Co-Simulation of an Avionics Interface Device

FINAL REPORT

Team SDDEC21-02
Mathew Weber – Collins Aerospace
Dr. Phillip Jones
Spencer Davis
Matt Dwyer
Braedon Giblin
Cody Tomkins
Prince Tshombe
sddec21-02@iastate.edu
https://sddec21-02.sd.ece.iastate.edu

Created: December 1, 2021

Executive Summary

Custom hardware has become one of the crucial drivers in the great revolution of complex electronics products across many industries. While customized hardware, in the form of ASICs or FPGAs, allows corporations to create highly efficient hardware in terms of performance and energy efficiency, the design cycle tends to be very slow compared to traditional software implementations.

Much of this inefficiency comes from the need to develop and verify both hardware and firmware concurrently throughout the design process. However, without the hardware, the firmware is difficult to test and verify, and without the firmware, hardware is equally troublesome to implement. Recent verification developments in the hardware realm have drastically improved the speed at which RTL designs can be created and verified. Furthermore, modeling tools such as SystemC have allowed quickly implemented models that accurately model a potential RTL system.

However, only recently has there been a push to enable software verification without the backing hardware present. Co-simulation is a solution designed to allow a SystemC hardware model – implemented much quicker than an RTL design – to interact in real-time with a processor emulator running the system under test.

Our task was to explore the co-simulation system and expand it with documentation, demos, and real-time data manipulation, allowing the test system to be set up quicker, more accessible, and more feature-rich.

Development Standards & Practices Used

In this project, we utilize many practices related to open-source software. Our project hinges on us using open-source repositories and expanding and contributing to these such projects as well. Our team also utilized AGILE-like development, where we utilized a KANBAN style task board to keep track of our ongoing and defined tasks.

Summary of Requirements

1. Setup and execute a Cosim model using SystemC TLM backend and Xilinx QEMU processor simulator simultaneously
2. Expand the Cosim capabilities by implementing bi-directional memory communication
3. Model and test an off the shelf Linux driver for a memory-mapped peripheral
4. Generate documentation and demos that show the setup and usage of a co-simulation model

Applicable Courses from Iowa State University Curriculum

- CPR E 381 – Computer Organization and Assembly Level Programming

- CPR E 288 – Introduction to Embedded Systems
- CPR E 488 – Embedded Systems Design
- CPR E 308 – Principles of Operating Systems

New Skills/Knowledge acquired that was not taught in courses

Our team acquired new knowledge of hardware simulation platforms, including SystemC TLM and Xilinx QEMU. Furthermore, we gained experience with embedded Linux, using Buildroot to model our test system. Our team also explored Linux driver testing and learned how the driver may interface with a memory-mapped peripheral. Lastly, our team gained valuable insight into working with open-sourced project teams to progress a project beyond the scope of a traditional development undertaking.

Table of Contents

1 Introduction 5

1.1 Acknowledgement 5

1.2 Problem and Project Statement 5

1.3 Operational Environment 5

1.4 Requirements 5

1.5 Intended Users and Uses 6

1.6 Assumptions and Limitations 6

1.7 Expected End Product and Deliverables 6

2 Project Plan 6

2.1 Task Decomposition 6

2.2 Risks And Risk Management/Mitigation 7

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria 8

2.4 Project Tracking Procedures 9

2.6 Other Resource Requirements 9

2.7 Financial Requirements 10

3 Planned Design 10

3.1 Previous Work And Literature 10

3.2 Design Thinking 11

3.3 Proposed Design 11

3.4 Technology Considerations 12

3.5 Design Analysis 12

3.6 Development Process 13

3.7 Design Plan 13

4 Design Changes 14

4.1 IMU Alterations 14

4.2 Remote Port Alternatives 14

5 Implementation 15

5.1 Original Co-simulation Demo 15

5.2 Remote-Port Implementation 16

5.2.1 Communication Channels 16

5.2.1.1 File I/O 16

5.2.1.2 I/O Stream Pipes 16
5.2.1.3 Sockets 17
5.2.2 Protocols 17
5.2.2.1 UDP 17
5.2.2.2 Custom Protocol 17
5.2.2.3 Remote Port 18
5.3 ..... Industrial I/O Implementation (IIO) 18
6 Testing 19
7 Closing Material 20
7.1 Conclusion 20
7.2 References 20
8 Appendices 21
Appendix I: Operation Manual 21
IIO Demonstration Installation: 21
Prerequisites: 21
Expected Directory Format 21
Meta-repository – Configuration and setup 21
Building the demo 21
Running the demo 22
Exploring the simulation 22
IIO Subsystem Access 23
IIO Oscilloscope 23
Modifying the Demo 24
Simple Cosim Demo 24
Build setup 24
Run 25
Appendix II: Alternative Designs 25
Alternate Driver 25
Alternate Dynamic Data Streams 26
File IO: 26
UDP Custom Packet Protocol: 26
Sockets: 26

Appendix III: Other Considerations 27

1 Introduction

1.1 Acknowledgement

We want to give our client Matthew Weber a big thanks for providing us with the technologies we need and his technical help and patience throughout this project. We would also like to thank Dr. Phillip Jones for giving us technical advice and helping us solve problems we have had throughout this project. This project couldn't be done without them.

1.2 Problem and Project Statement

Problem: The Co-Simulation (co-sim) environment using Xilinx Quick Emulator (QEMU) in conjunction with Xilinx SystemC TLM libraries lacks good technical demonstrations and documentation.

Solution: This project aims to create demos and simulations that can be documented and used as examples for future software users. Our team's primary demo will be building will feature an arbitrary Linux driver running in QEMU simulation, with a SystemC backend capable of communicating bidirectionally with the host PC. An application hosted on the host PC can then interact with the SystemC backend, driving the backing registers of the Linux device.

1.3 Operational Environment

The project will be created in a Linux environment. We are using an Ubuntu 18.3.4 server. This version of Linux was selected based on prior co-sim demos, ensuring compatibility with all required tools. Our project will be simulating hardware, so we do not need special hardware for this design.

Our primary concern with our environment is maintaining correct versioning on all submodules. We can manage this by forking each repository and keeping our own "ground truth" versions that we know will work. Tracking versions that will adequately work in our toolchain is also critical to include in our documentation.

Co-sim is an open-source project, so we will be working with the project community to ask questions and get feedback from the project creators. Working with an open-sourced codebase means we must abide by their coding design and documentation standards.

1.4 Requirements

- Identify an off-the-shelf Linux driver for an I²C IMU device
- Bi-directional communication between the SystemC model and the host PC.
    - Communication must allow multiple devices to be modified by the front end
    - Must support read/writes to registers synchronized with QEMU accesses
    - The interface should be configurable and scalable
- Front end application to manage the host PCs connection with SystemC backend
- Documentation and demonstration of design, as well as a robust comparison between Cosim and previous simulation interfaces

1.5 Intended Users and Uses

Our intended users are corporations looking to utilize co-simulation for testing their products. For instance, the avionics community would be interested in these modeling chains to test software drivers before having novel avionics hardware designed and synthesized.

1.6 Assumptions and Limitations

We are assuming that:

- All source code will be published to an open-source repository
- We can freely use all SystemC libraries to model our communication of interfaces

Some of our limitations are:

1. Some group members have little experience using a Linux based operating system and need to learn much new material to be able to contribute
2. The amount of current documentation of the system process is relatively limited.
3. Our contributions and documentation will be constrained by what repository maintainers are interested in having in their projects.
4. As of right now, the co-simulation programs do not utilize a convenient user interface, which may make testing our new code difficult.

1.7 Expected End Product and Deliverables

Our end product consists of a set of demos and documentation. The documentation should be pushed to open-sourced projects. The demos consist of

- A baseline demo showing a ground truth setup of a co-simulation model using a Linux system to access a modeled real-time clock
- A more complex threading demo showing a SystemC state machine
- A demo showing external communication with a model via injecting dynamic data into the model at runtime
- A demo showing a custom Linux driver's interaction with the model

2 Project Plan

2.1 Task Decomposition

This project consists of multiple tasks. Below is a list of those overarching tasks and some of the intricacies involved in each:

- Initial Cosim demo and environment setup
    - Setup a shared computing environment for all members to use collaboratively.
    - Work through the initial demo provided by the client to learn the ropes of the tools at hand

    - Explore the technologies (SystemC, QEMU, TLM) and how they interact with one another in the simulated environment
- Modifying the Demo
    - Understanding how to modify the demo to add additional functionality or alter previous functionality of the timer register counter
    - Implement the Threading demo provided by our client to augment the initial Co-Simulation demo further
    - Better understand how all the software interacts and plan on how to add new demo features
    - Document the process for running the demo for addition to the repository through pull-request submission
- Reach out to public project maintainers about project direction
    - What additions would be welcomed by the development teams utilizing the same tools?
    - What other resources are available to aid in the contribution process?
    - What areas are most in need of support and extension?
- Implementing bi-directional communication of the host and the SystemC model
    - Develop a protocol for modifying any SystemC device data via communications from the host
    - Understand interactions between simulated hardware in SystemC, the simulated Linux software driver and OS in QEMU, and the input data from the host OS
    - Implement a front-end interface to be run on the host
    - Demonstrate controllability of the simulation via the host communication interface
- Identify and obtain a driver for an I^2C IMU and simulate the driver using co-simulation augmented by host-controlled communications.
    - Identify an open-source Linux driver for IMUs to use in a demo
    - Identify a common IMU for simulation
- Document the additional demonstration in detail
    - Record all steps to reproduce results from a beginner to intermediate experience level
    - Receive public feedback from the development community surrounding similar demos and the utilized tools.
    - Publish a final draft that is accepted for publication
    - Provide easy handles for other developers to extend the functionality of the demo and understand how to adapt it to their needs.

2.2 Risks And Risk Management/Mitigation

The overall risk for this project is relatively low. The risk is low because it is entirely in software development and utilizing demos already freely provided online. The most significant risk factor foreseen is the poor reception and feedback of our contributions to the public projects. This could occur for several reasons, such as poor maintainer support, unaligned goals for the project's future, or already generated documentation and additions.

We have worked with our client to develop another publication strategy to mitigate this risk if the primary public repositories do not favor our contributions. This would involve publishing our additional documentation and improvements on our own. Since each project is open-source,

meaning free to distribute and alter, there would be no licensing issues. While it would not be a part of the official documentation for the interacting projects, it would likely still contribute to the Co-Simulation development community as a whole.

In addition, when developing the low-level test drivers for the UDP communication protocol and other devices, our client has been generous in providing support from professionals in that area. As such, we will likely struggle at first to generate those low-level drivers, but with the consultant's help, most of those risks should be mitigated.

Finally, documentation for the tools we are utilizing and developing are significantly scarcer than other public tools are due to their limited use. Part of our project goals is to better the documentation provided for developers wishing to utilize these tools for co-simulation. However, we may run into roadblocks ourselves when trying to use some of these undocumented tools.

To mitigate the risk of unknown and undocumented tools, our team will be vigilant in documenting all tools and knowledge we gain along the way for our team and others. We will employ a fail-fast methodology of building prototypes and testing often. When we encounter poor results, we will be prepared to search for alternative solutions rather than waste time on a potentially poor solution. We also plan to be aggressive in outreach and support. We will seek guidance from our client and his team of experts, along with the development community of the tools at our disposal. We foresee a quick turn-around time if we need to pivot to a new tool with these resources.

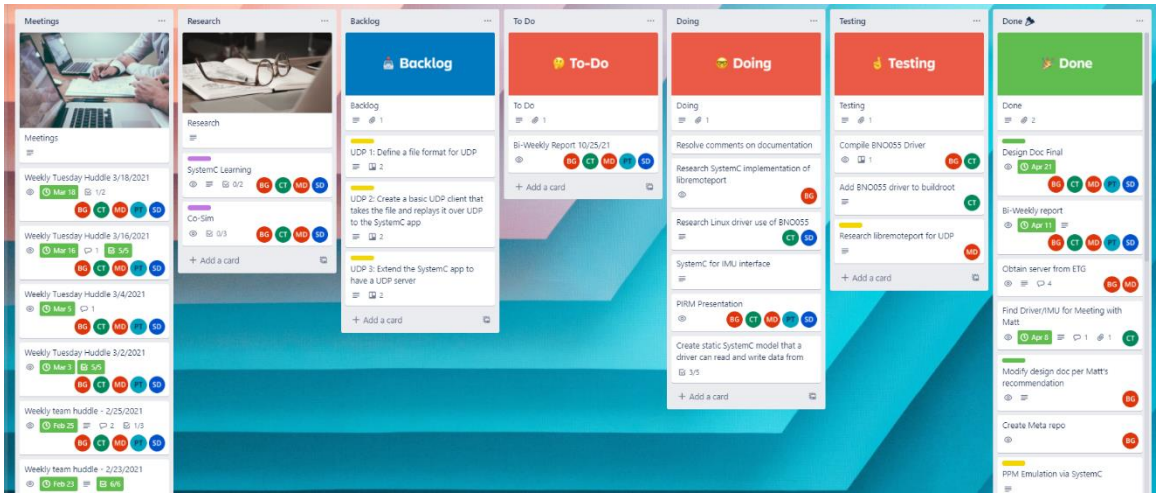
2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

- Initial Cosim demo and environment setup
    - Have everyone on the team complete the demo application
- Modifying the Demo
    - Make a pull request to the *Cosim-Demo* repo and get feedback
    - Have every member of the team understand and change the memory-mapped register data to the Real-Time Clock (RTC)
    - Make contact with the *Cosim-demo* repository managers to gain feedback on the additional features and documentation
    - Publish centralized startup documentation for the *Cosim-demo* repository
    - Create a list of milestones for implementing the threading framework described by the client.
- Implement a novel Linux device into our model, setting up a potential demo that would illustrate co-sim functionality
    - Configure Buildroot to include and set up any driver and user programs needed to start up the demo
    - Implement a hardware device in SystemC that will interface with the Linux driver and provide dynamic sample data
    - Document and describe the demo so that repository users in the future can replicate it.
- Implement working remote-port communication capabilities for bi-directional communication between TLM and Linux environment
    - Determine a candidate device for bi-directional memory-mapped simulation from a hardware and firmware side

    - Create a comprehensive document for adding new simulated devices from hardware (SystemC side) and software (Linux Buildroot in QEMU).
    - Develop a test application to aid in understanding both sides of the Linux socket protocol and custom data packets in use
- Public Contributions
    - Make documentation contributions to public Xilinx Co-Simulation repositories
    - Augment the initial demo application repo (*Cosim-demo*) to include additional bi-directional remote-port capabilities

2.4 Project Tracking Procedures

As our project progresses, we have used a Trello board to track our progress. This allows us to break each task out and assign it to our team members to complete one at a time. It also allows us to see a better bird's eye view of our project to gauge our progress and current successes, and other bottlenecks that might be occurring. We utilize Discord to communicate and share documentation, links, and other comments. This allows us to work collaboratively in that space and provide a single communication channel and resources for the project. We also utilize a shared Gitlab group to house the repositories we need to submit to the relevant repositories we plan to contribute to. These would also include our internal development code and documentation that we generate in the process. These tools will make up the primary sources of communication and collaboration for our development team during this project. A shared Google Drive folder is used to maintain all administrative documents, diagrams, and presentations among team members. Finally, when contributing to public code repositories hosted on Github, Github pull requests. Our team uses their associated comment systems to solicit feedback and suggestions from the open-source community.



2.6 Other Resource Requirements

For this project, a shared computing environment is needed for our team to develop the additions to the software described effectively. Since we are simulating complex processors in parallel and hardware devices attached to them, this requires a significant amount of computing resources. A powerful Linux server is needed to support these computing needs for our project. This is currently

being provided by the Department of Electrical and Computer Engineering and guaranteed until completing our project.

A team communication platform for weekly meetings is also required to communicate. We chose Discord as our preferred platform, as it is free and easy to use. This allows us to work remotely, hold meetings, and share information in real-time with one another when we cannot meet in person. Code repository and hosting services are again provided by the Department of Electrical and Computer Engineering. At the same time, other documentation and demos remain public or provided by our client and his team. Our project aims to utilize the current public resources and guidance to produce additional documentation, tools, and resources for other developers hoping to use the robust Co-Simulation framework provided by Xilinx with QEMU + SystemC-TLM structure. All deliverables aim to be open-sourced and accessible for anyone to use and readily available to make sense in public development channels.

2.7 Financial Requirements

This project did not require any financial resources beyond the services provided to us by the Iowa State Department of Electrical and Computer Engineering. We used a virtual machine hosted for us and did not require any hardware or special tools. Our project required no travel either for any of our team members.

3 Planned Design

3.1 Previous Work And Literature

Various simulation technologies exist for simulating both processor behavior and respective simulation environments individually/separately. However, the co-sim model combines the two. Though this technology exists, there lacks sufficient documentation and demonstrations.

In essence, the co-sim model as a toolchain is relatively new. Therefore, improving documentation and demos will be a significant focus of this project to make the technology more approachable to prospective users.

The majority of project work will be focused on extending the usefulness of an already existing simulation environment. This means that background research is somewhat limited in its scope to learning about the technologies already being used by the system. The project group is currently focused on learning about those technologies.

Background literature for this project includes SystemC tutorials, a Xilinx emulator user guide, co-simulation documents, and any other work found on the open-source forums.

Literature:

Banerjee, Amal, and Balmiki Sur. “SystemC-AMS and SystemC Combinations.” *SystemC and SystemC-AMS in Practice*, 2013, pp. 449–455., doi:10.1007/978-3-319-01147-9_17.

Ammari, Ahmed Chiheb, et al. “HW/SW Co-design for Dates Classification on Xilinx Zynq SoC.” *2020 26th Conference of Open Innovations Association (FRUCT)*, 2020, doi:10.23919/fruct48808.2020.9087548.

Xilinx. “Xilinx Quick Emulator User Guide.” 2019.

3.2 Design Thinking

Co-sim technologies exist yet are not well known within the target community. A relatively new tech, improving documentation and demos will make co-simulation tech more approachable for the community and hopefully allow for increased usage of these technologies among target constituents.

Our initial design thought to improve co-sim’s documentation and demonstration capabilities was to create a set of highly general examples that would serve as demonstrations. However, we decided to better serve the community by implementing a more specific, feature-rich example.

With this in mind, we settled on implementing an I^2C device that could be modified via both our QEMU simulated driver and a separate Host controlled remote port stream. This example is beneficial to the community because it represents a real-world use case where a Linux driver may need to be tested against a complex simulated set of hardware.

Finally, we can address much of the project’s requirements by improving documentation. Clear documentation is critical to a new user of the technology understanding how each of the project parts works in tandem.

3.3 Proposed Design

The primary goal of our project is to introduce the co-simulation technology better and decrease the learning threshold required for end-users to use the technology in their development workflow. Our design begins with documentation. This satisfies the primary non-functional requirements of our project in making a new user understand the value of co-sim while providing insight on how to initialize a co-sim environment.

Our documentation will describe, in detail, each component of the simulation interface and how that component interacts with the model as a whole. This crucial layer of visibility will allow someone evaluating the technology insight into how their use case may fit into the model.

Our primary functional requirement of a novel demonstration application that an evaluator can run and experiment with will be implemented via an IMU setup. We selected an IMU because it is a complex subsystem device commonly found in various applications, ranging from automotive, aviation, mobile, and more. We will demonstrate how our simulation model can take an off-the-shelf Linux driver and run it on our system via a host application while having the entire model system functionally indistinguishable from a real system with an IMU present despite the entirety of the IMU being modeled.

Furthermore, our design will control the backing IMU via a front-end interface that runs on the Host PC. The front-end interface will interact with the SystemC model via a remote port. The Remote Port connection, defined in the Xilinx SoC libraries, will enable our front-end application

to directly interact with the SystemC model to modify state machine values, such as sensor readings, on the fly.

Our design will allow an evaluator to quickly view a demonstration of a Linux driver operating seamlessly with a modeled IMU, with the IMU accurately reflecting changing sensor values over time as driven by a host application. This demo will provide a company insight into the value of a robust simulated hardware model and how driver debugging can be done without hardware ever having been developed.

3.4 Technology Considerations

Most technology decisions regarding this project have already been made due to the nature of the proposal by our client. This is because the specific use case and environment have already been described. Our primary goal is to better document the toolchain and develop additional channels between the host and the SystemC model to aid the development of the Co-Simulation process using existing infrastructure. The co-simulation model offers increased flexibility compared to a "real world + simulation" model. This removed the need for physical hardware devices since all hardware and software are simulated in the software technologies.

For this project, we are using QEMU for the ARM processor simulation. We are using device description files for a Zynq 7000 System. However, we are not constrained by the particular board we are targeting and intend for our work to be general enough so that anyone can replicate it targeting a different board. We are modeling the Programmable Logic (PL) section of our FPGA using SystemC. This design decision was made for us by previous co-simulation work.

Ubuntu Linux was chosen as our host OS as all of our tools readily support it. Running our software environment on a Ubuntu platform allows us to get support from ETG and easily share our team members' environments.

Finally, we chose Buildroot to generate our embedded Linux image run on our simulated processor. We selected Buildroot because it is straightforward to get an image built and running, which is widely supported. Other possible options would have been Xilinx PetaLinux and Yocto; however, as the Xilinx repository maintainers offered no objection to us not using Xilinx PetaLinux, we elected to go with the simplest solution.

3.5 Design Analysis

We analyzed our design on two merits: how valuable our contributions to this technology will be for future development teams interested in using co-simulation and how successful our implementation will be. At the moment, we are highly confident that our design will fulfill its purpose of providing new support materials for evaluators. Our client has evaluated and approved our strategy, who shares our optimism about our design.

Our current progress on our design has made our team confident that it is achievable in the specified time frame. We have implemented the basis of a remote port into the SystemC model, and we have begun work on integrating the IMU into our model. Combining each of these components will be the topic of our next semester.

3.6 Development Process

Though this project doesn't fit into any "specific" development process, it most closely resembles the Agile approach because it is completed through small, iterative progress chunks with frequent feedback and demonstrations. Our group has chosen to use Trello to track progress.

This development process was selected because it allows for high client involvement and is readily applicable to the system when other development processes logically make less sense.

3.7 Design Plan

The project focus for this semester will mainly center around improving demos and documentation to improve the approachability of co-sim technology. Next semester, the project focuses on adding increased functionality to the system by extending our interface and implementing front-facing interaction methods.

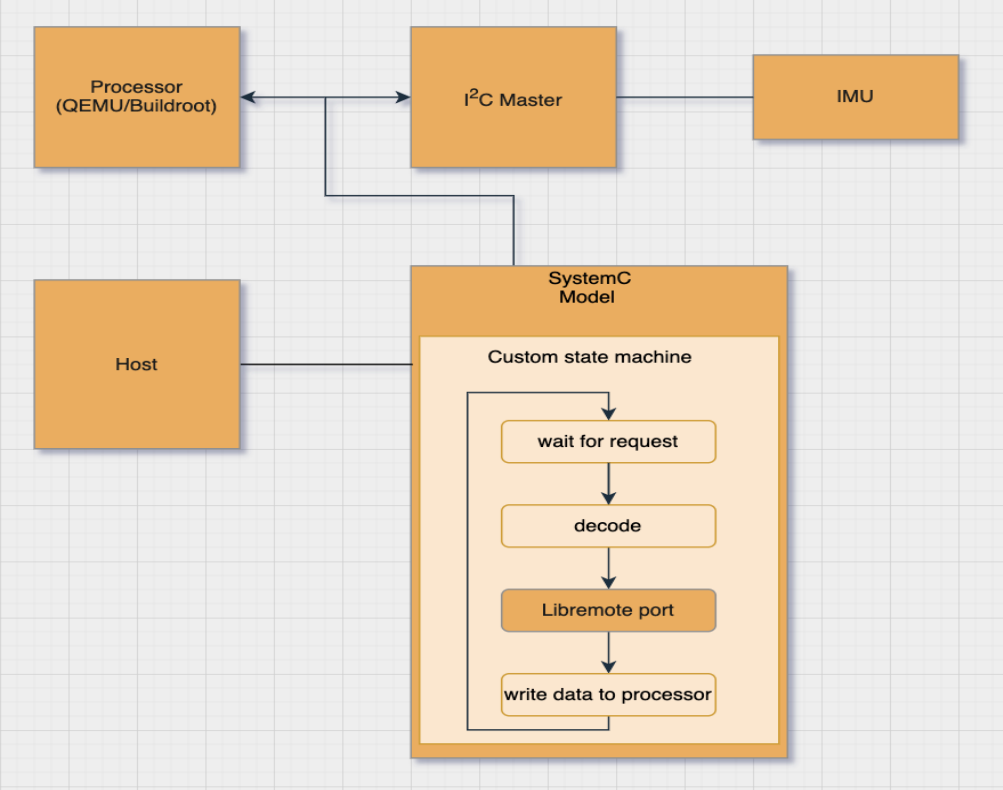


Figure 1: Design Diagram

Figure 1 above shows a high-level implementation of our design. The Host machine runs a QEMU simulation of our ARM processor on the Xilinx Zynq-7000 SoC with a custom compiled version of Buildroot. A Xilinx remote port connects to the SystemC TLM instance containing our I²C bus to the BNO055 IMU emulated device run atop our SystemC server device. This communicates by means of another Xilinx remote port to the host application used to control our SystemC server from outside our simulated environment.

4 Design Changes

4.1 IMU Alterations

Our original design used the Bosch BNO055 IMU as our modeled device. The intention was to model the IMU I²C interface via SystemC and then access the IMU via the generic driver provided with the device. However, we ran into many issues with this approach. Getting the generic driver installed into our system ended up being difficult, though we did eventually get the driver up and running. However, we learned that the driver did not provide any hardware interface and left that decision up to the system designer. Our group explored several ways to implement this hardware interface: via a character driver, mmap, and direct memory reads. However, each of these methods seemed clunky and seemed to detract from the intention of the demo: showing a Linux driver interacting with the SystemC.

We elected to change our design to address the hardware interfacing problems with our IMU by instead relying on the Industrial IO (IIO) subsystem of Linux to handle the hardware mapping to user space. Enabling IIO involved modifying our Buildroot setup to install an IIO daemon that would handle IIO interactions and set up a custom IIO device and trigger that would regularly sample our IIO device and forward the data through the IIO daemon to user space.

The IIO system provided several benefits to our original design. It provided a reasonable way to map IMU data from a device into user space easily and subsequently off the model by enabling networking and creating an SSH port forwarding tunnel to our host machine. Using this process, we eventually visualized our data using IIO tools. The IIO system also provided us with an excellent space to begin the implementation of our hardware interaction. As sampling and forwarding the data was all taken care of for us (after configuring and enabling the appropriate triggers and data buffers), we had to point the system towards our memory regions that represented our device and provide a reasonable data structure store the data in.

Our goal with this demo was to provide a future co-simulation user with a starting point to simulate their own devices. We determined that this IIO functionality was more helpful than our original design as IIO is generalized. We also demonstrated end-to-end communication through our model, including pulling data off our simulation as it ran to visualize or process on a host machine.

4.2 Remote Port Alternatives

One of the critical aspects of our project was to implement means of communication from the host system into the model, enabling the model to be dynamic at run time as opposed to statically defined at compile time. Xilinx employees managing the co-sim project pointed our team toward libremoteport, a socket system for the SystemC models that allowed bi-directional dataflow between the model and a host. However, we ran into many issues with libremoteport along the way.

Libremoteport was not a complete, polished library. It was in active development as part of the co-simulation initiative by Xilinx and lacked documentation and clear usage instructions. As such, our team struggled to get bi-directional communication between our host program and the model. Without this communication, the dynamic aspect of our project seemed unobtainable.

Our team attempted to mitigate this risk by exploring alternate solutions. We looked into using file IO to allow the SystemC model to read data from files; however, this did not seem to meet the requirements we provided, and there was no clear path towards implementing a file IO system within SystemC. We similarly examined pipes and came to the same conclusion: there was no simple way to dynamically pipe data into the model in the bidirectional sense that we needed.

Examining the requirements further, it was clear that even libremoteport didn't satisfy the design requirements precisely, as the data port for the model we were asked to implement was supposed to be entirely decoupled from the execution of the model. In other words, our requirements indicated that the remote data server should be able to arbitrarily connect and disconnect from the model, modifying data fields. Libremoteport offered a bi-directional means of modifying data, though it required that the socket connection be initialized and locked in at startup. So, the dynamic port would have been tightly coupled with the model, requiring all three interfaces (the SystemC model, the QEMU model, and the dynamic data server) to be concurrently started up and executed together.

Our solution to this problem was to allow the dynamic data server to offer a port of its own that would allow arbitrary connections. This would allow systems to arbitrarily connect and disconnect from the server and read and write data, while the actual libremoteport server would remain constant and connected. However, as we could never get libremoteport to exchange data with the models successfully, we were unable to implement and test this design additional. However, future teams working on the project could implement this feature to enable the data port to be decoupled.

However, a fixed data server in the model was still not ideal, as the original model would become more complex to run and set up. An ideal solution would be to design an entirely new socket system for SystemC that would have the capabilities we needed. This, however, did not seem possible for us to implement with the time and resources we had, so we were unable to explore this idea. More information about this work can be found in the implementation section.

5 Implementation

5.1 Original Co-simulation Demo

One of the big goals of our project was to create better documentation so newer users would have a baseline to start working with Xilinx's co-simulation. We did this in two ways: thoroughly documenting an already existing Xilinx demo and creating our demo with its own thorough documentation.

The first thing we did was create a tutorial on downloading, compiling, and running Xilinx's main baseline demo. We first gathered the bash scripts necessary to get the demo running. This demo starts by creating an instance of QEMU to host the Zynq-7000 processor needed to connect to the SystemC models. Once QEMU and SystemC are connected, the user can read register values. In this environment, the user is reading the value of the system clock over time. Each time the user reads the value using the command "devmem 0x40000000" the register value will be displayed. This shows QEMU hosting simulated firmware connecting to the simulated hardware of SystemC.

If the user follows the bash script, all downloading and compilation should be done for them. We then created a markdown file containing these bash scripts and information on how things work. This file can be found in Appendix 1. This gives the user a better understanding of how each part of co-simulation works together. We opened a git ticket with the Xilinx open-source community containing this file. After some recommendations from the maintainers at Xilinx, we made revisions and got the documentation pushed to the repository.

We also created documentation to support a demo involving PPM, or Pulse Position Modulation, state machine. PPM is a way in which a digital signal can be modulated to take on an analog value (similar to PWM). We implemented PPM as a state machine using accurate time devices. We then took a bare metal PPM driver and executed it on QEMU to demonstrate successful reads and writes of each channel. This proof of concept showed how to test software without entirely written VHDL logic.

5.2 Remote-Port Implementation

A primary challenge of our design goals was how to send transactional messages between our different simulated systems. While simple, in a complete system, you have multiple different types of transactions that can occur. These include but are not limited to reading data from another device, writing data to another device, serving read requests, serving write requests, synchronization of the simulated environments, and initial handshakes between devices. While our basic implementation did not implement all of those components, a full implementation would require all of those seamless transactions to occur between devices.

When considering these challenges, various solutions presented themselves. Sockets seemed like a logical solution to the inter-program communication that needed to occur. While files or pipes (command line) could be possible, each presented significant drawbacks. After initial research and use-case example testing, sockets were a clear front runner for our chosen communication medium.

5.2.1 Communication Channels

5.2.1.1 File I/O

Files had the issue of being a complete resource that would not provide bi-directional communication seamlessly. File I/O operations also can incur significant latencies, as data needs to be written to and from the file system. Files also imply non-volatile and catchable data that would not be necessary for our implementations. While one could argue that everything in Linux is treated as a file, and by using sockets, it is similar in concept, the additions that sockets afforded would be necessary for our later implementation.

5.2.1.2 I/O Stream Pipes

Pipes also presented challenges, such as locking command line input and requiring closely coupled simulation execution. Building on top of the SystemC framework, overhauling the command line would not be a gentle undertaking. Custom logging backends allow variable verbosity levels that manage the application's input/output streams. As such, even if our programs were to implement their iostream communication protocol, this would have likely been a messy and variable method. Additionally, running the simulated environments in such a closely coupled manner requires an overly complex application execution structure and makes for a clunky experience.

5.2.1.3 Sockets

Sockets were the answer to our communication woes. Allowing for bi-directional communication and management allowed for seamless binding and listening for transmitted messages. Sockets come in several flavors: Unix sockets, UDP sockets, TCP sockets. While all would have been acceptable candidate sockets, we used Unix sockets, as our data would remain on our host computer. Ultimately, this choice was unrelated to the performance of our applications, as the libremoteport library supports all three versions. We also had access to examples in which QEMU would utilize the Unix sockets for communication with the SystemC device. This made the modification of SystemC to accept an additional socket a semi-trivial task.

5.2.2 Protocols

The next decision was which protocol to implement over our chosen communication method. Here, we explored multiple formats for sending the bi-directional packets required. These included: specialized UDP packets, custom structures, and remoteport. While remoteport was ultimately chosen as the protocol, we would devote our development effort to all to exhaustively explore possible implementations.

5.2.2.1 UDP

UDP was another protocol seriously considered for passing data between the SystemC and host application. Some initial work was done to implement this method, with an altered UDP packet structure and remoteport packets. This work can be seen in Appendix II: Alternative Implementations, where some preliminary diagrams and research is included. In retrospect, this method would have likely cut down on development time and challenges. While this would have accelerated our path to a demo, it would have been severely lacking in features that would have been reimplemented. This was a factor in using the remoteport library over a custom UDP packet structure.

5.2.2.2 Custom Protocol

Various custom protocols were imagined when deciding on one to implement between our SystemC device and the host application. While a custom protocol provided a lot of flexibility and control, ultimately, it would prove to likely require a significant amount of engineering effort to implement all that we would need entirely. Basic protocols were tested to allow for reads and writes to the machine, but these proved rudimentary. This coincided with the similar professional implementation of remote port providing similar functionality. Deciding against reinventing the wheel, we choose to avoid forging this path ultimately, although some of our early demo prototypes did implement primary custom data packets in this manner.

In retrospect, this path may have provided more benefits than initially thought. While the work to do so would have been non-trivial, the engineering effort to operate and develop a functional remote port implementation proved far greater than initially thought. In retrospect, a basic proof-of-concept (POC) with a custom protocol would have likely been faster, and within our project scope, we choose to work with the same protocol implemented by QEMU and in the libremoteport library, as it would likely be better suited for publication and usage by others.

5.2.2.3 Remote Port

Xilinx originally developed the remote port protocol during a DARPA Co-Simulation project for communication between simulation environments. Although the libraries and QEMU implementations are open-sourced, not much public documentation exists. This protocol was recommended by the open-source developers at Xilinx, as it was the protocol utilized for the connection with QEMU. For a good reason, it was our choice or protocol. Not only did it already have support for all of the data movement, synchronization, and configuration data we required, but it also had public decoders and the libremoteport library supporting it. This made it the prime candidate for our development, despite lacking documentation and public demos. This would prove to be one of our most significant hurdles in the project development, as understanding the protocol and implementing our backend processor and handler was a larger undertaking than initially expected. While we could reuse significant portions of the library to decode and transcode data into the correct packet forms, sending this data and interpreting it proved challenging. Debugging was helpful in some cases, but with the system of 3 distinct simulation environments, replication and isolating our bugs was a challenge.

The code and documentation for our remote port implementation can be found in our associated code repositories. It includes a QEMU Buildroot implementation that connects to a basic SystemC model, creating a shared memory device and the connections for the QEMU SoC and the host application. The host application creates a socket, and SystemC connects to QEMU and our host application. They all communicate over Unix sockets with the remoteport protocol and, in theory, can transfer data back and forth in any direction from any of the clients. Writing or reading data in the memory-mapped ranges causes the operations to affect the memory device, which is mapped according to both QEMU and the host application.

We were unable to realize the entirety of this demo implementation ultimately. While we could complete the QEMU and Buidlroot device implementation and the shared memory model, our host application still lacks some functionality. It can create our connection socket and communicate with SystemC over the initial hello handshakes. It can read any incoming data packets from the SystemC model and attempt to write data packets to the bus. While this does not cause any errors in our system, we were unable to resolve the issue of the values not propagating and committing to the shared memory structure. There appears to be some area where our reads and writes are failing to take effect, but it is still unclear at this time. Given another week of development, our team is confident we would have been able to complete this demo.

5.3 Industrial I/O Implementation (IIO)

For our IIO implementation and demos, we created custom Linux Kernel configurations, Busybox configurations, and Buildroot configuration files for enabling IIO functionality, including network support between devices. The default driver example code was then modified to support allocating and reading from mapped IO memory. While initially, our research suggested that the Memmap tool would allow us to create this memory link and bypass any virtual address space issues, our client advised us that this in fact, would not be an ideal path to pursue. Thus, we were able to support this functionality at the kernel level instead. Our IIO sample configuration was also modified to provide a reasonable launch point for integrating with SystemC model. This integration was crucial, as without it we would have required an I^2C device master implementation which would have created additional unnecessary scope in our project. Our goal was to provide a simple

connection between the SystemC device simulated environment and the memory-mapped IIO device using the path of least resistance.

A Buildroot external configuration directory was also designed to allow all Buildroot materials, including kernel patches, post-install setup scripts, packages, configurations, DTS files, and more, to be included out of tree and automatically integrated. This streamlined the integration and demo process, as users can utilize the automated configuration scripts and files to configure the complex Linux systems and tools required. The IIO subsystem can then be accessed via libbiio over an SSH port, enabling IIO data to be tunneled off the simulation and processed elsewhere. This was done by reading a URI handle and plotting the values in a graphical oscilloscope tool developed by Analog Devices. We created a module that holds static values in registers for the SystemC implementation. We also created a circular buffer that provides a pseudo-random value every time it is read via a thread and a simple mathematical implementation of a sine wave. These basic data response functions provide dynamic dummy data, which could be extended to realistic and meaningful implementations.

6 Testing

For our original documentation, our testing came from running our bash scripts in clean environments to test the user experience and functionality from a new user's perspective. After running all the commands, we could get the demo up and running as expected. We also reached out to the open-source community and our peers for feedback on our documentation. This provided us with many suggestions for improvement, significantly improving the quality and support for our documentation. This methodology stemmed from the assumption that most users will be running our demos in novel environments, as they are targeted at beginners to the Co-Simulation landscape. During our monthly project presentations, we also gained feedback on the overall accessibility and understanding-at-a-glance, as "curb appeal" is essential when sifting through technologies, attempting to determine where to allocate your time when beginners are looking for possible solutions.

For each demo we authored, manual tests were performed in systematic manors to test the functionality of reading and writing the expected data. The *devmem* command proved helpful in allowing for this direct memory access from the high-level Buildroot shell, making this testing process streamlined and quick. It also helped allow new users to test the functionality of the demos and ensure any additions they may make are also seen correctly in the system.

For the Xilinx demo documentation, an open-source repository maintainer provided vital feedback and assisted us in testing our documentation, and gave us feedback in a clean cloud environment. This provided a unique perspective to our documentation and ensured that our guide would cover a wide range of environments that users may attempt to deploy it in. It is also essential that we collaborate with the open-source community on these additions. Since our project is open source in nature, ensuring that the repository maintainers and other users can replicate their demo with our documentation and ensure we abide by their repository standards proves that our work is acceptable and up to their high standards. Once our pull request was accepted and merged, we considered that a successful implementation.

We replicated this process for all of the documentation and demos we crafted. While not all of it fit well in the public repositories maintained by Xilinx, we still made efforts to receive feedback from them and others, including our client, about our documentation. What was not published to the Xilinx repositories we have published in the form of multiple organized projects in Gitlab. These have been included as part of our final body of work on our website.

7 Closing Material

7.1 Conclusion

Co-simulation is a compelling tool developers can use to test their hardware without needing an actual prototype. The additions we have made to the Xilinx repository should be helpful for new and existing users of their co-simulation repository. Although we were unable to finish all tasks given to us by our client, we believe we have improved the Xilinx repository by adding documentation for their example demonstrations and their libremoteport repository. The main demo we have created highlights a practical use case for a new user of co-simulation.

We hope that users unfamiliar with how co-simulation works will have a more accessible time diving in. Our group has already had users like this reaching out for assistance. Within the first two weeks of the publication of our SystemC TLM Cosim Demo and documentation, a Ph.D. student from the University of Florida reached out to a group member asking for assistance with co-simulation technologies and about our tutorial specifically. This helped us identify shortcomings within our documentation, and it helped show that our project has already begun to have a positive impact on the target user base.

We are confident that we have tied off the incomplete pieces of our project well enough that a future developer will be able to pick it up where we left off. We have provided good documentation of our previous problems and given them an idea of what we were attempting to do. If we one day see our implementations completed by another developer, we can safely say our contributions were impactful and vital.

7.2 References

J. Komlodi and V. Garhwal, “Co-simulation,” *Confluence*. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/862421112/Co-simulation. [Accessed: 09-Mar-2021].

Xilinx, “systemctlm-cosim-demo,” *GitHub*. [Online]. Available: https://github.com/Xilinx/systemctlm-cosim-demo. [Accessed: 09-Mar-2021].

8 Appendices

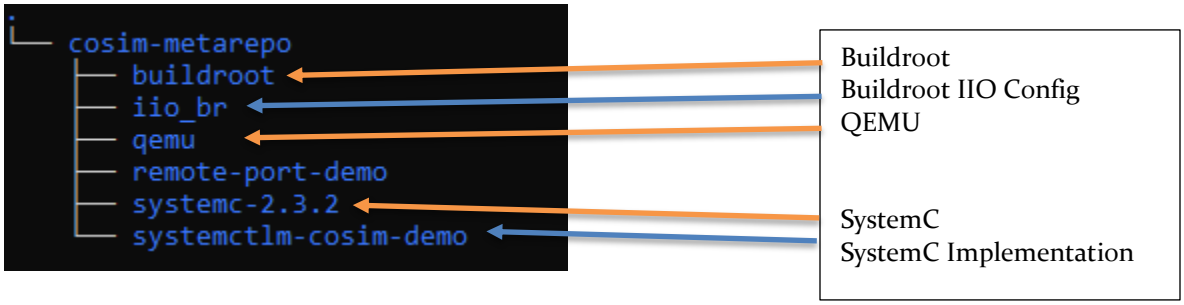
Appendix I: Operation Manual

IIO Demonstration Installation:

Prerequisites:

- SystemC Version 2.3.2 installed
- QEMU (Xilinx-v2020.2) Installed
- Buildroot installed

Expected Directory Format



Meta-repository – Configuration and setup

As a convenience, a single repository has been provided that provides the correct directory structure and dependency versions as submodules, along with scripts and makefiles that facilitate installing depencies, building the demo, and running the demo.

First, the repository must be clone with submodules initializes recursively

```
>> git clone --recurse-submodules
            https://git.ece.iastate.edu/sddec21-02/cosim-metarepo.git
```

After the repository has finished downloading all submodules, enter the directory and start the dependency installation

```
>> cd cosim-metarepo
>> sudo make install
```

This phase of the installation will configure each dependency to use the appropriate build settings and paths, and then build and install each dependency. This process may take some time and will result in QEMU and SystemC being installed on the host machine.

Building the demo

After all dependencies have been built, it is time to build the demo. The can be accomplished with a call to make

```
>> make
```

This make call will first initialize a kernel config and Buildroot config for Buildroot and then begin the process of building and patching the kernel in to the required configuration. Upon conclusion, a root file system will be generated that includes all needed Linux components, libraries, and kernel source modifications for the demo to run. All Buildroot configurations are out-of-tree, with a custom configuration file, kernel patches, and custom board support packages.

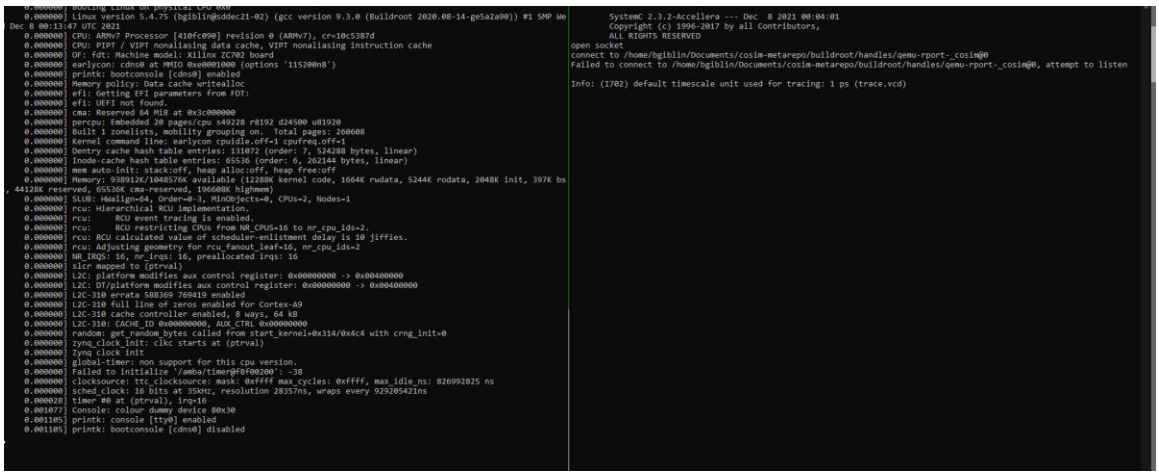
Next, the SystemC model will be built. The SystemC model contains a backend for the IIO interface and defines a set of memory-mapped registers that may be accessed over the shared data bus between the SystemC simulation and QEMU.

Running the demo

Finally, the demo may be run by simply calling the start_cosim script:

```
>> ./start_cosim
```

This script will start a TMUX instance, with two screens split side by side. The screen on the left represents QEMU, and you should begin seeing it boot up Linux.



The window on the right side is the debug console for SystemC. While booting up, you will not see any activity. However, the info line present indicates that it has bound to the socket and is ready to exchange data.

Once the Linux build has finished booting, you may log in using the login “root”.

Exploring the simulation

At this stage, you may begin exploring some of the features of the simulation. As a pre-boot task, networking and IIO daemons have been brought up. A first experiment would be to test the memory interface provided by our SystemC model. Our memory interfaces only support 16 bit, word aligned reads in the range `0x40000800 – 0x40000806`. As shown below, a read may be done with the “devmem” command.

```
>> devmem 0x40000800 16 # Perform a 16 bit read at 0x40000800
```

```
# devmem 0x40000800 16
0x0000
# devmem 0x40000800 16
0x061F
# devmem 0x40000800 16
0x0B4F
# devmem 0x40000800 16
0x0EC7
# devmem 0x40000800 16
0x0FFF
#
```

Figure 2: Several memory reads of register 0

```
Info: IIO: READ: Register 0x00

Info: IIO: READ: Register 0x00

Info: IIO: READ: Register 0x00

Info: IIO: READ: Register 0x00

Info: IIO: READ: Register 0x00
```

Figure 3: SystemC log of Memory Reads

IIO Subsystem Access

We can now begin accessing the IIO subsystem. The IIO subsystem provides several tools for examining an IIO device.

```
>> iio_info
```

This command prints out all of the attached IIO devices and their channels. You should see 2 IIO devices, our dummy device, and an ADC, as well as a trigger device. We can now verify the functionality of our trigger, the IIO daemon, and our memory-mapped registers.

```
>> iio_readdev -u ip:localhost -t trigger0 -s 8 -b 8 iio:device1
voltage0 | hexdump
```

Using a software-defined trigger, this will complete a read of the device's voltage channel, using 8 samples and 8 buffered reads.

```
# iio_readdev -u ip:localhost -t trigger0 -s 8 -b 8 iio:device1 voltage0 | hexdu
mp
0000000 0ec7 0b4f 061f 0000 f9e1 f4b1 f139 f001
0000010
#
[SystemC-C0:bash*
```

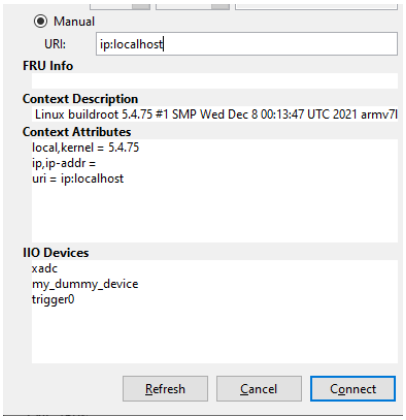
As we can see, the command successfully read several data values. Furthermore, you should notice a flurry of register access on our SystemC model.

IIO Oscilloscope

To view the IIO data on a real-time oscilloscope, first install the IIO oscilloscope tool from Analog devices on a host machine supporting GUIs. The oscilloscope can be installed on any machine, provided there exists a network connection between the simulation host and the desired GUI host. Next, open an SSH port forward tunnel from the QEMU simulation to your machine running the oscilloscope.

```
>> ssh -R 30431:localhost:30431 -N username@hostpc
```

Now, on your Oscilloscope host, you may start the oscilloscope, select “Manual”, and fill in “ip:localhost” as the URI:



Pressing “Enter”, you should see the context and IIO devices fields populate with appropriate information. Clicking connect will open up an IIO debug window and an oscilloscope. Add the channels you wish to the plot and set the sample count to be relatively small, about 20.

You may now run the oscilloscope in either step mode or continuously, and you should observe the 4 data channels plotted on the oscilloscope.

Modifying the Demo

You may expand the demo by modifying the SystemC behavior of the registers, changing the channel configuration, or adjusting the IIO low-level driver behavior. To change the SystemC model, simply modify “systemctlm-cosim-demo/iio.cc”. You may also freely add any additional files, so long as you add them to the makefile by appending them to the SC_OBJS variable. Once finished, simply call “Make” from either the main repository root or the systemctlm-cosim-demo directory.

Buildroot may be modified in several ways as well. You may choose to add packages to Buildroot, by calling

```
>> make menuconfig
```

from inside of the Buildroot directory. Furthermore, you may modify the kernel further by applying the patches located in iio_br/linux/patches, making more modifications, and then recreating and replacing the patch file.

```
>> patch -ru linux-5.4.75.orig -i linux-5.4.75-01-something.patch

>> diff -ruN linux-5.4.75.orig/ linux-5.4.75/ > linux-5.4.75-01-
something.patch
```

Once you are done, call utils/brmake from inside of the Buildroot directory to re-build the Linux system or call make from the main repository root.

Simple Cosim Demo

Build setup

This demo uses SystemC version 2.3.2. The makefile requires that both are installed into the following directory: SystemC: /usr/local/systemc-2.3.2/. For systems with these files installed to different directories, make the Makefile point to the correct directory by setting the variables SYSTEMC. After this, you will need to clone the libremote-port submodule by running the following command: $ git submodule update –-init libsystemctlm-soc. After this command,

everything required for the demo will have been installed. The system can then be built by running make. If SystemC is in a separate directory from the one mentioned above, you will need to specify the directory by setting the variables mentioned above. Configuration can also be done by creating a .config.mk file. The following options can be set: HAVE_VERILOG=n, HAVE_VERILOG_VERILATOR=n, HAVE_VERILOG_VCS=n.

Run

When running the system, it is imperative that the program can link to your SystemC/TLM libraries. Arguments must also be given to the application. The first argument points to the QEMU machine-path to use, while the second is the icount value to use. These values should line up with the QEMU command line arguments.

Our project demonstration utilizes a Zynq-7000 machine, though others are available. For our processor, in one terminal, in the demo directory: LD_LIBRARY_PATH=/usr/local/systemc-2.3.2/lib-linux64/ ./zynq_demo \ unix:./qemu-tmp/qemu-rport-_cosim@0 1000000

In another terminal the PS needs to be started. In our projects case this involves starting up a PetaLinux QEMU session and use the Linux kernel to probe the SystemC side. Another option is to start your own kernel with the necessary drivers or a bare-metal application.

For instructions on how to start the PetaLinux QEMU session, please see the following link: http://www.wiki.xilinx.com/Co-simulation

APPENDIX II: ALTERNATIVE DESIGNS

Alternate Driver

One of the primary alternate designs we considered was using a different driver than IIO. We elected to go with IIO because the kernel provides a dummy driver that has most of the business logic and front end already implemented, and has a full suite of tools that allow for accessing the various facets of the IIO system. We also found it hugely benifical that an out of the box daemon was provided that could serve IIO data over the network, making it easy to take data off of our model.

We do regret not being able to implement a complete I²C master device. This was our original goal, and would have enabled us to simulate a more realistic example where we are talking to a device over a bus. However, the master ended up being too complicated for us to model accurately in the amount of time we had, so we were forced to abandon this idea. However, we do still think a good next step for this project would be to add the ability to emulate real bus accesses from the driver instead of using mapped IO memory

In the end, we feel that even if we got I²C to work in our model, it would not have changed our demo as we likely still would have used the IIO front end. This is partially the reason why IIO was so attractive to us; it provided a huge amount of flexibility in the backend by simply providing a buffered reader that we needed to implement.

Alternate Dynamic Data Streams

Our team investigated other data streams that could have been used to dynamically change our model on the fly. Some streams we looked at included file IO, UDP sockets, and Berkely sockets. We ended up dismissing each of these for various reasons.

File IO:

File IO had the advantage of being simple, however it lacked the versatility we were hoping for. While we could easily write lists of data into files and have SystemC models read that data in, it would not allow on the fly data adjustments, bi-directionally, that we were aiming for. Furthermore, it didn't allow the data provider to be arbitrarily connected and disconnected while the model was being run.

UDP Custom Packet Protocol:

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Address																															
4	32	Length																Read/Write															

Initially we had begun work on a custom UDP based protocol that would be used for operating between the Host application and the SystemC model. Above you can see some of the initial diagrams used when planning out the data field ordering and alignment. While this was only ever a theoretical implementation, it was an initial phase in the exploration of protocols for our implementation. This basic design was created by starting with a base UDP packet, and altering the fields to better suit our needs. The idea of consistency with the UDP protocol raised initial questions. Was it better to extend the already dominant packet structure and disregard data that was not of importance to our use case, or to completely alter its design, leaving only the general structure in tact. Both had benefits and drawbacks in this case, but they were mute once exploring other options.Ultimately we settled and honed in on the remote port protocol, but looking back, this implementation did have its beinifits.

While it would have been significantly underdeveloped when compared to the remote port infrustructure, the simplicity would have likely allowed us to complete additional protions of our project goals due to the large portion of work that was devoted to remote prot development. With that in mind, the remote port protocol would likely be the protocol of choice in future projects if it were not for the lack of documentation and implementation examples. When those components are accounted for, the remote port protocol shows great promiss in this space. Looking back, this was likely one of the more significant areas that we would have liked to taken into better consideration looking back, as it may have saved us significantly in development time by simiplying our model interactions, yet would have made our examples far less portable, a cornerstone of our project goals.

Sockets:

Our exploration into native sockets was promising. Our original plan was to use UDP sockets to accomplish the dynamic data element. However, the problem we encountered was that UDP sockets were really designed to work with the model and would have required significant amount of changes to the SystemC backend to get it to work. We looked into Unix Sockets (Berkely

Sockets) as well, as we felt this would be easier to implement, but there was still no way to put data arbitrarily into the model. Eventually, the open-sourced maintainers recommended that we use libremote-port, which essentially mapped a Unix socket into the model. While this solution was likely the best, we underestimated the difficulties we'd have getting this sparsely document library to work, despite the best efforts of our team. We perhaps would have had more success with some of the easier systems, but we elected to try to implement the one that best fit our requirements.

APPENDIX III: OTHER CONSIDERATIONS

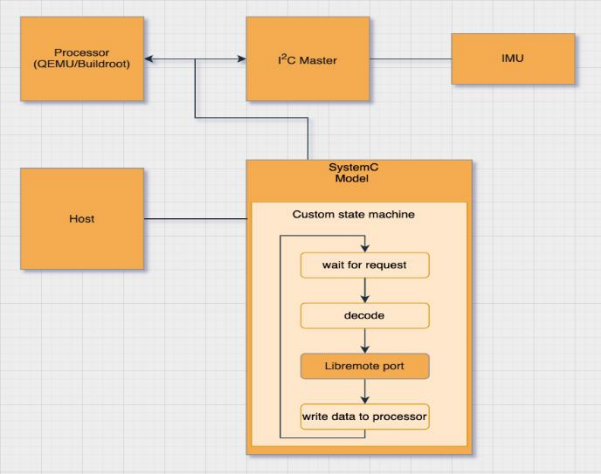


Figure 3: Xilinx QEMU Mixed Simulation Environment

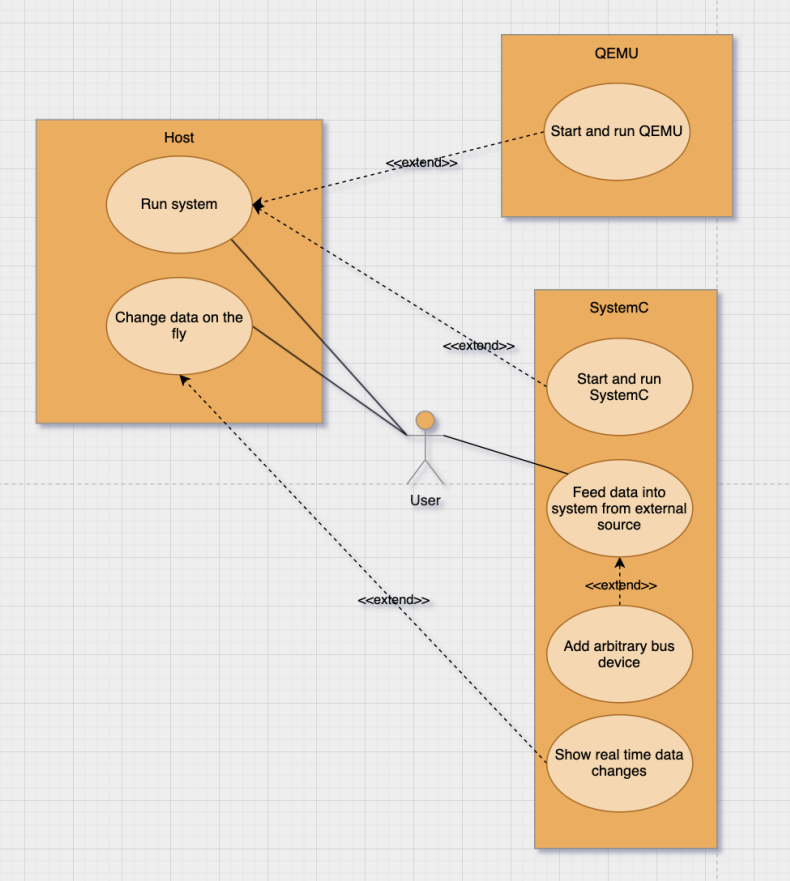


Figure 4: Use Case Diagram for our remote port demo

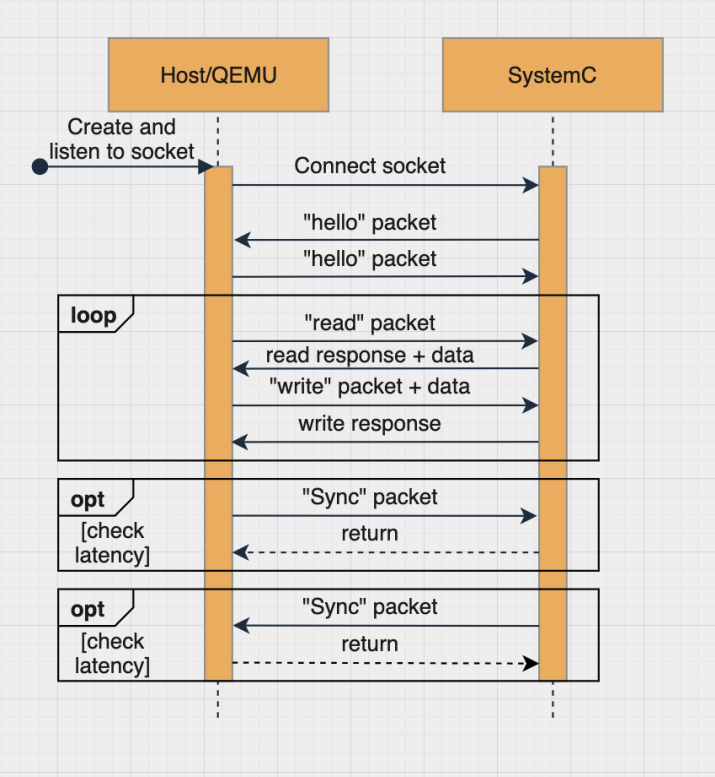


Figure 5: Sequence diagram of a two-way handshake between a host PC and SystemC

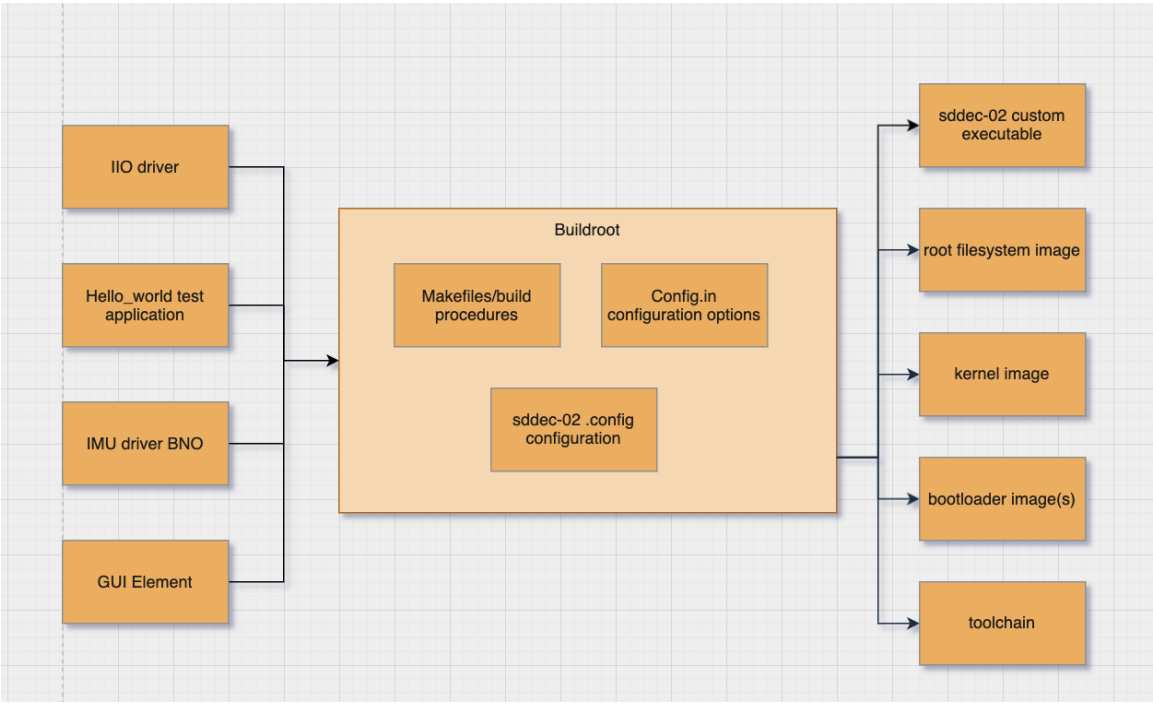


Figure 6: Buildroot software-interaction schematic

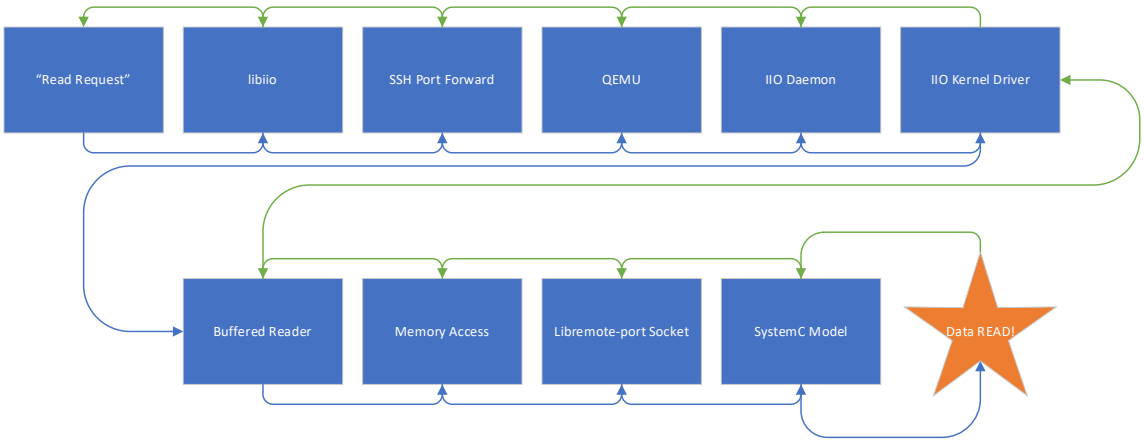


Figure 7: IIO Datapath