Optimal Evaluation Clocking of Self-Resetting Domino Pipelines*

Kenneth Y. Yun Ayoob E. Dooply
Dept. of Electrical and Computer Engineering
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0407
E-mail: {kyy,adooply}@ucsd.edu

Abstract

We describe a high performance clocking methodology for domino pipelines. Our technique maximizes the clock rate of the circular pipeline (“ring”) while maintaining the ring cycle time to be the worst-case combinational logic delay around the ring. It is relatively immune to global clock skew, incurs no latch overhead, allows up to 50% time borrowing, and offers a robust way of preventing race-through problems, adjusted for the worst-case time borrowing.

1 Introduction

In order to meet explosive performance demands in high-performance systems, designers are beginning to experiment with unconventional techniques, such as time borrowing and self-resetting logic. Growing emphasis on these “circuit-centric” techniques to improve the performance is traced back to the lack of revolutionary microarchitectural techniques in recent years. These new circuit techniques tend to have two common themes: (1) to tame latch overhead and clock skew and (2) to achieve near “average-case” performance.

In this paper, we describe a clocking method for domino pipelines that addresses both. Specifically, our clocking method adds nearly zero latch overhead, yields very high tolerance to clock skew and jitter, minimizes the clock period of the circular pipeline (ring) while maintaining the ring cycle time to be the worst-case combinational logic delay around the ring. In other words, the clock period of our domino ring is \( \frac{1}{2} \sum_{i=1}^{N} T_i \), where \( N \) is the number of pipeline stages and \( T_i \) is the worst-case combinational logic delay of the \( i \)th pipeline stage.

Our technique is similar to the technique used for transparent latch based designs [4] in this regard; however, it incurs nearly zero latch overhead because “roadblocking” (or latching) mechanisms are embedded in domino circuits. It allows time borrowing to “smoothen” the stage-to-stage delay variation as in [2], but it can tolerate a greater variation by allowing up to 50% time borrowing across the pipeline stage boundary. It offers a robust way of placing “roadblocks” (or equivalent to latches in static circuits), adjusted for the worst-case time borrowing. Finally, the minimum delay constraint to prevent the “race-through” problem is extremely mild, i.e., the minimum delay needs to be just large enough to cover the local skew (skew between clocks within the same pipeline stage).

The technique used in this paper draws from a large body of work in pipeline clocking techniques [6], latch-based pipeline circuits [3], domino circuits [2], time borrowing or cycle stealing techniques [4, 1], and self-resetting logic [5].

2 Problem Statement

Consider a circular domino pipeline with \( N \) pipeline stages as depicted in Figure 1. We assume that functional partitioning and coarse-grain timing partitioning have been done manually or by other means by designers. We further assume that we can determine the worst-case combinational logic delay of each pipeline stage using SPICE or other accurate timing analysis tools. The problem is to devise a clocking method that achieves the minimum cycle time around the circular domino pipeline without changing the number of pipeline stages.

![Figure 1. A circular pipeline. Stage 8 is also numbered as stage 0 to highlight its predecessor relationship to stage 1.](image)

3 Background

Below, we define clock skew, as used in this paper, and review the notion of time borrowing and its role in optimal clocking of pipelines.

3.1 Definition: Global Clock Skew and Local Clock Skew

A global clock skew is the skew between two clocks in different pipeline stages, and a local clock skew is the skew between two clocks within the same pipe stage. For example, the skew between \( \Phi_i \) and \( \Phi_{i+1} \), \( t_{\Phi_i \uparrow \rightarrow \Phi_{i+1} \downarrow} \), is considered a global skew, whereas the skew between \( \Phi_i \) and \( \Phi_i \), \( t_{\Phi_i \uparrow \rightarrow \Phi_i \downarrow} \), or \( t_{\Phi_i \downarrow \rightarrow \Phi_i \uparrow} \), is considered a local skew, as illustrated in Figure 2.

3.2 Time Borrowing

The clock rate of a synchronous pipeline is normally determined by the worst-case pipeline stage delays. However, not all pipeline stages incur the same worst-case delay, which means that some stages may have significant “dead time.” Time borrowing is a technique used to remedy this problem, i.e., to clock the pipeline
faster by re-allocating the slack across pipeline boundaries, as illustrated in Figure 3. The worst-case delays of pipe stages $P_1$ and $P_2$ are $T_1$ and $T_2$ respectively ($T_1 > T_2$). If the clock period is set to $T_1$, then $P_2$ has the dead time of $T_1 - T_2$. In this case, it is conceivable to set the clock period to $(T_1 + T_2)/2$, if the processing time for $P_1$ can be increased by intentionally skewing $F_2$ by $T_1 - T$, as long as the minimum delay of $P_3$ is long enough to cover the skew amount. In general, it is theoretically possible to achieve a clock period of $\frac{1}{\Delta} \sum_{i=1}^{N} T_i$, where $N$ is the number of pipe stages and $T_i$ is the worst-case delay of the $i^{th}$ pipe stage. In practice, however, it is infeasible to skew the clock by an arbitrary amount of time.

![Figure 2. Global skew vs. local skew. (a) An example clock tree; (b) Timing diagram.](image)

3.3 Optimal Clocking of Static Latch-Based Pipelines

In latch-based pipelines, we can achieve the same effect without intentionally skewing the clock as shown in Figure 4. $\Delta_i = \max(\Delta_{i-1} + \delta_i, 0)$, where $\Delta_{i-1}$ is the cumulative borrowed time for data entering pipe stage $i$ and $\delta_i = T_i - T$. Clearly, $\Delta \geq 0$, i.e., the “fast” data must wait for the latch to become transparent. The pipeline functions correctly, as long as the worst-case cumulative time borrowed never exceeds 50% of a clock period (for 50% duty cycle clock), i.e., $\Delta < 0.5T$.

![Figure 4. Time borrowing in latch-based pipelines.](image)

Table 1 shows an example of cumulative time borrowing. The worst-case cycle time of the ring ($\sum_{i=1}^{8} T_i$) is 16ns, so the clock period is set to 2ns. Column 3 (with subheading 1) lists the cumulative time borrowed at each pipe stage for a cycle beginning at stage 1 with zero time borrowed, e.g., a cycle that starts with data entering stage 1 immediately following a global reset. Column 4 (with subheading 4) applies to a cycle beginning at stage 4. In both cases, the worst-case cumulative time borrowing of 0.9ns occurs in stage 6.

Note that the global clock skew (skew between clocks in different pipe stages) only affects the cumulative time borrowing, not the clock period, as long as the peak cumulative time borrowed does not exceed 0.5T (for 50% duty cycle clock). Column 5 of Table 1 lists the effective clock period ($\hat{T}$) after the skew of $\pm 0.1$ns is taken into account. In this case, $\hat{\delta}_i = T_i - \hat{T}$; $\Delta_i = \max(\Delta_{i-1} + \hat{\delta}_i, 0)$.

<table>
<thead>
<tr>
<th>Pipe stage</th>
<th>$T_i$</th>
<th>$\Delta_i$</th>
<th>$\hat{T}$</th>
<th>$\Delta_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.9</td>
<td>0.0</td>
<td>1.9</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>2.1</td>
<td>0.1</td>
<td>2.1</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>2.2</td>
<td>0.3</td>
<td>1.9</td>
<td>0.3</td>
</tr>
<tr>
<td>4</td>
<td>1.9</td>
<td>0.2</td>
<td>2.1</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td>2.3</td>
<td>0.5</td>
<td>1.9</td>
<td>0.5</td>
</tr>
<tr>
<td>6</td>
<td>2.4</td>
<td>0.9</td>
<td>2.1</td>
<td>0.8</td>
</tr>
<tr>
<td>7</td>
<td>1.5</td>
<td>0.4</td>
<td>1.9</td>
<td>0.4</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
<td>0.1</td>
<td>2.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 1. Cumulative time borrowing. $T = 2$ns. Assume that the cumulative time borrowed is 0 at power-up.

4 Self-Resetting Domino Pipelines Using Two-Phase Evaluation Clocks

A block diagram of a simple self-resetting circuit (without forks or joins) is shown in Figure 5.
Roadblock placement constraints are described below. We use before the roadblock in the pipe stage, as depicted in Figure 6.

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}

For the remainder of this subsection, we assume that the self-resetting feature handles the precharging. Clocks are only used to regulate the flow of data, e.g., to control roadblocks. In the next subsection, we describe the constraints on precharging for the scheme to work correctly. Assume that there are $R$ logic stages before the roadblock in the pipe stage, as depicted in Figure 6. Roadblock placement constraints are described below.\footnote{In order to simplify exposition, henceforth, when we say “a token moving through a pipe stage,” we mean “computation progressing through a pipe stage.” Likewise, when we say “a token entering a pipe stage,” we mean “new data entering a pipe stage.”}
5 Self-Resetting Circuit Implementation Issues

There are several practical issues that need to be addressed for robust self-resetting circuits.

1. For every fork in a pipe stage, we need to ensure that all the immediate successor stages in the branches are “non-blocking.” For example, substage 1 of the pipe stage shown in Figure 8 cannot drive substage 9 directly because the output of substage 1 may be reset before the roadblock at substage 9 is lifted ($\Phi$ rises). Thus substage 8 is needed. If substage 8 does not exist, then a precharged buffer can be inserted in its place.

2. Precharging predecessors to a join in a pipe stage requires special attention. In general, it is sufficient to precharge all the predecessors after the last one completes evaluation. However, further optimization is possible in virtually every practical circuit. For example, substage 7 in Figure 8 is a domino multiplexor with a dual-rail select input. The bottom path to the multiplexor is a long path, and the top path a short one. We assume that select inputs are set up before the long path completes computation. Thus substage 6 can be self-precharged safely, regardless of which input is selected. However, to precharge substage 10, the circuit needs to know if a path has been selected. If the long path has been selected, then substage 10 can be precharged safely since its output is not needed. Likewise, if the short path is selected, it can be self-precharged. However, if the selection has not been made, the circuit needs to wait, in case the short path is selected.

3. Precharge pulse generation and buffer delays are in the critical path of the precharge cycle. The precharge buffer delay depends on the width of the datapath. For 64-bit datapaths, precharge buffers incur about 3 FO4 (fanout of 4) delay. In our experimental implementation (64-bit datapath), the sum of precharge pulse generation and precharge buffer delays is 5 FO4 delays (see Figure 9). Clearly, a full-blown completion detection is an overkill, i.e., a “coarse” completion detection, which may sometimes declare completion earlier than the actual completion time, is sufficient, as long as $t^b_j - \tau^b_{j+1} > \Delta_{early-actual}$.

6 Experimental Results

We showed the practicality of self-resetting circuits using our experimental implementation (8-stage ring; each stage with 6-8 domino substages). The clock period selected was 12.5 FO4 (clock rate of 400MHz in 0.6$\mu$m CMOS). The average domino stage delay was 2 FO4. The precharge pulse generation and buffer delay was 5 FO4, whereas the precharge pulse width was 4 FO4. Thus the precharge cycle time was 11 FO4 — well within a clock period.

![Precharge sequencing constraints](image)

![A complex self-resetting pipeline stage](image)

![A SPICE simulation result of our experimental circuit](image)

<table>
<thead>
<tr>
<th>Delay</th>
<th>ns</th>
<th>FO4</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock period</td>
<td>2.5</td>
<td>12.5</td>
</tr>
<tr>
<td>domino stage eval.</td>
<td>0.4</td>
<td>2</td>
</tr>
<tr>
<td>precharge pulse gen.</td>
<td>1.0</td>
<td>5</td>
</tr>
<tr>
<td>precharge pulse width</td>
<td>0.8</td>
<td>4</td>
</tr>
<tr>
<td>precharge cycle</td>
<td>2.2</td>
<td>11</td>
</tr>
</tbody>
</table>

Acknowledgment

We would like to thank M. A. Horowitz and D. Harris of Stanford University for many interesting discussions and suggestions.

References