Formal Reset Recovery Slack Calculation at the Register Transfer Level

Chih-Neng Chung
GIEE
National Taiwan University
Taipei, Taiwan

Chia-Wei Chang
Department of EE
National Central University
Jhongli, Taiwan

Kai-Hui Chang
Avery Design Systems, Inc.
Andover, MA, USA

Sy-Yen Kuo
GIEE
National Taiwan University
Taipei, Taiwan
Email: sykuo@cc.ee.ntu.edu.tw

***Abstract*—Reset is one of the most important signals in many designs. Since reset is typically not timing critical, it is handled at late physical design stages. However, the large fanout of reset and the lack of routing resources at these stages can create variant delays on different targets of the reset signal, creating reset recovery problems. Traditional approaches address this problem using physical design methods such as buffer insertion or rerouting. However, these methods may invalidate previous optimization efforts, making timing closure difficult. In this work we propose a formal method to calculate reset recovery slacks for registers at the register transfer level. Designers and physical design tools can then utilize this information throughout the design flow to reduce reset problems at later design stages.**

I. Introduction

Reset brings a chip into a known state so that the chip can function properly. Although this signal is important, it is used infrequently and is typically not timing critical. Therefore, this signal is often routed after most physical design process is done. However, routing resource may be scarce at this point, forcing reset nets to take long detours that can create propagation delays which may be large enough for registers to have their reset de-asserted at different cycles. This can create incorrect reset states and can render the chip useless. One way to address this problem is to resort to physical design practices such as buffer insertion or rerouting of some nets. However, doing so may create perturbations significant enough to invalidate previous timing-optimization efforts. In practice we have seen physical design teams spending weeks trying to solve this problem but ended up removing the reset to some registers. To this end, techniques that automate the reset removal process have been proposed [7]. However, the scalability of such techniques is an issue.

In this work we take a different approach by calculating "reset recovery slacks", or "reset slacks", at the Register Transfer Level (RTL). The reset slack of a register is the number of cycles that its reset de-assert event can be delayed without affecting the correctness of the reset sequence. Our approach is based on symbolic simulation [3], [4] and supports both fixed as well as flexible reset sequences. Therefore, it can be applied to various types of designs and reset schemes. By utilizing the flexibility provided by reset slacks, better circuit optimization and faster timing closure can be achieved.

II. Background

In this section, we explain the concept of Boolean quantification and then review existing solutions for the reset problem.

A. Boolean Quantification

Quantification is an operation that eliminates variables in a Boolean formula. There are two types of quantifications: universal ($\forall$) and existential ($\exists$). We only use the latter in this work and it is defined as follows. Given a function F and inputs $x_1 ... x_n$. Suppose x_j is existentially quantified, then:

$$\exists x_j : F(x_1, x_2, ...x_j...x_n) = F(x_1, x_2, ...0, ...x_n) \vee F(x_1, x_2, ...1, ...x_n)$$

Quantification is used to handle flexible inputs so that all possible values are considered during reset slack calculation.

B. Reset Problems and Existing Solutions

Two excellent introductions of different reset schemes and the associated problems can be found from [8], [9]. In particular, reset recovery delays due to long reset nets can cause registers to enter metastable states and corrupt reset correctness. One solution is to use physical design methods but doing so can perturb existing timing optimizations [5]. Another solution is to reset only part of the registers [7]. However, such an approach may suffer reset nondeterminism problems [6]. Alternatively, circuits designed specifically to solve the reset problem have been proposed, such as [2] and [10].

III. Reset Slack Calculation

In this section we first formulate the reset recovery slack calculation problem, and then propose two algorithms to solve the problem: one for fixed inputs and one for flexible inputs.

A. Problem Formulation

In this work we focus on the RTL and consider reset recovery delay based on clock cycles. We illustrate the reset recovery slack calculation problem using an example shown in Figure 1. There are three registers in the example: "Reg1", "Reg2" and "State Reg". "State Reg" is a register whose value must be correct after reset. We call such registers "key registers" in this work and assume that they are given.

To model reset propagation delay to "State Reg", we introduce "reset cutpoints" as the "Delay" block in the figure shows for delay insertion. The original reset signal is named "Reset" and the delayed version is named "Reset2" after the cutpoint. In this work we inject a symbol to model arbitrary delay and use symbolic simulation [3] to analyze its effect. From the timing diagram on the left of Figure 1, we can see that due to the delay, reset recovery (or de-assert) time for "State Reg" is one cycle later than Reg1 or Reg2. If this circuit can still work correctly even though reset recovery is 1 to N cycles late for a reset cutpoint, then we say that the registers after the cutpoint have reset slack N. The goal of this work is to find slacks for all reset cutpoints.

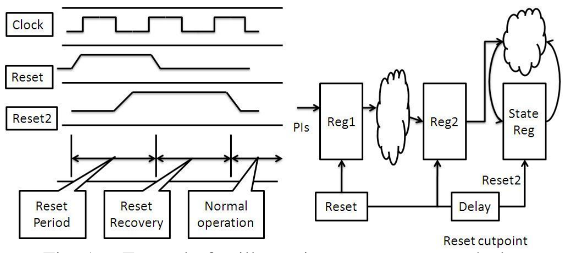

Fig. 1. Example for illustrating reset recovery slacks.

Formally, the problem for deriving reset recovery slack can be formulated as follows. Given a circuit, a set of reset cutpoints and the maximum reset recovery delay, find the maximum slack for each reset cutpoint (up to the maximum reset recovery delay) so that no matter what the delays are for those cutpoints, as long as the delays are within the slacks then the design will still be reset correctly. In this work, "reset correctly" means the reset state when delays exist matches the state when no delay exists. Note that we do not consider reset assert delay problems by assuming all registers are reset simultaneously at some point during the reset sequence — this clears pre-reset nondeterminism.

B. Reset Slack Calculation — Fixed Inputs

Our reset slack calculation algorithm for fixed inputs is shown in Figure 2. Its inputs are a set of key registers $regs$ and the maximum allowed slack m. Its output is the slack of each register, which is saved in the register's $slack$ field. In our approach, we first inject delay symbols into reset cutpoints to model reset recovery delay. From line 2 to line 7 we perform symbolic simulation for m cycles using the given fixed input patterns. At each cycle, we check the symbolic trace of each register reg that is still in the $regs$ to see if its symbolic trace is a constant. If so, the register has a slack at least as large as $cycle$ and can remain in $regs$. Otherwise, the maximum slack of the register is $cycle - 1$ and is removed from $regs$. After symbolic simulation finishes, if there are still registers in $regs$, then their slacks are at least m. Therefore, on lines 8-9 we assign m as the slacks of those registers.

To implement function $strace\_check\_constant$, we build a SAT instance from the symbolic trace and then use a SAT solver to check if the trace can have different output values.

The reason why a constant symbolic trace at cycle $cycle$ represents a reset slack at the cycle is because it means the value of the register is not affected by the symbols we injected to model reset recovery delays. In other words, the reset state will not be affected no matter what the delays are. Once the slack for each register is known, the slack for a design block can be calculated by finding the minimal slack among all the block's registers. This algorithm can be applied to designs

```
procedure $fixed\_input\_slack\_calculation(regs, m)$
1   inject reset recovery delay symbols;
2   for ($cycle$= 1; $cycle \leq m$; $cycle$++)
3     symbolic simulate one cycle;
4     foreach ($reg \in regs$)
5       if ($strace\_check\_const(reg.strace)$ == false)
6         $reg.slack = cycle - 1$;
7         $regs$ = $regs \setminus reg$;
8   foreach ($reg \in regs$)
9     $reg.slack = m$;
10  return;
```

Fig. 2. Reset slack calculation for known and fixed inputs.

where reset sequences are known and are found to be scalable because only a few symbols are injected.

C. Reset Slack Calculation — Flexible Inputs

In this section we propose an algorithm to handle reset sequences that contain flexible (unknown) inputs by building a Boolean function, called $reset\_checker$, to check whether or not there will be reset problems. Its inputs are circuit stimulus and real recovery delays at circuit operation time, and the output is "1" if there are reset problems. We then show how to use this function to calculate the reset slack for each register based on existential quantification.

1) Building Reset Correctness Checker: Reset checker examines whether values in key registers when reset recovery delays exist match those when no delays exist. In this work we assume that the constrained-random testbench generates all possible inputs that the design can have during reset and the slack period. The algorithm for building $reset\_checker$ is shown in Figure 3. We use subscript tb to represent symbols injected for the random variables to model all possible primary input values, and we use subscript $delay$ to represent symbols that model reset recovery delay. The inputs to the algorithm are the testbench ($tbench$), the design with reset recovery delay inserted (dut_{delay}), the reference design without delay (dut_{ref}), and the number of reset + slack cycles to be checked (m). The testbench generates input patterns for both dut_{delay} and dut_{ref}. We denote a register reg in dut_{delay} using reg_{delay} and the same register in dut_{ref} with reg_{ref}. The $strace$ field of the register saves the register's symbolic trace.

As shown in the algorithm, we use a design without any reset recovery delays as the golden design to obtain the correct states at each cycle. Given that the scalar values returned by $random in the testbench are replaced with symbols in our algorithm, the registers may have symbolic traces instead of scalar values. On line 1 we inject reset recovery delay symbols into dut_delay and then symbolically simulate the testbench and two designs for m cycles on lines 2-3. During symbolic simulation, whenever a $random is encountered for a variable v, we replace the scalar value returned by v with a symbol v_{tb}. Since a symbol represents all possible inputs, we use the symbol to model all the inputs that can be generated by the testbench. On lines 4-5 we build a miter using the XOR operation to check whether the symbolic traces of registers in dut_{delay} and the corresponding ones in dut_{ref} are the same. The built Boolean function is saved in $reset\_checker$, which is returned on line 6 when the algorithm finishes.

```
procedure $build\_reset\_checker(dut_{delay}, dut_{ref}, tbench, m)$
1  inject reset recovery delay symbols;
2  for ($cycle$= 1; $cycle \leq m$; $cycle$++)
3     symbolic simulate $tbench$, $dut_{delay}$ and $dut_{ref}$ one cycle
      while replacing $random for variable $v$ with symbol $v_{tb}$;
4     foreach ($reg \in dut.registers$)
5        $reset\_checker \mid= reg_{delay}.strace ^\wedge reg_{ref}.strace$;
6  return $reset\_checker$;
```

Fig. 3. Algorithm to build the reset checker function.

The first way to use the reset checker function is after the reset recovery delays are known, we can feed design inputs and reset recovery delays to this function to determine whether this particular set of input patterns will cause reset problems. Another way to use to the function is to perform existential quantification on all symbols from testbench (i.e., v_{tb}). If the Boolean function is not constant "1" after all the quantifications, then there exists at least one combination of reset recovery delays that will not cause reset problems. One can then use a SAT solver to enumerate all the delays that make the function "0" — these are all the possible safe reset recovery delays. The reason why existential instead of universal quantifications are used is to make sure when the output of the function is "0", the states in dut_{delay} will match the reference model. Since the function after existentially quantifying a variable v is "0" only when both $v = 0$ and $v = 1$ are "0", variable v can be safely eliminated without creating an incorrect checker function.

2) Calculating Reset Slacks: Although one may think that reset slacks can be calculated easily by performing existential quantification on the delay symbols, this is not true due to the binary encoding of delay symbols. For example, if bit 2 can be successfully quantified without making the checker function constant "1", it means the reset recovery delay is either 0 or 4 (assume the least significant bit is 0). However, it is not clear whether delays between 1 to 3 are acceptable. To address this problem, we need a new encoding so that a continuous range of slacks can be derived whenever one symbol bit is quantified.

To achieve this goal, we introduce a new symbol, sd, for each delay symbol $delay$. The bit-length of sd is the same as the maximum allowed slack for $delay$. The range of $delay$ is controlled by sd so that if bit n of sd (denoted as $sd[n]$) can be successfully quantified, then the register has slack up to $n + 1$ (the slack is 0 if none of the bits can be quantified).

```
procedure $delay\_encoding(delay, sd, m)$
1  for ($n$=$m$-1; $n \geq 0$; $n$- -)
2     if ($sd[n]$ == 1 && $delay > n + 1$)
3        $delay$= 0;
4  if ($sd$ == 0)
5     $delay$= 0;
6  return $delay$;
```

Fig. 4. Algorithm to generate the delay encoding function for quantification.

This can be achieved by the algorithm shown in Figure 4. The inputs to the algorithm are the original delay symbol $delay$, a new symbol sd, and the maximum allowed slack m. The function that we need will be returned as the output. Such a function can be generated using a symbolic simulator to simulate the pseudo code in the algorithm. The symbolic trace of $delay$ returned in line 6 is the required function.

With this encoding, we can calculate the reset slacks of registers much more easily, and the algorithm is shown in Figure 5. The inputs to this algorithm are the maximum slack m and the reset checker generated from algorithm $build\_reset\_checker$ with delays injected in line 1 of the algorithm replaced with the ones encoded using the $delay\_encoding$ algorithm. The outputs of the algorithm are the bits of the sd symbols that can be successfully quantified for each reset cutpoint. If the maximum bit of sd that can be quantified is n, then the corresponding reset has slack $n + 1$. In the algorithm, $checker.sds$ is a set that contains all the sd symbols generated due to our encoding of delays.

```
procedure $flexible\_input\_slack\_calculation(checker)$
1  $checker$=existentially quantify all $delay$ and $tb$ symbols
   in $checker$;
2  foreach ($bit \in bit\_of(checker.sds)$)
3     $checker2 = existential\_quantify(checker, bit)$;
4     if ($checker2$ is not constant "1")
5        $checker$ = $checker2$;
6        $succeed\_sds = succeed\_sds \cup bit$;
7  return $succeed\_sds$;
```

Fig. 5. Reset slack calculation algorithm for flexible inputs.

In the algorithm we first quantify all the symbols injected to model delays and $random in the testbench. Next, we iteratively quantify each bit of the checker's sd symbols in lines 2-3, and then check whether the the quantified function, $checker2$, becomes constant "1". If not, then $checker$ is replaced by $checker2$ and the bit is added to the set of successfully quantified sd set ($succeed\_sds$). Finally, $succeed\_sds$ is returned in line 7. In the algorithm, lines 2-6 are based on a greedy approach where we try a potential slack range at a time and discard the range that creates reset problems.

Note that the solution we found is neither optimal nor unique. Depending on the order of quantification, slacks can be moved from one reset cutpoint to another. If one would like to have more slacks for a certain blocks, then the sd symbols for the reset of those blocks should be quantified first.

IV. Experimental Results

To evaluate our methods, we applied our techniques to two processors, DLX and Alpha, from [12]. Both processors are 5-stage pipelined and run the MIPS-lite and a subset of the Alpha instruction set, respectively. The number of key registers is 55 for DLX and 60 for Alpha, and the total number of bits for those registers are 1153 for DLX and 1251 for Alpha. We conducted our experiments on a Linux machine running Ubuntu 8.04 with AMD Phenom II x4 940 CPU and 8G memory. Our implementation is based on a commercial symbolic simulator [13] and the ABC package [14].

A. Results for Fixed Inputs

In our first set of experiments we applied fixed inputs to the processors. We introduced one reset cutpoint for each register to obtain the slack for each register. The maximum reset slack for our analysis was six. We used two sets of inputs in this experiment. The first set contained only NOP (no-operation)

TABLE I
DISTRIBUTION OF RESET SLACKS FOR THE DESIGNS.

Design	Runtime	Memory	RS=0	RS=1	RS=2	RS=3	RS=4	RS=5	RS=6	Total
DLX(NOPs)	1.21s	6.5M	3	2	1	1	1	0	47	55
DLX(ADDs)	27.9s	14.6M	3	3	2	4	2	1	40	55
Alpha(NOPs)	1.67s	6.9M	5	4	1	2	0	0	48	60
Alpha(ADDs)	103.06s	35.7M	8	8	1	6	0	0	37	60

instructions, and the second set contained only ADD (addition) instructions. The results are summarized in Table I. In the table, column "RS=N" shows the number of registers with maximum slack equal to N. The runtime and memory usage are also included in the table.

From the results, we can see that our algorithm can find reset slacks for most registers. In fact, many of the registers have slack equal to six. The major reason is that the applied instructions only exercised a small portion of the design. As a result, many key registers did not have value changes at all and could have large reset slacks. This observation is supported by comparing the results with different input instruction sets: when ADDs were used instead of NOPs, the slacks for several registers reduced. The reason is that when ADDs were used, a larger portion of the design was exercised, thus reducing the slacks for some registers. In practice, since reset sequences are known for most designs and those sequences typically do not exercise the designs much, one can expect that most registers will posses some reset slacks.

B. Results for Flexible Inputs

In the second set of experiments we applied our algorithm to calculate reset slacks in DLX when inputs are flexible. The inputs were from a constrained-random testbench. In this experiment, we applied m NOPs followed by n flexible instructions after the main reset was de-asserted and varied these numbers to evaluate the performance of our algorithm. The total number of symbolic simulation cycles was $m + n$. In the DLX design, there were 19 implicitly grouped internal reset signals to its registers and sub-modules. We regrouped these reset signals into five domains according to their stages in the pipeline and inserted a reset cutpoint for each domain. Each reset cutpoint had two inputs: $delay$ and sd. Symbols were injected into these inputs according to the algorithm in Section III-C.

The results are summarized in Table II. In the table we show the runtime of our algorithm, memory usage, number of variables to quantify for each input sequence (#Var), the maximum number of AIG nodes during quantification (#Node), and the slack we calculated for each reset cutpoint (Sk1-5). From the results, we can see that our algorithm could find reset slacks effectively and efficiently: reset slacks were found for most reset domains and runtime was within 1 minute. The only reset domain with 0 slack was domain 1, which contained the program counter. Since the address saved in the counter begins to advance right after reset is de-asserted, it is expected to have 0 reset slack. The results also show that when a larger number of flexible inputs was allowed, the reset slacks reduced. This is because a larger portion of the design was exercised by the flexible inputs which will change the values of some registers.

TABLE II
RESET SLACK ANALYSIS RESULT FOR FLEXIBLE INPUTS.

Input seq	Runtime	Mem. usage	#Var.	#Node	Sk1	Sk2	Sk3	Sk4	Sk5
m=0, n=3	57.14s	48.4M	119	19752	0	1	2	3	1
m=4, n=0	0.29s	13.6M	28	116	0	1	2	3	4
m=1, n=3	56.49s	43.6M	183	6282	0	1	2	3	2
m=4, n=1	0.37s	18.3M	40	205	0	1	2	3	4
m=2, n=3	24.06s	43.2M	188	6528	0	1	2	3	3

From the results, we also found that our runtime was short but it highly depended on the number of variables to quantify and the maximum number of AIG nodes during quantification. This suggests that Boolean quantification played an important role in our algorithm. Research on finding good quantification order exists [1], [11], and this is our future work.

V. CONCLUSION

Reset is one of the most important signals for many circuits. However, since it is used infrequently, it is often routed at late design stages. Due to the lack of routing resources, long wires may be necessary, which create reset recovery issues that can render the circuit useless. In this work we propose several reset slack calculation techniques that can obtain the reset recovery delay tolerance for design registers. This information gives designers more flexibility to arrange the reset signals and allow physical design tools to perform better optimizations. Our empirical results show that our methods can effectively and efficiently calculate the reset slacks for two processor designs

ACKNOWLEDGMENT

The authors want to thank Juergen Dirks (LSI Corporation, Germany) for motivating this work. This research was supported by National Science Council, Taiwan, under Grant NSC 97-2221-E-002-216-MY3.

REFERENCES

[1] P. A. Abdulla, P. Bjesse and N. Eén, "Symbolic Reachability Analysis Based on SAT-Solvers," *TACAS'00*, pp. 411-425
[2] M. Bazes,"Circuitry and Method for Reset Discrimination," *US Patent*, 5442310, Aug. 15, 1995
[3] V. Bertacco, "Scalable Hardware Verification with Symbolic Simulation," *Springer*, 2005
[4] R. E. Bryant, "Symbolic Simulation – Techniques and Applications," *DAC'90*, pp. 517-521
[5] K.-H. Chang, I. L. Markov, V. Bertacco, "SafeResynth: A New Technique for Physical Synthesis," *Integration: the VLSI Journal*, Jul. 2008, pp. 544-556
[6] H.-Z. Chou, H. Yu, K.-H. Chang, D. Dobbyn, S.-Y. Kuo, "Finding Reset Nondeterminism in RTL Designs – Scalable X-Analysis Methodology and Case Study," *DATE'10*, pp. 1494-1499
[7] H.-Z. Chou, K.-H. Chang and S.-Y. Kuo, "Accurately Handle Don't-Care Conditions in High-Level Designs and Application for Reducing Initialized Registers," *IEEE Trans. on CAD*, Apr. 2010, pp. 646-651
[8] C. E. Cummings and D. Mills, "Synchronous Resets? Asynchronous Resets? I Am So Confused! How Will I Ever Know Which to Use?" *SNUG*, 2002
[9] C. E. Cummings, D. Mills and S. Golson "Asynchronous & Synchronous Reset Design Techniques-Part Deux," *SNUG*, 2003
[10] D. M. Gilday and P. L. Harrod, "Reset Synchronisation," *US Patent Application*, 20100138640 A1, Jun. 3, 2010
[11] F. Pigorsch, C. Scholl, and S. Disch, "Advanced Unbounded Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling," *FMCAD'06*, pp. 89-96
[12] Bug UnderGround, http://bug.eecs.umich.edu
[13] Avery Design Systems Inc., http://www.avery-design.com
[14] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu /∼alanmi/abc/abc.htm