SMT Solver With Hardware Acceleration

Yean-Ru Chen, *Member, IEEE*, Si-Han Chen, and Shang-Wei Lin

***Abstract*—Satisfiability modulo theories (SMTs), an extension of Boolean satisfiability (SAT) problem, is widely used in many application domains because of its rich expressiveness. Thus, there are many works trying to speedup the process of SAT/SMT solving. In this work, we develop a framework by proposing a new hardware architecture to solve the SMT problem for the theory of quantifier free linear real arithmetic (QF-LRA) to speedup the SMT solving process. The new proposed architecture framework consists of a hardware SAT solver and a hardware Simplex solver. Our hardware SAT solver has an optimized Boolean constraint propagation process with a pipeline structure and a nonchronological backtracking mechanism, while our hardware Simplex solver supports parallel operation flow inside the Simplex iteration to execute the selection operations parallelly with the pivot operation and the row selection mechanism to avoid unnecessary row computation and increase the resource utilization. According to our experimental results, the proposed framework can achieve from 2.539 up to 1561.181 times speedup compared with software SMT solvers in our selected 40 benchmarks.**

***Index Terms*—Boolean satisfiability (SAT), hardware acceleration, SAT solver, satisfiability modulo theories (SMTs), Simplex algorithm.**

I. Introduction

GIVEN a Boolean formula over a set of Boolean variables, the Boolean satisfiability problem (usually abbreviated as SAT) is the problem of determining whether there exists an interpretation (i.e., an assignment to the set of Boolean variables) that satisfies the given Boolean formula. SAT plays a very important role in either theoretical computer science or practical applications because many problems can be translated into (or encoded as) SAT problems such that once a solution is found in the (translated) SAT problem, its original corresponding problem is also solved accordingly. With this correspondence, improving the performance of SAT solvers is equivalent to improving the ability to solve the original problem. Thus, there are many works trying to speedup the process of SAT solving, including implementing SAT solvers in hardware [17], [21], [23].

Satisfiability modulo theories (SMTs), an extension of the SAT problem, is to determine the SAT of a first-order formula with respect to decidable first-order theories. Unlike the SAT problem, in which the variables can only be of Boolean type, the atoms in the SMT problem can be of any type depending on the theories that the SMT formula is based on. With this extended expressiveness, SMT can provide not only richer representation for more real-world problems but also a more efficient abstraction level compared with SAT. Because of its richer expressiveness, SMT is utilized in plenty of application domains, especially in formal verification, ranging from recent topics like smart contract [4] and neural network [18] to register-transfer level (RTL) circuit design [8], like hardware Trojan detection [15] and test generation of using concolic testing [3]. Following the same philosophy of the fact that improving the performance of SMT solvers is equivalent to improving the ability to tackle the above formal verification problems, the goal of this work is to speedup the SMT solving process by constructing an SMT solver with hardware acceleration.

In this work, we focus on the theory of *quantifier free linear real arithmetic* (QF-LRA), which is widely used in practice, e.g., [4] and [18]. The traditional solving procedure of the QF-LRA SMT problem is implemented by an SAT solver together with a Simplex [11] based theory solver. The works of [17], [21], and [23] adopt hardware implementations for the SAT solver, which improves the SAT solver's performance. In addition, Bayliss *et al.* [7] implemented the Simplex solver on an FPGA to improve its performance. Inspired by these works, we believe that there exists an opportunity to provide better performance for SMT solving by adopting and combining hardware SAT solvers with hardware Simplex solvers. Here, we present a prototype of solving SMT problems for the QF-LRA theory based on proposing new architectures for both hardware SAT solver and hardware Simplex solver. The proposed hardware architecture is based on the offline lazy approach. It can achieve from 2.539 up to 1561.181 times speed up, compared with the software SMT solvers in our selected 40 test benchmarks. We summarize the contributions of this work as follows.

1) We propose a hardware architecture design consisting of a hardware SAT solver and hardware Simplex solver for solving SMT problems of the QF-LRA theory.
2) The proposed hardware SAT solver (c.f. Section IV-B) is based on the DPLL algorithm, which supports the pipelined and parallelism design of Boolean constraint propagation (BCP) as well as nonchronological backtrack mechanism for better performance.
3) The proposed hardware Simplex solver (c.f. Section IV-C) cooperates with the hardware SAT solver for SMT solving. Its design is based on the two-phase Dantzig Simplex algorithm. It adopts four pipelined multiply accumulators to provide parallel calculation, which features in the design of parallel

Manuscript received 15 February 2022; revised 19 May 2022 and 1 August 2022; accepted 9 September 2022. Date of publication 26 September 2022; date of current version 19 May 2023. This work was supported in part by MOST under Grant 110-2221-E-006-212. This article was recommended by Associate Editor L. Amaru. *(Corresponding author: Yean-Ru Chen.)*

Yean-Ru Chen and Si-Han Chen are with the Department of Electrical Engineering, National Cheng Kung University, Tainan City 701, Taiwan (e-mail: chenyr@mail.ncku.edu.tw).

Shang-Wei Lin is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.

Digital Object Identifier 10.1109/TCAD.2022.3209550

operation flow between Simplex steps to increase the resource utilization and the row selection mechanism to avoid unnecessary computations.

4) We evaluate the performance of our approach (c.f. Section V) against other three types of solving procedures with different implementation methods (hardware versus software) and different procedures (offline versus online).

The remainder of this article is organized as follows. Section II gives a brief introduction to the SAT, SMT problem, and the DPLL algorithm. In Section III, we introduce the related work of the approaches of solving the SMT problem and previous design of hardware SAT and Simplex solver. Section IV, respectively, introduces the proposed hardware architectures of the hardware SAT solver and hardware Simplex solver, including the explanations of the procedure of solving the SMT problem by using the hardware solvers. The experimental results are demonstrated and analyzed in Section V. Finally, the conclusions are presented in Section VI.

II. Preliminary

In this section, we define the necessary notations and briefly recall SAT/SMT problems and their solving techniques. Appendix A in the supplementary material is referred for details.

A. SAT Problem and SAT Solving

A Boolean variable is a variable whose value ranges over $\mathbb{B} = \{0, 1\}$. A *clause* C can be constructed by applying logic connectives, disjunction ($\vee$) and conjunction ($\wedge$), among literals. A clause is called an *unit clause* if it only has one literal. A *propositional formula* f over a set of Boolean variables V_B can be constructed by applying logic connectives among clauses. An *assignment* for a set of Boolean variables $\psi : V_B \rightarrow \mathbb{B}$ is a function assigning either 0 or 1 to each variable $v \in V_B$.

Definition 1 (SAT Problem): Given a propositional formula f over the set of Boolean variables V_B, the SAT problem is to determine whether there exists an assignment ψ (which could be partial) for V_B that makes formula f evaluate to 1. If so, the formula f is called *satisfiable*, and ψ is also called a *solution* to f. If there is no solution to f, then, f is called unsatisfiable (UNSAT).

When solving SAT problems, we usually consider formulas in conjunctive normal form (CNF) [5], in which each clause only has disjunctive literals, and clauses are connected only by conjunctions. In this article, we only consider CNF formulas.

To solve the SAT problem, as formulated in Definition 1, Davis *et al.* [12] proposed the DPLL algorithm, which constitutes the backbone of modern SAT solvers. Because of the page limit, we leave the introduction to the DPLL algorithm in Appendix A (c.f. Algorithm 1) in the supplementary material.

B. SMT Problem and SMT Solving

A problem of SMTs is an extension of an SAT problem. In an SAT problem, the formula is composed of Boolean variables, while in an SMT problem, each Boolean variable is extended as a predicate, called an atom. An atom can be formulated as $P(t_1, \ldots, t_n)$, where P is a predicate symbol and t_i is either a variable or a term constructed by applying logical connectives or functions on atoms, for $1 \leq i \leq n$. For example, in the theory of linear integer arithmetic (LIA), $(x + y > 2)$ is an atom, where x and y are variables over integers.

Different theories have different forms of atoms. Common theories considered in SMT problems include equality and uninterpreted functions (EUF), linear arithmetic (LA), difference logic (DL), bitvectors (BVs), arrays (AR), etc. In this work, we consider the LA theory, in which an atom has the following form:

$$(a_1x_1 + a_2x_2 + \cdots + a_nx_n) \odot c \tag{1}$$

where $\odot \in \{\leq, <, \neq, =, >, \geq\}$, c is a constant, and $a_1, a_2, \ldots, a_n$ are the coefficients of variables $x_1, x_2, \ldots, x_n$, respectively. If the variables range over integers, the theory is called LIA theory, while the variables range over real numbers, the theory is called linear real arithmetic (LRA) theory.

In practice, an SMT problem could have atoms of more than one theory, which enriches the expressiveness but also increases the complexity to the problem to be solved. In addition, the variables can be quantified by the universal ($\forall$) and existential ($\exists$) quantifies. A formula is called *quantifier-free*, if it does not contain any quantifiers. In this work, we focus on the quantifier-free LA (QF-LA) theory. For example, the following formula f_2 is a QF-LA formula:

$$f_2 : (x + 2y > 0) \wedge ((x + 2y < 1) \vee \neg(2x + 4y \geq -2)). \tag{2}$$

Formula f_2 has three atoms $(x + 2y > 0)$, $(x + 2y < 1)$, and $(2x + 4y \geq -2)$ over two variables x and y. We use the symbol V_{atom} to denote a set of atoms. Here, we define a *Boolean abstraction* function $\Phi : V_{\text{atom}} \rightarrow V_B$ mapping an atom to a Boolean variable and its reverse (Boolean refinement) function $\Phi^{-1} : V_B \rightarrow V_{\text{atom}}$ mapping a Boolean variable back to an atom. Intuitively, Φ abstracts every atom as a Boolean variable.

The definitions of *clause* and CNF for SMT Formulas are the same as SAT ones except that a literal in an SMT formula is an atom or its negation. Actually, an SMT formula can be transformed to an SAT formula by applying Φ on its atoms, as shown in the following example.

Example 1: Formula f_2 has a set of atoms $V_{\text{atom}} = \{(x + 2y > 0), (x + 2y < 1), (2x + 4y \geq -2)\}$. By applying Boolean abstraction, we can obtain a set of Boolean variables $V_B = \Phi(V_{\text{atom}}) = \{A, B, C\}$, where A refers to $(x + 2y > 0)$, B refers to $(x + 2y < 1)$, and C refers to $(2x + 4y \geq -2)$, respectively. Based on Boolean abstraction Φ, formula f_2 can be transformed into a Boolean formula $f_2' : (A \wedge (B \vee \neg C))$.

Thus, we extend the symbol Φ for an SMT formula f so that $\Phi(f)$ denotes the Boolean formula obtained by substituting each atom with its corresponding abstract Boolean variable. For example, $f_2' = \Phi(f_2)$ in Example 1. With the notations defined, we can formally define an SMT problem as follows.

Definition 2 (SMT Problem): Given a formula f over the set of atoms V_{atom}, the SMT problem is to determine whether there exists an assignment ψ (which could be partial) for $\Phi(V_{\text{atom}})$ that makes formula $\Phi(f)$ evaluate to 1, and the assignment is theory consistent in the corresponding atoms.

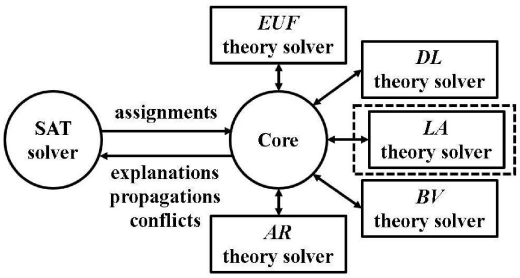

Fig. 1. Schematic of SMT solving flow [1].

If so, the formula f is called *satisfiable*; otherwise, it is called UNSAT.

The theory consistency means that the combination of atoms does not lead to any conflicts based on the theory. Let us take the LA theory for example. Two atoms $(x > 0)$ and $(x < 0)$ can be abstracted as two Boolean variables A and B, respectively. The assignment assigning 1 to A and 1 to B is not theory consistent because it is impossible that variable x is positive and negative value at the same time.

Typically, solving an SMT problem consists of two steps: 1) finding an assignment of the Boolean abstraction of the SMT formula and 2) checking whether the conjunction of the atoms is theory consistent or not.

Example 2: In Example 1, to solve the SMT problem for f_2, we first obtain its Boolean abstraction formula $\Phi(f_2) : (A \wedge (B \vee \neg C))$. In step 1, we find an assignment ψ_3 making $\Phi(f_2)$ be evaluated to 1, where $\psi_3(A) = 1$ and $\psi_3(B) = 1$. In step 2, we need to check whether this assignment is theory consistent. Consider the conjunction of atoms $(x + 2y > 0)$ and $(x + 2y < 1)$. There exist real values of x and y making both inequations hold. Thus, the assignment is theory consistent, and f_2 is QF-LRA satisfiable. However, if we consider formula f_2 in the QF-LIA theory, it is not satisfiable because among the eight possible assignments for Boolean variables A, B, and C, there is no one which is theory consistent.

In general, given an SMT formula f, finding an assignment to satisfy formula $\Phi(f)$ can be done by using an SAT solver, which is often implemented using the DPLL algorithm. However, an extra solver usually called the *theory solver*, is required to perform the theory consistency checking. The schematic of SMT solving can be depicted in Fig. 1, where the *core* is used to distribute each atom to the appropriate theory solver. For different theories, different theory solver is required. For the LRA theory, the common algorithms that are used to implement the LRA theory solver are the *Simplex* algorithm and the *Fourier–Motzkin* algorithm. Most of the well-known software SMT solver, like Z3 [13], MathSAT [9], and CVC4 [6] adopt variants of Simplex algorithm to implement the LRA theory solver.

III. Related Works

In this section, we first introduce the approaches to solve the SMT problem, next, we introduce the related works of hardware SAT and Simplex solver.

A. Approaches of Solving the SMT Problem

The approach of solving the SMT problem can be separated into two categories: 1) eager approach and 2) lazy approach [22].

For eager approach, its concept is to encode the SMT problem into an equivalently satisfiable Boolean formula, and the formula can be solved by the SAT solver. For example, by applying small-domain encoding for the DL theory, the variables with the value range N can be encoded by using N or lbN Boolean variables. The eager approach can get the benefit of high performance SAT solver for advantage, but the size of the Boolean formula will grow exponentially instead, and then, it will become inefficient to solve if the Boolean formula grows too large.

On the other hand, the lazy approach solves the SMT problem by the solver that is the integration of an SAT solver and one or more theory solvers. In the lazy approach, the SAT solver is in charge of enumerating truth assignments that can satisfy the Boolean abstraction of the input SMT problem, and the theory solver is to check the consistency in the theory of the set of literals corresponding to the enumerated assignments. Most state-of-the-art SMT solvers use the lazy approach to implement the solver. Moreover, the lazy approach can be further separated into two kinds of procedures, offline and online procedures. The main difference between those two procedures is the way of integration of the SAT and the theory solver. For the offline procedure, the SAT and theory solver should be integrated independently. The SAT solver is used as a black-box solver to find a satisfiable assignment for the Boolean abstraction formula of the SMT problem and the theory solver does not participate in the SAT solving process. Once the assignment is found theory inconsistency, the negation of the current assignment will be added as a clause back to the SAT problem, and the SAT solver will start from the scratch to find another assignment. In order to provide a more efficient solving procedure, the online procedure adopts an algorithm call DPLL(T), which is a variant of the DPLL algorithm. The main difference between those two algorithms is that in DPLL(T), the theory solver will be involved in the major steps of DPLL algorithm, like Decision Making (lines 12 and 13 in Algorithm 1) [see Appendix A in the supplementary material, unit propagation (line 3), learning and backtracking (line 6)]. With the tighter integration of DPLL and theory solver, the theory solver can provide useful information for the DPLL process to generate the assignment based on the theory or prevent the process from making the decision that may cause theory inconsistency. In order to implement a theory solver that has those functions to cooperate with the DPLL(T) procedure, there are algorithms that are particularly designed for the DPLL(T) to provide more features or better performance [14]. Obviously, the online procedure has a tighter integration of the SAT and theory solving process, which can provide better performance and stability. But it also needs more implementation effort on modifying and integrating the two processes, especially on the SAT solver. On the other hand, the offline procedure is suitable for prototyping by using the existing SAT or theory solver to construct the procedure.

B. SAT Solver Implementation

The development of reconfigurable hardware SAT solver starts before 2000, the advantage of using hardware acceleration for solving the SAT problem is to provide more high-level parallelism than traditional software solver, which can evaluate and imply multiple variables at the same time.

There are two categories of the algorithm to solve the SAT problem, complete and incomplete algorithm. For incomplete algorithm, it regards SAT problem as an optimization problem and attempts to satisfy the maximum number of clauses, but it can not ensure to find a solution or conclude UNSAT if the problem is UNSAT. The well-known incomplete algorithms are GSAT (greedy local search) and WSAT (GSAT with random walk). This type of algorithm provides mass parallelism and does not need complex control, the hardware SAT solver proposed by Kanazawa and Maruyama [17] is based on WSAT algorithm.

For the complete algorithm, it can always find a solution when the problem is satisfiable or conclude UNSAT if it is UNSAT. The widely used complete algorithm is DPLL algorithm which is used in most of the state-of-the-art SAT solvers, as illustrated as follows. The work proposed by Skliarova and Ferrari [23] is an application-specific hardware SAT solver, and it consumes a lot of memory resource for backtracking procedures. Safar *et al.* [21] have added nonchronological conflict-directed backtracking and design some pipeline structures for this backtracking task. Ustaoglu *et al.* [25] proposed a hardware SAT solver that features the conflict clause learning, which can avoid the solver gets trapped in nonsolution regions of the search space, in order to provide better performance. Our SAT solver is also DPLL-based, not an application-specific solver but a general SAT solver architecture. In SAT solving, BCP procedure is most time-consuming in DPLL-based algorithm. Our main contribution in our SAT solver is to propose a new hardware architecture with parallel and pipeline BCP design to speed up the solving procedure.

C. Simplex Solver Implementation

Bayliss *et al.* [7] proposed the first hardware implementation of the Simplex algorithm. The Simplex algorithm they adopt is using the Dantzig pivot strategy. They also point out several parallelism opportunities that can apply in the Simplex iteration. For the entering variable selection, it can be done by adopting a tree of comparators for parallel comparison and the ratio test of the leaving variable selection can be performed in parallel as well. For the pivot operation, which is typically costing the most computation time in the Simplex algorithm. Its multiply–subtract operation applies on the Simplex tableau can be performed in higher level parallelism in the hardware implementation than the software. Aside from the opportunities of the intraiteration parallelism, their implementation can also allow the streaming of several problems in the pipeline of their hardware architecture. They use the 2-D block partitioning scheme to store the Simplex tableau by breaking it into several small regularly sized blocks. They also present two wordlength setups for the value representation of each element.

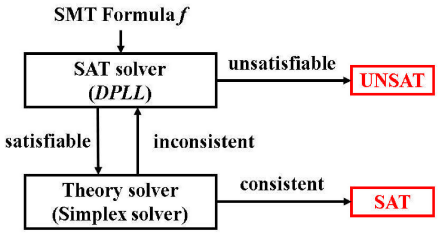

Fig. 2. Operation flow of solving QF-LRA SMT problem.

In the design implementation, they use a binary tree structure of comparators for the entering variable selection, and adopt the cross-multiplication by using a sequential multiplier for the ratio test in the leaving variable selection. They implement the FIFO buffer to store the full Simplex tableau in the form of the 2-D block. In the pivot operation step, the pivot row data would be loaded from the FIFO buffer into a fully pipelined divider to scalar the pivot element into one. Then, the pivot operation will be applied to each block, the corresponding elements from the pivot column will be latched and the elements from the divided pivot row will be retrieved for the operation. The calculation of a block of data is operated by using processing elements, each processing element contains a fully pipelined multiplier and a subtractor. They implement three kinds of sizes of the 2-D block, which represents different levels of parallelism. Our proposed Simplex solver adopts double precision floating point as data format to guarantee the value precision of the result. The solver also has the design of parallel operation flow to increase the resource utilization and row selection mechanism to avoid the unnecessary computation.

IV. Methodology

In this section, we first introduce the mechanism of solving the QF-LRA SMT problem with SAT/Simplex solver. Next, we describe our design of hardware SAT solver and our proposed hardware Simplex solver. Finally, we explain the whole operation procedure of solving QF-LRA SMT problem.

A. Solving QF-LRA SMT Problem

The procedure of solving the QF-LRA SMT problem can be separated into two parts called assignment decision and theory consistency check. They can be collaboratively achieved by SAT solver and Simplex solver, as shown in Fig. 2.

Briefly introduce here is that the SMT problem should first transformed into an SAT problem. Then, the SAT problem is solved by the SAT solver and outputs an assignment for each atom if the SAT problem is satisfiable. Second, the Simplex solver checks the theory consistency of atoms based on the given assignment. If the result of the Simplex solver is QF-LRA consistency, then, the SMT problem is satisfiable. The SMT problem is UNSAT when all of the possible assignments are not QF-LRA theory consistent.

B. SAT Solver Implementation

Now, we introduce our hardware SAT solver design used in the SMT solving procedure.

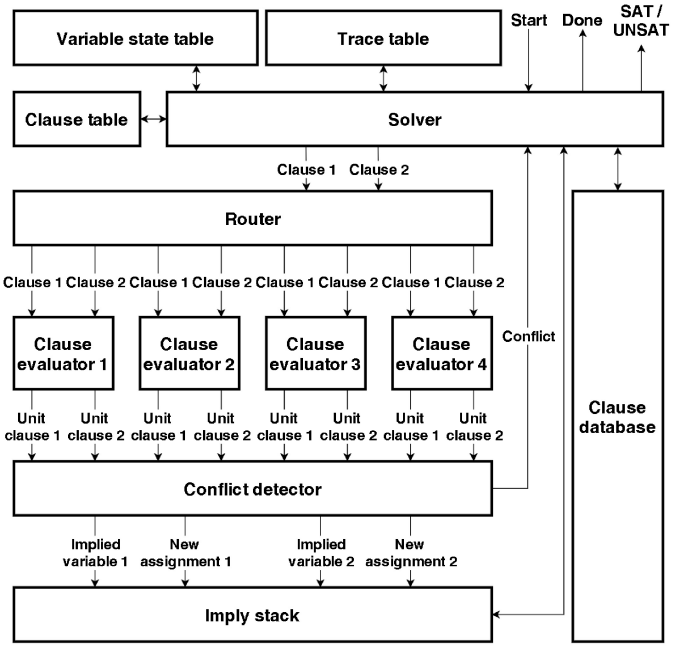

Fig. 3. Design diagram of our hardware SAT solver.

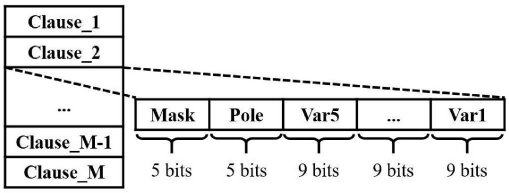

Fig. 4. Data structure of *Clause database*.

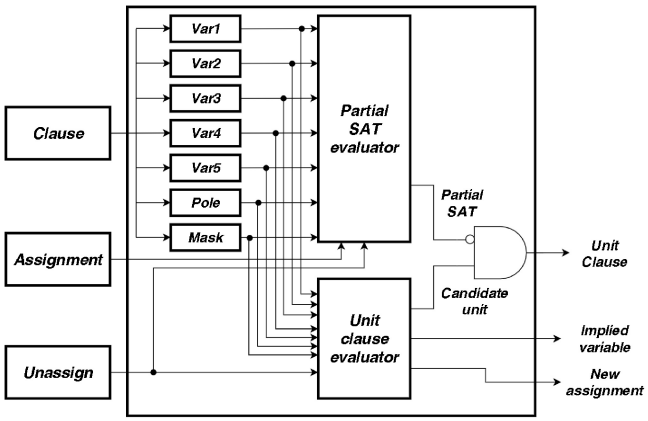

Fig. 5. Design diagram of *Sub clause evaluator*.

1) Design Concept and Settings: Our hardware SAT solver design is based on the DPLL algorithm. It supports nonchronological backtracking and contains the design of parallel BCP execution, which evaluates and detects conflict simultaneously by our designed pipelining structure. The solver can support totally up to 511 variables and 1023 clauses in 5-SAT format, i.e., the number of variables in each clause can be arbitrary to up to five variables. Fig. 3 shows the architecture of the proposed SAT solver design.

Variable state table, *Trace table*, *Clause table*, *Clause database*, and *Imply stack* are five storage units to store the information of the SAT problem or the internal data during solving procedure. To provide high-level parallelism of BCP operation, our SAT solver includes four *Clause evaluator* modules, and each of them contains two *Sub clause evaluator* modules so that it can execute eight clauses at the same time. The inputs to the *Clause evaluator* modules are organized by the *Router* module. After executing BCP, the *Conflict detector* module will be involved to check whether the current assignment exists any conflicts or not. The *Solver* module is the controller of the whole SAT solver process.

2) Design Architecture: Now, we introduce the important modules of the proposed SAT solver in detail.

1) *Variable State Table: Variable state table* is used to store three pieces of information of each variable. It identifies if the variable exists in the CNF formula or not, shows if the variable has been assigned or not, and records the assigned value and index of the variable.
2) *Clause Database: Clause database* is used to store the information of each clause that exists in the CNF formula. Fig. 4 shows its data structure. Each address represents the information of one clause and stores the index of variables contained in this clause. It also records whether the variable has negation or not (i.e., Pole), and there is also a field left for identifying the variable's validity (i.e., Mask) because the number of variables in a clause can be arbitrary (up to five variables).
3) *Clause Table: Clause table* is used to store each variable that occurs in which clauses. It is implemented with a two-port memory structure, and sets each location to store two clause indexes such that the solver can read two clauses at a time. Each variable occupies a range of locations in this table, and the start and end address of the variable should be recorded and used while accessing the clause information of the variable.
4) *Trace Table: Trace table* is the implementation of the stack ψ to store the variable assign information. This table can provide the information of variable assignment type (decide or imply) for backtracking to the previous state and de-assign the related variables. Two fields named *variable* and *value* store the index of the assigned variable and its assignment, respectively. There is also a *type* field for identifying that the variable is assigned by decision or implication.
5) *Clause Evaluator: Clause evaluator* is used to detect unit clause and generates the implication if the clause is a unit clause. Each *Clause evaluator* module contains two *sub clause evaluators*. Fig. 5 shows the design diagram of *Sub clause evaluator* module. This module takes the information of the clause and the current assignment of each variable to detect unit clause and apply implication. The output *Unit clause* denotes whether this clause is a unit clause or not. *Implied variable* and *New assignment* identify the variable and its assignment of the implication, respectively, if the clause is a unit clause. There are two modules *Partial SAT evaluator*

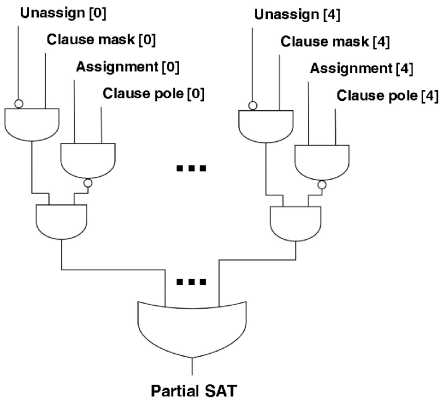

Fig. 6. Design diagram of *Partial SAT evaluator*.

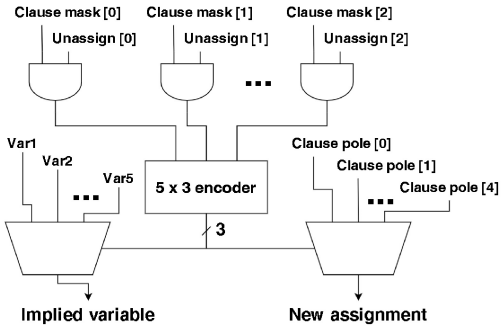

Fig. 7. Design diagram of *Unit clause evaluator*.

and *Unit clause evaluator* in the *Sub clause evaluator* module. The *Partial SAT evaluator* module is responsible for detecting whether the clause is partially satisfied. If the clause is partial SAT, then, it cannot become the candidate of a unit clause. Fig. 6 is the design diagram of the *Partial SAT evaluator* module. Fig. 7 is the design diagram of the *Unit clause evaluator* module. The module will output the assignment and the variable index of the implication if the clause is the unit clause. It first takes the assignment situation of the variables in the clause to detect whether there exists only one variable that has not been assigned yet. If that situation exists, the implication will be applied to the variable, and its index and the implied value will be outputted by the *Implied variable* and *New assignment*, respectively.

6) *Imply Stack: Imply stack* is used to store the implications because of BCP. It stores the implied variable, and the assignment of implication. Those implications will launch new BCP execution for further assignment propagation.
7) *Conflict Detector: Conflict detector* is used to apply conflict checking. Once there is a new implication after BCP operation, the *Conflict detector* module will be activated to check whether the implied variable has been implied to another assignment or not. If that situation exists, it means this BCP procedure causes conflict; otherwise, the new implied variable and its assignment will be stored in the *Trace table*.

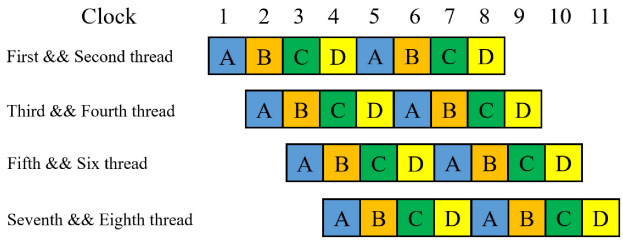

Fig. 8. Pipelined BCP procedure.

3) Pipelined BCP Procedure: In order to use the circuit resources efficiently and increase the throughput, our BCP procedure is optimized to be a four stages pipeline design. The proposed pipeline procedure is shown in Fig. 8.

1) *Stage A:* The solver accesses the *Clause table* to get the index of clauses containing the new assigned variable.
2) *Stage B:* The solver can use the clause index to get the clause information from *Clause database*.
3) *Stage C:* The clause is evaluated by *Clause evaluator* in this stage. To provide a static procedure of BCP execution, the clauses are sent to the *Clause evaluator* in sequence. Then, the clause distribution is handled by the *Router* module.
4) *Stage D:* After evaluating clauses, the solver detects conflict if there is a new implication. If no conflicts, the implied variable and its assignment will be stored into *Imply stack*.

If the *Imply stack* is not empty, it means that there still has the assignment to be propagated, and the solver will pick an implication from *Imply stack* as a new BCP task. The BCP procedure terminates if the *Imply stack* is empty or a conflict occurs. The advantage of this pipeline process is to hide the latency of memory access so that the solver does not need to stall while accessing the memory, which can increase the resource utilization.

4) Backtracking Procedure: Once a conflict occurs, the solver launches the backtracking procedure to backtrack to the previous decision in stack ψ and de-assign the related variables. To make the backtracking more efficiently, we design it as *nonchronological* method. Different from the chronological backtracking that jumps back to the most recent decision in ψ stack, the nonchronological backtracking can backtrack to the decision where the variable is most related to the detected conflict.

In the BCP operation, all variables in the clauses that have been evaluated are recorded. If the conflict occurs, these variables are the ones that may cause the conflict. Our designed nonchronological backtracking contains the following steps.

In first step, the solver determines the previous checkpoint to backtrack. When the conflict arises, the solver will select the latest pushed variable that is not only assigned by decision and is related to the conflict but also has *nonflip assignment* to apply backtracking. The *nonflip assignment* means that the variable has not been assigned to the other value by the decision procedure yet.

Next, after deciding the variable for backtracking, the solver will backtrack to the previous checkpoint. The variable pushed

later to the ψ stack than the backtrack variable, will be de-assigned in the *Variable state table*. Thus, the pointer of *Trace table* will decrease to the position of the backtrack variable as well.

Finally, the assignment of the backtrack variable is flipped to the other value. Then, the BCP process is launched. If no such backtrack variable exists in the ψ stack, it represents that the solver has traversed all traces but finds no satisfiable assignments for this problem. Then, solving procedure will be terminated and return *UNSAT* result.

Several demonstrated benchmark results are provided in Section V-A to show that the performance of our proposed hardware SAT solver is competitive or even performs better than several previous work [21], [23], [24], [25].

C. Simplex Solver Implementation

Now, we introduce the proposed hardware Simplex solver.

1) Simplex Algorithm: The Simplex algorithm is used to solve the optimization problem which is composed of a linear objective function and a set of linear constraint functions. The constraint functions in the Simplex must be in a certain format, so they should be transformed into the standard format by adding slack (or surplus) and artificial variables. The Simplex problem is typically presented in a form of a Simplex tableau. Each column of the tableau represents the variables in the Simplex problem, and each row represents the functions.

Our proposed hardware Simplex solver is based on *Two phase Simplex algorithm*. The goal of the first phase is to make sure all artificial variables can be optimized to be 0 by minimizing the sum of all artificial variables. Because there is an implicit non-negativity constraint for all variables in the Simplex problem, if the minimization result is 0, then, it indicates that all artificial variables have been optimized to be 0 and the second phase can be launched next. However, if the optimization result of the first phase is not 0, it means that there is no optimal solution for this Simplex problem.

We use the *Dantzig strategy* for pivot selection, which is shown in Fig. 9. The steps of this strategy for one Simplex iteration are illustrated as follows.

1) *Pivot Column Selection (Entering Variable Selection):* Choose the most negative element in the row of the objective function to become the pivot column. If there is no column that has a negative element, then, the optimal solution has been found, and the Simplex algorithm will terminate or entering the second phase if the optimization result of first phase is 0.
2) *Pivot Row Selection (Leaving Variable Selection):* Calculate the ratio, $\text{ratio}_i = \text{RHS}_i / a_{i,\text{pivot\_col}}$, for each positive element in the pivot column, where i is the row index, a and RHS represent the element and right hand side value, respectively. Choose the row with the smallest ratio to become the pivot row. The element $a_{\text{pivot\_row,pivot\_col}}$ is called the pivot element.
3) *Pivot Operation:* This step contains two operations. First, divide all elements in the pivot row with the pivot element to let the pivot element become one. Next, apply the operation of (3) for each remaining row so that all elements in the pivot column will become 0 except the pivot element

$$\text{new\_}a_{i,j} = \text{ori\_}a_{i,j} - \left(a_{\text{pivot\_row},j} * a_{i,\text{pivot\_col}}\right). \quad (3)$$

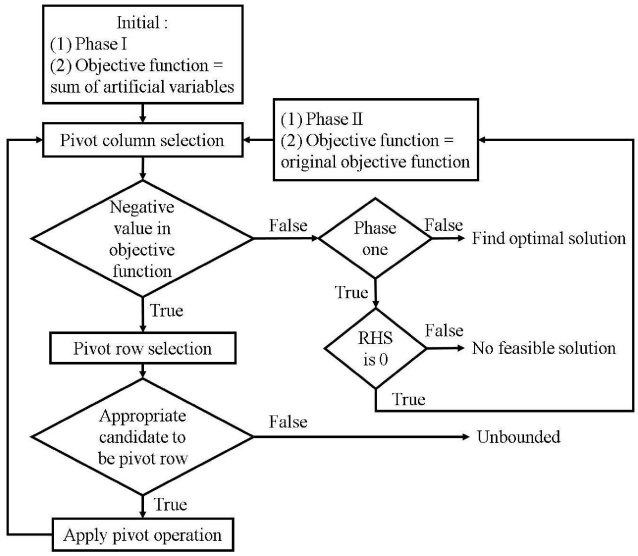

Fig. 9. Two-phase Dantzig Simplex operation flow.

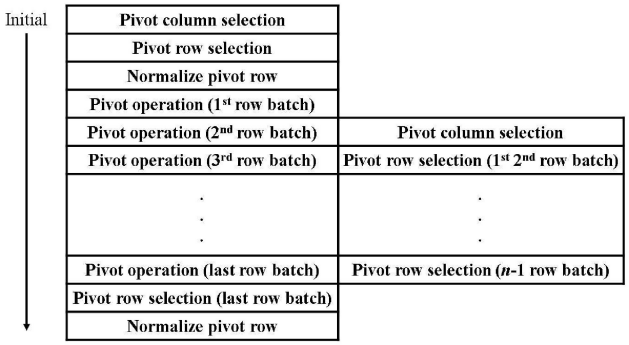

Fig. 10. Parallel operation flow of the Simplex operation.

2) Design Concept and Settings: Although each step in a Simplex iteration seems highly dependent on the result of the previous step, there are still several opportunities to adopt parallel and pipeline operation. For example, originally in each Simplex iteration, pivot operation applies calculations for each element in the Simplex tableau. This is the most computationally expensive step. So we design a new hardware architecture to make these operations be easily performed in parallel. Furthermore, we improve this performance to be more efficient by using a pipelined structure so that the parallel mechanism can not only be used in a single iteration but also between iterations. Fig. 10 is our proposed parallel operation flow of the Simplex operation.

At the beginning of the solving procedure, pivot column selection and pivot row selection perform sequentially. The whole computation of the pivot operation can be decomposed into several suboperations. Normalizing pivot row is the first operation, and the remaining multiply–subtract calculation can be organized into several row batches; that is, the

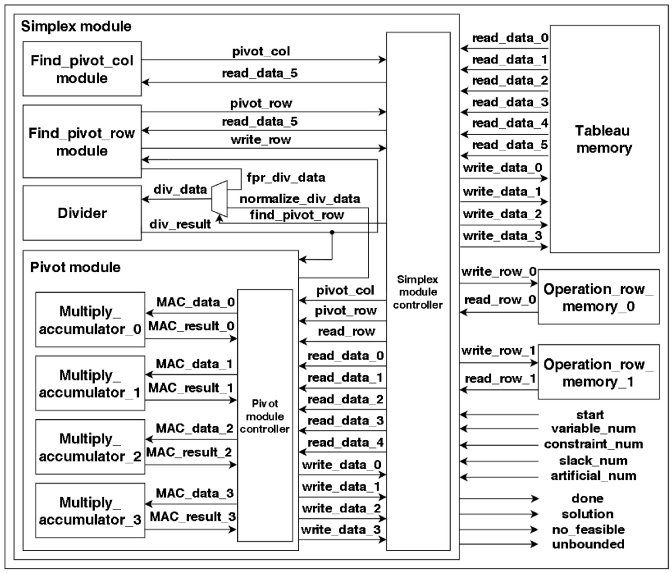

Fig. 11. Design diagram of Simplex solver.

multiply–subtract calculation can be applied to several rows simultaneously. The size of each batch depends on the design's capability of parallelism. The first row batch contains the rows of the two objective functions. Once the pivot operation of the first row batch is finished, the elements in the objective function row are ready for the next Simplex iteration; thus, the pivot column selection can be launched to find the pivot column for the next iteration. The same concept can be applied on pivot row selection too. The pivot row selection can be applied on the previous row batch which finished the pivot operation. While the whole pivot operation is accomplished, the procedure only needs to check the rows in the last row batch for pivot row selection and the pivot operation of the next Simplex iteration can be launched immediately.

The advantage of this parallel operation flow is not only to reduce calculation time and increase module utilization but also brings benefits to hardware design area cost. The reason is explained as follows. Typically, the pivot operation of each row batch takes much more computation time than pivot column selection and pivot row selection, so the hardware design of the modules in charge of those selections can be simple and area efficient. We do not need to have some parallel mechanism like tree comparators commonly used in such the conventional circuit designs for pivot column selection or parallel ratio calculation of pivot row selection, and thus, our design can significantly save more area cost.

3) Design Architecture: According to the Simplex operation flow, there are three main modules of our hardware Simplex solver, and each of them is in charge of one operation step of a Simplex iteration. Fig. 11 is the architecture diagram of the Simplex solver design. *Find_ pivot_col* module is responsible for pivot column selection, *Find_ pivot_row* module is for pivot row selection and *Pivot* module is in charge of the pivot operation. *Tableau memory* is used to store the whole Simplex tableau of the problem. To avoid unnecessary row calculation during the pivot operation, the *Operation_row memory* is responsible for recording the rows that are going to be applied pivot operation because our designed parallel operation flow needs two *Operation_row memory* to record the rows of the current and the next iteration.

The data format in our proposed design is the IEEE754 double precision floating point. Note that the selection and termination condition of the Simplex method is highly dependent on the calculation result, and thus using higher precise value is essential to ensure the correctness of the result and the termination of the solving algorithm. The solver includes a hardware FPU from [2] to handle the floating point calculation. The design uses four multiply accumulators with 5-staged pipeline in the pivot module to provide parallel calculation for pivot operation so that our solver can handle four rows in a single row batch. On the other hand, based on our parallel operation flow, one divider is included to handle both of ratio calculation in the *Find_ pivot_row* module and normalize pivot row operation in the *Pivot* module because those two operations do not be launched at the same time.

Now, we introduce each module of our Simplex solver.

1) *Tableau Memory:* This module is used to store the Simplex tableau of the problem. Its size is $512 \times 2,048$, which can handle the Simplex problem with up to 1027 variables and 512 functions (including 510 constraint functions and two objective functions for two Simplex phases). The solver's data format is IEEE754 double precision floating point, so the data width of *Tableau memory* is 64 bits. In order to provide multiple data accessing at the same time for parallel calculation, the *Tableau memory* has a total of six read ports and four write ports, while five of the read ports and four of the write ports are for parallel pivot calculation, and one extra read port is for pivot column and pivot row selection. We use the first two rows to be the objective function row of Simplex's phase I and phase II, respectively. The remaining rows are the constraint functions of the Simplex problem. The first 1027 columns are for variables in the Simplex problem, and the last column is for the RHS value. The succeeding 510 columns are for slack (or surplus) variables, and the remaining 510 columns are for artificial variables. Because our solver can support up to 510 constraint functions, the maximum number of the slack (or surplus) and artificial variable is 510.
2) *Find_ pivot_col:* This module is used to decide the pivot column. The comparison tasks are done sequentially, instead of in parallel, because the pivot column selection will be operated with the pivot operation that usually takes much more computation time. Thus, comparing one element at a time is actually more efficient.
3) *Find_ pivot_row:* This module is used to apply pivot row selection, which shares the same design concept with the *Find_ pivot_col* module. For each row, it takes two cycles to read the element on the pivot column and the RHS value (i.e., $a_{i,\text{pivot\_col}}$ and RHS_i, respectively), then uses the divider to calculate the ratio value. Once the calculation is finished, it applies ratio comparison.

Because calculating and comparing the ratio value for a row batch usually takes less computation time than pivot operation, this module adopts the *row_boundary* signal to ensure that the pivot row selection is not applied to those rows which have not finished the pivot operation. When the pivot operation is accomplished, the *row_boundary* will be set to the number of constraint functions of the Simplex problem to let the module finish the ratio check of the rest rows.

4) *Pivot:* This module is used to apply pivot operation, which contains two steps, normalizing pivot row with the pivot element and applying the multiply–subtract operation to the rest of the rows. For pivot row normalization, the main purpose is to divide all elements on the pivot row with the pivot element. Because division is often much more time-consuming and high area costly than multiplication, and it is not appropriate to adopt parallel calculation by implementing many dividers in the design. What we design in this solver is to calculate the multiplicative inverse of the pivot element first, then, we use a four pipelined multiply accumulators to apply multiplication on the pivot row in parallel so that we can take advantage from the existing multiply accumulators. The pivot operation will be launched after the normalization is finished. The parallel mechanism in pivot operation is to calculate four rows at the same time, and each multiply accumulator handles one row. Every element in the row is calculated through multiply accumulator's pipeline. According to (3), each pivot operation needs three data. While considering the operation of a row, $a_{i,j}$ and $a_{\text{pivot\_row},j}$ will change each cycle, so it needs four read ports for $a_{i,j}$ and one extra read port for $a_{\text{pivot\_row},j}$. For $a_{i,\text{pivot\_col}}$, this value remains unchanged during the same row operation, so the pivot operation of each row starts from the pivot column, which can use those four read ports to get the four $a_{\text{pivot\_row},j}$ values, and stores them in the solver for the succeeding operations. In order to improve the performance, the information of the number of variables, slack (or surplus) variables, and artificial variables are adopted to avoid unnecessary column calculation. Furthermore, the mechanism of row selection can also improve the performance because the *Pivot* module will get the row information of each row batch from the *Operation_row memory*, and this is also for avoiding unnecessary row calculation.

5) *Operation_row memory:* This module is used to provide the rows that are necessary for applying the pivot operation for the *Pivot* module. According to (3), if the value of $a_{i,\text{pivot\_col}}$ is 0, it is unnecessary to apply pivot operation for this row. Because the tableau of the Simplex problem is often a sparse matrix, proving such row information can be useful for speeding up the pivot operation. The size of *Operation_row memory* is 128 with 37 bits data width. The data stores the information of each row batch containing rows to apply pivot operation, and the data format is shown in Fig. 12. The MSB indicates whether this data are valid or not, and the remaining bits stores the four row indexes of this row batch. If the number of rows in one row batch is less than four, the rest field will be filled in 0 as dummy bits. Because the *Find_pivot_row* module reads $a_{\text{pivot\_row},j}$ for ratio comparison, it records the row's index to *Operation_row memory* while its value is not 0, and the *Operation_row memory* gathers four of the row indices into a row batch for the *Pivot* module. Moreover, the condition of row selection can not only be selected by the value 0 but also by a threshold value. In this design, we decide to select the row by using a threshold value 10^{-12}, and the row index will be stored in *Operation_row memory* if the $a_{\text{pivot\_row},j}$ is $\geq 10^{-12}$. We further discuss different results when we use different threshold values in Section V. Since the design adopts parallel operation flow, it needs two memory designs to record two pieces of information. One memory is used to record the row index (i.e., the output) from *Find_pivot_row* module, and the other memory is used to output the row batch information to the *Pivot* module.

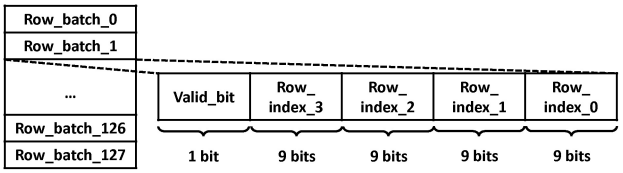

Fig. 12. Data format of *Operation_row memory*.

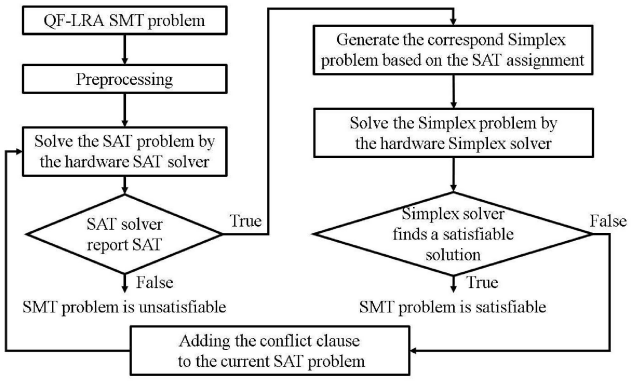

Fig. 13. Work flow of solving QF-LRA SMT problem with SAT and Simplex.

D. Integration of SAT and Simplex Solver

To solve QF-LRA SMT problems, we have to integrate the hardware SAT solver and our proposed Simplex solver. The whole solving procedure is shown in Fig. 13.

The first step is to generate the corresponding SAT problem of the SMT problem. Then, the hardware SAT solver will solve that SAT problem. If the SAT problem is UNSAT, the procedure can conclude that the SMT problem is UNSAT. Otherwise, when the SAT problem is satisfiable, and the SAT solver will output a feasible assignment of every atom in the original SMT problem. In the next step, the hardware Simplex solver will take the Simplex problem which is generated by the atoms of the SMT problem and is corresponding to the given

assignment to check whether the assignment can be QF-LRA theory consistency.

If the Simplex solver finds a feasible solution of the Simplex problem based on the given SAT assignment, then, the procedure can conclude that this SMT problem is satisfiable under the QF-LRA theory; otherwise, it means that the result is QF-LRA inconsistency if the Simplex solver cannot find any feasible solutions. Then, the current assignments will generate external conflict clauses and then be added back to the current SAT problem to make the hardware SAT solver solve again to find another assignment.

The details of each solving step are introduced as follows.

1) Preprocessing: In the preprocessing step, the main purpose is to transform the SMT problem into an SAT problem and links the functions inside the SMT problem to its corresponding Boolean variable.

For transforming the SMT problem into SAT problem, the first step is to rewrite the functions in the SMT problem into a certain format and identify each atom with a Boolean variable. The functions in the SMT problem should be rewritten into $\geq$, $\leq$ or $=$ form. For the function with $>$ or $<$ relation, it will be rewritten into $\leq$ or $\geq$, respectively, by adding a negation to the function. For example, the function $(x - y > 0)$ will be rewritten into $\neg(x - y \leq 0)$. The function with $\neq$ will be transformed into the negation of an equation. In the next step, the Tseitin transformation technique is adopted to rewrite the SMT problem into CNF format. Moreover, due to the implementation constraint (that is, five variables in each clause) of our hardware SAT solver, all of the input CNF Formulas must be transformed into 5-SAT form.

Note that we have a preparatory procedure to generate conflict clauses based on the QF-LRA theory to avoid SAT solver providing the assignment that can cause obvious QF-LRA theory conflict. This preprocess works as follows. The function pair with the same left-hand side coefficient will be selected, and then, the procedure checks whether the selected function pair exists possible conflict assignment or not. For example, the selected functions are $l_1 = (x-y \geq 5)$ and $l_2 = (x-y \geq 0)$, the assignment $l_1 = \text{True}$ and $l_2 = \text{False}$ will cause QF-LRA inconsistency, so the preprocessed conflict clause $(\neg l_1 \vee l_2)$ will be added into the original SAT problem to guide the SAT solver not to give out such a set of assignments.

2) Simplex Problem Generation: While the SAT solver provides a set of assignments, the Simplex solver will be involved to check whether the assignment is QF-LRA consistent or not by solving the corresponding Simplex problem.

The Simplex problems are generated by using all the functions in the SMT problem except the disequalities. The constraint functions of the Simplex problem are the formulas in the SMT problem, and then, a function with $\geq$ relation is selected as the objective function. The purpose of this Simplex problem is to find the minimum value of the objective function. If the Simplex problem can not find a feasible solution, it means that the conjunction of formulas based on the current assignment is QF-LRA inconsistent. Otherwise, the current assignment set is theory consistent; that is, the feasible solution is found. The disequalities should be treated separately because they need to be verified by the Simplex solution. If the current solution violates the disequality, it will be checked into two cases ($>$ and $<$). The selected objective function can also be formulated with $\leq$ relation as long as we change the goal of the Simplex problem to find the maximum value of the objective function.

The functions in the Simplex problem must be considered with the current assignment. When the function's corresponding assignment is False, we need to further check the original relation. We need to change the relation of this function to $>$ if the original relation is $\leq$, or change it to $<$ if the original is $\geq$. The function remains the same if its assignment is True. Since the Simplex problem can only support the functions with $\geq$, $\leq$, or $=$ relation, the function must be modified to $\geq$ (or $\leq$) if its relation is $>$ (or $<$). The modification is done by adding or subtracting a value ϵ to its RHS value, where ϵ is a positive value that is very close to zero. Then ϵ will be added to RHS for transforming $>$ function to $\geq$ function and subtracted from RHS for $<$ function to $\leq$ function.

Moreover, because there is an implicit non-negativity constraint for all the variable inside the Simplex problem, the variable x without lower bound constraint will be rewritten into $x^+ - x^-$, in order to represent the negative part of x with x^-. Every variable in the SMT problem is free to be any real number, so the variable extension will be applied to every variable inside the SMT problem.

3) External Conflict Clause Learning: If the Simplex solver reports that the current assignment exists QF-LRA inconsistency, the next step is to let the SAT solver find another assignment. To prevent the SAT solver from providing the same assignment causing QF-LRA inconsistency, the external conflict clause learning should be adopted to guide the decision. External conflict clause learning is done by adding a clause related to the current assignment into the SAT problem. If the current assignment of one variable is True, then, adding the variable with negation into the clause, and vice versa.

For example, if the assignment of three variables x, y, and z is $\{x = \text{True}, y = \text{False}, z = \text{False}\}$, then, its corresponding external conflict clause is $(\neg x \vee y \vee z)$. By adding this external conflict clause back to the problem, the SAT solver will not give out the same assignment again.

The generated external conflict clause will also be transformed into the 5-SAT format before adding to the CNF due to our hardware SAT solver implementation constraint. We also need to consider the limitations of the number of clauses and variables of our hardware SAT solver design. To reduce the size of the external conflict clause, those variables that the always be True (or False) are removed from the external conflict clause as they have no effect to the learning result.

V. Evaluation

In this section, we demonstrate the evaluation result of the hardware Simplex solver and the integration of hardware SAT and Simplex solver for solving the QF-LRA SMT problem.

A. Evaluation of Proposed Hardware SAT Solver

Our proposed hardware SAT solver runs at 100 MHz and supports up to 511 variables. The SAT solver configuration

TABLE I
SAT PERFORMANCE COMPARISONS WITH SKLIAROVA AND FERRARI [23]

Benchmark	Exec. sec. [23]	Our Exec. sec.	SpeedUp
uf20-10	5.04×10^{-3}	1.19×10^{-5}	423.5
flat30-100	2.29×10^{-2}	1.54×10^{-5}	1487.0
ii8a1	9.14×10^{-3}	2.38×10^{-5}	384.0
par8-4-c	1.91×10^{-2}	1.17×10^{-4}	163.2
hole6	1.92×10^{-2}	4.32×10^{-3}	4.4

TABLE II
SAT PERFORMANCE COMPARISONS WITH SAFAR *et al.* [21]

Benchmark	Exec. sec. [21]	Our Exec. sec.	SpeedUp
par8-1c	2.029×10^{-5}	1.713×10^{-5}	1.2
aim-50-2_0-yes1-2	5.284×10^{-5}	4.329×10^{-5}	1.2
aim-100-3_4-yes1-4	9.4×10^{-2}	2.66×10^{-2}	3.5

TABLE III
SAT PERFORMANCE COMPARISONS WITH USTAOGLU *et al.* [24]

Benchmark	Exec. sec. [24]	Our Exec. sec.	SpeedUp
hole6	6.0×10^{-2}	4.32×10^{-3}	13.9
hole7	6.1×10^{-1}	4.88×10^{-2}	12.5
hole8	7.0	6.0	1.2
uf100-07	3.0×10^{-2}	1.10×10^{-4}	272.7
uf100-010	2.8	1.64×10^{-4}	17073.2

TABLE IV
SAT PERFORMANCE COMPARISONS WITH USTAOGLU *et al.* [25]

Benchmark	Exec. sec. [25]	Our Exec. sec.	SpeedUp
hole7	3.3×10^{-1}	4.88×10^{-2}	6.8
hole9	15.3	1.89×10^{-5}	806878.3
uf100-010	5.8×10^{-1}	1.64×10^{-4}	3536.6
uf125-01	1.2	3.85×10^{-3}	311.7

by Quartus 11.0 is executed on 2.20-GHz CPU with 256-GB memory, running CentOS 6.10. The logic utilization of our solver is 25%, and it is composed of ALUTs 15% (27 259/182 400), memory ALUTs 1% (656/91 200), and dedicated logic registers 9% (16 270/182 400). Total block memory bits costs 1% (65 536/14 625 792).

Here, we provide the performance comparison tables for our proposed hardware SAT solver with several famous works which are, respectively, proposed by Skliarova and Ferrari [23] in 2004, by Safar *et al.* [21] in 2011, and by Ustaoglu *et al.* in both 2018 [24] and 2019 [25]. Note that the SAT benchmarks used in this work are obtained from SATLIB Benchmark Problems [16]. We only demonstrate the results of the benchmarks used in each corresponding related work, which can conclude the results in a fair and appropriate way.

As shown in Tables I–IV, our hardware SAT solver does have better performance than the four related works. It means that our designed hardware SAT solver for the proposed hardware SMT solver in this work can even work faster than some famous related works. The speed-up is obtained as the execution time of reference work divided by ours.

B. Evaluation of Proposed Hardware Simplex Solver

Our proposed hardware Simplex solver can achieve 100-MHz clock frequency under TSMC 90-nm process, and the synthesis area is 612751.5 um^2. Table V shows the information of seven selected Simplex benchmarks for testing the design.

TABLE V
PROBLEM INFORMATION OF SELECTED SIMPLEX BENCHMARK

Benchmark	Variables	Constraints	Slack variables	Artificial variables
AFIRO	32	28	19	8
BLEND	83	75	31	43
BRANDY	249	221	54	166
LOTFI	308	154	58	111
SCAGR7	140	130	45	91
SCFXM1	457	331	143	187
SCORPION	358	389	108	340

First of all, as we described in *Operation_row memory*, the condition of row selection for pivot operation can be a threshold. Thus, we tried three threshold values to compare the performance. As shown in Table VI, the first column shows the result without row selection, the second column shows that we use the exact value 0 as the select condition (i.e., we select the row while the value is not 0), and the remaining three columns use different threshold values as the select conditions. Note that every improvement result (i.e., runtime speed up) with the row selection is obtained from comparing to the results without row selection mechanism.

The results show that the row selection mechanism can achieve the performance improvement more than ten times compared to the design without it. Moreover, we set the threshold value of the row selection to 10^{-12} in our implementation based on the test result. It is not only because setting the threshold value to 10^{-10} and 10^{-7} can not provide better performance compared to 10^{-12}, but also to ensure the termination of design and the correctness of solution by applying a more rigorous condition.

Here, we also compare the performance of our solver with LpSolve which is one of the most well-known software linear-programming solvers. In order to provide an appropriate and fair test environment for both LpSolve and our hardware Simplex solver, the scaling type in LpSolve is set to *Linear* and the pivot rule is set to *Dantzig*. Table VII shows the respective runtime results of LpSolve and our hardware Simplex solver. It demonstrates that our hardware Simplex solver can achieve 1.809 to 93.396 times speed up compared to LpSolve.

By adopting the double precision floating point for the value representation, our hardware Simplex solver can provide the precise result and solution as well, which can guarantee the correctness of the consistency check in the SMT solving procedure. Due to the page limit, we provide the value precision error ratio between the hardware solver's result and the golden result in Appendix B in the supplementary material. Briefly speaking, our experimental results show that the hardware solver's result is exactly the same as the golden result in most benchmarks, and the maximum error ratio is at a scale of 10^{-3}. Therefore, our proposed hardware Simplex solver not only achieves better calculation performance but also provides the precise solution.

C. Evaluation of SolvingQF-LRA SMT Problem

We select 40 QF-LRA benchmarks from SMT-LIB to test the SMT solving procedure of using hardware SAT and

TABLE VI
RUNTIME COMPARISON OF DIFFERENT ROW SELECT CONDITION

Benchmark	Without row selection	Value $\neq 0$		Value $\geq 10^{-12}$		Value $\geq 10^{-10}$		Value $\geq 10^{-7}$	
	Runtime (ms)	Runtime	Speed up	Runtime	Speed up	Runtime	Speed up	Runtime	Speed up
AFIRO	0.168	0.150	1.123	0.150	1.123	0.150	1.123	0.150	1.123
BLEND	4.248	2.997	1.417	2.488	1.707	2.477	1.715	2.477	1.715
BRANDY	147.110	82.256	1.788	69.110	2.129	70.415	2.089	70.408	2.089
LOTFI	31.390	10.516	2.985	5.528	5.678	5.528	5.678	5.528	5.678
SCAGR7	20.010	12.421	1.611	8.586	2.331	8.586	2.331	8.582	2.331
SCFXM1	444.530	201.660	2.204	80.349	5.533	87.256	5.095	87.154	5.101
SCORPION	311.080	65.265	4.766	29.296	10.619	29.296	10.619	29.296	10.619

TABLE VII
RUNTIME RESULTS OF LPSOLVE AND OUR HARDWARE SIMPLEX SOLVER

Benchmark	LpSolve runtime (ms)	Hardware Simplex solver runtime (ms)	Speed up
AFIRO	14	0.150	93.396
BLEND	33	2.488	13.262
BRANDY	125	69.110	1.809
LOTFI	124	5.528	22.430
SCAGR7	56	8.586	6.523
SCFXM1	184	80.349	2.290
SCORPION	142	29.296	4.847

TABLE VIII
PROBLEM INFORMATION OF PARTIAL SMT BENCHMARK

#	Benchmark	Type	Original CNF size	
			Variables	Clauses
A	constr.-cooking07	SAT	173	173
B	constr.-temp.-mach.-shop-2-3-A03	SAT	73	94
C	constr.-tms-2-3-light-05	SAT	93	96
D	gasburner-prop3-1	UNSAT	19	24
E	pursuit-safety-1	UNSAT	23	32

Simplex solver. However, due to the page limit, we only take five of the benchmarks to explain our experimental results, whose information is listed in Table VIII, including the type, the CNF size, and the numbers of variables and clauses. Note that the "original CNF size" column shows the corresponding SAT problem size of the SMT benchmark after preprocessing.

We illustrate why we design the experiments and analyze the corresponding results from the perspective of two questions.

Q−1: Does making SMT solver hardware really bring any benefits?

In order to analyze the benefits brought by the hardware, we compose a software SMT solver based on offline approach (we call it offline SW SMT solver) using MiniSAT and LpSolve, and compare it with our proposed hardware SMT solver (we call it offline HW SMT solver).

The reason why we choose MiniSAT and LpSolve is because MiniSAT and our proposed hardware SAT solver are both DPLL-based SAT solvers, while LpSolve and our proposed hardware Simplex solver are both solvers using basic simplex algorithm. Such a comparison would be fairer.

Table IX records the experimental results of our excerpted five benchmark cases. If we compare the time required for verification, the offline HW SMT solver we proposed is at least 2.583 times faster than the offline SW SMT solver, and up to 198.133 times. The "Iteration" column identifies the number of communication iterations between the SAT and Simplex solver during the solving procedure.

In fact, for the comparison between the offline HW SMT solver and the offline SW SMT solver, we provide more detailed data for all of the 40 benchmark cases in Table XII in Appendix B in the supplementary material. Overall, with the offline approach, the performance of hardware version is about 2.583 to 1561.181 times faster than the software version.

It should be noted that in the two versions of the offline approach, we did not include the time of problem format transformation and problem downloading into the calculation. The main reason is that we want to focus on the execution time of solvers. In this experiment, we purely analyze how much performance improvement from adopting a new hardware architecture for designing an SMT solver.

The results are enough to convince us to propose a new hardware architecture of SMT solver. However, we would like to further investigate the following second question.

Q−2: Can hardware design really compensate for the performance gap between different approaches?

We know that the SMT algorithm can be roughly divided into two types: 1) offline and 2) online. The software SMT solvers adopting the online approach are almost always more efficient than those using the offline approach. Therefore, we want to know if the proposed offline hardware SMT solver can compensate for the performance gap between two approaches.

In order to show the performance gap between different approaches, we first compare the offline SW SMT solver with one of the most efficient and famous online SMT solvers named Z3. Table X shows the experimental result for the same selected five benchmark cases. From Table X, we can see that Z3 is 1.934 to 117.665 times faster. Obviously, the online approach can usually perform more efficiently than the offline approach in solving SMT problems.

Then, for the same benchmark cases, we instead use our proposed offline HW SMT solver to compare with Z3. In some cases, as shown in Table XI, we can see that even with the offline approach, the hardware version of the SMT solver is possible to compensate the performance gap of the original approaches, and even perform better! In Table XI, due to the hardware design, there 80% cases (i.e., 4 cases) get better results than the performance of Z3. Similarly, we experiment with more benchmarks. Due to the page limit, we organize the detailed results in Table XIII of Appendix B in the supplementary material.

Note that we do not intend to express that the offline HW SMT solver can achieve better performance than Z3 on all benchmarks. In fact, there are still many cases where Z3 is faster. What we want to emphasize is that for the offline

TABLE IX
COMPARISON BETWEEN HARDWARE AND SOFTWARE PROCEDURE

#	HW (ms)				SW (ms)				HW
	Iteration	SAT	Simplex	Total	Iteration	MiniSAT	LpSolve	Total	SpeedUp
A	1	0.097	14.284	14.381	1	3.516	33	36.516	2.539
B	8	0.321	7.399	7.720	178	800.645	729	1529.645	198.133
C	50	4.685	98.010	102.695	40	94.228	171	265.228	2.583
D	5	0.037	0.208	0.245	5	7.213	16	23.213	94.643
E	7	0.058	0.362	0.419	7	11.985	22	33.985	81.019

TABLE X
COMPARISON BETWEEN SOFTWARE PROCEDURE AND Z3

Benchmark	Iteration	SW (ms)	Z3 (ms)	Z3 SpeedUp
A	1	36.516	17	2.148
B	178	1529.645	13	117.665
C	40	265.228	14	18.945
D	5	23.213	12	1.934
E	7	33.985	12	2.832

TABLE XI
COMPARISON BETWEEN HARDWARE PROCEDURE AND Z3

Benchmark	HW (ms)	Z3 (ms)	HW SpeedUp
A	14.381	17	1.182
B	7.720	13	1.684
C	102.695	14	0.136
D	0.245	12	48.926
E	0.419	12	28.608

approach designing in a new hardware architecture can show competitive performance in the results of some cases, even better than the online approach. Therefore, it is definitely possible to consider proposing a new hardware architecture for the online approach and expect that it might be able to verify the complex SMT problems more efficiently.

D. Discussion

Here, we want to further discuss two topics derived from our work. The first one is the scalability of our proposed work. To handle large-scale SMT problems, there are two feasible approaches. The first approach is to extend the current design by duplicating the hardware implementation of our SMT solver. The mechanism of our proposed hardware architecture (including pipeline/processing elements for parallel BCP execution, the parallel operation flow, row selection mechanism in the Simplex solver, etc.,) is modularized and able to be composed to constitute a larger design. Table XIV in Appendix in the supplementary material shows the synthesized area size of our hardware solvers. The FPU module in the hardware Simplex solver dominates the size (93%). The area cost and design complexity of FPU module mainly depend on the solution resolution, i.e., numerical accuracy. That is why our hardware Simplex solver provides competitive numerical accuracy compared with related works. Fortunately, the same FPU design can be used when we extend the Simplex solver. The modules needed to be modified are mainly those used for storing the problem information and the corresponding controllers (7%). The second approach is to adopt divide-and-conquer techniques [19], [20] in the SAT part, i.e., to decompose a large SAT problem into subproblems to be solved separately in parallel (in duplicated copies of our hardware SAT solver) and then compose the subresults to obtain the final result. In our design, since SAT and Simplex work offline to each other, the parallel SAT solving does not affect the design of the Simplex solver.

The second topic is to discuss why our solver can achieve better performance than other solvers. For the part of our hardware SAT solver, it is mainly from the benefits of our proposed optimized BCP process with a pipeline structure and a nonchronological backtracking mechanism. For our hardware Simplex solver, it features the parallel operation flow between Simplex iterations to execute the selection operations parallelly with the pivot operation and the row selection mechanism to avoid unnecessary row computation while the subtrahend is (or close enough) to 0. Our hardware Simplex solver having such a special design can save lots of time because such a scenario is not rare to see due to that the Simplex tableau is often a sparse matrix.

VI. CONCLUSION

We propose a new hardware architecture of SMT solver to solve the QF-LRA problem, including a hardware SAT solver and a hardware Simplex solver. According to our experimental results, the proposed hardware SMT solver can not only provide precise solutions but also achieve from 2.539 up to 1561.181 times speedup compared with software SMT solvers in our selected 40 benchmarks. Furthermore, we optimistically expect that our work is a good starting point for exploring the hardware architecture design of SMT online approach.

REFERENCES

[1] "Satisfiability modulo theories—Stanford university." 2019. [Online]. Available: https://web.stanford.edu/class/cs357/lecture10.pdf
[2] "FPnew—New floating-point unit with transprecision capabilities." Accessed: Apr. 19, 2021. [Online]. Available: https://github.com/pulp-platform/fpnew
[3] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *Proc. Des. Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 1538–1543.
[4] L. Alt and C. Reitwiessner, "SMT-based verification of solidity smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds. Berlin, Germany: Springer Int., 2018, pp. 376–388.
[5] C. Baier and J. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.
[6] C. Barrett *et al.*, "CVC4," in *Computer-Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Germany: Springer, 2011, pp. 171–177.
[7] S. Bayliss, C.-S. Bouganis, G. A. Constantinides, and W. Luk, "An FPGA implementation of the simplex algorithm," in *Proc. IEEE Int. Conf. Field Program. Technol.*, 2006, pp. 49–56.

[8] M. Bozzano *et al.*, "Encoding RTL constructs for MathSAT: A preliminary report," *Electron. Notes Theor. Comput. Sci.*, vol. 144, no. 2, pp. 3–14, 2006.
[9] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MATHSAT 4 SMT solver," in *Computer-Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Germany: Springer, 2008, pp. 299–303.
[10] C. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. New York, NY, USA: Academic, 1973.
[11] E. K. P. Chong and S. H. Zak, *Simplex Method*, Wiley, 2008, ch. 16, pp. 333–370.
[12] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.
[13] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Germany: Springer, 2008, pp. 337–340.
[14] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Computer-Aided Verification*, T. Ball and R. B. Jones, Eds. Berlin, Germany: Springer, 2006, pp. 81–94.
[15] N. Fern, I. San, and K. Cheng, "Detecting hardware trojans in unspecified functionality through solving satisfiability problems," in *Proc. 22nd Asia South Pac. Des. Autom. Conf. (ASP-DAC)*, 2017, pp. 504–598.
[16] "SATLIB benchmark suite." Accessed: Aug. 11, 2000. [Online]. Available: https://www.cs.ubc.ca/hoos/SATLIB/benchm.html
[17] K. Kanazawa and T. Maruyama, "An approach for solving large SAT problems on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 1, pp. 1–21, Dec. 2010.
[18] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," 2017, *arXiv:1702.01135*.
[19] L. L. Frioux, S. Baarir, J. Sopena, and F. Kordon, "Modular and efficient divide-and-conquer SAT solver on top of the painless framework," in *Proc. Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS)*, 2019, pp. 135–151.
[20] C. Tian Q. Fan, Z. Duan, and H. Du, "Clustering and partition based divide and conquer for SAT solving," in *Proc. 10th Int. Conf. Mobile Ad-Hoc Sens. Netw.*, 2014, pp. 299–307.
[21] M. Safar, M. Watheq El-Kharashi, M. Shalan, and A. Salem, "A reconfigurable, pipelined, conflict directed jumping search SAT solver," in *Proc. Des. Autom. Test Europe*, 2011, pp. 1–6.
[22] R. Sebastiani, "Lazy satisfiability modulo theories," *J. Satisfiability Boolean Model. Comput.*, vol. 3, pp. 141–224, Dec. 2007.
[23] I. Skliarova and A. B. Ferrari, "A software/reconfigurable hardware SAT solver," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 4, pp. 408–419, Apr. 2004.
[24] B. Ustaoglu, S. Huhn, D. Große, and R. Drechsler, "SAT-lancer: A hardware SAT-solver for self-verification," in *Proc. ACM Great Lakes Symp. VLSI*, 2018, pp. 479–482.
[25] B. Ustaoglu, S. Huhn, F. S. Torres, D. Große, and R. Drechsler, "SAT-hard: A learning-based hardware SAT-solver," in *Proc. 22nd Euromicro Conf. Digit. Syst. Des. (DSD)*, 2019, pp. 74–81.

Yean-Ru Chen (Member, IEEE) received the Ph.D. degree from the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan, in 2014.

She is an Assistant Professor with the Department of Electrical Engineering, National Cheng Kung University (NCKU), Tainan City, Taiwan. She worked with MediaTek Inc., Hsinchu, Taiwan, as a Senior Formal Verification Engineer in 2014, then worked with Cadence Design Systems Inc., Hsinchu, in 2015, and joined NCKU in 2016. She is recently working on applying formal verification methods on robustness analysis of neural network, security vulnerability detection of processors, and quantum circuit verification. Her research interests include formal verification, on-chip safety/security-critical system verification, and digital circuit design.

Si-Han Chen received the M.S. degree in electrical engineering from National Cheng Kung University, Tainan City, Taiwan, in 2021.

He is currently a Research Engineer of Computing and Information Systems with Singapore Management University, Singapore. His recent research interests include digital circuit design, formal verification, and quantum system design.

Shang-Wei Lin received the Ph.D. degree in computer science and information engineering from National Chung Cheng University, Chiayi, Taiwan, in 2010.

He is an Assistant Professor with the School of Computer Science and Engineering, Nanyang Technological University (NTU), Singapore. In 2011, he was a Postdoctoral Researcher with the School of Computing, National University of Singapore (NUS), Singapore. From 2012 to November 2014, he was a Research Scientist with Temasek Laboratories, NUS. From December 2014 to April 2015, he was a Postdoctoral Research Fellow with the Singapore University of Technology and Design, Singapore. He joined NTU as an Assistant Professor in April 2015. Recently, he is working on applying formal methods on program analysis/verification, smart contract analysis/verification, and quantum program analysis/verification. His research interests include formal verification, formal synthesis, embedded system design, cyber–physical systems, security systems, multicore programming, and component-based object-oriented application frameworks for real-time embedded systems.