Resetting Asynchronous QDI Systems

Thesis by
Xiaofei Chang

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

Submitted September, 2013

Acknowledgements

I wish to thank my advisor Dr. Alain Martin for his patience and guidance. Thanks also to my seniors Chris Moore and Sean Keller for their discussion and encouragement. At last, I would like to thank my wife for her support and love.

Abstract

Quasi Delay-Insensitive (QDI) systems must be reset into a valid initial state before normal operation can start. Otherwise, deadlock may occur due to wrong handshake communication between processes. This thesis first reviews the traditional Global Reset Schemes (GRS). It then proposes a new Wave Reset Schemes (WRS). By utilizing the third possible value of QDI data codes - reset value, WRS propagates the data with reset value and triggers Local Reset (LR) sequentially. The global reset network for GRS can be removed and all reset signals are generated locally for each process. Circuits templates as well as some special blocks are modified to accommodate the reset value in WRS. An algorithm is proposed to choose the proper Local Reset Input (LRI) in order to shorten reset time. WRS is then applied to an iterative multiplier. The multiplier is proved working under different operating conditions.

Contents

List of Figures

List of Tables

1 Introduction

Asynchronous systems with the only delay assumption of isochronic forks are called Quasi Delay-Insensitive (QDI) systems [1]. QDI systems are composed of concurrent modules called processes. Processes communicate with each other by sending and receiving actions on channels. Each channel is comprised of one or several directed data wires and one directed enable wire. Data wires from a sender process to a receiver process encode the message being sent while the enable wire from a receiver process to a sender process is used to notify the sender process that the message has been received.

During normal operation, communication between processes occurs concurrently with computation inside each process. However, in order for a QDI system to transit into normal operation after power-on, it needs to be driven into the valid initial state that is determined by Martin's synthesis. The procedure that brings a system into its valid initial state after power-on is called reset. The valid initial state of a QDI system is determined by the valid initial state of each process in the system. The valid initial state of a process is determined by valid initial values of all wires inside the process. Therefore each wire in the system needs to be driven to the valid initial value before normal operation of the system starts.

This thesis discusses two methods to reset a QDI system, namely Global Reset Scheme (GRS) and Wave Reset Scheme (WRS). They are different in terms how they distribute the reset signal to each process. GRS distributes Global Reset (GR) to each process and resets them at the same time by asserting GR. For WRS, GR is connected to only a small number of processes. When GR is asserted, reset data is generated at the outputs of these processes and propagates to other processes. Once reset data arrives at a process, it triggers Local Reset Generator (LRG) of that process. LRG asserts Local Reset (LR) and forces the process to output reset data. The input or inputs that drive LRG are Local Reset Input (LRI). This generation and propagation of reset data continues until all processes have output reset data.

During reset phase, if no timing assumption other than isochronic fork [2] is made, the value of each wire needs to be checked to make sure the system is in the valid initial state. However, checking each wire requires a large number of logic gates which introduce an unacceptable overhead in terms of area, delay and power consumption. Instead, a reset timing assumption is made that guarantees the system will reach the valid initial state within reset time. Reset time is determined by the longest path that contains reset gates at the head and tail and logic gates in between. GR needs to be asserted at least as long as reset time in order for all wires to be driven to valid initial

values.

The rest of the thesis is organized as follows. Section 2 and Section 3 respectively discuss GRS and WRS including operation protocol, pipeline templates and special blocks. Section 4 discusses the choice of LRI in order to reduce reset time for WRS. The WRS is applied to a multiplier in Section 5. Section 6 concludes the thesis.

2 GRS

For a given process P, the initial value of any input wire I is determined by the gate inside the neighboring process that drives I. Initial values of internal wires and output wires are determined by gates inside P that drive them. If initial values of input wires and corresponding gates can drive all internal wires and output wires to the valid initial values, P will be reset to the valid initial state once neighboring processes have been reset to the valid initial states. Otherwise, reset logic needs to be added in order to force internal wires and output wires to assume valid initial values.

Added reset logic converts logic gates into reset gates. Each reset gate has an extra input that is connected to GR. When GR is asserted, the output of a reset gate is driven to the valid initial value independent of the other inputs. If all logic gates in a QDI system are converted into reset gates, the system is guaranteed to be in the valid initial state because GR forces each wire to assume the valid initial value. However, converting all logic gates into reset gates slow down normal operation of the system because reset gates are slower than corresponding logic gates. Therefore, only necessary logic gates should be converted into reset gates. To be more specific, only logic gates that can't drive their outputs to valid initial values based on valid initial values of inputs should be converted into reset gates. Once there are enough reset gates, each wire will be driven to the valid initial value when GR is asserted and the system will be driven to the valid initial state. After reset phase, GR is deasserted and normal operation starts. Reset gates function in the same way as the original logic gates during normal operation.

2.1 Operation Protocol

Delay-Insensitive (DI) codes are used in QDI systems for data communication. DI codes encode validity and neutrality of data within the data itself. There are many DI codes, among which dual-rail codes and 1-of-n codes are normally used. Each wire in dual-rail codes or 1-of-n codes is used for one value of the data. When all wires are 0s, the data is neutral. When one of the wires is 1, the data is valid. For example, dual-rail codes are shown in Table 1. When $(wire1, wire0) = (0, 0)$, data is neutral. When $(wire0, wire1) = (0, 1)$ or $(wire0, wire1) = (1, 0)$, data is valid 1 or valid 0. The rest combination of values $((wire1, wire0) = (1, 1))$ is invalid.

Similarly, 1-of-4 codes are shown in Table 2. When all four wires are 0s, the data is neutral. When one of the wires is 1, the data is respectively valid 0, 1, 2, 3. All the other combinations of values are invalid.

Value	$wire1$	$wire0$
neutral	0	0
valid 1	1	0
valid 0	0	1

Table 1: Dual-Rail Data Codes

Value	wire3	wire2	wire1	wire0
neutral	0	0	0	0
valid 3	1	0	0	0
valid 2	0	1	0	0
valid 1	0	0	1	0
valid 0	0	0	0	1

Table 2: 1-of-4 Data Codes

Besides data wires encoded as 1-of-n, each channel contains another *enable* wire. Data wires from a sender process to a receiver process encode the message being sent while the *enable* wire is used by the receiver process to notify the sender process that the message has been received. The communication protocol between a sender process and a receiver process is shown in (1), where $v()$ and $n()$ are validity and neutrality tests.

$$\begin{aligned} &\text{sender:} \quad data\uparrow;\ [\neg enable];\ data\downarrow;\ [enable] \\ &\text{receiver:} \quad [v(data)];\ enable\downarrow;\ [n(data)];\ enable\uparrow \end{aligned} \tag{1}$$

Processes in QDI systems can be divided into two groups, Initial Processes (IP) and Non-Initial Processes (NIP). During normal operation, IP start by sending valid data while NIP start by waiting for valid data. For QDI systems with GRS, the defined operation protocol is valid for a chosen Handshake Expansion (HSE) implementation of the Communication Hardware Processes (CHP) description. During reset phase when active-low GR is asserted, data wires between processes are reset to neutral while *enable* wires are driven to 0s. Once GR is deasserted, neutral data wires drive *enable* wires to 1s and NIP are ready to receive valid data while IP start sending valid data. Therefore during normal operation, IP send and receive alternating valid and neutral data while NIP receive and send alternating valid and neutral data. Alternating valid and neutral data propagates inside the system forever.

2.2 Pipelines

In order to increase throughput, computation is pipelined. Each process forms a pipeline stage. Most pipeline stages repeat actions of receiving data from inputs, computing functions of data and sending results through outputs as described by CHP in (2), where $I_0, I_1, ...I_{n-1}$, $O_0, O_1, ..., O_{m-1}$ and $f_0(X), f_1(X), ..., f_m(X)$ are respectively inputs, outputs and functions while X refers to the set of variables $\{x_0, x_1, ..., x_{n-1}\}$. Because they share similar communication sequences, they can be implemented with templates. There are two types of templates to implement pipeline stages: pipelines with split control and data and fine-grain integrated pipelines.

$$*[I_0?x_0, I_1?x_1, ..., I_{n-1}?x_{n-1}; O_0!f_0(X), O_1!f_1(X), ..., O_{m-1}!f_{m-1}(X)] \quad (2)$$

2.2.1 Pipelines with Split Control and Datapath for GRS

In the first approach, control and datapath of each pipeline stage are separated and implemented independently. As shown in Figure 1, each pipeline stage contains three major components. CTRL is control logic that sequences communication actions. REG is a register used to store data received from inputs. FUNC applies function on stored data and sends results through outputs. REG and FUNC form datapath that is separated from control logic. GR *reset* is connected to all three blocks.

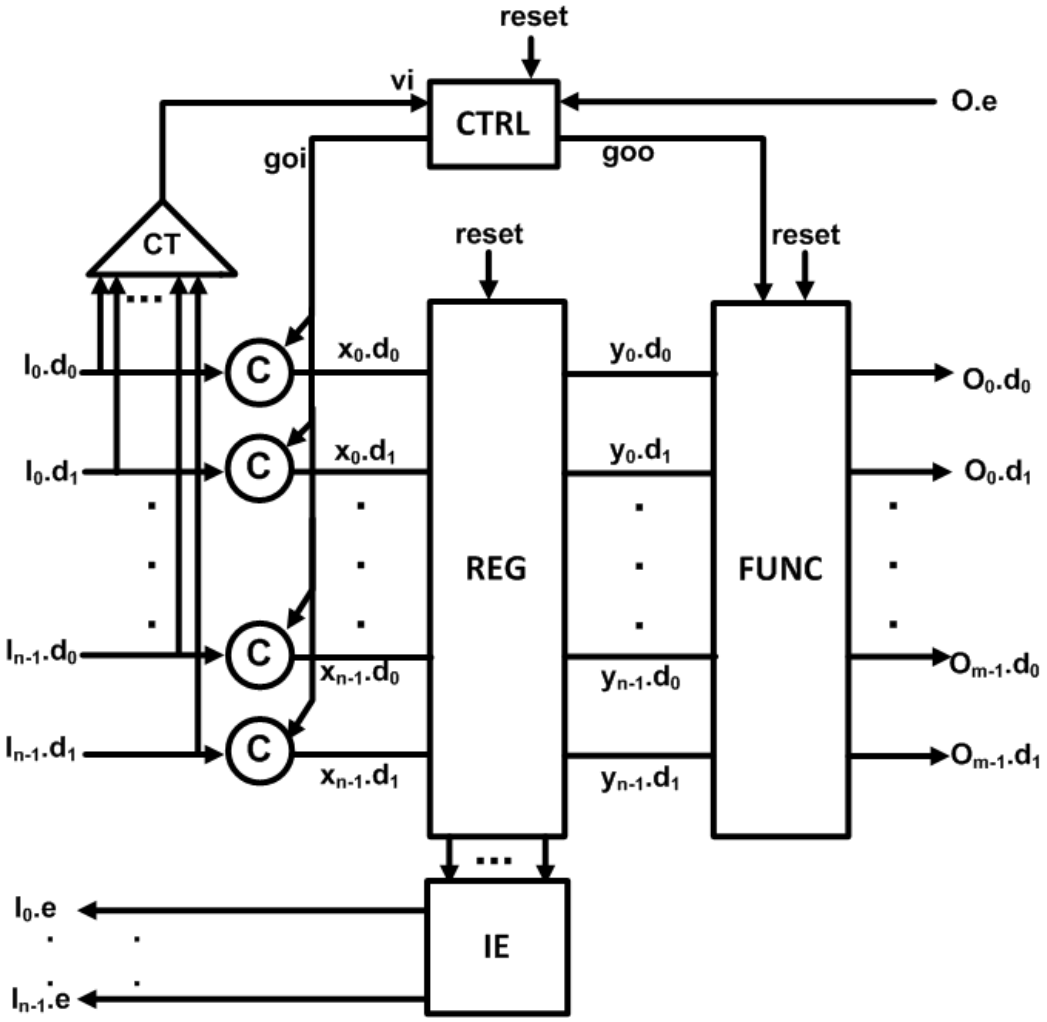

Figure 1: Pipeline Stage with Split Control and Datapath for GRS

2.2.1.1 Control Logic The expected state transition of CTRL is described by HSE as shown in (3). CTRL first enables REG to receive data from inputs ($goi\uparrow$). Once data has been latched by REG ($[(x_0 \wedge y_0) \vee (x_1 \wedge y_1)]$), CTRL acknowledges the input ($I.e\downarrow$). It then enables FUNC to compute functions on received data and output computed results ($goo\uparrow$). Other statements in (3) are used to complete four-phase handshake protocol.

$$*\,[[vi]; goi\uparrow; [\neg vi]; goi\downarrow; goo\uparrow; [\neg O.e]; goo\downarrow; [O.e]] \tag{3}$$

The circuit implementation of CTRL is shown in Figure 2. The state transition of the circuit is described by HSE in (4).

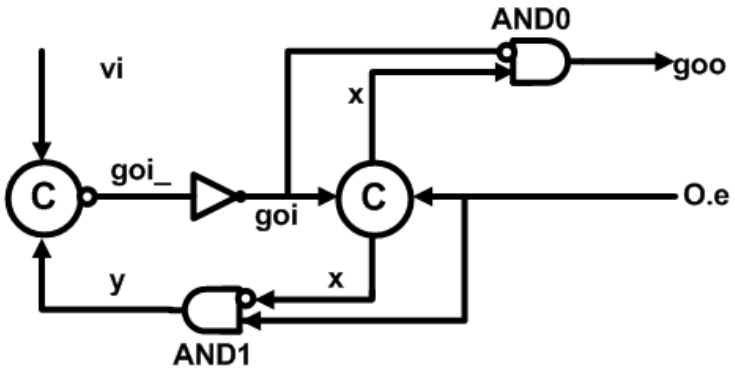

Figure 2: Control Logic for GRS

$$
\begin{array}{l}
\quad [\neg reset]; [\neg vi], [\neg O.e], reset\_\uparrow; (y\downarrow); goi\_\uparrow; goi\downarrow; x\downarrow; goo\downarrow; \\
\quad [reset]; [O.e]; \\
*\,[[vi], y\uparrow; goi\_\downarrow; goi\uparrow; (x\uparrow; y\downarrow); \\
\quad [\neg vi]; goi\_\uparrow; goi\downarrow; goo\uparrow; [\neg O.e]; x\downarrow; goo\downarrow; [O.e]]
\end{array}
\tag{4}
$$

If the internal variables are removed, (4) is simplified to (5).

$$
\begin{array}{l}
\quad [\neg reset]; [\neg vi], [\neg O.e], reset\_\uparrow; goi\downarrow; goo\downarrow; \\
\quad [reset]; [O.e]; \\
*\,[[vi]; goi\uparrow; [\neg vi]; goi\downarrow; goo\uparrow; [\neg O.e]; goo\downarrow; [O.e]]
\end{array}
\tag{5}
$$

The nonterminating repetition part inside $*[\ ]$ of (5) implements the expected transition in (3). Therefore, the circuit implementation is correct.

The input vi of CTRL is the output of the CT block that represents the Completion Tree. It's implemented as shown in Figure 3. The validity of each bit is combined with the C-element tree to generate vi.

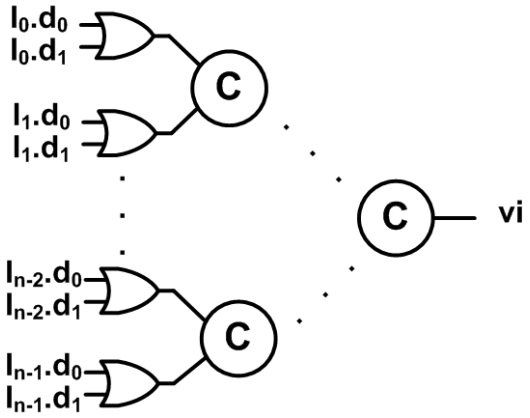

Figure 3: Completion Tree

2.2.1.2 Register REG is comprised of n 1-bit registers. The 1-bit register is implemented as shown in Figure 4. During reset phase, $(x_0.d_0, x_0.d_1) = (0, 0)$. M0 and M1 are cut off. The cross-coupled inverters may enter the metastable state. However, the probability that the metastable state retains through the whole reset phase is negligible. Therefore, before normal operation starts, $(y_0.d_0.y_0.d_1) = (0, 1)$ or $(y_0.d_0.y_0.d_1) = (1, 0)$.

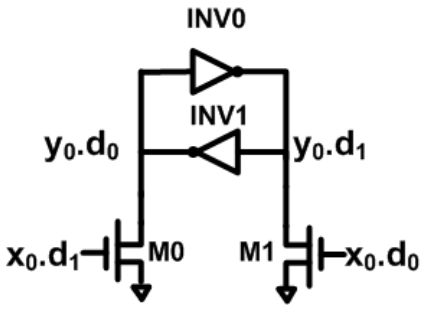

Figure 4: Register for GRS

During normal operation, if (x_0, x_1) is neutral, (y_0, y_1) retains the previous data. If (x_0, x_1) is valid, it is latched into (y_0, y_1).

The IE block in Figure 1 is used to generate *enable* signals for all inputs. It contains n copies of circuits shown in Figure 5. During reset phase, *enable* signals are driven to 0s. During normal operation, if the input is neutral, the enable signal is driven to 1. If the input is valid and latched by the register, the enable signal is driven to 0.

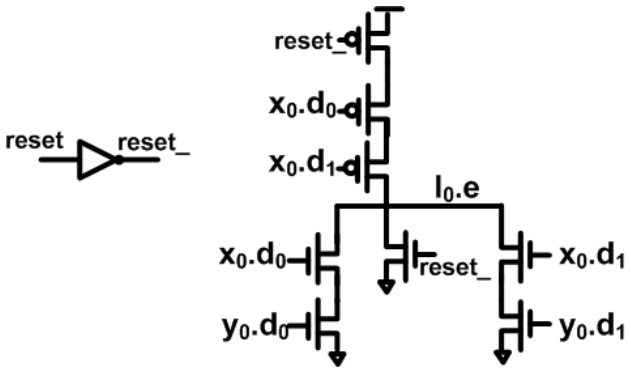

Figure 5: Input Enable Generator for GRS

2.2.1.3 Function Block FUNC contains m copies of circuit blocks shown in Figure 6 to generate m different outputs. f_0 can be any combinational logic function with inputs of goo and $(y_0.d_0, y_0.d_1)$ to $(y_{n-1}.d_0, y_{n-1}.d_1)$. The state transition of the circuit block is described in (6).

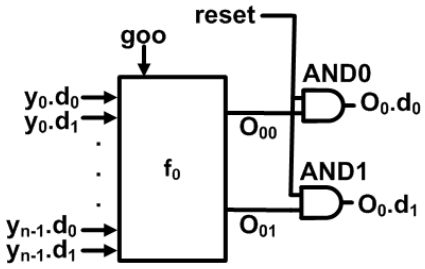

Figure 6: Function Block for GRS

$$
\begin{aligned}
&([\neg reset]; O_0.d_0\downarrow, O_0.d_1\downarrow),\\
&([\neg goo]; O_{00}\downarrow, O_{01}\downarrow), ([(y_0.d_0 \text{ xor } y_0.d_1) \wedge ... \wedge (y_{n-1}.d_0 \text{ xor } y_{n-1}.d_1)]);\\
&[reset];\\
*\,&[[goo \wedge f_{01}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{00}\downarrow, O_{01}\uparrow; O_0.d_0\downarrow, O_0.d_1\uparrow\\
&[]goo \wedge f_{00}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{01}\downarrow, O_{00}\uparrow; O_0.d_1\downarrow, O_0.d_0\uparrow\\
&[\neg r]; O_{00}\downarrow, O_{01}\downarrow; O_0.d_0\downarrow, O_0.d_1\downarrow]
\end{aligned}
\tag{6}
$$

During reset phase, $(O_0.d_0, O_0.d_1) = (0, 0)$. During normal operation, the logic function is applied at the registered data and the results are sent out through outputs.

2.2.1.4 Complete Pipeline Stage The state transition of the whole pipeline stage shown in Figure 1 is described by HSE in (7). During reset phase, all inputs and outputs are neutral and all enable signals are driven to 0s. During normal operation, alternating valid and neutral data comes from the inputs and corresponding alternating valid and neutral data is generated at the outputs. Therefore the implementation is consistent with GRS operation protocol.

$$
\begin{array}{l}
[\neg reset]; [n(I_0) \wedge ... \wedge n(I_{n-1})], \\
(O_0.d_0\downarrow, O_0.d_1\downarrow, ..., O_{m-1}.d_0\downarrow, O_{m-1}.d_1\downarrow), \\
I_0.e\downarrow, ..., I_{n-1}.e\downarrow, [\neg O_0.e \wedge ... \wedge \neg O_{m-1}.e]; \\
vi\downarrow; goi\downarrow; goo\downarrow, x_0.d_0\downarrow, x_0.d_1\downarrow, ..., x_{n-1}.d_0\downarrow, x_{n-1}.d_1\downarrow; \\
[reset]; I_0.e\uparrow, ..., I_{n-1}.e\uparrow, [O_0.e \wedge ... \wedge O_{m-1}.e]; \\
* [[v(I_0) \wedge ... \wedge v(I_{n-1})]; vi\uparrow; goi\uparrow; \\
[I_0.d_0 \rightarrow x_0.d_0\uparrow; y_0.d_0\uparrow [] I_0.d_1 \rightarrow x_0.d_1\uparrow; y_0.d_1\uparrow], \\
..., \\
[I_{n-1}.d_0 \rightarrow x_{n-1}.d_0\uparrow; y_{n-1}.d_0\uparrow [] I_{n-1}.d_1 \rightarrow x_{n-1}.d_1\uparrow; y_{n-1}.d_1\uparrow]; \\
I_0.e\downarrow, ..., I_{n-1}.e\downarrow; [n(I_0) \wedge ... \wedge n(I_{n-1})]; vi\downarrow; goi\downarrow; \\
x_0.d_0\downarrow, x_0.d_1\downarrow, ..., x_{n-1}.d_0\downarrow, x_{n-1}.d_1\downarrow; I_0.e\uparrow, ..., I_{n-1}.e\uparrow; goo\uparrow; \\
[f_{00}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_0.d_0\uparrow \\
[] f_{01}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_0.d_1\uparrow], \\
..., \\
[f_{(m-1)0}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{m-1}.d_0\uparrow \\
[] f_{(m-1)1}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{m-1}.d_1\uparrow]; \\
[\neg O_0.e \wedge ... \wedge \neg O_{m-1}.e]; goo\downarrow; \\
O_0.d_0\downarrow, O_0.d_1\downarrow, ..., O_{m-1}.d_0\downarrow, O_{m-1}.d_1\downarrow; [O_0.e \wedge ... \wedge O_{m-1}.e]]
\end{array}
\tag{7}
$$

2.2.2 Fine-Grain Integrated Pipelines for GRS

In the second approach, control logic and datapath are integrated into a single component. Three commonly used templates, namely Weak-Conditioned Half Buffers (WCHB), Precharged Half Buffers (PCHB) and Precharged Full Buffers (PCFB) [3], are modified to adapt to GRS.

2.2.2.1 WCHB Dual-Rail Buffer for GRS WCHB dual-rail buffer is described in (8) and implemented in Figure 7.

$$
\begin{array}{l}
[\neg reset]; I.e\downarrow, [\neg O.e \wedge \neg I.d_0 \wedge \neg I.d_1]; O.d_0\downarrow, O.d_1\downarrow; \\
[reset]; I.e\uparrow; \\
*\,[[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow\, []I.d_1 \wedge O.e \rightarrow O.d_1\uparrow]; I.e-\downarrow \\
[\neg I.d_0 \wedge \neg I.d_1 \wedge \neg O.e]; O.d_0-, O.d_1-; I.e+]
\end{array}
\tag{8}
$$

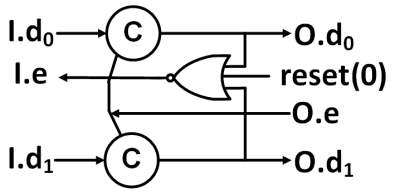

Figure 7: WCHB Dual-Rail Buffer for GRS

$reset(0)$ indicates that once the active-low $reset$ is asserted, $I.e$ is driven to 0 independent of $O.d_0$ and $O.d_1$. The neighboring processes should follow the same operation protocol. $O.e$ is driven to 0 during reset phase. In addition, $(I.d_0, I.d_1) = (0, 0)$ because the outputs of IP are reset to neutral value during reset phase. Both inputs to the C-elements are 0s and $(O.d_0, O.d_1) = (0, 0)$. The neutral data from IP propagates to NIP and generates neutral data at outputs of NIP. The neutral data further propagates to other NIP until all processes have output neutral data.

Once $reset$ is deasserted, $I.e$ is driven to 1 since $(O.d_0, O.d_1) = (0, 0)$. $O.e$ from the neighboring process is driven to 1 as well. Once valid data arrives at $(I.d_0, I.d_1)$, it will be propagated to $(O.d_0, O.d_1)$. The behavior of the implementation is consistent with the operation protocol of GRS.

2.2.2.2 PCHB Dual-Rail Buffer for GRS PCHB dual-rail buffer for GRS is described in (9) and implemented as shown in Figure 8.

$$
\begin{array}{l}
[\neg reset]; I.e\downarrow, en\downarrow, [\neg O.e]; O.d_0\downarrow, O.d_1\downarrow, [\neg I.d_0 \wedge \neg I.d_1]; \\
[reset]; I.e\uparrow; \\
*\,[[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow\, []I.d_1 \wedge O.e \rightarrow O.d_1\uparrow]; I.e\downarrow; \\
[\neg O.e]; O.d_0\downarrow, O.d_1\downarrow; [\neg I.d_0 \wedge \neg I.d_1]; I.e\uparrow]
\end{array}
\tag{9}
$$

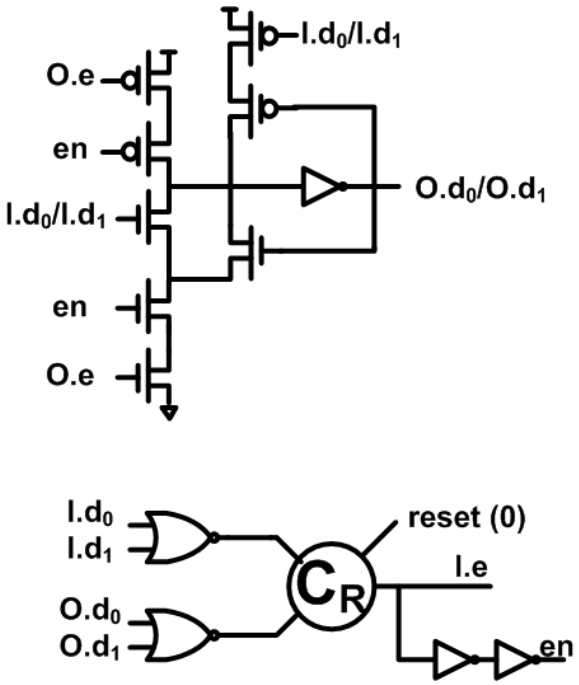

Figure 8: PCHB Dual-Rail Buffer for GRS

C_R is the reset gate for a normal C-element. Depending on whether the initial value is 0 or 1, it is implemented as shown in Figure 9.

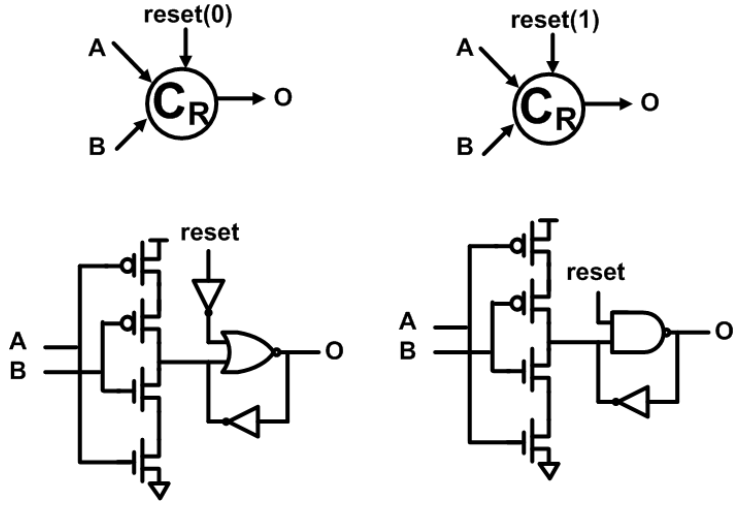

Figure 9: Reset Gate for C-element

When *reset* is asserted, $I.e$ is driven to 0 as well as *en*. The neighboring

processes should follow the same operation protocol. $O.e$ is driven to 0 during reset phase. Therefore $O.d_0$ and $O.d_1$ are driven to 0s. Similarly, the output of the previous stage should be driven to $(0,0)$, i.e., $(I.d_0, I.d_1) = (0,0)$.

Once $reset$ is deasserted, $I.e$ is driven to 1 as well as en since $(I.d_0, I.d_1) = (0,0)$ and $(O.d_0, O.d_1) = (0,0)$. Similarly, $O.e$ from the neighboring process is driven to 1. Once valid data arrives at $(I.d_0, I.d_1)$, it will be propagated to $(O.d_0, O.d_1)$. The behavior of the implementation is consistent with the operation protocol of GRS.

2.2.2.3 PCFB Dual-Rail Buffer for GRS PCFB dual-rail buffer for GRS is described in (10) and implemented as shown in Figure 10.

$$\begin{array}{l} [\neg reset]; I.e\downarrow, x\downarrow, [\neg O.e]; O.d_0\downarrow, O.d_1\downarrow, [\neg I.d_0 \wedge \neg I.d_1]; \\ [reset]; I.e\uparrow, x\uparrow; \\ *\,[[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow \,[]I.d_1 \wedge O.e \rightarrow O.d_1\uparrow]; I.e\downarrow; x\downarrow \\ [\neg O.e]; O.d_0-, O.d_1-; [\neg I.d_0 \wedge \neg I.d_1]; I.e\uparrow; x\uparrow] \end{array} \tag{10}$$

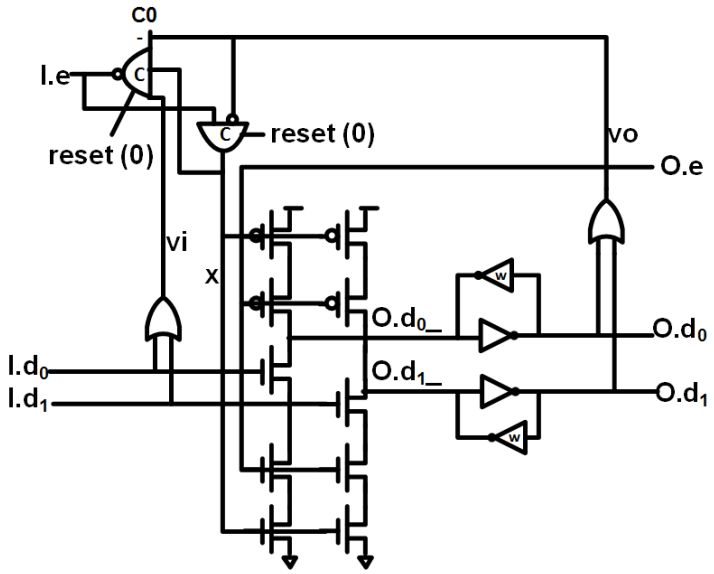

Figure 10: PCFB Dual-Rail Buffer for GRS

$C0$ is an asymmetrical C-element. It is implemented as described by Production Rule Set (PRS) in (11).

$$\begin{array}{r} \neg reset \vee (vi \wedge x \wedge vo) \rightarrow I.e\downarrow \\ reset \wedge \neg vi \wedge \neg x \rightarrow I.e\uparrow \end{array} \tag{11}$$

When *reset* is asserted, $I.e$ is driven to 0 as well as the internal variable x. The neighboring processes should follow the same operation protocol. $O.e$ is driven to 0 during reset phase. Therefore, $(O.d_{0\_}, O.d_{1\_}) = (1, 1)$ and $(O.d_0, O.d_1) = (0, 0)$. Similarly, the output of the previous stage should be driven to $(0, 0)$, i.e., $(I.d_0, I.d_1) = (0, 0)$.

Once *reset* is deasserted, $I.e$ is driven to 1 as well as the internal variable x. Similarly, $O.e$ from the neighboring process is driven to 1. Once valid data arrives at $(I.d_0, I.d_1)$, it will be propagated to $(O.d_0, O.d_1)$. The behavior of the implementation is consistent with the operation protocol of GRS.

3 WRS

For WRS, the Global Reset (GR) is connected to Initial Processes (IP). Once GR is asserted, IP will output reset data that is data with reset value. Reset value is the third possible value besides neutral and valid values for given data codes. Reset data propagates and triggers the Local Reset Generator (LRG) of each process. LRG asserts the Local Reset (LR) and forces the process to output reset data. This propagation of reset data continues until all processes have been reset. After that, GR is deasserted and neutral data will be generated from IP. Neutral data propagates and overwrites all reset data. In addition, LRG can't be triggered by neutral or valid data and no reset data will be generated. The system will operate normally with only neutral and valid data.

3.1 Operation Protocol

In order to include the third possible value - reset value, both dual-rail codes and 1-of-n codes need to be modified. For the modified dual-rail codes, the rest value $(wire1, wire0) = (1, 1)$ is used to be the reset value. For the modified 1-of-n ($n \geq 3$) codes, reset value is defined as two wires being 1s while the rest wires being 0s. Reset value is chosen in this way to make implementation of LRG simple. For example, modified dual-rail codes and 1-of-4 codes for WRS are respectively shown in Table 3 and Table 4.

Value	$wire1$	$wire0$
neutral	0	0
valid 1	1	0
valid 0	0	1
reset	1	1

Table 3: Dual-Rail Data Codes for WRS

For QDI systems with WRS, GR is only connected to IP. During reset phase when GR is asserted, IP will output reset data. Reset data propagates to other processes and triggers LRG as shown in Figure 11. LRG drives active-low LR to 0 and the process will output reset data independent of other inputs. This propagation of reset data continues until all processes have output reset data. The time it takes between that GR is asserted and all processes have output reset data is called reset time.

Value	wire3	wire2	wire1	wire0
neutral	0	0	0	0
valid 3	1	0	0	0
valid 2	0	1	0	0
valid 1	0	0	1	0
valid 0	0	0	0	1
reset	0	0	1	1

Table 4: 1-of-4 Data Codes for WRS

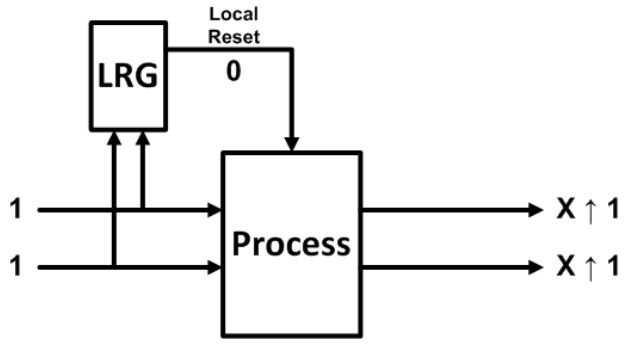

Figure 11: Process with LRG for WRS

After that, GR is deasserted and IP will output alternating neutral and valid data. Neutral data propagates and overwrites all reset data. Besides, no new reset data will be generated since LRG will not be triggered by neutral or valid data. From then on, the system has only neutral and valid data propagating inside. Therefore, processes in the system with WRS follow the operation protocol of receiving and generating reset data followed by receiving and generating alternating neutral and valid data.

3.2 Pipelines

As in QDI systems with GRS, there are two approaches to implement pipelines in QDI systems with WRS. However, templates need to be modified in order to accommodate the extra reset value in dual-rail or 1-of-n codes for WRS.

3.2.1 Pipelines with Split Control and Datapath for WRS

In the first approach, control and datapath of each pipeline stage are separated and implemented independently. As shown in Figure 12, each pipeline

stage still contains three major components: CTRL, REG and FUNC as in Figure 1. The difference is instead of resetting all three blocks with GR, two LR $reset_0$ and $reset_1$ are generated by NAND0 and NAND1. $reset0$ resets CTRL while $reset1$ resets REG and FUNC.

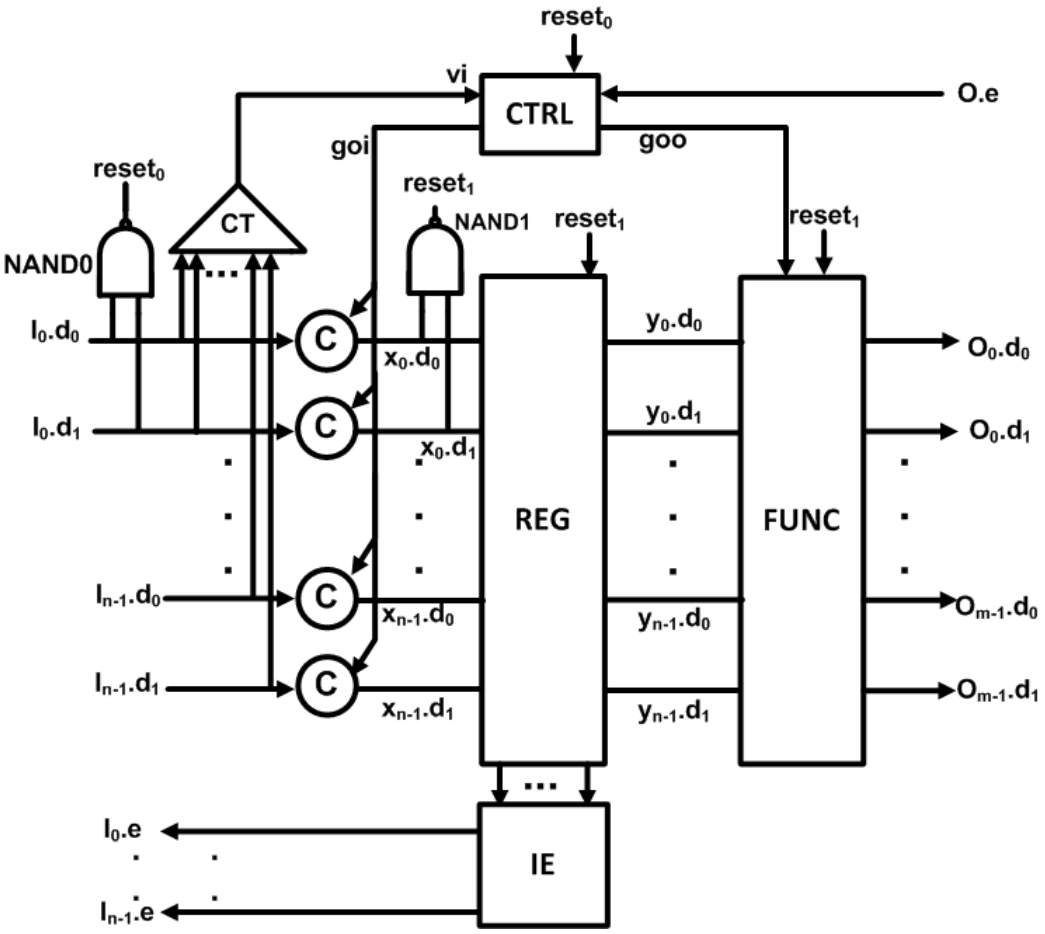

Figure 12: Pipeline Stage with Split Control and Datapath for WRS

3.2.1.1 Control Logic The circuit implementation of CTRL is shown in Figure 13. The state transition of the circuit is described by HSE in (12).

$$
\begin{aligned}
&[\neg reset_0]; [vi], [\neg O.e], x\downarrow, y\uparrow; goi_{\_}\downarrow; goi\uparrow; goo\downarrow;\\
&[reset_0]; y\downarrow; [\neg vi]; goi_{\_}\uparrow; goi\downarrow; [O.e];\\
*\,&[[vi], y\uparrow; goi_{\_}\downarrow; goi\uparrow; x\uparrow; y\downarrow;\\
&[\neg vi]; goi_{\_}\uparrow; goi\downarrow; goo\uparrow; [\neg O.e]; x\downarrow; goo\downarrow; [O.e]]
\end{aligned}
\tag{12}
$$

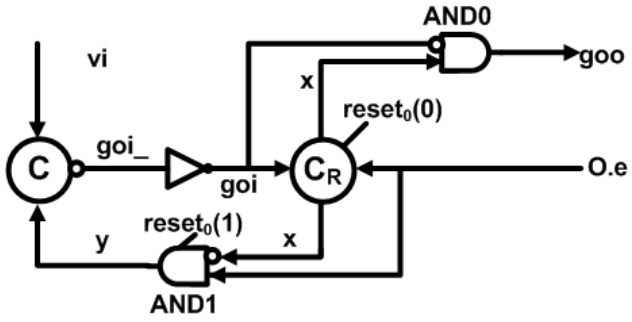

Figure 13: Control Logic for WRS

If the internal variables are removed, (12) is simplified to (13).

$$\begin{aligned} & [\neg reset_0]; [vi \wedge \neg O.e]; goi \uparrow; goo \downarrow; \\ & [reset_0]; [\neg vi \wedge O.e]; goi \downarrow; \\ * & [[vi]; goi \uparrow; [\neg vi]; goi \downarrow; goo \uparrow; [\neg O.e]; goo \downarrow; [O.e]] \end{aligned} \tag{13}$$

For the nonterminating repetition part inside $*[\ ]$, channels L and R respectively implement the expected transition in (3). Therefore, the circuit implementation is correct.

3.2.1.2 Register The implementation of REG shown in Figure 4 is not suitable for WRS. During reset phase of WRS, both $x_0.d_0$ and $x_0.d_1$ are driven to 1s and M1 and M0 are closed. Direct conducting path between VDD and GND forms through INV0 and M1 or through INV1 and M0. Therefore REG is modified as shown in Figure 14. During reset phase, $reset_1$ is 0 and both M2 and M3 are cut off. Therefore, no direct conducting path between VDD and GND is formed. During normal operation, $reset_1$ is always 1 and M2 and M3 are always conducting. If neutral data arrives, REG retains the previous data. If valid data arrives, it overwrites the data stored in REG.

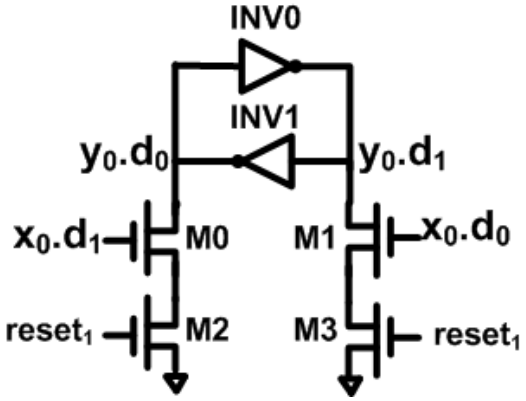

Figure 14: Register for WRS

The IE block in Figure 12 is used to generate *enable* signals for all inputs. It contains n copies of circuits shown in Figure 15. During reset phase, $x_0.d_0$ and $x_0.d_1$ are driven to 1s. $y_0.d_0$ and $y_0.d_1$ are complementary. $I_0.e$ is driven to 0 as well as all other *enable* signals for inputs. No *reset* signal needs to be inserted. During normal operation, if the input is neutral, the enable signal is driven to 1. If the input is valid and latched by the register, the enable signal is driven to 0.

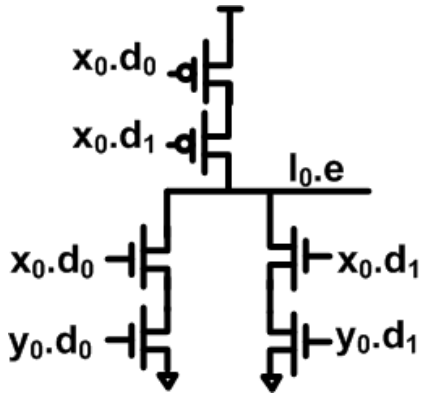

Figure 15: Input Enable Generator for WRS

3.2.1.3 Function Block FUNC contains m copies of circuit blocks shown in Figure 16 to generate m different outputs. f_0 can be any combinational logic function with inputs of *goo* and $(y_0.d_0, y_0.d_1)$ to $(y_{n-1}.d_0, y_{n-1}.d_1)$. The state transition of the circuit block is described in (14).

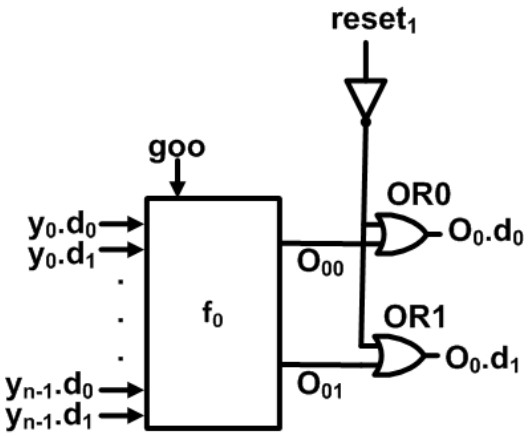

Figure 16: Function Block for WRS

$$
\begin{aligned}
&([\neg reset_1]; O_0.d_0\uparrow, O_0.d_1\uparrow),\\
&([\neg goo]; O_{00}\downarrow, O_{01}\downarrow), ([(y_0.d_0 \text{ xor } y_0.d_1) \wedge ... \wedge (y_{n-1}.d_0 \text{ xor } y_{n-1}.d_1)]);\\
&[reset_1]; O.d_0\downarrow, O.d_1\downarrow;\\
*\,&[[goo \wedge f_{01}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{00}\downarrow, O_{01}\uparrow; O_0.d_0\downarrow, O_0.d_1\uparrow\\
&[]goo \wedge f_{00}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{01}\downarrow, O_{00}\uparrow; O_0.d_1\downarrow, O_0.d_0\uparrow\\
&[\neg r]; O_{00}\downarrow, O_{01}\downarrow; O_0.d_0\downarrow, O_0.d_1\downarrow]
\end{aligned}
\tag{14}
$$

When LR $reset_1$ is asserted, reset data is generated at the output $(O.d_0\uparrow, O.d_1\uparrow)$. When $reset_1$ is deasserted and neutral data is generated at the output. After that, r assumes alternating 1 and 0 and corresponding valid and neutral data is generated at the output.

3.2.1.4 Complete Pipeline Stage The state transition of the whole pipeline stage is described in (15). When reset data arrives at the Local Reset Input input (LRI) ($[I_0.d_0 \wedge I_0.d_1]$), reset data is generated at all outputs $(O_0.d_0\uparrow, O_0.d_1\uparrow, ..., O_{m-1}.d_0\uparrow, O_{m-1}.d_1\uparrow)$. All enable signals are driven to 0s.

During normal operation, alternating valid and neutral data comes from the inputs and corresponding alternating valid and neutral data is generated at the outputs. Therefore the implementation is consistent with WRS operation protocol.

$$
\begin{aligned}
&[I_0.d_0 \wedge I_0.d_1 \wedge ... \wedge I_{n-1}.d_0 \wedge I_{n-1}.d_1]; reset_0\downarrow; vi\uparrow; goi\uparrow;\\
&goo\downarrow, x_0.d_0\uparrow, x_0.d_1\uparrow, ..., x_{n-1}.d_0\uparrow, x_{n-1}.d_1\uparrow;\\
&reset_1\downarrow; [y_0.d_1\uparrow, y_0.d_0\downarrow []y_0.d_0\uparrow, y_0.d_1\downarrow], ...,\\
&[y_{n-1}.d_1\uparrow, y_{n-1}.d_0\downarrow []y_{n-1}.d_0\uparrow, y_{n-1}.d_1\downarrow],\\
&O_0.d_0\uparrow, O_0.d_1\uparrow, ..., O_{m-1}.d_0\uparrow, O_{m-1}.d_1\uparrow;\\
&I_0.e\downarrow, ..., I_{n-1}.e\downarrow, [\neg O_0.e \wedge ... \wedge \neg O_{m-1}.e];\\
&[\neg I.d_0 \wedge \neg I.d_1 \wedge ... \wedge \neg I_{n-1}.d_0 \wedge \neg I_{n-1}.d_1];\\
&reset_0\uparrow, vi\downarrow; goi\downarrow; x_0.d_0\downarrow, x_0.d_1\downarrow, ..., x_{n-1}.d_0\downarrow, x_{n-1}.d_1\downarrow;\\
&I_0.e\uparrow, ..., I_{n-1}.e\uparrow; reset_1\uparrow;\\
&O_0.d_0\downarrow, O_0.d_1\downarrow, ..., O_{m-1}.d_0\downarrow, O_{m-1}.d_1\downarrow; [O_0.e \wedge ... \wedge O_{m-1}.e]\\
*\,&[[v(I_0) \wedge ... \wedge v(I_{n-1})]; vi\uparrow; goi\uparrow;\\
&[I_0.d_0 \rightarrow x_0.d_0\uparrow; y_0.d_0\uparrow []I_0.d_1 \rightarrow x_0.d_1\uparrow; y_0.d_1\uparrow], \qquad (15)\\
&...,\\
&[I_{n-1}.d_0 \rightarrow x_{n-1}.d_0\uparrow; y_{n-1}.d_0\uparrow []I_{n-1}.d_1 \rightarrow x_{n-1}.d_1\uparrow; y_{n-1}.d_1\uparrow];\\
&I_0.e\downarrow, ..., I_{n-1}.e\downarrow; [n(I_0) \wedge ... \wedge n(I_{n-1})]; vi\downarrow; goi\downarrow;\\
&x_0.d_0\downarrow, x_0.d_1\downarrow, ..., x_{n-1}.d_0\downarrow, x_{n-1}.d_1\downarrow; I_0.e\uparrow, ..., I_{n-1}.e\uparrow; goo\uparrow;\\
&[f_{00}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_0.d_0\uparrow\\
&[]f_{01}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_0.d_1\uparrow],\\
&...,\\
&[f_{(m-1)0}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{m-1}.d_0\uparrow\\
&[]f_{(m-1)1}(y_0.d_0, y_0.d_1, ..., y_{n-1}.d_0, y_{n-1}.d_1) \rightarrow O_{m-1}.d_1\uparrow];\\
&[\neg O_0.e \wedge ... \wedge \neg O_{m-1}.e]; goo\downarrow;\\
&O_0.d_0\downarrow, O_0.d_1\downarrow, ..., O_{m-1}.d_0\downarrow, O_{m-1}.d_1\downarrow; [O_0.e \wedge ... \wedge O_{m-1}.e]]
\end{aligned}
$$

3.2.2 Fine-Grain Integrated Pipelines for WRS

3.2.2.1 WCHB Dual-Rail Buffer for WRS The state transition of WCHB dual-rail buffer for WRS is described by HSE in (16) and The circuit is shown in Figure 17. C-element is implemented by Majority gate shown in Figure 18.

$$
\begin{array}{l}
[I.d_0 \wedge I.d_1]; reset\downarrow; O.d_0\uparrow, O.d_1\uparrow; I.e\downarrow; [\neg O.e]; \\
[\neg I.d_0 \wedge \neg I.d_1]; reset\uparrow; O.d_0\downarrow, O.d_1\downarrow; I.e\uparrow; \\
*\,[[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow \,[]I.d_1 \wedge O.e \rightarrow O.d_1\uparrow]; I.e\downarrow; \\
[\neg I.d_0 \wedge \neg I.d_1 \wedge \neg O.e]; O.d_0\downarrow, O.d_1\downarrow; I.e\uparrow]
\end{array}
\tag{16}
$$

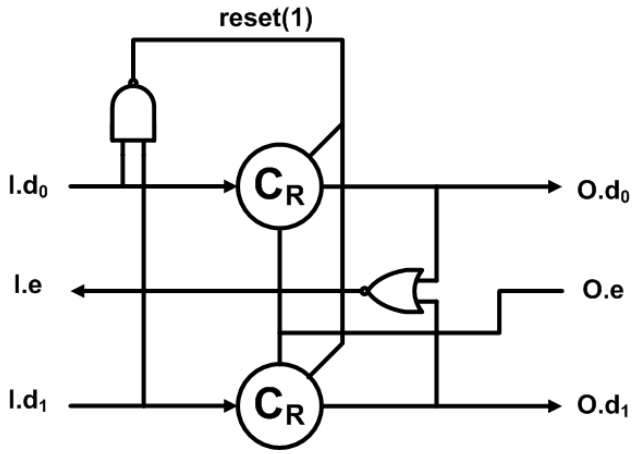

Figure 17: WCHB Dual-Rail Buffer for WRS 1

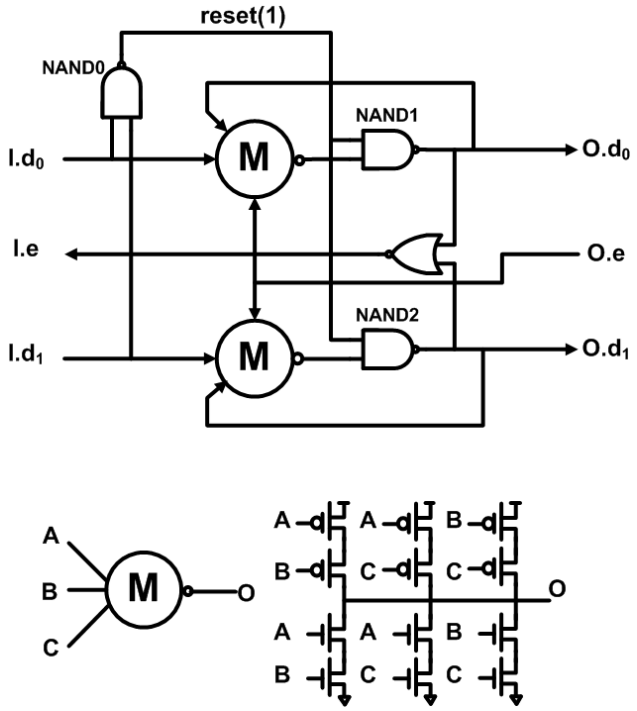

Figure 18: WCHB Dual-Rail Buffer for WRS 2

When reset data arrives ($[I.d_0 \wedge I.d_1]$), LR *reset* is driven to 0 and reset data is generated at the output ($O.d_0 \uparrow, O.d_1 \uparrow$). During normal operation, *reset* is kept at 1 and the WCHB dual-rail buffer for WRS is reduced to the standard WCHB dual-rail buffer. When alternating neutral and valid data arrives at the input, it is propagated to the output. The implementation is consistent the operation protocol of WRS.

3.2.2.2 WCHB 1-of-4 Buffer for WRS The state transition of WCHB 1-of-4 buffer for WRS is described by HSE in (17) and The circuit is shown in Figure 19. PCHB and PCFB 1-of-4 buffers with WRS can be implemented similarly.

$$
\begin{array}{l}
[I.d_0 \wedge I.d_1]; reset\downarrow; O.d_0\uparrow, O.d_1\uparrow, O.d_2\downarrow, O.d_3\downarrow; \\
I.e\downarrow; [\neg O.e]; \\
[\neg I.d_0 \wedge \neg I.d_1 \wedge \neg I.d_2 \wedge \neg I.d_3]; reset\uparrow; \\
O.d_0\downarrow, O.d_1\downarrow, O.d_2\downarrow, O.d_3\downarrow; I.e\uparrow; \\
*\,[[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow \,[]\, I.d_1 \wedge O.e \rightarrow O.d_1\uparrow \\
\;\; []I.d_2 \wedge O.e \rightarrow O.d_2\uparrow \,[]\, I.d_3 \wedge O.e \rightarrow O.d_3\uparrow]; I.e\downarrow; \\
\;\; [\neg I.d_0 \wedge \neg I.d_1 \wedge \neg I.d_2 \wedge \neg I.d_3 \wedge \neg O.e]; \\
\;\; O.d_0\downarrow, O.d_1\downarrow, O.d_2\downarrow, O.d_3\downarrow; I.e\uparrow]
\end{array}
\tag{17}
$$

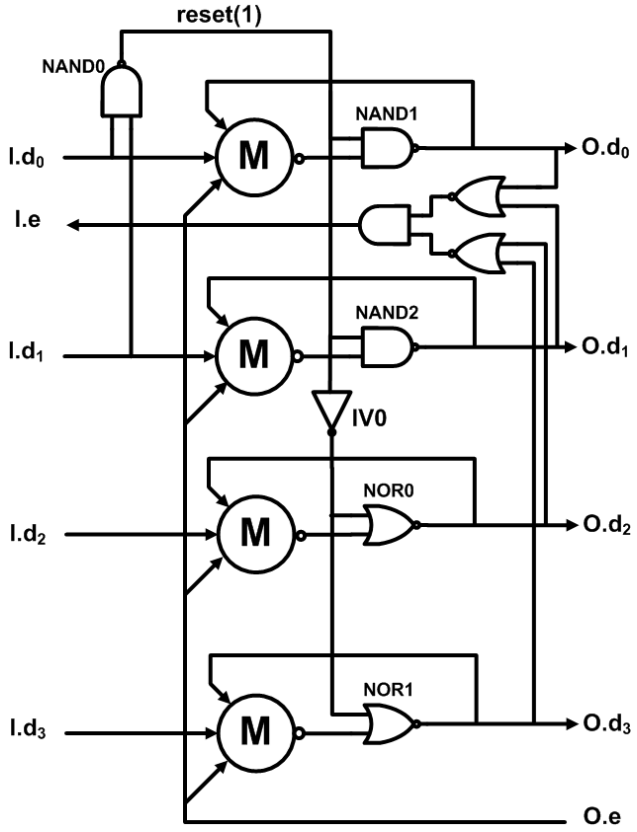

Figure 19: WCHB 1-of-4 Buffer for WRS

As mentioned, reset value of 1-of-4 data codes is $(I.d_0, I.d_1, I.d_2, I.d_3)$ = (1,1,0,0). Therefore, only $I.d_0$ and $I.d_1$ need to be checked in order to determine whether the input data is reset data. As n in the 1-of-n codes

increases, the overhead of LRG remains the same. This is why reset value is chosen as 2 wires being 1s while the rest wires being 0s.

3.2.2.3 PCHB Dual-Rail Buffers The state transition of PCHB dual-rail buffer for WRS is described by HSE in (18) and The circuit is shown in Figure 20.

$$\begin{aligned} & [I.d_0 \wedge I.d_1]; reset\downarrow; O.d_0\uparrow, O.d_1\uparrow; I.e\downarrow; [\neg O.e]; \\ & [\neg I.d_0 \wedge \neg I.d_1]; reset\uparrow; O.d_0\downarrow, O.d_1\downarrow; I.e\uparrow; \\ * & [[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow [] I.d_1 \wedge O.e \rightarrow O.d_1\uparrow]; I.e\downarrow; \\ & [\neg O.e]; O.d_0\downarrow, O.d_1\downarrow; [\neg I.d_0 \wedge \neg I.d_1]; I.e\uparrow] \end{aligned} \tag{18}$$

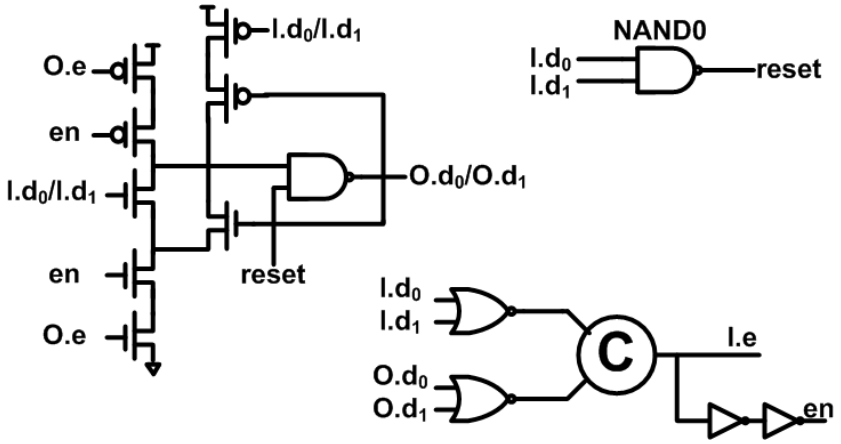

Figure 20: PCHB Dual-Rail Buffer for WRS

When reset data arrives ($[I.d_0 \wedge I.d_1]$), *reset* is driven to 0 and reset data is generated at the output ($O.d_0\uparrow, O.d_1\uparrow$). During normal operation, *reset* is kept at 1 and the PCHB dual-rail buffer with WRS can be reduced to the standard PCHB dual-rail buffer. When alternating neutral and valid data arrives at the input, it is propagated to the output. The implementation follows the operation protocol of WRS.

3.2.2.4 PCFB Dual-Rail Buffers The state transition of PCFB dual-rail buffer with WRS is described by HSE in (19) and The circuit is shown in Figure 21.

$$
\begin{aligned}
&[I.d_0 \wedge I.d_1]; reset\downarrow; x\uparrow, O.d_0\uparrow, O.d_1\uparrow; I.e\downarrow; [\neg O.e];\\
&[\neg I.d_0 \wedge \neg I.d_1]; reset\uparrow; x\downarrow; I.e\uparrow, O.d_0\downarrow, O.d_1\downarrow; x\uparrow;\\
*\,&[[I.d_0 \wedge O.e \rightarrow O.d_0\uparrow\, []I.d_1 \wedge O.e \rightarrow O.d_1\uparrow]; I.e\downarrow; x\downarrow;\\
&([\neg O.e]; O.d_0\downarrow, O.d_1\downarrow), ([\neg I.d_0 \wedge \neg I.d_1]; I.e\uparrow)]
\end{aligned}
\tag{19}
$$

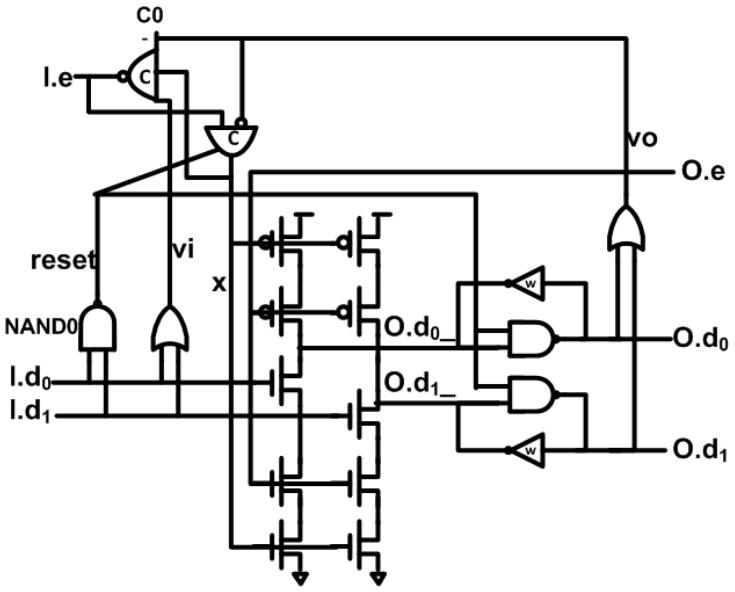

Figure 21: PCFB Dual-Rail Buffer for WRS

When reset data arrives ($[I.d_0 \wedge I.d_1]$), *reset* is driven to 0. Reset data is generated at the output ($O.d_0\uparrow, O.d_1\uparrow$) and the internal variable x is driven to 1. During normal operation, *reset* is kept at 1 and the PCFB dual-rail buffer with WRS can be reduced to the standard PCFB dual-rail buffer. When alternating neutral and valid data arrives at the input, it is propagated to the output. The implementation follows the operation protocol of WRS.

3.2.2.5 Buffers with Logic Logic function can be embedded into buffer templates. In this section, PCHB templates with logic are used for demonstration. WCHB and PCFB templates with logic can be implemented similarly. The templates with logic are comprised of two cases: processes with unconditional inputs/outputs and processes with conditional inputs/outputs. Processes with unconditional inputs/outputs receive inputs and generate outputs in each iteration. However, for processes with conditional inputs/out-

puts, depending on values of some inputs, they may or may not receive other inputs or generate outputs.

Processes with unconditional inputs/outputs can be described by CHP in (20), where $I_0, I_1, ..., I_{n-1}$ are n inputs, $O_0, O_1, ..., O_{m-1}$ are m outputs, X is the set of all variables $\{x_0, x_1, ...x_{n-1}\}$, $f_0, f_1, ..., f_{m-1}$ are functions to generate $O_0, O_1, ..., O_{m-1}$. In each iteration, processes with unconditional inputs/outputs always receive data from all inputs, apply functions to the data and send results through outputs.

$$*[I_0?x_0, I_1?x_1, ..., I_{n-1}?x_{n-1}; O_0!f_0(X), O_1!f_1(X), ..., O_{m-1}!f_{m-1}(X)] \quad (20)$$

It is assumed I_0 is LRI. The choice of LRI will be discussed in Chapter 4. All inputs/outputs are assumed to be dual-rail encoded. (Processes with other 1-of-n data encoding can be similarly implemented). The process with unconditional inputs/outputs is implemented with PCHB templates in (21).

$$\begin{aligned}
i &\in [0..n-1] \\
j &\in [0..m-1] \\
I_0.d_0 \wedge I_0.d_1 &\rightarrow reset\downarrow \\
\neg I_0.d_0 \vee \neg I_0.d_1 &\rightarrow reset\uparrow \\
reset \wedge \neg O_j.e \wedge \neg en_j &\rightarrow O_j.d_0\downarrow \\
reset \wedge \neg O_j.e \wedge \neg en_j &\rightarrow O_j.d_1\downarrow \\
\neg reset \vee (O_j.e \wedge en_j \wedge f_{j0}(\{I_i.d_0, I_i.d_1\}) &\rightarrow O_j.d_0\uparrow \\
\neg reset \vee (O_j.e \wedge en_j \wedge f_{j1}(\{I_i.d_0, I_i.d_1)\} &\rightarrow O_j.d_1\uparrow \\
\neg I_i.d_0 \wedge \neg I_i.d_1 &\rightarrow vI_i\downarrow \\
I_i.d_0 \vee I_i.d_1 &\rightarrow vI_i\uparrow \\
\neg O_j.d_0 \wedge \neg O_j.d_1 &\rightarrow vO_j\downarrow \\
O_j.d_0 \vee O_j.d_1 &\rightarrow vO_j\uparrow \\
vI_i \wedge (\{\wedge_{h\in[0,m-1]|O_h}\text{depends on }{}_{I_i}vO_h\}) &\rightarrow I_i.e\downarrow \\
\neg vI_i \wedge (\{\wedge_{h\in[0,m-1]|O_h}\text{depends on }{}_{I_i}\neg vO_h\}) &\rightarrow I_i.e\uparrow \\
vO_j \wedge (\{\wedge_{k\in[0,n-1]|O_j}\text{depends on }{}_{I_k}vI_k\}) &\rightarrow en_j\downarrow \\
\neg vO_j \wedge (\{\wedge_{k\in[0,n-1]|O_j}\text{depends on }{}_{I_k}\neg vI_k\}) &\rightarrow en_j\uparrow
\end{aligned} \quad (21)$$

The variables vI_i and vO_j refer to the validity of input I_i and output O_j. For example, when input I_0 has reset or valid data ($I_0.d_0 \vee I_0.d_1 = 1$), $vI_0 = 1$. When input I_0 has neutral data ($I_0.d_0 = I_0.d_1 = 0$), $vI_0 = 0$.

$f_{j0}(\{I_i.d_0, I_i.d_1\})$ and $f_{j1}(\{I_i.d_0, I_i.d_1\})$ respectively generate valid output $O_j.d_0$ and $O_j.d_1$ based on subset of inputs. In order to acknowledge I_i ($I_i.e\downarrow/I_i.e\uparrow$), the validity of the input itself as well as all the outputs that depend on the input must be 1/0. For example, if O_0, O_2 and O_3 are three outputs that depend on I_0, $I_0.e$ is generated as shown in (22).

$$\begin{aligned} vI_0 \wedge vO_0 \wedge vO_2 \wedge vO_3 &\rightarrow I_0.e\downarrow \\ \neg vI_0 \wedge \neg vO_0 \wedge \neg vO_2 \wedge \neg vO_3 &\rightarrow I_0.e\uparrow \end{aligned} \tag{22}$$

Similarly, in order to generate enable signal en_j ($en_j-/en_j\uparrow$), the validity of the output as well as all the inputs that the output is dependent on must be 1/0.

Processes with conditional inputs/outputs can be described in (23). There are n condition inputs C_i, k conditions that are function of C_i, m data inputs I_j and p outputs O_h. The process receives condition inputs in each iteration. Based on values of condition inputs, it determines which condition among $cond_0, cond_1, ..., cond_{k-1}$ is $true$. Based on the $true$ condition, it selectively receives data from some inputs, computes functions on the received data and sends results through some outputs.

$$\begin{aligned} &*\,[\{C_i?c_i|_{i\in[0,n-1]},\};\\ &\quad [cond_0 \rightarrow \{I_{j_0}?d_{j_0}|_{j_0\in[0,m-1]},\};\{O_{h_0}!f_{h_0}|_{h_0\in[0,p-1]},\};\\ &\quad []cond_1 \rightarrow \{I_{j_1}?d_{j_1}|_{j_1\in[0,m-1]},\};\{O_{h_1}!f_{h_1}|_{h_1\in[0,p-1]},\};\\ &\quad .\\ &\quad .\\ &\quad .\\ &\quad []cond_{k-1} \rightarrow \{I_{j_{k-1}}?d_{j_{k-1}}|_{j_{k-1}\in[0,m-1]},\};\{O_{h_{k-1}}!f_{h_{k-1}}|_{h_{k-1}\in[0,p-1]},\};\\ &\quad]] \end{aligned} \tag{23}$$

Processes with conditional inputs/outputs can be implemented as shown in (24). It is assumed I_0 is LRI. $cond_u$ is a function $g_u()$ of condition inputs. For example, if there are two condition inputs and $g_u()$ is logic AND, $g_u(\{c_0, c_1\}) = c_0 \wedge c_1$. The variables vI_j and vO_h refer to the validity of input I_j and output O_h. $f_{h_f}(\{cd_u, I_j.d_0, I_j.d_1\})$ and $f_{h_t}(\{cd_u, I_j.d_0, I_j.d_1\})$ respectively generate outputs $O_h.d_0$ and $O_h.d_1$ based on conditions and subset of inputs. In order to acknowledge I_j ($I_j.e\downarrow/I_j.e\uparrow$), the conditions inside which I_j exists are first determined. If in any condition, the condition, validity of I_j and validity of all outputs that depend on I_j are 1s, $I_j.e$ is driven to 0. When all conditions, validity of I_j and validity of outputs that depend on I_j are 0s,

$I_j.e$ is driven to 1. Similarly for en_h, if in any condition where O_h exists, the condition, validity of O_h and validity of all inputs that O_h depends on are 1s, en_h is driven to 0. If for all conditions where O_h exists, the conditions, validity of O_h and validity of inputs that O_h depends on are 0s, en_h is driven to 1.

For both processes with conditional/unconditional inputs/outputs, during reset phase, reset data arrives at different inputs at different time. When reset data arrives at inputs other than LRI I_0, garbage data may be generated at outputs. However, once reset data arrives at I_0, LR $reset$ is driven to 0 and reset data is generated at all outputs. The transition from garbage data to reset data is monotonic during reset phase; that is, the garbage data at the outputs will be overwritten by reset data and reset data will remain through the whole reset phase. Therefore, given enough time, reset data can traverse the whole system and every process in the system will generate reset data at its output. All validity signals are driven to 1s and all $I.e$ and en are driven to 0s.

GR is then deasserted and IP starts to generate neutral data. Like reset data, neutral data arrives at different inputs at different time. If neutral data hasn't arrived at I_0, reset data remains at the outputs no matter whether neutral data has arrived at other inputs. When neutral data arrives at I_0, $reset$ is driven to 1. At this moment, $O.e$ and en are still 0s because outputs haven't changed from reset data yet. Therefore all the outputs will generate neutral data. If some inputs still have reset data, they will not accidentally generate wrong data at the output. It is because when an input I has reset data, vI is 1. $I.e$ and all en that depends on vI are still 0s. Neutral data remains at the relevant outputs. Given enough time, all reset data will be overwritten by neutral data. LRG keeps LR at 0 and no reset data will be generated any more. Processes operate normally with alternating neutral and valid data.

$$
\begin{aligned}
i &\in [0, n-1] \\
j &\in [0, m-1] \\
h &\in [0, p-1] \\
u &\in [0, k-1] \\
I_0.d_0 \wedge I_0.d_1 &\rightarrow reset\downarrow \\
\neg I_0.d_0 \vee \neg I_0.d_1 &\rightarrow reset\uparrow \\
g_u(\{c_i\}) &\rightarrow cond_u\uparrow \\
\neg cond_u &\rightarrow cond_u\downarrow \\
\neg I_j.d_0 \wedge \neg I_j.d_1 &\rightarrow vI_j\downarrow \\
I_j.d_0 \vee I_j.d_1 &\rightarrow vI_j\uparrow \\
\neg O_h.d_0 \wedge \neg O_h.d_1 &\rightarrow vO_h\downarrow \\
O_h.d_0 \vee O_h.d_1 &\rightarrow vO_h\uparrow \\
\{\vee_{u|I_j \textit{ in } cond_u}(cond_u \wedge vI_j \wedge (\{\wedge_{h\in[0,m-1]|O_{h_u} \textit{ depends on } I_j} vO_{h_u}\}))\} &\rightarrow I_j.e\downarrow \\
\{\wedge_{u|I_j \textit{ in } cond_u}(\neg cond_u \wedge \neg vI_j \wedge (\{\wedge_{h\in[0,m-1]|O_{h_u} \textit{ depends on } I_j} \neg vO_{h_u}\}))\} &\rightarrow I_j.e\uparrow \\
\{\vee_{u|O_h \textit{ in } cond_u}(cond_u \wedge vO_h \wedge (\{\wedge_{k\in[0,n-1]|O_h \textit{ depends on } I_{j_u}} vI_{j_u}\}))\} &\rightarrow en_h\downarrow \\
\{\wedge_{u|O_h \textit{ in } cond_u}(\neg cond_u \wedge \neg vO_h \wedge (\{\wedge_{k\in[0,n-1]|O_h \textit{ depends on } I_{j_u}} \neg vI_{j_u}\}))\} &\rightarrow en_h\uparrow \\
reset \wedge \neg O_h.e \wedge \neg en_h &\rightarrow O_h.d_0\downarrow \\
reset \wedge \neg O_h.e \wedge \neg en_h &\rightarrow O_h.d_1\downarrow \\
\neg reset \vee (O_h.e \wedge en_h \wedge f_{h_f}(\{cd_u, I_j.d_0, I_j.d_1\}) &\rightarrow O_h.d_0\uparrow \\
\neg reset \vee (O_h.e \wedge en_h \wedge f_{h_t}(\{cd_u, I_j.d_0, I_j.d_1\}) &\rightarrow O_h.d_1\uparrow
\end{aligned}
\tag{24}
$$

3.3 Special Blocks

Besides processes described in (2), there are other special processes with different CHP description and must be implemented separately.

3.3.1 Source/Sink

Source is described in (22). It constantly sends *true* or $false$ data through outputs. Since there is no input, LR can't be generated by LRG. Source needs a GR input directly.

$$*[O!true/false] \tag{25}$$

The circuit of Source $*[O!true]$ is shown in Figure 22. ($*[O!false]$ can be implemented similarly) During reset phase, $reset$ is 0. Reset data is generated at the output. During normal operation, $reset$ is driven to 1. Alternating neutral and valid *true* data is generated at the output.

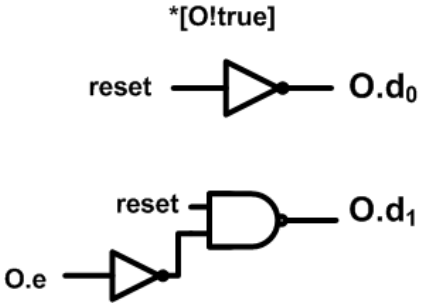

Figure 22: Source for WRS

Sink is described in (23). It keeps receiving data from inputs. For the acknowledgement signal $I.e$, when reset or valid data arrives at the input, it should be driven to 0. When neutral data arrives at the input, it should be driven to 1. Therefore it can be simply implemented as a NOR gate shown in Figure 23. No LR needs to be generated.

$$*[I] \tag{26}$$

Figure 23: Sink for WRS

3.3.2 Initial Buffer

Instead of waiting for data to arrive at the input first, Initial Buffer (IB) starts operation by sending stored data as shown in (27). x can be 1 or 0 initially.

$$*[O!x; I?x] \tag{27}$$

The implementation of IB is shown in Figure 24. Each C-element has GR (*reset*) as input. Once *reset* is asserted, outputs of C-elements are driven to appropriate values. As mentioned processes that are not controlled by GR will receive reset data followed by alternating neutral and valid data. This pattern is implemented by IB. During reset phase, $(I_3.d_0, I_3.d_1) = (0, 0)$ while $(O.d_0, O.d_1) = (1, 1)$. Initial data is stored in $(I_1.d_0, I_1.d_1)$ and $(I_2.d_0, I_2.d_1)$. If the stored data is 1, $(I_1.d_0, I_1.d_1) = (I_2.d_0, I_2.d_1) = (0, 1)$. Otherwise $(I_1.d_0, I_1.d_1) = (I_2.d_0, I_2.d_1) = (1, 0)$. Therefore, the first three data coming out of IB is reset data, neutral data and valid data. After that, IB operates like a normal buffer: receiving alternating neutral and valid data from input and generating alternating neutral and valid data at the output.

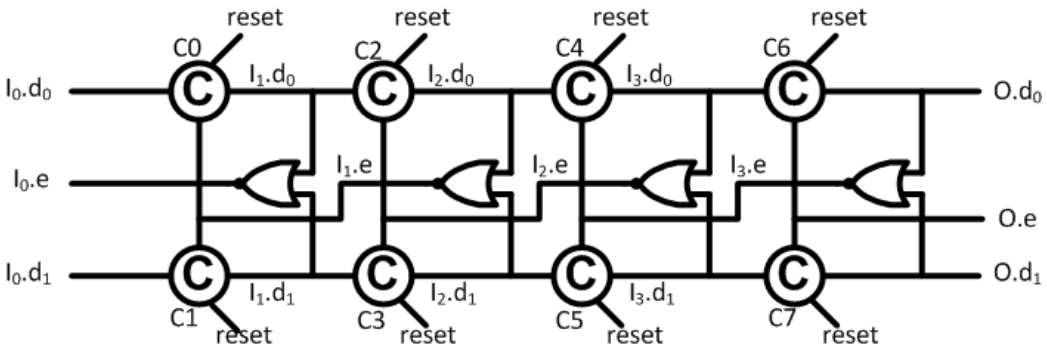

Figure 24: Initial Buffer for WRS

The duplicate copy of stored data is necessary to prevent reset data $(I_0.d_0, I_0.d_1) = (1, 1)$ from overwriting the stored data in IB. During reset phase, reset data propagates through the whole system and will arrive at $(I_0.d_0, I_0.d_1)$. If there is no $(I_2.d_0, I_2.d_1)$ but just $(I_1.d_0, I_1.d_1)$ to store the data, $I_2.e$ would be directly connected to C0 and C1. Since $(I_3.d_0, I_3.d_1) = (0, 0)$ during reset phase, $I_2.e = 1$. When *reset* is deasserted, C0 and C1 may fire first and the stored data is overwritten. However, with the duplicate copy of token, $I_1.e$ is 0 and it prevents reset data from propagating through C0 and C1. Only after $(I_0.d_0, I_0.d_1)$ is overwritten with neutral data, it can

overwrite $(I_1.d_0, I_1.d_1)$. At that time, the data is kept at $(I_2.d_0, I_2.d_1)$. The neutral data at $(I_1.d_0, I_1.d_1)$ will not overwrite $(I_2.d_0, I_2.d_1)$ until valid data at $(I_2.d_0, I_2.d_1)$ propagates to $(I_3.d_0, I_3.d_1)$. Therefore, reset data starting from IB will stop at the input of IB and be overwritten by neutral data. When all reset data has been overwritten, the system correctly transits into normal operation with only neutral and valid data propagating inside.

3.3.3 Channel Arbiter

Channel arbiter described in (28) is used to arbitrate between two nonmutually exclusive input channels. Inputs I_0 and I_1 are nonmutually exclusive while outputs O_0 and O_1 are mutually exclusive. It is assumed both channels are encoded in dual rail. The thin bar "|" on the third row indicates that I_0 and I_1 are nonmutually exclusive.

$$\begin{array}{l} *\,[[I_0.d_0 \rightarrow O_0.d_0\uparrow \,[]I_0.d_1 \rightarrow O_0.d_1\uparrow];[\neg O_0.e];I_0.e\downarrow; \\ \quad [\neg I_0.d_0 \wedge \neg I_0.d_1];O_0.d_0\downarrow, O_0.d_1\downarrow;[O_0.e];I_0.e\uparrow \\ |\quad [I_1.d_0 \rightarrow O_1.d_0\uparrow \,[]I_1.d_1 \rightarrow O_1.d_1\uparrow];[\neg O_1.e];I_1.e\downarrow; \\ \quad [\neg I_1.d_0 \wedge \neg I_1.d_1];O_1.d_0\downarrow, O_1.d_1\downarrow;[O_1.e];I_1.e\uparrow] \end{array} \tag{28}$$

The channel arbiter is implemented in Figure 25 [1]. "arb" in the figure is a basic arbiter implemented in Figure 26. It guarantees that its outputs are not driven to 1s at the same time.

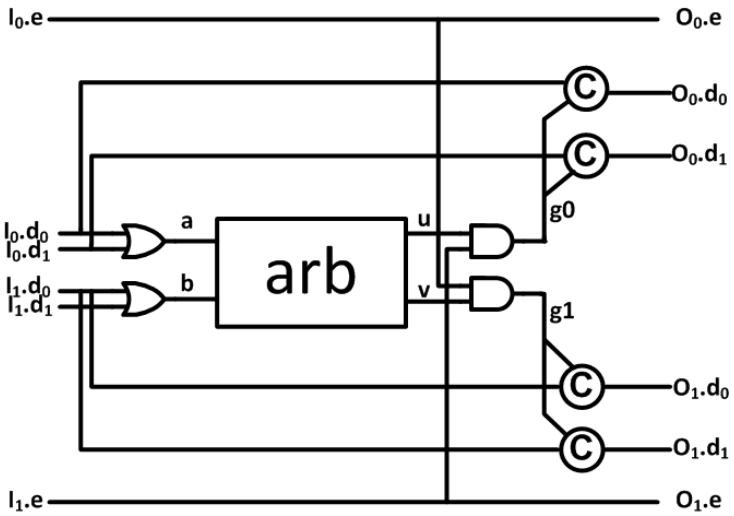

Figure 25: Channel Arbiter

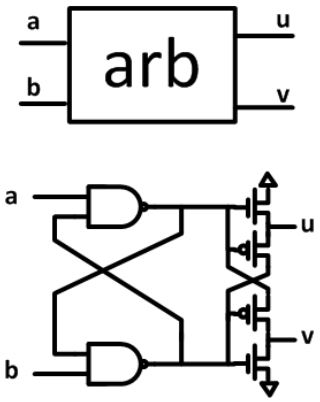

Figure 26: Basic Arbiter

As shown in Figure 25, when I_0 or I_1 but not both has valid data or both I_0 and I_1 have neutral data, the valid or neutral data will propagate to the corresponding output. when both I_0 and I_1 have valid data, a and b are driven to 1s. The basic arbiter will nondeterministically drive (u, v) to $(1, 0)$ or $(0, 1)$ which enables the valid data from I_0 to propagate to O_0 or valid data from I_1 to propagate to O_1.

In order to apply WRS to the channel arbiter, LRG NAND0 is added as shown in Figure 27. It is assumed I_0 is LRI. Once reset data arrives at I_0, *reset* is driven to 0 and reset data is generated at outputs. When neutral data arrives at I_0 and I_1 after GR is deasserted, *reset* is driven to 1 and neutral data will propagate to O_0 and O_1. After that, the channel arbiter starts its normal operation.

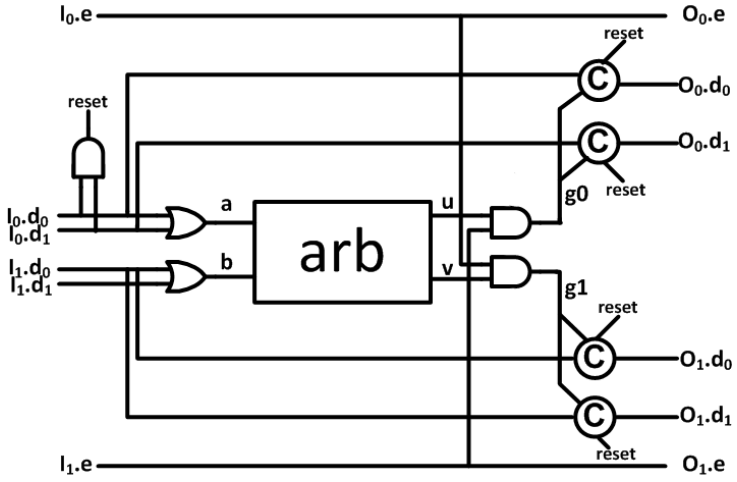

Figure 27: Channel Arbiter for WRS

3.3.4 Slack Zero Process

The static slack of a pipeline is maximum number of messages that can be inserted into the pipeline, with none being removed [5]. PCFB are full buffers and have slack 1. WCHB and PCHB are half buffers and have slack 1/2. WCHB and PCHB are called half buffers because two of them connected together can form a full buffer that has slack 1. There are also slack-zero processes that has slack 0. Therefore they can't hold any message in them. For example, a merge process is described in (29). If control input c is *true*, the merge process receives data from $I1$ and sends data through O. Otherwise, the merge process receives data from $I0$ and sends data through O.

$$*[C?c; [c \rightarrow I1?x; O!x[]\neg c \rightarrow I0?x; O!x]] \tag{29}$$

If the merge process is implemented with PCHB templates, it is shown in (30).

$$
\begin{aligned}
I_0.d_0 \vee I_0.d_1 &\rightarrow vI_0 \uparrow \\
\neg I_0.d_0 \wedge \neg I_0.d_1 &\rightarrow vI_0 \downarrow \\
I_1.d_0 \vee I_1.d_1 &\rightarrow vI_1 \uparrow \\
\neg I_1.d_0 \wedge \neg I_1.d_1 &\rightarrow vI_1 \downarrow \\
O.d_0 \vee O.d_1 &\rightarrow O \uparrow \\
\neg O.d_0 \wedge \neg O.d_1 &\rightarrow O \downarrow \\
vO \wedge C.d_0 \wedge vI_0 &\rightarrow I_0.e \downarrow \\
\neg vO \wedge \neg C.d_0 \wedge \neg vI_0 &\rightarrow I_0.e \uparrow \\
vO \wedge C.d_1 \wedge vI_1 &\rightarrow I_1.e \downarrow \\
\neg vO \wedge \neg C.d_1 \wedge \neg vI_1 &\rightarrow I_1.e \uparrow \\
I_0.e \wedge I_1.e &\rightarrow C.e \uparrow \\
\neg I_0.e \vee \neg I_1.e &\rightarrow C.e \downarrow \\
C.e \wedge O.e &\rightarrow en \uparrow \\
\neg C.e \wedge \neg O.e &\rightarrow en \downarrow \\
en \wedge ((C.d_0 \wedge I_0.d_0) \vee (C.d_1 \wedge I_1.d_0)) &\rightarrow O.d_0 \uparrow \\
\neg en &\rightarrow O.d_0 \downarrow \\
en \wedge ((C.d_0 \wedge I_0.d_1) \vee (C.d_1 \wedge I_1.d_1)) &\rightarrow O.d_1 \uparrow \\
\neg en &\rightarrow O.d_1 \downarrow
\end{aligned}
\tag{30}
$$

If it is implemented with slack-zero processes, it is shown in (31).

$$
\begin{aligned}
I_0.d_0 \wedge C.d_0 &\rightarrow i00c0 \uparrow \\
\neg I_0.d_0 \wedge \neg C.d_0 &\rightarrow i00c0 \downarrow \\
I_1.d_0 \wedge C.d_1 &\rightarrow i10c1 \uparrow \\
\neg I_1.d_0 \wedge \neg C.d_1 &\rightarrow i10c1 \downarrow \\
I_0.d_1 \wedge C.d_0 &\rightarrow i01c0 \uparrow \\
\neg I_0.d_1 \wedge \neg C.d_0 &\rightarrow i01c0 \downarrow \\
I_1.d_1 \wedge C.d_1 &\rightarrow i11c1 \uparrow \\
\neg I_1.d_1 \wedge \neg C.d_1 &\rightarrow i11c1 \downarrow \\
i00c0 \vee i10c1 &\rightarrow O.d_0 \uparrow \\
\neg i00c0 \wedge \neg i10c1 &\rightarrow O.d_0 \downarrow \\
i01c0 \vee i11c1 &\rightarrow O.d_1 \uparrow \\
\neg i01c0 \wedge \neg i11c1 &\rightarrow O.d_1 \downarrow \\
O.e \wedge \neg C.d_0 &\rightarrow I_0.e \uparrow \\
\neg O.e \wedge C.d_0 &\rightarrow I_0.e \downarrow \\
O.e \wedge \neg C.d_1 &\rightarrow I_1.e \uparrow \\
\neg O.e \wedge C.d_1 &\rightarrow I_1.e \downarrow \\
I_0.e \wedge I_1.e &\rightarrow C.e \uparrow \\
\neg I_0.e \vee \neg I_1.e &\rightarrow C.e \downarrow
\end{aligned}
\tag{31}
$$

Slack-zero processes don't introduce any sequence. Whatever arrives at the input will go through some logic function and the result of which will be output. True rail and false rail are separately driven to 1 when subset of input rails are 1s. Therefore, when all inputs have reset data, i.e. all input rails are 1s, all output rails are driven to 1s. No LR needs to be generated for slack-zero processes.

4 Global Reset Insertion

As mentioned, the Global Reset (GR) is connected to Initial Processes (IP). IP will output reset data once GR is asserted. There must be enough IP so that reset data can reach all processes in the system.

For a system with GRS, once GR is asserted, all channel wires between different processes are driven to 0s. When GR is deasserted, Non-Initial Processes (NIP) wait for valid data. Only IP such as initial buffers and sources start by sending valid data. Since QDI systems are working with GRS, the generated valid data from initial buffers and sources must be able to reach all processes in the system. Otherwise, part of the system stays idle and doesn't function at all.

When the system is modified from GRS to WRS, the number of processes don't change, neither do the connections among different processes. Therefore, reset data generated from initial buffers and sources must be able to reach all processes in the system with WRS. It is sufficient to have initial buffers and sources as IP.

Although reset data can always reach all processes in the system with WRS, reset time can vary depending on the choice of Local Reset Input (LRI). If reset data arrives at LRI of a process late, reset data will be generated late at the output of the process which will affect the resetting of the next process. In the end reset time is large. Therefore, LRI should be chosen as the first input of a process that gets reset data during reset phase.

A systematic approach of finding LRI is demonstrated in this chapter. First, the system is modelled as a directed graph $G \equiv (V, E)$. V is a set of vertices while E is a set of edges. Each vertex $v \in V$ represents a process while each edge as an ordered pair $e \equiv (v_i, v_j) \in E$ represents a channel connecting process v_i to process v_j. If there is an edge $(v_i, v_j) \in E$, vertex v_j is adjacent to vertex v_i.

Each edge connects exactly two vertices. Fork is implemented inside vertices. For example, if one output O_u from process u needs to connect to inputs I_v and I_w of two different processes v and w, instead of connecting the same output O_u to both I_v and I_w, two identical copies of outputs O_{u_0} and O_{u_1} will be generated from u and respectively connect to I_v and I_w. In addition, there is no edge starting from and ending with the same vertex.

A path from vertex u to vertex v is a sequence of edges starting from u and ending with v. If there is a path from u to v, v is reachable from u. A graph is connected if it contains a path from u to v or a path from v to u for any pair of vertices u and v. A graph is strongly connected if it contains a path from u to v and a path from v to u for any pair of vertices u and v. G

in the QDI systems is connected but not necessarily strongly connected.

4.1 Breadth First Search (BFS)

As introduced in [4], Breadth First Search (BFS) systematically explores edges of G to discover every vertex that is reachable from the root vertex v. It also generates a Breadth-First Tree (BFT) whose root is v. The tree contains all reachable vertices from v.

The pseudocode of BFS is shown in Figure 28. The algorithm has G and v as inputs. Each vertex in G has two attributes, color and parent. Color is used to distinguish different states of a vertex. Initially all vertices are white. When a vertex is traversed the first time, it becomes gray. If all adjacent vertices of a gray vertex have been traversed, it becomes black. When a vertex v is traversed the first time in the course of scanning the adjacent vertices of an already traversed vertex u, u is the parent of v and v is the child of u.

$BFS(G, v)$ works as follows. Line 2 creates an empty queue Q which is used to store the first-time traversed vertices. $Q.enqueue(v)$ inserts vertex v into Q while $Q.dequeue()$ removes the first element in Q and returns it. Line 3 creates an empty tree T. $T.add(u, v)$ inserts v into T as the child of u. After the execution of the algorithm, T is BFT. Line 4-6 mark all vertices white and set their parent NIL. The root vertex v is first marked gray and added into T and Q as in Line 7-9. The while loop in Line 10-18 iterates as long as Q is not empty. During each iteration, the first element n is removed from Q. If any of its adjacent vertices s is white which indicates s hasn't been traversed, s will be marked gray and s's parent is set to n. s is then inserted into T and Q. When all its adjacent vertices have been traversed, n is marked black.

4.2 Breadth First Search with Multiple Roots (BFSMR)

In order to find LRI in a QDI system with WRS, BFS is run to generate a BFT. The edges in the BFT are LRI of child vertices. Normally, there are more than one IP in the QDI system with WRS, BFS is modified to BFSMR that starts the search with multiple roots. Correspondingly, instead of BFT, Breadth-First Forest (BFF) is generated. Edges in BFF are LRI of child vertices. All processes represented by the roots are directly controlled by GR.

```
1  procedure BFS(G, v)
2    create an empty queue Q
3    create an empty tree T
4    for each vertex u ∈ V(G)
5      u.color = white
6      u.parent = NIL
7    v.color = gray
8    T.add(null, v)
9    Q.enqueue(v)
10   while Q is not empty
11     n = Q.dequeue()
12     for each s ∈ n.adjacentvertices
13       if s.color = white
14         s.color = gray
15         s.parent = n
16         T.add(n, v)
17         Q.enqueue(s)
18     n.color = black
```

Figure 28: Pseudocode of Breadth First Search (BFS)

4.2.1 Pseudocode

The pseudocode of BFSMR is shown in Figure 29. Only Line 7-10 are different from Figure 28. Instead of inserting single root into T and Q initially, all vertices that represent IP are inserted.

4.2.2 Proof of Correctness

The correctness of the pseudocode is proved with loop invariant: Q contains nothing or all gray vertices.

4.2.2.1 Initialization Before the iteration of the while loop, Q contains all gray vertices that represent IP in the system as shown in Line 7-10. Processes other than IP in the system are marked white in Line 4-6. The loop invariant is true.

4.2.2.2 Maintenance Line 12 removes vertex n from Q. During the same iteration of the while loop, Line 19 marks n black. If Q originally

```
1  procedure BFSMR(G, V)
2    create an empty queue Q
3    create an empty tree T
4    for each vertex u ∈ V(G)
5      u.color = white
6      u.parent = NIL
7    for each vertex that represents IP
8      v.color = gray
9      T.add(NIL, v)
10     Q.enqueue(v)
11   while Q is not empty
12     n = Q.dequeue()
13     for each s ∈ n.adjacentvertices
14       if s.color = white
15         s.color = gray
16         s.parent = n
17         T.add(n, v)
18         Q.enqueue(s)
19     n.color = black
```

Figure 29: Pseudocode of BFS with Multiple Roots

contains all gray vertices, after this removal of n which is marked black, Q contains nothing or all gray vertices.

Another operation related to Q is the insertion inside the for loop in Line 13-18. For each adjacent vertex of n, if it is white, it is marked gray and inserted into Q. Since the vertex inserted into Q has always been marked gray beforehand, if Q contains nothing or all gray vertices before the for loop, Q contains nothing or all gray vertices after the for loop. Therefore, the loop invariant is maintained.

4.2.2.3 Termination There are two for loops and one while loop in the pseudocode. The two for loops will terminate because the number of total vertices and IP are finite.

The color of reachable vertices from roots can only change from white to gray and then to black. Inside the while loop, only white vertices can be marked gray and inserted into Q as shown in Line 13-19. The total number of reachable vertices from roots is finite. Therefore, the total number of white vertices that can be inserted into Q is finite. During each iteration, one

vertex is removed from Q and marked black. Therefore, after finite number of iterations, Q will become empty, all vertices will be marked black and $BFSMR(G, V)$ will terminate.

5 Iterative Multiplier

In this chapter, an 8-bit iterative multiplier is implemented to evaluate whether a QDI system can be reset properly with WRS and start normal operation without deadlock. The behavior of the iterative multiplier is described by CHP in the Appendix.

5.1 Iterative Multiplier

The iterative multiplier shown in Figure 30 stores two multiplicants $I0$ and $I1$ in register $m0$ and $m1$. $m0$ is a parallel-in serial-out register. It receives 8-bit data and outputs it bit by bit starting with the Least Significant Bit (LSB). $m1$ receives 8-bit data and outputs the same data for eight times. If the bit from $m0$ is 1, the multiplexer mux accepts data from $m1$ and outputs it to the adder $add8$; otherwise it outputs 0 to $add8$.

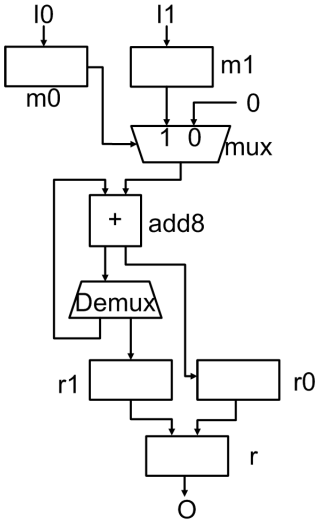

Figure 30: Iterative Multiplier

The LSB of the sum from $add8$ goes to $r0$ while the carry-out bit cascading with the other seven Most Significant Bits (MSB) of the sum goes to the demultiplexer $demux$. $demux$ sends out 0 to $add8$ initially in order to sum with the first data from mux. After that, it accepts data from $add8$ and sends data back for seven times before it sends data to $r1$. $r0$ is a serial-in parallel-out register while $r1$ is a normal register. In the end, $r1$ and $r0$ store the higher and lower byte of the result respectively. r combines the two bytes and outputs O.

Most of processes in the multiplier are implemented with PCHB templates. The rest are special blocks: *Demux* is an AP that starts by sending 0 to *add*8. A Source process continues sending 0 to one of the inputs of *mux*. Some functions such as copy, merge and split are implemented by slack-zero processes.

5.2 Simulation

The multiplier is simulated with an environment that accepts data from O, applies some function to the received data and sends higher and lower bytes of the results to $I0$ and $I1$ respectively. Therefore the system is closed and operates forever.

The multiplier resets and operates correctly under different process technologies, process corners and operating voltages. Reset time is shown in Figure 31. The X-axis specifies different simulation conditions. "LP" refers to TSMC 40nm Low-Power technology while "GS" refers to TSMC 40nm General-Purpose technology. The number following "LP" or "GS" is the operating voltage. For example, "06" is 0.6V while "10" is 1V. Three series of data represent three different process corners, i.e. TT (Typical NMOS, Typical PMOS), SF (Slow NMOS, Fast PMOS) and FS (Fast NMOS, Slow PMOS).

When the operating voltage increases, the reset time reduces. In addition, at the same voltage, LP process resets slower than GS process as expected.

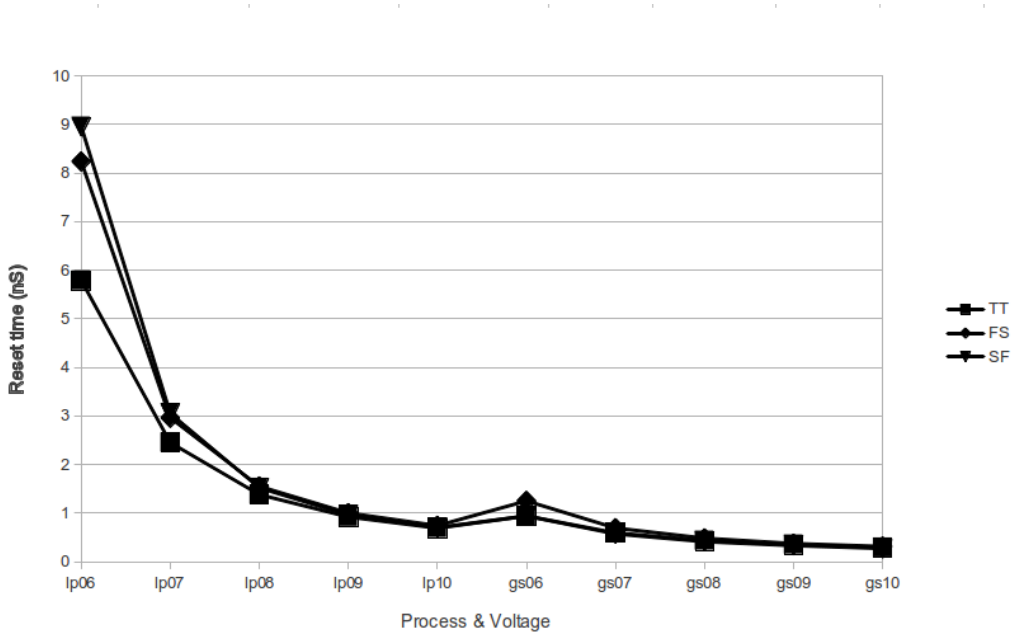

Figure 31: Reset Time for Different Process Corners

Similar results can be observed for cycle time in Figure 32. Cycle time is the time spent in computing one multiplication.

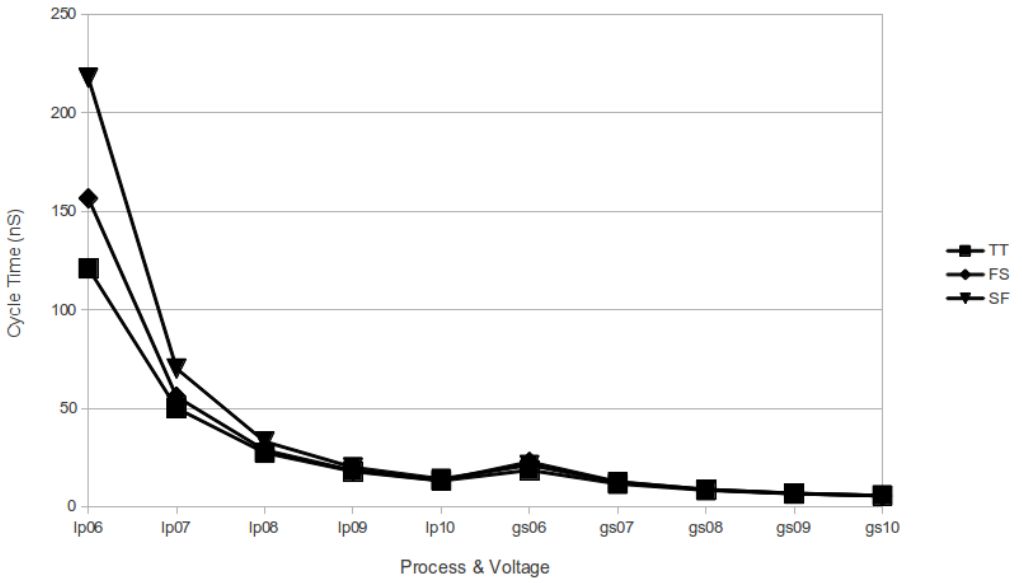

Figure 32: Cycle Time for Different Process Corners

LR can be replaced by GR in order to shorten reset time. However, distribution of GR may require a large network of rails once GR needs to be distributed to more processes. If every LR is replaced by GR, WRS changes to GRS. For the multiplier, reset signals for processes are gradually changed from LR to GR. The corresponding reset time for the multiplier is shown in Figure 33.

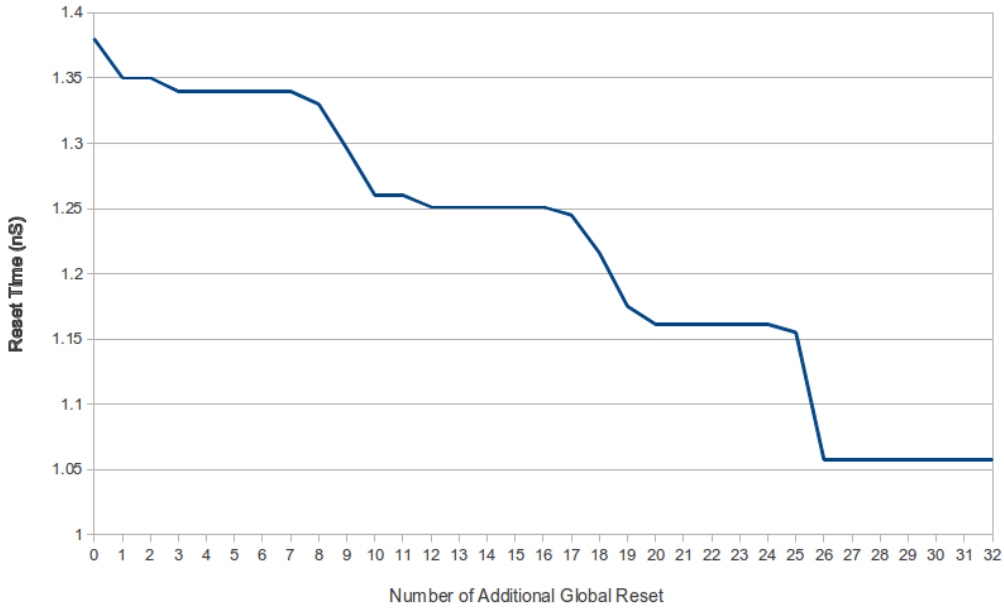

Figure 33: Reset Time with Additional Global Reset

As expected, the trend of reset time is decreasing as more LR are changed to GR. However, for some modification of reset signals from LR to GR, the reset time keeps constant. This is because the changed processes don't stay in the critical path of propagation of reset data.

6 Conclusion

In this thesis, reset schemes for QDI systems have been examined. Circuit implementation and operation protocol for both GRS and WRS have been discussed. Reset time of systems with WRS is dependent on the choice of LRI. An algorithm has been proposed to systematically choose the LRI in order to shorten the reset time. The proposed WRS has been applied to an iterative multiplier that operates correctly under different operating conditions.

In some sense WRS is a more general reset scheme for QDI systems than GRS. LR in WRS can be changed to GR in order to shorten reset time. This has been illustrated by the multiplier application. Once all LR for WRS are changed to GR, WRS changes to GRS. Therefore, GRS is a special case of WRS. If there are more LR in the system, the large network of rails and buffers distributing GR can be removed. All reset signals become local. On the other hand, having more GR will reduce the reset time of the QDI system. The choice of the number of GR is application dependent.

Both GRS and WRS rely on reset timing assumption because the state of the system before reset is "demonic" - all values are possible. We believe it is impossible to implement an asynchronous reset without any timing assumption.

Appendices

CHP Description of The Iterative Multiplier

```
process mult()(I0?, I1?: byte; O!: word)
chp{
  var x0, x1: byte;
  var y: word;
  *[I0?x0, I1?x1; y:=x0 * x1; O!y]
}
meta{
  instance m0: mult0; instance m1: mult1;
  instance mx: mux;   instance ad: add8;
  instance dx: demux; instance r1: rmsb;
  instance r0: rlsb;  instance r: result;

  connect I0, m0.I;
  connect I1,       m1.I;
  connect     m0.O,       mx.C;
  connect           m1.O, mx.I;
  connect     ad.I0,      mx.O;
  connect     ad.I1,dx.O0;
  connect     ad.O, dx.I;
  connect     ad.LSB,     r0.I;
  connect           dx.O1,      r1.I;
  connect all j:0..7:     r.I0[j],    r0.O[j];
  connect     r.I1,             r1.O;
  connect O,  r.O;
}

process mult0()(I?: byte; O!: bit)
chp{
  var x: byte;
  *[I?x; O!x[0]; O!x[1]; O!x[2]; O!x[3]; O!x[4]; O!x[5]; O!x[6]; O!x[7]]
}

process mult1()(I?, O!: byte)
chp{
  var x: byte;
  *[I?x;  <<; i:0..7: O!x>>]
```

```
}

process mux()(I?, O!: byte; C?: bit)
chp{
  var c: bit; var x: byte;
  *[C?c, I?x; [  c -> O!x
              []~c -> O!0]]
}

process demux()(I?, O0!, O1!: byte)
chp{
  var x: byte;
  *[O0!0; <<;i:0..6:I?x; O0!x>>; I?x; O1!x, O0!0]
}

process add8()(I0?, I1?: byte; O!: byte; LSB!:bit)
chp{
  var x0, x1, yp: byte;
  var y: word;

  *[I0?x0, I1?x1; y:=x0+x1; <<,i:0..7: yp[i]:=y[i+1]>>; O!yp, LSB!y[0]]
}

process rmsb()(I?, O!: byte)
chp{
  var x: byte;
  *[I?x; O!x]
}

process rlsb()(I?: bit; O[0..7]!: bit)
chp{
  var y: byte;
  *[ <<; i:0..7: I?y[i]>>; <<, i:0..7:O[i]!y[i]>> ]
}

process result()(I0[0..7]?: bit; I1?: byte; O!: word)
chp{
  var y: word; var x0, x1: byte;
  *[<<,i:0..7:I0[i]?x0[i]>>, I1?x1; <<, i:0..7: y[i]:=x0[i], y[i+8]:=x1[i]>>; O!y]
}
```

References

[1] A. J. Martin and M. Nystrom, "Asynchronous techniques for system-on-chip design" *Proc. IEEE* Volume 94, Issue 6, pp. 1089-1120, Oct. 2006.

[2] A. J. Martin, "The limitation to delay-insensitivity in asynchronous circuits" *Sixth MIT Conference on Advanced Research in VLSI*, pp. 263-278, 1990.

[3] A. M. Lines "Pipelined Asynchronous Circuits" M.S. thesis, CS, Caltech, Pasadena, CA, 1995

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms", 3rd ed. MIT Press and McGraw-Hill 2009, Ch. 22.4, pp. 449-451.

[5] P. Prakash, A. J. Martin "Slack Matching Quasi Delay-Insensitive Circuits", ASYNC 2006, pp. 195-204, 2006.