there is a defined character for requesting a response from the receiver, namely ENQ (0x05). We send this character to the receiver, then wait until APBUART interrupts. The interrupt is not checked in this minimal application, so a button interrupt from GRGPIO would trick this syscall into thinking that it can read from APBUART when there might be no data there.

Note that the processor will stall if the user fails to respond to the inquiry. It also assumes that only functions that request one single character from STDIN are invoked.

With these two syscalls in place, we can write firmware programs which have simple user I/O functionality available.

Handling the interrupts: irq.c

The PicoRV interrupt controller and handling is very simple in structure. The controller is very simple in that it only masks ingoing interrupts. If multiple interrupts occur between the time that the first interrupt occurs, and the interrupt handler wrapper in start.S is invoked, then the interrupt handler is supposed to handle all the relevant interrupts at once.

```
#include <stdio.h>

uint32_t *irq(uint32_t *regs, uint32_t irqs)
{
    // Use prints sparingly, as they make the handlers much slower.
    //printf("\n\n===== INTERRUPT HANDLER CALLED =====\n");

    if ((irqs & (1<<5)) != 0) {
        printf("INTERRUPT HANDLER: Received data in APBUART! \n");
    }

    // GRGPIO IRQ (Xilinx ZC702, SW7 button)
    if ((irqs & (1<<6)) != 0) {
        printf("INTERRUPT HANDLER: Registered button press! \n");
    }

    // AHBSTATUS IRQ -- not implemented
    if ((irqs & (1<<7)) != 0) {
        printf("INTERRUPT HANDLER: The AHBSTATUS interrupts! \n");
        printf("it wants to tell you.\n");
    }

    // GPTIMER IRQ (it ticks!)
    if ((irqs & (1<<8)) != 0) {
    }

    //printf("===== EXITING INTERRUPT HANDLER =====\n\n");

    return regs;
}
```

As one might see, interrupt prioritization and quick interrupt handling is a bit difficult in this non-nested scheme. A crude interrupt prioritization scheme is to check the topmost priority interrupt first in the handler. However, if a higher-priority interrupt was to occur while the interrupt handler is invoked, then it will be blocked until the handler has finished. There is no support for nested interrupts.

Therefore, writing minimal handlers for PicoRV is crucial, and the shorter the better. We have not tested out anything more crucial than timers and user I/O, but we found that having a large amount of console prints in the handler will significantly increase handle time. This is easily explained by the nature of our system. We have already displayed the `_write` stub in a earlier section. This writes characters out one character at a time, and there is significant latency since it has to wait for a recipient to read out the characters on the other side before it can continue. An external factor possibly directly influences the interrupt handler response time, which is not good. Therefore, user I/O in the handler should be avoided when response is critical.

Wrapping it all together: `riscv.ld`

`riscv.ld` is originally a work included in the RISC-V GNU toolchain. By referring to the GNU ld documentation [58], and keeping in mind that linking is – in very coarse terms – just resolving various symbols and defining where to put them, we can construct the firmware with the layout we want.

We place the compiled `start.o` first at `0x4000_0000`, followed by `irq.o`. Placing the `firmware.o` at `0x4001_0000` leaves more than enough space to fit `start.o` and `irq.o` (the linker would complain otherwise). Topmost, we define the stack pointer address, and the `_ftext` symbol, which is then used to resolve the corresponding symbol references in `start.o`. We also make sure to make the ENTRY point to the starting symbol of `start.o`, `reset_vec`.

4.10.6 Compiling the firmware

Finally, it is a matter of bringing it all together. `start.S`, `firmware.cc`, and `irq.c` has to be compiled without the standard libraries first, as this will be linked in by the `riscv.ld` linker script. In the software folder, we have a Makefile which does this:

```
...

firmware.elf: firmware.o syscalls.o start.o irq.o
    $(CC) $(LDFLAGS) -o $@ $^ -T riscv.ld $(LDLIBS)

irq.o: irq.c
    $(CC) -nostdlib -o irq.o -c irq.c

start.o: start.S
    $(CC) -nostdlib -o start.o -c start.S

...
```

Finally, we need to obtain a SREC file which we can use to load our program into RAM by using GRMON. This is also done in the Makefile with `make firmware.srec`:

```
...

firmware.srec: firmware.elf
    $(RISCV_TOOLS_PREFIX)objcopy -O srec --pad-to=0x40100000 \
    --gap-fill 0 firmware.elf firmware.srec

...
```

Note that we take care to zero-pad sections, and a bit of space above the end of the linked program, in order to assure that heap space is zeroed. When the ZC702 is powered up, there is usually only random values in the RAM, while the compiled program assumes zeroed-out memory (not having it leads to errors). We have not had the time to change the linking so that this is not assumed.

While we have more memory available (the stack will not need that much space), keeping the SREC small cuts down the time it takes to load the program into RAM. A 3MB version takes a better part of a minute via GRMON (which transfers via AHB/JTAG), so 100MB (almost half the available memory) simply takes too long for efficient development.

Results

By following the implementation depicted in the previous chapter, we have managed to end up with a functioning prototype which shows that RISC-V can indeed function in a GRLIB system. Some assembly required.

5.1 Final design

Our final, tested implementation, is depicted in figure 5.1.

The LEON3, and its related debug support unit and interrupt controller is gone. While the LEON3 could have stayed on chip as it does not run unless explicitly told to (via GRMON, for instance), removing this redundant hardware results in faster simulation and implementation times.

The interrupts driven by some cores have been moved. GRGPIO is only connected to one button, SW7. GPTIMER has one timer configured which interrupts after one second when enabled. APBUART interrupts upon receiving data. AHBSTATUS is not really used. AHBJTAG turned out to become our main way of communicating with the design, as the design's APBUART does not seem to be connected to the development board's UART.

GRMON is used for loading software to RAM, and to perform AHB reads to status registers during debugging. For program runs, GRMON is shut down, and we delegate the JTAG debug link to our small UartMonitor routine. The monitor puts the UART in FIFO debug mode, and the PicoRV's firmware makes sure to check that the register is empty before writing. This crude but simple flow control ensures that we manage to read out all printout from PicoRV (elsewise, the processor would print out all characters at a faster pace than our monitor could consume them, ending in UART transmitter FIFO overflow/valid data being shifted out).

UartMonitor can also respond to input requests from PicoRV. PicoRV first sends a "ENQ" ASCII character, and then waits for a receiver data interrupt from the APBUART. The monitor must respond to this enquiry, or the program hangs.

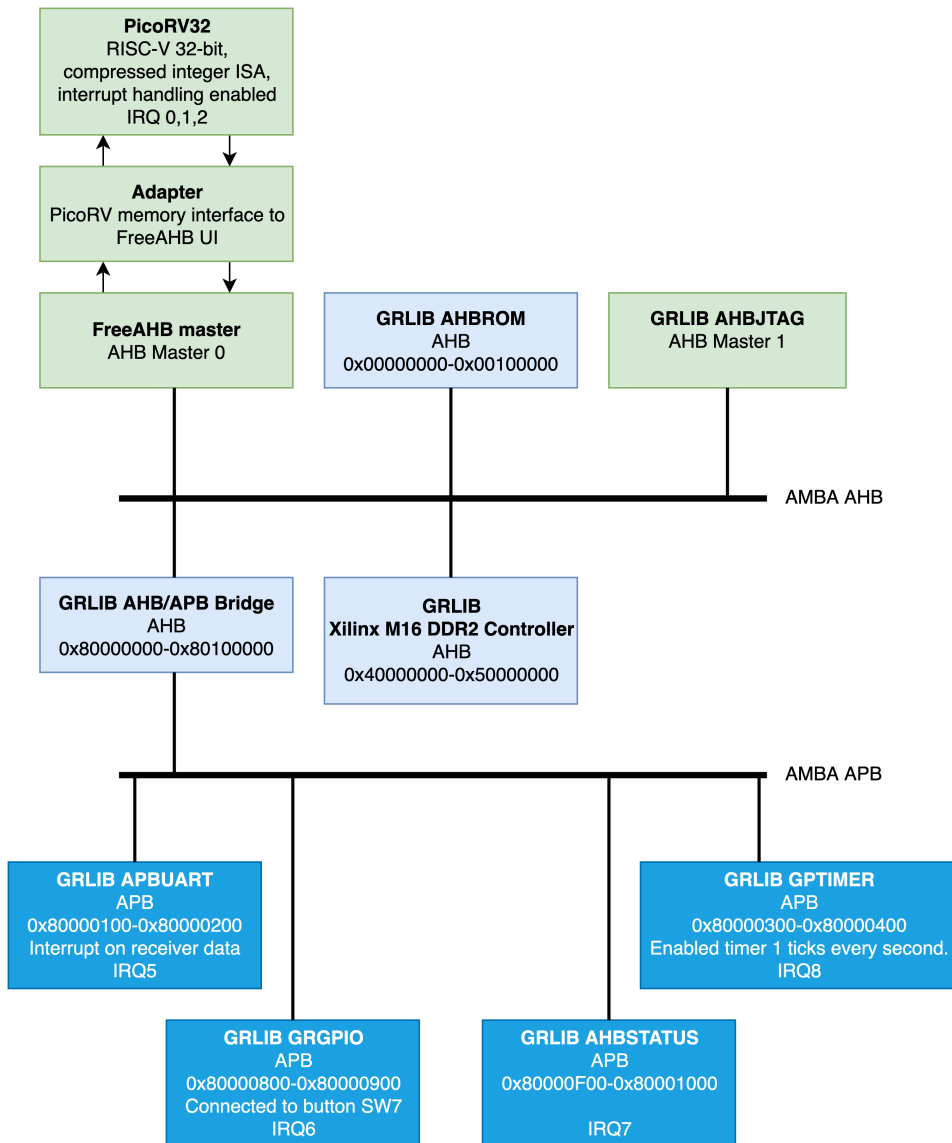


Figure 5.1: The final ZC702 implementation which can run sample RISC-V compiled firmware, that utilizes the GRLIB peripherals. Figure created in draw.io.

5.2 Memory organization

The out-of-the-box GRLIB design includes a Xilinx DDR2 controller which is connected to a region of the ZC702's RAM, and is presented to us as system address 0x40000000-0x50000000. This gives 256MB of space, which is plenty of space. The memory organization is depicted in figure 5.2:

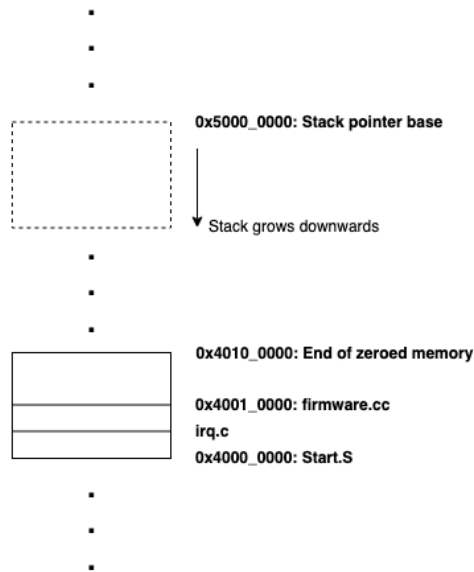


Figure 5.2: Main memory organization for the prototype ZC702 design. The stack has no maximum size set, so in extreme cases, it can grow into the heap and the firmware. The maximum size of the firmware and heap space, based on testing, is limited by the amount of zero-padding we perform with objcopy, as program runs are not always reliable without. We have not had the time to further dig into this issue, but we guess that the compiled firmware assumes memory to be zero-initialized when that is not the case. Figure created with draw.io.

Large parts of the software chain are derived from the original PicoRV repository.

Recall that start.S enables interrupts and zeroes processor registers on startup. It allocates memory for saving register state under interrupts, and also includes an assembler routine for entering and exiting interrupt handling. Irq.c is the actual interrupt handler, which must check the interrupt line bits in order to determine what handlers to run. Firmware.cc initializes GRLIB peripherals and contains the main application.

5.3 Building and running the prototype

The final source code is provided in the master branch of the thesis' repository [64], commit *e425a90*, alongside detailed installation and building instructions.

In short:

1. Install all the required build tools and drivers, and make sure they are available in the user's path.
2. Build the hardware with the GRLIB vivado build flow. This is done by going to `riscvy-grlib/designs/leon3-xilinx-zc702/`, and running `make vivado`.
3. Turn on the ZC702 board.
4. Set the onboard SW10 to '01'.
5. Connect a microUSB cable to the Digilent platform USB.
6. Program the board with `make program-zc702`, in the same build folder as before.
7. Press the SW5 button once to do a HRESET on the board to start it.
8. Go to `riscvy-grlib/lib/riscv/picorv/software`, and make the firmware with `make firmware.srec`.
9. Load to RAM. Start GRMON in the software folder, `grmon -digilent`, then run `load firmware.srec` in the GRMON console.
10. Exit GRMON to free the Digilent platform USB connection.
11. Go to `riscvy-business/uartMonitor`, run `make` to build the monitor, then start it with `./UartMonitor -d JtagSmt1`.
12. When the UartMonitor has done its initial checks, you can now enable the PicoRV by pressing SW7.

The main routine is a very simple proof of concept to showcase the usage of GRG-PIO interrupt, APBUART receiver interrupt and UartMonitor enquiry, and finally the one-second timer. Not following the prompts precisely (pressing the button on an enquiry, for example) will cause malfunction, as we have not spent time on checking the interrupt type on handler return or the like.

Following the prompts should give you the following in UartMonitor:

```
===== MAIN APPLICATION =====
Press SW7 to continue.
INTERRUPT HANDLER: Registered button press!
Button pressed
Requesting input from user...

***UARTMONITOR*** Received ENQUIRY! Enter ONE character, then press enter, please:
k
***UARTMONITOR*** You entered k! Sending it to Pico via UART...

INTERRUPT HANDLER: Received data in APBUART!
We successfully received this character: k
Now, let's count from five!
5..
4..
3..
2..
1..
THE END!

===== MAIN APPLICATION =====
Press SW7 to continue.
█
```

Figure 5.4: The main application in firmware.cc, as seen from UartMonitor at runtime. The user presses the corresponding button on SW7 when prompted. Then, a letter is entered into UartMonitor and sent to PicoRV. Finally, the PicoRV counts for us by utilizing GPTIMER, before jumping back to the start of the main routine.

Chapter 6

Discussion

In this discussion, we seek to answer our two research questions: how difficult is it to port an IP core library from being focused on one ISA, to work with another? How can the process be improved? We will answer both interchangeably throughout this chapter.

6.1 Evaluating the porting process

We did ultimately achieve our goal, which was to create a proof of concept GRLIB design which was driven by a PicoRV processor. The biggest delay we experienced in this project was the amount of time it took to verify and correct the hardware. We identify a lack of hardware design experience, testbench design, and a lack of free tools which provide similar features as paid equivalents as the main reason as to why verification took over twice the amount of time allotted to it. We will expand on these latter two issues later.

The main architectural challenges were mostly correctly identified early on when researching for this project. Due to the high flexibility of the RISC-V ISA, address space width and corresponding data bus widths were not an issue. If RISC-V only came in a 64-bit variant, for example, we would have to somehow adapt the RISC-V 64-bit wide bus to the width that GRLIB assumes (32-bit). We could just leave the upper 32 bits unused, as a simple solution, but it would be yet another issue.

Handling mixed endianness and memory alignment

The mixed endianness environment was by far the biggest thorn in our side. It is in theory not that hard to handle, but when we placed most of the responsibility of handling mixed endianness in the adapter and in the PicoRV firmware, the code could at times get a bit tedious (remembering when to reverse the byte order of data and when not to). There were bugs during implementation, because we forgot about the endianness issue briefly. Manually handling mixed endianness for an embedded solution where a system designer knows the tidbits well is by all means possible, but this would not be an elegant, general solution.

Having a bus protocol which supports mixed endianness, such as AMBA AXI4, would be more elegant, but comes at the cost of increased hardware complexity.

One flaw which cost us some time, was that we assumed that the ZC702 memory system (the Xilinx RAM) would allow unaligned writes, which it did not. Here we solely blame lack of experience and a failure to test our assertions, as this assertion could have been checked early on in the ZC702 testbench.

Port performance

As we have stated before, the solution we have produced is not focused on performance, it is focused on being a proof of concept. We unfortunately have not been able to compile the Dhrystone benchmark due to time (even though the PicoRV repository has a folder for it at the ready), but we can identify three performance bottlenecks nevertheless:

First, the lack of caches between the PicoRV processor and the system bus (AHB) means that we do not make use of the *principle of locality* (if we did use caches, we would have to make sure to not cache memory mapped I/O, such as the GRLIB peripheral registers). Second, the firmware makes PicoRV stall if the UART is not entirely full.

Finally, and a bit out of scope, the PicoRV is meant to be a coprocessor in a design, not a high-performance main processor [41]. There is room to performance improve it somehow, with faster multiplication and faster interrupts, but ultimately, one should select a different core for better performance. We did at the start of the project wish to integrate the Rocket Chip Generator into the design (as it comes with L1 and L2 caches as part of its design), but felt like we already had enough unknowns at hand at the time, and we wished to keep it simple. Now with this base implementation approach complete, we would feel comfortable with expanding the technology stack with CHISEL and diving into RCG in order to somehow derive a design which could be integrated to the ZC702.

Master of none

The final observation we make about the entire process, is that we ultimately felt stretched out a bit thin. We highly believe that the division of labor that is depicted in section 2.1 holds, based on our limited time working in the industry as well. The bottom line is that we recognize that we have barely scratched the surface of the various areas we have touched upon in the implementation – testbench design, hardware specification, baremetal software, Newlib, design specification, on-chip debugging (JTAG) and on-chip development. Much more can be learned and done in each area, but the more ambitious our design gets, the more dependent we are on being part of a team where everyone is specialized within different areas. Despite the advancements of development tools and hardware definition languages, we believe that computer design is still a large undertaking involving various engineering disciplines in order to succeed.

6.1.1 Pros and cons of utilizing existing IP core libraries

Utilizing GRLIB greatly aided us in achieving a final working design. However, we still identify some pros and cons we would like to consider.

First, GRMON comes with many sample designs for many target boards. This includes technology mappings for interfacing the development board's FPGA with the rest of the development board's clocks, on-board memory, and interface ports. The wide support, including for the ZC702, meant that most, if not all of the board implementation work (technology mapping, creating netlists) could be considered out of scope for our work, saving considerable amounts of work. There were, however, times in which we had issues with this scope limitation, and not really getting to know how the technology mapping worked.

For instance, we discovered rather late in the development process that the APBUART in the GRLIB design was not connected to the physical board's UART USB-connection. This is mentioned briefly in the sample design's README, and we also tried to get data from the physical UART with no success. Because of our scope limitation, we therefore turned to the debug link that worked, JTAG, and added additional scaffolding (putting the APBUART in debug mode, and making the firmware halt when the UART transmitter FIFO was not ready) in order to create UartMonitor. It might turn out that fixing the actual wiring between APBUART and the ZC702 UART port is not that much work, but due to our scope limitation, we did not try to dive into the GRLIB techmaps and see if this could be done.

Another limitation with the library, which we did not recognize early on, is that it is (naturally) geared towards Cobham Gaisler's LEON series of processors. We see this with BCC, which compiles programs with the LEON in mind, and especially with GRMON, an essential tool for on-chip development and debugging of a system containing GRLIB components (which, at least in the development version, is only helpful with running LEON programs). Since all the components are open source and documented, and the debug links that GRLIB uses are standardized and well-documented, we have shown it perfectly doable to create on-chip debugging software of our own with UartMonitor. The issue is that this equates to more to develop in order to bring up a first prototype.

6.2 The importance of development tools for productivity

In this project, we have used a wide array of software. Some of it is open software, some have evaluation versions and student editions. The common denominator with most of the evaluation software we have touched upon (ActiveHDL, Vivado, VHDL2Verilog converter) is that the functionality is limited, and performance is throttled, making them less of a viable option for actual development.

Community initiatives such as Icarus Verilog, GTKWave, the powerful GNU toolchain, and Newlib, make it possible for minor actors with sparse resources to attempt actual hardware development. One choice we did in this project – mixing HDLs – turned out to cripple us when time came to simulate the combined Verilog and VHDL design, as we then had to rely on slow student editions of commercial software in order to successfully verify our design. In hindsight, we should have recognized that mixed-language support isn't quite there yet in the open-source tools which we have surveyed, and therefore constrain ourselves to stick to one HDL. We would then have had to stick with VHDL, as most parts of the GRLIB library is specified with it. This would have allowed us to stick to open source alternatives such as GHDL, allowing for a throttle-free simulation process.

While the efforts put into the hardware development tools mentioned are impressive, we kept asking why we did not see more major alternatives out there. In the application software world, we see a huge variety of IDEs, software libraries, frameworks, programming languages, and free software which bridges many gaps found in an application software developer's world. Why is the hardware tool ecosystem (seemingly) not as diverse as the one in the software world?

The problem nature of hardware versus software

We believe that some of the answer resides in the problem nature of the two ecosystems. Application software development is highly relevant for a wide range of professions, and they range from simple Fibonacci number generators to advanced machine learning libraries such as TensorFlow. With little previous background knowledge, hobbyists and scientists can start to develop software, thanks to many powerful software abstractions and programming concepts such as object orientation and encapsulation, many which map to real-world metaphors.

Hardware development, be it for desktop computing, embedded, or server computing, can be considered to be a very specific field, requiring strong knowledge in electrical engineering and/or computer engineering in order to produce efficient results and address design challenges. In the case of desktop and cloud computing, the hardware's purpose is usually to aid the application software designer by providing powerful abstractions which allow the application programmer to make full use of the hardware: parallel programming with MPI or CUDA, mutexes and locks, and maybe the most powerful abstraction of them all: programming languages.

Hardware development is a work intensive and a complex undertaking, and the problem only grows in size with continued technological advancements. In recent years, hardware meets famous challenges such as TDP limitations (computers shrink in size, but cooling proves to be an issue), the memory wall, and dark silicon. Effectively addressing these issues requires a great grasp within many of its fields.

Hardware designs become more complex, and so does its tools

John Chilton of Synopsys coins it well: "The problems of designing today's multibillion transistor chips are huge and systemic. It's increasingly hard for a small group to solve these problems in a meaningful way" [34].

Even the base building blocks of modern hardware development seem nontrivial. The two dominant HDLs, Verilog and VHDL, which both are defined by its IEEE standards, have been revised multiple times over the years. Most, if not all open HDL tools we have surveyed, only support a subset of some version of the language it aims to simulate.

Creative approaches such as trying to translate from one HDL to another is a good idea, but difficult in practice. For one, there is the lack of tools that do so, and the problem is seemingly quite complex as the parsing of – for instance VHDL – is shown to be ambiguous at times [89]. Another issue is that even with the hardware design translated, there still is the issue of translating corresponding testbenches as well.

Increasing momentum leads to a growing ecosystem

However, despite seeing a lack of a variety of tools now, we have reasons to believe that the winds are changing. Initiatives such as CHISEL3, a Scala-based HDL to make hardware development even more software-like with many of its abstractions, and FIRRTL (Flexible Intermediate Representation for RTL), both from the Freechipsproject, seem to pave a new path for how to do hardware design. CHISEL3 designs can be compiled to FIRRTL, a sort of AST, which then can be used to generate Verilog (if needed). Now that full open-source system design is more possible than ever, with growing amounts of IP core collections such as GRLIB and OpenCores, and with a free ISA that is gaining serious momentum (RISC-V), we might be able to see significant increase in efforts to make open-source EDA tools, as hardware becomes more and more accessible to "everyone".

6.3 The importance of a good verification infrastructure

Our development process was dominated by verification. A significant reason why was that we initially were not aware of how important verification is, especially for hardware design, until we started running into problems. While verification will always be a fact of life for hardware development, we believe that improvements can be made in order to empower the designer and make the process more efficient.

6.3.1 Evaluating our verification infrastructure

Mismatch between what is being tested in different simulators

A huge time drain initially when running system simulations in ActiveHDL and the Verilog testbenches, was that we encountered errors in ActiveHDL simulation in the RISC-V side of the design, which were not present in the Verilog testbench `verification/topdesign_tb`, which should have been the case.

Namely, while the Verilog testbench got correct reads from the memory system every time, the ActiveHDL version did not – Sometimes doubling one read, skipping a read to the next address (when expected to), which quickly starts faulty behavior on startup. The mismatch error was due to a fault in the Verilog testbench. Namely, we forgot to modify the original testbench’s memory behavior, in which the `RDATA` port of PicoRV was directly connected to the corresponding testbench memory, bypassing the FreeAHB adapter.

By correcting this error by making the Verilog testbench also fetch its memory via FreeAHB, we then observed the same incorrect behavior as in the ActiveHDL testbench.

During development, there has also been similar issues when developing testbenches. We fail to ensure that the testbenches are tested under as similar conditions as possible: The same configurations for PicoRV, and differing endianness in the adapter depending on whether it is just the pure Verilog testbench or the full system version, are the two major ones.

Poor testbench coverage

Over the course of debugging, we discovered that the Verilog testbenches were lacking for two reasons:

1. **Oversimplified stimulus.** The inputs we fed into the DUV was too simple and tested too few cases, and we discovered in the field that untested conditions did cause errors. In addition, the signals going out of the DUV was rarely checked, as we only checked to see if some base signals were asserted. For read and write tests, for example, we never did actually check to see if the returned read data was correct, or that write data was correctly placed in memory.

Another oversimplification was that of the test environment. We observed initially that FreeAHB only tested a single-master setup in which it was always granted the bus, and the slave always responded with an OKAY to the transfer immediately. In the end design, there has to be multiple masters (The PicoRV master and AHBJTAG, at least, if we wish to perform AHB reads and writes for debugging), and therefore the bus request/grant behavior will need testing.

2. **Untested assertions.** For example, the AHB specification has a lot of tidbits which are difficult to keep in mind at the same time – such as having to insert an explicit wait cycle at the end of an incremental burst of unspecified length in order to signal end of transfer to the arbiter. Another difficulty early on was to fully understand the FreeAHB UI, and when to change what values. In short, we made faulty assertions which we never tested, or hardcoded as assertions. If we did, the simulator could tell us immediately when it detected a violated assertion.

Too manual and repetitive verification methods

The testbenches we have worked to modify to our purposes are still lacking in the way that they are rigged for manual verification only. We get a wavedump of all the accessible signals, and we can enable debug prints to the simulator console which can print out the currently executed instruction in the processor, and memory requests. This is fine when testing the components in isolation to see if it works out of the box, but they are inadequate when cross-dependency bugs start to appear, and it becomes a problem of figuring out where the bug originates, and why. When the design complexity increases, it becomes more and more difficult to be certain where to look for errors.

If we were to redesign, we would try to flesh out a larger set of testbench stimuli (which tests a wider range of behaviors than simple reads and writes), and also a set of assertions which checks various behaviors on a test run.

Recognize the concurrent nature of hardware

Coming from a mostly software-centered background, we failed to recognize the challenges that rise when one deals with an artifact in which everything runs simultaneously on synchronous events, without halting, in contrast to most sequential software, in which finding the error becomes a matter of looking around the failed instruction (usually).

As we mentioned earlier, we made many rather trivial errors. But when these were made simultaneously, it became very difficult to find the error sources without a proper verification infrastructure which could limit the search scope. Even if we were to find one of the bugs, it was not necessarily immediately obvious that we had corrected one, as other bugs still interfered with the outputs from the DUV.

6.3.2 Proposing a better way forward

How can we improve our verification infrastructure? We believe that being able to easily adjust the scope of the DUV to be important. Our problem however, is that many of the Verilog testbenches overlap greatly when it comes to what to instantiate, which stimulus sets to feed, and which signals to monitor. This duplication of work across testbenches caused errors, and if there was a change to a hardware design, then it would likely also require changing the four testbenches to adjust to this new behavior. It is error-prone and highly inefficient.

What we would improve, is to design a modular testbench from the beginning, which can be configured to test the entire system design, or as little as just a single component. In this manner, we can define a complete set of expected stimuli over all port connections, and assertions for all connections. Most importantly, we can write a single test suite, and then run it on any part of the design that we require in the verification process.

This ensures that all verification is done with the same assertions, instantiations, and stimuli, and that there is one single "source of truth". If we change one of the hardware components that are covered by the modular testbench, then we only have to update the modular testbench once (instead of modifying possibly two-three testbenches with the same update).

6.3.3 Key takeaways from literature

Janick Bergeron's "*Writing Testbenches using System Verilog*" [35] includes many great observations regarding verification and testbenches.

First and foremost: "*Verification is not a testbench, nor is it a series of testbenches. Verification is a process*" [35]. In hindsight, we see that failing to recognize this was a costly mistake. Our approach was to quickly bring up some very simple testbenches, sometimes just slightly adapting existing ones, to get some simple wavedumps in which we could manually verify our design. It turned out that we made wrongful assumptions, and we added features to various components later, and the highly fragmented, not very well thought out verification process, made the patching of the verification testbenches tedious. While the verification process got us to a prototype, it was very much due to the fact that we could assume that most of the system's IP was already verified (PicoRV, GRLIB components, ZC702 technology mapping). Testbenches with results that need manual verification, which doesn't test all FSM cases, and which doesn't cover all AHB bus behaviors, will lead to problems, and it simply doesn't scale well.

We also underestimated the scale of verification. In the realms of writing simple modules for application software, a few sessions of GDB was perhaps everything that was needed in order to debug a C program. When moving on to a much more complex undertaking, and where the behavior is always parallel and also has to consider electrical phenomena, verification becomes a much larger effort. "*Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers*" [35].

Our work process could also perhaps need some pair programming. As Bergeron highlights: “*Verifying your own design verifies against your interpretation, not against the specification*” [35]. We experienced this dilemma when developing the adapter, as we made several interpretations about both the memory system, the FreeAHB UI, and the AHB bus protocol. We did test aspects of our interpretations of these, and these naturally passed the testbenches (in the sense that we believed that the wavedumps showed the correct behavior). But when the design was actually wired together and we got to test our design in the real world, it became really clear that we had interpreted many aspects of the surrounding design wrong, as nothing functioned.

The literature distinguishes between the term *verification* and *testing*. Verification is considered to be more related to checking whether your netlist (final design) behaves like intended, based on the specification. Testing, meanwhile, refers to checking whether the implementation of that netlist to a target technology was successful [35]. In our implementation, we have not really distinguished between the two, as the hardware implementation (in the sense of technology mapping, netlist generation, optimization) has been done for us by the Vivado toolchain. We believe that the techniques used in testing can also be used during verification, as long as one designs for it. We will elaborate this when we discuss scan-based testing shortly.

Finally, we summarize five key takeaways from Bergeron [35]:

- **Automate.** Automating the test process removes human error (misreading the wave-dump). Automating improves portability – a new set of eyes doesn't need to know how to perform the manual sections of the test. And most importantly, removing the need for manual input or cross-referencing cuts down the time it takes to determine if the design behaves properly. An example from our process: we checked correct PicoRV execution, by comparing the debug assembly and memory print from the PicoRV during reads, with what we were expecting, given the software's firmware SREC. This could easily be automated with a Python parser and comparator.
- **State assertions.** This overlaps slightly with the previous point. During our work, we found difficulty with understanding all the edge cases and assumptions done in the AHB specification, and how the ZC702 GRLIB system handled memory alignment. Instead of stating these assertions in code (by the use of assert or other testbench functionality), we manually checked it in the wavedumps. One problem is the manual aspect, but by far the biggest issue is that it becomes really difficult for someone else (who does not know our assumptions) to understand why we have designed something the way we have. Writing proper comments is also an alternative approach, but this again requires manual source code inspection.
- **Scan-based testing.** The idea is simple: make all the design's registers available by connecting them in a scan-chain. In normal operation, the registers does not concern themselves with the scan-chain, but when the system enters scan mode, the registers form a single, long shift register [35]. Sound familiar? It should, as JTAG is designed for exactly this type of testing. In our implementation, the testing was performed by utilizing AHBJTAG in order to communicate with device registers. If we were aware of the capabilities of JTAG early on, we could make sure to write tests for the on-board implementation, which could closely correspond to the simulation testbenches for each system module.

- **Reuse is about trust.** As the literature states, for a hardware developer to reuse other modules, they need to trust it. Trust is gained by providing formal verification which proves that the module works. This point is especially important in the case of FreeAHB. Initially, when we had issues with the hardware design, we quickly started pointing fingers at FreeAHB, since we were not quite happy with the provided testbenches (which only tested a subset of the behaviors that it had implemented). It turned out that most of the problems were caused by our lack of understanding of how to interface with the component. Ironically, our verification that is attached to this project does not really do much better, and the best proof we can provide is "try to run it, and see for yourself!". So if we were designing with the intent of others to reuse our work, we would have to focus much more on providing a more rigid verification infrastructure which speaks for itself.
- **Divide and conquer:** design testbenches as you would a system. Earlier we commented on how our main testbench infrastructure should have been, by designing for modularity and flexibility, and defining all the testbench code in one place. The literature supports the modularity, but suggests to think of module testbenches as building blocks instead. Start with module testbenches, then incorporate these in testbenches of ever increasing scope, until you reach your top-level design. While we did create module testbenches in our implementation, and then created testbenches which incorporated more and more of the design, our main problem was that we simply copied and expanded upon the module testbenches, instead of working on how to just instantiate them in a system testbench, resulting in redundant and hard to maintain testbench code.

6.4 Future works

There were many potential candidate projects that arose during this thesis, but which we did not have the time to further investigate. We therefore propose the following possible future works, which are closely related to this project:

1. Dive into the radically different paradigm which the CHISEL flow introduces, creating an HDL from scratch which addresses many of the woes from experience with Verilog and VHDL. What can the community do to attract attention from peers and the industry? How to incentivize the industry on spending manhours to port their existing works into CHISEL?
2. Expand upon our work, use a similar approach as we have done in the implementation to integrate a state of the art core such as the Rocket core in a Rocket Chip Generator design, and then try to performance optimize. Can it match the LEON3 from the base design?
3. What will GRLIB do? Cobham Gaisler surprizingly announced in a press release early 2019 that they joined the RISC-V Foundation [90]. In the same press release, they announced that they would be launching a VHDL-based RISC-V64GC towards the end of the same year. It will be interesting to see how they address the various architectural challenges imposed by the current state of the library.
4. How do we go from our implementation, to running an embedded OS? Initiatives such as Yocto [91] could be worth looking into. With all of the hardware open, putting a OS on top should be very possible.

Conclusion

In this thesis, we have demonstrated how to construct a RISC-V based computing platform, by adapting and reusing IP from the GRLIB IP core library. Utilizing an existing GRLIB design for the Xilinx ZC702 board gave us a tested starting point, in which we could derive our RISC-V computation platform.

The main design challenge was to determine how to adapt to these IP cores, which are originally intended for a 32-bit SPARC-V8 processor – the LEON3. Due to the high degree of flexibility of the RISC-V ISA, which comes in many address widths and instruction set extensions, it was possible for us to find an open core, PicoRV, which could easily be implemented on the ZC702's FPGA thanks to its focus on small size and high max clock frequency. In order for PicoRV and the GRLIB design to communicate, we had to introduce an adapter and the openly available FreeAHB master in order to interface with the system bus.

The other differences on the architecture level was how interrupt controllers were designed, and differing memory endianness. The endianness is especially relevant when communicating with the memory-mapped I/O registers found on the GRLIB peripherals.

To utilize the peripherals, corresponding firmware has been developed which utilizes the memory mapped I/O registers for correct operation. Finally, we use the Digilent Adept SDK in order to create a JTAG application which can communicate with the final RISC-V design through the GRLIB AHBJTAG module.

Studying our implementation process and our final prototype, we identify two main improvements. For one, the verification process for hardware design should be taken into great consideration from the onset in order to save significant amounts of time spent on debugging. Downprioritizing good testbench design and coverage ended up slowing down the implementation process when bugs in the implementation appeared. Second, we find that the current state of open-source hardware development tools to be an issue. We observe that in the same communities as we find RISC-V processor implementations, new technologies such as CHISEL and FIRRTL have been released to address hardware development challenges. So while the current situation is lacking, there is good reason to believe that the situation will improve as RISC-V gains more momentum.

Bibliography

- [1] C. Celio, D. Patterson, and K. Asanovic, “The Berkeley Out-of-Order Machine (BOOM) Design Specification,” tech. rep., University of California, Berkeley, Dec 2016.
- [2] K. Asanović, 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,” Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.
- [3] Cnxsoft, “Arm Compares Arm and RISC-V Benefits in RISC-V Basics Website.” <https://www.cnx-software.com/2018/07/09/arm-vs-riscv-benefits/>, Jul 2018.
- [4] A. Bhadange, “SHAKTI Processor Program.” <http://shakti.org.in/about.html>, Nov 2018.
- [5] “Home - SiFive.” <https://www.sifive.com/>.
- [6] lowRISC Project, “lowRISC project page.” <https://www.lowrisc.org/>, 2018.
- [7] A. Frumusanu, “Arm Announces Neoverse N1 and E1 Platforms and CPUs: Enabling A Huge Jump In Infrastructure Performance.” <https://www.anandtech.com/show/13959/arm-announces-neoverse-n1-platform>, Feb 2019.
- [8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Elsevier, 2014.
- [9] A. Wasserman, “Why we need RISC-V.” <https://hackernoon.com/why-we-need-risc-v-f94e3929891b>, Jan 2018.

- [10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a Scala embedded language,” in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 1212–1221, IEEE, 2012.
- [11] SiFive Inc., “SiFive TileLink Specification, Version 1.7.1.” https://sifive.cdn.prismic.io/sifive%2F57f93ecf-2c42-46f7-9818-bcdd7d39400a_tilelink-spec-1.7.1.pdf, Dec 2018.
- [12] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [13] S. Borkar and A. A. Chien, “The Future of Microprocessors,” *Commun. ACM*, vol. 54, pp. 67–77, May 2011.
- [14] M. B. Taylor, “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,” in *DAC Design Automation Conference 2012*, pp. 1131–1136, IEEE, 2012.
- [15] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, “Challenges in computer architecture evaluation,” *Computer*, vol. 36, no. 8, pp. 30–36, 2003.
- [16] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.10*. RISC-V Foundation, May 2017.
- [17] C. Mellor, “EU plans for domestic exascale supercomputer chips: A RISC-y business.” https://www.theregister.co.uk/2018/07/17/europes_exascale_supercomputer_chips/, Jul 2018.
- [18] M. Valero, “European Processor Initiative RISC-V.” <https://content.riscv.org/wp-content/uploads/2018/05/11.15-11.45-EXASCALE-RISC-V-Mateo.Valero-9-5-2018-1.pdf>, May 2018.
- [19] K. Asanović and D. Patterson, “Instruction sets should be free: The case for RISC-V,” 2014.
- [20] Real World Technology Forums, “ARM announces Ares.” <https://www.realworldtech.com/forum/?threadid=183440&curpostid=183486>.
- [21] Cobham Gaisler AB, “GRLIB IP Core User’s Manual.” <https://www.gaisler.com/products/grlib/grip.pdf>, September 2018.
- [22] V. A. Pedroni, *Circuit design with VHDL*. MIT press, 2004.
- [23] J. Cavanagh, *Digital design and verilog HDL fundamentals*. CRC Press, 2017.

- [24] P. J. Bricaud, “IP reuse creation for system-on-a-chip design,” in *Proceedings of the IEEE 1999 Custom Integrated Circuits Conference (Cat. No. 99CH36327)*, pp. 395–401, IEEE, 1999.
- [25] GitHub, “Atom - The hackable text editor for the 21st century.” <https://atom.io/>, 2019.
- [26] RISC-V, “GitHub: riscv/riscv-tools repository.” <https://github.com/riscv/riscv-tools>, Oct 2018.
- [27] GTKWave, “Welcome to GTKWave.” <http://gtkwave.sourceforge.net/>.
- [28] S. Williams, “Icarus Verilog.” <http://iverilog.icarus.com/>.
- [29] T. Gingold, “GHDL GitHub repository.” <https://github.com/ghdl/ghdl>.
- [30] Open Circuit Design, “Qflow 1.1: An Open-Source Digital Synthesis Flow.” <http://opencircuitdesign.com/qflow/>.
- [31] Xilinx, “Xilinx vivado design tools.” <https://www.xilinx.com/products/design-tools/vivado.html>.
- [32] Intel, “Intel® Quartus® Prime Software.” <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html>.
- [33] Mentor Graphics, “ModelSim®.” <https://www.mentor.com/products/fv/modelsim/>.
- [34] L. Joselyn, “The road to success is long and hard for EDA start ups.” <http://www.newelectronics.co.uk/electronics-technology/the-road-to-success-is-long-and-hard-for-eda-start-ups/58277/>, Dec 2013.
- [35] J. Bergeron, *Writing testbenches using SystemVerilog*. Springer Science & Business Media, 2007.
- [36] C. Felton, “I don’t often convert VHDL to Verilog but when I do ...” <https://www.fpgarelated.com/showarticle/718.php>, 2018.
- [37] SynaptiCAD, “SynaptiCAD EDA Software.” http://www.syncad.com/hdl_translators.htm.
- [38] H. Sutter and J. Larus, “Software and the concurrency revolution,” *Queue*, vol. 3, no. 7, pp. 54–62, 2005.
- [39] Oracle Documentation, “6.1 Considerations for Porting Device Drivers.” https://docs.oracle.com/cd/E37670_01/E52461/html/ch06s01.html#, Jul 2018.

BIBLIOGRAPHY

- [40] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1*. RISC-V Foundation, May 2017.
- [41] “Official PicoRV32 repository.” <https://github.com/cliffordwolf/picorv32>.
- [42] VectorBlox, “GitHub: VectorBlox/orca repository.” <https://github.com/VectorBlox/orca>, Oct 2018.
- [43] RISC-V Foundation, “List of RISC-V Cores.” <https://riscv.org/risc-v-cores/>, 2018.
- [44] Regents of the University of California, “The 3-Clause BSD Licence.” <https://opensource.org/licenses/BSD-3-Clause>, 1988.
- [45] OnchipUIS, “GitHub: onchipuis/mriscv repository.” <https://github.com/onchipuis/mriscv>, Feb 2018.
- [46] Massachusetts Institute of Technology (MIT), “The MIT Licence.” <https://opensource.org/licenses/MIT>.
- [47] “ISC Licence.” <https://opensource.org/licenses/ISC>.
- [48] Cobham Gaisler AB, “GRLIB IP Library User’s Manual.” <https://www.gaisler.com/products/grlib/grlib.pdf>, September 2018.
- [49] Cobham Gaisler AB, “Configuration and Development Guide.” <https://www.gaisler.com/products/grlib/guide.pdf>, September 2018.
- [50] SPARC International, *The SPARC architecture manual: version 8*. Prentice Hall, 1992.
- [51] Cobham Gaisler AB, “Cobham Gaisler Processors.” <https://www.gaisler.com/index.php/products/processors>, 2018.
- [52] ARM, “AMBA 2 Specification, Rev. 2.0.” <http://www.arm.com>, 2000.
- [53] GNU Project, “GNU General Public License, version 2.” <https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html>.
- [54] The OpenRISC Community, “The OpenRISC Instruction Set Architecture.” <https://openrisc.io/architecture>.
- [55] OpenRISC Project, “GitHub: openrisc/or1200 repository.” <https://github.com/openrisc/or1200>.
- [56] OpenRISC Project, “GitHub: openrisc/mor1kx repository.” <https://github.com/openrisc/mor1kx>.
- [57] QEMU, “QEMU - the FAST! processor emulator.” <https://www.qemu.org/>.
- [58] GNU Project, “GCC, the GNU Compiler Collection.” <https://gcc.gnu.org>.

- [59] SiFive, “SiFive - Chip Generator coming soon.” <https://www.sifive.com/chip-designer>, 2018.
- [60] SiFive, “SiFive DesignShare - Introducing a better way to licence IP for SoCs.” <https://www.sifive.com/designshare/>, 2018.
- [61] Bluespec Inc., “Bluespec - innovate with confidence.” <https://bluespec.com/>.
- [62] “lowRISC GitHub Repositories.” <https://github.com/lowRISC>.
- [63] Git. <https://git-scm.com/>.
- [64] K. Monsen, “GitHub: N35N0M/riscvy-business repository.” <https://github.com/N35N0M/riscvy-business>, Jun 2019.
- [65] Krevanth, “GitHub: krevanth/FreeAHB.” <https://github.com/krevanth/FreeAHB>, May 2017.
- [66] “GitHub: The GNU toolchain for RISC-V.” <https://github.com/riscv/riscv-gnu-toolchain>.
- [67] ALDEC, “ActiveHDL: FPGA Design Creation and FPGA simulation.” https://www.aldec.com/en/products/fpga_simulation/active-hdl.
- [68] Cobham Gaisler AB, “Cobham Gaisler website.” <https://www.gaisler.com>, 2018.
- [69] The Cygwin authors [sic], “CYGWIN: Get that Linux feeling - on Windows.” <https://www.cygwin.com/>.
- [70] Diligent, “Diligent Adept 2 software.” https://reference.diligentinc.com/reference/software/adept/start?redirect=1#software_downloads.
- [71] C. Vinschen and J. Johnston, “Newlib homepage.” <https://sourceware.org/newlib/>.
- [72] Canonical, “Ubuntu - The leading operating system for PCs, IoT devices, servers and the cloud.” <https://www.ubuntu.com/>.
- [73] H. Cook, W. Terpstra, and Y. Lee, “Diplomatic Design Patterns: A TileLink Case Study.”
- [74] A. S. Tanenbaum and T. Austin, *Structured computer organization*. Pearson Education, 2013.
- [75] Unknown author, “SREC: UNIX manual page copy.” <http://www.amelek.gda.pl/avr/uisp/srecord.htm>.

BIBLIOGRAPHY

- [76] “Xilinx ZC702 User Guide.” https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf, Dec 2013.
- [77] Cobham Gaisler AB, “GRMON3 User’s Manual (Evaluation version).” <https://www.gaisler.com/doc/grmon-eval/grmon3.pdf>, February 2019.
- [78] Cobham Gaisler AB, “BCC User’s Manual.” <https://www.gaisler.com/j25/doc/bcc2.pdf>, 2019.
- [79] Xilinx, “Instantiating a Verilog Module in a VHDL Design Unit.” https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ism_p_instantiating_verilog_module_mixedlang.htm, March 2019.
- [80] J. Bentley, “Programming pearls,” *Communications of the ACM*, vol. 28, no. 9, pp. 896–901, 1985.
- [81] “IEEE Std 1149.1-2001: IEEE Standard Test Access Port and Boundary-Scan Architecture,” 2001.
- [82] G. Pietrzak, “Wikimedia: JTAG Scan Chain illustration.” https://commons.wikimedia.org/wiki/File:Jtag_chain.svg.
- [83] “Zynq-7000 SoC Technical Reference Manual.” https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
- [84] “7 series FPGAs Configuration User Guide.” https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.
- [85] D. M. Ritchie and B. W. Kernighan, *The C programming language: second edition*. Bell Laboratories, 1988.
- [86] Freedom Embedded, “Hello world for bare metal ARM using QEMU.” <https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/>.
- [87] C. Vinschen and J. Johnston, “Newlib: syscalls documentation.” <https://www.sourceware.org/newlib/libc.html#Syscalls>.
- [88] “GitHub: RISC-V Assembly Programmer’s Manual.” <https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md>.
- [89] L. Lorencl, R. Schoneckerl, and Z. Kiivkal, “A Note on the Parsing of Complete VHDL,”
- [90] Cobham Gaisler AB, “Cobham joins RISC-V Foundation.” https://www.cobham.com/media/2103951/cobham_gaisler_risc-v_2019-02-26.pdf, 2019.

- [91] Yocto Project, “Yocto Project – Its not an embedded Linux distribution – it creates a custom one for you.” <https://www.yoctoproject.org/>.
- [92] GNU Project, “GNU: What is copyleft?.” <https://www.gnu.org/licenses/copyleft.en.html>, 2018.
- [93] Unlicense.org, “Unlicense Yourself: Set Your Code Free.” <https://unlicense.org/>.
- [94] R. Krohalev, “Software Licenses Explained.” <https://shakuro.com/blog/software-licenses-explained/>, Dec 2017.
- [95] Mozilla Foundation, “Mozilla Public License Version 2.0.” <https://www.mozilla.org/en-US/MPL/2.0/>.
- [96] GitHub Inc, “GNU General Public License v2.0.” <https://choosealicense.com/licenses/gpl-2.0/>, 2018.
- [97] RISC-V Foundation, “RISC-V Foundation FAQ.” <https://riscv.org/faq/>, Nov 2018.
- [98] The Apache Software Foundation, “Apache licence version 2.0.” <https://www.apache.org/licenses/LICENSE-2.0>, 2004.
- [99] M. Wolf, *Computers as components: principles of embedded computing system design*. Elsevier/Morgan Kaufmann, 2012.
- [100] E. Altman, D. Kaeli, and Y. Sheffer, “Welcome to the opportunities of binary translation,” *Computer*, vol. 33, no. 3, p. 40–45, 2000.
- [101] A. D’Antras, C. Gorgovan, J. Garside, and M. Luján, “Low overhead dynamic binary translation on ARM,” in *ACM SIGPLAN Notices*, vol. 52, pp. 333–346, ACM, 2017.
- [102] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, “The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges,” *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003.

Appendices

A review of software licences

In our pursuit of a true open-source computer architecture, it is in our opinion paramount that contributors understand the implications placed by various licence types.

A.1 Copyright versus copyleft

A term that often arises in the open-source community is that of copyleft. In contrast to the traditional copyright, where one places restrictions or protections on a piece of intellectual work in order to control access and distribution to that work, copyleft looks to be its stark contrast.

Copyleft, as stated by the GNU Project, “is a general method for making a program (or other work) free (in the sense of freedom, not “zero price”), and requiring all modified and extended versions of the program to be free as well” [92]. This contrasts approaches such as releasing into the public domain or using the Unlicence [93], where such work can possibly be incorporated into proprietary solutions and subsequently copyrighted. In pure technical terms, a copyleft licence is still some sort of copyright, but it is one that shares the licence with others on the notion that derived works will also be shared under the same licence.

A.2 Permissive, copyleft and hybrid approaches

Within the realm of open-source targeted licences, Krohalev [94] presents three main categories: permissive licences, copyleft licences, and hybrids.

Permissive licences are those which allow developers to use the licenced work in their own projects – commercial or free – without major restrictions. This category is arguably the most realistic in terms of how the world works today. While the ideal is that all software should be open and free, a commercialized world requires developers to be paid somehow or to give them some incentive in order to do work. By being able to utilize free software

while giving credit, it allows developers to make a living out of their commercial software by being able to incorporate open-source works.

Copyleft licences are on the idealistic side of things, representing what is arguably the utopia of the free software movement. If a project uses copyleft-licensed libraries and works, then the entire work usually has to utilise the same licence, effectively making the entire work free. This is part of the genius behind the GNU GPL variants of licences [53] – they effectively “spread”, as many projects derive from GNU GPL licenced works, and that effectively makes them copyleft too.

Hybrid approaches try to achieve a “best of both worlds” solution, by incorporating the strengths of both. The Mozilla Public Licence Version 2.0 (MPL 2.0) [95] is a commonly used variant. Here, all modified parts of the original work has to remain under the same MPL2.0 Licence, while all new original work can be licenced differently (free or closed).

A.3 The licences involved in this project

GRLIB, the hardware library we will be looking at, is licenced under the GNU General Public Licence (GPL) version 2. This is a strong and popular copyleft variant. Modified or derived versions of these works will also have to be released under the same licence, meaning that all source code must also be made readily available. Changes in the code must also be stated [96].

The RISC-V ISA is “free and open for use by anyone in all types of implementations without restrictions” [97]. Although a little vague, the foundation states that developers are free to use the ISA for “commercial exploitation” and open-source implementations, while fully commercial implementations are required to become members of the RISC-V Foundation. It falls under the copyleft category.

The Rocket Chip generator, one of the more popular RISC-V ISA modular platform generators, has a whole cocktail of licences. The contributions from SiFive, one of the largest RISC-V chip designer businesses which features some of the RISC-V founders, falls under the Apache Licence Version 2 [98]. The Apache licence falls under the permissive category – you are allowed to use it commercially, distribute it, modify it, patent it, use it privately, and maybe important for many – you do not have to disclose the source code of derivative work. The most important aspect is that the licence must be listed, and that changes made from the original work made must be stated somewhere. The chisel-jtag component falls under a custom licence, which seems like a custom variant of the MIT Licence [46] – it is pretty permissive, but you are required to provide the licence and give notice of copyright of the related software, and a standard “no liability or warranty” clause. The same goes for all other components of the Rocket Chip generator, which is provided for the most part by UC Berkeley.

Appendix B

Application software porting issues

B.1 Software layer organization

For most embedded systems and SoCs, the following figure provides a realistic overview of a system’s software layering:

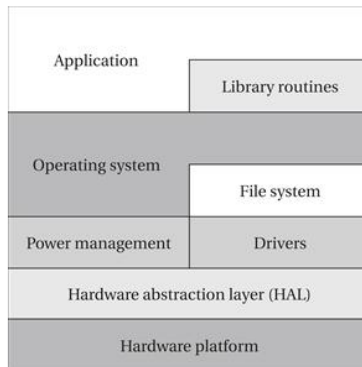


Figure B.1: "Software layers of an embedded system". Figure from [99].

Initially, we are presented with a hardware platform. In order to productively use a hardware platform, there should be some hardware abstraction layer (HAL) which encapsulates some of the grittier details of the hardware. On top of this, we may add power management features and device drivers, which usually take advantage of the HAL if present. With basic drivers in place for system critical functions, we can introduce a file system, and if needed, an operating system (OS). The OS encapsulates the underlying layers for the everyday user, and handles crucial functionality such as virtual memory, interrupt handling and process separation. On top of this again, we can create library routines for common user tasks, which are callable by user applications by the use of an Application Binary Interface (ABI). [99]

B.2 Software porting techniques

When porting application software between platforms, there are several concerns one should address. Altman et. al. [100] present three main approaches to porting application layer software:

1. Use a special processor mode to execute legacy code.
2. Recompile the entire program with new build tools.
3. Runtime interpretation or translation of legacy code.

We will discuss each of these approaches in turn.

Provide a special processor mode for legacy software

This essentially translates to added hardware. Intel does this with x86, and translates it into micro-operations that are used on modern Intel processors.

The benefits of this approach is that you spend some extra hardware, but then add little overhead in software, potentially making it very fast and energy efficient.

For many applications this is not a preferable option, as it adds hardware to the design, taking up costly area. In addition, if one is to support multiple foreign ISAs, then the hardware spent might be too dominant in the overall design. However, if one new ISA wants to take over the market leader – x86 – spending hardware to add compability and speed up legacy code execution might be preferable.

Porting of software at the application layer

The main idea is pretty simple: take the source code from which the applications were compiled and recompile them with a toolchain targeting the new ISA. Since the GNU project has added support for RISC-V in their industry-standard toolchain, it is in theory possible to recompile all application software as long as the original source code is available.

One weakness of this approach is that it does not take into account the problem of legacy software. It is common for businesses to depend on old, possibly unmaintained software where the original source code is lost. Therefore, this quickly becomes an impossible alternative for many.

Runtime interpretations

This is usually the most considered approach when it comes to binary translation since it adds little overhead in hardware, in exchange for some possible software overhead. Within this category, Altman et. al [100] provide three approaches:

1. Emulation.
2. Static translation.
3. Dynamic translation.

Emulation is an approach that is often seen for legacy applications, where the code might not have been actively maintained for some time, or the original source code for building the application is missing. It is implemented either through the use of a software system emulator such as QEMU [57], or through bare-metal virtual machines that run the legacy code and its system on a virtual machine guest. The effort of making such emulators is said to be relatively simple [100].

Static translation shares many of the strengths of compilation: analyzing and translating the entire application before running it enables optimizations that cannot be done at runtime. It is done purely in software, with no extra hardware support needed. However, it requires active user involvement per translation (read: invoking the static translator, then run the application).

The **dynamic translation** approach contrasts the static translations in strengths and weaknesses: there is minimal user involvement for performing actual translations, but runtime overhead is added. While a dynamic translator cannot perform the same optimizations as static translation, it can perform some runtime only optimizations such as optimizing often taken control paths and caching translations.

Both translation approaches open their own can of worms, however. For example, there might be a mismatch between available processor registers on the new ISA contradicting the old. On an implementation of dynamic translation from AArch32 to the AArch64 ISA, called MAMBO-X64, one issue was that AArch32 provided 48 floating point registers, while the new AArch64 architecture only supported 32 floating point registers. This requires that some register state has to be held in memory, and that the dynamic interpreter constantly has to keep track of live values and perform register remapping between basic blocks in order to ensure correct execution [101].

Other translation issues include memory-mapped I/O – it is difficult to determine at runtime whether or not a memory address refers to some memory-mapped I/O or not. Another issue is determining overlapping memory requests when performing dynamic translation optimization [102]. Atomic instructions might also differ between ISAs, leading to a possible source of bugs for concurrent programs.

Further care must be given to interactions between the OS and application levels. Specific syscalls might not have a translation in similar operating systems [100]. In the case of MAMBO-X64, mentioned earlier, a system emulator was needed to map syscalls and memory mapping between 32-bit and 64-bit Linux. MAMBO also has to address the issue of differing calling conventions and memory alignments.

Original version of the FreeAHB master

We have done minor modifications to the source material in cases where lines did not fit the page. No other modifications have been done. This code is used as a base for discussion and further improvement when describing our approach in interfacing this AHB master with the RISC-V core and with GRLIB.

```
/*
 * Title           : AHB 2.0 Master
 *
 * License         : MIT license
 *
 * Target          : ASIC/FPGA
 *
 * Author          : Revanth Kamaraj
 *
 * This RTL describes a generic AHB master with support for single and
 * burst transfers. Split/retry pipeline rollback is also supported.
 * The entire design is driven by a single clock i.e., AHB clock. A global
 * asynchronous active low reset is provided.
 *
 * ----->          NOTE: THE DESIGN IS IN AN EXPERIMENTAL STATE.    <-----
 *
 * MIT License
 *
 * Copyright (C) 2017 Revanth Kamaraj
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 */
```

Appendix C. Original version of the FreeAHB master

```
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in all
* copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*****/

module ahb_master #(parameter DATA_WDT = 32, parameter BEAT_WDT = 32) (

/*****
* AHB interface.
*****/
input          i_hclk,
input          i_hreset_n,
output reg [31:0] o_haddr,
output reg [2:0] o_hburst,
output reg [1:0] o_htrans,
output reg [DATA_WDT-1:0] o_hwdata,
output reg          o_hwrite,
output reg [2:0] o_hsize,
input          [DATA_WDT-1:0] i_hrdata,
input          i_hready,
input          [1:0] i_hresp,
input          i_hgrant,
output reg          o_hbusreq,

/*****
* User interface.
*****/

output reg          o_next, // UI must change only if this is 1.
input          [DATA_WDT-1:0] i_data, // Data to write.
// Can change during burst if o_next = 1.
input          i_dav, // Data to write valid.
// Can change during burst if o_next = 1.
input          [31:0] i_addr, // Base address of burst.
input          [2:0] i_size, // Size of transfer. Like hsize.
input          i_wr, // Write to AHB bus.
input          i_rd, // Read from AHB bus.
input          [BEAT_WDT-1:0] i_min_len, // Minimum guaranteed length of burst.
input          i_cont, // Current transfer continues previous one.
output reg [DATA_WDT-1:0] o_data, // Data got from AHB is presented here.
output reg [31:0] o_addr, // Corresponding address is presented here.
output reg          o_dav // Used as o_data valid indicator.
);

/*
* NOTE: You can change UI signals at any time if the unit is IDLING.
*
* NOTE: o_next is a combinational signal.
```

```

*
* NOTE: When reset is released, the signals on the UI should represent an
* IDLING state. From there onwards, you can change the UI at any time but once
* you change it, further changes can be made only if o_next = 1. You could
* perhaps connect o_next to read_enable of a FIFO.
*
* To go to IDLE, you must follow this...
*     To set the unit to IDLE mode, make
*         i_cont = 0, i_rd = 0 and i_wr = 0 (or)
*         i_wr = 1 and i_dav = 0.
* on o_next = 1. As mentioned above, you change UI signals without having
* o_next = 1 but once changed you must change them again only when o_next = 1.
*
* NOTE: The first UI request of a burst must have valid data provided in case
* of a write. You CANNOT start a burst with the first UI request having
* wr = 1 and dav = 0.
*
* NOTE: Most UI inputs should be held constant during a burst.
*/

/*****
* Localparams
*****/

/*
* SINGLE, WRAPs are currently UNUSED.
* Single transfers are treated as bursts of length 1 which is acceptable.
*/
localparam [1:0] IDLE    = 0;
localparam [1:0] BUSY   = 1;
localparam [1:0] NONSEQ = 2;
localparam [1:0] SEQ    = 3;
localparam [1:0] OKAY   = 0;
localparam [1:0] ERROR  = 1;
localparam [1:0] SPLIT  = 2;
localparam [1:0] RETRY  = 3;
localparam [2:0] SINGLE = 0; /* Unused. Done as a burst of 1. */
localparam [2:0] INCR   = 1;
localparam [2:0] WRAP4  = 2;
localparam [2:0] INCR4  = 3;
localparam [2:0] WRAP8  = 4;
localparam [2:0] INCR8  = 5;
localparam [2:0] WRAP16 = 6;
localparam [2:0] INCR16 = 7;
localparam [2:0] BYTE   = 0;
localparam [2:0] HWORD  = 1;
localparam [2:0] WORD   = 2; /* 32-bit */
localparam [2:0] DWORD  = 3; /* 64-bit */
localparam [2:0] BIT128 = 4;
localparam [2:0] BIT256 = 5;
localparam [2:0] BIT512 = 6;
localparam [2:0] BIT1024 = 7;

/* Abbreviations. */
localparam D = DATA_WDT-1;
localparam B = BEAT_WDT-1;

```

Appendix C. Original version of the FreeAHB master

```
/*
*****
* Flip-flops.
*****/

reg [4:0] burst_ctr;          // Small counter to keep track of current burst count.
reg [B:0] beat_ctr;         // Counter to keep track of word/beat count.

/* Pipeline flip-flops. */
reg [1:0] gnt;
reg [2:0] hburst;           // Only for stage 1.
reg [D:0] hwdata [1:0];
reg [31:0] haddr [1:0];
reg [1:0] htrans [1:0];
reg [1:0] hwrite;
reg [2:0] hsize [1:0];
reg [B:0] beat;           // Only for stage 2.

/* Tracks if we are in a pending state. */
reg      pend_split;

/*
*****
* Signal aliases.
*****/

/* Detects the first cycle of split and retry. */
wire spl_ret_cyc_1 = gnt[1] && !i_hready && (i_hresp == RETRY || i_hresp == SPLIT);

/* Inputs are valid only if there is a read or if there is a write with valid data. */
wire rd_wr      = i_rd || (i_wr && i_dav);

/* Detects that 1k boundary condition will be crossed on next address */
wire blk_spec   = (haddr[0] + (1 << i_size)) >> 10 != haddr[0][31:10];

/*
*****
* Misc. logic.
*****/

/* Output drivers. */
always @* {o_haddr, o_hburst, o_htrans, o_hwdata, o_hwrite, o_hsize} =
{haddr[0], hburst, htrans[0], hwdata[1], hwrite[0], hsize[0]};

/* UI must change only if this is 1. */
always @* o_next = (i_hready && i_hgrant && !pend_split);

/*
*****
* Grant tracker.
*****/
/* Passes grant throughout the pipeline. */
always @ (posedge i_hclk or negedge i_hreset_n)
begin
if ( !i_hreset_n )
gnt <= 2'd0;
else if ( spl_ret_cyc_1 )
gnt <= 2'd0; /* A split retry cycle 1 will invalidate the pipeline. */
else if ( i_hready )
gnt <= {gnt[0], i_hgrant};
end
```

```

/*****
* Bus request
*****/
always @ (posedge i_hclk or negedge i_hreset_n)
begin
/* Request bus when doing reads/writes else do not request bus */
if ( !i_hreset_n )
o_hbusreq <= 1'd0;
else
o_hbusreq <= i_rd | i_wr;
end

/*****
* Address phase. Stage I.
*****/
always @ (posedge i_hclk or negedge i_hreset_n)
begin
if ( !i_hreset_n )
begin
/* Signal IDLE on reset. */
htrans[0] <= IDLE;
pend_split <= 1'd0;
end
else if ( spl_ret_cyc_1 ) /* Split retry cycle I */
begin
htrans[0] <= IDLE;
pend_split <= 1'd1;
end
else if ( i_hready && i_hgrant )
begin
pend_split <= 1'd0; /* Any pending split will be cleared */

if ( pend_split )
begin
/* If there's a pending split, perform a pipeline rollback */

{hwdata[0], hwrite[0], hsize[0]} <= {hwdata[1], hwrite[1], hsize[1]};

haddr[0] <= haddr[1];
hburst <= compute_hburst (beat, haddr[1], hsize[1]);
htrans[0] <= NONSEQ;
burst_ctr <= compute_burst_ctr(beat, haddr[1], hsize[1]);
beat_ctr <= beat;
end
else
begin
{hwdata[0], hwrite[0], hsize[0]} <= {i_data, i_wr, i_size};

if ( !i_cont && !rd_wr ) /* Signal IDLE. */
begin
htrans[0] <= IDLE;
end
else if ( (!i_cont && rd_wr) || !gnt[0] || (burst_ctr == 1 && o_hburst != INCR)
|| htrans[0] == IDLE || blk_spec )
begin
/* We need to recompute the burst type here */

```


Appendix C. Original version of the FreeAHB master

```
haddr[0] <= !i_cont ? i_addr : haddr[0] + (rd_wr << i_size);
hburst   <= compute_hburst (!i_cont ? i_min_len : beat_ctr,
!i_cont ? i_addr : haddr[0] + (rd_wr << i_size) , i_size);
htrans[0] <= rd_wr ? NONSEQ : IDLE;

burst_ctr <= compute_burst_ctr(!i_cont ? i_min_len : beat_ctr - rd_wr,
!i_cont ? i_addr : haddr[0] + (rd_wr << i_size) , i_size);

beat_ctr  <= !i_cont ? i_min_len : ((hburst == INCR) ? beat_ctr : beat_ctr - rd_wr);
end
else
begin
/* We are in a normal burst. No need to change HBURST. */

haddr[0]  <= haddr[0] + ((htrans[0] != BUSY) << i_size);
htrans[0] <= rd_wr ? SEQ : BUSY;
burst_ctr <= o_hburst == INCR ? burst_ctr : (burst_ctr - rd_wr);
beat_ctr  <= o_hburst == INCR ? beat_ctr : (beat_ctr - rd_wr);
end
end
end
end

/*****
* HWDATA phase. Stage II.
*****/
always @ (posedge i_hclk)
begin
if ( i_hready && gnt[0] )
{hwdata[1], haddr[1], hwrite[1], hsize[1], htrans[1], beat} <=
{hwdata[0], haddr[0], hwrite[0], hsize[0], htrans[0], beat_ctr};
end

/*****
* HRDATA phase. Stage III.
*****/
always @ (posedge i_hclk or negedge i_hreset_n)
begin
if ( !i_hreset_n )
o_dav <= 1'd0;
else if ( gnt[1] && i_hready && (htrans[1] == SEQ || htrans[1] == NONSEQ) )
begin
o_dav <= !hwrite[1];
o_data <= i_hrdata;
o_addr <= haddr[1];
end
else
o_dav <= 1'd0;
end

/*****
* Functions.
*****/
function [2:0] compute_hburst (input [B:0] val, input [31:0] addr, input [2:0] sz);
begin
compute_hburst = (val >= 16 && no_cross(addr, 15, sz)) ? INCR16 :
```

```

(val >= 8  && no_cross(addr, 7, sz)) ? INCR8 :
(val >= 4  && no_cross(addr, 3, sz)) ? INCR4 : INCR;

$display($time, "val = %d, addr = %d, sz = %d, compute_hburst = %d",
val, addr, sz, compute_hburst);
end
endfunction

function [4:0] compute_burst_ctr(input [B:0] val, input [31:0] addr, input [2:0] sz);
begin
compute_burst_ctr = (val >= 16 && no_cross(addr, 15, sz)) ? 5'd16 :
(val >= 8  && no_cross(addr, 7, sz)) ? 5'd8  :
(val >= 4  && no_cross(addr, 3, sz)) ? 5'd4  : 5'd0;

$display($time, "val = %d, addr = %d, sz = %d,
compute_burst_ctr = %d", val, addr, sz, compute_burst_ctr);

end
endfunction

function no_cross(input [31:0] addr, input [31:0] val, input [2:0] sz);
if ( addr + (val << (1 << sz )) >> 10 != addr[31:10] )
no_cross = 1'd0; // Crossed!
else
no_cross = 1'd1; // Not crossed
endfunction

////////// END OF RTL. START OF DEBUG CODE //////////
`ifndef SIM // Define SIM only during verification.

wire [31:0] beat_ctr_nxt = !i_cont ? (i_min_len - rd_wr) :
((hburst == INCR) ? beat_ctr : beat_ctr - rd_wr);

initial
begin
$display($time, "DEBUG MODE ENABLED! PLEASE MONITOR CAPITAL SIGNALS IN VCD...");
end

`ifndef STRING
`define STRING reg [256*8-1:0]
`endif

`STRING HBURST;
`STRING HTRANS;
`STRING HSIZE;
`STRING HRESP;

always @*
begin
case(o_hburst)
INCR:   HBURST = "INCR";
INCR4:  HBURST = "INCR4";
INCR8:  HBURST = "INCR8";
INCR16: HBURST = "INCR16";
default:HBURST = "<----?????---->";
endcase

```

```
case(o_htrans)
SINGLE: HTRANS = "IDLE";
BUSY:   HTRANS = "BUSY";
SEQ:    HTRANS = "SEQ";
NONSEQ: HTRANS = "NONSEQ";
default:HTRANS = "<----????---->";
endcase

case(i_hresp)
OKAY:   HRESP = "OKAY";
ERROR:  HRESP = "ERROR";
SPLIT:  HRESP = "SPLIT";
RETRY:  HRESP = "RETRY";
default:HRESP = "<---????---->";
endcase

case(o_hsize)
BYTE    : HSIZE = "8BIT";
HWORD   : HSIZE = "16BIT";
WORD    : HSIZE = "32BIT"; // 32-bit
DWORD   : HSIZE = "64BIT"; // 64-bit
BIT128  : HSIZE = "128BIT";
BIT256  : HSIZE = "256BIT";
BIT512  : HSIZE = "512BIT";
BIT1024 : HSIZE = "1024BIT";
default : HSIZE = "<----????---->";
endcase
end

`endif

endmodule
```

Appendix D

The PicoRV to FreeAHB adapter

This adapter is an original work for this project. It translates the signals between the PicoRV memory interface and the FreeAHB UI, and can be configured for a big-endian or little-endian AHB bus.

```
// *****  
// picorv32_to_freeahb_adapter  
// *****  
  
module picorv32_freeahb_adapter #(parameter BIG_ENDIAN_AHB = 1) (  
    input          clk,  
    input          resetn,  
    input          enable,  
  
    // FreeAHB interface  
    output reg     [31:0] freeahb_wdata,  
    output reg     freeahb_valid,  
    output reg     [31:0] freeahb_addr,  
    output reg     [2:0] freeahb_size,  
    output reg     freeahb_write,  
    output reg     freeahb_read,  
    output reg     [31:0] freeahb_min_len,  
    output reg     freeahb_cont, // Continues prev transfer  
    output reg     [3:0] freeahb_prot,  
    output reg     freeahb_lock,  
  
    input          freeahb_next,  
    input          [31:0] freeahb_rdata,  
    input          [31:0] freeahb_result_addr, // Not used.  
    input          freeahb_ready, // rdata contains valid data.  
  
    // Native PicoRV32 memory interface  
    input          mem_valid,  
    input          mem_instr,  
    output reg     mem_ready,  
    input          [31:0] mem_addr,
```

```

input      [31:0]    mem_wdata,
input      [3:0]    mem_wstrb,
output     [31:0]    mem_rdata

```

```

);
// Arguably, this complexity could/should lie in the AHB master.
// But we'd rather write up a new AHB master at this point, so we
// place the complexity here for the time being.
//
// Note that rdata from the bus is only valid when HREADY is raised with
// HRESP OKAY.
if (BIG_ENDIAN_AHB == 1) begin
    assign mem_rdata[31:24] = freeahb_rdata[7:0]; // Byte A+3
    assign mem_rdata[23:16] = freeahb_rdata[15:8]; // Byte A+2
    assign mem_rdata[15:8]  = freeahb_rdata[23:16]; // Byte A+1
    assign mem_rdata[7:0]   = freeahb_rdata[31:24]; // Byte A

end
else begin
    assign mem_rdata      = freeahb_rdata;

end

reg [3:0] write_ctr;// Used when composing wstrb into individual writes.
reg      pending_write;
reg      pending_write_finish;
reg      pending_read;
reg      enabled = 0;

always @(posedge clk or negedge resetn) begin
    // IDLE memory system conditions
    if (!resetn && enabled) begin
        enabled <= 0;
        freeahb_valid      <= 1'b0;
        freeahb_write      <= 1'b0;
        freeahb_read       <= 1'b0;
        freeahb_cont       <= 1'b0;
        freeahb_lock       <= 1'b0;
        freeahb_min_len    <= 0;
        freeahb_size       <= 0;
        freeahb_prot       <= 0;
        mem_ready          <= 1'b0;
        write_ctr          <= 0;
        pending_write      <= 1'b0;
        pending_write_finish <= 1'b0;
        pending_read       <= 1'b0;
    end

    else if (!enabled && enable) begin
        enabled <= 1;
    end

    else if (!resetn || !mem_valid || mem_ready || !enabled) begin
        freeahb_valid      <= 1'b0;
        freeahb_write      <= 1'b0;
        freeahb_read       <= 1'b0;
    end
end

```

```

        freeahb_cont      <= 1'b0;
        freeahb_lock     <= 1'b0;
        freeahb_min_len  <= 0;
        freeahb_size     <= 0;
        freeahb_prot     <= 0;
        mem_ready        <= 1'b0;
        write_ctr        <= 0;
        pending_write    <= 1'b0;
        pending_write_finish <= 1'b0;
        pending_read     <= 1'b0;
end

// *****
// READS
//*****
// READ transfer start
else if (mem_valid && mem_wstrb == 4'b0000 && !pending_read) begin
    freeahb_addr      <= mem_addr;
    freeahb_size      <= 3'b010;
    freeahb_read      <= 1'b1;
    freeahb_min_len   <= 0;
    freeahb_prot      <= mem_instr ? 4'b0000 : 4'b0001;
    pending_read      <= 1'b1;
end

// READ transfer complete
else if (mem_valid && mem_wstrb == 4'b0000
        && pending_read && freeahb_ready) begin
    //$display("READ BASE ADDR %h, RDATA %h", mem_addr, mem_rdata);
    mem_ready        <= 1'b1;
    freeahb_valid    <= 1'b0;
    freeahb_read     <= 1'b0;
    freeahb_write    <= 1'b0;
    freeahb_cont     <= 1'b0;
    pending_read     <= 1'b0;
end

// *****
// WRITES
// *****

// WRITE sequences
// The mem IF outputs WSTRB (write strobes), but this is a AXI4 signal,
// and does not suit AHB. We therefore have to translate the strobes to
// individual AHB transfers.
else if (mem_valid && mem_wstrb != 4'b0000 && write_ctr < 4
        && !pending_write && !pending_write_finish) begin
    // If we are to do a write, and freeahb indicates it is ready.
    // ADDRESS PHASE
    if (mem_wstrb[write_ctr]) begin
        freeahb_valid    <= 1'b0;
        freeahb_addr     <= mem_addr + write_ctr;
        freeahb_size     <= 3'b000; // byte
        freeahb_write    <= 1'b1;
        freeahb_cont     <= 1'b0;
        freeahb_prot     <= mem_instr ? 4'b0000 : 4'b0001;
    end
end

```

```

        pending_write      <= 1'b1;
    end

    // If we do not write this round,
    // we must make sure to make the FreeAHB idle.
    else if (!mem_wstrb[write_ctr]) begin
        freeahb_write      <= 1'b0;
        write_ctr          <= write_ctr + 1;
    end

end

// Write sequence finished
else if (mem_valid && mem_wstrb != 4'b0000 && !pending_write
        && !pending_write_finish && write_ctr == 4) begin
    mem_ready      <= 1'b1;
    freeahb_write <= 1'b0;
    freeahb_valid <= 1'b0;
    write_ctr     <= 0;
    freeahb_wdata <= 0; // Not necessary, but makes it easier to debug.
end

// For 32-bit reads, we simply clear the read bit on the UI.
// For writes, we must now drive data if we are in the address phase.
else if (mem_valid && freeahb_next &&
        (pending_read || pending_write || pending_write_finish) ) begin
    freeahb_read <= 1'b0;

    if (pending_write) begin
        case (write_ctr)
            0: begin
                if (BIG_ENDIAN_AHB == 1)
                    freeahb_wdata[31:24] <= mem_wdata[7:0];
                else
                    freeahb_wdata[7:0]   <= mem_wdata[7:0];
            end
            1: begin
                if (BIG_ENDIAN_AHB == 1)
                    freeahb_wdata[23:16] <= mem_wdata[15:8];
                else
                    freeahb_wdata[7:0]   <= mem_wdata[15:8];
            end
            2: begin
                if (BIG_ENDIAN_AHB == 1)
                    freeahb_wdata[15:8] <= mem_wdata[23:16];
                else
                    freeahb_wdata[7:0]  <= mem_wdata [23:16];
            end
            3: begin
                if (BIG_ENDIAN_AHB == 1)
                    freeahb_wdata[7:0] <= mem_wdata[31:24];
                else
                    freeahb_wdata[7:0] <= mem_wdata[31:24];
            end
        endcase
    end
end

```

```
                end
            endcase
            freeahb_valid <= 1'b1;
            pending_write <= 1'b0;
            pending_write_finish <= 1'b1;
        end
    else if (pending_write_finish) begin
        // $display("WRITE BASE ADDR %h, MEM WDATA %h", mem_addr, mem_wdata);
        pending_write_finish <= 1'b0;
        freeahb_write <= 1'b0;
        freeahb_valid <= 1'b0;
        write_ctr <= write_ctr + 1;
    end
end
end
endmodule
```


Appendix E

UartMonitor

UartMonitor is an original work for this project, which builds upon a short demo provided in the Digilent Adept 2 SDK. It utilizes the aforementioned SDK to communicate with the ZC702's JTAG scan chain. The monitor is used to communicate with PicoRV in the GRLIB FPGA design, by requesting AHB reads and writes with AHBJTAG via the Xilinx TAP. We do so, because the APBUART is not mapped to the onboard UART, so we need to poll it in debug mode instead via the JTAG debug link.

```
// UART VIA JTAG routine
// Written as part of the RISC-VY-BUSINESS master thesis code repository.
// Written by Kris Monsen, utilizing the Digilent Adept SDK, extending
// the demo provided by the Digilent SDK.

// As a courtesy, we reproduce the original licence for the demo.
// Most of it is left as-is, but most things after printing out IDCODES is new.
// (except ErrorExit, ShowUsage and the actual API calls, obviously).

/*****
/*
/*  DtgtDemo.cpp  --  DJTG DEMO Main Program
/*
/*****
/*  Author:    Aaron Odell
/*  Copyright: 2010 Digilent, Inc.
/*****
/*  Module Description:
/*      DJTG Demo demonstrates how to read in IDCODEs from the
/*      JTAG scan chain. Codes for some Digilent FPGA boards are
/*      given below.
/*
/*      Nexys2: 0x41c22093
/*              0xf5046093
/*
/*      Basys2: 0x11c10093
/*              0xf5045093
/*
/*****
```

Appendix E. UartMonitor

```
/* Revision History: */
/* */
/* 03/16/2010(AaronO): created */
/* */
/*****/

#define _CRT_SECURE_NO_WARNINGS

/* ----- */
/* Include File Definitions */
/* ----- */

#if defined(WIN32)

/* Include Windows specific headers here.
*/
#include <windows.h>

#else

/* Include Unix specific headers here.
*/

#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "dpcdecl.h"
#include "djtg.h"
#include "dmgr.h"
#include <signal.h>

/* ----- */
/* Local Type and Constant Definitions */
/* ----- */

/* ----- */
/* Global Variables */
/* ----- */
HIF hif;

/* ----- */
/* Local Variables */
/* ----- */

/* ----- */
/* Forward Declarations */
/* ----- */

void ShowUsage(char* szProgName);
void ErrorExit();
void ResetThenIdle();
```

```

void UpdateInstructions(BYTE* instructions, int amount_of_bits);
void UpdateData(BYTE* payload, int amount_of_bits);
void ReadData(BYTE* readbuffer, int amount_of_bits);
void HandleInterrupt(int code);
void ReadWordFromAhb(BYTE* address, BYTE* readBuffer);
void BinaryPrintOfWord(BYTE* wordBuffer);
void WriteWordToAhb(BYTE* address, BYTE* writeBuffer);
void PrintUartRegisters();

/* ----- */
/*                               Procedure Definitions                               */
/* ----- */
/**
 * main
 **
 * Parameters:
 **     cszArg      - number of command line arguments
 **     rgszArg    - array of command line argument strings
 **
 * Return Value:
 **
 * Errors:
 **
 * Description:
 **     Run the program.
 */

// Source for ctrl+c interrupt handling:
// https://stackoverflow.com/questions/4217037/catch-ctrl-c-in-c
// And also The C Programming Language: Second Edition
static volatile int exitProgram = 0;

// Important addresses to word-sized registers on the APBUART...
static BYTE uartData      [] = {0x00,0x01,0x00,0x80};
static BYTE uartStatus   [] = {0x04,0x01,0x00,0x80};
static BYTE uartControl   [] = {0x08,0x01,0x00,0x80};
static BYTE uartFifoDebug [] = {0x10,0x01,0x00,0x80};

int main(int cszArg, char* rgszArg[]) {
    int i;
    int cCodes = 0;
    BYTE rgbSetup[] = {0xaa, 0x22, 0x00};
    BYTE rgbTdo[4];

    INT32 idcode;
    INT32 rgIdcodes[16];

    /* Command checking */
    if( cszArg < 3 ) {
        ShowUsage(rgszArg[0]);
        ErrorExit();
    }
    if( strcmp(rgszArg[1], "-d") != 0 ) {
        ShowUsage(rgszArg[0]);
        ErrorExit();
    }

    // DMGR API Call: DmgrOpen

```

```
if(!DmgrOpen(&hif, rgpszArg[2])) {
    printf("Error: Could not open device. Check device name\n");
    ErrorExit();
}

// DJTG API CALL: DtgtgEnable
if(!DtgtgEnable(hif)) {
    printf("Error: DtgtgEnable failed\n");
    ErrorExit();
}

/* Put JTAG scan chain in SHIFT-DR state.
 * RgbSetup contains TMS/TDI bit-pairs.
 */
// DJTG API Call: DtgtgPutTmsTdiBits
if(!DtgtgPutTmsTdiBits(hif, rgbSetup, NULL, 9, fFalse)) {
    printf("DtgtgPutTmsTdiBits failed\n");
    ErrorExit();
}

/* Get IDCODES from device until we receive a value of 0x00000000 */
do {

    // DJTG API Call: DtgtgGetTdoBits
    if(!DtgtgGetTdoBits(hif, fFalse, fFalse, rgbTdo, 32, fFalse)) {
        printf("Error: DtgtgGetTdoBits failed\n");
        ErrorExit();
    }
    // Convert array of bytes into 32-bit value
    idcode = (rgbTdo[3] << 24) | (rgbTdo[2] << 16)
             | (rgbTdo[1] << 8) | (rgbTdo[0]);

    // Place the IDCODEs into an array for LIFO storage
    rgIdcodes[cCodes] = idcode;

    cCodes++;

} while( idcode != 0 );

/* Show the IDCODEs in the order that they are connected on the device */
printf("Ordered JTAG scan chain:\n");
for(i=cCodes-2; i >= 0; i--) {
    printf("0x%08x\n", rgIdcodes[i]);
}

// END OF DEMO, START OF OUR APPLICATION.

// First, reset the TAPs, and then put ARM to BYPASS (1111),
// and Xilinx to USER1.
ResetThenIdle();
printf("\n");
DWORD current_frequency;
DWORD oneMHZ = 1000000;
DtgtgGetSpeed(hif, &current_frequency);
printf("JTAG clock speed after reset: ");
printf("%Id", current_frequency);
printf("\n");
```

```

DjtgSetSpeed(hif, oneMHZ, &current_frequency);
DjtgGetSpeed(hif, &current_frequency);
printf("Successfully set JTAG clock speed to: ");
printf("%Id", current_frequency);
printf("\n\n");

// Handle interrupts. A interrupt will cause the program to shutdown cleanly.
signal(SIGINT, HandleInterrupt);

// Perform an initial read on the UARTs data, status, control and debug reg.
printf("===== INITIAL UART STATUS =====\n");
PrintUartRegisters();

// Set APBUART to transmit and receive enable, receiver interrupt,
// and in FIFO debug mode, and with FIFOs available bit.
BYTE setUartToDebug[] = {0x07, 0x08, 0x00, 0x80};
WriteWordToAhb(uartControl, setUartToDebug);

printf("===== UART SET TO FIFO DEBUG MODE =====\n");
PrintUartRegisters();

printf("===== CHARACTER WRITE AND READ TEST ===== \n");
BYTE writeH[] = {0x48, 0x00, 0x00,0x00};
WriteWordToAhb(uartData, writeH);

BYTE readBuffer[4];
// Check status register...
ReadWordFromAhb(uartStatus, readBuffer);
BinaryPrintOfWord(readBuffer);

// And read...
ReadWordFromAhb(uartFifoDebug, readBuffer);
printf("We got %c from the UART transmitter!\n\n", readBuffer[0]);

printf("===== \n");
printf("Now monitoring APBUART... Press Ctrl+C to stop.\n");
printf("===== \n");

char userInput;
BYTE response[4] = {0x00,0x00,0x00,0x00};
while (exitProgram == 0) {
    // Read the status register...
    ReadWordFromAhb(uartStatus, readBuffer);

    // If the status register indicates data exists (TE != 1)...
    if ( !(readBuffer[0] & 1<<2) ) {
        // Read that data...
        ReadWordFromAhb(uartFifoDebug, readBuffer);

        // If it is an enquiry (ENQ, 0x05),
        // we should prompt for a char from the user.
        if (readBuffer[0] == '\x05') {
            printf("\n\n***UARTMONITOR*** Received ENQUIRY!");
            printf("Enter ONE character, then press enter, please: \n");
            userInput = getchar();
            getchar(); // We dont want the newline character.
            printf("***UARTMONITOR*** You entered %c!", userInput);

```

```
        printf("Sending it to Pico via UART...\n\n");
        response[0] = userInput;
        WriteWordToAhb(uartFifoDebug, response);
    }
    else {
        // Print that character.
        printf("%c", readBuffer[0]);
    }
}

// Disable Dtgt and close device handle
if( hif != hifInvalid ) {
    // DGTG API Call: DtgtDisable
    DtgtDisable(hif);

    // DMGR API Call: DmgrClose
    DmgrClose(hif);
}

return 0;
}

void PrintUartRegisters(){
    BYTE readBuffer[4];

    printf("UART Data Register \n");
    ReadWordFromAhb(uartData, readBuffer);
    BinaryPrintOfWord(readBuffer);

    printf("UART Status Register \n");
    ReadWordFromAhb(uartStatus, readBuffer);
    BinaryPrintOfWord(readBuffer);

    printf("UART Control Register \n");
    ReadWordFromAhb(uartControl, readBuffer);
    BinaryPrintOfWord(readBuffer);

    printf("UART FIFO Debug Register \n");
    ReadWordFromAhb(uartFifoDebug, readBuffer);
    BinaryPrintOfWord(readBuffer);
}

void BinaryPrintOfWord(BYTE* wordBuffer){
    for (int i=3; i>=0; i--) {
        for (int j=0; j<8; j++){
            printf("%d", !!(wordBuffer[i] << j) & 0x80));
        }
        printf(" ");
    }
    printf("\n\n");
}

// Expects a 32-bit address, and a 32-bit buffer with valid bytes to write.
void WriteWordToAhb(BYTE* address, BYTE* writeBuffer){
```

```

// Arm DAP to BYPASS, Xilinx TAP to USER1 ('AHBJTAG Command Register')
BYTE xilinx_user1_and_arm_bypass[] = {0xc2, 0x03};
UpdateInstructions(xilinx_user1_and_arm_bypass, 10);

// AHBJTAG to perform desired write by setting AHBJTAG Command Register
// 34: W/R, 33-32: Size, 31-0: Addr.
BYTE grlib_uart_dr_write[] = { address[0],
                               address[1],
                               address[2],
                               address[3],
                               0x06
                               };
UpdateData(grlib_uart_dr_write, 36);

// Setting Arm DAP to BYPASS, Xilinx TAP to USER2 ('AHBJTAG Data Register')
BYTE xilinx_user2_and_arm_bypass[] = {0xc3, 0x03};
UpdateInstructions(xilinx_user2_and_arm_bypass, 10);

// 32: SEQ (this is the next write), 31-0: Big-Endian AHB Read/Write data.
// This write is not SEQ. Remember an extra bit for BYPASS reg.
BYTE writeSequence[] = { writeBuffer[0],
                          writeBuffer[1],
                          writeBuffer[2],
                          writeBuffer[3],
                          0x01};
UpdateData(writeSequence, 34);

// Read the 33-bit AHBJTAG data register until SEQ bit is 1
BYTE ahbdata_tdo[5] = {0x00, 0x00, 0x00, 0x00, 0x00};
do {
    ReadData(ahbdata_tdo, 40);
} while(!(ahbdata_tdo[4] & 1) & exitProgram == 0);
}

// Expects a 32-bit address, and a 32-bit buffer to return the read word to.
void ReadWordFromAhb(BYTE* address, BYTE* readBuffer){
// Arm DAP to BYPASS, Xilinx TAP to USER1 ('AHBJTAG Command Register')
BYTE xilinx_user1_and_arm_bypass[] = {0xc2, 0x03};
UpdateInstructions(xilinx_user1_and_arm_bypass, 10);

// AHBJTAG to perform desired read by setting AHBJTAG Command Register
// 34: W/R, 33-32: Size, 31-0: Addr.
BYTE grlib_uart_dr_read[] = { address[0],
                               address[1],
                               address[2],
                               address[3],
                               0x02};
UpdateData(grlib_uart_dr_read, 36);

// Setting Arm DAP to BYPASS, Xilinx TAP to USER2 ('AHBJTAG Data Register')
BYTE xilinx_user2_and_arm_bypass[] = {0xc3, 0x03};
UpdateInstructions(xilinx_user2_and_arm_bypass, 10);

// Read the 33-bit AHBJTAG data register until SEQ is 1 (Read finished!)
BYTE ahbdata_tdo[5] = {0x00, 0x00, 0x00, 0x00, 0x00};
int counter = 0;
do {

```

```
        ReadData(ahbdata_tdo, 40);
        counter++;
        if (counter>10) {
            printf("MONITOR: Possible bus thrashing detected!\n");
            counter = 0;
        }
    } while (!(ahbdata_tdo[4] & 1) && exitProgram == 0);

    memcpy(readBuffer, ahbdata_tdo, 4);
}

void HandleInterrupt(int code) {
    exitProgram = 1;
}

// Custom helpers.
void ResetThenIdle() {
    BYTE sequence[] = {0xaa, 0x02};
    if(!DjtgPutTmsTdiBits(hif, sequence, NULL, 6, fFalse)) {
        printf("DjtgPutTmsTdiBits failed\n");
        ErrorExit();
    }
}

void UpdateInstructions(BYTE* instructions, int bitCount) {
    BYTE IdleToShiftIr[] = {0x03}; // First hex is the only relevant.
    BYTE ShiftIrToIdle[] = {0x03}; // First three bits are the only relevant

    // Go from IDLE to SHIFT-IR state
    if(!DjtgPutTmsBits(hif, fFalse, IdleToShiftIr, NULL, 4, fFalse)) {
        printf("Pushing TMS bits failed!");
        ErrorExit();
    }

    // Shift in instructions to IRs.
    if(!DjtgPutTdiBits(hif, 0, instructions, NULL, bitCount-1, fFalse)){
        printf("Pushing TDI bits failed!");
        ErrorExit();
    }

    // Go from SHIFT-IR to IDLE state.
    if(!DjtgPutTmsBits(hif, fFalse, ShiftIrToIdle, NULL, 3, fFalse)) {
        printf("Pushing TMS bits failed!");
        ErrorExit();
    }
}

void UpdateData(BYTE* payload, int amount_of_bits) {
    BYTE idle_to_shift_dr[] = {0x01}; // First three bits are relevant
    BYTE shift_dr_to_idle[] = {0x03}; // First three bits are the only relevant

    // Go from IDLE to SHIFT-DR state
    if(!DjtgPutTmsBits(hif, fFalse, idle_to_shift_dr, NULL, 3, fFalse)) {
        printf("Pushing TMS bits failed!");
        ErrorExit();
    }
}
```

```

// Shift in payload to DRs.
// cbits != count of bits. The first bit seems to be clocked in anyway.
if(!DjtgPutTdiBits(hif, fFalse, payload, NULL, amount_of_bits-1, fFalse)){
    printf("Pushing TDI bits failed!");
    ErrorExit();
}

// Go from SHIFT-DR to IDLE state.
if(!DjtgPutTmsBits(hif, fFalse, shift_dr_to_idle, NULL, 3, fFalse)) {
    printf("Pushing TMS bits failed!");
    ErrorExit();
}
}

void ReadData(BYTE* readbuffer, int amount_of_bits) {
    BYTE idle_to_shift_dr[] = {0x01}; // First three bits are relevant
    BYTE shift_dr_to_idle[] = {0x03}; // First three bits are the only relevant

    // Go from IDLE to SHIFT-DR state
    if(!DjtgPutTmsBits(hif, fFalse, idle_to_shift_dr, NULL, 3, fFalse)) {
        printf("Pushing TMS bits failed!");
        ErrorExit();
    }

    // Read amount_of_bits of DR out via TDO
    // For some DRs, such as the AHB/JTAG Data Register (USER2), it is vital
    // that we are careful about what we shift in. Shifting in a SEQ bit
    // will cause that unit to read from the next sequential address.
    if(!DjtgGetTdoBits(hif, fFalse, fFalse, readbuffer, amount_of_bits, fFalse)) {
        printf("Error: DjtgGetTdoBits failed\n");
        ErrorExit();
    }

    // Go from SHIFT-DR to IDLE state.
    if(!DjtgPutTmsBits(hif, fFalse, shift_dr_to_idle, NULL, 3, fFalse)) {
        printf("Pushing TMS bits failed!");
        ErrorExit();
    }
}

/* ----- */
/** ShowUsage
**
** Parameters:
**     szProgName    - name of program as called (from rgpszArg[0])
**
** Return Value:
**     none
**
** Errors:
**     none
**
** Description:
**     Demonstrates proper paramater usage to the user

```

```
*/

void
ShowUsage(char* szProgName) {
    printf("Error: Invalid paramaters\n");
    printf("Usage: %s -d <device> \n\n", szProgName);
}

/* ----- */
/** ErrorExit
**
** Parameters:
**     none
**
** Return Value:
**     none
**
** Errors:
**     none
**
** Description:
**     Disables DJTG, closes the device, and exits the program
*/
void ErrorExit() {
    if( hif != hifInvalid ) {

        // DJGT API Call: DjtgDisable
        DjtgDisable(hif);

        // DMGR API Call: DmgrClose
        DmgrClose(hif);
    }

    exit(1);
}

/* ----- */

/*****/
```

Appendix F

RISC-V Verilog toplevel to GRLIB system VHDL Wrapper

This is the wrapper used to interface the RISC-V side Verilog with the GRLIB VHDL.

```
-- This wrapper is a combination of grlib.pdf's chapter 8.3
-- (adapted for a master, not a slave),
-- and Xilinx' suggestion for instantiating a Verilog module to a VHDL library.
```

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.config_types.all;
use grlib.config.all;
use grlib.stdlib.all;
use grlib.devices.all;

entity picorv_grlib_ahb_master is
    generic (master_index : integer := 2);
    port (
        rst      : in    std_ulogic;
        clk      : in    std_ulogic;
        enable   : in    std_ulogic;
        ahbmi    : in    ahb_mst_in_type;
        ahbmo    : out   ahb_mst_out_type);
end;

architecture pico of picorv_grlib_ahb_master is
    component pico_ahb_master
        port (
            HCLK      : in    std_ulogic;
            HRESETn   : in    std_ulogic;
            HIRQ      : in    std_logic_vector(31 downto 0);
            HGRANTx   : in    std_ulogic;
```

```

HREADY      :   in  std_ulogic;
HRESP       :   in  std_logic_vector(1 downto 0);
HRDATA      :   in  std_logic_vector(31 downto 0);

BUSREQx     :   out std_ulogic;
HLOCKx      :   out std_ulogic;
HTRANS      :   out std_logic_vector(1 downto 0);
HADDR       :   out std_logic_vector(31 downto 0);
HWRITE      :   out std_ulogic;
HSIZE       :   out std_logic_vector(2 downto 0);
HBURST      :   out std_logic_vector(2 downto 0);
HPROT       :   out std_logic_vector(3 downto 0);
HWDATA      :   out std_logic_vector(31 downto 0);
ENABLE      :   in  std_ulogic);

end component;

-- GRLIB Plug&play information --
constant HCONFIG: ahb_config_type := (
  -- Only the first config word must be defined for masters
  -- (all other words are slave memory mappings)
  -- Note that the free version of GRMON does not allow us
  -- to use custom vendors or partids.
  0      => ahb_device_reg (VENDOR_CONTRIB, CONTRIB_CORE1, 0, 0, 0),
  others => X"00000000");

begin
  ahbmo.hconfig      <= HCONFIG;
  ahbmo.hindex       <= master_index;
  ahbmo.hirq         <= (others => '0'); -- We do not drive any interrupts.

  wrapped_picorv: pico_ahb_master
    port map(
      HCLK           => clk,
      HRESETn        => rst,
      HIRQ           => ahbmi.hirq,
      HGRANTx        => ahbmi.hgrant(master_index),
      HREADY         => ahbmi.hready,
      HRESP          => ahbmi.hresp,
      HRDATA         => ahbmi.hrdata,

      BUSREQx        => ahbmo.hbusreq,
      HLOCKx         => ahbmo.hlock,
      HTRANS         => ahbmo.htrans,
      HADDR          => ahbmo.haddr,
      HWRITE         => ahbmo.hwrite,
      HSIZE          => ahbmo.hsize,
      HBURST         => ahbmo.hburst,
      HPROT          => ahbmo.hprot,
      HWDATA         => ahbmo.hwdata,
      ENABLE         => enable);
end;

```

Appendix G

Firmware.cc

The main firmware file used for running a implemented PicoRV on a GRLIB system. A derived work based on various works from the PicoRV repository [41].

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <algorithm>

/*
 * This is a modified version of firmware.cc,
 * found in the original PicoRV repository:
 * https://github.com/cliffordwolf/picorv32
 */

// Defined in start.S
extern "C" uint32_t WaitForInterrupt();

class ExampleBaseClass
{
public:
    ExampleBaseClass() {
        std::cout << "ExampleBaseClass()" << std::endl;
    }

    virtual ~ExampleBaseClass() {
        std::cout << "~ExampleBaseClass()" << std::endl;
    }

    virtual void print_something_virt() {
        std::cout << "ExampleBaseClass::print_something_virt()" << std::endl;
    }

    void print_something_novirt() {
        std::cout << "ExampleBaseClass::print_something_novirt()" << std::endl;
    }
};
```

```

/*
 * INITIALIZE GRGPIO
 *
 * grgpio_ipol: Specify that the SW7 button is active high.
 * grgpio_iedge: SW7 is to interrupt on the rising edge.
 * grgpio_imask: Allow SW7 to raise interrupts.
 *
 */
volatile int grgpio_ipol = 0x01000000;
volatile int grgpio_iedge = 0x01000000;
volatile int grgpio_imask = 0x01000000;
*(volatile int*)0x80000810 = grgpio_ipol;
*(volatile int*)0x80000814 = grgpio_iedge;
*(volatile int*)0x8000080C = grgpio_imask;

printf("-- Enabled GRGPIO button interrupt!\n");

/* SETUP, BUT DO NOT ENABLE, GPTIMER
 *
 * The prescaler is 8 bits, while timer1 is 32 bits.
 * We set the reload value of prescaler and timer1 so that timer1 will
 * underflow and cause an interrupt every second, based on a 83MHZ clock.
 */
volatile int gptimer_scaler_reload_value = 0xFF000000;
volatile int gptimer_timer1_reload_value = 0x7AF20400;

*(volatile int*)0x80000304 = gptimer_scaler_reload_value;
*(volatile int*)0x80000314 = gptimer_timer1_reload_value;
printf("-- Set GPTIMER prescaler and timer1 so that timer1 can \n");
printf("  interrupt every second when enabled.\n\n");

printf("Testing the stack... \n\n");
ExampleBaseClass *obj = new ExampleBaseClass;
obj->print_something_virt();
obj->print_something_novirt();
delete obj;

obj = new ExampleSubClass;
obj->print_something_virt();
obj->print_something_novirt();
delete obj;

std::vector<unsigned int> some_ints;
some_ints.push_back(0x48c9b3e4);
some_ints.push_back(0x79109b6a);
some_ints.push_back(0x16155039);
some_ints.push_back(0xa3635c9a);
some_ints.push_back(0x8d2f4702);
some_ints.push_back(0x38d232ae);
some_ints.push_back(0x93924a17);
some_ints.push_back(0x62b895cc);
some_ints.push_back(0x6130d459);
some_ints.push_back(0x837c8b44);
some_ints.push_back(0x3d59b4fe);
some_ints.push_back(0x444914d8);
some_ints.push_back(0x3a3dc660);

```

```

some_ints.push_back(0xe5a121ef);
some_ints.push_back(0xff00866d);
some_ints.push_back(0xb843b879);

std::sort(some_ints.begin(), some_ints.end());

for (auto n : some_ints)
    std::cout << std::hex << n << std::endl;

std::cout << "Setup and quicktest done. Now we run the main application.\n\n"
    << std::endl;

// Values for GPTIMER's TIMER1. Refer to GRLIB IP core manual for details.
volatile int timer1_enable_with_interrupt = 0x0D000000;
volatile int* gptimer_timer1_control = (volatile int*)0x80000318;

// Value to store the character we receive.
char response;

// Main routine
while (true){
    printf("===== MAIN APPLICATION =====\n");
    printf("Press SW7 to continue.\n");
    WaitForInterrupt();
    printf("Button pressed\n");
    printf("Requesting input from user...\n");
    response = getchar();
    printf("We successfully received this character: %c \n", response);
    printf("Now, let's count from five!\n");

    for (volatile int i=5; i>0; i--){
        printf("%i...\n", i);
        *gptimer_timer1_control = timer1_enable_with_interrupt;
        WaitForInterrupt();
    }
    printf("THE END!\n\n");
}

return 0;
}

```

