Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 208b

Systems Group, Department of Computer Science, ETH Zurich

System Modeling Co-Design

by
Sven Knobloch

Supervised by
Reto Achermann, Lukas Humbel, Prof. Dr. Timothy Roscoe

Sept. 16, 2018

inf | Informatik
Computer Science

Contents

1 Abstract **iii**

2 Introduction **1**

3 Background **3**
3.1 Existing Work . . . 3
3.1.1 VHDL[5] & Verilog[6] . . . 3
3.1.2 SystemC[7] . . . 3
3.1.3 gem5[8] . . . 3
3.2 Sockeye[1, 2] . . . 3
3.2.1 Sockeye's Use Case . . . 4
3.2.2 Sockeye Concepts . . . 4
3.2.3 Goals using Sockeye . . . 4
3.3 LISA and LISA+[11] . . . 5
3.3.1 LISA+ Concepts . . . 5
3.3.2 ARM Fast Models[9] . . . 6

4 ARM Server Base System Architecture **7**
4.0.1 Architecture . . . 7
4.0.2 Importance to Barrelfish . . . 7
4.1 Translation of the SBSA . . . 8
4.2 Conclusion . . . 9

5 Design & Implementation **11**
5.1 Restrictions & Assumptions . . . 11
5.2 Mappings . . . 12
5.3 Existing Sockeye Framework . . . 15
5.4 Existing LISA+ Framework . . . 15
5.5 Additional Utilities . . . 15
5.6 Extension of Sockeye Syntax . . . 15
5.7 Auxiliary File . . . 16

6 Results & Discussion **18**
6.1 Evaluation . . . 18
6.2 Dynamic Components . . . 18
6.3 Efficiency . . . 18
6.4 LISA+ . . . 18
6.5 Closing Remarks . . . 19

7 Future Work **21**
7.1 Integration with Mackerel[3] . . . 21
7.2 Extension of Library Components . . . 21
7.3 Power Domain . . . 21

8 Appendix **22**
8.1 SBSA Sockeye Files . . . 22
8.1.1 SBSA . . . 22
8.1.2 Processing Element . . . 23
8.1.3 Generic Interrupt Controller . . . 23

8.1.4 Generic Watchdog 24
8.1.5 Generic UART 24
8.1.6 Generic Timer 25
8.2 Minimal System Image 26
8.2.1 Sockeye Description 26
8.2.2 Auxiliary File 31

1 Abstract

In recent years, hardware systems and systems on a chip (SoCs) have become increasingly diverse. In order to build software for such a large variety of systems, better ways have to be found to target and test software for these platforms. One such solution is hardware simulation, which allows for quick, efficient and inexpensive experimentation of software with a large variety of systems. In addition, simulation allows software developers to better integrate their software with specific hardware platforms to increase performance, efficiency and overall interoperability.

The Barrelfish group as operating system developers are extremely interested in increasing performance and compatibility with a large variety of systems. They have developed a domain specific specification language, Sockeye[1, 2], to describe system models as a hardware decoding net to query for addresses and interrupt information at runtime. This project evaluates the viability of Sockeye as a language to generate system models from and successfully demonstrates how to generate valid hardware simulators from these models using ARM's Fast Models[9] framework and their modeling language LISA+[11]. ARM has also published their Server Base System Architecture[4], which defines a common standard for ARMv8 platforms. This project also analyzes and evaluates the Server Base System Architecture and determines its compatibility with the Barrelfish ecosystem, specifically how well it integrates with Sockeye for system modeling. The Server Base System Architecture is found to be a good starting point for a standard but is too loose in its restrictions to provide concrete information to developers.

2 Introduction

In recent years, hardware systems and systems on a chip (SoCs) have become increasingly diverse. Since these systems are not homogeneous, developers have a hard time writing and maintaining code for so many different platforms that each have varying instruction sets and system capabilities. Additionally, targeting all these platforms without simulation is not only difficult and time consuming, but also expensive. With all these different systems out there each one would have to be purchased and tested individually. However, simulating these systems has negligible costs and testing can easily be automated, making continuous integration simple and efficient as well. Specifically for operating system and embedded developers this is beneficial because platform coverage and hardware interoperability is crucial to the viability of software. They are also able to interact more closely with a large variety of different types of hardware to provide better performance and optimizations.

Additionally, modeling and general hardware software co-design can lead to more efficient, performant and compatible systems. In the past, these systems scaled up with the introduction of newer and faster hardware, but the pace of these advancements is slowing due to natural physical limitations. These limitations forced developers to find new, better ways to improve the performance of their systems and thus the concept of hardware software co-design was formed.

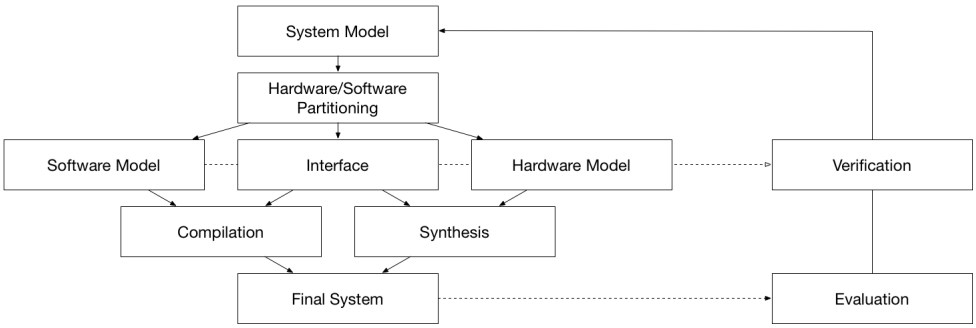

Figure 1: The hardware software co-design process.

Hardware software co-design allows software developers to work more closely with targeted hardware, so it becomes much simpler to optimize performance overhead. These optimizations include reducing memory usage, power consumption, processing time for tasks, resource contention, etc. These performance gains are significant and are the next step in improving future generations of hardware and software.

Barrelfish is a multikernel based research operating system being developed at ETH Zürich that focuses primarily on systems with a numerous cores in order to address the rising demand for scalability as well as to run on a large variety of systems and architectures. The Barrelfish group utilizes hardware software co-design tools, like the ARM Fixed Virtual Platforms[10], to aid in the development of its operating system. These tools are mainly utilized to fit the software to existing and upcoming hardware since the Barrelfish group targets a large variety of platforms. Internally, the group has developed additional tools, including Sockeye[1, 2] and Mackerel[3], to facilitate the development process

and help formally reason about these systems.

This project concerns itself with Barrelfish and expanding its usage of hardware software co-design tools, namely hardware modeling and simulation. This is done by using an existing Barrelfish specific design language, Sockeye, and determining its feasibility as a hardware software co-design tool by planning and constructing a compiler that can build a working system simulator from a given model. Additionally, the ARM Server Base System Architecture[4] is also investigated and analyzed for its overall viability and more specifically how well it can be represented in Sockeye. The first section will cover background information, including existing hardware software co-design tools, internal and external to Barrelfish, followed by the review of the Server Base System Architecture in the second section. This is then followed by a comprehensive description of mappings of language constructs from Sockeye to LISA+. These mappings are used to construct a compiler that generates working system models in LISA+ from Sockeye descriptions files. Finally, a minimal system model is shown and evaluated, followed by a final overview of the entire project in addition to some recommended future work that can be done to expand on these results.

3 Background

3.1 Existing Work

Work in the field of hardware software co-design includes everything ranging from system modeling to automatic hardware synthesis and software verification. This project focuses on modeling and system simulation, therefore existing hardware modeling and simulation tools, including low level hardware description languages like VHDL and Verilog as well as full fledged system simulators like gem5, are examined. These tools are considered as possible targets for this project to use for system simulation.

3.1.1 VHDL[5] & Verilog[6]

VHDL and Verilog are hardware description languages used to model electronic systems and digital circuits. These languages are low level and are frequently used for designing physical hardware and cover many finer levels of detail such as propagation time, signal edges and blocking/non-blocking assignments. In terms of hardware software co-design there exist various hardware simulators that use these languages to define hardware and then run other programs on the generated system.

3.1.2 SystemC[7]

SystemC is set of C++ classes and macros that can be used to construct a hardware simulator. It relates closely to VHDL and Verilog in regards to low level implementation details but also provides a higher level of abstraction to facilitate understanding and workflow, classifying it as a system level modeling language rather than a hardware one. It also provides additional utilities that can be used to measure power and energy. SystemC is often used as a base for other system modeling tools, for example LISA+.

3.1.3 gem5[8]

gem5 is a combination of two older simulators, M5 and GEMS. It provides a modular architecture and high level abstractions to facilitate modeling. There is also a large set of predefined architectures, including x86, ARM, MIPS, PowerPC and SPARC, which makes it a very attractive option for operating system developers that wish to target multiple platforms. Configuration is done via Python but the simulator itself is implemented in C++ to provide a performant and accurate system. Additionally, gem5 provides integration with existing SystemC projects, which can be used to co-simulate additional functionality on top of the existing system.

3.2 Sockeye[1, 2]

Sockeye is declarative domain specific language developed by the Barrelfish group to describe systems on a chip. The language is used to describe what is known as a hardware decoding net. This is essentially a model of a hardware

system that encapsulates the interactions between systems, in particular memory and interrupts but also has support for other domains like clock systems and power.

3.2.1 Sockeye's Use Case

The Barrelfish group uses Sockeye to describe systems that the operating system targets. At its core, Sockeye translates to a series of Prolog statements, a logic programming language, that can be queried for information about the system. Barrelfish's internal system knowledge base is built on top of these statements. This is used by the operating system during runtime to query hardware system properties and perform appropriate tasks based on the results, i.e. system component and peripheral discovery. Another use of this specification is formal verification of systems. Sockeye can also be used to construct a model in Isabelle, a formal verification language, which in combination with the Barrelfish implementation can provide guarantees about system safety and performance.

3.2.2 Sockeye Concepts

Nodes Sockeye provides an extensive system to describe what are called nodes. These nodes represent components in a hardware system and can be connected to one another, just like hardware. Nodes cover multiple domains, including memory, interrupts, power and clocks, which help to describe the type of connections a node has as well as its functionality. Each node also has definitions as to how it behaves: accepting, which implies that the node is end destination for a set of addresses, and mapping, which implies that the node translates one set of addresses to another.

Modules Sockeye organizes these nodes into modules. Modules contain nodes and definitions on how these nodes connect to others. Additionally, modules can also instantiate sub-modules and bind to their input and output ports. This allows for mapping and passing address domains into and out of components to model the given system. In fact, the entire system is modeled as one top-level module, usually with various other sub-modules. The ability to instantiate sub-modules also allows for reuse of common components such as UARTs, DRAM, etc.

3.2.3 Goals using Sockeye

Sockeye was chosen for this project not only because it can provide a comprehensive description of a system, but also because of its convenience, since the Barrelfish group already has systems modeled in it. The goal is to be able to take these files and generate a working system image that can then be used to run and debug the operating system as well as allow testing on multiple platforms with little to no overhead. Additionally this can be used to test systems that do not physically exist from hardware manufacturers yet, but have a preliminary specification so that Barrelfish can already be compatible when the system is released.

3.3 LISA and LISA+[11]

LISA, the Language for Instruction Set Architectures, is a language developed by ARM to describe instruction set architectures. This was later extended to LISA+ which added the capability to model entire systems as well as the individual components that they are comprised of, like Sockeye does. LISA+ is closely tied to the C and C++ languages, utilizing them to define behaviors. In contrast to Sockeye's hardware decoding net, LISA+ focuses primarily on more hierarchal physical descriptions and overall system structure, interactions and behaviors.

3.3.1 LISA+ Concepts

Components LISA+ uses components to represent a system. Each component can have a multiple subcomponents that comprise further functionality. Each component includes the following sections: resources, includes, composition, behavior, ports and connections.

Resources The resources section contains all local resources declarations that belong to a component. These declarations fall into one of three classes: registers, memory and parameters. Registers define a resource that stores data. They represent the physical registers in hardware and have support for various attributes and annotations, including read/write access limitations, bit width and address. Memory defines a range of space where data can be stored. It represents physical memory banks and also can be annotated with various attributes such as read/write access limitations, endianness and minimum addressable size. Parameters give a way increase component configurability by abstracting values to be defined at a later point, such as compile time or even run-time. They can be given default values to use that can then be overridden at a later point.

Includes The includes section is a place to put `#include` preprocessor statements. Similar to the C/C++ preprocessor it will insert the linked files and behaviors where specified.

Composition The composition section is where subcomponents can be defined. This section exists to provide a way to construct a system component hierarchy in order to utilize the reusable components. In addition, the parameter values of the subcomponents can be specified here. The parameters are specified by name, meaning values can be left to their given defaults or overridden with another desired value. These parameter values can also be other parameters, allowing a component to pass configuration to a higher level.

Behavior The behavior section defines the behavior of the component. Multiple different behaviors can be specified in this section, similar to functions, which are called at the appropriate times. There are also several special purpose behaviors that act like hooks for specific system actions, such as initialization, reset or termination. It also provides behaviors that can control the simulation itself, such as starting or stopping. These behaviors are written with C like syntax and can also be connected to external C or C++ code.

Ports Ports allow components to communicate with one another. There are three different types of ports: internal, master and slave. Internal ports are not exposed externally and can be used to wire together subcomponents inside of the component. Master ports are external ports that can be used to request read or write accesses on a connection. In contrast, the slave ports are also external ports but they respond to these read or write accesses. Ports communicate using protocols and a master and slave must implement the same protocol if they are connected. Ports can also be marked as addressable, which allows partial connections and connections to multiple other ports.

Connections Connections permit the wiring of component ports with one another. The connections are limited to the internal ports of a component or any ports of included subcomponents from the composition section. Here address ranges can be specified if the port is addressable.

Protocols Protocols specify the behavior of connections and how ports may interact. The behaviors are written similar to the component behaviors mentioned in the previous section. Each protocol behavior can be specified as optional, meaning there exists a default implementation, and as master or slave, corresponding to what type of port this behavior targets. The most used protocols are the `PVBus`, `Signal` and `ClockSignal` protocols, which are used for memory IO, interrupts and clock signals, respectively. Other examples of protocols include `SerialData` for serial communication and `VirtualEthernet` for ethernet communication.

3.3.2 ARM Fast Models[9]

ARM provides a framework, built on top of LISA+, to build system models, known as ARM Fast Models. This was first released with their Fixed Virtual Platforms[10], which provide explicit platform definitions that developers can run as a simulator. With the Fast Models, developers can specify their own systems instead of relying on the standard platforms that ARM provides. The framework comes with a set of standard library components that reflect specific hardware that ARM has released, such as the `PL011_UART` or the `GICv3`. Since ARMv8 is one of the platforms Barrelfish targets, it makes sense to use a framework provided by the manufacturer as well as their predefined components.

4 ARM Server Base System Architecture

ARM is working on providing a standardized hardware system architecture using the ARMv8 architecture. This platform is known as the ARM Server Base System Architecture, or ARM SBSA for short. The intention of this standardized system is to provide a simplified hardware interface to firmware and operating system developers to increase reliability, portability and compatibility amongst independently developed hardware and software systems. This is in direct response to the disorder of the ARMv7 systems which had no such unifying specification. It is an optional specification but ARM anticipates hardware manufacturers will follow it in order to ensure their systems' interoperability. Additionally, the manufacturers are free to add additional functionality as they see fit as ARM does not want to restrict these platforms, only provide a common interface.

4.0.1 Architecture

The server base is divided into multiple tiers of functionality, each level building off of the previous. As of right now, the first three levels (0, 1, and 2) have been removed from the specification, making level 3 the newest standard, and levels 4 and 5 have been added. The level 3 specification provides a list of various components and functionality that a system must provide in order to comply with the standard. It includes specification of processing elements, memory maps, interrupt assignments, I/O virtualization, clock/timer subsystems, watchdogs and peripherals. It also describes the behavior of the wakeup and power systems. There also exists an optional additional specification, known as level 3 firmware, which further specifies some of the previously mentioned components and behaviors. The next level, level 4, adds additional requirements for the processing elements as well as additional MMU and PCI Express requirements. Level 5 goes on to refine the processing element requirements even further and also touches on the interrupt controller, the MMU and the clock.

More specifically, the level 3 specification provides a list of various feature requirements that processing elements must provide in order to comply. These requirements include being ARMv8 compatible and supporting features such as paging/superpaging support, being little endian and having individual PPIs, or private peripheral interrupts. The PPIs are given specific interrupts IDs as well. It also requires the use of secure execution levels by the system firmware to ensure firmware integrity. The interrupt controller has to conform to ARM's GICv3 standard. If the I/O virtualization is implemented, the system must include a compatible stage 2 system MMU, in the form of an SMMUv2 or SMMUv3. For clock and timer purposes, the system must provide a generic timer that holds the system counter and runs at least at a frequency of 10MHz and does not roll over within 10 years. It goes into detail about the wakeup behavior of the processors and timers, specifying that these timers must be able to wake the processors via private peripheral interrupts.

4.0.2 Importance to Barrelfish

As operating system developers, the Barrelfish group is extremely interested in the SBSA. Being able to target one system architecture specification and

then being able to run on all compliant systems saves time and effort as well as increasing the overall appeal of the operating system. Part of this research focuses on analysis and critique of the SBSA. This includes discussing how the SBSA can be integrated into the Barrelfish ecosystem, what sort of issues and incompatibilities may be present between the two systems as things stand, and some recommendations as to what would improve the SBSA. The aim is to be able to provide comprehensive feedback on the usefulness of the SBSA to ARM.

4.1 Translation of the SBSA

For the description of the SBSA in Sockeye, the individual component requirements are inspected and translated as closely as can be represented in Sockeye. One recurring problem that is seen is the fact that most of these components require concrete implementations to actually be used in a system. This means the behavior and physical addresses must be defined before the system can be instantiated. However, the representation of the SBSA is still translated to Sockeye to demonstrate how such a system would look. A concrete implementation is presented in a later section that utilizes components from the ARM Fast Models standard library in place of the generic components mentioned in the SBSA to create a concrete system model.

PE Architecture The processing elements are mostly described in a behavioral fashion, something that Sockeye is not designed to deal with. Nevertheless, the connections of the individual elements are representable. One thing to note however is that the additional behavioral requirements, such as cryptography support, presented by levels 4 and 5 are unable to be represented differently than the level 3. This makes it impossible to differentiate which level the given Sockeye description is for. See the processing element entry in the appendix for an example.

Memory Map The memory map is a more complicated issue. The specification states that there is no mandated standard memory map, but that the system memory map is left to the firmware data. This is problematic for trying to generalize over the SBSA itself because Sockeye as well as LISA and simulators in general require a well defined description of a system in order to model it properly. Sockeye is able to represent these individual components, like the UART, Watchdog or optionally the SMMU, but cannot instantiate a working system without specified concrete addresses. The fact that the entire address space is mappable is possible to represent in Sockeye but again is not of much use without a concrete memory map.

Interrupt Controller The interrupt controller is fairly straightforward to translate to Sockeye. Interrupts are already representable using the interrupt domain and the interrupt mappings are well defined and simple to express using Sockeye as well. See the generic interrupt controller entry in the appendix for an example.

I/O Virtualization I/O virtualization is an issue as well, similar to the memory map. Primarily, support for it is implementation specific and therefore op-

tional. However if it is present, the system has to include an SMMU which is modelable in Sockeye, butthe address of the SMMU is determined from the system firmware and therefore also requires a concrete system image. Additionally, if the system supports virtualization, all memory must be rerouted through the SMMU, which changes the layout of the entire system. Having something this fundamental be optional can lead to further inconsistencies amongst implementations and make it more difficult to model properly.

Clock and Timer Subsystem Clocks are well defined in Sockeye so this representation is also fairly straightforward using the clock and interrupt domains. These connections are simple and can therefore be easily modeled. See the generic timer entry in the appendix for an example.

Wakeup Semantics & Power State Semantics Wakeup semantics and power state semantics are primarily behavioral aspects of the specification. Since Sockeye does not deal with component behavior, these specifications are just assumed to work correctly with the corresponding component connections that are included in the clocks and processing elements. Additionally, the power domain is currently undergoing some changes to provide a more comprehensive representation of these systems and therefore this will be deferred until that work is concluded.

Watchdogs The generic watchdog requires the clock and interrupt domains. Just like the clock and timer subsystems, these are already present and therefore trivial to represent in Sockeye. See the generic watchdog entry in the appendix for an example.

Peripheral Subsystems The peripheral subsystem is also mostly implementation specific. It requires the adherence to certain standards, such as XHCI and TPM, if the system components are present. These are all behavioral attributes and therefore are not of much relevance to Sockeye. However, one thing it does specify is the inclusion of a UART. This is fairly trivial to express but, as previously stated, the specification has some minor details, including that the UART has to be non-secure and how to route the interrupt, but no information about concrete placement. See the generic uart entry in the appendix for an example.

4.2 Conclusion

Overall there are several up and down sides to the SBSA as well as some incompatibilities with Sockeye. The SBSA gives us certain guarantees about what system functionality is to be expected, like a UART or a watchdog timer, but information on accessing these services is left to the system firmware. This can be useful when including precompiled software that relies on these components as they are guaranteed to work. However, specification also relies heavily on optional or implementation specific constructs, which not being able to rely on certain services being present defeats the purpose of having a standard. Additionally, there are some other features that would greatly help operating system

developers adapt to the SBSA and more specifically help representation of the server base in languages like Sockeye.

Secure State In the specification it is mentioned that the systems are expected to use secure state. The fact that this is expected and not required is somewhat problematic. This can lead to inconsistencies amongst hardware and software implementations for the server base.

Memory Map There is no standard memory map given in the specification. This means that system models cannot actually represent a generic implementation of the SBSA without making assumptions about concrete addresses and mappings. Having to make these assumptions severely limits the utility that the specification provides because they may conflict with actual implementations. Having to utilize the system firmware to retrieve memory mappings is what systems are currently doing, using ACPI and the like. The type of firmware is not specified either, leaving interaction with the system to be implementation specific as well. Since the specification is fairly vague in this regard, it ends up providing little to no additional information and reliability in terms of system capabilities.

I/O Virtualization I/O virtualization is completely optional. Having an important system such as virtualization be optional prevents developers from being able to rely on its benefits. Therefore, developers do not gain any actual advantage from the SBSA and have to plan for determining support on their own.

Level 3 - Firmware Just like I/O virtualization, the level 3 firmware extension is also optional and suffers from the same pitfalls. In addition, it seems inconsistent with the multi-tier structure. Why not make this another tier of the architecture? Also, do higher tiers have the option to implement this as well, e.g. how is a level 4 system that is also compliant with the level 3 firmware extension expressed any differently than a standard level 4 system? These inconsistencies should definitely be defined in a more clear and structured way.

Final Thoughts In summary, the SBSA needs a stricter set of requirements. This should include a well defined method of interacting with the system firmware, such as mandating ACPI compliance or some other common firmware interface. Having the firmware and the actual system details both be implementation specific leaves a large amount of room for uncertainty and incompatibility. Furthermore, the behavioral specifications and component requirements are only beneficial to a limited extent. ACPI and other system firmware interfaces already provide mechanisms for peripheral discovery and the like and the SBSA provides little benefit on top of this. Also, the lack of a concrete memory map becomes impossible to formally reason about and represent a system this abstract, further limiting it's use.

5 Design & Implementation

Even though Sockeye and LISA+ are not designed for the same purpose, there still exists significant overlap in the language concepts. Nevertheless, there are still some parts of Sockeye that do not exist directly in LISA+ and have to be mapped in some other way or can not be represented at all. For the sake of simplicity, Sockeye is translated to LISA+ in a syntactic manner rather than from the instantiated model of the hardware decoding net. This greatly simplifies the conversion because of the similarities in the syntax and prevents the need to re-extract details such as module instantiations from the decoding net model. This section covers what subset of Sockeye is covered by this project as well as what assumptions are made for some of the language constructs. It also talks about the mappings that do exist between these languages and how they are constructed.

5.1 Restrictions & Assumptions

Property Expressions In Sockeye, it is possible to annotate certain node definitions with parameters in the form of boolean expressions, for example `read && !write`. This provides additional information for the connection but this cannot be represented in LISA+. While LISA+ does support `read` and `write` annotations, it does this on the individual memory nodes and registers rather than per connection. Property expressions are therefore ignored.

Conversion Nodes Sockeye provides a way to convert domains via conversion nodes. Since LISA+ uses protocols to define connections and connected nodes need the same protocol in order to communicate, this is not representable with the current tools. LISA+ does do some conversions internally but these are implemented with behaviors and would require custom component behavior, which this project does not cover. Conversion nodes will cause an error at compile time since it has undefined behavior.

Power Domain In parallel to this project, the power domain in Sockeye is being re-examined and redefined to make sure that it properly captures all the desired functionality. Since power is still under construction, implementation of this domain is deferred and will be revisited when that work is concluded.

Wildcards Wildcards in Sockeye are used as syntactic sugar to speed up writing the specification files. They ideally should represent the original address range that was specified in a node's declaration but are repurposed for use with LISA+. LISA+ differentiates between addressable and non-addressable ports, where it is invalid to access a non-addressable port with an address. This mainly affects the clock domain and parts of the interrupt domain but also plays into how the memory nodes were translated. In this new context, wildcards will be translated as specifying no address, which represents a direct connection between nodes. Since this change affects memory nodes as well, any definitions for memory nodes should explicitly state the address range they target and should avoid using wildcards outside of the previously mentioned context. On the other hand, clocks have to use wildcards to be mapped properly. Interrupts vary by case, depending on if the interrupt is a single line or a vector.

Accepting Sockeye allows all nodes to accept, meaning that the node is the endpoint for that range of addresses. In Sockeye, it makes sense to list nodes as endpoints for clocks, interrupts and memory but in LISA+ these interactions are defined by behavior. For example, if a node accepts some interrupt vector, it will provide some functionality that is tied to the port to deal with the incoming interrupt. This works well for generic memory since it just store some data and has no real behavior behind it, but other domains cannot be expressed without behaviors. For this reason, accepts for the memory domain are handled but are ignored for interrupts and clocks.

Forall Statements Forall statements are another problem for LISA+. Sockeye's forall statement can be used to condense descriptions and make them easier to read. However, when this is coupled with parameterization, it can lead to issues for LISA+. In most cases the statements can be unrolled into other statements, but many descriptions parameterize foralls. Since the parameters are unknown at compile time, it becomes impossible to unroll them and LISA+ has no similar language construct to represent them either.

Module Array Instantiations Similarly to the forall statements, module array instantiations have issues with parameters. Normally, the instantiations could be flattened or prefixed in some way in order to emulate the effect, but this again becomes an issue with parameterization. In many cases descriptions can have a variable number of cores which are instantiated in this way but LISA+ does not support any form of array instantiation.

Node Arrays Unlike the modules, nodes are translated to ports instead of components, which do support array instantiation. Nevertheless there is an issue with the way array sizes are represented in Sockeye vs LISA+. Sockeye lets the user specify a range whereas LISA+ only lets the user specify an array size. Additionally, these sizes cannot be parametrized or include expressions. Because of this, node arrays in Sockeye should provide a base/limit range that starts at 0 and ends at the desired size.

Integer Overflow The translation from Sockeye to LISA+ uses natural number operations to maintain the use of parameters in ranges and expressions. LISA+ supports these expressions but calculates them using 64 bit integers. Therefore some expressions may overflow, such as a `(0 bits 64)` because of a 64 bit shift.

Named Types The current Sockeye parser has a small bug in the way it parses named types. It reads them as a singleton range with a constant value rather than a named type, which cannot be properly parsed. Since this parsing is not working correctly, translation of the named types to LISA+ will have to wait until the underlying parser implementation is fixed.

5.2 Mappings

Natural Number Operations Sockeye supports several different types of expressions to manipulate natural numbers. Some of these expressions are di-

rectly supported by LISA+, more specifically: Literals, Addition, Subtraction, and Multiplication. Since certain expressions are parameterized and these parameters can change at runtime, any expressions are translated to expressions that LISA+ can also evaluate at runtime. LISA+ supports expressions in most places, these operations are translated so that they can then be evaluated at LISA+'s runtime. Further operations are defined as follow:

Slice The slice operation takes an expression and a range to specify which bits to extract. In essence, this represents a bitwise shift followed by a bitmask. Note that the ranges are inclusive. The pseudocode is as follows:

```
fn slice(value, range_low, range_high) {
    shift = value >> range_low;
    mask = (1 << (range_high - range_low + 1)) - 1;
    return shift & mask;
}
```

Concat The concat operation takes an expression and a slice operation to specify what to concatenate. This can be expressed as a shift and a bitwise or. Note that the ranges are inclusive. The pseudocode is as follows:

```
fn concat(value, concat, range_low, range_high) {
    shift = value << (range_high - range_low + 1);
    return shift | concat;
}
```

Address Ranges Sockeye provides multiple formats for specifying addresses. These are: Singleton Range, Base/Limit Range and Bits Range. LISA+ ranges are written as a minimum address and a maximum address and therefore the ranges must be converted into this format. These are implemented as follows:

Singleton Range The singleton range specifies a range with a single value. This value is interpreted as a vector index for the purposes of interrupts.

Base/Limit Range The base/limit range specifies a range given a base address and a limit address. The base is used as the minimum address and the limit is used as the maximum.

Bits Range The bits range specifies a base address and a number of bits that can be specified. The base is used as the minimum address and the maximum is the given by `base address + ((1 << bits) - 1)`.

Address Sets Sockeye permits address sets as well as single address ranges. Since LISA+ does not have support for sets, these addresses must be flattened to a single range. This is done by multiplying out the minimums to get the set's minimum and the maximums to get the set's maximum. The individual addresses require knowing the max ranges of the node definition.

Node Definitions Node definitions are represented as LISA+ ports. Input/Output nodes are translated to slave/master ports, respectively. Internal ports remain internal ports in LISA+. Since ports require protocols in LISA+, each used domain is assigned a matching protocol. The clock domain maps to the `ClockSignal` protocol, the interrupt domain maps to the `Signal` protocol and the memory domain maps to the `PVBus` protocol.

Maps Mapping for the clock domain is very straightforward. The addresses are all wildcards since the clock is a signal in LISA+ and not a range. For interrupts there exists the option of mapping directly or optionally specifying a singleton range to specify the vector index that is to be mapped. The memory addresses are slightly more complicated. Memory ranges in LISA+ cannot simply be routed using a port, but must use the `PVBusDecoder` component from the standard library to map the ranges. Therefore, whenever a memory node is defined and provides some mapping, a new subcomponent is added to the composition section with the well known `_DECODER` suffix and the entire node is routed to the decoder component. Any mappings are then mapped from the decoder instead of the node itself.

Accepts Accepts are implemented for memory only. When a memory node accepts, an instance of `RAMDevice` is created for the node, with the suffix `_MEMORY`, that has the same size as the accepting node. Since the node may provide arbitrary mappings to this memory, a `PVBusDecoder` must exist for it, either being newly created or reused if one already exists for this node. The addresses can then be freely mapped from the node to the instantiated memory.

Overlays Overlays are supported for all domains. Connection ranges in LISA+ may overlap one another but later connections override previous ones. This is convenient to implement overlays with, since putting overlays first will allow other mappings or accepts to simply override part of the connection as needed. The overlay connections cover the entire range of the node.

Binds Binds are simple to represent in LISA+ as well. A bind creates a connection from the target port to the internal port on the component, with an optionally specified address range.

Constants Constants are translated as parameters in LISA+. Since LISA+ has support for parameters default values, these can be conveniently used as constants since this parameter is not publicly exposed and therefore will not be overridden. The parameter is of type `int` as the constants in Sockeye must be natural numbers.

Modules Modules are represented as components in LISA+. They are instantiated in a very similar way to Sockeye and can be directly mapped over.

Parameters Parameters are slightly more tricky but not too much. LISA+ component arguments require names. Since the names are given in the module declaration in Sockeye, they can just be added in order of declaration to the arguments given to the instantiation.

5.3 Existing Sockeye Framework

Sockeye's existing implementation is written in Haskell. Primarily, it acts as a parser for `.soc` files that returns an abstract syntax tree. There is also a backend implementation for both Prolog and Isabelle which takes the abstract syntax tree and transforms it into code for the respective language. In a similar fashion, the new addition will take the form of a LISA backend and output code that can then be used to generate a system model. One thing to note, however, is that as of right now the parser does not perform any linking or symbol interpretation. Languages like Prolog do not require this as they can figure out the linking themselves, however this is not necessarily the case for LISA and can affect the implementation details.

5.4 Existing LISA+ Framework

A large part of the utility of LISA+ comes from its existing library of standard components. This project relies on a variety of those components in order to work properly and also greatly facilitated the development. At the core of the library sit the multiple processor models that ARM provides. For testing, the `ARMCortexA57x1CT` component was used. The `PVBusDecoder` and `RAMDevice` components were used to implement accepting and mapping for memory nodes. Additionally there are provided examples in the form of the ARM Fixed Virtual Platforms, which are standard system implementations from ARM built using LISA+. These were instrumental to the understanding of LISA+ and used to reverse engineering an initial working system image.

5.5 Additional Utilities

In addition to the Sockeye compiler, this project uses various other utilities to help generate the system and boot Barrelfish. The bootloader and UEFI implementations are provided by Linaro, a company that specializes in tools for the ARM ecosystem. Additionally, another Barrelfish internal tool, Hagfish, is used to bootload the kernel with the expected multiboot structure. Since Hagfish can be booted as an UEFI application, using the Linaro bootloader and UEFI implementations make the boot process very straightforward and simple to do.

5.6 Extension of Sockeye Syntax

In order to utilize all of the functionality that LISA provides, support for *extern* modules was added. This serves to incorporate the existing components that are provided by the LISA standard component library. The proposed *extern* syntax is as follows:

```
'extern' 'module' Ident '(' (Address Ident)* ')' '{'
    // input/output node definitions
'}'
```

The extern modules support both named parameters and input/output ports but no internal nodes or node definitions. The named parameters are supported by LISA and the ports help to define the I/O of the module. In essence it allows

for binding to external modules/components without having to know or specify the internal implementation. Since the standard module also contains nodes and node definitions, these can be safely ignored inside an external module. In the future this can also be checked in the semantic checker to prevent such definitions.

5.7 Auxiliary File

The majority of mappings from Sockeye to LISA+ work very well, but there are some exceptions. Primarily these exceptions stem from attempting to represent certain LISA+ constructs in Sockeye when trying to work with the library components. For example, LISA+ components can have required parameters of types other than natural numbers, including strings and booleans. Since Sockeye cannot represent these parameters another solution is needed. One solution would be to introduce these argument types to the language but the whole idea of this project is to map the language, not slowly change it into LISA+. The solution that was chosen was to have an auxiliary file, in the form of a JSON file, to provide the additional information.

The format of the file reflects the issues that it covers. The first is the mentioned parameter issue. The JSON lets the user specify the component instance and module by name and can optionally redefine the value that should be given in LISA+. Another issue that arose is the fact that Sockeye identifiers do not cover the same set of characters, like the "-" character, which is interpreted as a minus symbol in Sockeye. In addition to replacing the parameter value, there is also the option to change the name of the given parameter. A third issue that arose was mapping connections that utilize protocols other than `PVBus`, `ClockSignal` or `Signal`, like UARTs, which use the `SerialData` protocol. In the JSON, there is a section to specify connections directly by giving the name, port and optionally an address. The syntax of the JSON file is as follows:

```
{
    "connections": [ // Multiple connections are allowed
        {
            "moduleName": /* string */,
            "source": {
                "name": /* string */,
                "port": /* string */,
                "address": /* optional string */
            },
            "target": {
                "name": /* string */
                "port": /* string */,
                "address": /* optional string */
            }
        }
    ],
    "parameters": [ // Multiple parameters are allowed
        {
            "moduleName": /* string */,
            "component": /* string */,
```

```
            "name": /* string */,
            "translation": /* optional string */,
            "value": /* optional string */
        }
    ]
}
```

6 Results & Discussion

6.1 Evaluation

Overall the translation from Sockeye to LISA+ is successful. The generated system image boots into UEFI using the Linaro bootloader. There are some minor complaints from the system about missing components that were stripped away but nothing that leads to a system failure. It then proceeds into the UEFI subsystem, where it can load the kernel using the Hagfish bootloader. The terminal outputs the standard Barrelfish boot information and successfully allocates a RAM capability to initialize the system. See the appendix for a description of the minimal system image.

6.2 Dynamic Components

In order to check the robustness of the implementation, components were moved around. The system image that was used has two UARTs which are each attached to their own telnet consoles. When booted, the bootloader and kernel output using the UART connected to `terminal`. To show that the system responds to component changes, the UARTs were swapped. As expected, the new system output to the other UART, which is connected to `terminal1`. Additionally, another image was tried with reduced memory regions. When these addresses are unavailable, the kernel fails to allocate a RAM capability and panics accordingly. Clearly the system generated responds to the changes in the system description in expected ways.

6.3 Efficiency

While the generated models are very good in terms of performance and component mapping, they are not perfect. The generated system model produce unneeded components, specifically the additional internal ports, memory decoders and RAM devices. Many of these connections are a result of Sockeye only being able to map submodule nodes to internal nodes. LISA+ supports directly mapping submodules ports to other submodule ports and therefore can circumvent the need for internal utility nodes. The impact of these additional connections is yet to be profiled. The efficiency of the compilation process itself is negligible in comparison to the rest of the Barrelfish build process. The compiler time is in the milliseconds range whereas the build process is more towards the minutes range.

6.4 LISA+

LISA+ provides a comprehensive set of language constructs and other utilities required to model a system. The language itself is fairly straightforward and simple to use and several of the provided library of components are extremely useful to have. Additionally, debugging with the built in debugger frequently comes in handy when inspecting memory layouts. Not all of the language aspects were used, like behaviors, but rest was extremely helpful for this project.

There is however one minor complaint concerning the components in the standard library. First off, there are several components used in the Fixed Virtual Platforms that aren't included in the standard library. One of these is

the `IntelStrataFlashJ3`. This component is already implemented by ARM and wouldn't have to be reimplemented or downloaded independently. Furthermore, the standard library provides certain components without the required associated components, for example, the `FlashLoader` component. This component requires some flash memory that uses the `FlashLoaderPort` protocol to load in data but no components, like the `IntelStrataFlashJ3`, are included in the standard library order to utilize it. Finally, there are several components that behave like some magical black box with inputs and outputs. The `GICv3IRI_Filter` component has externally defined behavior and just lists a series of input and output ports. Since this component is more or less just an aggregate of smaller subcomponents and their behaviors, it would be beneficial to implement this component in a more idiomatic way using subcomponents, internal behavior and the like. Not only would this benefit developers more since they then have the opportunity to inspect these components to debug functionality as well as learn from the implementation, but in general it would also be clearer as to what the expected behavior and usage of this component is from the system's perspective.

6.5 Closing Remarks

The goal of translating a Sockeye model into a fully functional LISA+ simulator was successfully achieved. There are limitations on both ends that cause some problems with the translation, which may require special attention to choices of syntax or prevent the translation altogether, but a large subset of Sockeye works for this purpose. Also, a slight modification to Sockeye itself was made in the form of the new `extern` syntax. Using these tools a minimal system image was modeled in Sockeye and translated into a fully functional LISA+ simulator. This model was also used to test the compiler's robustness by swapping components, like the UARTs, around as well as restricting memory address to check behavior under these conditions. In both cases, the model functioned as expected, properly swapping which terminal the system was communicating with and panicking on the missing addresses, respectively. In addition to the compiler, the ARM System Server Base Architecture was also evaluated. Overall the specification was found to be a solid basis for behavioral requirements but provided too much freedom for implementations, leading to uncertainty regarding available features from a developer's perspective.

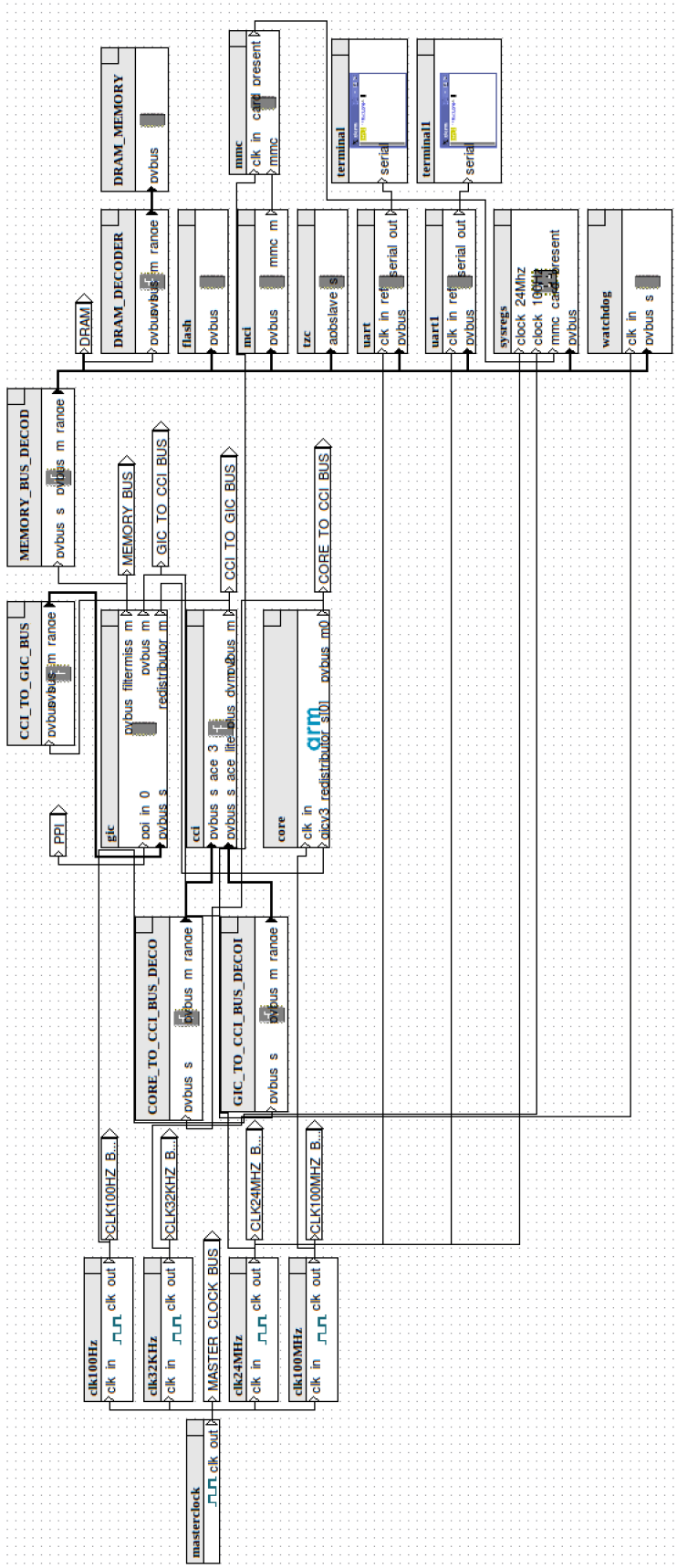

Figure 2: The barebones system image in LISA+.

7 Future Work

7.1 Integration with Mackerel[3]

In addition to Sockeye, the Barrelfish group has developed several other tools to aid with hardware/software co-design. Among these is Mackerel, a device specific language used to describe hardware devices, specifically register formats and hardware data structures. These descriptions can be used to generate C header files that can be included directly in the source code of Barrelfish itself.

One future extension to Sockeye could be the integration of Mackerel into module descriptions to further specify the physical register layouts of the individual components. For example, in the SBSA the register formats are given for the watchdog and UART. Using Mackerel, these registers could be specified to provide a more comprehensive model of the system as well as to then generate the corresponding registers in LISA as well.

7.2 Extension of Library Components

Since the behavior of the models used in this project depends solely on the existing library components, it would be potentially beneficial to write a custom library to describe other behaviors not found in the standard library. This could include new components that are not necessarily from ARM, such as x86 systems, just additional crucial system components that are needed by a more complex system model or just add some basic utility components that can be included for something like protocol conversions.

7.3 Power Domain

As previously stated, the power domain is currently undergoing some fundamental changes. As soon as these changes are implemented, a future improvement would be to figure out how to map the concepts over and reflect these changes in the existing code base.

8 Appendix

8.1 SBSA Sockeye Files

8.1.1 SBSA

```
module SBSA(nat cores,
            nat uartbase,
            nat gicbase,
            nat rfbase,
            nat cfbase) {
    instance TIMER of GenericTimer
    TIMER instantiates GenericTimer(cores)

    TIMER binds [
        CNTNSIRQ[*] to CNTNSIRQ[*];
        CNTSIRQ[*] to CNTNSIRQ[*];
        CNTHPIRQ[*] to CNTHPIRQ[*];
        CNTVIRQ[*] to CNTVIRQ[*]
    ]

    intr (0) CNTNSIRQ[0 to cores - 1]
    intr (0) CNTSIRQ[0 to cores - 1]
    intr (0) CNTHPIRQ[0 to cores - 1]
    intr (0) CNTVIRQ[0 to cores - 1]

    forall core in (0 to cores - 1) {
        PROCESSING_ELEMENTS[core] binds [
            MEMORY to MEMORY
        ]

        CNTNSIRQ[core] maps [
            (*) to PROCESSING_ELEMENTS[core].CNTNSIRQ at (*)
        ]

        CNTSIRQ[core] maps [
            (*) to PROCESSING_ELEMENTS[core].CNTSIRQ at (*)
        ]

        CNTHPIRQ[core] maps [
            (*) to PROCESSING_ELEMENTS[core].CNTHPIRQ at (*)
        ]

        CNTVIRQ[core] maps [
            (*) to PROCESSING_ELEMENTS[core].CNTVIRQ at (*)
        ]
    }

    instance PROCESSING_ELEMENTS[0 to cores - 1] of ProcessingElement
    PROCESSING_ELEMENTS[*] instantiates ProcessingElement
```

```
    instance GIC of GenericInterruptController
    GIC instantiates GenericInterruptController

    instance UART of GenericUART(uartbase)
    UART instantiates GenericUART

    instance WATCHDOG of GenericWatchdog(rfbase, cfbase)
    WATCHDOG instantiates GenericWatchdog

    memory (0 bits 64) MEMORY

    MEMORY maps [
        (0 bits 15) to GIC.MEMORY at (gicbase bits 15);
        (0 bits 12) to WATCHDOG at (rfbase bits 12);
        (0 bits 12) to WATCHDOG at (cfbase bits 12);
        (0 bits 6) to UART at (uartbase bits 6)
    ]
}
```

8.1.2 Processing Element

```
module ProcessingElement {
    input intr (0) CNTNSIRQ
    input intr (0) CNTSIRQ
    input intr (0) CNTHPIRQ
    input intr (0) CNTVIRQ

    output(0 bits 64) MEMORY
}
```

8.1.3 Generic Interrupt Controller

```
module GenericInterruptController {

    input memory (0 bits 15) MEMORY
    MEMORY maps [
        (0x1000) to DISTRIBUTOR at (0 bits 12);
        (0x2000) to CPU_INTERFACE at (0 bits 13);
        (0x4000) to VIRTUAL_INTERFACE_CONTROL at (0 bits 12);
        (0x5000) to VIRTUAL_INTERFACE_CONTROL_CPU at (0 bits 12);
        (0x6000) to VIRTUAL_CPU_INTERFACE at (0 bits 13)
    ]

    memory (0 bits 12) DISTRIBUTOR
    DISTRIBUTOR accepts [
        (*)
    ]

    memory (0 bits 13) CPU_INTERFACE
```

```
    CPU_INTERFACE accepts[
        (*)
    ]

    memory (0 bits 12) VIRTUAL_INTERFACE_CONTROL
    VIRTUAL_INTERFACE_CONTROL accepts [
        (*)
    ]

    memory (0 bits 12) VIRTUAL_INTERFACE_CONTROL_CPU
    VIRTUAL_INTERFACE_CONTROL_CPU accepts [
        (*)
    ]

    memory (0 bits 13) VIRTUAL_CPU_INTERFACE
    VIRTUAL_CPU_INTERFACE accepts [
        (*)
    ]
}
```

8.1.4 Generic Watchdog

```
module GenericWatchdog {
    input memory (0 bits 12) REFRESH_FRAME
    REFRESH_FRAME accepts [
        (*)
    ]

    input memory (0 bits 12) CONTROL_FRAME
    CONTROL_FRAME accepts [
        (*)
    ]
}
```

8.1.5 Generic UART

```
module GenericUART {
    output intr (0) UARTINTR

    input memory (0 bits 6) MEMORY
    MEMORY maps [
        (*) to UART_FRAME at (0 bits 6)
    ]

    memory (0 bits 6) UART_FRAME
    UART_FRAME accepts [
        (*)
    ]
}
```

8.1.6 Generic Timer

```
module GenericTimer(nat cores) {
    output memory (0 bits 64) CNTVALUEB

    output intr (0) CNTNSIRQ[0 to cores - 1]
    output intr (0) CNTSIRQ[0 to cores - 1]
    output intr (0) CNTHPIRQ[0 to cores - 1]
    output intr (0) CNTVIRQ[0 to cores - 1]
}
```

8.2 Minimal System Image

8.2.1 Sockeye Description

```
extern module ClockDivider((0 bits 64) mul) {
    input clock (0) clk_in
    output clock (0) clk_out
}
extern module MMC {
    input clock (0) clk_in
    input memory (0 bits 64) mmc

    output intr (0) card_present
}
extern module PL180_MCI {
    input memory (0 bits 64) pvbus

    output memory (0 bits 64) mmc_m
}
extern module IntelStrataFlashJ3((0 bits 64) size) {
    input memory (0 bits 64) pvbus
}
extern module TZC_400((0 bits 64) master_id_from_label) {
    input memory (0 bits 64) apbslave_s
}
extern module TelnetTerminal {
    input memory (0 bits 64) serial_in
}
extern module PL011_Uart {
    input clock (0) clk_in_ref
    input memory (0 bits 64) pvbus
}
extern module GICv3IRI_Filter ((0 bits 64) reg_base, (0 bits 64) reg_base_per_redistributo
    input intr (0) ppi_in_0[0 to 15]
    input memory (0 bits 64) pvbus_s

    output memory (0 bits 64) pvbus_filtermiss_m
    output memory (0 bits 64) pvbus_m
    output intr (0) redistributor_m
}
extern module CCI400((0 bits 64) cache_state_modelled, (0 bits 64) broadcastcachemain, (0
    input memory (0 bits 64) pvbus_s_ace_3
    input memory (0 bits 64) pvbus_s_ace_lite_plus_dvm_2

    output memory (0 bits 64) pvbus_m
}
extern module SP805_Watchdog {
    input clock (0) clk_in
    input memory (0 bits 64) pvbus_s
}
```

```
extern module VE_SysRegs((0 bits 64) sys_proc_id0, (0 bits 64) sys_proc_id1, (0 bits 64) s
    input memory (0 bits 64) pvbus
    input clock (0) clock_100HZ
    input clock (0) clock_24Mhz
    input intr (0) mmc_card_present
}
extern module ARMCortexA57x1CT((0 bits 64) CLUSTER_ID, (0 bits 64) dcache_state_modelled,
    input clock (0) clk_in
    input intr (0) gicv3_redistributor_s

    output memory (0 bits 64) pvbus_m0
    output intr (0) CNTPNSIRQ
}
extern module MasterClock {
    output clock (0) clk_out
}

module Barebones {
    /* Composition */

    // Clocks
    instance masterclock of MasterClock
    masterclock instantiates MasterClock

    instance clk100Hz of ClockDivider
    clk100Hz instantiates ClockDivider(100)

    instance clk32KHz of ClockDivider
    clk32KHz instantiates ClockDivider(32000)

    instance clk24MHz of ClockDivider
    clk24MHz instantiates ClockDivider(24000000)

    instance clk100MHz of ClockDivider
    clk100MHz instantiates ClockDivider(100000000)

    // Memory
    instance flash of IntelStrataFlashJ3
    flash instantiates IntelStrataFlashJ3(0x4000000)

    instance mci of PL180_MCI
    mci instantiates PL180_MCI

    instance mmc of MMC
    mmc instantiates MMC

    // Serial
    instance uart of PL011_Uart
    uart instantiates PL011_Uart
```

```
instance terminal of TelnetTerminal
terminal instantiates TelnetTerminal

instance uart1 of PL011_Uart
uart1 instantiates PL011_Uart

instance terminal1 of TelnetTerminal
terminal1 instantiates TelnetTerminal

// GIC
instance gic of GICv3IRI_Filter
gic instantiates GICv3IRI_Filter(
    0x2f000000,
    0,
    0,
    0,
    224,
    0x2f020000,
    1
)

// Processor
instance core of ARMCortexA57x1CT
core instantiates ARMCortexA57x1CT(
    0,
    0,
    0,
    0x2C000000,
    0,
    1,
    1,
    1
)

// Watchdog
instance watchdog of SP805_Watchdog
watchdog instantiates SP805_Watchdog

// Miscellaneous
instance cci of CCI400
cci instantiates CCI400(
    0,
    0x0,
    0x8,
    0x7,
    0x0,
    0x2C000000
)
```

```
instance sysregs of VE_SysRegs
sysregs instantiates VE_SysRegs(
    0x07330477,
    0xff000000,
    0xff000000,
    0xff000000,
    1
)

instance tzc of TZC_400
tzc instantiates TZC_400(
    1
)

/* Local Nodes */
clock (0) MASTER_CLOCK_BUS
clock (0) CLK100HZ_BUS
clock (0) CLK32KHZ_BUS
clock (0) CLK24MHZ_BUS
clock (0) CLK100MHZ_BUS

memory (0 bits 48) MEMORY_BUS
memory (0 bits 48) CORE_TO_CCI_BUS
memory (0 bits 48) GIC_TO_CCI_BUS
memory (0 bits 48) CCI_TO_GIC_BUS
memory (0 bits 48) DRAM

intr (0) PPI[0 to 15]

/* Connection */

// Clocks
masterclock binds [clk_out to MASTER_CLOCK_BUS]
clk100Hz binds [clk_out to CLK100HZ_BUS]
clk32KHz binds [clk_out to CLK32KHZ_BUS]
clk24MHz binds [clk_out to CLK24MHZ_BUS]
clk100MHz binds [clk_out to CLK100MHZ_BUS]

MASTER_CLOCK_BUS maps [
    (*) to clk100Hz.clk_in at (*);
    (*) to clk32KHz.clk_in at (*);
    (*) to clk24MHz.clk_in at (*);
    (*) to clk100MHz.clk_in at (*)
]

CLK100HZ_BUS maps [
    (*) to sysregs.clock_100Hz at (*)
]

CLK100HZ_BUS maps [
```

```
    (*) to sysregs.clock_100Hz at (*)
]


CLK32KHZ_BUS maps [
    (*) to watchdog.clk_in at (*)
]

CLK24MHZ_BUS maps [
    (*) to uart.clk_in_ref at (*);
    (*) to uart1.clk_in_ref at (*);
    (*) to sysregs.clock_24Mhz at (*);
    (*) to mmc.clk_in at (*)
]

CLK100MHZ_BUS maps [
    (*) to core.clk_in at (*)
]

// Memory
gic binds [
    pvbus_m to GIC_TO_CCI_BUS;
    pvbus_filtermiss_m to MEMORY_BUS
]

core binds [
    pvbus_m0 to CORE_TO_CCI_BUS;
    CNTPNSIRQ to PPI
]

cci binds [
    pvbus_m to CCI_TO_GIC_BUS
]

CORE_TO_CCI_BUS maps [
    (0 bits 48) to cci.pvbus_s_ace_3 at (0 bits 48)
]

GIC_TO_CCI_BUS maps [
    (0 bits 48) to cci.pvbus_s_ace_lite_plus_dvm_2 at (0 bits 48)
]

CCI_TO_GIC_BUS maps [
    (0 bits 48) to gic.pvbus_s at (0 bits 48)
]

PPI[0] maps [
    (*) to gic.ppi_in_0[14] at (*)
]
```

```
    MEMORY_BUS maps [
        (0x000C000000 to 0x000FFFFfffff) to flash.pvbus at (*);
        (0x001C050000 to 0x001C05ffff) to mci.pvbus at (*);
        (0x001C090000 to 0x001C09ffff) to uart.pvbus at (*);
        (0x001C010000 to 0x001C01ffff) to sysregs.pvbus at (*);
        (0x001C0A0000 to 0x001C0Affff) to uart1.pvbus at (*);
        (0x002A490000 to 0x002A49ffff) to watchdog.pvbus_s at (*);
        (0x002A4A0000 to 0x002A4A0fff) to tzc.apbslave_s at (*)
    ]

    MEMORY_BUS overlays DRAM

    DRAM accepts [
        (0 bits 48)
    ]

}
```

8.2.2 Auxiliary File

```
{
    "connections": [
        {
            "moduleName": "Barebones",
            "source": {
                "name": "uart",
                "port": "serial_out"
            },
            "target": {
                "name": "terminal",
                "port": "serial"
            }
        },
        {
            "moduleName": "Barebones",
            "source": {
                "name": "uart1",
                "port": "serial_out"
            },
            "target": {
                "name": "terminal1",
                "port": "serial"
            }
        },
        {
            "moduleName": "Barebones",
            "source": {
                "name": "gic",
                "port": "redistributor_m",
                "address": "0"
```

```
            },
            "target": {
                "name": "core",
                "port": "gicv3_redistributor_s",
                "address": "0"
            }
        },
        {
            "moduleName": "Barebones",
            "source": {
                "name": "mci",
                "port": "mmc_m"
            },
            "target": {
                "name": "mmc",
                "port": "mmc"
            }
        },
        {
            "moduleName": "Barebones",
            "source": {
                "name": "mmc",
                "port": "card_present"
            },
            "target": {
                "name": "sysregs",
                "port": "mmc_card_present"
            }
        }
    ],
    "parameters": [
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "reg_base",
            "translation": "reg-base"
        },
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "reg_base_per_redistributor",
            "translation": "reg-base-per-redistributor",
            "value": "\"0.0.0.0=0x2f100000\""
        },
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "GICD_alias",
            "translation": "GICD-alias"
        },
```

```
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "gicv2_only",
            "translation": "gicv2-only"
        },
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "SPI_count",
            "translation": "SPI-count"
        },
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "ITS0_base",
            "translation": "ITS0-base"
        },
        {
            "moduleName": "Barebones",
            "component": "gic",
            "name": "ITS_TRANSLATE64R",
            "translation": "ITS-TRANSLATE64R"
        },
        {
            "moduleName": "Barebones",
            "component": "core",
            "name": "dcache_state_modelled",
            "translation": "dcache-state_modelled"
        },
        {
            "moduleName": "Barebones",
            "component": "core",
            "name": "icache_state_modelled",
            "translation": "icache-state_modelled"
        }
    ]
}
```

References

[1] Daniel Schwyn *Hardware Configuration With Dynamically-Queried Formal Models* http://www.barrelfish.org/publications/ma-schwyda-hwconf.pdf October 2017

[2] The Barrelfish Group *Sockeye in Barrelfish - Technical Note 025* http://www.barrelfish.org/publications/TN-025-Sockeye.pdf August 2017

[3] The Barrelfish Group *Mackerel User Guide - Technical Note 2* http://www.barrelfish.org/publications/TN-002-Mackerel.pdf May 2013

[4] ARM *ARM Server Base System Architecture 5.0 - Platform Design Document* https://static.docs.arm.com/den0029/50/Q1-DEN0029B_SBSA_5.0.pdf 2018

[5] *VHDL* http://www.vhdl.org

[6] *Verilog* https://ieeexplore.ieee.org/document/1620780/

[7] *System C* http://www.systemc.org/home/

[8] *gem5* http://gem5.org

[9] *ARM Fast Models* https://developer.arm.com/products/system-design/fast-models

[10] *ARM Fixed Virtual Platforms* https://developer.arm.com/products/system-design/fixed-virtual-platforms

[11] *LISA+* https://developer.arm.com/docs/101092/latest

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SYSTEM MODELING CO-DESIGN

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
KNOBLOCH	SVEN

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, SEPTEMBER 15th 2018	Sven Knobloch

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.