BACHELOR INFORMATICA

UNIVERSITEIT VAN AMSTERDAM

Extensible Simulation of Functionally Transparent SystemC-based System Architectures

R.A.J. Wacanno

May 1, 2020

Supervisor(s):
drs. T.R. Walsta,
drs. A. van Inge,
ing. E.H. Steffens

Signed: Signees

Abstract

Contents

CHAPTER 1

Introduction

One of the many advantages of software design—when compared to traditional hardware design, is the relative low bar of entry. Anyone with a computer, some time and the right amount of motivation, is able to start working on projects or teach him or herself the ins and outs of various programming languages or theories in programming. This used to be in stark contrast to hardware design. Trying to design custom hardware components or learning about the inner workings of certain components was done in either a strictly theoretical fashion, or in a more involved practical fashion. The first of these tactics typically entails reading up on the needed technical specifications from manuals and applying this to an abstract implementation in the form of diagrams. This way, testing of the functional correctness of a design had to be tested purely based on deduction from this theoretical implementation, without real world testing. Practical testing could also be a possibility, but this came with other disadvantages, like having to buy necessary components, equipment, and needing to have the required technical knowledge to handle said components and equipment.

With the dawn of more capable computational hardware, it becomes more and more feasible to design hardware in a more software-like fashion and test its functionality in a purely software based simulation framework. Applying these techniques to designing and understanding hardware eliminates the need for buying actual hardware, which, like software design, dramatically lowers the bar of entry.

Apart from this, hardware simulation introduces more possible advantages when compared to using real components. When running more complex components, like integrated circuits, the inner workings are hidden in a black box-like fashion. Without the use of specialised probing equipment, it is impossible to see what goes on inside such parts. With further increased complexity, having a look inside the a used components might even become impossible. By using simulated hardware,

This research in particular focuses on the simulation of computer system architectures. Central to these architecture is of course the central processing unit of CPU, which is inherently a complex components comprised of a number of discrete components brought together on a single piece of silicon. In order to perform the simulation of such a component, an appropriate description language is needed. In this case, the C++ library SystemC is chosen for the software-based implementation of the simulated system architectures.

In the end, one might still want to test a hardware design written in software in the real world. Therefore, this project aims to implement a systematic way in which the SystemC-based hardware implementation can be translated to a HDL-based implementation. Such an implementations should in turn be executable of the right FPGA hardware, and, as such, the software-described architecture implementation can be tested in a true hardware context.

1.1 Research question

In order to fulfil the needs presented in the introduction, a set of solid research questions have been formulated:

- How can a system architecture be accurately simulated using a SysteC-based framework?
- What systematic approach can be used to visualise the internal structure of a SystemC-based archtitecture in terms of components and data streams?
- What communication mechanisms are required to ensure external extensibility of a simulation framework?
- What constraints are placed on a SystemC hardware description when this description is to be converted to a VHDL-based description?

1.2 Outline

The following writing will introduce the theoretical concepts needed in order to perform this research in chapter 2. Based on this theoretical knowledge, the desired framework is implemented in an iterative fashion. The process of this implementation and the motivations behind specific implementation decisions is described in chapter 3. In chapter 4, results of tests done with the implemented framework are presented. Finally, chapter 5 features concluding remarks, a reflection on the research process of this project and possible future work which might follow from this project.

CHAPTER 2

Theoretical background

The following sections give a brief outline of the theoretical knowledge used to perform this research.

2.1 Simulation models

Since this research's main focus is on simulating system architectures, it is important to identify various methods of simulation. As is true for many thing, simulation can be done in numerous ways, and each method can be seen as more or less applicable in certain specific use cases.

2.1.1 Continuous-time simulation

The first simulation technique covered in this theoretical introduction is known as continuous simulation. This technique makes use of continuous models in order to simulate a given system. Such a continuous model is usually formulated using using (a series of) differential equations.

A classic example of a system which can be simulated using differential equations is the size of a given population [Birta and Arbez, 2013]. Let's say there is an animal population with a size $P(t)$ where t is a certain point in time. This simple definition already highlights the continuous nature of such a model, as the equation simulates the population size for a continuous time domain. With the population size as stated above, the population increase at a given point in time can be formulated using the differential equation $\frac{dP}{dt} = b(t) - d(t)$. In this equation $b(t)$ equates the rate of birth and $d(t)$ the rate of death at time t. Using the following system of equations, this model can be elaborated further into a full fledged model:

$$b(t) = P(t) \cdot 0.5 \tag{2.1}$$

$$d(t) = P(t)^2 \cdot 0.03 \tag{2.2}$$

Parameters like 0.5 and 0.03 as used in the example above, are determined based on experimentation or observations as they occur in the system which is sought to be simulated or based on assumptions about this particular system. Having formulated these equations, their behaviour can be plotted using a vector field. With such a diagram, expected behaviour will become apparent. This is also the case for the example model, as can be seen in its corresponding vector field in figure 2.1, where, for example, a clear equilibrium point can be observed at $P(t) \approx 26.5$. Based on these comparatively simple operations, one is able to predict that, in the case of this example population, any population with an initial size of $P(0) < 26.5$ is expected to increase in size as time passes, whereas a population with an initial size of $P(0) > 26.5$ will shrink according to this model.

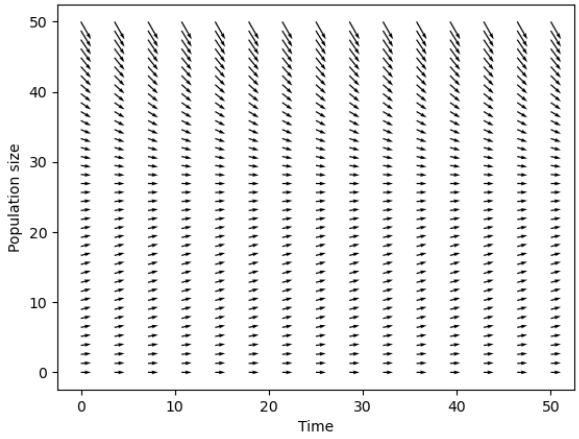

Figure 2.1: Vector plot of the differential equation system as described in section 2.1.1

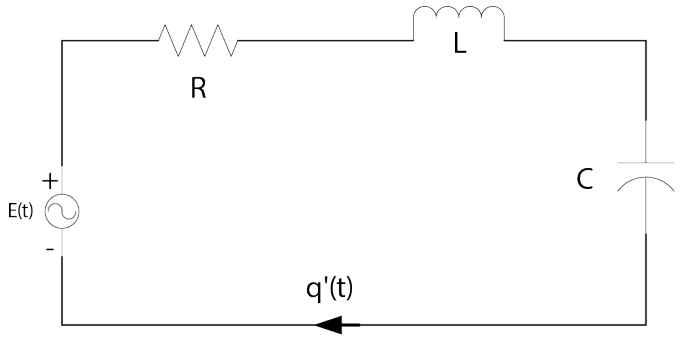

Figure 2.2: Example circuit [Birta and Arbez, 2013, p. 251]

Although the example given above is rather simplistic, continuous models allow for the simulation of systems with more complexity. As long as equations can be devised which accurately describe the desired characteristics of a system, such a system should theoretically be modelable using continuous methods. Herein lies however also one of the disadvantages of continuous models, as the succes of a given model greatly depends on the ability to formulate the appropriate equations. This fact is describes in Kohl et al. [2016] as it applies to the modelling of hardware using equations.

Application in hardware modelling

Knowing this, one might wonder how such techniques could be applied to the simulation of computing hardware, as these components would on a first glace not lend themselves to a behavioural description using mathematical equations. As it tuns out however, behaviour of fundamental electronic components like capacitors, resistors, inductors and voltage sources and the the way these parts interact with each other inside a circuit can also be modelled using mathematical equations. This fact might not be too surprising to anyone with an above-average background in the field of physics as it pertains to electricity.

An example of this ability to model electrical circuits using continuous modelling is featured in Birta and Arbez [2013]. A circuit, as features in figure 2.2, contains a voltage source ($E(t)$), a resistor (R), a capacitor (C) and an inductor (I) in series. The book formulates an equation (equation 2.3) which is able to model the state of the circuit based on Kirchoff's voltage law. Without getting into extraneous detail about the specifics of electrical engineering, it follows from this equation that the state of the components present inside the circuit can be derived from the initial values of the state variables ($q(t)$ and $q'(t)$ at t_0) and the value of the input variable ($E(t)$).

$$Lq''(t) + Rq'(t) + \frac{q(t)}{C} = E(t) \tag{2.3}$$

These general characteristics of electronic components have also been used in a more computational context. In the past, the use of electrical analogue computing equipment, like the machine in figure 2.3 were used to perform simulations on physics models—e.g. tidal calculations or certain models used by the oil industry. These analogue computers contain circuits with basic electrical components whose electrical properties are similar to certain mathematical constructs. As such, a circuit built from the appropriate components is able to be used as the electrical analogue of an equation. When looking at the simulation of such machines, we are presented with a peculiar chicken and egg problem. It is clear that the analogue machine in this example was used for continuous simulation purposes, and as such, simulating it using a continuous model should logically be possible. While no example of such a simulator appears to exist, is need be, it should be possible to device a model to simulate the functionality of such a computer.

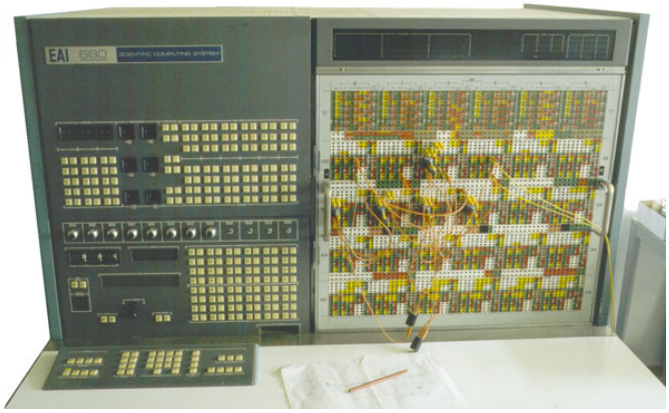

Figure 2.3: Electronic Associates Inc. Model 680 [Dooijes]

While not used to simulate the specific functioning of computing hardware, continuous models can still play a role in contemporary research. Using a model based on mathematical equations, certain characteristics of a given hardware component can be examined. For example, Berg et al. [2006] describes a method of predicting caching behaviour of a program, in particular data locality in the context of multi-threaded applications, based on a mathematical model. A given application is executed and during this execution certain architecture-independent data is collected. Following the execution, this collected data is fed into a series of equations, each of which aims to predict certain metrics. Another metric which can be modelled using continuous simulation is the power consumption of a given architecture [Kohl et al., 2016].

From these two examples, it becomes apparent that in the context of digital system components and system architectures, a continuous model does not satisfy the need for accurate functional simulation.

For the purposes of simulating the functionality of system architectures as performed in this research, continuous simulation and continuous models have been deemed impractical, and inappropriate to the particular needs of the simulator. As will be discussed in the following subsection (2.1.2), there exists a more suitable simulation technique with respects to the functional simulation of digital system architectures.

2.1.2 Discrete-time simulation

Converse to the aforementioned continuous simulation there exists discrete simulation. Instead simulating a system with a theoretically infinite time resolution, the state of the system is simulated at

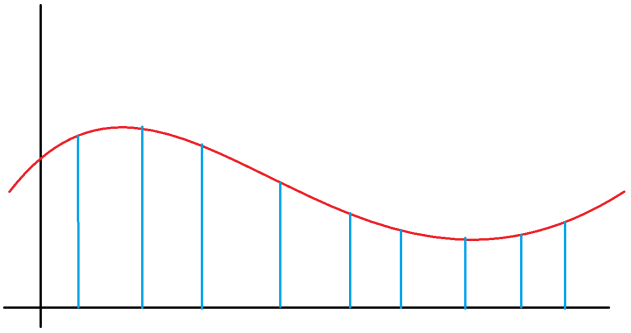

Figure 2.4: [TEMPORARY FIGURE] Continuous signal versus sampled values

discrete time intervals. A helpful analogy to illustrate the difference between these two simulation techniques might be that of the sampling of audio signals—technically any electronic signal. From a purely physical perspective sound is just a difference in pressure propagating through a gaseous medium. This phenomenon is continuous, as at any point in time the system has a certain state. When we want to record this sound however, the continuous nature poses a problem, as recording the signal with an infinite time resolution would require an infinite amount of storage. Instead, the signal is measured at a specific regular interval. From these sampled measured values, the original signal can be reconstructed. This deference between an original signal (red) and sampled values (blue) is illustrated in figure 2.4.

Discrete event modelling

While discrete modelling can be done using equations, as was true for its continuous counterpart, discrete simulation can also be based on the occurrence of events and the relationship between these events. Such event-driven modelling lends itself particularly well to the simulations of systems where there is a clear action-reaction relation between actors. As will be further illustrated in section 2.3, this latter characteristic holds true for computer system architectures, where components react to a given stimulus and this reaction might in turn be the triggering stimulus for one or more other components.

Choosing which events are allowed to be handled at a particular stage of the simulation depends of whether the simulation is either time driven or event driven. In the former case, handling of events takes place based on the the clock inside the simulation. Inside such a simulation, there is a global time and events are scheduled based on their starting time.

An example of this technique **[EXAMPLE]**

In case of the event driven approach, events are handled as they are triggered by other events. There is no set regular time between each simulation step, instead, timing between stages is based on the duration of events and the order in which new events are triggered.

this technique can be illustrated using a slightly tweaked version of the example of the former technique. **[EXAMPLE]**

2.2 SystemC

Central to this research is the C++ class library SystemC [Association et al., 2011]. SystemC is a library aimed at allowing the user to implement hardware functionality on various different levels of abstraction. [mist nog spul]

What makes SystemC especially suitable for this research is the way it allows for multiple levels of abstraction. On the one hand, using SystemC, an entire system architecture can be constructed by combining atomic logic gates who's individual implementation involves trivial bit operations. On the other side of the spectrum, SystemC allows for the implementation of a components complete

functionality using C/C++ code, while still using SystemC as its interface to communicate with other components.

With the use of the example module in listing 2.1, the sections below will illustrate the basic principles and functionality of SystemC.

```
SC_MODULE(example)
{
    sc_in<bool> clock{ "CLOCK" }, poll{ "POLL" };
    sc_out<sc_uint<20>> out{ "VALUE" };
    and_g and1{ "AND1" };
    not_g not1{ "NOT1" };

    sc_signal<int> sig{ "SIG" };

    void handle_poll()
    {
        out.write(1337);
    }

    void handle_clock()
    {
        while (true)
        {
            out.write(5318008);
            wait();
        }
    }

    SC_CTOR(example)
    {
        SC_METHOD(handle_poll)
        sensitive << poll;

        SC_THREAD(handle_clock)
        sensitive << clock.pos();

        and1.out(sig);
        not1.in(sig);
    }
};
```

Listing 2.1: SystemC example module

2.2.1 SystemC components and constructs

Data types

Apart from the new classes it introduces, SystemC also comes with some data types which are useful in a hardware design context.

`sc_int<w>` and `sc_uint<w>` are the respective signed and unsigned fixed width integer types in SystemC. Using the template variable `w`, the amount of bits used to represent the value of the integer in question can be specified. These data types ensure the correct bit width independent on the architecture it is compiled on, as opposed to C/C++ built-in types like `int`. The only limitation of these types is the fact that they are limited to a width of 64 bits. In order to use integer values with arbitrary bit width, `sc_bigint<w>` and `sc_biguint<w>` can be used as the respective counterparts of the aforementioned types.

In order to steam line the passing of and the operations on groupings of individual bits, SystemC includes a so called bit vector in the form of `sc_bv<n>`. With this vector comprising of `n` bits, each individual bit can be addressed and manipulated. This eliminates the need to perform bit masks on existing data types like `int` and allows for the formation of more flexible amounts of bits—other than the usual 8, 32 and 64 bits.

When dealing with true hardware signals, value analogies using only two values—i.e. 'true' or 'false' or 1 and 0—actually don't suffice. In the real world, the values of a signal are characterised using one

of the following possible categories: false (0), true (1), intermediate value (X) and floating value (Z). These possible values can be encoded in SystemC using the `sc_logic` data type. Several logic values can be combined using the `sc_lv<w>` type.

Modules

The fundamental building block of any SystemC-described hardware component is the module. Modules are packets containing the internal structure of a given component and its functionality. Listing 2.1 shows the definition of a new module called `example` (line 1-35) and the instantiation of 2 e existing modules `and_g` and `not_g` in lines 5 and 6 respectively.

Ports

In order to facilitate in- and outbound communication from withing modules, SystemC offers so called ports. The three basic types of ports `sc_in<T>`, `sc_out<T>` and `sc_inout<T>` handle inbound, outbound and a combination of the two respectively. Each port has to specify the type of data which is transported by way of this port using the template variable `T`. In the example in listing 2.1, the ports of the module are declared in lines 3 and 4. In this case, there are two input ports of type `bool` and one output port of type `sc_uint<20>`. Processes can read from a port using the `read()` method and write to a port using the `write([value])` method.

Signals

The aforementioned modules with their respective ports exist on their own, but somehow need to be connected. To establish this connection the supplied type `sc_signal<T>` can be used. As with the ports, for each signal the desired data type which is to be transported along the signal is specified using template variable `T`. In the example in listing 2.1 a new signal is declared in line 8 and the signal is connected between the output of the and gate in line 32 and the input port of the not gate in line 33.

2.2.2 Processes

Having described the structural components of a SystemC hardware design, the following outlines the way in which functionality of components can be implemented in SystemC. As described in the introduction of this section, functionality of SystemC can be constructed on various levels of abstraction. The functional implementation of a component like a half-adder can be done by combining logic gates inside a overarching module, or bu implementing the addition logic inside the half-adder module using C/C++ code. The latter of these techniques and also the functionality of the logic gates used in the former technique both rely on the use of processes inside a SystemC module.

Methods and threads

Processes inside a SystemC module come in two main distinct varieties, namely methods and threads.

Simulation scheduling

2.2.3 Constructing hierarchy

SystemC allows for the embedding of components according to a desired hierarchy. This allows for the construction of more basic types, which in turn can be combined to form more complex structures. In principle, this means that based on mere atomic logic gates (e.g. AND, OR, NOT and XOR), or even single transistors, a whole system architecture can be implemented. A rudimentary example of this process is given in listing 2.1, where through the combination of an AND gate and a NOT gate inside the the `example` module using a signal, a new NAND gate is constructed.

2.3 System architectures

2.3.1 MIPS

2.4 Existing simulation solutions

This project is not the first in the field of architecture simulation, and, as such, similar solutions have been devised in order to satisfy the need for a system architecture environment. The following are a few of these existing packages, their aims and a comparisons of these aims and the aims of this research.

2.4.1 ArchC

While not necessarily a simulation framework on its own, ArchC calls itself an Architecture Description Language (ADL) which intends to formulate a language, based on generic SystemC, which can be used to implement system architectures [Rigo et al., 2004].

2.4.2 PTLsim

A project more akin to the simulator envisioned bu this project is PTLsim [Yourst, 2007].

2.4.3 gem5

Binkert et al. [2011] Menard et al. [2017]

2.4.4 SIM-PL

A project which close to this project in terms of functionality

2.5 Used libraries

2.6 VHDL

Côté and Zilic [2002] Chen [2011]

2.6.1 Hardware description language

2.6.2 VHDL —

2.6.3 SystemC to VHDL conversion

Constraints to SystemC

2.7 Ethical implications

CHAPTER 3

Implementation

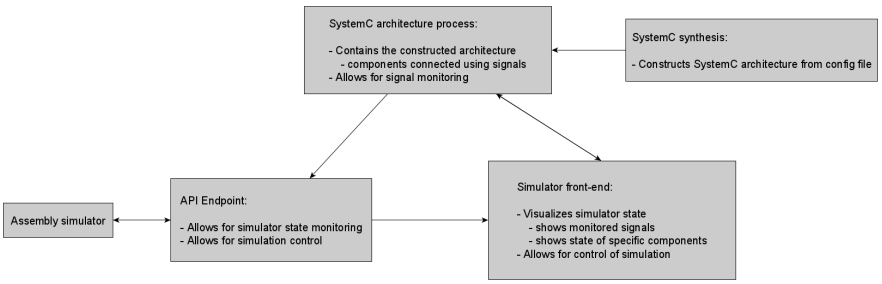

Figure 3.1: Global program design

3.1 SystemC synthesis

Before any simulation can be run, the architecture has to be implemented first. To aid in this process, the framework offers a way to synthesise a SystemC module based on a configuration file. This allows for the abstraction of certain implementational specifics, such as special components used by the simulation framework—some of which will be introduced further in this implementation chapter, without the user having to integrate these specifics manually. The configuration file used by the SystemC synthesis is TOML (Tom's Obvious, Minimal Language) [Preston-Werner and Gedam, 2020].

A popular mark-up language for purposes like these is XML, used by SIM-PL for example. TOML was chosen instead however

3.2 Architecture simulation

The basic global organisation of this projects simulation framework is outlined in figure 3.1.

3.2.1 Process communication

Simulation events

Application programming interface

3.2.2 Testing architecture

Instruction-set architecture

Before implementation of the testing architecture could commence, an ISA (instruction-set architecture) had to be made. This ISA outlines each of the instructions which has the be supported by the system architecture and the specifications of these instructions; details like instruction word length and instruction argument layout. Since this architecture would primarily be used for testing purposes, the ISA would not have to be as extensive as those for architectures like x86, ARM or MIPS. Therefore, only a few instructions were selected for implementation. The selection was carefully however, to ensure that while the architecture would be basic, its limited instruction-set could theoretically be used to implement somewhat sophisticated assembly programmes. The chosen instructions and their argument layout is shown in table 3.1.

Hardware design and implementation

Based on the ISA, an architecture was designed which supported each of the outlined instructions. This first implementation design is illustrated schematically in figure 3.2.

Instruction	*Op-code*	*Arguments*
add	0x0	*destination register, oprand register 1, oprand register 2*
addi	0x1	*destination register, oprand register 1, immediate value*
jmp	0x2	*jump address*
mv	0x4	*destination register, source register*
	0x5	*destination memory address, source register*
	0x6	*destination register, source memory address*
	0x1	*destination memory address, source memory adress*

Table 3.1: Instructions present in the ISA

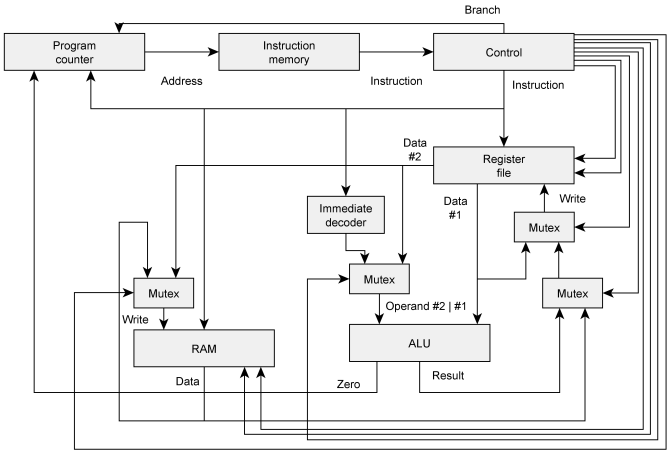

Figure 3.2: [TEMPORARY FIGURE] Schematic diagram of the test architecture

CHAPTER 4

Results

CHAPTER 5

Conclusions

Bibliography

IEEE Standards Association et al. Ieee std. 1666–2011, open systemc language reference manual, 2011.

Erik Berg, Håkan Zeffer, and Erik Hagersten. A statistical multiprocessor cache model. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–99. IEEE, 2006.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

Louis G Birta and Gilbert Arbez. *Modelling and simulation*. Springer, 2013.

Rui Chen. Synthesizable systemc to vhdl compiler design. 2011.

Christian Côté and Zeljko Zilic. Automated systemc to vhdl translation in hardware/software codesign. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, pages 717–720. IEEE, 2002.

E.H. Dooijes. Electronic associates inc. model 680 scientific computing system. URL `https://ub.fnwi.uva.nl/computermuseum/eai.html`.

Johannes Kohl, Marc Reichenbach, Carsten Demel, and Dietmar Fey. A systemc based framework for cycle accurate processor simulation and parameter analysis. *IFAC-PapersOnLine*, 49(25):92–97, 2016.

Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and systemc: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 62–69. IEEE, 2017.

Tom Preston-Werner and Pradyun Gedam. toml-lang/toml, Apr 2020. URL `https://github.com/toml-lang/toml`.

Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. Archc: A systemc-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73. IEEE, 2004.

Matt T Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE, 2007.