Simulation of the Delay Insensitive Asynchronous Circuits

Hossein Pedram
Computer Engineering Dept.
AmirKabir University
pedram@ce.aku.ac.ir

Zia Moghadam
Computer Engineering Dept.
AmirKabir University
zia@arayandeh.net

Abstract

Delay Insensitive (DI) circuits are a type of asynchronous circuits where the component delays and wire delays are assumed unbounded. It is possible to describe these circuits in the form of some concurrent processes that exchange information. This description may be transformed to a switch–level description that consists of a number of Production Rules (PRs). Each PR consists of a Guard and an Assignment. Whenever a PR’s guard is activated, the related assignment has to accomplish.

Considering the greater number of switches in the DI asynchronous circuits with respect to equivalent synchronous circuits, the speed of simulation of these circuits is low using the existing synchronous simulators. Moreover, to simulate DI asynchronous circuits with usual synchronous simulators, one needs to transform the code for DI circuits at PR level to a form that is usable by synchronous circuit simulators.

Based on the stability characteristic of DI circuits, the execution of active PRs may occur regardless of their temporal order, hence, speeding up the simulation process. The PR level simulator [1] that we have designed is independent of the order of execution of the active PRs.

A problem arises for a PR with an OR operator as a guard. Whenever one of the inputs to the OR operator is activated, the proper assignment takes place. Due to the fact that the order of execution of active PRs during simulation is irrelevant, the simulator may consider two schedules for the related assignment, where the former schedule may have an execution time after the latter schedule.

This paper illustrates that the design of our simulator is such that it can execute active PRs with OR operators out of their timing order without problems.

Keywords: Asynchronous Design, Delay Insensitive, Production Rule, Asynchronous Simulation Software.

1. Introduction

By definition, asynchronous Delay Insensitive (DI) circuits assume unbounded delay both for the components and for the wires [2] [7]. DI asynchronous circuits may be described in the form of concurrent processes that exchange information using the CHP (Communicating Hardware Processes) language. Utilizing the Martin’s Synthesis Method [10], one can translate the CHP description to a switch-level description that consists of a number of
Production Rules (PRs) [8] [9]. Each PR has the general format of $G \rightarrow S$, where $G$ (the PR guard) is a Boolean expression, and $S$ (the PR assignment) is the allocation of a value (0 or 1) to a signal. As an example $a \lor b \rightarrow c \uparrow$ is a PR. Whenever the Boolean expression $a \lor b$ turns true, the value of signal $c$ is set to 1.

An important problem that prevents the popularity of asynchronous design methods is the lack of software aids needed to automate the design procedure. A member of this set of software is the simulator software. The existing simulators for synchronous circuits do not fit properly to simulate asynchronous circuits, including the DI circuits. One reason is the larger scale of asynchronous circuits with respect to their synchronous circuit's version, especially in large designs [3] [4] [5] [6], which forces much greater simulation times. Another reason is the required transformation of asynchronous code to a form suitable for synchronous circuits simulators. Also, synchronous circuits require some assumptions that impose limitations on the simulator that slows it down compared to the asynchronous case. Ignoring these assumptions, one can develop faster simulators for asynchronous circuits.

The DI circuits have a characteristic named stability, which states whenever $G \rightarrow S$ is a PR, and if guard $G$ is active, $G$ has to stay active until the PR executes (the proper assignment takes place). In DI circuits, all PRs must obey stability. Hence, if the guards of two PRs, named $T_1$ and $T_2$, are activated at the same time, the execution of either of them should not deactivate the other one. So, the operation of the DI circuit should be independent of the order of execution of active PRs.

In the next section, using this characteristic, an algorithm for high-speed simulation of asynchronous circuits is presented.

2. Algorithm for simulating DI circuits

A PR that its guard is activated, and hence, the value of its assignment signal changes, is called effective. As an example, in $a \lor b \rightarrow c \uparrow$, if the value of $c$ is zero and either of $a$ or $b$ turn one, the PR becomes effective.

According to stability property that was explained earlier, one can use the following algorithm to simulate the DI circuits at the PR level:

```
for ever
{
    • Select a PR from the effective set.
    • Remove the PR from the effective set and execute it.
    • Insert the PR’s that are enabled, in the effective set.
}
```

As an example, suppose that the list of PRs describing a DI circuit is as follows:

- $a \rightarrow b \uparrow$ after 2
- $a \rightarrow c \uparrow$ after 1
- $b \land c \rightarrow a \downarrow$ after 4
- $\neg a \rightarrow b \downarrow$ after 3
- $\neg a \rightarrow c \downarrow$ after 5
- $\neg b \land \neg c \rightarrow a \uparrow$ after 6

In above, a time delay is assumed for each PR. If the guard of PR is activated, the time delay designates the amount of time after which the related assignment accomplishes. Suppose the initial values of $a$, $b$, and $c$ signals are zero. Simulation of this circuit with the algorithm is as table 1.
Table 1: Simulation of DI circuit example

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Effective set</th>
<th>Schedule list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, time=0</td>
<td>0, time=0</td>
<td>0, time=0</td>
<td>¬b ∧ ¬c → a↑ after 6</td>
<td>(a=1, time=6)</td>
</tr>
<tr>
<td>1, time=6</td>
<td>0, time=0</td>
<td>0, time=0</td>
<td>a → b↑ after 2</td>
<td>(b=1, time=8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a → c↑ after 1</td>
<td>(c=1, time=7)</td>
</tr>
<tr>
<td>1, time=6</td>
<td>1, time=8</td>
<td>0, time=0</td>
<td>a → c↑ after 1</td>
<td>(c=1, time=7)</td>
</tr>
<tr>
<td>1, time=6</td>
<td>1, time=8</td>
<td>1, time=7</td>
<td>b ∧ c → a↓ after 4</td>
<td>(a=0, time=12)</td>
</tr>
<tr>
<td>0, time=12</td>
<td>1, time=8</td>
<td>1, time=7</td>
<td>¬a → b↓ after 3</td>
<td>(b=0, time=15)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>¬a → c↓ after 5</td>
<td>(c=0, time=17)</td>
</tr>
<tr>
<td>0, time=12</td>
<td>1, time=8</td>
<td>0, time=17</td>
<td>¬a → b↓ after 3</td>
<td>(b=0, time=15)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>¬a → c↓ after 5</td>
<td>(c=0, time=17)</td>
</tr>
<tr>
<td>0, time=12</td>
<td>0, time=15</td>
<td>0, time=17</td>
<td>¬b ∧ ¬c → a↑ after 6</td>
<td>(a=1, time=23)</td>
</tr>
</tbody>
</table>

In the last column of table 1, each schedule illustrates that at the specified time, the designated value must be allocated to the signal. As seen in the example, each signal has a specified time for itself, and the simulator does not have a common time. Accordingly, the simulations of the signals do not advance together and synchronously, rather, it is possible for a signal to have simulation time of 17 while another signal has it at 8.

Due to the fact that schedules inside the list are not required to execute in order, one can manage them in a list without temporal ordering (such as FIFO), and without updating. This statement that is concluded from the stability characteristic in DI circuits, may increase the speed of simulation.

3. Graph structure of the compiled code

To simulate, we first need to compile the PR level code that describes the DI circuit, and develop a graph, which contains the information of that code. Next, the simulator core performs simulation on this graph. We now discuss the structure of the graph.

The graph structure consists of a number of nodes, where each node is either a LabelNode or an OperatorNode. These nodes get activated and deactivated during the simulation process.

Each signal possesses two LabelNodes: a NodeOne and a NodeZero. Whenever the signal value turns one, NodeOne is activated, and if the signal resets to zero, NodeZero gets activated.

There is one or more OperatorNode between every two LabelNodes. The OperatorNodes consist of AND (∧), OR (∨), and NOP (¬) operators. With these OperatorNodes, one can develop the Boolean expression of the PR guards. NOP operator is for a PR that sets a signal to 1 or 0 based on 1 or 0 of another signal. The AND operator activates its output node whenever all its input nodes are active. The OR operator activates its output node whenever one of its input turns active. The NOP operator has just one input, which its activation causes the output to turn active.

As a small example, consider a PCHB buffer with the following set of PRs.

\[
\begin{align*}
\text{Re} \land \text{Ll} \rightarrow \text{Rl}_- \downarrow & \quad \text{Rl}_- \rightarrow \text{Rl} \downarrow \\
\neg\text{Re} \land \neg\text{Ll} \rightarrow \text{Rl}_+ \uparrow & \quad \neg\text{Rl}_- \rightarrow \text{Rl} \uparrow \\
\text{Re} \land \neg\text{Lh} \rightarrow \text{Rh}_- \downarrow & \quad \text{Rh}_- \rightarrow \text{Rh} \downarrow \\
\neg\text{Re} \land \neg\text{Lh} \rightarrow \text{Rh}_+ \uparrow & \quad \neg\text{Rh}_- \rightarrow \text{Rh} \uparrow \\
\neg\text{Rh} \land \neg\text{Rl} \rightarrow \text{Le} \uparrow & \quad \text{Rl} \lor \text{Rh} \rightarrow \text{Le} \downarrow
\end{align*}
\]
The graph that is developed for the PCHB buffer is illustrated in figure 1.

As shown in figure 1, the One and Zero nodes for each signal are separated. As a result, if the value of a signal sets to 1, it only affects those places that a 1 can have effect, and it is ineffective in places where a 0 may be effective.

To illustrate the operation of the simulator using the above graph, consider the following PR:

\[ \text{RI}_1 \rightarrow \text{RI} \]

If the signal \( \text{RI}_1 \) sets to 1, node \( \text{RI}_1-\text{One} \) activates, and a schedule to reset the signal RI to 0 is placed in schedules list. Whenever this schedule executes the value of RI resets to 0 and node \( \text{RI}-\text{Zero} \) in the graph of figure 1 activates.

Although the stability property of DI circuits speeds up the simulation, this simulation technique creates a problem for the OR operator that is discussed below.

### 4. Simulating the OR operator problem

Consider the following PR:

\[ a \lor b \rightarrow c \uparrow \text{after } 5 \]

Suppose that \( a \) is set to 1 at time 20, and \( b \) is set at time 30. Since the Simulator does not execute the active PRs in their temporal order, it is possible to set \( b \) first, and set \( a \) next. According to this ordering, a schedule is created to set \( c \) to 1 at time \( 30+5=35 \). Next, the simulator will set \( a \) and create a schedule to set \( c \) at time 25. So, the simulator will set \( c \) at time 35, later, it notices that \( c \) had to be set at time 25 (figure 2).
As illustrated in figure 2, the simulator has to advance the timing to set the signal $c$. This operation is called Retiming. Now the problem is whether Retiming affects the correctness of the simulator.

According to the definition of DI circuits, changing the time for the $c$ signal to turn 1 (from 35 to 25) should not affect the operation of the simulation, if during the interval (25 to 35) $c$ does not turn 0. The guard of $a \lor b \rightarrow c \uparrow$ has to turn inactive, for $c$ to turn 0. To do this, during the interval that $a$ is 1 and $b$ is going to turn 1, $a$ should turn 0. Figure 3 illustrates this situation. During the interval that $a$ is 0 and $b$ tends to turn 1, $c$ may turn 0.

5. The analysis of the OR operator simulation under different situations

Now we consider the circuits in which $a$ and $b$ have waveforms such as those of figure 3. Two cases are analyzed; in case 1, $c$ turns 0 and sets $b$ to 1. In case 2, setting $b$ to 1 is considered independent of $c$ turning 0.

Case 1: $b$ turning 1 depends on $c$ turning 0.

In this case we investigate a situation where setting $a$ to 1 sets $c$ to 1, and later, resetting $a$ to 0 resets $c$. Next, resetting $c$ to 0 sets $b$ to 1 (figure 4).

In this case, the simulator performs correctly, because it cannot set $b$ first, and then set $a$ (the case that requires Retiming). In other words, setting $b$ to 1 requires the simulator have reset $c$ earlier, and also have $a$ reset to 0 earlier than $c$. Hence, the simulator can set $b$ to 1 only when it has converted the value of $a$ from 1 to 0.

Case 2: setting $b$ to 1 is independent of resetting $c$ to 0.

In this case, the two following situations are investigated for signal $c$.

a) Signal $c$ is an output of a combinational gate.

Because $c$ is an output of a combinational gate, thus

$$
\neg a \land \neg b \rightarrow c \downarrow
$$

Resetting $a$ to 0 activates the guard of $\neg a \land \neg b \rightarrow c \downarrow$ and tries to reset signal $c$ to 0. On the other hand, setting $b$ to one activates the guard of $a \lor b \rightarrow c \uparrow$ and tries to set $c$ to one. As a result, depending on the delay between resetting $c$ following resetting $a$, and the delay between setting $c$ following the setting of $b$, we will get different waveforms for $c$.

Figure 5 illustrates the situation where resetting $a$ to zero resets $c$, and later $c$ sets to one as a result of setting $b$ to one.
However, if the delay related to resetting \( c \) to 0 is so large that \( c \) tends to set to 1 first, the following waveform (figure 6) will result (the guard of \( \neg a \wedge \neg b \rightarrow c \downarrow \) deactivates before execution due to the setting of \( b \) to 1). Consequently, this circuit is not a DI circuit, and the operation of the simulator will not be analyzed.

**Figure 6**

**b) Signal \( c \) is output of a state holding gate:**
Suppose that the PR that resets \( c \) to 0 has the format of \( d \rightarrow c \downarrow \) (in general, \( d \) is a Boolean expression). For \( c \) to reset to 0 in the interval between resetting \( a \) to 0 and setting \( b \) to 1, \( d \) must set to one in this interval (figure 7).

**Figure 7**

However, as seen in figure 7, this circuit is not a DI circuit, because \( e \) tends to reset to 0 following the setting of \( d \) to 1, and also \( c \) tends to set to 1 following the setting of \( b \) to 1. As a result, \( c \)’s waveform depends on delays. Due to the fact that this is not a DI circuit under these circumstances, it is not necessary to investigate the operation of the simulator in this case.

The mentioned circuit \((d \rightarrow c \downarrow)\) is a DI circuit if in the interval between resetting \( a \) to zero and setting \( b \) to 1, \( d \) does not turn 1. In this case \( c \) doesn’t turn 0 and stays at 1 (figure 8).

**Figure 8**

If the simulator sets \( b \) to 1 and then sets \( a \) to 1, it is working properly, because during the interval that \( e \) is Retimed, \( c \) does not turn 0.

According to the above discussion, one concludes that the simulator has no problem regarding the OR operator. The only thing that it may require is the Retiming operation. The implementation of Retiming is discussed in the next section.

**6. Retiming**

As observed earlier, the simulator may require Retiming to simulate the OR operation. As an example in the waveform in figure 9, (for \( a \lor b \rightarrow c \uparrow \)) first a schedule is made to turn \( c \) to 1 at time 35. Later, a second schedule is made to turn \( c \) to 1 at time 25 (figure 9).

**Figure 9**

When the second schedule to turn \( c \) to 1 is made at time 25, Retiming causes the
schedule at time 35 to be discarded when it is about to be executed.

The important point to note is that the complexity of Retiming is minimum. It only suffices to discard the first schedule in Retiming. The Retiming operation does not propagate to schedules that are caused by the transition on the discarded schedule, because the schedule is discarded before execution.

7. Simulation Results

A comparison is made between the simulation method based on out of order execution of PRs and the simulation method based on temporally ordered execution is presented. The comparison illustrated the speed up gained by the former method.

For the out of order case, the list of active schedules is implemented as a FIFO. New schedules are added to the end of the list, and schedules to be executed are extracted from top of the list.

For the temporally ordered case, the list of active schedules is implemented as a Queue. Each new schedule is added to the list in a sorted way, and schedules to be executed are obtained from the top of the list. In this case, the active schedules reside in the list based on their temporal order.

To compare the simulation results, a 32 bit asynchronous adder that is designed using the PCHB technique is selected. The adder consists of 32 one-bit full-adders with ripple carry.

The simulation cycle includes 100 million events. The simulation times for the two cases is shown in the table below.

<table>
<thead>
<tr>
<th></th>
<th>Simulation times for an asynchronous 32-bit adder (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>515</td>
</tr>
<tr>
<td>QUEUE</td>
<td>691</td>
</tr>
</tbody>
</table>

According to the mentioned timings, the simulation time based on out of order execution (FIFO) is 34% better than the simulation time for the temporally ordered case (Queue).

8. Conclusion

Using the stability property of the DI circuits, we designed the PR level simulator that executes PRs with activated guard out of the temporal order. We showed that this simulation method operates correctly for DI circuits that include OR operator, and it only requires a simple Retiming. The results of this simulation are equivalent to the simulation results that force the execution of active PRs according to their timing order.

Our simulator which is based on stability property of the DI circuits is much faster than simulators that execute active PRs according to their temporal order, and it preserves the correctness of simulation in DI circuits with OR operator.

9. References


