CESNET technical report number 4/2004

How to Formalize FPGA Hardware Design [1]

Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, and Pavel Šimeček

October 6, 2004

1 Abstract

In this report, a formal view of an FPGA hardware design is presented. An approach of how elementary FPGA design entities can be modeled in terms of Kripke structures is presented here. The report is also focused on capturing the problems of modeling synchronous parts of hardware design together with its asynchronous parts.

2 Introduction

In formal verification, abstraction plays a crucial role due to the state *explosion problem* [Bar02]. Abstraction is related mostly to simplification of specification of complex system behavior. The most crucial fact concerning abstraction is that of ensuring the abstraction to be correct and still containing the critical parts of the system under verification.

In the following sections, ways of how to abstract away from complex physical properties of an FPGA hardware design [FPGA] are discussed. We focus on the approach of modeling basic elements of the FPGA-based hardware design in the form of a *Kripke structure* [MC]. Moreover, the problems of capturing both synchronous and asynchronous parts of a hardware design together in one model are also solved.

3 Principal Entities of Hardware Design

The logical structure of a typical hardware design specified using the state-of-the-art HDL languages such as VHDL [VHDL] or Verilog [Ver] is modular. It is composed from small units (*entities*) interconnected together by wires along

[1]This work is supported by the FP5 project No. IST-2001-32603, the CESNET activity Programmable hardware, and the GACR grant No. 201/03/0509.

which *events* representing changes of logical values flow. Each entity has an interface which is given by a set of *input signals* and *output signals*. Events occurring in input signals control the behavior of the entity. Values of output signals are combined from values of input signals with respect to the current state of the entity.

Following the physical aspects of hardware, changes of values of signals occur with some infinitesimal delay, so called *delta delay* [VHDL]. We have to point out that we abstract from this delay whenever it does not affect the properties we verify. We call this kind of abstraction the *zero delay abstraction*. This abstraction relies on the same idea as the synchronous hypothesis known from synchronous languages [Esterel]. As this abstraction is very strict and can hide some very critical aspects of hardware design, we will discuss it in details later on in this report.

From the behavioral point of view, we distinguish two basic elements of hardware design – *combinational logic elements* (and-gates, or-gates,...) and *sequential logic elements* (latches and flip-flops). Values of signals in combinational logic elements can be sensed only in the moment in which they occur. In particular, the combinational logic elements instantaneously transform values from input signals to values of output signals. In contrary, the purpose of a sequential logic element is to save a value of a signal over time. We call this kind of behavior *registered behavior* with respect to the fact that the sequential logic is composed from registers and other memory elements.

The crucial observation of the registered behavior is that an assignment of a value to a registered signal is always put in the context of a conditional specification. In other words, assignments to registered signals are guarded. If a particular guarding condition does not hold, the value of the relevant registered signal remains the same. This conservative behavior takes its place here even if the value of the input signal, which has to be stored, has been possibly changed. Moreover, assigning to a registered signal may be controlled also by events occurring in some *clock signal*. Details of this behavior will be discussed later.

Here we would like to give examples of two most basic sequential logic elements – a **latch** and a **flip-flop** circuits. We focus on highlighting differences in behavior of these circuits. In Figure 1 there is an example of a level sensitive **latch**. The VHDL process statement construction in this figure is the key part of definition of the latches behavior. The circuit has two input signals – the data signal `in` and the signal `gate`. There is also one output signal `out`. The circuit is sensitive on changes occurring in both its input signals. The behavior is such that whenever `gate` has high value then the output `out` changes *asynchronously* (instantaneously) with any change of the signal `in`. In all situations in which `gate` is low, `out` is constant and retains its current value. As the most significant property, we highlight the asynchrony of the latch behavior. In other words, the signal `gate` acts as a gate guarding a direct connection between the signal `in` and the signal `out`. A sample timing diagram is depicted in the right part of the figure.

```
process (gate, in)
  begin
  if gate = '1' then
    out <= in;
  end if;
end process;
```

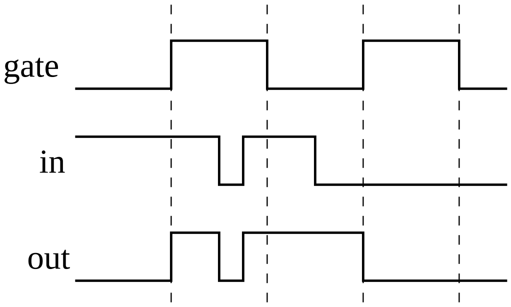


Figure 1: Latch

Another basic sequential logic elements are **flip-flops**. Their behavior is similar to that of latches with one crucial difference. Unlike latches, a flip-flop is sensitive only on the clock signal. Assigning a value to a flip-flop is synchronous with ticks of the clock. An example of an edge sensitive flip-flop is in Figure 2.

The expression of the if-statement requires not only `clk` to be high, but also the rising edge in that signal. The output `out` signal will be updated to the value of `in` just only in the moment when `clk` turns from low to high. Corresponding timing diagram visualizing a typical flip-flop signal flow is depicted in Figure 2.

```
process (clk)
  begin
  if clk'event and clk = '1' then
    out <= in;
  end if;
end process;
```

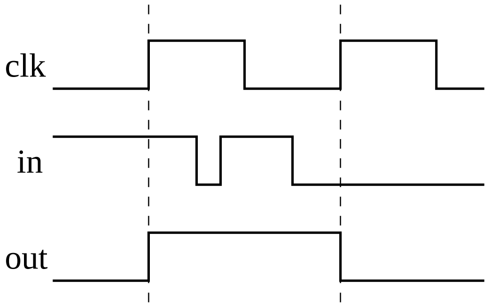


Figure 2: Flip-Flop

With respect to comparison of aspects of the circuits described above we distinguish two kinds of behavior of registered signals – the *synchronous* and the *asynchronous behavior*. In the following two examples we show how the *synchronous* and the *asynchronous behavior* can be combined. For the first example (Figure 3) we use a flip-flop register with an asynchronous reset. In this case, the signal `out` is reset to low value whenever the `reset` signal turns to high. This behavior is absolutely independent of the `clk` signal. With respect to the `in` signal the circuit behaves as a flip-flop.

```
process (clk, reset)
  begin
  if reset = '1' then
    out <= '0';
  elsif clk'event and clk = '1' then
    out <= in;
  end if;
end process;
```

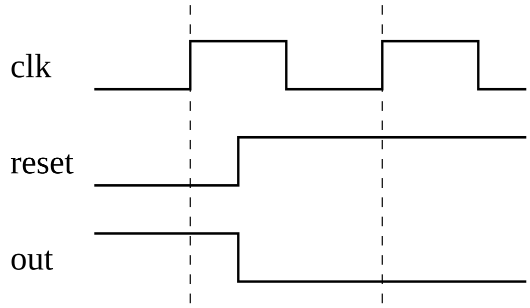


Figure 3: Flip-Flop with an Asynchronous Reset

In Figure 4 we present the second example, a flip-flop register with a synchronous reset. In this case, the `out` signal is reset synchronously with the clock. Note that the reset take its effect just with the raising edge of the `clk` signal.

```
process (clk)
  begin
  if clk'event and clk = '1' then
    if reset = '1' then
      out <= '0';
    else
      out <= in;
    end if;
  end if;
end process;
```

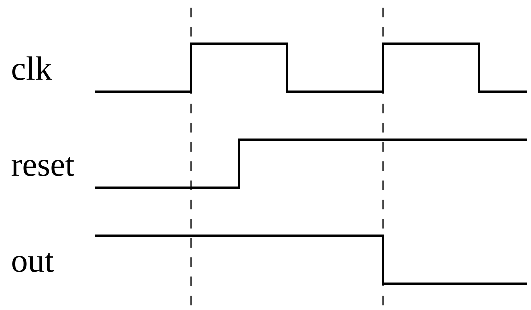
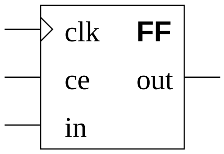


Figure 4: Flip-Flop with a Synchronous Reset

In the previous paragraphs we have mentioned that the zero delay abstraction has to be done carefully in some cases; the following example of Figure 5 demonstrates such a case.

There are three flip-flop registers (`FF0`, `FF1`, and `FF2`) with synchronous gates. All of them have the same structure. In their interfaces there are input signals `in`, `ce`, and the registered output signal `out`. Each signal `ce` performs the synchronous gate and guards synchronous change of the output signal `out` to the actual value of the `in` signal. In the register `FF0`, there are both signals `ce` and `in` constantly set to high value.

Hence, the gate is still opened and (each) rising edge of clock signal sets the output signal `out` to high signal too. The input `ce` of the register `FF1` is connected to the output of `FF0` and so the gate of `FF1` is just opened in the first rising edge of the clock. That is the place to be careful. As there is a short delay in the signal transfer in the real hardware, we have to keep on mind that for the first rising edge of the clock the gate of `FF1` is closed. Similarly, at the second rising




```
process (clk)
  begin
  if clk'event and clk = '1' then
    if ce = '1' then
      out <= in;
    end if;
  end if;
end process;
```

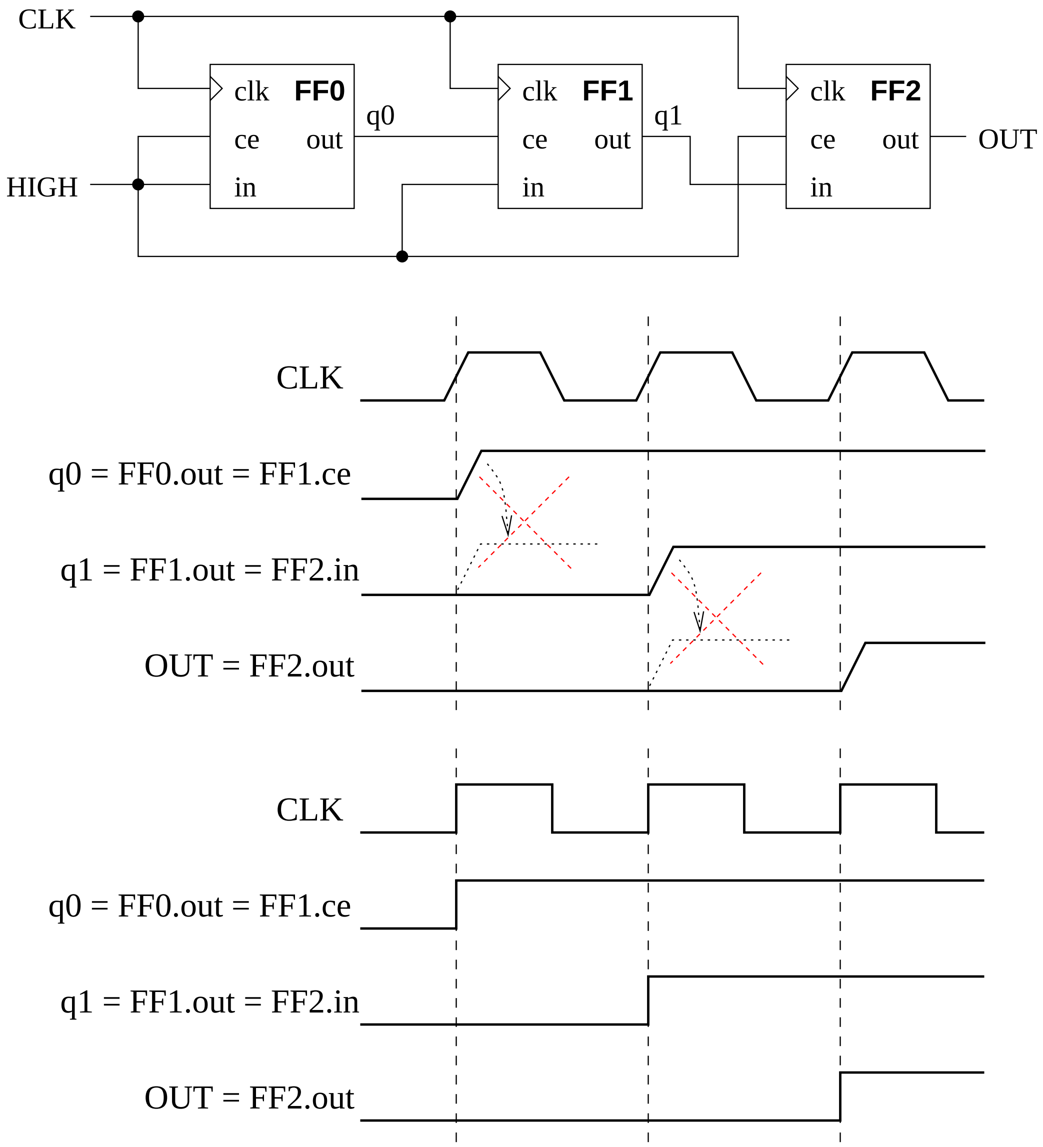


Figure 5: Zero Delay Abstraction

edge of the clock the input signal `in` of the register `FF2` is delayed and so the output `out` is once more set to the low signal at the second rising edge of the clock.

To sum up, whenever the setting to the registered signal is controlled by an edge of a signal, the input values taken in to account during this setting are those immediately before the controlling signal edge.

4 A Formal View of Basic Hardware Design Entities

For the model checking approach we need to specify the model formally as a finite state transition system where states represent current signal values and transitions represent their discrete changes. For this purpose we use *Cadence SMV* language [SMV] which allows us to encode such a model. Moreover, using the SMV tool we can prove the properties we claim about the model. For specification of such properties we use a temporal logic. In this section we describe how this modeling is done.

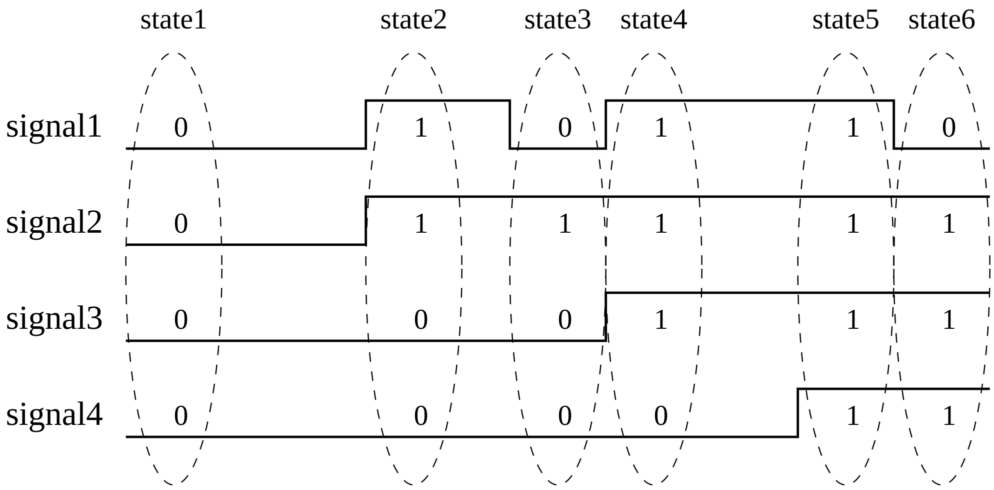


Figure 6: Timing Diagrams and SMV trace

Each state of a transition system can be expressed as a vector of current values of signals in a particular discrete point of time. In the timing diagram depicted in Figure 6 states are represented as columns of 0s and 1s. Assuming the zero delay abstraction, two-value domain is used for modeling of signal values (high – \$1\$ and low – \$0\$). Each transition models instantaneous change of some signals with respect to their current values contained in the source state. The target state then contains the new values of the signals being changed.

Now recall the two kinds of logic elements we have defined in the previous section, the combinational logic and the sequential logic. In the following paragraphs we focus on formal representation of both of them.

The combinational logic is captured by the notion of states. Relations between signal values in a particular state originate just a model of some combinational logic elements. As the delta delay has no influence on behavior of combinational logic, the zero delay abstraction fits here well.

More difficult is to capture the behavior of sequential logic elements. Changes of registered signals are modeled by transitions. A change of the value of a specific registered signal is implied by some values in the preceding state.

```
do {
  if (next(clk) = 1)
    next(out) <= next(in);
  }
```

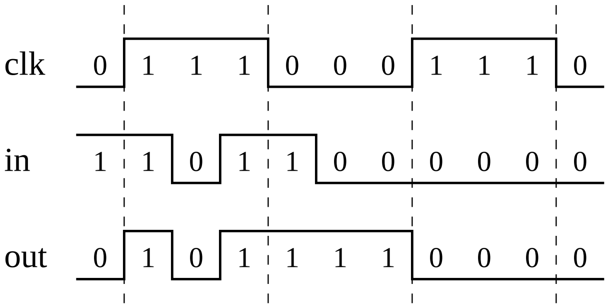


Figure 7: Latch in Cadence SMV

In the case of a latch, any change of the `out` signal is guarded by a simple condition which requires the clock signal to be high. The transition system which models the behavior of the latch is encoded in SMV and showed in the left-hand part of Figure 7. An example of a trace of this transition system is depicted on the right-hand side of the figure. Note that any change of any signal, even if it occurs asynchronously with the clock, defines a transition from the current state to a new state. In every state in which `clk` is 1, the signal `out` has the same value as the signal `in`; otherwise it keeps itself previous value. This models the asynchronous behavior of the latch. Due to this asynchrony, zero delay abstraction does not violate soundness of the model.

```
do {
  if (clk = 0 & next(clk) = 1)
    next(out) <= in;
  }
```

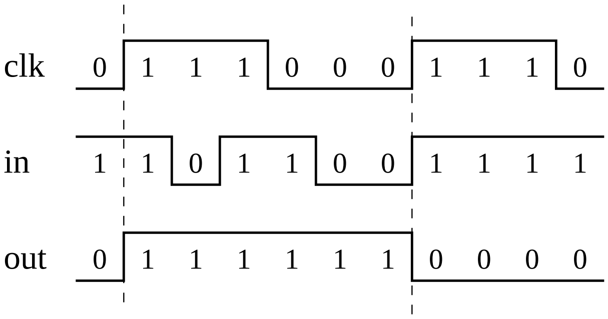
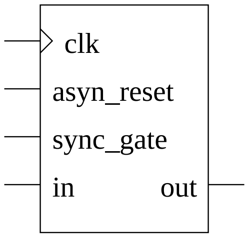


Figure 8: Flip-Flop in Cadence SMV

In contrary to the latch case, the modeling of a flip-flop has to follow results of the "careful construction" at the end of the previous section. Hence, whenever the current value of clock signal is zero and the next value is one, the next value of the output signal is set to the current value of input. In Figure 8 there is a SMV code of a flip-flop and an example of its trace.

```
process (clk, asyn_reset)
  if asyn_reset = '1' then
    out <= '0';
  elsif clk'event and clk = '1' then
    if syn_gate = '1' then
      out <= in;
    end if;
  end if;
end process;
```




```
do {
  if (next(asyn_reset) = 1)
    next(out) := 0;
  else if (clk = 0 & next(clk) = 1)
    if (syn_gate = 1)
      next(out) := in;
  }
```

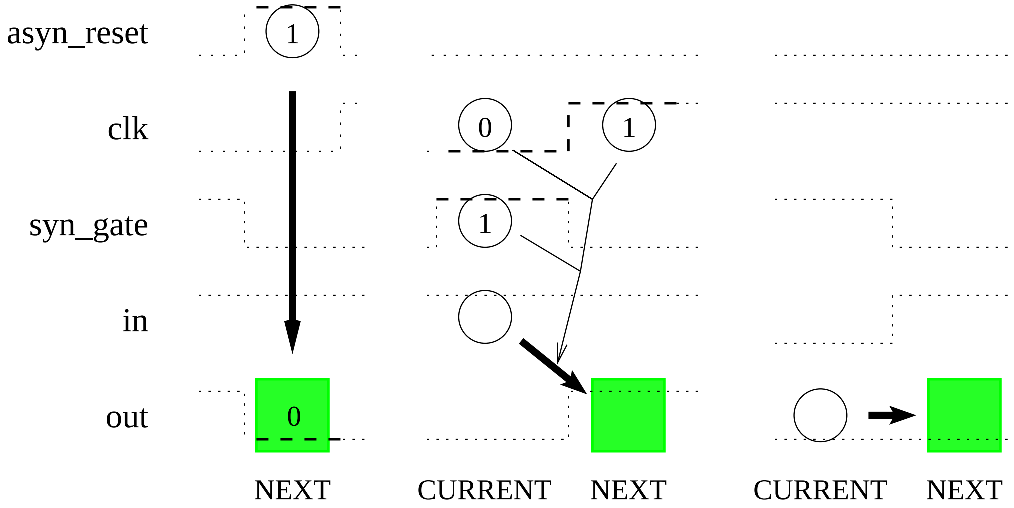


Figure 9: Register with an asynchronous reset and a synchronous gate

The Figure 9 illustrates an example where both synchronous and asynchronous approaches are combined.

5 Two and more clocks

In this section, a situation where the zero-delay abstraction cannot be used is discussed. In VHDL, whenever no delay time is specified for a signal assignment statement, a so called delta-delay is assumed. The delta delay represents an infinitesimal delay needed for signal distribution (as has been mentioned in the Section 2). Besides of the light speed, the main delay contribution is caused by combinational logic between registers. The more logical combinators are sequentially connected, the longer the delay is. It is necessary to ensure the longest delay between any two registers to be shorter than one period of the clock signal which controls the registers. To solve this problem it is sufficient to find the longest sequence of combinational logic (with respect to the caused delay) and compare its delay with the period of the relevant clock signal. In the case of single clock HW design, this problem is easy to solve. Zero-delay abstraction can be used here without lost of generality.

In the case of two or more clocks in the design, the situation is more complicated. The problem is that there could be some registers used by two (or more) clocks, hence the value of such a register could be read at the same time when another part of the design (i.e. controlled by different clock) might be writing into this register. The situation is depicted in the Figure 10 showing two counter `counter1` and `counter2` and their difference signal `diff2`. The `counter1` register is driven by the clock `clk1` and the registers `counter2` and `diff2` are driven by the clock `clk2`. The timing diagram illustrates the situation when the sub-design, which is responsible for counting of the new value of the `diff2` register, is reading from `counter1` just in the moment when its value is not stabilized yet.

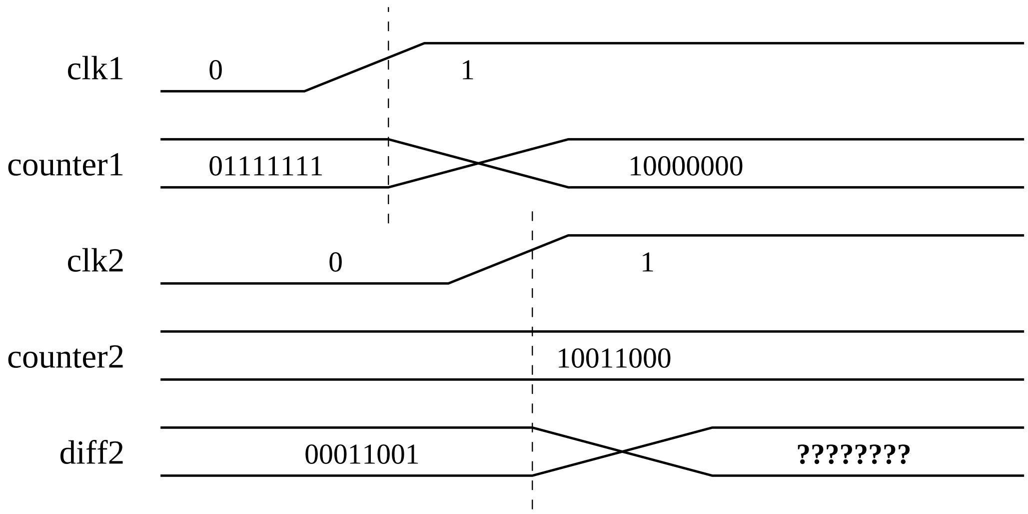


Figure 10: Two counters controlled by two different clocks. The difference signal `diff2` is driven by clock `clk2`.

To model this behavior of the design precisely, we cannot abstract away from the delta-delay. Our approach how to capture this problem is based on the idea of adding one "chaotic" state to the model of the register. In this state, the register returns an undefined value. In the next state, the register is stabilized at the exact value. By that way, an instant of indeterminacy of the signal value is modeled. Thus, the model representing the asynchronous part of the design can then deal with this situation. It can be seen in Figure 11 how such a model looks like for the case of the example given above.

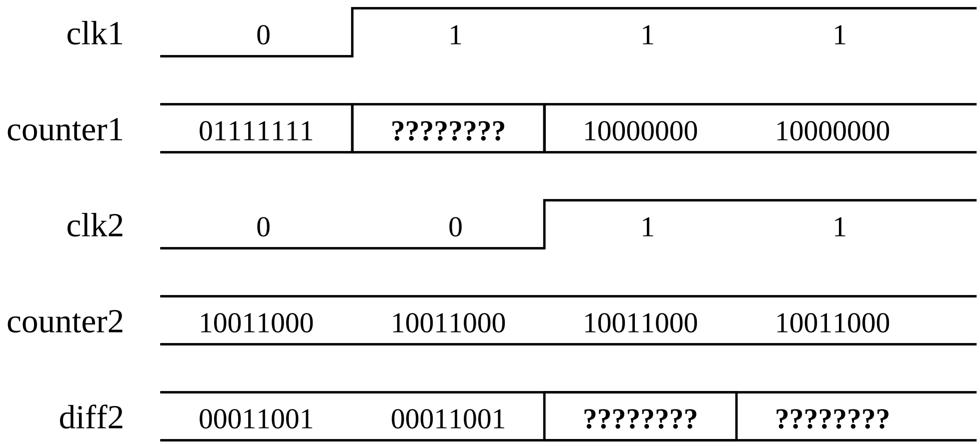


Figure 11: An approach to modeling two counters controlled by two different clocks.

To ensure the correctness of the presented approach to modeling of designs with more than one clocks, we have to evaluate consequences of addition of the "chaotic" state to the synchronous subpart of the design. First of all it can be easily seen that the earliest moment when the rising edge of the clock can arise is the next state. With respect to this fact we can argue that the model of the synchronous subpart of the design is untouched by this change. In Figure 12 there is a timing diagram of the common behavior of the synchronous subpart of the design.

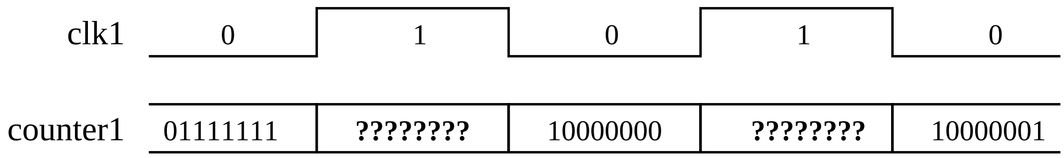


Figure 12: Synchronous part of a HW design.

6 Conclusion

In this report, we have determined the basic elements of hardware design and we have presented an approach how these entities can be modeled in terms of Kripke structures. We have focused on the problematics of modeling synchronous parts of the hardware design together with asynchronous parts. We have successfully applied the techniques described here in formal verification of the Liberouter hardware design, especially in the verification of the asynchronous FIFO unit [LibWWW].

References

[Bar02] Barnat J., Brázdil T., Krčál P., Řehák V., and Šafránek D.: *Model checking in IPv6 Hardware Router Design.*
CESNET technical report 8/2002.

[FPGA] Xilinx, Inc.: *DS031-1 Virtex-II 1.5V Field Programmable Gate Arrays.*
October 2001.

[LibWWW] Liberouter: *Liberouter Project WWW Pages.*
`http://www.liberouter.org/`

[SMV] Cadence SMV: *Cadence SMV WWW Pages.*
`http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`

[Ver] Daniel C. Hyde: *Handbook on Verilog HDL.*
`www.eg.bucknell.edu/~cs320/1995-fall/manual.pdf`

[VHDL] Ashenden Peter J.: *The VHDL Cookbook.*
`http://tech-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf`

[MC] Edmund M. Clarke, Orna Grumberg and Doron A. Peled: *Model Checking.*
Cambridge : MIT Press, 1999.

[Esterel] Gerard Berry: *The Foundations of Esterel.*
Cambridge : MIT Press, 1998.