QuteRTL: Towards an Open Source Framework for RTL Design Synthesis and Verification

Hu-Hsi Yeh[1], Cheng-Yin Wu[2], and Chung-Yang (Ric) Huang[1,2]

[1] Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
[2] Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan

Abstract. We build an open-source RTL framework, QuteRTL, which can serve as a front-end for research in RTL synthesis and verification. Users can use QuteRTL to read in RTL Verilog designs, obtain CDFGs, generate hierarchical or flattened gate-level netlist, and link to logic synthesis/ optimization tools (e.g. Berkeley ABC). We have tested QuteRTL on various RTL designs and applied formal equivalence checking with third party tool to verify the correctness of the generated netlist. In addition, we also define interfaces for the netlist creation and formal engines. Users can easily adopt other parsers into QuteRTL by the netlist creation interface, or call different formal engines for verification and debugging by the formal engine interface. Various research opportunities are made possible by this framework, such as RTL debugging, word-level formal engines, design abstraction, and a complete RTL-to-gate tool chain, etc. In this paper, we demonstrate the applications of QuteRTL on constrained random simulation and property checking.

Keywords: Synthesis, Verification, Open Source, Framework.

1 Introduction

In a typical EDA (Electronic Design Automation) software, a quality front-end is necessary for reading in complex design and extracting significant information for later executions. A quality front-end should be capable of reading in all the defined descriptions and translating them into efficient data structures. Traditional academic tools, such as SIS [1], VIS [2], and MVSIS [3], focus on the Boolean-level optimization algorithms that can improve the quality of circuits in various aspects. They are robust enough and, at the same time, scalable for practical use. In the past decades, people from industry and academia have adopted and developed their synthesis and verification tools from these tools. However, as the design paradigm moves to Register-Transfer-Level (RTL) and up, most of the new research have to deal with the high-level design constructs, syntax, and semantics. Without a robust front-end, the applicability of these tools will be limited.

Recently, Berkeley ABC [4], which is a software system for synthesis and verification, has become very popular in both academia and industry. It proposes: (1) fast and scalable logic optimization based on And-Inverter Graphs (AIGs), (2) optimal-delay DAG-based technology mapping for standard cells and FPGAs, and (3) innovative algorithms for integrated sequential optimization and verification. However, it still has incomplete support on design formats; for example, it cannot read in most of the descriptions in RTL Verilog, hierarchical BLIF and BLIF-MV, and it mainly handles the specialized format－BLIF, which is bit-level. Therefore, we need to resort to other tools to translate the RTL design into the BLIF format. Consequently, we will then lose most of the high-level design intents such as FSM, counter, and control/data separation, etc., which can be useful in guiding the design verification.

On the other hand, there are also some open-source front-ends, including VIS and Icarus Verilog [5]. The front-end of VIS acts as an intermediate role to translate designs into BLIF format. It does not completely keep the high-level design intents and does not have complete support for HDL. On the other hand, Icarus Verilog aims at simulation and FPGA synthesis. It still has some known and unknown bugs and the author continues releasing patches.

We implement a quick and quality RTL front-end (QuteRTL) which supports most of the synthesizable RTL Verilog with different library formats and can synthesize the design to word-level circuit netlist. The key features of QuteRTL include: (1) complete Verilog support, (2) flexible design view: word-level or bit level; hierarchical or flatten, (3) formally verified by commercial equivalent checker, and (4) complete netlist creation interface for other parsers (e.g. VHDL/System Verilog parser) and engine interface for external solvers (e.g. BDD, MiniSAT [6], and Boolector [7]).

As an exemplar application of the QuteRTL framework, we publish an Automatic Target Constraint Generation (ATCG) technique in [8] to address the bottleneck in the constrained random simulation flow. Instead of focusing on the constraint solving techniques as other research [9, 10] do, we propose an alternative approach to alleviate the burden of the users by automatically generating high-quality constraints with the support of QuteRTL. In another application, we devise a property-specific sequential invariant extraction algorithm in [11] to improve the performance of the SAT-based unbounded model checking (UMC). We first utilize QuteRTL to extract the property-related predicates and their corresponding high-level design constructs such as FSMs and counters. Thus, we can quickly identify the sequential invariants and then utilize them to refine the inductive hypothesis [12] in induction-based UMC, and to improve the accuracy of reachable state approximation in interpolation-based UMC [13, 14].

The rest of the paper is organized as follows: in Section 2, we first give an introduction of the architecture and interfaces of QuteRTL. Section 3 presents the tool implementation and data structure, and Section 4 presents the applications of QuteRTL. In Section 5, we give a user guide and some demo examples for general users. Finally, we conclude the paper in Section 6.

2 Architecture of QuteRTL Framework

In this section, we will present our RTL synthesis and verification framework — QuteRTL. Section 2.1 gives an overview of the framework while Section 2.2 describes the design and engine interfaces of QuteRTL. Finally, Section 2.3 provides a comparison between QuteRTL and other open-source front-ends

2.1 Overview of QuteRTL Framework

Figure 1 shows the architecture of QuteRTL framework, which can be separated into two parts, RTL synthesis and circuit verification/debugging. In the RTL synthesis part, the RTL design is first translated into some intermediate representations, for example, Control-Data Flow Graph (CDFG). Then, QuteRTL resolves such temporary models by elaborating an equivalent circuit netlist and extracting plenty of design intents, including hierarchy information, FSM, counter structures, etc. These intents can help both test pattern generation and safety/liveness property checking in the circuit verification/debugging part. For general users, we release the source code of our parsers, netlist creation procedure and interface functions.

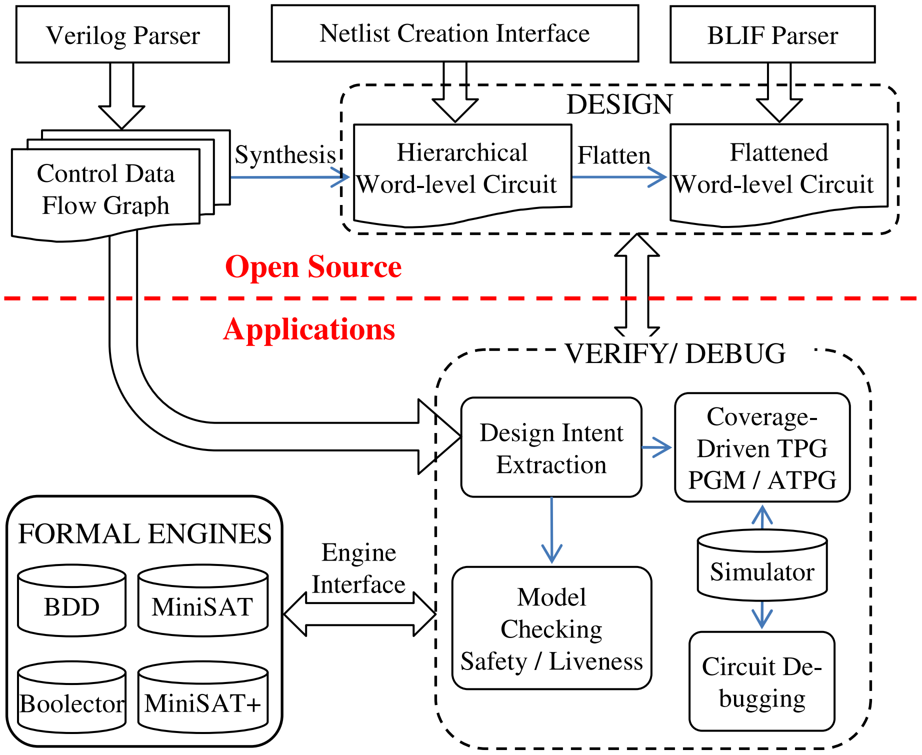

Fig. 1. Architecture of QuteRTL

In the view of design, we have both hierarchical and flattened word-level circuit structure in QuteRTL. Using the hierarchical structure, we can analyze designs more systematically and identify predicates easily in the original RTL. For example, QuteRTL can determine the independence between modules with the hierarchy, and then the information is utilized to alleviate design complexity. For formal engines, the search space can be pruned significantly; for simulators, the efficiency can be improved by the divide-and-conquer algorithm. On the other hand, for circuit redundancy elimination and global optimization, QuteRTL can flatten the design into a single circuit netlist. When flattening the hierarchical design, it will collect the necessary cells in depth first search from PO to PI, and remove redundant cells, which come from bad coding styles or function-less buffers.

In the view of circuit netlist, most logic optimization tools perform their algorithm on bit-level logic netlist, but they rarely handle the word-level netlist. The proposed tool in the paper can completely translate netlist into what logic optimization tools support. That is, QuteRTL can output both the word-level or the bit-level netlist, or even the mixed-level netlist. In addition, it can utilize some word-level circuit components to assist logic optimization tools. For example, QuteRTL can use high-speed adders, says carry look-ahead adders, to substitute carry ripple adders, or Booth's multipliers for high speed designs.

2.2 Supported Features of QuteRTL

Various features are supported by QuteRTL. To illustrate them more clearly and succinctly, we categorize them as follows:

Design Formats. As shown in Figure 1, QuteRTL supports several kinds of design input formats, which include not only Verilog but also other well-known word-level or Boolean network, for instance BLIF and BTOR. Moreover, we provide a complete set of interface functions for interactive netlist creation. The biggest advantage is that anyone can simply call our netlist creation functions to build up a hierarchical word-level network in QuteRTL despite what input formats of the designs are. Hence, any word-level or Boolean network can be intuitively constructed in QuteRTL with the help of these interface functions. On the contrary, QuteRTL also supports corresponding design output formats, including both hierarchical and flattened structural Verilog, and BLIF.

Design Intent Extraction. Design intents contain useful information to help optimization or verification tools improve design and dependability, but many tools and research abandon the information when they proceed. In QuteRTL, the synthesized circuit can be easily annotated to the original RTL structure before logic optimization. Thus, we can extract some design intents from the circuit netlist and CDFG. These design intents include local FSMs, counters, constraints, and invariants.

Interface for Verification and Debugging. After constructing the target design, we can adopt the following interface functions to verify or debug the properties. We split these functions into two parts:

1. Property Specification Interface: We support various types of assertion specifications. These assertions including simple CTL safety and liveness properties in either AG(*p*) or EG(*p*) format, where *p* can be specified as an auxiliary Boolean output signal formulated from the design; for instance $a + b < c$ or $x * y > 10$. Besides, part of System Verilog Assertions (SVA) semantics is also supported for common industrial instances.
2. Engine Interface: Formal engines are crucial to both verification and debugging, especially in formal approaches. However, every engine embraces its individual interface functions, so users need to use the respective interface functions when applying different solvers. It causes maintainability problems in the interfaces for the solvers. Therefore, we integrate those interfaces into a union set of functions that are conformable to different verification and debugging needs in QuteRTL. The integrated engine interface makes the usage of formal engines simple and unified. That is, users can specify which formal engine they expect to adopt in their applications.

2.3 Comparison with other Open Source RTL Front-End

In this subsection, we discuss the comparison between QuteRTL and other open-source front-ends, including VIS and Icarus Verilog. The VIS group releases a Verilog HDL front-end VL2MV, which compiles a subset of Verilog into an intermediate format BLIF-MV (a multi-valued extension of BLIF). With the support of VL2MV, VIS is able to synthesize finite state systems and verify properties of such system. Besides, VL2MV extracts a set of interacting FSMs which preserve the behavior of the source Verilog defined in terms of the simulated results. However, the front-end does not guarantee the extracted FSMs are optimal, and is not able to handle full set Verilog language due to its dynamic nature.

Another open source RTL front-end Icarus Verilog aims at simulation and FPGA synthesis. It can support richer syntax for simulation in RTL language, and generate the text or waveform output of the simulation results. Icarus Verilog is intended to work mainly as a simulator, although its synthesis capabilities are improving. However, the tool focuses on generating specific netlist format for FPGA synthesis, and it is hard to utilize novel formal techniques on the specific netlist.

To apply modern formal techniques to industrial RTL design, we implement a quick and quality RTL front-end QuteRTL. It can synthesize most of the synthesizable RTL with different library (Verilog and Liberty) formats into word-level circuit netlist. For design verification, QuteRTL also supports other design input formats, for example BLIF and BTOR, etc. Besides, users can easily implement novel formal techniques on the word-level circuit netlist, for example UMC, property directed reachability (PDR) [15], etc.

3 Tool Implementation

In this section, we describe the implementation of QuteRTL, which consists of a Verilog parser, an RTL synthesizer, and a circuit flattening procedure.

3.1 Parser and Preprocessor

Verilog Parser. We use Lex and Yacc to implement the Verilog parser. If the syntax of the design conforms to Verilog Backus-Naur Form (BNF), the parser will parse corresponding syntax trees for a start. It also checks the grammars of the syntaxes and lints for Verilog. Then we construct CDFG of the design from the syntax trees for each module. For the purpose of design synthesis and verification, we focus on the synthesizable Verilog subset, which includes synthesizable "for loop", "task" and "function" declarations, etc.

Preprocessor. The preprocessor mainly handles macro substitution, hierarchy construction, and parameter overriding. After generating the CDFGs, we first perform a simple substitution and expand the occurrence of each argument in macro using the replacement text. For the modules containing macros, we revise their CDFGs. Next, we construct a hierarchical tree to represent the relation of the module instances in the design, and then perform parameter overriding from top to down in the hierarchical tree to set up the overridden parameter for each module instance. After the steps, the CDFGs and hierarchical tree are ready for synthesis.

3.2 RTL Synthesis and Circuit Flattening

Data Structure of Circuit Netlist. Figure 2 shows the data structure of circuit netlist in QuteRTL. We use three components–Cell, InPin, and OutPin to describe a circuit. The Cell contains OutPin(s) to fan out to other cells and an InPin list to receive multiple fanins from other Cells to construct the circuit netlist. The pins can be multiple bits to describe word-level netlist.

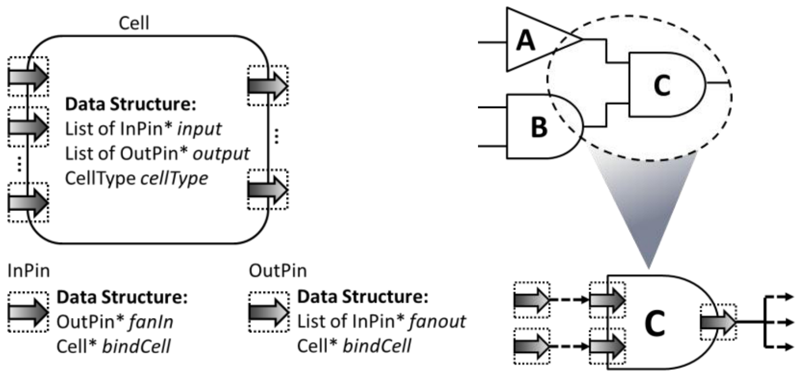

Fig. 2. The data structure of circuit netlist in QuteRTL

The types of cells can be classified as follows:

- Operator cell: arithmetic, relational, equality, logical, bit-wise, reduction, shift cell, and multiplexer
- IO cell: primary input, primary output, and primary inout
- Sequential cell: flip-flop and latch
- Module cell: module instantiation
- Modeling cell: bit-merging, bit-splitting, bus, memory, bufif, etc.

The operator cells are synthesized from the common operators in Verilog. For example, the multiplexers are synthesized from conditional operator (?:) or conditional block (if, case, etc.). For the instances used in a module, we model them as module cells in the hierarchical view of design. Besides, to support the specific elements in circuit, we create some modeling cells for net, bus, memory, and high impedance. Please note that the pins in word-level netlist are multiple bits, so we use bit-merging (bit-splitting) cells to concatenate (slice) pins to form specific fanins to other cells.

RTL Synthesis Procedure. The synthesis procedure translates CDFGs to the circuit netlist data structures we defined above. The synthesizer first traverses the CDFG of each module and flattens each variable to the data structure "SynVar". Figure 3 gives an example to show the relations between RTL and SynVar. In the data structure, each node contains the data and conditional fanins, which are respectively synthesized from data predicate list (DPL) and control predicate list (CPL) of the variable. The tree structure represents the priority of control predicates in nodes, and then we connect these pins with multiplexers. If the variable is in a sequential block or is not fully assigned in a combinational block in the original Verilog code, the output of the last multiplexer will be connected to a sequential cell (flip-flop/latch). Finally, the synthesized circuit netlist is illustrated in Figure 4. In order to back-annotate the netlist information to the original RTL code, we just synthesize the RTL design to an equivalent circuit netlist without optimizing the netlist during this procedure.

Circuit Flattening. The circuit flattening is to generate a flattened circuit netlist which is functionally equivalent to the hierarchical netlist. The implementation includes the concretion of instance models (i.e. module cells) and the removal of redundant cells (ex. buffers, non-fan-out cells). First, we traverse the hierarchical tree built in preprocessor, and duplicate the non-IO cells (except top level module) to a new flattened module. Simultaneously, we make the connections between cells within the same hierarchical module, and record the connections between different hierarchical modules. After duplicating all necessary cells, we connect the cells in different hierarchical and then traverse the flattened netlist to remove the redundant cells.

```
always@ (posedge clk) begin
   if (rst) begin
      x <= 1;
      y <= 0;
   end
   else begin
      y <= x;
      if (x < 100)
         x <= y+x;
   end
end
```

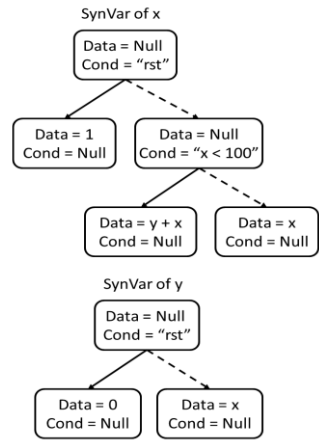

Fig. 3. RTL synthesis procedure

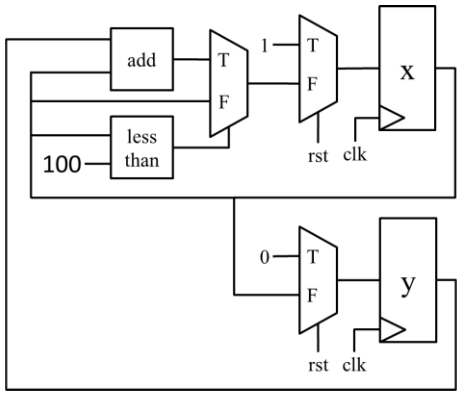

Fig. 4. The synthesized circuit

4 Applications of QuteRTL

In this section, we introduce two applications of QuteRTL, which include intent extraction in Section 4.1 and model checking in Section 4.2.

4.1 Intent Extraction

For FSM extraction, we categorize the types of FSM as either explicit FSMs or implicit FSMs according to the definition of state values. In an explicit FSM, its state values are explicit defined as parameters or constants, while there are no explicit state values defined in an implicit FSM, where the state values are implicitly embedded in conditions or expressions. In our implementation of the extractor, we extract both of them and identify counters. Note that we extract the explicit FSMs based on the coding styles [16] and implicit FSMs from the transition relations computed by BDDs [17]. The extraction algorithm is mainly performed in the three steps: candidate state variable extraction, state transition extraction, and state transition graph (STG) construction. We briefly express these steps as follows:

1. Candidate state variable identification: In sequential blocks of Verilog, we first treat the variables in left hand side of assignments as possible state variables. Then, we traverse the data dependency list of the possible state variable to find a loop of assign statements to identify the candidate state variables.
3. State transition extraction: In this step, we extract the state transition relation from each candidate state variable. For explicit FSM, we can extract a set of state pair (S_i, S_j), which represents the state transition from S_i to S_j. While for implicit FSM, we traverse the assignments of the candidate state variables to build the state transition relation in binary decision diagram (BDD).
4. State transition graph construction: For explicit FSM, we use the set of state pair to construct the STG. In order to extract the STG of implicit FSM, we will traverse the BDD to enumerate all transition conditions and relations.

Further, these extracted FSMs are utilized to identify the sequential invariants and then improve the property proving capabilities in [11]. On the other hand, for constrained random simulation, we proposed an ATCG technique [8] based on QuteRTL. In that work, we extract compact constraints for a set of coverage holes from the circuit netlist and CDFG. The experimental results show that the extracted constraints indeed help simulation achieves the highest coverage and smallest runtime when compared to both random and directed simulations.

4.2 Model Checking

The powerful characteristics of our QuteRTL that retain word-level information with high-level design intent provide us an adequate circuit abstraction level for researching on word-level verification and debugging problems. With the prosperous SMT solvers, it becomes practical and ideal to apply model checking on our word-level netlist with a word-level solver.

There are basically two approaches to implement a model checking algorithm on QuteRTL. First, we can adopt the provided engine interface functions to realize a new model checking algorithm. This is commonly used by almost all the verification algorithms we have implemented. Second, we can dump out word-level netlist from QuteRTL and then call the solvers by their supported interfaces. When transforming word-level functions into CNF for Boolean SAT engines, such as adder, multiplier, comparators, etc., we perform naïve bit-blasting technique with better encodings.

Traditional SAT-based model checking algorithms, including bounded model checking (BMC), k-induction, and their extensions such as simple-path and interpolation-based, can be simply implemented with circuit traversal and transforming individual gate function into corresponding solver input formula (e.g. CNF). Without loss of generosity, all the Boolean model checking algorithms can be implemented on QuteRTL. Moreover, our word-level framework provides even better capability in coping with more complex designs and realistic properties by abstraction and refinement techniques, for instance, predicate abstraction, interpolation, design intent extraction, and probabilistic inferences.

5 Availability for General Users

For general users, we release our RTL front-end source code and the compiled QuteRTL executable in the following website:

```
http://dvlab.ee.ntu.edu.tw/~publication/QuteRTL/
```

In this section, we first give a brief overview to the command-line interface of QuteRTL. Then we show some examples related to what QuteRTL can do for general users through our user-friendly command-line interface. Users can also download these examples in our website, which include a general RTL to gate synthesis flow, an example to construct hierarchical word-level netlist, and a property checking instance.

5.1 A Brief Description to QuteRTL Command-Line Interface

Similar to most tools from EDA vendors, QuteRTL supports friendly command-line interface for users. Our commands are usually composed by one or two mandatory key words followed by a set of required/optional parameters. For example, command to parse an input design from a single file or filelist is “REAd DEsign [-Verilog | -Blif] <[-Filelist] (string filename)>”. We can see the command is named by “REAd DEsign”, where the upper case letters are mandatory for command-line parser. Parameters in square brackets indicate optional arguments, and those in angle brackets indicate required arguments. More detailed description to our command rules can be found in our website, and we will mention some of them in our examples later.

Besides, there is a command “HELp” for showing all available commands, or showing detailed usage of each command (for instance, “HELp REAd DEsign”).

5.2 Example: RTL to Gate Synthesis Flow

In the first example, we are going to show the synthesis flow of QuteRTL. The adopted designs are “i2c” and “usb_phy” from OpenCore [18]. We present the commands of the flow in Figure 5. Note that users can run the series of commands from a batch file using “dofile” command or execute argument “-f” to specify the batch file.

In the first line of Figure 5, we record the commands we are going to execute throughout the program into a log file, which can be used as batch file in the future run. Then we parse the Verilog design from the file list. Note that users must write all related files in the file list for QuteRTL once. After the Verilog design is parsed, the command “syn” performs synthesis procedure to transform the design into a word-level circuit netlist, and the command “flat” flatten the design into a single flattened module. Note that internal signals in the flattened module will be renamed.

After our front-end processing, QuteRTL can output either hierarchical design by using the command "write design" or flattened circuit by using the command "write ckt". As shown in Figure 5, after performing circuit flattening, we output a hierarchical word-level netlist in Verilog format (i2c_design.v) in line 5, and a flattened one (i2c_ckt.v) in line 6. Besides, we can also output the BLIF format of the design (i2c_ckt.blif). The BLIF format is a suitable input for other research on Boolean network, and easily transformed into other related formats, for instance, AIG. Accompanied with the macro library file "lib2.v", users can run the Cadence Conformal LEC [19] equivalence checker script to check the equivalence among original design and all generated output designs. Figure 6 shows the partial synthesized word-level circuit of "i2c". In the figure, rectangles, trapezoids, and ellipses respectively represent flip-flops, multiplexers, and other operating gates.

```
// Example : flow_i2c.dofile
1.   set log -cmd flow_i2c.dofile
2.   read des -f filelist
3.   syn
4.   flat
5.   write des i2c_design.v
6.   write ckt i2c_ckt.v
7.   write ckt -blif i2c_ckt.blif
8.   q -f
```

```
// Example : flow_usb.dofile
1.   set logfile -cmd flow_usb.dofile
2.   read design -f filelist
3.   synthesis
4.   flatten
5.   write design usb_design.v
6.   write ckt usb_ckt.v
7.   write ckt usb_ckt.blif -blif
8.   quit -f
```

Fig. 5. Batch files for RTL to gate synthesis flow example

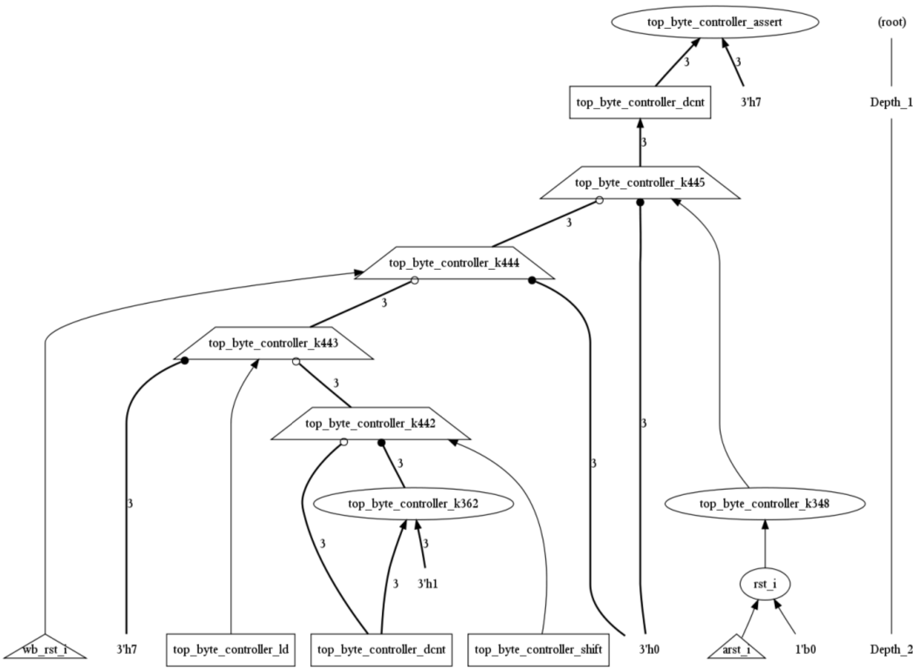

Fig. 6. The part synthesis word-level circuit of i2c

5.3 Example: Hierarchical Word-Level Netlist Creation

As shown in Fig 1, QuteRTL has a complete set of interface functions for netlist creation. Especially, we also support users to construct design through our command-line interface. It is especially convenient to build small designs for instant experiments.

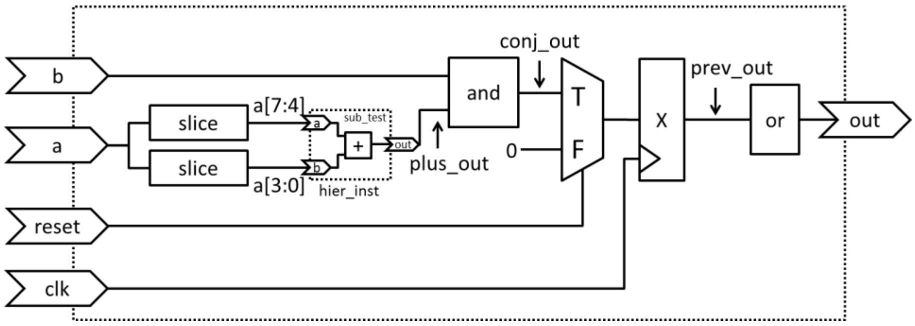

Fig. 7. An example of circuit netlist

Suppose we want to construct the circuit netlist in Figure 7. We present two scripts: "construct_flat.dofile" for constructing the design with only one module, and "construct_hier.dofile" for constructing a hierarchical design, which is functionality identical to the former. The batch files are shown in Figure 8 and 9, respectively. (A portion of commands in "construct_hier.dofile" is omitted in Figure 9 due to space concerns.)

```
// Example : construct_flat.dofile
create design flat_design
define net -PI clk 1
define net -PI reset 1
define net -PI a 8
define net -PI b 4
define net -PO out 1
define net prev_out 4
define net plus_out 4
define net conj_out 4
define net a[3:0] 4
define net a[7:4] 4
define cell SLICE a[7:4] a 7 4
define cell SLICE a[3:0] a 3 0
define cell ADD plus_out a[7:4] a[3:0]
define cell AND conj_out plus_out b
define cell DFF prev_out conj_out clk reset
define cell OR out prev_out
```

Fig. 8. Batch file for design construction with single module

```
// Example : construct_hier.dofile
create design hier_test
define module sub_test
define net a 3 0
define net a 4
define net -PI a 4
define net -PI b 4
define net -PO out 4
define cell add out a b
change module
define net -PI clk 1
define net -PI reset 1
…
define cell or out prev_out
define inst sub_test hier_inst a[7:4] a[3:0] plus_out
flat
write des
write ckt
write ckt -blif
```

Fig. 9. Batch file for hierarchical design construction with multiple modules

At first, we create a new design named “flat_design” (line 1) in Figure 8. Then we use the command “DEFine NET” to create word-level nets with widths. Parameter “[-PI | -PO | -PIO]” is used if such the net is also an I/O port. Note that some illegal names to Verilog, e.g. “a[3:0]” in line 10, will be renamed by QuteRTL; hence it is convenient for general users. Then we construct cells from line 12 to the end, which include a register with synchronous reset in line 16 (we omit the reset value and use default value). In this example, although all nets are defined before cells, actually the only restriction is that all the I/O nets of the defined cell should be defined before. Hence, users can construct a netlist with great flexibility in QuteRTL. Note that commands for cell definition can be comparably complex, due to different type of word-level cells. Users can type “HELp DEFine CELL” to see the detailed usages in the command line mode.

Next, we construct a hierarchical design with the batch file “construct_hier.dofile” in Figure 9. After constructing design “hier_test” in line 1, we define a sub-module “sub_test” in line 2. Now, our current scope is transformed into module “sub_test”. Hence all nets and cells defined in line 3-8 will be constructed in module “sub_test”. After “sub_test” is constructed, a simple command “CHAnge MODule” will bring us back to the parent module, which is “hier_test” in the case. Note that it is impossible to enter into sub-module “sub_test” again for incremental construction further. Once a sub-module is defined, we expect that it will eventually be instantiated in other modules. The command for module instantiation is “DEFine INST”, as shown in line 26, where an instance named “hier_inst” is constructed. In this command, I/O nets defined after the instance name, namely, “a[7:4], a[3:0], and plus_out”, will be connected to I/O ports of module “sub_test” in the order identical to the I/O port defined in “sub_test” previously. Hence, I/O relation of “hier_inst” will be “plus_out” = “a[7:4]” + “a[3:0]”.

Note that when building a hierarchical design through those commands, users can write out the hierarchical Verilog directly, or write out circuit after flatten, as introduced in Section 5.2.

5.4 Example: Property Checking

One of the important applications to QuteRTL is word-level verification and debugging. In the last example, we utilize QuteRTL to perform safety property checking on a simple traffic light controller. We simplify the design to only two primary inputs (clock and reset) and only one output (time_left), which shows clock cycles left before the light changes to the next. As light is changed, we reset “time_left” to the number of cycles, which is the time to keep the same light: 60 for RED, 40 for GREEN and 5 for YELLOW. Initially, light is RED and time_left is zero, so the light will turn GREEN in the next cycle. We illustrate the state transition graph of the design in Figure 10.

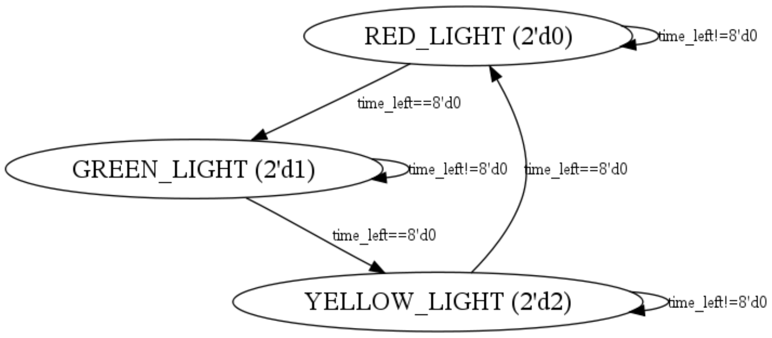

Fig. 10. State transition graph for traffic light design example

We adopt three safety properties for verifying the traffic light design: First, we assert that time_left should never exceed 60, as the longest time in the same light is 60. Second, we assert the light will never turn YELLOW, which should be proven false for this design. Finally, we assert light will never turn to an unknown state, which is encoded as “2’d3” in the design.

Figure 11 show the batch file for property checking. In line 4, 5, and 6, we set three formulas as the three safety properties, and then three “MODel CHecking” commands will call the formal engine to check whether these properties are true or not. Users can see the first and third properties are proved, and the second one is disproved with a 42-cycle trace from initial state.

```
// Example : traffic_light_check.dofile
1.  read des Traffic.v                        // Property Checking : AG(formula 1)
2.  syn                                   7.  model check 1
3.  fla                                       // Property Checking : AG(formula 2)
4.  set formula 1 "time_left <= 8'd60"    8.  model check 2
5.  set formula 2 "Light_Sign != 2'd2"        // Property Checking : AG(formula 3)
6.  set formula 3 "Light_Sign != 2'd3"    9.  model check 3
```

Fig. 11. Batch file for property checking on traffic light design example

In this example, we do not specify anything but property to model checker; however, QuteRTL allows users to change solvers and model checking algorithms by setting parameters in the command, or even to specify a file for counter-example trace dump in Value Change Dump (VCD) file format. Figure 12 shows the waveform of a counterexample for the second property. It disproves the property with a 42-cycle trace from initial state.

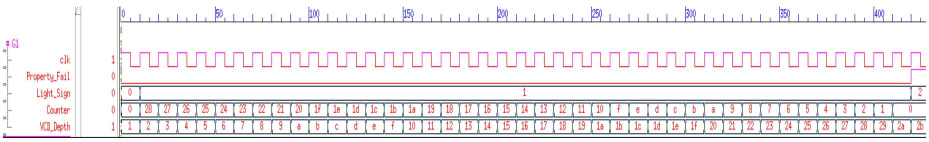

Fig. 12. The waveform of a counterexample of formula 2

6 Conclusion

We construct an open source framework for RTL design synthesis and verification, and verify the correctness and robustness of the framework with a third party tools — Cadence Conformal LEC and Berkeley ABC. With the framework, various research directions on RTL can be made possible. In the future, we will develop some techniques on RTL design debugging with the extracted design intents.

References

1. SIS, `http://embedded.eecs.berkeley.edu/pubs/downloads/sis/index.html`
2. VIS, `http://vlsi.colorado.edu/~vis/`
3. MVSIS, `http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/`
4. Berkeley ABC, `http://www.eecs.berkeley.edu/~alanmi/abc/`
5. Icarus Verilog, `http://iverilog.icarus.com/`
6. MiniSAT, `http://minisat.se/`
7. Boolector, `http://fmv.jku.at/boolector/`
8. Yeh, H.-H., Huang, C.-Y.: Automatic Constraint Generation for Guided Random Simulation. In: Asia and South Pacific Design Automation Conference, pp. 613–618 (2010)
9. Kitchen, N., Kuehlmann, A.: Stimulus generation forconstrained random simulation. In: International Conference on Computer-Aided Design, pp. 258–265 (2007)
10. Wu, B.-H., Yang, C.-J., Tso, C.-C., Huang, C.-Y.: Toward an Extremely-High-Throughput and Even-Distribution Pattern Generator for the Constrained Random Simulation Techniques. In: International Conference on Computer-Aided Design, pp. 602–607 (2011)
11. Yeh, H.-H., Wu, C.-Y., Huang, C.-Y.: Property-Specific Sequential Invariant Extraction for SAT-based Unbounded Model Checking. In: International Conference on Computer-Aided Design, pp. 674–678 (2011)
12. Thalmaier, M., Nguyen, M.D., Wedler, M., Stoffel, D., Bormann, J., Kunz, W.: Analyzing *k*-step induction to compute invariants for SAT-based property checking. In: Design Automation Conference, pp. 176–181 (2010)
13. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
14. Vizel, Y., Grumberg, O.: Interpolation-Sequence Based Model Checking. In: Formal Methods in Computer Aided Design, pp. 1–8 (2009)
15. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: Formal Methods in Computer Aided Design, pp. 125–134 (2011)
16. Liu, C.-N., Jou, J.-Y.: A FSM Extractor from HDL Description at RTL Level. In: Asia Pacific Conference on Hardware Description Languages, pp. 33–38 (1998)
17. Touati, H., Savoj, H., Lin, B., Brayton, R.K., Sangiovanni-Vincentelli, A.: Implicit State Enumeration of Finite State Machines using BDDs. In: International Conference on Computer-Aided Design, pp. 130–133 (1990)
18. OpenCores, `http://www.opencores.org`
19. Cadence Conformal LEC, `http://www.cadence.com/products/`