Modeling and Synthesis of Multi-rail Multi-protocol QDI Circuits

V. Brégier, B. Folco, L. Fesquet, M. Renaudin
TIMA laboratory, 46, avenue Felix Viallet
30031 GRENOBLE cedex-France
Vivian.Bregier@imag.fr

Abstract

Quasi-Delay-Insensitive (QDI) circuits behave correctly independently of arbitrary delays in gates or wires, which gives them many advantages. However, there are a lot of restrictions for the design of this type of circuits, making them difficult to synthesize automatically. This paper presents a new methodology to model and synthesize QDI asynchronous circuits. The targeted circuits use the multi-rail data encoding, and use different handshaking protocols for channel-based communications. The proposed synthesis algorithms are based on Multi-valued Decision Diagrams. This article proves formally that the circuits generated with this technique are QDI. An example is given to illustrate the circuit modeling and the synthesis.

1 Introduction

Asynchronous circuits are circuits that do not have a global signal to synchronize them. The synchronisation between the blocks is done locally. Those circuits show very interesting properties: speed, low power consumption, noise emission, security, robustness, reusability, etc.

Today, the industry needs powerful tools similar to the synchronous ones to start with the asynchronous technology.

This work is part of the TAST [1, 2] (Tima Asynchronous Synthesis Tool) project, which aims to develop such tools. The synthetized circuits in TAST are quasi-delay insensitive, or QDI, which are the most robust asynchronous circuits [3]. Today, such circuits use significantly more transistors than their synchronous equivalents. Many efforts are directed towards the circuit optimisation and transistor reduction. The main difficulty is to preserve the property of quasi-delay insensivity.

In this paper, we present a technique to model and synthetize a QDI circuit. The model expresses both the direct and the return (acknowledgment) path, and supports different communication protocols. We prove that the circuit generated from this model is QDI. Then, it is possible to transform the circuit through its model, and thus optimize it, while preserving its QDI property.

2 Quasi-delay insensitivity

2.1 Communication protocols

The behavioral description of each asynchronous block is based on the protocol used to communicate with its neighbors. To implement a bidirectional indication, two kinds of protocols are usually used: the two-phase protocol (called « half-handshake ») and the four-phase protocol (called « full-handshake ») [4]. In all cases, it is important to note that every transaction of the emitter is signaled by a request signal and acknowledged by a signal of the receiver. In this paper we consider four-phase protocols (see Figure 1). A request transition is followed by an acknowledgement transition, and an acknowledgement transition is followed by a request transition.

Figure 1 shows the following different phases of a four-phase protocol:

Phase 1: The receiver receives a request, makes the computation and sends the acknowledgement signal.

Phase 2: The emitter lowers the request signal.

Phase 3: The receiver lowers the acknowledgment signal.

Phase 4: The emitter can send a new request if data are available.

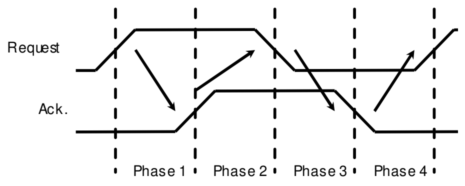

Figure 1: The four-phase protocol

In a computation, the communications are composed of several phases. However, the sequential scheduling between the input and the output of these phases is not the more efficient method. We can schedule them differently, with different protocols, to reduce the protocol latency, and increase the circuit performances. The generated circuit will depend on the protocol.

The first implemented protocol used to synthesize circuits is a four-phase sequential protocol. This protocol is the simplest. The second is a four-phase WCHB protocol ("Weak Condition Half-Buffer", [5]). It schedules the different phases of the communication and thus increases the performances of the circuit.

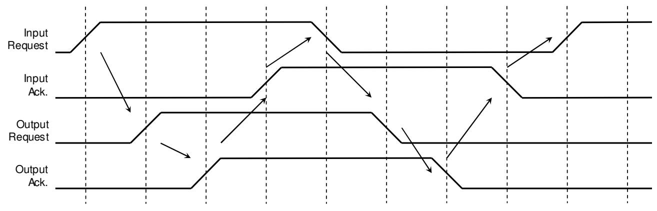

Figure 2: Scheduling of the sequential protocol

- **Sequential protocol:**

This protocol schedules the phases sequentially, as shown Figure 2. When the circuit receives a request, it computes the output, sends the output request, waits for the output acknowledgment, and acknowledges its inupts.

- **WCHB Protocol** (Weak Condition Half-Buffer) [5]

Figure 3 shows the operation performed by the "WCHB protocol :

- When the input raises the request and the circuit is ready for a computation, the circuit produces the output result. It raises the output request and the input acknowledgment.
- Then the input may then change its value, so the circuit has a half buffer to keep the output value while it has not been acknowledged.
- When the input has lowered its request and the output has raised its acknowledgment, the circuit lowers the output request and the input acknowledgment.
- When the output has lowered its acknowledgment, the circuit can make a new computation.

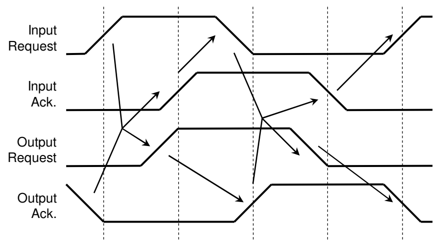

Figure 3: Scheduling of the WCHB protocol

2.2 Data encoding

We have seen that the communication protocol uses a request-acknowledgment cycle. But if the circuit is delay insensitive, we have to guarantee the data correctness on the input when the request arrives. One way to do this is to encode the request signal with the data, in a delay-insensitive encoding.

In this approach we have the following property: the data is detected unambiguously, without any temporal assumption. It implies robustness, portability and ease of use.

There are several delay-insensitive codes. The 1-of-N encoding (also called "multi-rail"), where N is the number of values we want to encode, is presented here. In this code, a data is *invalid* if all the wires have the value 0. If rail number i has the value 1, then the data has the value i. Codes where more than one wire has the value 1 are considered as errors.

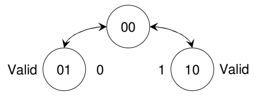

Figure 4: 1-of-2 code (dual-rail). Having both rails to 1 is forbidden. To reach the value 0 (encoded 01) from the value 1 (10), the data has to become invalid (00).

It is the most common DI code. The 1-of-2 code (also called "dual-rail"), illustrated Figure 4, is one of the most popular, because it is similar to the binary encoding used in synchronous circuits. However, some works reported studies on the design of 1-of-N asynchronous circuits, with $N > 2$ [6].

2.3 Quasi Delay Insensitivity

A definition of the Quasi-delay insensitive (QDI) circuits is given by Alain Martin in [3]: QDI circuits are those whose correct operation does not depend on the delays of operators or wires, except for certain wires that form isochronic forks.

A fork in a circuit corresponds to a gate output being used as the input of more than one gate, as illustrates Figure 5.

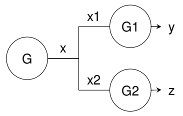

Figure 5: A fork

If a fork is isochronic, some transitions on x do not need to be acknowledged by a transition of both y and z – the output of gates $G1$ and $G2$, respectively. For instance, transition $x\uparrow$ causes transition $x1\uparrow$ and $x2\uparrow$. Transition $x1\uparrow$ causes (and is acknowledged by) transition $y\uparrow$. But transition $x2\uparrow$ does not cause a transition on z. Hence, the completion of transition $x2\uparrow$ has to be justified by timing assumptions. We assume that when transition $x1\uparrow$ has been acknowledged by transition $y\uparrow$, transition $x2\uparrow$ is also completed. This is the "isochronicity assumption" [3].

3 Circuit description using MDDs

3.1 Multi-valued Decision Diagrams

Multi-valued Decision Diagrams are a generalization of BDDs (Binary Decision Diagrams, [7]), presented in [8] :

Definition 1 (Multi-valued Decision Diagram) :
A Multi-valued Decision Diagram taking values in Y is a rooted directed acyclic graph:
Each non-terminal vertex v is labeled by a multi-valued variable $\mathrm{var}(v)$, which can take values in the range $range(v)$. Vertex v has arcs directed towards $|range(v)|$ children vertices, denoted $child_k(v)$ for $k \in range(v)$.
Each terminal vertex u is labeled by a value $value(u) \in Y$.

We also define a path of a MDD, since this notion will play a very important role

Definition 2 (Path) :
Let M be a MDD.
Let $v_0, v_1, \ldots, v_k$ be non-terminal vertices of M and u a terminal vertex of M, let $n_0, n_1, \ldots, n_k$ be integers such

that for all $0 \le i \le k$, $n_i \in range(v_i)$. $(v_0 : n_0, v_1 : n_1, \ldots, v_k : n_k, u)$ *is a path of* M *if and only if for all* $0 \le i \le k$, $child_{n_i}(v_i) = v_{i+1}$ *, and* $child_{n_k}(v_k) = u$.

3.2 Semantic of a MDD

We want to describe the circuit that is to be synthesized with a MDD. For this, we need to attach a specific meaning to the MDD that is modeling the circuit. Figure 6 illustrates an example of MDD, describing a small ALU : the circuit described has two dual-rail inputs, *A* and *B*, and one 3-rails input, *I*, which selects the operation computed : *A and B*, *A or B*, or *A*.

What makes the decision diagram structure interesting for our needs is that it exhibits the mutual exclusivity between the values of an input: a given input can only have one value during a computation. Each path of the MDD corresponds to a possible state of the circuit inputs. At a given instant, only one path can describe the values of these inputs.

We define the notions of active, inactive and transient path:

Let $C = (v_0 : n_0, v_1 : n_1, \ldots, v_k : n_k, u)$ be a path of a given MDD.

Definition 3 (Active path) :
C is active if and only if for all $0 \le i \le k$, $\mathrm{var}(v_i)$ *is valid and has the value* n_i .

Definition 4 (Inactive path) :
C is inactive if and only if for all $0 \le i \le k$, $\mathrm{var}(v_i)$ *is invalid.*

Definition 5 (Transient path) :
C is transient if and only if for all $0 \le i \le k$ *, if* $\mathrm{var}(v_i)$ *is valid, then its value is* n_i .

The output of the circuit implements the 4-phase protocol. The circuit has two computing phases: up-going phase and down-going phase.

• **Up-going phase:** The output of the circuit stays invalid as long as no path of the MDD is active. It can only become valid if a path is active. When the output becomes valid, the circuit gets in the down going phase.

• **Down-going phase:** Let *C* be the active path at the beginning of the down-going phase. The output of the circuit stays valid as long as *C* is transient. It can only become invalid if C becomes invalid. When the output becomes invalid, the circuit gets in the up-going phase.

Note that the different path of a MDD are mutually exclusive: during a computation, two distinct path cannot both be active.

3.3 Acknowledgment

The circuits which we want to generate must implement the 4-phase protocol. So we have to produce the output, but also the signals that acknowledge the inputs. To produce such signals, we extract from a MDD the needed information to acknowledge each input, in the form of secondary MDDs, or *acknowledgement MDDs*.

An acknowledgement MDD specifies a signal. We consider a signal as a particular case of multi-rail, which is valid when the signal is 1 and invalid when the signal is 0. It has only one possible valid value, which implies only one terminal vertex in the MDD.

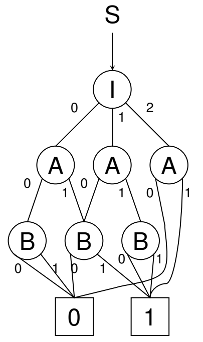

Figure 6: An example of MDD, describing a small ALU. The circuit has 3 inputs, A, B and I.

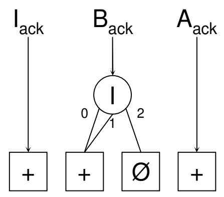

Figure 7: An example of acknowledgement MDDs. + denotes the (unique) valid value, and Ø the invalid value.

As shows Figure 7, in some cases an input of the circuit must not be read to produce the output. Then, it must not be acknowledged: its acknowledgement signal has to stay invalid. Therefore, we add a special terminal vertex, noted ∅, which allows some path to maintain the signal invalid.

To generate the acknowledgement MDD of an input variable, we have to isolate the reading information of this variable, as shows the algorithm Figure 8.

```
exract(mdd, variable)=
   reurn copy_until(root(mdd), variable);

copy_until(vertex, variable) =
   if is_terminal(vertex) then
      return build_terminal(Invalid);
   else if var(vertex) == variable then
      return build_terminal(Valid);
   else
      for i in range(vertex) do
         c = child(i, node);
         childs[i] = copy_until(c, variable);
      v = var(vertex);
      r = ramge(vertex);
      return build_vertex(v, r, childs);
```

Figure 8: Algorithm of extraction of the acknowledgement MDD of an input variable from a MDD

The inputs must not be acknowledged before the output is produced; otherwise the inputs might be invalidated by the environment before the circuit produces the output. In fact, we must be able to encode the different implementations of the 4-phase protocol (sequential, WCHB, etc.) in the MDD and the acknowledgement MDDs. To do this, we modify them, to encode the protocol information

- **Sequential implementation**

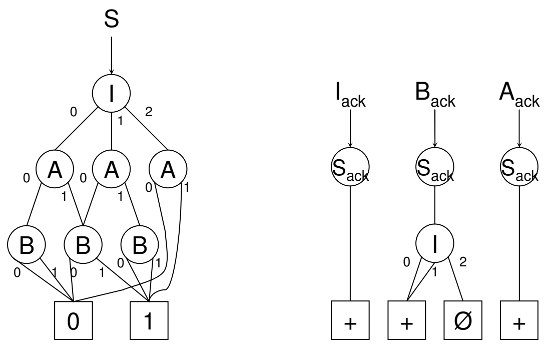

Figure 9: An example of MDD and acknowledgement MDD using sequential protocol

The four phases of the protocol are sequential: we have to wait for the acknowledgement of the output to acknowledge the inputs. It is sufficient to add a S_{ack} vertex at the root of each acknowledgement MDD, as shown Figure 9. Then, each input will only be acknowledged when the acknowledgement signal of the output has been received.

- **WCHB implementation**

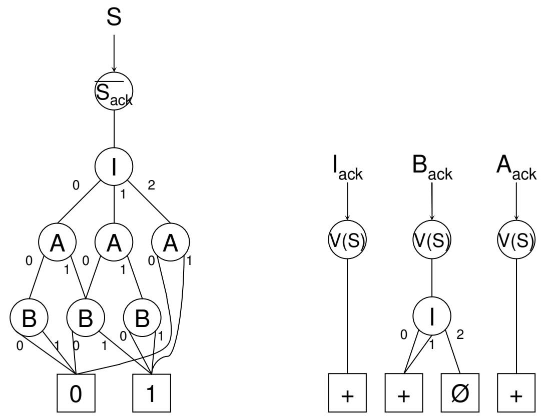

Figure 10: An example of MDD and acknowledgement MDD using WCHB protocol V(S) is an input signal of the circuit, computing the validity of the output data.

Here, the protocol synchronizes the up-going and down-going phases of the input and output channels, as explained in section 2.1.To encode this, we insert a $V(S)$ vertex (corresponding to the validity of the output)[1] at the root of each acknowledgement MDD, and a $\overline{S}_{ack}$ vertex at the root of the MDD.

The $\overline{S}_{ack}$ vertex on the MDD will make the circuit produce the output only when the acknowledgment signal of the output is low, and keep the output active until it has been acknowledged by the output, and thus represents the half-buffer needed. The $V(S)$ on the acknowledgment MDDs makes the circuit acknowledge its inputs as soon as the output is produced. Figure 10 illustrates how the WCHB protocol is modeled in the representation.

4 Synthesis

4.1 Generation of the MDD

We show here how to generate a MDD from a boolean expression. The boolean expression has to express several things:

- the information that an input must be read or not,
- the value that the output must take, depending on the inputs read,
- all this, keeping in mind that the inputs can take more than two values (multi-rail).

For the first statement, we decide that an input that does not appear in a branch of the expression will not be read in that branch. This means that for a dual rail input b, we will not be allowed to simplify $b \vee \overline{b}$, since this would mean not reading b anymore.

The expressions are constructed with a generalization of the boolean operations, as presented in [9]:

- A product-type operation, $\wedge$, that generalizes the boolean `and`
- A sum-type operation, $\vee$, that generalizes the boolean `or`
- Literal unary operations that generalize the boolean `not`, defined by $x^S = \begin{cases} m-1 & if\ x \in S \\ 0 & otherwise \end{cases}$, where m m is the number of rails of x and S is a subset of the values that x can take: $S \subset [0, m-1]$.

Theorem 1

Every multiple-valued function can be decomposed with respect to a variable x_i, as

$$f(x_1, \dots x_n) = \sum_{j=0}^{m-1} x_i^{\ j} \wedge f_{|x_i = j}$$

where $f_{|x_i=j} = f(x_1, \dots x_{i-1}, j, x_{i+1}, \dots x_n)$ *are co-factors of f with respect to* x_i.

We use Theorem 1 to build the MDD, as shows Figure 11.

As an example, we take the following expression, where i is a 1-of-3 input, a and b are 1-of-2 inputs:

$$(i^0 \wedge (a \wedge b)) \vee (i^1 \wedge (a \vee b)) \vee (i^2 \wedge a)$$

This expression represents a multiplexer.

For this expression, if we want to implement a sequential communication protocol, we obtain the MDDs of Figure 9.

If we want to implement a WCHB communication protocol, we obtain the MDDs of Figure 10.

[1]We consider that the validity of the output is calculated by the environment, therefore it is an input of the circuit.

```
build_mdd(expr) =
   if is_constant(expr) then
      return build_terminal(expr);
   else
      a = get_some_variable(expr);
      for i in range(a) do
         cofact = cofactor(expr, a, i);
         c[i] = build_mdd(cofact);
      return build_vertex(a, range(a), c);
```

Figure 11: Algorithm of construction of a MDD from an expression

4.2 Circuit Synthesis

We present here an algorithm to synthesize a circuit modeled by a MDD. This algorithm synthesizes circuits similar to DIMS [10], but with multi-rail inputs and outputs, and with a protocol. We use the same algorithm to synthesize the direct path of the circuit (to generate the output), and the return path (to generate the acknowledgement of the inputs).

As shown Figure 12, for each path of the MDD, we generate a Muller C-element, with each vertex of the path as inputs. Then, we regroup all paths that produce the same value of the output with an "Or" gate. The output of this gate is a wire of the output of the circuit. Thus, the circuits obtained are two level logic circuits. The first level is composed of Muller gates, the second level of OR gates.

```
array_of_lists tmp;

synthesize(mdd m) =
   iter_path(root(m), empty_list());
   for i in range(m)
      out[i] = create_or(tmp[i]);

iter_path(vertex v, wires_list w) =
   if is_teminal(v) then
      m = create_muller(w);
      add_to_list(m, tmp[value(v)]);
   else
      for i in range(v) do
         curr_w = get_wire(var(v), i);
         new_w = add_to_list(curr_w, w);
         iter_path(child(i, v), new_w);
```

Figure 12: Algorithm of synthesis of the circuit described by a MDD.

Further decomposition and optimization of the two-level circuits is beyond the scope of this paper. It is currently investigated in the group.

Figure 13 shows the circuit synthesized using the MDDs of Figure 9, which implements a sequential protocol; Figure 14 shows the circuit synthesized using the MDDs of Figure 10, which implements a WCHB protocol. The WCHB protocol requires a little extra logic, but the circuit will run faster.

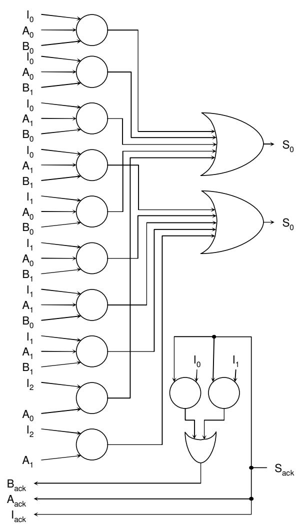

Figure 13: QDI Circuit using sequential protocol

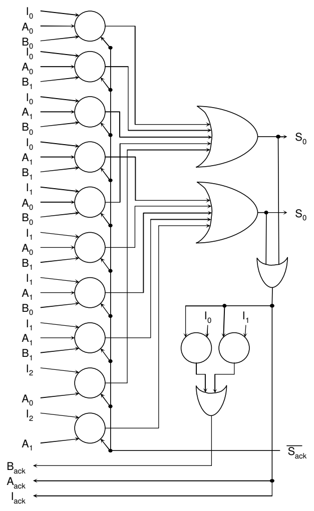

Figure 14: QDI Circuit using WCHB protocol

5 Demonstration

In the previous parts, we have shown a method to synthesize circuits from a MDD representation. We claimed that the circuits synthesized are QDI, but it is not obvious. In this part, we prove formally that the circuit generated from a MDD is really QDI.

5.1 Quasi-delay insensibility

[11] shows a necessary and sufficient condition for a set of production rules to be quasi-delay insensitive. We quote a few definitions for this theorem:

Definition 6 (Production rule) :
A production rule is a construct of the form $G \mapsto t$, where t is a transition and G is a boolean expression known as the guard of the production rule.

A gate with output x, pull-up network B^+ and pull-down network B^-, corresponds to the two following production rules: $B^+ \mapsto x\uparrow \quad B^- \mapsto x\downarrow$

Definition 7 (Result) :
We define the predicate R, the result of a transition: $R(x\uparrow) = x$ and $R(x\downarrow) = \neg x$.

Definition 8 (Non-interference) :
The production rules $B^+ \mapsto x\uparrow$ and $B^- \mapsto x\downarrow$ are said to be non-interfering in a computation if and only if $\neg B^+ \vee \neg B^-$ is always true in the computation.
A set of production rules is non-interfering if and only if every production rule in the set is non-interfering.

Definition 9 (Stability) :
A production rule $G \mapsto t$ is said to be stable in a computation if and only if G can change from true to false only in those states of the computation in which $R(t)$

holds. A production rule set is said to be stable if and only if every production rule in the set is stable.

Theorem 2 (Quasi-delay insensitivity) :
A circuit is quasi-delay insensitive if and only if the production rule set describing it is stable and non-interfering.

Theorem 2 is based on the notion of production rules. Therefore, we have to generate a set of production rules describing the circuit to apply this theorem, and prove that the circuit is QDI.

5.2 Generation of production rules

We need to generate the production rules describing the circuit that we want to synthesize from a given MDD. We generate production rules from the MDD specifying the outputs and the acknowledgement.

Moreover, the circuit model used in [11] is closed (each gate output is the input of another gate), so we also have to define the production rules of the environment of the circuit. We know that the environment implements the four-phase protocol; it is then easy to generate those production rules.

5.2.1 Production rules for the MDD

In this part, the production rules corresponding to the synthesized circuit are generated. Let's consider, for each terminal vertex, all paths that lead to it:

Let f be a terminal vertex of the MDD, let $C_i = \left(n_0^i : s_0^i, n_1^i : s_1^i, \ldots, n_{k^i}^i : s_{k^i}^i, f\right)$ be all the paths of the MDD leading to f.

Let S_j^i be the s_j^i th child of input $\operatorname{var}\left(n_j^i\right)$. Let Y_f be the f th child of the output digit.

We generate several production rules for vertex f :

- Some production rules specify the up going phase:

The wire corresponding to f must go to 1 if C_i, one of the paths, becomes active:

$$C_0 \vee C_1 \vee \ldots \vee C_p \mapsto Y_f \uparrow$$

$$S_0^i \wedge S_1^i \wedge \ldots \wedge S_{k^i}^i \mapsto C_i \uparrow, \text{ for } 0 \le i \le p$$

- Some production rules specify the down going phase:

The wire corresponding to f must go to 0 if all the paths, are inactive:

$$\neg C_0 \wedge \neg C_0 \wedge \ldots \wedge \neg C_p \mapsto Y_f \downarrow$$

$$\neg S_0^i \wedge \neg S_1^i \wedge \ldots \wedge \neg S_{k^i}^i \mapsto C_i \downarrow, \text{ for } 0 \le i \le p$$

5.2.2 Production rules for the environment

To use the theorem, the circuit needs to be closed. This is obtained by closing the circuit with an environment which must be modeled too. We know that the environment follows the four-phase protocol.

For each input A , we have to generate two production rules: $\neg A_{ack} \mapsto A_i \uparrow$ and $A_{ack} \mapsto A_i \downarrow$, where A_{ack} is the acknowledgement signal of A , and i the value read on A during the computation.

Similarly, for the digit Y , we define two production rules:

- $Y_0 \vee Y_1 \vee \ldots \vee Y_n \mapsto Y_{ack} \uparrow$
- $\neg Y_0 \wedge \neg Y_1 \wedge \ldots \wedge \neg Y_n \mapsto Y_{ack} \downarrow$

where Y_{ack} is the acknowledgement signal of Y , and n its number of wires.

5.3 Non-Interference of the production rules

Let's consider two production rules, $B^+ \mapsto x \uparrow$ and $B^- \mapsto x \downarrow$. They are non-interfering in a computation if and only if $\neg B^+ \vee \neg B^-$ is always true during the computation.

First, we show that the production rules for a path C_i of the MDD are non-interfering:

$$S_0^i \wedge S_1^i \wedge \ldots \wedge S_{k^i}^i \mapsto C_i \uparrow,\ \neg S_0^i \wedge \neg S_1^i \wedge \ldots \wedge \neg S_{k^i}^i \mapsto C_i \downarrow$$

We have to show that

$$A = \neg\left(S_0^i \wedge S_1^i \wedge \ldots \wedge S_{k^i}^i\right) \vee \neg\left(\neg S_0^i \wedge \neg S_1^i \wedge \ldots \wedge \neg S_{k^i}^i\right)$$

is always true during the computation.

$$A = \neg\left(\left(S_0^i \wedge S_1^i \wedge \ldots \wedge S_{k^i}^i\right) \wedge \left(\neg S_0^i \wedge \neg S_1^i \wedge \ldots \wedge \neg S_{k^i}^i\right)\right)$$

$$A = \neg\left(S_0^i \wedge \neg S_0^i \wedge S_1^i \wedge \neg S_1^i \wedge \ldots \wedge S_{k^i}^i \wedge \neg S_{k^i}^i\right)$$

$$A = \neg(0 \wedge 0 \wedge \ldots \wedge 0)$$

$$A = 1$$

So A is always true, **the production rules for the paths are non-interfering.**

Let f be a terminal vertex of the MDD, corresponding to the output wire Y_f. The production rules acting on Y_f are:

$$C_0 \vee C_1 \vee \ldots C_p \mapsto Y_f \uparrow,\ \neg C_0 \wedge \neg C_1 \wedge \ldots \neg C_p \mapsto Y_f \downarrow$$

We have to show that

$$B = \neg\left(C_0 \vee C_1 \vee \ldots \vee C_p\right) \vee \neg\left(\neg C_0 \wedge \neg C_1 \wedge \ldots \wedge \neg C_p\right)$$

is always true during the computation.

$$B = \neg\left(\left(C_0 \vee C_1 \vee \ldots \vee C_p\right) \wedge \left(\neg C_0 \wedge \neg C_1 \wedge \ldots \wedge \neg C_p\right)\right)$$

$$B = \neg\left(\left(C_0 \wedge \neg C_0 \wedge \ldots\right) \vee \ldots \vee \left(C_p \wedge \neg C_0 \wedge \ldots\right)\right)$$

$$B = \neg(0 \vee \ldots \vee 0)$$

$$B = 1$$

So B is always true, **the production rules producing the output are non-interfering.** This is also true for acknowledgment MDDs, as the return path of the circuit is synthesized like the direct path.

5.4 Stability of the production rules

A production rule $G \mapsto t$ is stable if and only if G can change from 1 to 0 only in the states in which $R(t)$ holds. We have to show that all the production rules are stable. But the proof depends on the implementation of the 4-phase protocol that we use. That's why we introduce Lemma 1, which states the part of the proof depending on the protocol implementation:

Lemma 1 *An input A cannot be acknowledged before the output is produced:*

The acknowledgement signal A_{ack} cannot reach the value 1 as long as the output isn't valid (as long as all the wires Y_i of the output have the value 0).

It cannot reach the value 0 as long as the output is valid (as long as one wire Y_i of the output has the value 1).

We show for each implementation of the 4-phase protocol that Lemma 1 is true:

- **Sequential implementation**

The production rule that validates A_{ack} is $C_0 \vee C_1 \vee \cdots \vee C_p \mapsto A_{ack} \uparrow$. So C_{i_0}, one of the C_i, must have the value 1 so that A_{ack} may reach the value 1.

By construction of the acknowledgement MDD of A, we know that Y_{ack} appears in every path, so the production rule that validates C_{i_0} is of the form $Y_{ack} \wedge (\ldots) \mapsto C_{i_0} \uparrow$. So Y_{ack} must have the value 1 for C_{i_0} to take the value 1.

The production rule that validates Y_{ack} is $Y_0 \vee Y_1 \vee \ldots \vee Y_n \mapsto Y_{ack} \uparrow$, so A_{ack} cannot reach the value 1 as long as all the Y_i have the value 0.

Similarly, the production rule that invalidates A_{ack} is $\neg C_0 \wedge \neg C_1 \wedge \cdots \wedge \neg C_p \mapsto A_{ack} \downarrow$, so C_{i_0}, the unique C_i that has the value 1, must reach 0 so that A_{ack} reaches the value 0.

Like previously, the production rule that invalidates C_{i_0} is of the form $\neg Y_{ack} \wedge (\ldots) \mapsto C_{i_0} \downarrow$, so Y_{ack} must have the value 0 for C_{i_0} to take the value 0. The production rule that invalidates Y_{ack} is $\neg Y_0 \wedge \neg Y_1 \wedge \ldots \wedge \neg Y_n \mapsto Y_{ack} \downarrow$

So A_{ack} cannot reach 0 as long as there exists a Y_i that has the value 1.

- **WCHB implementation**

The production rule that validates A_{ack} is $C_0 \vee C_1 \vee \cdots \vee C_p \mapsto A_{ack} \uparrow$. So C_{i_0}, one of the C_i, must have the value 1 for A_{ack} to take the value 1.

By construction of the acknowledgement MDD of A, we know that $V(Y)$ appears in every path, so the production rule that validates C_{i_0} is of the form $V(Y) \wedge (\ldots) \mapsto C_{i_0} \uparrow$. So Y must be valid for C_{i_0} to take the value 1. So A_{ack} cannot reach the value 1 as long as all the Y_i have the value 0.

Similarly, the production rule that invalidates A_{ack} is $\neg C_0 \wedge \neg C_1 \wedge \cdots \wedge \neg C_p \mapsto A_{ack} \downarrow$. So C_{i_0}, the unique C_i that has the value 1, must reach 0 so that A_{ack} reaches 0.

Like previously, the production rule that invalidates C_{i_0} is of the form $\neg V(Y) \wedge (\ldots) \mapsto C_{i_0} \downarrow$. So Y must be invalid for C_{i_0} to take the value 0. So A_{ack} cannot reach 0 as long as there exists a Y_i that has the value 1.

Let us now prove that the production rules are all stable.

5.4.1 Production rules that validate a path

Let's consider a path C_{i_0}. The production rule that validates C_{i_0} is $S_0^i \wedge S_1^i \wedge \ldots \wedge S_{k^i}^i \mapsto C_{i_0} \uparrow$.

Consider that we are in the state where the guard is true, but the transition $C_{i_0} \uparrow$ has not happened.

We show that in this state, the guard cannot become false.

Assume that the guard becomes false. It means that A, one of the S_j^i, reaches the value 0. The production rule that invalidates A is $A_{ack} \mapsto A_i \downarrow$. So it means that A_{ack} has the value 1.

The set of production rules satisfies Lemma 1, so this means that the output Y is valid. The production rule that validates the output is $C_0 \vee C_1 \vee \ldots \vee C_p \mapsto Y_f \uparrow$.

Since the paths C_i are mutually exclusive, and since the guard to C_{i_0} was just valid, the other path cannot reach the value 1. So C_{i_0} has reached the value 1, which is impossible because the transition $C_{i_0} \uparrow$ has not happened.

So the guard cannot become false in this state, **the production rule is stable.**

5.4.2 Production rules that validate the output

Consider a wire Y_f of the output. The production rule that validates Y_f is $C_0 \vee C_1 \vee \ldots \vee C_p \mapsto Y_f \uparrow$.

Consider that we are in the state where the guard is true, but the transition $Y_f \uparrow$ has not happened, so all the wires of Y have the value 0. We show that in this state, the guard cannot become false.

The paths C_i are mutually exclusive, so there is a unique C_{i_0} that has the value 1. Assume that C_{i_0} reaches the value 0. The production rule that invalidates C_{i_0} is $\neg S_0^i \wedge \neg S_1^i \wedge \ldots \wedge \neg S_{k^i}^i \mapsto C_{i_0} \downarrow$.

Let A be one of the S_j^i. C_{i_0} has reached the value 0, so A must have the value 0. The production rule that invalidates A is $A_{ack} \mapsto A_i \downarrow$. So it means that A_{ack} has the value 1. The set of production rules satisfies Lemma 1, so this means that the output Y is valid, which is impossible because the transition $Y_f \uparrow$ has not happened yet.

So the guard cannot become false in this state, **the production rule is stable.**

5.4.3 Production rules that invalidate a path

Let's consider a path C_{i_0}. The production rule that invalidates C_{i_0} is $\neg S_0^i \wedge \neg S_1^i \wedge \ldots \wedge \neg S_{k^i}^i \mapsto C_{i_0} \downarrow$.

Consider that we are in the state where the guard is true, but the transition $C_{i_0} \downarrow$ has not happened. We show that in this state, the guard cannot become false. Suppose that the guard becomes false. It means that A, one of the S_j^i, reaches the value 0. The production rule that validates A is $\neg A_{ack} \mapsto A_i \uparrow$. So it means that A_{ack} has the value 0. The set of production rules satisfies Lemma 1, so this means that the output Y is invalid.

The production rule that invalidates the output is $\neg C_0 \wedge \neg C_1 \wedge \ldots \wedge \neg C_p \mapsto Y_f \downarrow$, so C_{i_0} has reached the value 0, which is impossible because the transition $C_{i_0} \downarrow$ has not happened.

So the guard cannot become false in this state, **the production rule is stable.**

5.4.4 Production rules that invalidate the output

Consider a wire Y_f of the output. The production rule that invalidates Y_f is $\neg C_0 \wedge \neg C_1 \wedge \ldots \wedge \neg C_p \mapsto Y_f \downarrow$.

Consider the state where the guard is true, but the transition $Y_f \downarrow$ has not happened yet, so Y_f has the value 1. We show that in this state, the guard cannot become false:

Let C_{i_0} be one of C_i. It has the value 0. Suppose that C_{i_0} reaches the value 1. The production rule that validates C_{i_0} is

$$S_0^i \wedge S_1^i \wedge \ldots \wedge S_{k^i}^i \mapsto C_{i_0} \uparrow$$

Let A be one of the S_j^i. C_{i_0} has reached the value 1, so A must have the value 1. The production rule that validates A is $\neg A_{ack} \mapsto A_i \uparrow$. So it means that A_{ack} has the value 0. The set of production rules satisfies Lemma 1, so this means that the output Y is invalid, which is impossible because the transition $Y_f \downarrow$ has not happened yet.

So the guard cannot become false in this state, **the production rule is stable.**

6 Conclusion

A new approach to model QDI circuits has been presented. The technique, based on a generalization of the Binary Decision Diagram called Multi-valued Decision Diagram, has been demonstrated on an example of circuit.

The model represents both the data path and the return path of the circuit; QDI circuits have been synthesized from this model using multi-rail logic, with several communication protocols. Moreover, the generated circuits are formally proved to be quasi-delay-insensitive.

Acting on the model modifies the circuit but preserves its QDI property, thus the model allows QDI optimizations of the circuit. Further decompositions and optimizations of the circuits are currently investigated in the group.

References

[1] A.V. Dinh Duc, L. Fesquet, and M. Renaudin. Synthesis of QDI Asynchronous Circuits from DTL-style Petri Net. in 11th IEEE/ACM International Workshop on Logic & Synthesis. 2002. New Orleans, Louisiana.

[2] A.V. Dinh Duc, J.-B. Rigaud, A. Rezzag, A. Sirinanni, J. Fragoso, L. Fesquet, and M. Renaudin. TAST CAD Tools. in ACiD-WG workshop. 2002. Munich, Germany.

[3] A.J. Martin, The Limitations to Delay-Insensitivity in Asynchronous Circuits, in Advanced Research in VLSI, W.J. Dally, Editor. 1990, MIT Press. p. 263--278.

[4] M. Renaudin, Asynchronous circuits and systems: a promising design alternative. Microelectronic Engineering, 2000. 54(1-2): p. 133 - 149.

[5] A.M. Lines, Pipelined Asynchronous Circuits, Caltech, CS-TR-95-21, 1995.

[6] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, and P. Vivet, A New Contactless Smart Card IC using On-Chip Antenna ans Asynchronous Microcontroller. Journal of Solid-State Circuits, 2001. 36: p. 1101-1107.

[7] R. Dreschler and B. Becker, Binary Decision Diagrams, Theory and Implementation. Kluwer Academic Publishers ed. 1998: Kluwer Academic Publishers.

[8] T. Kam, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, Multi-valued decision diagrams: Theory and applications. International Journal on Multiple-Valued Logic, 1998. 4(1-2): p. 9-24.

[9] S. Hassoun and T. Sasao, Logic Synthesis and Verification. Kluwer Academic Publishers ed. 2003: Kluwer Academic Publishers.

[10] J. Sparso, J. Staunstrup, and M. Dantzer-Sorensen, Design of delay insensitive circuits using multi-ring structures, in Proc. European Design Automation Conference (EURO-DAC). 1992, IEEE Computer Society Press: Hamburg, Germany. p. 15--20.

[11] R. Manohar and A.J. Martin, Quasi-delay-insensitive Circuits are Turing Complete, in Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems. 1996, IEEE Computer Society Press.