Reproducing Synchronization Bugs with Model Checking

Karen Yorav, Sagi Katz, and Ron Kiper

Galileo Technology, Israel
{kareny,sagi,ron}@galileo.co.il

Abstract. In this paper we describe our experience in reproducing synchronization bugs using a model checker. We demonstrate how model checking technology can be utilized for more than just model checking. Synchronization bugs are caused by physical phenomena which cause the actual behavior of a chip to be different than predicted according to the functional model. Traditionally, verification methods such as dynamic simulation and model checking use a synchronous model, whereas the actual behavior is according to an asynchronous model. Because of this, synchronization bugs are very hard to trace. Using a model checker we were able to create a model closer to the actual behavior, and retrace many synchronization bugs. Because model checking allows us to introduce non-determinism when checking a VLSI design, and because of its ability to produce counter examples for specifications that fail, we find that model checking is the ideal tool for reproducing synchronization bugs.

1 Introduction

Finding bugs as early in the design flow as possible is the goal of all types of verification techniques. The main method for functional verification used by the VLSI industry today is *dynamic simulation*. In **Galileo Technology** we use formal verification methods to complement the use of dynamic simulation. The main formal verification method used is (symbolic) *Model Checking* [2]. Model checkers are automatic tools, capable of proving that a design meets its specification. Model checking has two major advantages: it is exhaustive, i.e. it covers all possible combinations of inputs, and when the design does not comply with its specification the model checking tool presents a counter example to prove this. The main limitation of model checking is that it can handle only relatively small parts of a design.

When a functional bug is found in the early stages of design, it is relatively easy to fix. However, sometimes a bug is revealed only after the first prototype is manufactured and tested in the lab. In this case it is far more difficult to identify the cause of the erroneous behavior. An interesting case is when the bug that is seen in the lab was proven to be impossible by the model checking tool. These bugs are caused by physical phenomena which cause the chip to behave differently than expected according to the functional model used for verification.

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 98–103, 2001.

One of the possible causes for such bugs, although not the only one, is the violation of timing constraints on the boundary between asynchronous modules. A typical case is when there are multiple clocks in the design, as in most chips today. Problems arise when a signal (the output of a flip flop) from one clock domain is passed into a different clock domain. Because of certain physical phenomena, two signals that were coordinated in the source clock domain (changed value together) may be passed to the target clock domain with one cycle delay between them. If the design in the target clock domain depends on the coordination between these signals it can cause a **synchronization bug**.

Synchronization bugs are problematic for dynamic simulation, since the model used for dynamic simulation is a deterministic synchronous model, and does not take into account the possibility of the extra delay. To reproduce synchronization bugs we need a non-deterministic model in which each signal crossing the clock domain boundaries is either delayed by one clock cycle or not. It is also necessary to check all possible timings of events, which is impossible with simulation. Although the extra delay is caused by the violation of timing constraints, even if we introduce exact delays on gates and wires into the dynamic simulation, synchronization bugs will not be found, since the delay is caused by a physical phenomena (meta-stable states) which is not simulated.

The method we use is to replace **synchronizer** modules [1] in the design with non-deterministic modules. Synchronizers are modules which are placed on the signals of asynchronous interfaces. Because the space complexity of model checking is very sensitive to the number of variables (flip flops) in the design and the complexity of the behavior, we cannot simply replace all the synchronizers with a sophisticated model that represents the real life behavior. We use several models, each more sophisticated than the previous. We start with the most simple and small, and move to the others only if necessary.

This work is unique in that it shows that it is possible to use model checking in order to verify properties at a level which includes asynchronous physical behaviors, behaviors which do not appear at the register transfer level.

The paper is organized as follows. In Sect. 2 we give a brief overview of model checking of multi-clock systems and explain how synchronizers are used. In Sect. 3 we describe our method for reproducing synchronization bugs, and in Sect. 4 we conclude with some general remarks about our work.

2 Model Checking of Multi-clock Designs

Most real-life designs are multi-clock systems because different parts of the design need to run at different paces. In such systems signals from one clock are driven into flip-flops running on a different clock. Model checking of such systems is either performed using a single clock, which does not represent all the possible behaviors, or using multiple clocks, which is possible only on very small blocks.

A D-flip-flop passes the value from its input (called "d") to its output (called "q") at every transition of the clock from zero to one. The correct behavior of a flip flop is guaranteed only under certain conditions of *setup* and *hold* times on d,

which means that d must be stable for a certain amount of time before and after the rising edge of the clock. If these conditions are not maintained it is possible that the flip flop will pass the wrong value to its output, or enter a meta-stable state, which means that the output of the flip-flop is undetermined and may stay this way until the next rising edge of the clock. Any other component of the design that examines the output of a flip flop in a meta-stable state may interpret it as either zero or one.

Assume that the output of flip flop a is the input to the flip flop b. If a is running on a different clock than b then whenever a changes value it may violate setup and hold conditions for b, since the change is not synchronized with b's clock, and b might output the wrong value, or enter a meta-stable state. The design methodology for multi-clock systems is that the interface between clock domains must use a full handshake protocol, so that the design on b's side must acknowledge the change in a's value before a is allowed to change again. This solves the problem of b getting the wrong value because it is guaranteed that by the next cycle a has been stable long enough and b will get the correct value. The only effect here is that b changes value one cycle later than it should have. The solution for the problem of meta-stable states is the introduction of a synchronizer between a and b.

A typical synchronizer [1] decreases the probability of an internal flip-flop entering a meta-stable state by placing two D-flip-flops in a row (Fig. 1). The probability of s_1 entering a meta-stable state because of a change in a's value is very small. If this happens, s_2 may stabilize on zero or one, or enter a meta-stable state itself. The probability of this is even smaller.

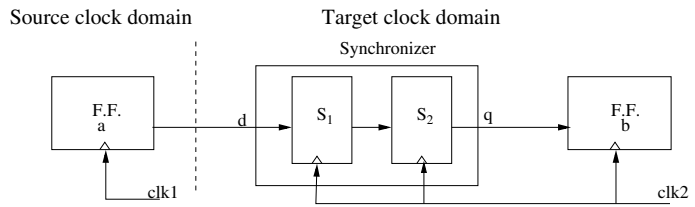

Fig. 1. A typical synchronizer

Each synchronizer creates a delay on its input signal, but it is possible to have different delays on different signals because of slight timing differences. This happens when two different signals that cross clock domains create different effects on their synchronizers - one will pass the change of value in time and the other "misses" a clock cycle (because there was not enough setup time) and passes the change only in the following cycle. Thus, signals that were coordinated in one clock domain are passed to the other clock domain with a delay of one cycle between them. Synchronization bugs occur when the correct behavior of the design inadvertently depends on the coordination of two signals, and this coordination is broken by the additional delay on one of them. This error will

not be revealed during verification with dynamic simulation since in simulation the coordination between signals is preserved.

3 Reproducing Synchronization Bugs

Normally, model checking fails to detect synchronization bugs because it ignores the non-deterministic behavior of synchronizers. We show how careful manipulation of the design can create a more accurate model. We present a simple version of the module with which we substitute the synchronizers in the design. We then describe more complicated, and more accurate modules. We need more than one version because using a complicated synchronizer may make the model checking task impractical. We strive to make the behavior of the synchronizer as simple as possible, not to add more variables (flip flops) than necessary, and not to add non-determinism if not necessary, while making sure that the behaviors we add are actually possible, or else we will find false bugs.

3.1 Introducing Early Signals

The situation is that a bug was found in the lab, on real hardware, and needs to be reproduced. A specification ψ claiming that the erroneous behavior is impossible is checked on the design and is proved to be correct. We suspect that this is a synchronization bug. Our goal is to refine the model of the design so that it is closer to the real behavior of synchronizers in a chip.

As explained in Sect. 2, it is possible that the first flip flop inside the synchronizer will not pass a change in value in the cycle that it happens, but only one cycle later. To model this behavior we override the synchronizer code with a non-deterministic environment model. Instead of a 1-cycle delay, the new synchronizer module may pass a change on the input signal 1-cycle early. The effect on the coordination between different signals is the same, since it allows for any two input signals that change value at the same cycle in the source clock domain to be passed to the target clock domain with a difference of 1 cycle between them (even though one of them is early instead of the other being late).

The new synchronizer module is written in the language of the environment, since this language allows for non-determinism. We use IBM's RuleBase model checker [3], and the modules are written in EDL - RuleBase's *Environment Description Language*. The semantics of this language is simple. The **next** clause describes the value of a variable in cycle $n+1$ based on variable values in cycle n. The expression $\{ex_1, \ldots, ex_k\}$ stands for the non-deterministic choice of one of the values $ex_1, \ldots, ex_k$. Following is the code for the *stable_early* synchronizer. Its input is either always passed on time, or always one cycle early.

```
Module stable_early (/*INPUT*/ d)(/*OUTPUT*/ q)
VAR
    s1, s2 : boolean;
    Status : {Early,no_shift};
ASSIGN
```

```
    next(s1) := d;
    next(s2) := s1;
    next(Status) := Status;
DEFINE
    q := if (Status=Early) then s1 else s2 endif;
```

The **VAR** section declares the internal variables of this module: $s1$, $s2$, and *Status*, which decides whether the input value d will be early or not. *Status* can have one of two possible values - *Early* or *no_shift*. In the *ASSIGN* section the behavior of the variables is defined so that $s1$ has the value of d delayed by one cycle, and $s2$ has the value of d delayed by two cycles. The value of *Status* never changes, and q is defined according to it.

The initial value for all the variables is left undefined, so *Status* can be either *Early* or *no_shift*. This means that each instance of the synchronizer will either be always early, or always on time. This behavior is reasonable because it is likely that a synchronizer that causes a delay will do so for long periods of time. We do not set an initial value for $s1$ and $s2$ because all of our designs have a *reset* signal, so initial values do not matter.

We replace every synchronizer in the design with the *stable_early* module, and check the same specification ψ. If it fails, the counter example produced by the model checker gives an example of how the different timing of signals passing between clock domains can cause an error. This counter example displays a behavior which, although impossible in the regular model used for model checking, is possible in the actual chip, and is therefore a valid counter example.

3.2 Other Versions of Synchronizer Modules

For a large design we do not want to replace all of the synchronizers with nondeterministic versions, since this can make model checking impossible. However, if we replace only some of the synchronizers then the *stable_early* module is no longer sufficient. If signals a and b pass through synchronizers, but only a's synchronizer is replaced, then there is only the possibility that a is early and not that b is early. For this reason we introduce the *stable_early_and_late* synchronizer, which allows for signals to be either early or late.

We add a variable $s3$ defined by: $next(s3) := s2$. We allow the *Status* variable to take an extra value "Late" so that if $Status = Late$ then $q := s3$. The result is that q can be delayed by either 1, 2, or 3 cycles after d. We can now replace some of the original synchronizers in the design with this version, leaving some of them intact. We restrict our model so that either there are no late synchronizers or there are no early synchronizers (we do mix *Early* and *no_shift*, or *Late* and *no_shift*). This prevents the possibility of two cycles difference between signals that were coordinated, which cannot happen in reality.

In both of the previous versions the delay caused by a single synchronizer was fixed, although in reality it is possible for a synchronizer to add a delay only part of the time. For this reason we have a *dynamic_early_and_late* synchronizer module, in which the *Status* variable can change value in the middle of a run.

We add an input boolean variable called *allow_change*, and define *Status* so that it cannot change when $allow\_change = False$. When $allow\_change = True$ the *Status* variable can non-deterministically change value (or not), the only limitation being that it is not allowed to change from *Early* to *Late* or visa versa, because such a change is physically impossible.

Most of the time when using this version we allowed *Status* to change only once in each run, because this behavior is closer to the real behavior of the chip (the delay of a synchronizer changes only very rarely). This also reduces the running times of model checking.

4 Our Experience and Conclusions

Galileo Technology develops VLSI chips for the communication markets. Formal verification using RuleBase is an integral part of the verification process of our products. The technique described here was used on several designs, some of which were very large. The design which started the work described here, for example, is a communication system controller, with around 1 Million gates. It is impossible to verify designs of this size using a model checker, so we verify smaller units separately. This unit includes two clock domains, running at different rates. Most of the time the model checking effort was done using the same clock for both domains, and using the *stable_early* or *stable_early_and_late* versions, but other versions were used as well, including complicated ones not described here. All together we were able to find several significant bugs which would have been extremely difficult to find otherwise. It is interesting to note that the main bug in this example was found using the simple *stable_early_and_late* module. In fact, most bugs are found using the simple stable versions, because most of the bugs are conceptual bugs, where the designer did not consider synchronization issues at all. In this case, it is enough to break the coordination between signals once and the design will exhibit wrong behavior.

The experience described in this paper shows that model checking is a very powerful tool. It can be used in many ways, to solve problems which are difficult to solve using other tools. The VLSI design flow is long and complicated, and we believe that there are other such problems where model checking can be applied. For example, we intend to investigate the possibility of using model checking to solve timing problems and to check performance.

References

1. L. Glasser and D. Dopperpuhl. *The design and analysis of VLSI circuits.* Addison-Wesley, 1985.
2. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.
3. The RuleBase homepage at IBM: `http://www.haifa.il.ibm.com/projects/verification/RB_homepage/`.