Debug Automation for Synchronization Bugs at RTL

Mehdi Dehbashi*
*Institute of Computer Science, University of Bremen
28359 Bremen, Germany
Email: dehbashi@informatik.uni-bremen.de

Görschwin Fey*†
†Institute of Space Systems, German Aerospace Center
28359 Bremen, Germany
Email: goerschwin.fey@dlr.de

***Abstract*—One major concern in the design of *Very-Large-Scale Integrated* (VLSI) circuits is debugging as design size and complexity increase. Automation of the debugging process helps to decrease the development cycle of VLSI circuits and consequently to achieve a higher productivity. This paper presents an approach to automatically debug synchronization bugs due to coding mistakes at RTL. In particular, we introduce an appropriate bug model and show how synchronization bugs are differentiated from other types of bugs by our approach. The experimental results on LGsynth93 and ITC-99 benchmark suites and RTL modules of OpenRISC and OpenSPARC CPUs show diagnosis accuracy and efficiency of the approach.**

***Keywords*—debug automation, synchronization bug, SAT-based debugging**

I. Introduction

Due to the increasing design size and complexity of VLSI circuits, the cost of VLSI systems verification and debugging has significantly increased. Verification tools check the correctness of a design against its specification. Upon detection of a design error, the error is returned as a counterexample. Having a counterexample, the debug process starts localizing and rectifying the bug. But this process is often a manual task which needs a large effort. Thus, automated approaches to design debugging are necessary to decrease the development cycle of VLSI products.

Design bugs at RTL are classified into three major classes: *logic bugs*, *algorithmic bugs* and *synchronization bugs* [1]. There is a range of approaches to automate the debugging process for logic bugs [2] [3] [4] [5]. Algorithmic bugs can have a severe impact on the correctness of a design and they usually require multiple major modifications to be fixed. Synchronization bugs are related to synchronization of data with respect to clock cycles in a design. For most of the synchronization bugs, a signal requires to be latched a cycle earlier or a cycle later in order to keep the correct timing behavior of signals in the design [1]. The most common fix for this class of design bugs is the manual addition or removal of flipflops to satisfy the correct timing behavior of the circuit. These bug models are called *missing flipflop* and *extra flipflop*.

In the pre-silicon stage, a design is verified against its specification by verification tools. A specification describes the correct timing behavior of a design. The work in [6] presents a formal method to specify the relations between multiple clocks and to model the possible behaviors. Then, a hardware design is verified against the specified clock constraints. An efficient clock modeling approach is presented in [7] to handle clock related challenges uniformly. The approach converts multiple clocks with arbitrary frequencies and ratios, gated clocks, multiple phases, latches and flip-flops in multi-clock synchronous system, into a single-clock model. Clock constraints are automatically generated to avoid unnecessary unrolling and loop-checks in *Bounded Model Checking* (BMC).

The work in [2] presents a model based on Boolean satisfiability to automate debugging of logic bugs. A circuit is enhanced with correction block in order to find the potential fault candidates. Abstraction and refinement techniques are used in [8] for handling the automated debugging of large designs with a better performance and reduced memory consumption. The work in [3] uses randomly generated counterexamples for debugging and applies automatic correction based on re-synthesis. An exact debugging approach based on *Quantified Boolean Formulas* (QBF) is proposed in [4], that creates high quality counterexamples to find fault candidates fixing any erroneous behavior. In [9], a pre-silicon debugging flow is proposed for testbench-based verification environments. The approach uses diagnostic traces to obtain more effective counterexamples and to increase the diagnosis accuracy. All of the mentioned works consider logic bugs in order to automatically localize and to rectify an erroneous behavior at the pre-silicon stage.

In this paper, we present an approach to automate the debugging of synchronization bugs at RTL. This is a class of bugs introduced while coding RTL. First, synchronization bugs (extra/missing flipflop) are modeled and converted into a Boolean satisfiability formula. Having a counterexample given by a verification tool, our approach automatically extracts potential fault candidates which explain the erroneous behavior of the corresponding counterexample. The erroneous behavior can be fixed by inserting or removing some flipflops in the circuit. By this, the approach automatically investigates the number of cycles that a signal needs to be latched earlier or later fixing the erroneous behavior of the circuit. Our approach utilizes a word-level model of the circuit which may contain Boolean as well as word-level operations. In this case, a flipflop and a signal can be a word with n bits. In particular, our methodology distinguishes synchronization bugs and logic/algorithmic bugs. The experimental results show effectiveness and diagnosis accuracy of our approach on LGsynth93 and ITC-99 benchmark suites and modules of OpenRISC and OpenSPARC processors. For OpenSPARC we show how our approach performs on a real synchronization bug.

The remainder of this paper is organized as follows. Section II introduces preliminary information on SAT-based debugging. Bug models are explained in Section III. Our debugging methodology is presented in Section IV. Section V explains how to model the correction of synchronization bugs in order to automatically debug a design using Boolean satisfiability. Then, the debugging algorithm is demonstrated in Section VI. Section VII presents experimental results on benchmark circuits. The last section concludes the work.

II. Bug Model-Free SAT-Based Debugging

When an implemented circuit fails verification, debugging procedure starts to find root causes of the observed error. Verification output is a set of counterexamples proving the existence of a bug in the circuit. A circuit is composed of components which specify the granularity of the debugging result. Gates or modules are considered as typical components, but also the hierarchical and structural information can be taken into account [10], [11], [2]. In examples of this paper, gates are considered as components for the sake of simplicity.

In [2], a SAT-based debugging approach is proposed. The approach searches for all possible fault candidates in the faulty circuit. To do debugging, first the faulty circuit is enhanced with correction block. Correction block is a multiplexer which is added at the output of each component. As Figure 1 shows,

This work has been supported in part by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative and in part by the German Research Foundation (DFG, grant no. FE 797/6-1).

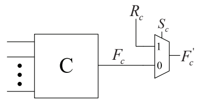


Fig. 1. Correction block

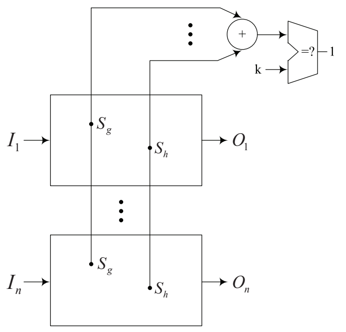


Fig. 2. Combinational debugging

the original output function F_c of component C is replaced by F_c'. The select line S_c of the correction block controls F_c' such that if S_c is activated, $F_c' = R_c$, where R_c is an unconstrained variable and a value for correcting the erroneous behavior may be injected, otherwise $F_c' = F_c$. The select line is also called *correction predicate*.

Given a faulty circuit and a set of counterexamples, one copy of the circuit is created for each counterexample. All copies are enhanced with correction blocks. Then the inputs and outputs of each created instance is constrained to inputs values and output value of the corresponding counterexample (Figure 2). In Figure 2 and Figure 3 only the select lines of the correction blocks are shown.

For debugging the SAT solver searches for a solution by activating some of correction predicates. A fault cardinality constraint controls number k of active correction predicates. The debugging procedure increases number k from 1 until a solution is found. For sequential circuits, the faulty circuit is unrolled as many time steps as the length of counterexample. In Figure 3, the length of counterexample is two time steps (two clock cycles). The correction block is added same as the combinational case. For sequential debugging, usually the same correction predicate is used for the same component in all time steps and for all counterexamples.

III. Synchronization Bug Model

Synchronization bugs occur when a signal is latched a cycle earlier or a cycle later in comparison to correct timing constraints. When a signal needs to be latched a cycle later, this behavior indicates lacking a flipflop on the corresponding signal. This bug model is called *missing flipflop*. When a signal requires to be latched a cycle earlier, this may be due to an additional flipflop on the corresponding signal. This bug model is called *extra flipflop*.

We denote a flipflop as $b = FF(a, clk, init)$, where a is the data input of the flipflop, clk indicates the clock frequency of the corresponding flipflop, and $init$ is the initial state of the flipflop. The data output of the flipflop is denoted by b. When a circuit has a single clock, and the initial states of flipflops are 0, a flipflop is denoted as $b = FF(a)$. In our framework, a flipflop and a signal can be a word with n bits as we use a word-level model of the circuit [12].

In the case of a missing flipflop on signal s, the correction of this bug is adding one flipflop on the signal in order to postpone the propagation of the corresponding signal one clock cycle. The correction of this synchronization bug is shown as $s_2 =^{+1} s_1$, where $=^{+1}$ is the correction flipflop, and signal s is decomposed to signals s_1 and s_2 in order to

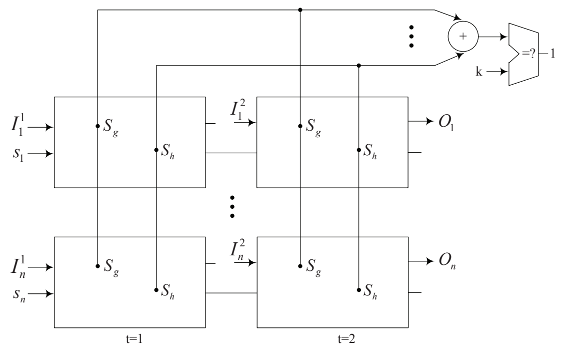


Fig. 3. Sequential debugging

add a flipflop on the propagation path of the signal. If signal s requires to be latched m clock cycles later, the correction is denoted as $s_2 =^{+m} s_1$, which indicates adding m consecutive flipflops on the propagation path of signal s.

When there is an extra flipflop $b = FF(a)$ in the circuit, deleting the corresponding flipflop causes the signal to be latched a clock cycle earlier. In this case, the correction is indicated as $b =^{-1} a$, where $=^{-1}$ shows a correction by removing the flipflop. Removing m consecutive flipflops in order to latch a signal m clock cycles earlier is denoted by $b =^{-m} a$.

IV. Debugging Methodology

At the design step, verification tools check the correctness of an implemented circuit according to the specification. If there is a contradiction between the behavior of the implemented circuit and the specification, this contradiction is returned as a counterexample.

Our debugging methodology is shown in Figure 5. The first step in Figure 5 shows the verification process. The outputs of the circuit and the specification are denoted by O_c and O_s, respectively. The input test vector is denoted by I. If the circuit and the specification have different output values, while the same test vector is applied on inputs, this difference indicates a counterexample. A counterexample is shown by $CE(I, O_c, O_s)$. A specification can be a formal specification, a simulation-based specification or golden data. We assume that all inputs and initial states are constrained by parameter I. Therefore there is no free input or free initial state in the circuit. The verification function is written as follows:

$$CE = Verification\,(Circ,\, Spec) \qquad (1)$$

Having a counterexample $CE(I, O_c, O_s)$, debugging starts. Here debugging uses the circuit and golden output values O_s in order to localize a bug.

In this case, first a new debugging instance is created without considering any bug model. The approach in [10] is utilized in order to localize a bug. In the approach, a multiplexer is inserted at the output of each component. When a multiplexer is activated, a new value is inserted at the output of the corresponding component fixing the erroneous behavior. This process is denoted as follows:

$$FCs = DBG\,(ModelFree,\, Circ,\, CE) \qquad (2)$$

The inputs of DBG are a circuit, a counterexample CE and a parameter $ModelFree$ which indicates bug model-free debugging should be performed to extract the set of fault candidates FCs. Bug model-free debugging was explained in Section II.

Given the set of potential fault candidates FCs, we investigate whether a fault candidate $FC \in FCs$ can be a *synchronization fault candidate* according to the synchronization bug model. *Synchronization correction block* is inserted at the

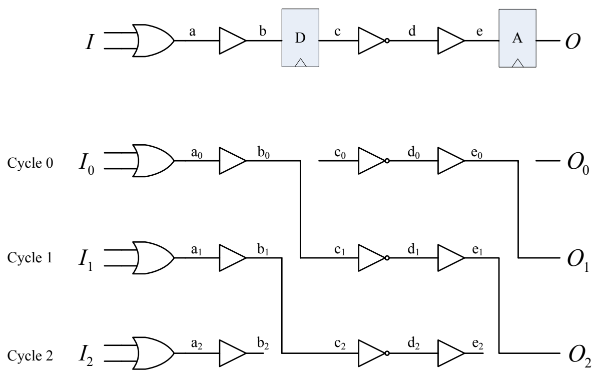


Fig. 4. Unrolled Correct Circuit

output of each fault candidate $FC \in FCs$. Synchronization correction block changes the behavior of a fault candidate according to the synchronization bug model. Synchronization correction blocks are explained in Section V in detail. If by activating a synchronization correction block, the erroneous behavior of the circuit is fixed, a *synchronization fault candidate* FC' is detected. This process is denoted as follows:

$$FCs' = DBG\,(Synch,\, Circ,\, CE,\, FCs) \tag{3}$$

The inputs of DBG in this process are a circuit, counterexample CE, set of fault candidates FCs and parameter $Synch$ which indicates the synchronization bug model is used for debugging. The output of the process is the set of synchronization fault candidates $FCs' \subseteq FCs$.

In the following steps of the debug flow, if the set of synchronization fault candidates is not null, this set (FCs') is returned as fault candidates for synchronization bugs. Otherwise, the set of fault candidates FCs is returned as fault candidates for other kinds of bugs, i.e., logic bugs and algorithmic bugs.

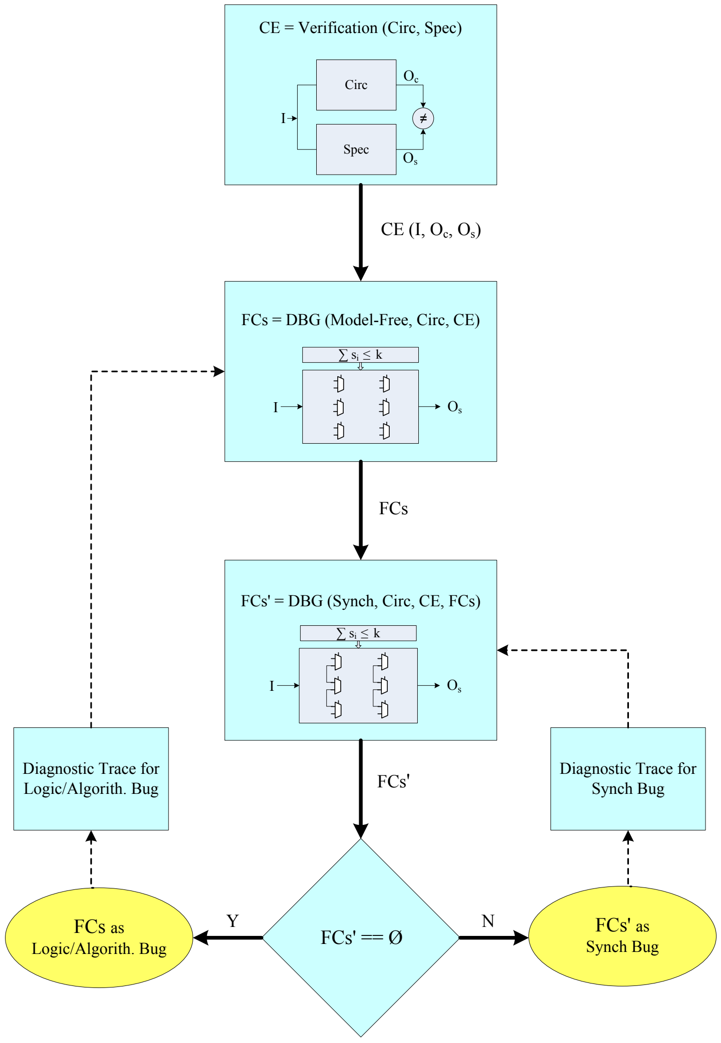
In Figure 5, dashed lines show some additional processes in order to increase diagnosis accuracy of debugging. Having the set of logic/algorithmic fault candidates, diagnostic traces can be generated (left dashed branch in Figure 5). Diagnostic traces differentiate the behavior of fault candidates and help to create high quality counterexamples. High quality counterexamples help debugging to decrease the number of fault candidates. One approach to generate diagnostic traces is presented in [9]. The approach does not need a bug model to generate diagnostic traces. Counterexamples obtained by diagnostic traces are used to iterate debugging and to increase the diagnosis accuracy.

In the case of synchronization fault candidates (right dashed branch in Figure 5), *synchronization diagnostic traces* can be generated to distinguish the behavior of synchronization fault candidates. Synchronization diagnostic traces can be generated according to the synchronization bug model. Counterexamples obtained by synchronization diagnostic traces help debugging to decrease the number of synchronization fault candidates. Diagnostic traces can be generated similar to other fault models [13]. In this work we focus only on the synchronization bug model and debugging algorithm without using diagnostic traces.

Our approach can also be used for circuits with multiple clock domains. In this case, first multiple clocks are converted into a single-clock model [7]. Having a single-clock model, our approach is utilized to localize the root cause of an error.

V. Synchronization Correction Block

For debugging, first the circuit is unrolled as many times as the number of clock cycles constituting the corresponding



CE = Verification (Circ, Spec)
Circ
Spec
I
Oc
Os
≠
CE (I, Oc, Os)
FCs = DBG (Model-Free, Circ, CE)
∑ si ≤ k
I
Os
FCs
FCs' = DBG (Synch, Circ, CE, FCs)
∑ si ≤ k
I
Os
FCs'
Diagnostic Trace for Logic/Algorith. Bug
Diagnostic Trace for Synch Bug
FCs as Logic/Algorith. Bug
Y
FCs' == Ø
N
FCs' as Synch Bug

Fig. 5. Debugging Methodology

counterexample. For example, if the length of the counterexample is three clock cycles, the circuit C is unrolled three times: C_0, C_1 and C_2. In this case, the input of a flipflop from clock cycle i is connected to the appropriate gates in clock cycle $i+1$. In the example of Figure 4, there is a circuit including two flipflops A and D. To do debugging, the circuit is unrolled three times. Flipflops A and D are removed. The input wire of flipflop D at cycles 0 and 1, b_0 and b_1, are connected to the output wire of the corresponding flipflop at cycles 1 and 2, c_1 and c_2, respectively.

In the example circuit of Figure 6, there is a missing flipflop bug. The location of the bug is shown by a red circle. To debug the circuit, it is copied three times. The input of flipflop A at cycle i is connected to the appropriate signal at cycle $i+1$. For debugging, we investigate at which wire of the circuit a flipflop is missing. Therefore, we need correction block at each wire of the circuit which is able to model the behavior of a flipflop at the corresponding wire. The green part in Figure 6 shows this model. Multiplexers are utilized to model a flipflop behavior at every wire of the circuit. When there is a set of fault candidates FCs given by bug model-free SAT-based debugging, correction blocks are inserted only on each fault candidate $FC \in FCs$. To model correction block for m consecutive missing flipflops, m consecutive correction blocks can be inserted on each wire.

If select line s at wire p is active ($s = 1$), a flipflop behavior at wire p is activated. Therefore, the output of the activated flipflop at cycle i is connected to the input of the corresponding

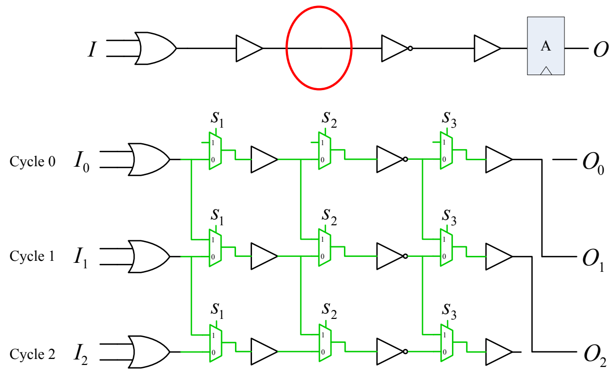


Fig. 6. Debugging Instance for Missing FF Bug

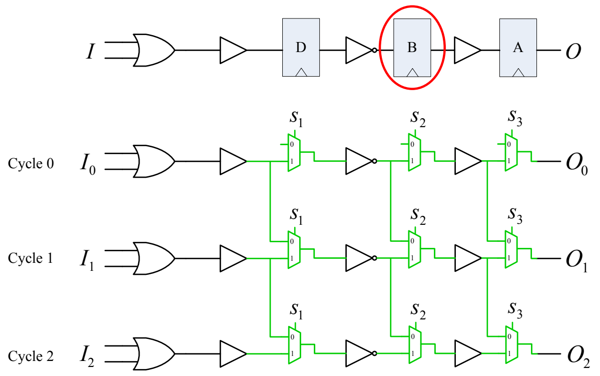


Fig. 7. Debugging Instance for Extra FF Bug

flipflop at cycle $i-1$. If the correction block is inactive ($s = 0$), the circuit at wire p has the behavior of a normal wire.

Having correction block at every wire of the circuit, the inputs and outputs of the model are constrained according to the input and output values of the corresponding counterexample. Then debugging answers the following question by activating the select lines: If a flipflop is added at wire p, can the erroneous behavior of the corresponding counterexample be fixed? In this case, a SAT solver is utilized to extract all possible fault candidates.

Figure 7 shows the model for the extra flipflop bug. In case, correction block is applied at the location of every flipflop in the circuit. Correction block for the extra flipflop bug has the reverse behavior in comparison to correction block for the missing flipflop bug. For this kind of bug, debugging investigates if a flipflop is removed, can the erroneous behavior of the corresponding counterexample be fixed? If there is a set of fault candidates FCs given by bug model-free SAT-based debugging, correction blocks are inserted only on flipflops which are fault candidates in set FCs.

When there is both a missing flipflop bug and an extra flipflop bug in a design, correction blocks for both kinds of bugs are required simultaneously. In this case, missing flipflop correction block is inserted on every wire of the circuit, while extra flipflop correction blocks are inserted only at existing flipflops. The whole timing variation is controlled by constraining the select lines of correction blocks.

For single bugs, the following constraint is added to the created debugging instance to control the select lines of multiplexers:

$$\sum_{i=1}^{n} s_i \leq 1 \tag{4}$$

Parameter n is the total number of correction blocks in the debugging instance. For multiple bugs, k bugs may be activated at a same time. In this case, the following constraint controls the behavior of multiplexers (correction blocks):

$$\sum_{i=1}^{n} s_i \leq k \tag{5}$$

VI. Algorithm

Figure 8 shows the algorithm of our synchronization debugging in pseudocode. The inputs of function debugging are a circuit, a counterexample CE and a set of fault candidates FCs given by bug model-free SAT-based debugging. The output of the function is a set of synchronization fault candidates FCs'. Set FCs' is a union of three sets $\mathcal{F}'_{Extra}$, $\mathcal{F}'_{Missing}$ and $\mathcal{F}'_{Mixed}$ (line 2). Set $\mathcal{F}'_{Extra}$ represents the set of extra flipflop fault candidates. Set $\mathcal{F}'_{Missing}$ represents the set of missing flipflop fault candidates. When a fault candidate is composed of multiple components such that some components of the fault candidate are missing flipflops while other components of the fault candidate are extra flipflops, this fault candidate is included in set $\mathcal{F}'_{Mixed}$. In the algorithm, the maximum number of synchronization bugs is given by parameter k_{max} (line 3) which limits variable k in Equation 5.

At the first step of the algorithm, correction blocks (CL) for extra flipflop and missing flipflop models are inserted at the locations of fault candidates $FC \in FCs$ (line 5). Set S_E denotes the set of select lines for extra flipflop correction blocks. Set S_M denotes the set of select lines for missing flipflop correction blocks. The whole set of select lines is stored in set S (lines 6-7). Line 8 constrains the inputs and the output of the created instance to the values given by the counterexample CE. In line 10, sets $\mathcal{F}'_{Extra}$, $\mathcal{F}'_{Missing}$ and $\mathcal{F}'_{Mixed}$ are initialized. In line 11, variable k is initialized to 1. Variable k indicates the number of bugs in the design. First the algorithm starts with the assumption of single bug ($k = 1$). The select lines of multiplexers are controlled by the constraint line 14. Then a SAT solver is called to find a solution (line 15). If there is any solution, all solutions will be extracted (line 17). The solutions are saved in set $Solutions$. Otherwise, the previous constraint of select lines is removed (line 22) and k increases (line 23). If variable k is less equal than parameter k_{max} (line 25), the algorithm iterates until finding a solution or reaching k_{max}.

After finding solutions, the solutions are categorized into three sets $\mathcal{F}'_{Extra}$, $\mathcal{F}'_{Missing}$ and $\mathcal{F}'_{Mixed}$ (lines 27-35). If all components of a solution Sol are included in set S_E (line 29), the fault candidate is considered as extra flipflop fault candidate (line 30). If all components of a solution Sol are included in set S_M (line 31), the fault candidate is considered as missing flipflop fault candidate (line 32). Otherwise, the fault candidate has some extra flipflop components and some missing flipflop components and is added to set $\mathcal{F}'_{Mixed}$ (lines 33-34).

VII. Experimental Results

In this section, we demonstrate the effects of our debugging approach experimentally on sequential circuits of LGsynth93 and ITC-99 benchmark suites and RTL modules of OpenRISC and OpenSPARC CPUs. The single synchronization bugs are randomly injected by removing a flipflop (Missing FF) or by adding a flipflop (Extra FF). The single logic bugs are randomly injected by replacing gates. For example an AND gate is replaced by an OR gate. For bounded sequential debugging, the circuits are unrolled up to thirty time steps.

```
function DBG(Synch, Circ, CE, FCs)
Output: $FCs' = \mathcal{F}'_{Extra} \cup \mathcal{F}'_{Missing} \cup \mathcal{F}'_{Mixed}$
Maximum Number of Synchronization Bugs: $k_{max}$

$\{S_E, S_M\} = Insert\_Extra\_Missing\_CLs(Circ, FCs)$
$S = S_E \cup S_M$
$s_i \in S,\ i = 1, 2, \ldots, n$
$Insert\_Constraint(CE)$

$\{\mathcal{F}'_{Extra}, \mathcal{F}'_{Missing}, \mathcal{F}'_{Mixed}\} = \{\emptyset, \emptyset, \emptyset\}$
$k = 1$
do
{
  $Insert\_Constraint(\sum_{i=1}^{n} s_i \leq k)$
  if $Solve() == SAT$ then
  {
    $Solutions = Extract\_All\_Solutions()$
    break
  }
  else
  {
    $Remove\_Constraint(\sum_{i=1}^{n} s_i \leq k)$
    $k = k + 1$
  }
} while $k \leq k_{max}$

foreach $Sol \in Solutions$ do
{
  if $\forall s_i \in Sol : s_i \in S_E$ then
    $\mathcal{F}'_{Extra} = \mathcal{F}'_{Extra} \cup Sol$
  else if $\forall s_i \in Sol : s_i \in S_M$ then
    $\mathcal{F}'_{Missing} = \mathcal{F}'_{Missing} \cup Sol$
  else
    $\mathcal{F}'_{Mixed} = \mathcal{F}'_{Mixed} \cup Sol$
}

return $\{\mathcal{F}'_{Extra} \cup \mathcal{F}'_{Missing} \cup \mathcal{F}'_{Mixed}\}$
end function
```

Fig. 8. Synchronization Debugging Algorithm

The number of unrolling steps for verification, model-free debugging and synchronization debugging is same for a given circuit.

The experiments are carried out on a Dual-Core AMD Opteron(tm) Processor 2220 SE (2.8 GHz, 32 GB main memory) running Linux. The techniques described in this paper are implemented using C++ in the WoLFram environment [12]. MiniSAT is used as underlying SAT solver [14]. Run time is measured in CPU seconds, and the memory consumption is measured in MB.

Having a buggy circuit, the debugging methodology of Figure 5 is called. The verification process returns an initial counterexample. Then, bug model-free SAT-based debugging searches for the potential fault candidates FCs. The set FCs is given to the synchronization debugging in order to find the synchronization fault candidates FCs'. If set FCs' is null, there are no single synchronization bugs according to the synchronization bug model.

Figure 9 shows a real synchronization bug reported in the Verilog code of the OpenSPARC processor [1]. The upper rectangle in Figure 9 indicates the correct code while the lower rectangle (the commented lines) indicates the buggy code. In the buggy code, the value of the 48-bit $tlb\_st\_data\_d1$ bus is assigned to the $lsu\_ifu\_stxa\_data$ bus in the same cycle. However, in the correct code, the data requires to be latched for one clock cycke between the two buses [1]. To evaluate our approach, first we activate the bug in the Verilog file $lsu\_qdp1.v$. The buggy Verilog file is given to our debugging approach to find fault candidates. Having one counterexample given by verification, bug model-free debugging finds three fault candidates in the buggy circuit. These three fault candidates are given to synchronization debugging. Synchronization debugging proves that all of these three fault candidates can be synchronization fault candidates (Figure 10). The synchronization fault candidates correspond to the buggy line in the Verilog code. Totally the approach spends 340.4 seconds and consumes 78.1 MB memory. Note that the

```
dff_s  #(48) ifu_std_d1 (
        .din    (tlb_st_data[47:0]),
        .q      (lsu_ifu_stxa_data[47:0]),
        .clk    (asi_data_clk),
        .se     (1'b0),      .si (),         .so ()
        );
`endif
```

```
// select is now a stage earlier, which should be
// fine as selects stay constant.
//assign  lsu_ifu_stxa_data[47:0] = tlb_st_data_d1[47:0] ;

// End - Bug3487.
```

Fig. 9. Synchronization bug in OpenSPARC processor, Verilog file $lsu\_qdp1.v$

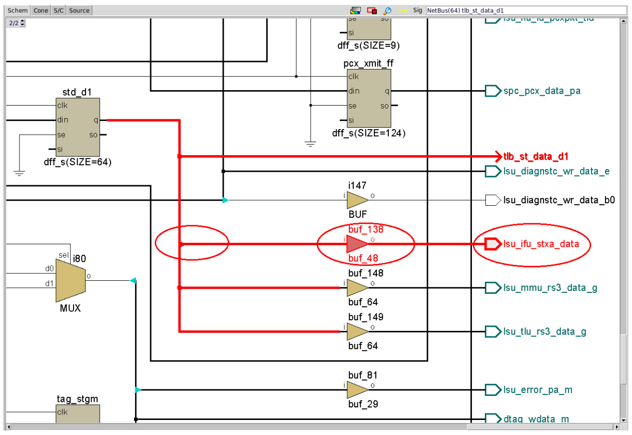


Fig. 10. Synchronization fault candidates in the circuit lsu_qdp1

visualized schema of the design is not a synthesized gate-level circuit. It is an intermediate representaion of the Verilog design which is used to convert a Verilog design to CNF (SAT formula) at word level [12]. We do not synthesize the Verilog designs to debug them.

Table I presents the experimental results for single faults which have been injected randomly. There are four sections in Table I. Section $Benchmarks$ shows the characteristics of the benchmarks (columns 1-3). Other sections show the experimental results when a missing flipflop, an extra flipflop or a logic bug is injected. Each section shows the final number of fault candidates ($\#FC$), the final number of synchronization fault candidates ($\#FC'$), the required run time ($Time$) and the maximum memory consumption (Mem). The time includes the verification time ($Ver.$), the time for bug model-free debugging ($DbgM$), the time for synchronization debugging ($DbgS$) and the total time ($Total$).

Consider the greatest common divider gcd in table; when there is a missing flipflop in circuit gcd, bug model-free debugging finds ten fault candidates ($\#FC = 10$). These ten fault candidates are given to synchronization debugging. Then synchronization debugging investigates whether a fault candidate $FC \in FCs$ can be a synchronization fault candidate according to the synchronization fault model. The output of synchronization debugging is a set of synchronization fault candidates $FCs' \subseteq FCs$. For circuit gcd, synchronization debugging finds six synchronization fault candidates ($\#FC' = 6$). The number of fault candidates ($\#FC$) is by definition always larger equal than the number of synchronization fault candidates ($\#FC'$). This is one reason that causes the debugging time to extract set FCs (column $DbgM$) to be longer than the time to extract set FCs' (column $DbgS$).

For extra flipflop bugs, in most of the cases, there are fewer synchronization fault candidates in comparison to the cases of missing flipflop bugs (comparison of column 5 and column 12). The reason is that flipflops may only be removed at places where they are located in the design. But adding of flipflops is possible at any signal. So the number of flipflops that can potentially be added is much larger than the number of flipflops that can potentially be removed.

TABLE I
EXPERIMENTAL RESULTS

Benchmarks			Missing FF							Extra FF							Logic Bug						
Circuit	#Gates	#FF	#FC	#FC'	Time				Mem	#FC	#FC'	Time				Mem	#FC	#FC'	Time				Mem
					Ver.	DbgM	DbgS	Total				Ver.	DbgM	DbgS	Total				Ver.	DbgM	DbgS	Total	
b01	49	5	8	3	0.0	0.2	0.0	0.2	15.5	8	1	0.1	0.2	0.0	0.3	15.6	7	0	0.0	0.1	0.0	0.1	15.4
b02	25	4	13	3	0.0	0.2	0.1	0.3	15.6	6	1	0.0	0.1	0.0	0.2	15.5	16	0	0.0	0.2	0.0	0.3	15.6
b04	707	66	26	9	2.4	11.1	1.0	14.4	17.6	36	12	2.5	20.2	1.2	23.8	17.6	18	0	14.7	35.7	2.5	52.8	19.3
b05	1054	34	22	7	0.7	6.7	0.5	7.9	16.5	10	2	0.7	4.3	0.3	5.3	16.4	2	0	0.7	2.3	0.2	3.2	16.3
b08	177	21	39	11	1.2	7.9	1.3	10.5	18.0	13	2	0.4	1.6	0.2	2.2	16.3	6	0	0.1	0.4	0.1	0.5	15.9
b10	211	17	55	5	0.5	11.1	0.6	12.3	17.3	4	1	0.1	0.4	0.1	0.6	15.8	3	0	0.1	0.6	0.1	0.7	15.8
b11	790	31	43	10	21.3	70.0	5.0	96.4	19.6	49	9	3.0	59.4	1.3	63.8	17.5	8	0	2.9	7.3	0.5	10.6	17.0
b12	1062	121	68	11	7.4	92.1	2.1	101.6	19.3	61	7	8.8	82.8	1.7	93.3	19.2	114	0	42.3	488.9	8.7	539.9	24.1
gcd	1012	59	10	6	0.7	6.4	0.5	7.5	16.4	3	1	8.1	16.5	0.9	25.6	17.9	6	0	17.0	36.0	2.8	55.7	19.1
phase_de.	1672	55	20	10	18.5	41.8	4.5	64.7	19.3	115	24	61.6	904.3	17.7	983.6	21.9	29	0	15.5	86.5	2.4	104.4	18.9
or1200_ctrl	5093	198	7	3	50.2	222.9	14.1	287.2	39.3	6	1	70.7	191.8	6.2	268.7	32.0	7	0	16.6	107.2	2.1	125.9	26.0
or1200_if	2029	69	7	4	2.9	19.5	1.0	23.4	20.6	8	3	3.3	24.5	1.1	28.8	22.7	6	0	5.3	27.4	0.7	33.4	22.7
or1200_lsu	1777	8	6	3	25.2	76.1	6.1	107.4	26.6	21	5	39.5	170.4	12.5	222.4	30.2	10	0	22.5	83.3	5.5	111.3	26.2
or1200_operand.	1485	66	13	6	0.9	13.4	0.8	15.0	20.1	14	7	1.0	12.8	0.9	14.7	19.9	12	0	0.6	9.4	0.3	10.2	18.6

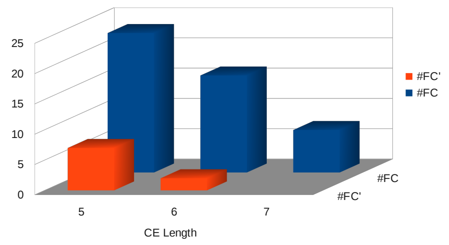


Fig. 11. Effect of CE length on distinguishing logic bug and synchronization bug (circuit b01)

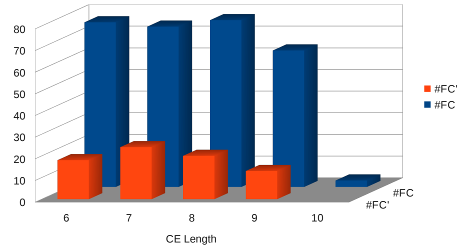


Fig. 12. Effect of CE length on distinguishing logic bug and synchronization bug (circuit b10)

When there is a logic bug in circuit $b01$, the number of fault candidates is seven ($\#FC = 7$), while the number of synchronization fault candidates is zero ($\#FC' = 0$). When there is no synchronization fault candidate, our algorithm can distinguish a logic bug from a synchronization bug. We detect that there is no synchronization bug in the circuit according to the synchronization bug model. In this case, the fault candidates FCs are returned as logic fault candidates as shown in Figure 5.

In principle, a logic bug may be correctable by a synchronization correction block with respect to the given counterexample. But in the examples of Table I, this case does not occur because of the length of counterexamples. The accuracy of distinguishing between a logic bug and a synchronization bug depends on the length of the counterexample, i.e., the number of clock cycles relevant to the counterexample. In Figure 11, we show the relation of counterexample length and distinguishing logic bugs versus synchronization bugs for circuit $b01$. When there is a logic bug in circuit $b01$, some random counterexamples with different lengths are generated. The diagram shows the number of synchronization fault candidates is zero when a counterexample is longer (CE length = 7). Therefore in this case, we can distinguish a logic bug from a synchronization bug. Figure 12 shows the effect of counterexample length on distinguishing logic bugs versus synchronization bugs for circuit $b10$. In this case when the length of the counterexample is 10, we can distinguish a logic bug from a synchronization bug. Not only the length of a counterexample but also the number and the quality of counterexamples can influence the accuracy of the classification.

VIII. CONCLUSION

We introduced an approach to automate debugging for synchronization bugs at *Register Transfer Level* (RTL). The correction for synchronization bugs is modeled in order to automatically debug a design using *Boolean satisfiability* (SAT). The approach automatically investigates the number of cycles that a signal needs to be latched earlier or later fixing the erroneous behavior of the circuit. Our methodology distinguishes synchronization bugs and logic/algorithmic bugs.

REFERENCES

[1] K. Constantinides, O. Mutlu, and T. M. Austin, "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation," in *International Symposium on Microarchitecture*, 2008, pp. 282–293.
[2] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.
[3] K. Chang, I. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," in *ASP Design Automation Conf.*, 2007, pp. 944–949.
[4] A. Sülflow, G. Fey, and R. Drechsler, "Using QBF to increase accuracy of SAT-based debugging," in *IEEE International Symposium on Circuits and Systems*, 2010, pp. 641–644.
[5] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, and R. Drechsler, "Increasing the accuracy of SAT-based debugging," in *Design, Automation and Test in Europe*, 2009, pp. 1326–1332.
[6] E. M. Clarke, D. Kroening, and K. Yorav, "Specifying and verifying systems with multiple clocks," in *Int'l Conf. on Comp. Design*, 2003, pp. 48–55.
[7] M. K. Ganai and A. Gupta, "Efficient BMC for multi-clock systems with clocked specifications," in *ASP Design Automation Conf.*, 2007, pp. 310–315.
[8] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Design, Automation and Test in Europe*, 2007, pp. 1182–1187.
[9] M. Dehbashi, A. Sülflow, and G. Fey, "Automated design debugging in a testbench-based verification environment," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 206–217, 2013.
[10] M. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.
[11] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.
[12] A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler, "WoLFram – a word level framework for formal verification," in *IEEE/IFIP Int'l Symposium on Rapid System Prototyping*, 2009, pp. 11–17.
[13] Y.-C. Lin, F. Lu, and K.-T. Cheng, "Multiple-fault diagnosis based on adaptive diagnostic test pattern generation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 5, pp. 932–942, 2007.
[14] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.