The Thesis Committee for Vivek Ramanathan
certifies that this is the approved version of the following thesis:

VWSIM: A Circuit Simulator

SUPERVISING COMMITTEE:

Warren A. Hunt, Jr., Supervisor

Kenneth McMillan

VWSIM: A Circuit Simulator

by

Vivek Ramanathan

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2023

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my thesis advisor, Warren A. Hunt, Jr.. This work would not have been possible without him. His genuine research and career mentorship, his sincere investment in my growth as an ACL2 programmer, and his valuable life advice are lessons I will carry with me throughout my life.

I would like to thank Matt Kaufmann, who not only enabled the use of floating-point arithmetic in VWSIM and created VWSIM's SPICE to S-expression parser, but also spent countless hours guiding me through rigorous ACL2 proofs. Thank you also to J Moore for his support in developing VWSIM's Gaussian elimination solver.

I would also like to thank Ivan Sutherland for creating the circuits we used to test the simulator and for offering many insightful discussions on RSFQ circuits. His unwavering support, mentorship, and detailed feedback were instrumental to this project. Thanks also to my colleagues Marly Roncken, Gary Delp, and Steven Ruben for their insight and feedback. Additionally, thanks to ForrestHunt, Inc. and the U.S. Army for supporting this work.

On a personal note, I would like to thank the amazing graduate students on the GDC 5th floor for creating such a positive and friendly work atmosphere. Finally, I am extremely grateful to my parents, brother, and partner for their love and continuous support over the years. This would have not been possible without them.

Abstract

VWSIM: A Circuit Simulator

Vivek Ramanathan, M.S.C.S
The University of Texas at Austin, 2023

SUPERVISOR: Warren A. Hunt, Jr.

VWSIM is a circuit simulator for rapid, single-flux, quantum (RSFQ) circuits. The simulator is designed to model and simulate primitive-circuit devices such as capacitors, inductors, Josephson junctions, and can be extended to simulate other circuit families, such as CMOS. Written in the ACL2 logic, VWSIM provides formal mathematical guarantees about each of the circuit models it simulates. The ACL2-based definition of the VWSIM simulator offers a path for specifying and verifying RSFQ circuit models.

Table of Contents

List of Figures

Chapter 1: Introduction

Circuit simulation has a long history. SPICE [17] was defined in the 1970s to provide circuit simulation; SPICE and its derivatives have been used broadly in circuit simulation for fifty years. The models for superconducting devices, such as Josephson junctions, are not typically included in CMOS-compatible circuit simulators. These superconducting devices provide a path toward high-speed, low-energy computing logics. One such superconducting computing logic is Rapid Single Flux Quantum (RSFQ).

We have defined the VWSIM circuit simulator with simulation models for RSFQ circuits [6]. This includes mathematical models for Josephson junctions (JJs), as well as resistors, capacitors, inductors, transmission lines, mutual inductance, voltage sources, current sources, and phase sources. Additionally, with the use of the ACL2 [8] theorem prover to implement the VWSIM electrical-circuit simulator, VWSIM enjoys mathematical properties not available in other simulation systems: termination, memory safety, logical consistency, and a formal semantics for reasoning about circuit models. Developing tools using ACL2 provides a number of benefits. Each ACL2 function, when defined, must be mathematically proven to terminate and proven to access reachable data only; thus, all ACL2 programs are known to terminate, without memory-reference errors.

Chapter 2: Background

2.1 Motivation

When the effort to develop VWSIM began, we identified properties we hoped RSFQ modeling and analysis would provide. We realized that available simulators for RSFQ circuits were all void of some properties we desired. These shortcomings did not impede our use of public-available simulators, but no simulator combined all of the properties we felt were important for investigating RSFQ circuit models thoroughly and accurately. Some of the goals for VWSIM include:

1. **ACCURATE:** We wrote VWSIM with clarity and simplicity in mind.
2. **RELIABLE:** VWSIM is proven to terminate without memory-access errors.
3. **FLEXIBLE:** VWSIM offers a general-purpose stimulus language.
4. **INTEGRAL:** VWSIM supports voltage and phase-based simulation.
5. **TIMING:** Fixed and variable time-step simulations are available.
6. **ACCESSIBLE:** All code is freely available; runs on multiple systems.
7. **INTEROPERABLE:** Use of VWSIM can be scripted with ACL2 or other tools.
8. **CAN BE PAUSED:** Any simulation may be paused, saved, and later re-started.
9. **EXTENSIBLE:** VWSIM can be extended with additional components and stimulus.
10. **ANNOTATION:** VWSIM models can be annotated with physical parameters.

11. **PROMPT:** VWSIM can be used interactively; signal traces are memory resident.

12. **FORMAL:** The VWSIM is written in a mathematical modeling language.

13. **INTROSPECTIVE:** VWSIM source code has been analyzed with the ACL2 system.

14. **ANALYSIS:** VWSIM code and circuit models can be analyzed symbolically by ACL2.

We began the development of our simulator for several reasons: to improve our understanding of the mathematics of JJ-based circuits, our failure to understand what existing simulators were doing, a lack of a way to program existing simulators to carry out collections of simulations, the lack of a way to alter a simulation after it is started, the lack of an interactive means to control the simulation process, the lack of a programmed method for inspecting results, and the lack of a formal semantics for JJ-based circuits. As we have developed our simulator, we have learned that specifying the interconnections of components does not provide an adequate model for the behavior of in JJ circuits. For instance, transformers are modeled by coupling two inductors, and JJs have a non-linear behavior, where tracking the phases, as well as the voltages and currents, is critical for performing accurate simulations.

2.2 Related Work

The design of VWSIM was strongly influenced by JoSIM [5]. JoSIM is a SPICE-compatible simulator released in 2018 that was developed as a replacement for the JSIM simulator. It provides a simple command-line interface to simulate circuit descriptions provided as "cir" files. It was the first simulator to include a phase-based simulation mode – phase is an important concept in modeling the mathematics of the Josephson junction. Developed in C++, JoSIM makes use of the highly

optimized SuiteSparse [4] sparse matrix libraries. Initially, we conducted our study of superconducting circuits using JoSIM as the simulation engine for circuit models we created using the Electric CAD system [14].

During our use of JoSIM, we encountered several JoSIM simulation results we could not explain nor understand – and, in some instances, we submitted a bug report. In one instance, we were misled for nearly a month before we were able to distill our confusion down to a model so small that we could compare our by-hand simulation with JoSIM results. Eventually, we decided that the best way we could really understand how an electric circuit simulator for JJs operates was to develop our own simulator.

Other circuit simulators commonly used in the superconducting community include WRspice and PSCAN2. WRspice [16] is a SPICE simulator, released in the late 1990s, that is designed for interactive use and provides an integrated graphical package. It also provides as well as a batch-mode, non-graphical application. WRspice also has features to allow Monte Carlo analysis and dispatching of jobs to multiple computers for parallel operations. PSCAN2 [13] is a SPICE-based circuit simulator that was released in 2016. Implemented as Python module, PSCAN2 can perform circuit simulation, calculate margins or circuit parameters, and optimize circuit parameters.

2.3 Rapid Single Flux Quantum

VWSIM can simulate circuits in the Rapid Single Flux Quantum (RSFQ) logic. RSFQ is an approach to use superconductors (materials with no electrical resistance) for computing, where digital bits are represented by a single quanta of magnetic flux in superconducting loops [11].

Computation in these superconducting circuits is performed by generating, storing, and/or transmitting magnetic flux quanta (bits). The electrical component that makes these operations possible in the RSFQ logic is the Josephson junction. A

Josephson junction (JJ) is a two-terminal device, made up of two or more superconductors coupled together by a thin layer of insulating or metallic material. These JJs are able to channel current up to a certain threshold, known as the critical current, without any voltage across it. Once the critical current is exceeded, the voltage across the JJ is non-zero. It is the switching of the JJ between the non-zero and zero voltage states that forms the basis of the RSFQ logic. Since the voltage applied on the JJ is low (zero or a small non-zero value), the power consumption of these JJs is also low. This has inspired the development of energy-efficient computing logics, such as RSFQ, that make use of JJs.

VWSIM uses the Resistively and Capacitively Shunted Josephson junction (RCSJ) model for JJs [10]. A JJ is modeled as a basic JJ – a junction that is always in the zero voltage state – in parallel (shunted) with a capacitor and non-linear resistor. VWSIM captures this model as a set of mathematical equations, which are used for simulation:

$$i(t) = I_c \sin \phi(t) + C\frac{dv(t)}{dt} + \frac{1}{R}v(t) \tag{2.1}$$

$$\dot{\phi}(t) = \frac{2e}{\hbar}v(t) \tag{2.2}$$

where $i(t)$ is the current through the junction at time t, I_c is the junction critical current, ϕ is the phase[1] across the junction, C is the capacitance of the parallel capacitor, $v(t)$ is the voltage across the junction at time t, R is the resistance of the nonlinear parallel resistor (the resistance changes with respect to the voltage across the junction), e is the electron charge, and $\hbar$ is Planck's constant.

[1]Phase is a concept widely used in superconducting systems; phase is proportional to the time integral of voltage. Phase is a potential. The current through an inductor is proportional to the phase across the inductor divided by its inductance. The current through a JJ is proportional to the sine function of the phase across it. Phase is specified in units of radians.

VWSIM also captures the sets of equations for resistors, inductors, capacitors, mutual inductance, and transmission lines.

2.4 ACL2

An unusual feature of VWSIM is that its definition has been analyzed by ACL2 [7, 8]. ACL2 is an interactive theorem proving system. It combines a Lisp-based programming language with a logical, reasoning engine. ACL2 is designed to support automated reasoning, which can be used for verifying software and hardware. VWSIM is defined within this system, where properties about the simulator can be mathematically proven to hold.

During the development of VWSIM, hundreds of theorems were proven to assure that VWSIM operates reliably; our specifications and proofs help assure that our data structures remain well-formed. This ability was an important component of our development process as it allowed us to confirm properties of functions used in VWSIM's operation. In addition, it is even possible to analyze the behavior of VWSIM with respect to user-supplied model input; we have not yet explored this capability with respect to VWSIM models, but such work has been performed on other projects [1, 15].

Chapter 3: VWSIM

3.1 Circuit simulation

VWSIM provides a simple-to-use interface through the ACL2 read-eval-print-loop. The user starts ACL2, loads the simulator, and then interactively uses the simulator by executing a series of commands. The simulator provides commands to run a simulation of an input circuit model (`vwsim`), access and save the simulation output in various formats (`vw-assoc, vw-output`), and plot the simulation output using the gnuplot graphing utility (`vw-plot`). The descriptions and various options for each of these commands is available in the VWSIM online documentation.[1]

The simulator is invoked by running the `vwsim` function. VWSIM requires the user to provide a SPICE (.cir) circuit description file, VWSIM-style circuit netlist or a LISP file with a previous VWSIM simulation state. The rest of the arguments are optional with default values. The input file formats will be described in the subsequent section.

A VWSIM simulation is best illustrated by a simple circuit description and its corresponding simulation. Below is a very simple, RC circuit schematic. This circuit is composed of three elements: a voltage source, a resistor, and a capacitor. Lines (wires) in a schematic are given names, such as `vs1`, `vc1`, and `gnd` as shown in Figure 3.1.

The RC circuit schematic diagram in Figure 3.1 can be written in the native netlist format as shown below. The name of this netlist is `*rc-netlist*` and it contains one module, `rc-module`. This module contains no external connections, as indicated by the `nil` in its second line. This module contains three components: a voltage source, a 1-ohm resistor, and a 1-farad capacitor. The first field in the

[1] `https://www.cs.utexas.edu/users/moore/acl2/manuals/latest/?topic=ACL2____VWSIM`

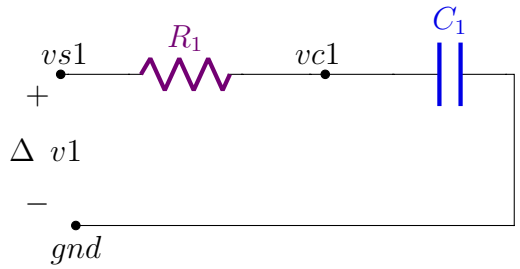

Figure 3.1: An RC circuit with a 1-Ohm resistor, R_1, and a 1-Farad capacitor, C_1.

component statements are occurrence names; they must be disjoint. The second field specifies the type of components; in this netlist there is a voltage source (`v`), a resistor (`r`), and a capacitor (`c`). The third field specifies the nodes to which each component is connected. The fourth field contains names to reference the current that passes through the individual components. The final (fifth) field specifies the value(s) for each component.

```
(defconst *rc-netlist*
  '((rc-module
     nil
    ; Name type  connections branch   value
     ((v1    v    (vs1 gnd)  (i-v1)   ((if ($time$< '1/5) '0 '1)))
      (r1    r    (vs1 vc1)  (i-r1)   ('1))
      (c1    c    (vc1 gnd)  (i-c1)   ('1))))))
```

The final field of the voltage source `v1` specifies its behavior: while simulation time is less than one 1/5 second, the voltage it produces is zero; once the time reaches one 1/5 of a second, the voltage source produces a one-volt output for the rest of the simulation.

As shown below, one may check whether a netlist is well-formed using the `NETLIST-SYNTAX-OKP` and `NETLIST-ARITY-OKP` predicates (functions). After checking that the circuit model is well-formed, the circuit can be simulated. The `vwsim`

form specifies a simulation starting at time 0, proceeding with a 1/5-second simulation time step until two seconds of simulation has been completed using a `voltage`-style simulation.

```
(netlist-syntax-okp *rc-netlist*)
(netlist-arity-okp  *rc-netlist*)

(vwsim *rc-netlist*
       :time-step 1/5
       :time-stop 2
       :sim-type 'voltage)
```

As the RC circuit model is simulated, VWSIM stores the simulation values for each time step. Upon completion, we can request specific simulation results. Using the `vw-output` command, we can fetch the voltage values across the resistor and capacitor, for all time steps, and write them to a comma-separated values (CSV) file.

```
(vw-output '((DEVV . R1) (DEVV . C1))
           :output-file "Testing/test-circuits/csvs/rc-voltages.csv")
```

VWSIM provides the `vw-plot` command to run gnuplot. Here, we plot the voltage across R1 and C1 with respect to time. The plot generated is shown in Figure 3.2. This capacitor-charge simulation shows voltage `V(C1)` approaching one volt. At the beginning of the simulation, the voltage across the resistor R1 increases quickly. This is due to the sudden change in voltage across the voltage source, V1, from zero to one volt at 1/5 seconds. Since the simulation runs with 1/5 second time-steps, the change from zero to one volt across V1 occurs in one simulation step. The voltage values for R1 between 0 seconds and 1/5 seconds are linearly interpolated by the gnuplot plotting software in Figure 3.2.

```
(vw-plot '((DEVV . R1) (DEVV . C1))
         "Testing/test-circuits/csvs/rc-voltages.csv")
```

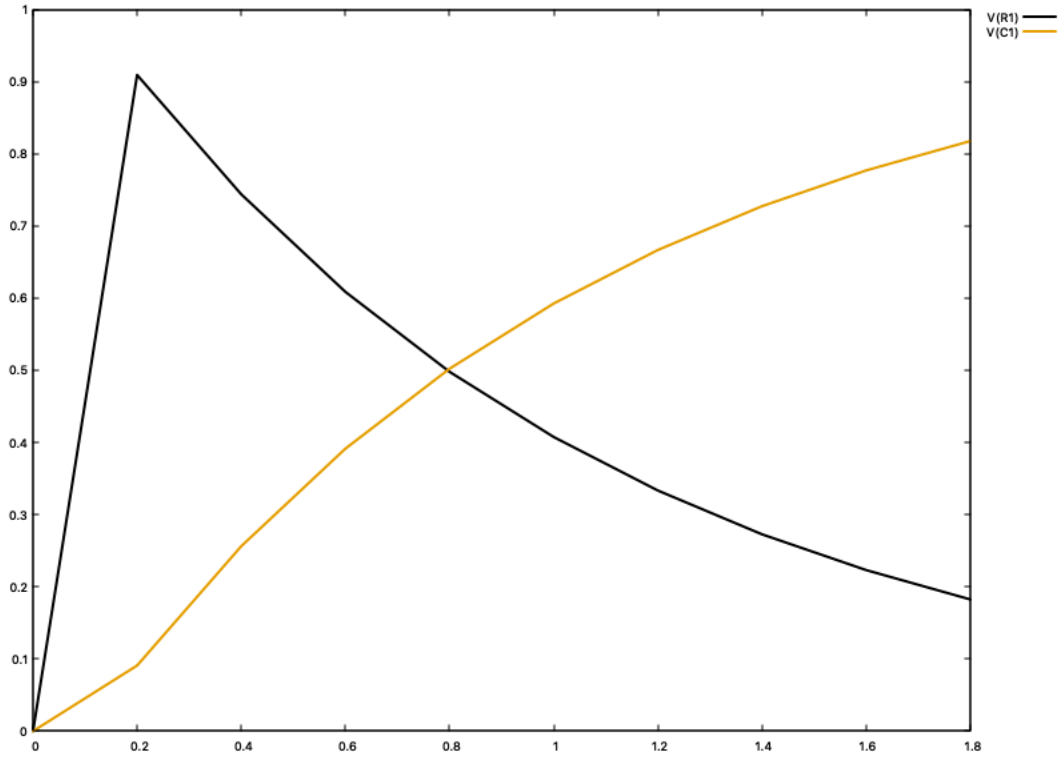

Figure 3.2: Plot of a one-second simulation of an RC circuit using the `vw-plot` command

VWSIM uses floating-point numbers as seen in the rc circuit simulation plot (Figure 3.2), but ACL2 provides only rational arithmetic natively. We note that there is an ongoing effort to modify ACL2 so it can use floating-point numbers to approximate ACL2's native rational-number arithmetic by appealing to contemporary, (IEEE-compatible [9]) floating-point implementations; we discuss this more in Section 5.1.

The rest of this chapter describes how VWSIM simulates a circuit: the expected input and output formats, and the pipeline VWSIM follows to perform a

transient analysis of a circuit.

3.2 Input descriptions

3.2.1 SPICE

SPICE provides many textual commands to define and simulate a circuit, of which VWSIM understand a small subset. The commands are written to a file with extension ".cir", which a SPICE-compatible simulator then reads in and attempts simulation of the circuit. The SPICE circuit descriptions can be hierarchical, i.e., circuit definitions can be broken down into smaller subcircuit definitions and then instantiated and connected with wires.

The SPICE commands that VWSIM currently understands are listed below.

Device commands: resistor, inductor, capacitor, Josephson junction, mutual inductance, transmission line, voltage source, current source, and phase source.

Simulation commands: transient analysis, global nodes, subcircuit definition, model statements, and print statements.

3.2.2 Hardware Description Language (HDL)

VWSIM also accepts heirarchical netlist descriptions in its native LISP-style HDL. At a high-level, a netlist is a list of modules where each module consists of a name, input/output nodes, and a list of occurrences. An occurrence is either a primitive device (resistor, inductor, etc.) or an instantiation of a defined module[2] .

Unlike SPICE, VWSIM HDL separates the circuit netlist description from it associated simulation commands (e.g., simulation times, print requests). Such simulation-control commands are provided as arguments when invoking the simulator.

VWSIM offers a collection of primitive circuit models, which are combined

[2]The full HDL syntax is available in the online documentation: `https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____VWSIM-HDL`

into a single model for eventual circuit simulation. These basic circuit models have mathematical descriptions based in circuit theory and physics.

3.3 Simulation output

VWSIM saves all values produced during a simulation; these values may be used for subsequent processing such a analysis and/or printing. The saved results can be processed in the ways described below. An advanced user of VWSIM can access the entire trace of simulation values and write programs to explore the data and even control the simulation as it advances; we will explore some of this advanced mechanism in Section 4.1.

A VWSIM user interacts with VWSIM through a (Lisp-style) command-line, and can request various types of output:

1. Comma-Separated Values (CSV) file: the user-requested values are written to a file in CSV format.

2. Association-list: the user-requested values are returned in a LISP-style association (key-value pair) list. The values can be accessed interactively by name using the `vw-assoc` command.

3. Saved (LISP) file: an entire simulation state is saved to a file, which can be provided to VWSIM at a later time so as to resume the saved simulation.

4. Graphical Plots: the user-requested values can be plotted directly from the ACL2 loop using gnuplot.

3.4 How VWSIM works

VWSIM accepts a hierarchical circuit description as a list of circuit modules, which it then flattens into a list of primitive devices with no module (subcircuit)

references. The flat netlist is checked for consistency, making sure that there are no undefined references or unrecognized components. With a valid, flat netlist, VWSIM extracts a system of symbolic equations; these equations include components (e.g., a specific resistor) and sources (e.g., some time-varying current source), where their, potentially time-varying, values are to be instantiated during simulation. The system of equations are used to build a symbolic matrix equation. If the resulting matrix, A, is well-formed (i.e., not singular) and the values, in b, are suitable, then VWSIM prepares to solve for x in matrix equation $Ax = b$ by instantiating the symbolic expressions in A and b with the current simulation values. The solution vector, x, represents the next-time values for the simulation variables for a time-step value specified by the user – this process is repeated until simulation time is exhausted. VWSIM records the voltages and/or phases of all wires (nodes) as well as the currents through all components except resistors and JJs. The currents through resistors and JJs can be calculated after the simulation, if requested. A VWSIM user can then process the simulation history, either by inspection, by analysis with an ACL2 program, or by visualizing some projection of the recorded information.

3.4.1 The VWSIM workflow

Given a textual description of a circuit, VWSIM will process it (see Figure 3.3):

1. Convert input file, if in SPICE-format, into Lisp S-expressions.
2. Transform Lisp S-expressions into list of circuit module references and simulation control statements.
3. Convert module references into a hierarchical netlist representation.
4. Flatten and sort hierarchical netlist into a list of primitive device references.
5. Create a symbolic version of the model; i.e., create the symbolic $Ax = b$ matrix using the modified nodal approach (MNA) [3] procedure.

6. Given a time step size (or using an automatically selected time step size), solve for x in $Ax = b$. Use these new x values to extend the simulation, and repeat this process until the end of simulation time.

7. Finally, process and/or save the time-value results produced by the simulation; for instance, print results into a file of circuit-node values or check the simulation results for specific relationships.

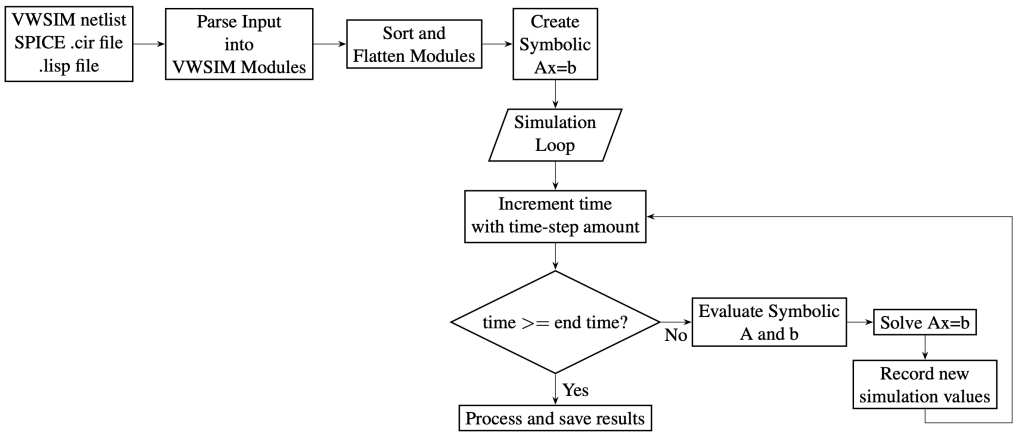

Figure 3.3: VWSIM flowchart

The following sections discuss a few of the steps mentioned in the VWSIM workflow in more detail.

3.4.2 Constructing the symbolic matrix equation

In Step 5 of the VWSIM workflow, the flattened netlist is converted into a matrix equation. The circuit analysis approach used by VWSIM to perform the conversion from netlist to matrix equation is Modified Nodal Analysis (MNA) [3].

The matrix equation is constructed by appealing to Kirchhoff's Current Law (KCL). KCL states that the sum of all the currents entering and exiting a "node" (a wire connecting two or more devices) must be zero. MNA states that the system of equations, and hence the matrix equation, will be constructed by applying KCL

to every node in the circuit, and by including additional equations that helps to solve for unknowns in each of the KCL equations. The reason we need these extra unknowns is that the current through a device can be complicated and depend on additional variables (such as the phase across the device). Indeed, some of these device current equations are non-linear, involving derivatives and integrals, which makes constructing a system of linear equations difficult. As suggested in [3] and [5], we linearize (and discretize) these equations by using an integration scheme, the trapezoidal integration method. Thus, VWSIM performs a sequential, transient analysis that starts with all variables, and their derivatives, initialized at 0, and then repeatedly advances the time in small increments to repeatedly solve for the unknown variables and their derivatives.

So, given the list of devices, their wire connections, and their component values (ex. 1 Ohm, 2/5 Farads), we can write down a system of equations based on the mathematical circuit-theoretic properties of each device. Since we have a system of linearized equations (as discussed above), we can construct a matrix equation ($Ax = b$). VWSIM has two simulation modes: voltage and phase. In the first mode, the unknowns in the x vector include the voltages of the nodes, and in the latter mode the unknowns in the x vector include the phases of the nodes (with respect to a reference node). Meanwhile, the A matrix and the b vectors generated will be symbolic; that is, the entries in the A and b vector will be mathematical expressions, which are called *terms*, that can be evaluated given numerical assignments to variables in the term.

VWSIM provides a language to specify mathematical terms (expressions). Syntactically, a term is one of the following:

1. Quoted constant
2. Symbol constant
3. Symbol variable

4. Function Call

A quoted constant is a Lisp `QUOTE` (') followed by a rational or double-precision floating point number. An ACL2 symbol constant is a Lisp `SYMBOL` that has * as the first and last character. A symbol variable is a Lisp `SYMBOL` that denotes a variable value that will be looked up at simulation time. `$time$` and `$hn$` are special symbol variables. `$time$` is the current simulation time as a floating point number. `$hn$` is the current simulation step size as a floating point number. A function call is of the form, `(fun-name arg0 arg1 ...)`, where fun-name is a supported floating-point arithmetic operator, such as `f+, f-, f-sin, ...`, boolean operator, such as `f-not, f=, f<`, or conditional, `if`[3]. Some examples of terms that can be evaluated by the VWSIM term evaluator are shown below.

```
'3
(f+ '1 '2)
(f+ '1.2d0 *pi*)
(f* '2 $hn$)
(if (f-not (f< '1 *phi0*)) '4 '5)
(if ($time$< '2/100) '1.0d0 '2.0d0)
```

Consider the following flattened description of that circuit – since the RC circuit was not hierarchical, the flattened version of the circuit is simply the list of its devices. To exemplify the MNA process, we will construct and inspect a symbolic matrix equation for this circuit.

```
(defconst *rc-flat-netlist*
 ; Name type  connections branch   value
```

[3]The complete VWSIM language is described in the VWSIM documentation: `https://www.cs.utexas.edu/users/moore/acl2/manuals/latest/?topic=ACL2____VWSIM-TERM`

```
'((v1    v    (vs1 gnd)  (i-v1)   ((if ($time$< '1/5) '0 '1)))
  (r1    r    (vs1 vc1)  (i-r1)   ('1))
  (c1    c    (vc1 gnd)  (i-c1)   ('1))))
```

Using the MNA technique, the RC circuit can then be converted to a matrix equation to solve for the voltages and currents in the circuit. Using the voltage-current equations for the resistor and capacitor, and choosing GND to be the reference node (value of 0), the matrix equation in 3.1 can be algorithmically generated by VWSIM. The MNA approach is used, where each device *stamps* its equations and constraints into the matrix equation [3]. The resulting symbolic A, x, b are shown in equation (3.1) below.

$$\mathbf{A}$$

$$\begin{bmatrix} \text{'0} & \text{'0} & \text{'1} & \text{'0} \\ \text{'0} & \text{(f0- (f/ \$hn\$ (f* '2 '1)))} & \text{'0} & \text{'1} \\ \text{'1} & \text{'0} & \text{(f-1/x '1)} & \text{(f0- (f-1/x '1))} \\ \text{'0} & \text{'1} & \text{(f0- (f-1/x '1))} & \text{(f-1/x '1)} \end{bmatrix}$$

$$\mathbf{x}$$

$$\begin{bmatrix} \text{i-v1} \\ \text{i-c1} \\ \text{vs1} \\ \text{vc1} \end{bmatrix}$$

$$\mathbf{b}$$

$$\begin{bmatrix} \text{(if (\$time<\$ '1/5) '0 '1)} \\ \text{(f+ (f- vc1 gnd) (f* (f/ \$hn\$ (f* '2 '1)) i-c1))} \\ \text{'0} \\ \text{'0} \end{bmatrix} \tag{3.1}$$

Notice that the symbolic matrices have terms in their entries. These terms will be evaluated at every time step so that we obtain a numerical matrix equation, which can then be solved for x. The term evaluation process is described in more detail in the next section. Notice that in the equation above, the simulation step size, `$hn$`, was used to linearize the voltage-current relation for the capacitor ($I = C * dV/dt$).

3.4.3 Evaluating and solving the matrix equation

After VWSIM generates a symbolic matrix for the circuit model, the simulation begins. VWSIM initializes the simulation start time as well as values for all of the simulation variables – node voltages/phases, device currents, and derivatives of these variables – at that time. The initialization value depends on the state of the simulation. If this is a new simulation, then VWSIM currently initializes all of these values to 0. If VWSIM is continuing a saved simulation, then the values are initialized to their values at the end of the previous simulation.

After initialization, the simulation time is incremented by a given time step size. The terms in the symbolic $Ax = b$ equation are evaluated and the resulting numerical equation is solved for x. These new x values are recorded as the values of the simulation variables at the current time the simulation. The process is repeated until the end of simulation time.

The terms in the symbolic matrix equation are evaluated using the VWSIM term evaluator. The evaluator takes as input the term to be evaluated, the current simulation time, and a map of variables to numerical values (the values recorded from the previous simulation time step). All of the terms in the A matrix and b vector are completely evaluated using the evaluator. The resulting numerical matrix equation is then solved for x using the simulator's Gaussian elimination solver.

3.5 Optimizations

Our effort involved the initial development of a simple, but slow simulator that we used to make sure that our approach was sound. We represented arrays as lists of lists. We used rational numbers for time and simulation state variables. We kept the relations in a symbolic form so we could read the equations produced by the MNA method. This approach allowed us to understand the operation of this kind of simulator thoroughly, but the execution performance was abysmal. The MNA method produces a matrix that is related to the size of the circuit, where two or three equations are needed for each component; transformers and delay lines require additional equations. Solving for x in $Ax = b$ is cubic in the dimension of x. After simulating circuits with a few tens of components, we were waiting for results that never appeared. This is not surprising considering we were using linear lists to represent our arrays; we started this way because it was simple.

We then began an incremental replacement of our list-based data structures for A with a sparse matrix. This change also caused a complete update of our solving code so it could operate correctly on data represented in a sparse matrix. In turn, each of these changes required increasingly difficult ACL2-"guard" proofs. An ACL2 "guard" is a mechanism that restricts a function to a certain domain. A guard proof ensures that the guarded recursive or non-recursive function remains in that domain. This allows an ACL2 user to specify types and much more on input arguments to a function, thus allowing an opportunity to increase execution efficiency. In fact, these guard proofs continue as we attempt to prove the guards of a sparse matrix solving code that is coded with ACL2's second-order `LOOP$ FOR` iteration constructs.

In order to handle increasingly larger circuit descriptions, several optimizations were implemented to reduce the time and memory consumption of a simulation.

Sparse Matrix Representation of Symbolic Equations: when circuit descriptions are converted to the equation $Ax = b$, the A matrix is usually very sparse. In order to leverage this property, the simulator only stores the non-zero

entries in each row of the A matrix. The A matrix is an array, where each entry (row) is an association list of pairs with a column index and a value. For example, `'((0 . 1) (5 . 2))` is row with a value of `1` in column `0`, `2` in column `5`, and `0` in all other columns. This matrix storage format reduces the amount of memory required drastically, while also decreasing the access and update times for each entry in the A matrix.

Using STOBJs to store the simulation state: Single-threaded objects, STOBJs, enable VWSIM to persistently store the simulation state, which allows for saving and resuming simulations, filtering and analyzing results, and more. Additionally, STOBJs provide support for arrays in ACL2. VWSIM makes extensive use of arrays to enjoy constant-time access and storage. The A matrix, b vector, the simulation output, and symbolic term evaluation results are all stored in arrays. This makes accessing and updating the simulation state faster than linked-list-style structures.

Symbolic subterm evaluation: to perform a simulation, all unique symbolic entries (terms) in the A and b matrices are gathered. From these symbolic terms, a list of all of their unique subterms is generated. For example, the unique subterms of the term `'(f* (f- '4 y) (f- '4 y))` are `Y, '4, (F- '4 Y), and (F* (F- '4 Y) (F- '4 Y))`. The list of all subterms are ordered so that each term's subterms are before it. This enables efficient evaluation of all unique terms in one sweep, and no term is evaluated more than once. The results are stored in the array field of the `ST-VALS` STOBJ. Hence, when the symbolic A and b matrices are being evaluated, each entry's evaluated value is quickly accessed from the array.

Fast, sparse-matrix solver: The first version of VWSIM used a rational-arithmetic-based solver that did not leverage the sparsity of circuit simulation matrices. We have written a series of linear equation solvers since then to solve $Ax = b$. The reason we wrote a series of them was to get a solver that was fast enough for large sparse matrices. We are, of course, aware that there are many extraordinarily fast linear equation solvers. We wrote one in ACL2 because ultimately we wish to prove

it correct. Our current solver is a Gaussian elimination solver. Provided a solution exists, Gaussian elimination reduces the set of equations to triangular form without changing the solution. Then, we can use back substitution to compute the solution, starting at the bottom-most row and working up.

Our current solver is just an optimization of this basic method. It uses a STOBJ to store A, x, and b as arrays. We do not augment A with b but instead put A into triangular form and then compute from it a program that says how to solve for x given any b. This allows us to do most of the work just once and then use the triangular form repeatedly for the different b's generated by the simulator. We are currently verifying the guards of this ACL2 solver.

Floating-point approximation: The native ACL2 rational arithmetic provides arbitrary precision at the cost of performance. As the lengths of simulations increase, the number of operations performed on the rational values increases. Since the precision of rational numbers is unbounded, the numerators and denominators continue growing, which increases memory usage and decreases performance. Additionally, VWSIM makes use of several approximation techniques (trapezoidal integration method), irrational numbers (π, e), and transcendental functions ($\sin(x)$, e^x), for which the use of arbitrary precision is superfluous. Thus, VWSIM makes use of double-precision, floating-point arithmetic to take advantage of fast, hardware-supported floating-point operations.

Chapter 4: Analysis and Results

In our efforts to understand Rapid, Single-Flux, Quantum technology, we created VWSIM as a means to interact with the circuit simulation to extract key properties and insights. Several design decisions were informed by this goal: symbolic matrices, symbolic term evaluation, ability to interact with and modify the simulation, a LISP-based solver, and, of course, developing this simulator in a formal language – ACL2. Some of these features make it difficult for VWSIM to perform as efficiently as other simulators, such as JoSIM. In fact, VWSIM simulates circuits at roughly a quarter of the speed of JoSIM. However, all of the aforementioned features enable us to perform the types of RSFQ modeling and analysis that we found lacking in other simulators. In this chapter, we walk through an example of using VWSIM to demonstrate and discuss some of the analytical capabilities available to the user.

4.1 An example circuit simulation

In this example, we will model, simulate, and analyze a Rapid Single Flux Quantum (RSFQ) "D-latch" circuit. A D-latch offers a mechanism to store a single fluxon (bit) in the RSFQ logic. The D-latch we define will have two input wires (D, C) and one output wire (Out). The D input requests the latch to store a fluxon. The C input requests the latch to release the stored fluxon through Out. If there is no fluxon stored in the latch, then a fluxon is not released through Out.

4.1.1 Simulation of a D-latch circuit

We will simulate the circuit in Figure 4.1. This top-level circuit, which tests the D-latch, provides fluxons through pulse generators (voltage sources). The generators are connected to Josephson Transmission Lines (JTLs), which transmit fluxons to the C and D inputs of the D-latch. The D-latch Out output wire is also connected to a

JTL, which transmits the output fluxons to a resistor. The schematic diagram of the D-latch subcircuit is shown in Figure 4.2. The full SPICE description of the tester circuit is in Chapter 5.2.

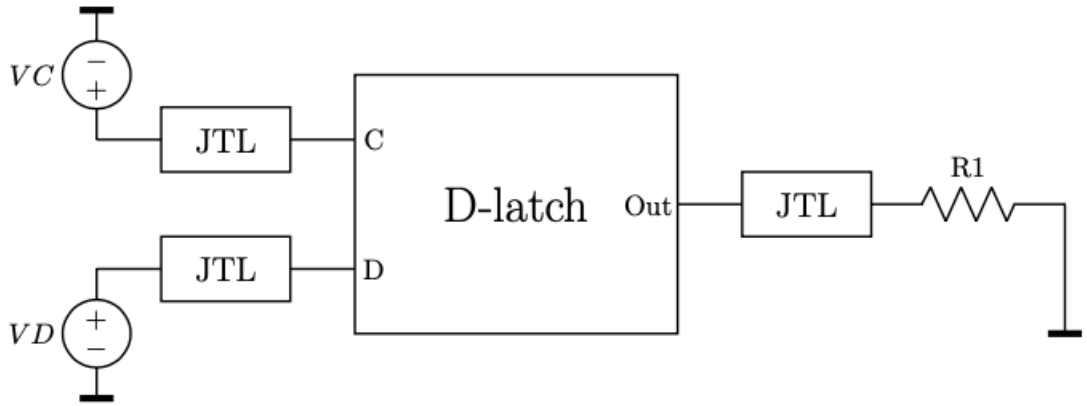

Figure 4.1: Schematic of Top-Level D-Latch Test Circuit

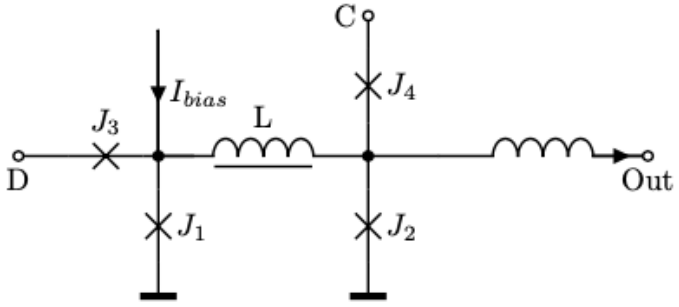

Figure 4.2: Schematic of D-Latch

We will simulate the D-latch tester circuit from 0 seconds to 100 picoseconds with a step size of 1/10 of a picosecond.

VWSIM can simulate the circuit using the following command:

```
(vwsim "Testing/test-circuits/cirs/d-latch.air"
```

```
:time-step (* 1/10 *pico*)
:time-stop (* 100 *pico*))
```

The arguments to the command above are as follows. The first argument to VWSIM is the file that contains the SPICE description of the D-latch circuit. The second argument is the time-step size, `:time-step`. For the time step size, we provide a LISP expression that will evaluate to a number. In this case, `(* 1/10 *pico*)` evaluates to 1/10 of a picosecond. VWSIM has several in-built constants including unit prefixes and fundamental physical and mathematical constants. Note that instead of `(* 1/10 *pico*)`, we could have provided the number `1/10000000000000`. The final argument is the time to stop the simulation, `:time-stop`, which is 100 picoseconds. The command above will then read the specified VWSIM circuit model, and attempt to parse, flatten, initialize, and simulate the D-latch model. If there is any processing error, VWSIM will stop the process at that point.

4.1.2 Save and Plot Simulation Results

As the D-latch circuit model is simulated, VWSIM stores the simulation values for each time step. Upon completion, we can request specific simulation results. We would like to inspect whether our D-latch is working as intended. We can look at the current through the quantizing inductor (L) in the D-latch. When there is a fluxon (bit) in the D-latch, there will be a larger current circulating in the `J1-L-J2` loop than when there is no fluxon stored in the latch. Using the `vw-output` command, we can inspect the current through inductor `L` in the D-latch.

Using the `vw-output` command, we will write the current through inductor L to a comma-separated values (CSV) file.

```
(vw-output '((DEVI . LL/XD_LATCH))
           :output-file "Testing/test-circuits/csvs/d-latch.csv")
```

VWSIM provides the `vw-plot` command to run the gnuplot plotting program from the ACL2 prompt. Here, we plot the phase across each JJ with respect to time. The output of the command is in Figure 4.3.

```
(vw-plot '((DEVI . LL/XD_LATCH))
         "Testing/test-circuits/csvs/d-latch.csv")
```

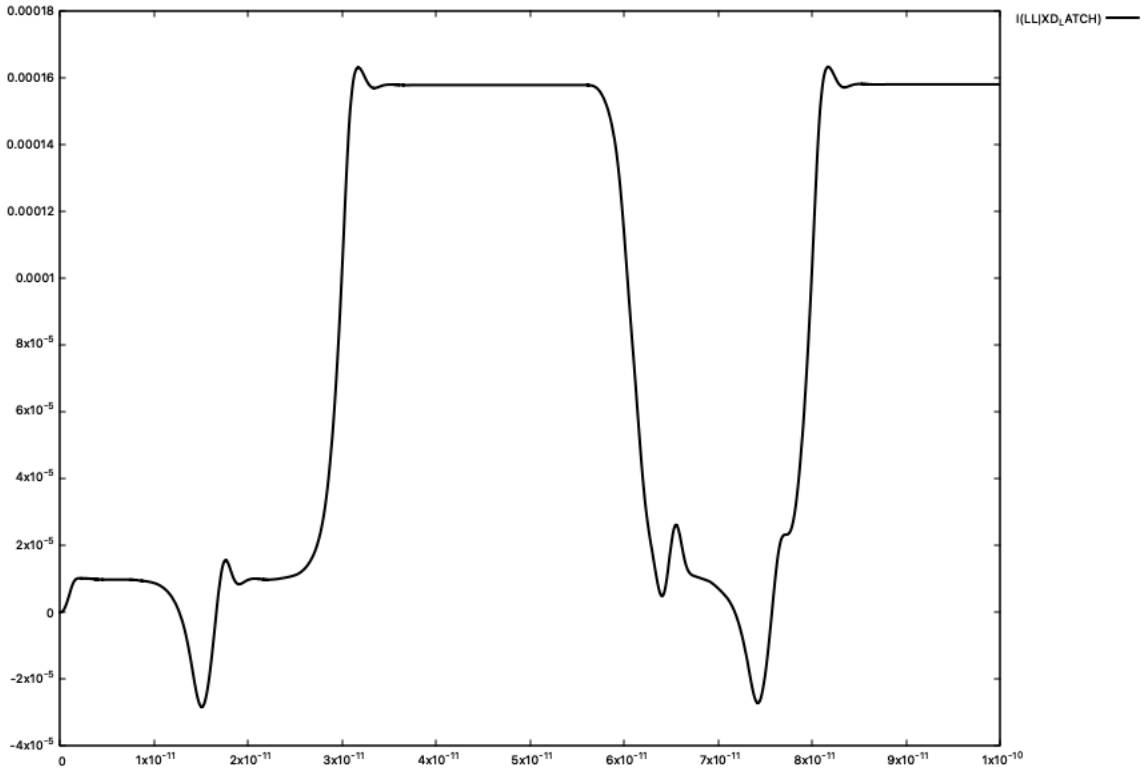

Figure 4.3: Plot of circulating current in the D-Latch loop

4.1.3 Analyze Simulation Results

Since all of the simulation results are available after a successful simulation, we can perform analysis on the circuit. We will identify when a fluxon passes through the Josephson junction `J1` in the D-latch, which suggests that a fluxon is about to be stored in the D-latch `J1-L-J2` loop.

We may wish to inspect the phase across J1 at all simulation times and determine when a fluxon passes through the JJ. This will be when the phase across the JJ has stepped up or down by a multiple of 2π (i.e. when the phase across the JJ has completed a full "turn" around the unit circle). To perform this analysis, consider the two following ACL2 function definitions.

```
(defun detect-fluxon-through-jj-recur (jj-phases times turns)
  ;; check if list of phases is empty
  (if (atom jj-phases)
      nil
    (let* (;; get next phase across the JJ
           (phase (car jj-phases))
           (2pi (* 2 (f*pi*)))
           (new-turns (truncate phase 2pi)))
      (if ;; detect a step up/down in 2pi phase across the JJ
          (not (= new-turns turns))
          ;; full fluxon detected
          (cons (car times)
                (detect-fluxon-through-jj-recur
                 (cdr jj-phases) (cdr times) new-turns))
        ;; full fluxon through the jj yet
        (detect-fluxon-through-jj-recur
         (cdr jj-phases) (cdr times) turns)))))

(defun detect-fluxon-through-jj (jj-phases times)
  ;; check if list of phases is empty
  (if (atom jj-phases)
      nil
    (let* ((2pi (* 2 (f*pi*)))
```

```
        ;; calculate how many times the JJ has already been turned
        ;; by 2pi radians
        (initial-phase (car jj-phases))
        (initial-turn (truncate initial-phase 2pi)))
    ;; find all times a fluxon passed through the JJ (i.e. the JJ
    ;; turned)
    (detect-fluxon-through-jj-recur
     (cdr jj-phases) (cdr times) initial-turn))))
```

The `detect-fluxon-through-jj` function takes two input arguments: a list of the phases across the JJ for each simulation time step and a list of the simulation times for each simulation time step. The function first determines how many times the phase across the JJ has already been turned at the beginning of the simulation. Then, it calls the helper function, which recurs through the rest of the phases and records each time the phase across the JJ completes a full turn.

We need to provide the phase across J1 for each time step and the list of simulation times for each time step to our `detect-fluxon-through-jj` function. We will access the phase values from the output alist produced by the `vw-output` command using the `vw-assoc` command. `vw-assoc` takes as input a key-value pair (in this case, we are requesting the phase across `J1`) and an alist produced by `vw-output`. It then returns the requested key along with its simulation values, if the key exists in the alist, otherwise it returns NIL. We can take the `cdr` of the list produced by vw-assoc to remove the key from the front of the list.

```
(vw-output '((PHASE . BJi/XJ1/XD_LATCH))
           :save-var 'loop-phases)

(detect-fluxon-through-jj
 (cdr (vw-assoc '(PHASE . BJi/XJ1/XD_LATCH) (@ loop-phases)))
 (cdr (vw-assoc '$time$ (@ loop-phases))))
```

which results in

```
(3.11e-11 8.11e-11)
```

This means that, during the simulation, a fluxon passed through J1 at 31 picoseconds and 81 picoseconds. We can check whether this is what we expected. The pulse generators produce a pulse on the D input at 20 picoseconds and 70 picoseconds, and on the C input at 5 picoseconds and 55 picoseconds. This sequence requests the latch to be emptied at 5 picoseconds, filled at 20 picoseconds, emptied at 55 picoseconds, and filled at 70 picoseconds. Our analysis is consistent with this description. A fluxon passed through J1 every time the latch was filled (20ps and 70ps). Note: the 11 picosecond delay from request to the fluxon passing through J1 is primarily due to the time it takes for the fluxons to pass through the JTLs before reaching the latch.

Chapter 5: Conclusion

5.1 Future Work

The development of VWSIM may have spawned more ACL2 development work than the core effort itself. To achieve useful simulation performance, we permitted the storage and use of floating-point numbers. In addition, our matrix-solve code is based on the ACL2 `LOOP$` construct for which we have little experience; guard proofs for the loop-with-loop, matrix manipulation code is complex and there is not any available library support.

With respect to performance, VWSIM is much slower than JoSIM. We have attempted to address this, but some issues are deeper than even the definition of the ACL2 system – they have to do with Lisp systems on which ACL2 is implemented. It is difficult to arrange for double-precision (64-bit), floating-point numbers to be used without them being *boxed* by the underlying Lisp systems on which ACL2 operates. The primary Lisp systems we use are Clozure Common Lisp (originally named Coral Common Lisp) (CCL [2]) and Steel Bank Common Lisp (SBCL [12]). On 64-bit systems, these two Lisp systems use part of 64-bit words (4 bits on CCL and 1 bit on SBCL) to indicate a (partial) data type. Because double-precision, floating-point numbers are defined to use 64 bits, additional storage must be allocated to indicate that a floating-point number is being referenced. Providing thorough type declarations helps, but much of this *boxing* activity occurs deep within host Lisp systems and beyond an ACL2 user's ability to control.

The ACL2 system authors (Kaufmann and Moore) are investigating the addition of double-precision, floating-point numbers as first-class ACL2 data objects. Such floating-point numbers will be forbidden to appear in a list, but they will be allowed to be stored/retrieved from STOBJs where such objects have been declared to be floating-point numbers. Further declarations will likely be needed when pass-

ing/returning floating-point numbers as arguments/return-values to/from ACL2 functions. When using floating-point numbers in `LET` and `LET*` expressions, additional declarations will be required. Any initial inclusion of floating-point numbers into ACL2 will require careful declaration and will not, initially, include much, if anything, in the way of semantics. For instance, negating the value of a floating-point number may be known to produce the negative of the original value, but the associativity of addition for three floating-point values will not be a theorem because there are many examples where it is not valid.

With respect to performance-oriented operations we can control, we can improve the performance of symbolic expression evaluator: currently we have eliminated duplicated evaluation, but we do not perform lazy evaluation of "if" expressions. We can improve the access speed of the simulation values we store and retrieve.

With respect to verification-oriented goals, we want to develop a means to specify and perform symbolic verification of RSFQ-based circuits. The cost of simulation in the number of circuit elements grows with the cube of the model size. Any such specification and verification system for RSFQ logic will need some kind of abstract way to specify RSFQ circuit behaviors. Presently, the VWSIM simulation operates at the level of circuit components (e.g., inductors, JJs), and the level of detail is too great to allow simulation of even medium-sized systems. We have been simulating many RSFQ circuits (transmission lines, latches, SFQ-DC converters, transformers, etc.) to figure out abstractions for RSFQ circuit models. For example, these abstractions could be timing diagrams that record the firing of JJs and the movement of fluxons (bits) in the circuit. We hope that these abstractions will elevate us to a digital logic, which would be easier to reason about than double-precision floating-point voltage, current, and phase values.

VWSIM could also be extended to perform parameter tuning and optimization. In VWSIM, simulation component values (e.g. resistor values, inductor values, Josephson junction areas) can be represented as symbolic variables that are instan-

tiated during a simulation. Using this instantiation mechanism, across multiple simulations of a circuit, margins can be determined for important user or heuristically-chosen circuit components. Building on this, using a cost function, such as reducing the area occupied by resistors on a chip, VWSIM could even be programmed to find the optimal circuit parameter values in the margin.

VWSIM is also uniquely positioned to perform partial, or even purely symbolic circuit simulation. During simulation, a symbolic system of equations is already created for the circuit model. The system can be simplified and possibly solved in terms of symbolic variables and their constraints. For example, these constraints could specify that a variable is always non-zero or even provide the variable's possible range of numerical values. This would allow us to perform analyses that could produce stronger results than numerical simulation. For example, the lower- and upper-bounds for the current through a device, the optimal inductance values for an inductor in an RSFQ quantizing loop, and whether the voltage across a Josephson junction remains in its expected operating range. Enabling symbolic simulation, however, would certainly be a difficult task. To name a few challenges, the solver would need to perform a symbolic Gaussian elimination, which may not always be possible depending on the constraints of the variables in the matrix. Additionally, we would need to formally model the numerical simulation values as infinite-precision numbers (ex. rational numbers or real numbers), instead of limited-precision floating-point numbers.

5.2 Summary

The development of VWSIM was very helpful in clarifying our understanding of the modeling and simulation of Josephson junction electrical-circuit models. Written in ACL2, several termination, memory safety, and logical consistency properties have been proven about the operation of VWSIM. Had we developed VWSIM in C, we are confident that we would still be debugging its operation. ACL2 also makes

interactive use of the simulator available through its read-eval-print-loop (REPL). The interactive VWSIM commands enable the user to access data through the REPL and perform complex, programmable analyses on circuit models.

During the development of VWSIM, simulation results were often compared to those produced by JoSIM. As we discovered discrepancies, we identified code that needed updating; this study, the development of a formal language for specifying JJ-based circuits, and the many RSFQ circuits we have interactively simulated and analyzed with the VWSIM simulator have improved our understanding of RSFQ circuits. This work will be instrumental in our effort to identify abstractions that will help us formally verify the operation of RSFQ circuits models.

Appendix

The following SPICE description is for a D-latch tester circuit. The description includes subcircuits for the Josephson Transmission Line (JTL) and RSFQ D-latch.

```
* A hand-written, SPICE description of an RSFQ D-latch.

*** The model line provides some of the Josephson junction (JJ)
*** parameters.
.model jmitll jj(rtype=1, vg=2.8mV, cap=0.07pF, r0=160, rN=16, icrit=0.1mA)

*** Overdamped JJ subcircuit

.SUBCKT damp_jj pos neg
BJi pos neg jmitll area=2.5
RRi pos neg 3
.ENDS damp_jj

*** Bias current source subcircuit

.SUBCKT bias out gnd
RR1 NN out 17
VrampSppl@0 NN gnd pwl (0 0 1p 0.0026V)
.ENDS bias

*** 4-stage JTL subcircuit

.SUBCKT jtl4 in out gnd
LL1 in net@1 2pH
XJ1 net@1 gnd damp_jj
```

```
Xbias1 net@1 gnd bias

LL2 net@1 net@2 2pH

XJ2 net@2 gnd damp_jj
Xbias2 net@2 gnd bias
LL3 net@2 net@3 2pH

XJ3 net@3 gnd damp_jj
Xbias3 net@3 gnd bias
LL4 net@3 net@4 2pH

XJ4 net@4 gnd damp_jj
Xbias4 net@4 gnd bias
LL5 net@4 out 2pH

.ENDS jtl4

*** D Latch circuit

.SUBCKT D_latch D C out gnd
XJ3 D net@1 damp_jj
XJ1 net@1 gnd damp_jj
Xbias1 net@1 gnd bias

LL net@1 net@2 12pH

XJ4 C net@2 damp_jj
XJ2 net@2 gnd damp_jj
```

```
LY net@2 out 2pH
.ENDS D_latch

*** TOP LEVEL CIRCUIT

* Fluxon pulses at 20p, 70p, ...
VD D gnd pulse (0 0.6893mV 20p 1p 1p 2p 50p)
Xjtl4d D rD gnd jtl4

* Fluxon pulses at 5p, 55p, ...
VC C gnd pulse (0 0.6893mV 5p 1p 1p 2p 50p)
Xjtl4c C rC gnd jtl4

* Create instance of D latch subcircuit
XD_latch rD rC out gnd D_latch

* Output JTL
Xjtl_latch out out2 gnd jtl4
RR1 out2 gnd 5

.END
```

Works Cited

[1] Bishop C. Brock and Warren A. Hunt, Jr. The `DUAL-EVAL` Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor. *Formal Methods in Systems Design*, 11:71–105, 1997.

[2] CCL: Clozure Common Lisp.

[3] Chung-wen Ho, Albert E. Ruehli, and Pierce A. Brennan. The Modified Nodal Approach to Network Analysis. *IEEE Tranactions on Circuits and Systems*, 22(6), June 1975.

[4] Timothy A. Davis. The SuiteSparse Documentation.

[5] Johannes Arnoldus Delport, Kyle Jackman, Paul le Roux, and Coenrad Johann Fourie. Josim—superconductor spice simulator. *IEEE Transactions on Applied Superconductivity*, 29(5):1–5, 2019.

[6] Warren A Hunt Jr, Vivek Ramanathan, and J Strother Moore. VWSIM: A circuit simulator. *Electronic Proceedings in Theoretical Computer Science*, 359:61–75, may 2022.

[7] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods. Springer US, 2000.

[8] Matt Kaufmann and Moore, J S. The ACL2 System Documentation.

[9] Microprocessor Standards Committee of the IEEE Computer Society. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[10] Terry P Orlando and Kevin A Dalin. *Foundations of Applied Superconductivity*. Addison-Wesley Publishing Company, Inc., 1991.

[11] Paul Bunyk, Konstantin Likharev, and Dmitry Zinoviev. RSFQ Technology: Physics and Devices. *International Journal of High Speed Electronics and Systems*, 11(1), 2001.

[12] SBCL: Steel Bank Common Lisp.

[13] Pavel Shevchenko. *PSCAN2 Superconductor Circuit Simulator*, October 2018.

[14] Steven M. Rubin. SPICE – Simulation Program with Integrated Circuit Emphasis.

[15] Warren A. Hunt, Jr. Mechanical Mathematical Methods for Microprocessor Verification. *Computer-Aided Verification Conference, Lecture Notes in Computer Science*, July 2004.

[16] Stephen R. Whitely. *WR*spice *Reference Manual*, January 2021.

[17] Wikipedia. SPICE – Simulation Program with Integrated Circuit Emphasis, 2022.