

MODELING OF DELAY-INSENSITIVE CIRCUIT BUILDING-BLOCKS USING THE HAMBURG DESIGN SYSTEM

Jesse M. Sacayanan and Joel R. Noche

Department of Electrical and Electronics Engineering
College of Engineering, University of the Philippines, Diliman

ABSTRACT

The operation of delay-insensitive circuits, a class of asynchronous logic circuits, is difficult to visualize. Models of delay-insensitive circuit building-blocks are created using the Hamburg Design System (HADES), a pure-Java framework for object-oriented component-based simulation. These models help designers and students visualize the operation of delay-insensitive circuits.

I. Introduction

Asynchronous circuits, digital logic circuits that do not use a global clock signal, have attracted attention this past decade due to their potential advantages over synchronous circuits [1, 2, 3]. Some asynchronous circuits have been shown to consume less power and have less electromagnetic emission than their synchronous counterparts [2]. However, asynchronous circuits are more difficult to design and test, which is one reason most current designs are synchronous.

One class of asynchronous circuits is particularly difficult to design. Delay-insensitive (DI) circuits make no timing assumptions on gate and wire delays [1]. DI circuits are made by connecting DI circuit building blocks. As long as the internal timing assumptions of these blocks are satisfied, the behaviors of circuits composed of these blocks are independent of the speed of operation of the blocks and of the delays in the wires connecting them. DI circuits are thus more robust than other asynchronous circuits, showing the same behaviors regardless of the technology used to implement them and of other physical factors like temperature.

The DI circuits in this work use dual-rail encoding with transition signaling. That is, two wires are used to represent each data signal: a voltage transition on one wire indicates the arrival of a zero, a transition on the other the arrival of a one. The voltage levels on the wires are of no significance, but are in most cases initially assumed to be zero.

Although software is available to verify if a certain circuit composed of DI building blocks has the desired behaviors [4, 5], deciding which building blocks to use and how to connect them to obtain a desired behavior is a challenge. In this paper we present models of DI circuit building blocks created using the Hamburg Design System (HADES) [6, 7], a pure-Java framework for object-oriented component-based simulation.

These models help designers and students visualize the operation of delay-insensitive circuits. The source code for the models and additional details are in [8].

II. Materials and Methods

2.1 Trace Theory

One popular way of representing DI behaviors is through trace theory [1]. A behavior is an interleaving of input and output events [9]. A specification is the set of all admissible and required interface behaviors of a system. The specification models a circuit module by help of a theory: a formal language (syntax) and a formal interpretation (semantics).

In trace theory, lower-case letters serve as symbolic names for communication events at similarly named communication ports. An event is an occurrence of the corresponding action (for example, a voltage transition). There is one-to-one mapping between actions and ports of a module. Sets of input and output actions in a specification are implicit and disjoint: $?$ or $!$ is appended to a symbol name to denote that the name stands for an input or an output action, respectively. These suffixes may be left out for internal signals or when no confusion may arise. [9]

Specifications are written by using the symbol names and applying the operations described in Table 1.

For example, for the specification $\text{pref}^*[a; (b|c)]$, some valid traces are a , ab , ac , aba , and $acab$, while some invalid traces are abc , aa , b , and $acaa$. For the specification $\text{pref}^*[(a|!b); c]$, some valid traces are b , ab , abc , bac , $abcba$, and $bacabc$, while some invalid traces are c , ac , and bb .

VERDECT [4, 5], a program designed to compare circuit specifications with an implementation, uses trace theory.

2.2 DI Building Blocks

A universal and minimal set of building blocks for DI circuits is presented in [9, 10]. Circuits using these building blocks include bit serial adders and multipliers [11]. Although these building blocks can be implemented in CMOS (complementary metal oxide semiconductor) technology [9, 10], these implementations are inefficient and impractical [12]. Event-based technologies are better suited for these building blocks than voltage-level based technologies. Building blocks implemented in RSFQ (rapid single flux quantum) superconductor technology [12] have been shown to compare favorably with standard RSFQ implementations of Boolean logic gates and have been used to design self-timed pipelined parallel adders [13].

Table 2 shows delay-insensitive circuit building-blocks from [9]. The third column shows the formal specification of these building-blocks written in trace theory [1].

The *Fork*, in general, is not isochronic, that is, a transition at the input port does not necessarily arrive at the two output ports at the same time. The output that transitions first is arbitrary.

The *Merge* ensures the orderly arrival of transitions coming from two input ports to an output port.

The *Tria* produces a transition on one of three output ports when it receives transitions from two corresponding input ports.

The *Sequencer* is for arbitration or mutual-exclusion. Two requests, $r0?$ and $r1?$, are accepted but only one of them, either $g0!$ or $g1!$, is granted at any time. The requester is expected to acknowledge through $c?$ before another request is granted.

The *Toggle* produces an output transition for each input transition, with the output transitions distributed between its two output ports alternately, beginning at the port marked with a circle.

The 2×1 *Join* allows transitions on either $b0?$ or $b1?$ (but not both) to cause transitions on $c0!$ or $c1!$, respectively, through a transition on $a?$.

Note that the *Ljoin* is not the same as the *Tria*; the output ports have different corresponding input ports.

The Fork, Merge, Sequencer, and Tria form a minimal set [9, p. 13], that is, any DI circuit can be created using just these blocks.

2.3 *HADES Implementation*

The models of the blocks are written and compiled as Java classes. The behavior of these models is governed by a series of conditions using the method `hasEvent()` from the class `PortStdlogic1164` included in HADES. The method `hasEvent()` returns `true` if there is a change of state on the port it is pointing to.

HADES always starts the simulation with all the wires in an ‘unknown’ state. The models then start to give outputs making the wires go from the ‘unknown’ state to another state, 0 or 1. Since method `hasEvent()` recognizes these transitions, these transitions must be ignored by the models since these transitions are not valid inputs. A problem arises when the number of transitions ignored is not equal to the number of inputs required by a particular model to produce an output, and that model requires two or more inputs. For example, in the 2×1 Join, assume that the first two transitions are “unknown to 0” transitions on $b0?$ and $a?$, then a transition of “unknown to 0” on $b1?$ occurs. If the first valid transition is on $a?$, then the model will produce a transition on $c1!$, which is a violation of its specification.

To address the problem, these models are implemented such that they will ignore the first n transitions at the start of the simulation (at time $t=0$ s), where n is the number of inputs for a certain component. Also, the outputs of the models are set to 0 at the start of the simulation. (This is arbitrary; initial outputs of 1 may also be used.)

Aside from the building-blocks shown in Table 2, two additional models were created: the initialized wire [1] and the delay node (see Figure 1). The delay node is used to model wire delays while the initialized wire has the trace theory specification $\text{pref}^*[b!;a?]$. The inclusion of the initialized wire simplifies many designs.

At first it may seem that a delay node is unnecessary because wire delays can be modeled as delays in the inputs and the fork and gate outputs. But it is shown in Section 3 how the initial ‘unknown’ state used by HADES can lead to incorrect behavior of circuits if no delay nodes are used.

The models were compiled under the package `dibb`. Java Runtime Environment version 1.4.1 was used to run and update HADES. The `jar` executable with update option was used to integrate the models with HADES.

III. Discussion of Results

The correct operation of the models were verified by implementing some case circuits in [9] (all of them using Forks): two decompositions of the 2×2 Join (one using Ljoins, the other using Trias and Merges), a decomposition of the 2-way Resource Arbiter (using a Sequencer, a 2×1 Join, and Merges), a 1-bit wide 4-place Elastic Fifo (composed of 2×1 Joins and Merges) and Modulo-2, -3, -4, and -5 Counters (using 2×1 Joins, Toggles, and Merges). Correct operation was observed for different component and wire delays. The test cases are in [8].

As an example, consider the Modulo- N Counter having the trace theory specification $\text{pref}^*[(a?;a!)^{N-1}a?;b!;b?;a!]$, where $(a?;a!)^{N-1}$ denotes a sequence of $N - 1$ instances of $(a?;a!)$. Thus, a Modulo-2 Counter has the specification $\text{pref}^*[a?;a!;a?;b!;b?;a!]$. Figure 2 shows a screen snapshot of the HADES Editor showing a Modulo-2 Counter with probes on all signals. Figure 3 shows the HADES Waveform Viewer showing some test waveforms for the Modulo-2 Counter, with all delays set to 0.2 second for illustration purposes. Note that the outputs of the blocks are initially in an ‘unknown’ state, but later go to a logic 0 or 1 because none of their inputs are in an ‘unknown’ state.

This initial ‘unknown’ state leads to problems in some circuits such as the Modulo-3 Counter shown in Figure 4. Shown is a graphical representation of the circuit’s logic levels after the trace $a?$. The input $a?$ is at a logic 1 and $b?$ is at a logic 0, which is consistent with the specification. But output $a!$ remains at a logic 0 because one of its inputs (the one leaving the Fork) is in an ‘unknown’ state. Because the 2×1 Join and the Forks are connected in a feedback loop, the signals of some of their inputs and outputs remain in an ‘unknown’ state. The behaviors of this HADES model do not agree

with the specification.

This problem is avoided by inserting delay nodes into the feedback loops. Figure 5 shows the modified circuit's logic levels after the trace `a?;a!`. Figure 6 shows waveforms verifying its correct behavior. (All delays here were set to 0.1 second for illustration purposes.)

IV. Conclusions and Recommendations

A set of delay-insensitive circuit building-block models for HADES that can be used to design delay-insensitive circuits was created. Additional behaviors (ignoring the initial input transitions from 'unknown' states) were included in these models and additional blocks (the initialized wire and the delay node) were created so that HADES could simulate delay-insensitive circuits correctly. The capability of the HADES simulation framework to allow design and simulation of delay-insensitive circuits was also demonstrated.

Aside from aiding designers visualize DI circuit behavior, the models can also be used as an educational tool to help promote research in delay-insensitive circuits and asynchronous circuits in general. Since HADES is designed with ease of use as one feature, this software may become appealing to students and may raise their interests in digital circuits in general.

References

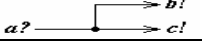
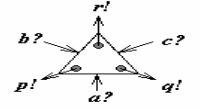
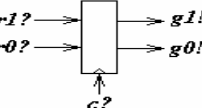
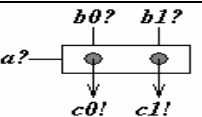
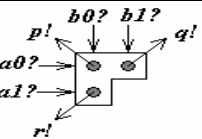
1. S. Hauck, "Asynchronous design methodologies: An overview," *Proceedings of the IEEE* **83** (1), 69–93 (1995).
2. C. van Berkel, M. Josephs, and S. Nowick, "Applications of asynchronous circuits," *Proceedings of the IEEE* **87** (2), 223–233 (1999).
3. I. Sutherland and J. Ebergen, "Computers without clocks," *Scientific American* **287** (8), 46–53 (2002).
4. J. Bergen and R. Berks, "VERDECT: A verifier for asynchronous circuits," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Special Issue on Asynchronous Computer Architecture* (1995).
5. VERDECT, <http://edis.win.tue.nl/verdict/> (Accessed 2002).
6. N. Hendrich, HADES Tutorial ver 0.9 (2002).
7. HADES simulation framework homepage, <http://tech-www.informatik.uni-hamburg.de/applets/hades/html/> (Accessed 2002).
8. J. Sacayanan, "Modeling and simulation of delay-insensitive circuit building-blocks using HADES simulation framework," Undergraduate student project, University of the Philippines, Diliman (2003).
9. P. Patra and D. Fussell, "Building-blocks for designing DI circuits," Technical Report TR93–23, Department of Computer Sciences, University of Texas at Austin (1993).
10. P. Patra and D. Fussell, "Efficient building blocks for delay insensitive circuits," in the *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 196–205 (1994).

11. P. Patra and. Fussell, “Fully asynchronous, robust, high-throughput arithmetic structures,” in the *International Conference on VLSI Design* (1995).
12. P. Patra, S. Polonsky, and D. Fussell, “Delay insensitive logic for RSFQ superconductor technology,” in the *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 42–53 (1997).
13. Y. Kameda, S. Polonsky, M. Maezawa, and T. Nanya, “Self-timed parallel adders based on DI RSFQ primitives,” *IEEE Transactions on Applied Superconductivity* **9**, 4040–4045 (1999).

Table 1
Trace theory commands [1]

Name	Syntax	Meaning	Example
Concatenation	$\langle \text{cmd1} \rangle ; \langle \text{cmd2} \rangle$	$\langle \text{cmd2} \rangle$ follows $\langle \text{cmd1} \rangle$	$a; b = \{ab\}$
Union	$\langle \text{cmd1} \rangle \langle \text{cmd2} \rangle$	Either $\langle \text{cmd1} \rangle$ or $\langle \text{cmd2} \rangle$	$a b = \{a, b\}$
Repetition	$*[\langle \text{cmd} \rangle]$	Zero or more concatenations of $\langle \text{cmd} \rangle$	$*[a] = \{\epsilon, a, aa, \dots\}$
Prefix-Closure	$\text{pref} \langle \text{cmd} \rangle$	Any prefix of $\langle \text{cmd} \rangle$	$\text{pref}(ab) = \{\epsilon, a, ab\}$
Projection	$\langle \text{cmd} \rangle \downarrow \langle \text{alph} \rangle$	Remove all symbols from $\langle \text{cmd} \rangle$ not contained in $\langle \text{alph} \rangle$	$abc \downarrow \{a, c\} = \{ac\}$
Weave	$\langle \text{cmd1} \rangle \parallel \langle \text{cmd2} \rangle$	Shuffling of $\langle \text{cmd1} \rangle$ and $\langle \text{cmd2} \rangle$, with shared symbols occurring simultaneously	$abc \parallel acd = \{abcd, acbd\}$

Table 2
Delay-insensitive building-blocks [9]

Name	Schematic	Specification
Fork		$\text{pref}^*[a?; (b! c!)]$
Merge		$\text{pref}^*[(a? b?); c!]$
Tria		$\text{pref}^*[(a? b?); p!] $ $((a? c?); q!) $ $((b? c?); r!)]$
Sequencer		$\text{pref}^*[(r0?; g0!) $ $*(r1?; g1!) $ $*((g0! g1!); c?)]$
Toggle		$\text{pref}^*[a?; c!; a?; d!]$
2x1 Join		$\text{pref}^*[(a? b0?); c0!] $ $((a? b1?); c1!)]$
Ljoin		$\text{pref}^*[(a0? b0?); p!] $ $((a0? b1?); q!) $ $((a1? b0?); r!)]$

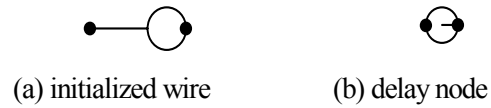


Figure 1. Initialized wire and delay node symbols

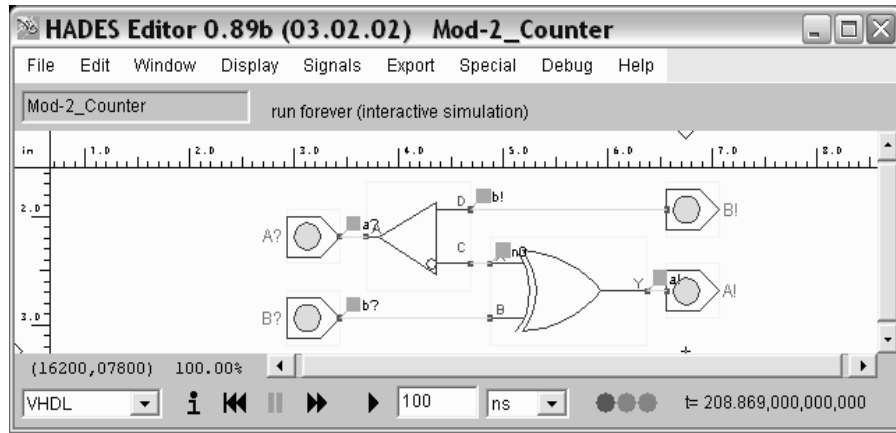


Figure 2. Screen snapshot of the HADES Editor and a Modulo-2 Counter

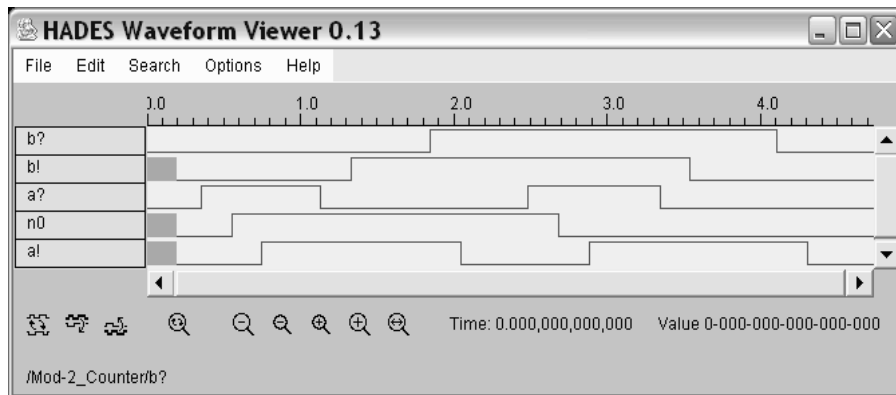


Figure 3. The HADES Waveform Viewer and test waveforms for the Modulo-2 Counter

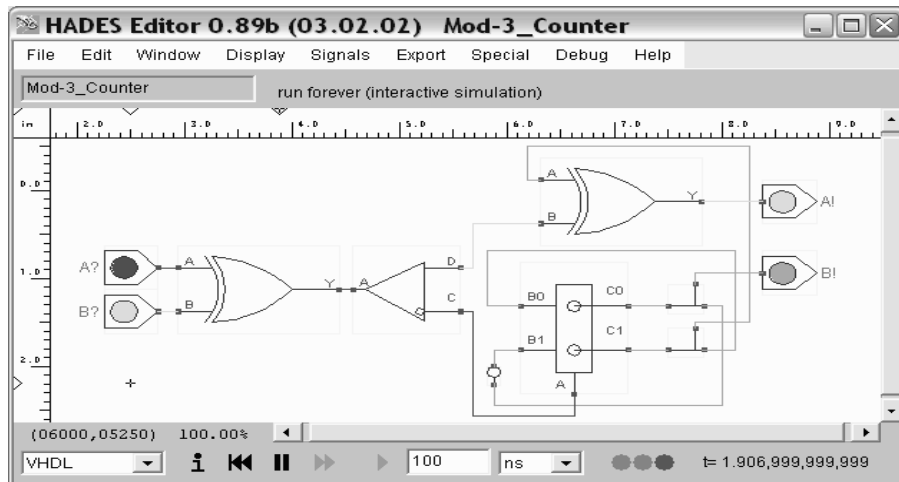


Figure 4. A Modulo-3 Counter showing incorrect behavior

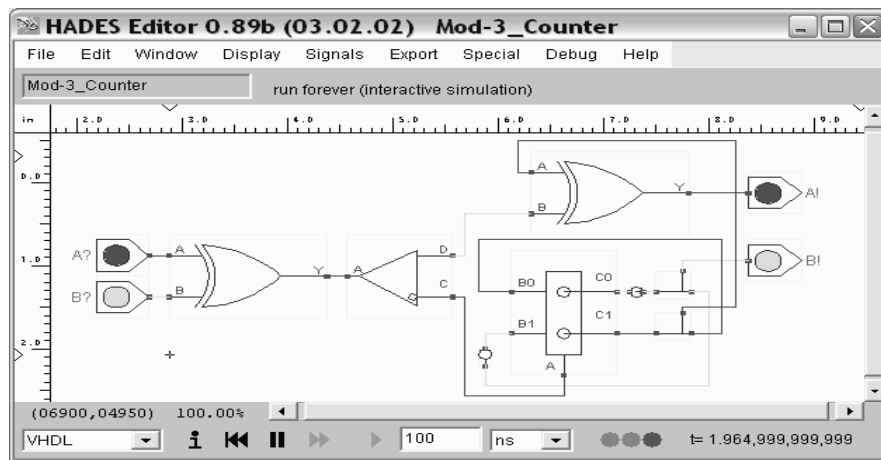


Figure 5. A Modulo-3 Counter showing correct behavior

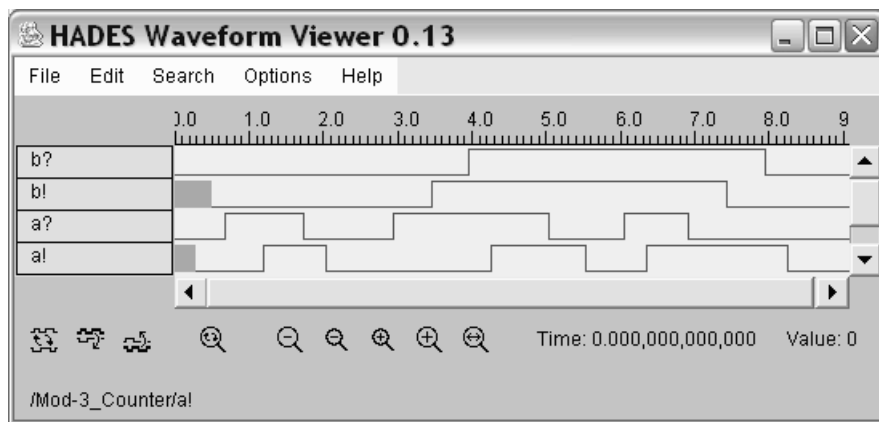


Figure 6. Waveforms showing correct behavior of the circuit in Figure 5