Register Transformations with Multiple Clock Domains*

Alvin R. Albrecht and Alan J. Hu

Department of Computer Science
University of British Columbia
{albrecht,ajh}@cs.ubc.ca

Abstract. Modern circuit design increasingly relies on multiple clock domains – at different frequencies and with different phases – in order to achieve performance and power requirements. In this paper, we identify a special case of multiple clocking that encompasses typical design styles, and we present a theory enabling a wide range of register transformations relating to the multiple clock domains. For example, we can perform pipelining, phase abstraction, and retiming across clock domain boundaries. We believe our theory will be useful to extend current work on formal hardware design, synthesis, and verification to multiple-clock-domain systems.

1 Introduction

Modern circuit design increasingly relies on multiple clocks running at different frequencies and with different phases. The reasons are to attain higher performance, to reduce power consumption, and to allow integration of diverse components into systems-on-chip.

Higher performance comes mainly through the use of level-sensitive (a.k.a. transparent or flow-through) latches. A level-sensitive latch allows its input to flow combinationally to its output when its clock input is high, but ignores its input and holds the value of its output when its clock input is low. To prevent combinational loops from fast signals "racing through" the latch when it is transparent, latch-based designs group the latches into several distinct, non-overlapping clock phases. Only one phase is transparent at a time. (See Fig. 1.) This multi-phase design style is more complex than a single-clock-domain design, but allows more relaxed circuit timing constraints, faster clock speeds, and higher performance (e.g., [5]).

Multiple clock domains are used to reduce power consumption because synchronous CMOS dynamic power dissipation is linear in the clock speed. For performance reasons, not all parts of a chip can run at a slow clock speed, but for power savings, not all parts of the chip should run at full speed, either. Designing with multiple clock domains allows different parts of a chip to run at their most efficient speeds. More elaborate techniques involving multiple clock phases can produce substantial additional power savings [16].

An exponential gap is widening between the number of transistors per chip and the number of transistors that a human designer can generate per unit time. The trend in

* This work was supported in part by a research grant from the National Science and Engineering Research Council of Canada.

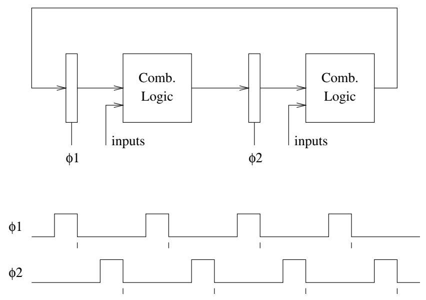

Fig. 1. Basic Two-Phase Clocking: The latches are transparent when their clock input is high. Two non-overlapping clocks ϕ_1 and ϕ_2 prevent both groups of latches from becoming transparent at the same time. The tic marks on the falling edges of the clocks show when the latch closes and holds its current value. This clocking style allows for higher performance designs than single edge-triggered clocking

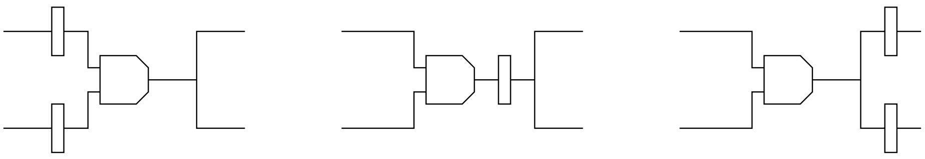

Fig. 2. Retiming allows moving registers (shown as small rectangles) forward or backward through arbitrary combinational logic gates and fanout stems. These three circuits have the same cycle-by-cycle behavior. Moving registers forward in the circuit (left-to-right in the figure) is called *forward retiming*. Moving registers backward is called *backward retiming*

design, therefore, is towards system-on-chip, in which multiple previously-designed IP blocks are integrated into a complete system on a single integrated circuit. Obviously, different blocks will have different performance and clocking needs, necessitating multiple clock domains on a single chip. For example, ARM's AMBA on-chip bus standard explicitly defines a high-performance clock domain and a separate, slower clock domain for peripherals [1]. As system complexity grows, additional clock domains will also be needed for slow, cross-chip communication.

With this increasing reliance on multiple-clock-domain design styles, new formalisms are needed to reason about, transform, and optimize these circuits. In this paper, we present a theory allowing a wide variety of register transformations to be done to multiply clocked circuits.

1.1 Background

Throughout this paper, we make extensive use of retiming, so we give a brief explanation here. Retiming is a behavior-preserving (at the cycle level) circuit transformation that allows moving registers forward and backward through combinational logic gates and fanout stems (Fig. 2). Leiserson and Saxe [12] laid the original foundation for retiming, showing how the problem of optimum retiming could be formulated as a linear program, and finding several important applications including clock period minimization, state minimization and pipelining of unpipelined circuits.

The linear programming formulation of retiming naturally leads to the concept of negative registers. A normal register delays a signal on a wire by one clock cycle. A negative register *accelerates* a signal by one clock cycle. A register/negative-register pair is equivalent to a no-op. Obviously, circuits with negative registers cannot be implemented directly[1], but negative registers are a useful bookkeeping formalism and can be retimed through a circuit just as a normal register.

One subtlety of retiming is that it can change the behavior of the circuit for the first few cycles after reset. For example, in Fig. 2, the rightmost circuit can produce unequal values at its two outputs for the first cycle after reset, whereas the other two circuits cannot. After the first cycle, the circuits behave identically. In general, forward retiming always allows preserving the initial behavior of the circuit, whereas backward retiming might not. In some applications, the initial behavior is not important, and both forms of retiming can be used freely. In other applications, the initial behavior is important, so either we restrict to forward and limited cases of backward retiming, or else we add special reset circuitry to force the retimed circuit to behave properly for the cycles following reset (e.g., [18,8, 17,3]). Similarly, our formalism supports more general register transformations if initial behavior is not important and imposes additional constraints if initial behavior must be preserved.

The initial behavior problem is related to a more general concept: retiming can be viewed as running a circuit forward or backward through time. We introduce a distinguished *host node* that is connected to all inputs and outputs of the circuit. The host node represents the external view of the behavior of the circuit. Retiming a register forward through the host node is equivalent to having the circuit step forward one clock cycle; retiming a register backward through the host node is equivalent to backing up the circuit one clock cycle [18].

1.2 Related Work

The retiming techniques described above can be applied only to circuits with a single type of register clocked from a single global clock, or within a single clock domain in a multi-clocked circuit. With the growing importance of multiple-clock-domain design, a few research results dealing with multiple clock domains have started to appear.

Hasteer et al. [10] concentrate on k-phase, level-sensitive clocking, where there are k non-overlapping clocks, all with the same frequency, and introduce the concept of phase abstraction, in which all but one phase of registers is eliminated. The resulting circuit has $1/k$ as many registers, reducing the state space of the circuit for verification purposes. Baumgartner et al. [2] achieve a similar result for two-phase circuits using a different formalism that preserves the initial behavior of the circuit and allows some different optimization. We will show how both of these works can be understood in terms of register transformations in our theory.

In a pure, well-formed multi-phase circuit, the sequence of register phases along any path is the same. (This restriction also applies to the circuits considered in the above

[1] Hassoun and Ebeling propose implementing negative registers via precomputation or prediction [9].

two works.) In that case, the different clock phases never interact in a way that prevents retiming [14,11]. Only very recently has work started to appear regarding more general instances of multiple clock domains [6,7]. In these works, the key idea is that only if all the registers at the inputs (outputs) of a gate are in the same clock domain, can they be retimed forward (backward) across that gate. Our work enables changing the clock domains of registers, thereby creating additional opportunities for these retiming techniques to work.

In general, our work is intended to provide tools necessary to reason about multiple-clock-domain circuits in a unified manner. Our theory supports retiming, phase abstraction, and a number of other transformations not published previously.

The theorem-proving community has long relied on temporal abstraction to relate hardware models with different granularities of time (e.g., [15, Sect. 4.1.3, Chap. 6]. The basic idea is to introduce a monotonic mapping function that takes time values from one model to time values in the other. Typically, the mapping is from time values for an abstract specification model into time values for a concrete implementation model. In contrast, our research deals with multiple granularities of time within a single circuit. Nevertheless, one could imagine embedding our work into the more classical framework by treating our Sampling Formula (Sect. 2.2) as a temporal mapping function, defining the behavior of circuit elements to include temporal abstraction, and then developing the necessary theories analogous to our derived transformations to reason about circuit behaviors involving these temporal abstractions. The main advantage of such an approach is that it may be possible to unite our work with other temporal mapping functions in a single theorem-proving system. Such a formalization is a possible direction for future research.

2 Register Transformations with Multiple Derived Clocks

2.1 The Multiple Derived Clocks Abstraction

In this subsection, we introduce our abstraction for multiple clock domains: the *multiple derived clocks* (MDC) abstraction. Just as the synchronous circuit model is an abstraction of the timing complexities of a single-clocked circuit, our MDC abstraction separates detailed low-level timing verification from higher-level functionality for multiple-clock-domain circuits.

First, we assume that the low-level clock routing and circuit timing analysis has been done properly. In particular, our model ignores clock skew (differences in timing for registers driven by the same clock), and we assume that signals have enough time to stabilize before they are latched. These premises are identical to the standard cycle-based view of a single-clocked, synchronous circuit, and allow separating the problem of timing analysis from issues of higher-level functionality.

If level-sensitive latches are used, we assume that the low-level timing analysis has properly accounted for the clocking scheme – in particular that adjacent latches have non-overlapping transparent periods. This model allows us to treat level-sensitive latches as edge-triggered latches, e.g., the circuit in Fig. 1 would behave exactly the same (assuming low-level timing constraints are met) if the latches were edge triggered at the tic marks. To map from an edge-triggered circuit back to a level-sensitive circuit requires the same low-level timing analysis on the level-sensitive circuit.

The general problem of interactions between multiple clock domains has long been studied (e.g., [5, pp. 473–475]). If clocks can be related arbitrarily, there is no way to eliminate the problem of metastability, and it is impossible to create a more abstract model. The key insight behind our MDC abstraction is that rarely do single-chip systems involve arbitrarily related clock domains. Instead, the clocks almost always maintain fixed, well-defined temporal relationships among themselves.

The reasons for this design style are practical. In order to avoid metastability and other timing problems, designers never want to risk having unknown clock timing. Furthermore, on a single chip, it is impractical to have multiple frequency references (e.g., crystals). As a result, all clocks on the chip are actually derived from a single, fast reference clock (e.g., [13]), using frequency dividers, phase-locked loops, etc.

Our MDC abstraction exactly captures this clocking style. We imagine a master reference clock with a certain clock period. Every clock in the system has a period that is some integer multiple λ of the master clock period and a phase offset θ with $0 \le \theta < \lambda$ that is an integer number of master clock periods. Every clock tick occurs at the same instant as a master clock tick. Each register triggers instantaneously on the ticks of its clock. Combinational logic is considered to have zero delay.

Accordingly, any clock in the system can be characterized by the pair (λ, θ). We define the convention that for all λ, all $(\lambda, 0)$ clocks tick simultaneously at reset. At all times, a (λ, θ) clock ticks exactly θ master clock periods later than the $(\lambda, 0)$ clock. Registers on a given clock are labeled with the (λ, θ) pair of their clock. If the reset value of a register is relevant, we will label a register with a triple (λ, θ, v), where v is the value of the register immediately after the circuit resets. Wires also belong to clock domains. A wire driven by a register belongs to the same clock domain as the register that is driving it. A wire driven by a gate belongs to the same clock domain as the inputs to the gate, if all the inputs have the same clock labeling. Otherwise, we can conservatively assign the wire to the clock domain $(1, 0)$, indicating that it can change as fast as the master clock. Inputs and outputs can also be assigned a period-phase pair to indicate how often the value changes or is sampled.

For example, in Fig. 1, we can imagine a master clock that ticks at twice the frequency of the ϕ_i, with the master clock ticks lining up with the falling edges (tic marks in the figure) of the ϕ_i. This master clock would be called $(1, 0)$, and ϕ_1 and ϕ_2 would be called $(2, 0)$ and $(2, 1)$ (or vice-versa, depending on which clock ticked at reset). If the inputs have the same period-phase label as the latches to its left, then each combinational logic block will also have the same period-phase.

Our theory does not directly handle gated clocks. In the most common usage, however, the gating function simply enables and disables clocking of some registers. In that case, we can treat a register with a gated clock as the same register with an ungated clock and the enable signal controlling a multiplexer (Fig. 3). While the loop limits retiming, we can still determine that the input is sampled and the output is driven by the (λ, θ) clock domain.

A series of registers will be denoted by the sequence of their period-phase pairs. For example, the expression $(2, 0)(2, 1)$ indicates a $(2, 0)$ register whose output is the

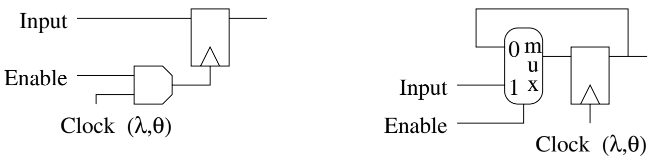

Fig. 3. The most common case of a clock gating can be handled by changing the circuit to one without clock gating. These two circuits are equivalent under a zero-delay timing model

input of a $(2, 1)$ register. We denote a series of n registers of the same type (λ, θ) by the expression $n(\lambda, \theta)$; or if the reset values of the registers are relevant, by the expression $n(\lambda, \theta, \sigma)$, where σ is a string of n reset values, one per register.

Abusing the notation slightly, we will denote a negative $(1, 0)$ register by $(-1, 0)$. A negative register written as a triple $(-1, 0, v)$ indicates that v is the initial value at the input to the register, which it will discard because it is a negative register.

2.2 Sampling Formula

Given a register of type (λ, θ), the following equation relates its input and output as a function of time:

$$\text{Out}[t] = \text{In}\left[\lambda \left\lfloor \frac{t-\theta}{\lambda} \right\rfloor - 1 + \theta\right]$$

where t is time measured in master clock periods, and the floor function $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

The circuit is "turned on" at time $t = 0$, when all registers take on their initial values. The equation relating individual register inputs and outputs is defined for all time, but when simulating the initial behavior of a physical circuit after reset, references to values at negative times in the equation should be considered references to the initial state of the register.

Our register transformation theory derives entirely from applying the sampling formula. For example, with two registers in series, the output of the first register drives the input of the second register, so $\text{In}_2[t] = \text{Out}_1[t]$. Let the two registers in series be $(\lambda_1, \theta_1)(\lambda_2, \theta_2)$. Then we can compute the output at time t as a function of the input as follows:

$$\begin{aligned}
\text{Out}_2[t] &= \text{In}_2\left[\lambda_2 \left\lfloor \frac{t-\theta_2}{\lambda_2} \right\rfloor - 1 + \theta_2\right] \\
&= \text{Out}_1\left[\lambda_2 \left\lfloor \frac{t-\theta_2}{\lambda_2} \right\rfloor - 1 + \theta_2\right] \\
&= \text{In}_1\left[\lambda_1 \left\lfloor \frac{\lambda_2 \left\lfloor \frac{t-\theta_2}{\lambda_2} \right\rfloor - 1 + \theta_2 - \theta_1}{\lambda_1} \right\rfloor - 1 + \theta_1\right]
\end{aligned}$$

In general, I/O equations for series combinations of registers can be found by working from right to left, successively making substitutions for $\text{In}_n[t]$ with $\text{Out}_{n-1}[t]$.

The I/O equations of three common series cases are summarized here:

$$\begin{aligned} n(1,0) &: \text{Out}[t] = \text{In}[t-n] \\ n(-1,0) &: \text{Out}[t] = \text{In}[t+n] \\ n(\lambda,\theta) &: \text{Out}[t] = \text{In}\left[\lambda\left\lfloor\frac{t-\theta}{\lambda}\right\rfloor - 1 + \theta - (n-1)\lambda\right] \end{aligned}$$

These relationships are used frequently in order to prove the transformations in the next section.

2.3 Derived Transformations

Using the sampling formula, we have derived several circuit transformations that allow adding registers, removing registers, and changing the clock domain of registers. This is obviously not a complete list of derivable transformations, but simply an illustration of some useful transformations that our theory can generate. Determining the completeness of our theory, finding a small and useful set of derived transformations, and determining how to use the transformations to optimize circuits are all open questions.

Except where noted, the rules are shown below in a more restricted form that preserves the initial behavior of the circuit. If the initial behavior is not important, all of the rules can be considered to omit the specification of initial register values. We use X to denote a don't-care value.

Backward (De)composition allows decomposing/composing faster-clocked registers before a slower-clocked register. Given $\frac{\lambda_1}{\lambda_2} \in I$ and $0 \le \theta_1 < \lambda_1$,

$$\lambda_2(1,0,\sigma)(\lambda_1,\theta_1,B) \equiv (\lambda_2,\theta_1 \bmod \lambda_2, A)(\lambda_1,\theta_1,B)$$

where σ is a string of λ_2 reset values, with the first θ_1 values from the right are X, then one A, then the rest are X. (If $\lambda_2 < \theta_1$, all of σ are don't-cares.)

Forward (De)composition allows decomposing/composing faster-clocked registers following a slower-clocked register. Given $\frac{\lambda_1}{\lambda_2} \in I$ and $0 \le \theta_1 < \lambda_1$,

$$(\lambda_1,\theta_1,A)\lambda_2(1,0,\sigma) \equiv (\lambda_1,\theta_1,A)(\lambda_2,\theta_1 \bmod \lambda_2, B)$$

where σ consists of: first $(\theta_1 \bmod \lambda_2 + 1)$ from the right are B, and the rest are A.

Register Transformer changes the types of registers on a clock domain boundary. Given $m\lambda_1 = n\lambda_2$ and $m, n \in I \ge 0$

$$(\lambda_1,\theta_1,A)m(\lambda_1,\theta_1,\sigma_1)(\lambda_2,\theta_2,C) \equiv (\lambda_1,\theta_1,A)n(\lambda_2,\theta_2,\sigma_2)(\lambda_2,\theta_2,C)$$

where σ_1 and σ_2 consisting entirely of the same value B is a sufficient condition to preserve the initial behavior of the circuit.

Common Period Rule decomposes adjacent registers on the same clock period. Given $0 \le \theta_1, \theta_2 < \lambda$, and let

$$n = \begin{cases} \theta_2 - \theta_1 & \text{if } \theta_2 > \theta_1 \\ \theta_2 - \theta_1 + \lambda & \text{otherwise} \end{cases}$$

then
(i)

$$(\lambda, \theta_1, A)(\lambda, \theta_2, B) \equiv n(1, 0, \sigma)(\lambda, \theta_2, B)$$

where σ is: first θ_2 from right are X, then A, rest are X, and
(ii)

$$(\lambda, \theta_1, A)(\lambda, \theta_2, B) \equiv (\lambda, \theta_1, A)n(1, 0, \sigma)$$

where σ is: first $\theta_1 + 1$ from right are B, rest are A.

Phase Abstraction allows elimination or insertion of certain phases in a multiphase clocking scheme. Given $0 < j < k \leq \lambda$ and $0 \leq \theta < \lambda$:
(i) if $[\theta + j] \bmod \lambda < [\theta + k] \bmod \lambda$,

$$(\lambda, \theta, A)(\lambda, [\theta + j] \bmod \lambda, X)(\lambda, [\theta + k] \bmod \lambda, B) \equiv (\lambda, \theta, A)(\lambda, [\theta + k] \bmod \lambda, B)$$

(ii) if $[\theta + j] \bmod \lambda > [\theta + k] \bmod \lambda$,

$$(\lambda, \theta, X)(\lambda, [\theta + j] \bmod \lambda, A)(\lambda, [\theta + k] \bmod \lambda, B) \equiv (\lambda, \theta, A)(\lambda, [\theta + k] \bmod \lambda, B)$$

Minimum Period Register Insertion allows elimination or insertion of $(1, 0)$ registers between two more slowly clocked registers. Given $0 \leq \theta_1 < \lambda_1$, $0 \leq \theta_2 < \lambda_2$ and either $\frac{\lambda_1}{\lambda_2} \in I$ or $\frac{\lambda_2}{\lambda_1} \in I$,

$$(\lambda_1, \theta_1, A)(\lambda_2, \theta_2, B) \equiv (\lambda_1, \theta_1, A)n(1, 0, \sigma)(\lambda_2, \theta_2, B)$$

where n is determined from:
(i) if $\frac{\lambda_2}{\lambda_1} \in I$ and $\theta_2 \bmod \lambda_1 \geq \theta_1 + 1$,

$$(\theta_2 \bmod \lambda_1) - \lambda_1 - \theta_1 \leq n \leq (\theta_2 \bmod \lambda_1) - \theta_1 - 1$$

(ii) if $\frac{\lambda_2}{\lambda_1} \in I$ and $\theta_2 \bmod \lambda_1 \leq \theta_1$,

$$(\theta_2 \bmod \lambda_1) - \theta_1 \leq n \leq (\theta_2 \bmod \lambda_1) - \theta_1 - 1 + \lambda_1$$

(iii) if $\frac{\lambda_1}{\lambda_2} \in I$ then for $0 \leq k \leq \frac{\lambda_1}{\lambda_2} - 1$,

$$1 - \lambda_1 + max\,[(k\lambda_2 + \theta_2 - \theta_1 - 1) \bmod \lambda_1] \leq n \leq min\,[(k\lambda_2 + \theta_2 - \theta_1 - 1) \bmod \lambda_1]$$

and σ is defined as follows: if $n < 0$, the first θ_1 values from the left are A, and the rest are X; if $n > 0$, then σ containing all A is sufficient to preserve initial behavior.

Domain Swallow allows neighboring registers in a different clock domain to be absorbed. Domain swallow is contrasted with the register transformer rule by noting that the absorbed register does not have to be sandwiched between two other registers. Although in some cases, it is possible to preserve initial behavior, the case analysis is cumbersome, so we present the transformation only for the steady-state behavior of the circuit. Given $0 \leq \theta_1 < \lambda_1$ and $0 \leq \theta_2 < \lambda_2$,
(i) if $\frac{\lambda_2}{\lambda_1} \in I$ and either $\frac{\theta_2 - \theta_1 - 1}{\lambda_1} \in I$ or $\frac{\theta_2 - \theta_1 - 1}{\lambda_2} \in I$ then

$$(\lambda_1, \theta_1)(\lambda_2, \theta_2) \equiv (1, 0)(\lambda_2, \theta_2)$$

(ii) if $\frac{\lambda_1}{\lambda_2} \in I$ and $\frac{\theta_2-\theta_1}{\lambda_2} \in I$ then

$$(\lambda_1, \theta_1)(\lambda_2, \theta_2) \equiv (\lambda_1, \theta_1)(1, 0)$$

(iii) if $\frac{\lambda_1}{\lambda_2} \in I$ and $\frac{\theta_2-\theta_1-1}{\lambda_2} \in I$ then

$$(\lambda_1, \theta_1)(\lambda_2, \theta_2) \equiv (1, 0)(\lambda_1, \theta_1)$$

Phase Change changes the phase of a register at the expense of more or fewer surrounding minimum period registers.

$$(\lambda, \theta, S) \equiv \begin{cases} n(-1,0)(\lambda, \theta - n, S)n(1, 0, S) & \text{if } 0 \leq n \leq \theta \\ (-1, 0, A)(\lambda, \lambda - 1, A)(1, 0, S) & \text{if } \theta = 0 \\ n(1, 0, X)(\lambda, \theta - n, S)n(-1, 0, S) & \text{if } n < 0 \end{cases}$$

Time Passage A forward movement of a $(1, 0)$ register past another register is equivalent to a single cycle passage of time on the register passed. A reverse movement of a $(1, 0)$ register past another register is equivalent to a single cycle reversal of time on the register passed. The analogous interpretation exists for the movement of $(-1, 0)$ registers, but with time reversed.

(a)

$$(1, 0, I)(\lambda, \theta, S) \equiv \begin{cases} (\lambda, \lambda - 1, I)(1, 0, S) & \text{if } \theta = 0 \\ (\lambda, \theta - 1, S)(1, 0, S) & \text{if } \theta \neq 0 \end{cases}$$

(b)

$$(\lambda, \theta, S)(1, 0, I) \equiv \begin{cases} (1, 0, X)(\lambda, \theta + 1, S) & \text{if } S = I \\ (1, 0, S)(\lambda, 0, I) & \text{if } \theta = \lambda - 1 \end{cases}$$

(c)

$$(\lambda, \theta, S)(-1, 0, S) \equiv \begin{cases} (-1, 0, X)(\lambda, \theta - 1, S) & \text{if } \theta > 0 \\ (-1, 0, A)(\lambda, \lambda - 1, A) & \text{if } \theta = 0 \end{cases}$$

(d)

$$(-1, 0, A)(\lambda, \theta, S) \equiv \begin{cases} (\lambda, \theta + 1, S)(-1, 0, S) & \text{always} \\ (\lambda, 0, S)(-1, 0, S) & \text{if } \theta = \lambda - 1 \text{ and } A = S \end{cases}$$

The notion of an environment host node having all outputs of a circuit tied to the input of the node and all inputs to a circuit tied to the output of the node allows forward retiming through the host node to represent forward time passage in the circuit. Likewise, reverse retiming through the host node represents reverse time passage in the circuit. Registers leaving the host node and entering the inputs to the circuit are initialized with the circuit's input values from the environment. Registers entering the host node and leaving the outputs of the circuit carry the output values sampled by the environment. This understanding of time passage in single clock domain circuits is described in [18].

In our general multitple-clock-domain case, $(1, 0)$ registers passing through the host node represent time passage in units of the quickest period in the circuit. The time passage rule described here propagates input values on $(1, 0)$ registers from the host node, through the circuit and back to the host node where they carry the output value sampled by the environment. A complete traversal of one such register passing through the host node represents one cycle of time passage.

Spontaneous Generation $(-1,0)$ registers accelerate signals along a wire by one cycle, effectively annihilating preceding $(1,0)$ registers. A $(-1,0)$ register followed by a $(1,0)$ register will not annihilate each other unless they hold the same state. A $(-1,0)$ register's "state" is the input value on its left that it annihilates. Since it is known what the input value to the left of the negative register is, it is safe to annihilate the same value to the right without changing circuit behavior.

$$wire \equiv \begin{cases} (1,0,S_1)(1,0,S_2)...(1,0,S_n)(-1,0,S_n)(-1,0,S_{n-1})...(-1,0,S_1) \\ (-1,0,S_1)(-1,0,S_2)...(-1,0,S_n)(1,0,S_n)(1,0,S_{n-1})...(1,0,S_1) \end{cases}$$

This rule is best understood in the context of a circuit's environment. Imagine taking a circuit and cutting a particular wire connecting two halves. Then simulate this whole circuit n cycles forward in time in the manner described earlier. In that case, $n(1,0)$ registers will accumulate on the left side of the cut wire, holding the n input values that would have been sampled on the right side of the cut wire had the cut not been made. On the right side of the wire, introduce the sequence $n(-1,0)n(1,0)$ and then retime the positive registers created on the right to the outputs to supply the host node and complete a single cycle time passage. Facing each other across the cut wire are the $n(1,0)$ registers on the left and the $n(-1,0)$ registers on the right. Reconnect the wire and using this rule, conditions on the state of the negative registers are found in order for the two sets to annihilate each other. These conditions on the state of the negative registers impose conditions on the state of the positive registers that were retimed toward the output. Effectively, negative registers act as place holders for expected input yet to be seen.

Each of these transformations has been described assuming a certain series of registers existed on some wire of the circuit. However, it is not the existence of registers per se that allows these transformations to occur, but rather the different rate of information supply and sampling that occurs between adjacent registers of different type. If it is known that a wire carries (supplies or is sampled for) a certain type of information, it can take the place of a register in any of the rules listed above, provided that this "phantom" register is not transformed by the application of the rule.

3 Examples

In this section, we give a few short examples showing the application of the above rules. In the first example (Fig. 4), we demonstrate how Hasteer et al.'s [10] phase abstraction can be viewed in our framework. This example circuit is taken from their paper. Based on the information being supplied and sampled, we see that each $(2,1)$ register is being supplied by a $(2,0)$ wire, and its output is being sampled by a $(2,0)$ register. (Alternatively, we can retime the leftmost and rightmost $(2,0)$ registers inward, and then retime them back out at the end.) Therefore, each wire in the middle of the circuit is $(2,0)(2,1)(2,0)$, and our phase abstraction rule lets us remove the $(2,1)$ registers.

Our next example (Fig. 5) demonstrates how we can change the clock domain of registers, allowing retiming through a gate that has different register types on its inputs. This is something that was impossible with earlier theories of retiming. The series of registers

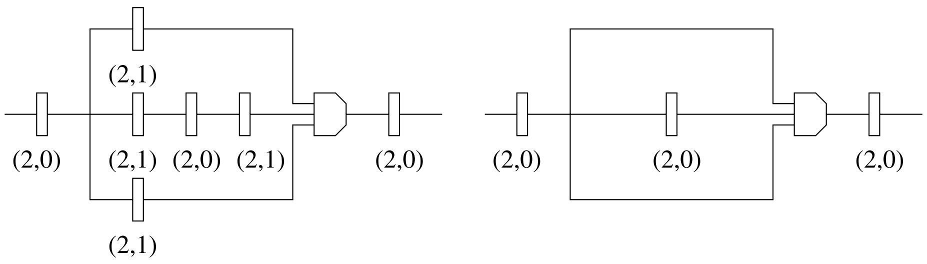

Fig. 4. Our phase abstraction rule converts $(2,0)(2,1)(2,0)$ to $(2,0)(2,0)$, allowing us to eliminate the $(2,1)$ registers

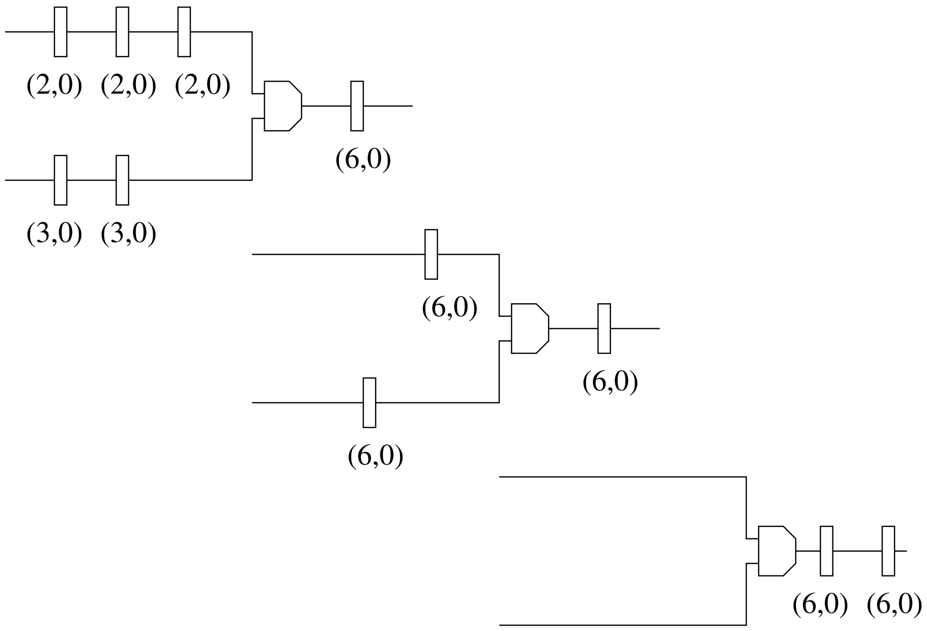

Fig. 5. The backward composition rule converts the $(3,0)(3,0)(6,0)$ register sequence into $(6,0)(6,0)$. Similarly, $(2,0)(2,0)(2,0)(6,0)$ becomes $(6,0)(6,0)$. Once the two inputs have been converted to the same register class, the registers can be retimed forward through the gate

are sampled by a $(6,0)$ register, so we can reason about the registers as $2(3,0)(6,0)$ for the top fanin and $3(2,0)(6,0)$ for the bottom fanin. Applying the backward composition rule converts both fanin branches to being $(6,0)(6,0)$. Now, we can retime the $(6,0)$ registers forward through the gate.

Our last example (Fig. 6) illustrates pipelining through different clock domains, another possibility created by our theory. The circuit shown illustrates incoming information arriving at a faster rate and being summarized into a slower rate output signal (e.g., imagine filtering an input signal). We can pipeline this circuit by inserting slow registers at the output and retiming them backwards through the circuit. The key trans-

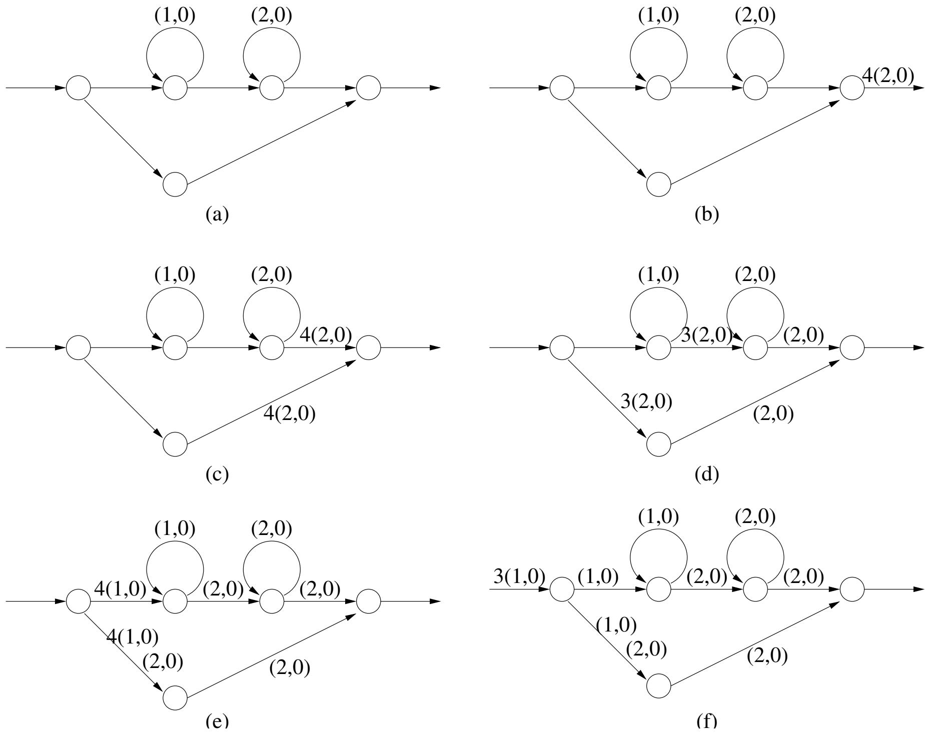

Fig. 6. For brevity, we indicate registers simply by the period-phase notation on edges. The circles denote arbitrary combinational logic. (a) Initially, we have a circuit with two clock domains that we wish to pipeline. (b) The output is being sampled at a $(2,0)$ rate, so we insert four $(2,0)$ registers at the output and start retiming them backwards, producing steps (c) and (d). In step (e), the backward decomposition rule allows us to convert $2(2,0)$ registers into $4(1,0)$ registers, which we can retime through the $(1,0)$ clock domain. (f) Finally, we retime $3(1,0)$ registers back through the first gate, leaving a single (1,0) register behind the combinational node to reduce cycle time

formation is that we can use the backward decomposition rule to convert registers from the slower clock domain into registers in the faster clock domain and then retime them into the faster part of the circuit.

4 Conclusion and Future Work

In this paper, we have introduced the multiple derived clock abstraction for multiple-clock-domain circuits and developed a theory for transforming the registers of these circuits. We have shown how these transformations allow a wide range of reasoning about and manipulation of multiple-clock-domain circuits in a unified framework.

Future work extends in both theoretical and practical directions. The obvious theoretical question is whether our theory is complete, or if we will need to introduce additional circuit transformation primitives.

Our theory suggests behavior-preserving circuit transformations, but it does not say how to use them. Ideally, one could derive optimum algorithms for applications such as minimizing clock period or state space, as has been done for single-clock-domain retiming. More likely, given negative complexity results on generalized retiming problems, we will need to develop good heuristics for using these transformations to optimize circuits.

To be useful in practice, our theory will need tool and language support. At the bare minimum, tools will need to be able to determine the clocking schemes used in a circuit. Possible ways to determine this information range from recognizing idiomatic styles of HDL coding, through user annotation of the HDL code, to new description languages with explicit language support for multiple clocking schemes. Unfortunately, the more elegant solutions will likely require greater methodological changes.

Finally, we hope that our theory will be useful to extend a wide range of formal design, synthesis, and verification techniques to multiple-clock domain circuits. On the design side, it may be possible to design an easy-to-prove-correct circuit, and then apply these transformations in a step-by-step, refinement based design/proof strategy to arrive at a high-performance, multiple-clock implementation. If such a strategy were automatable, our theory could provide additional design-space flexibility to a formal synthesis system. Similarly, our theory can provide new inference rules for applying theorem-proving to multiple-clock circuits. For model checking, modeling multiple-clock systems is a challenging task (e.g., [4]), which our theory could simplify, and our theory could allow more freedom for retiming to minimize the state space. Practical demands are forcing people to design with multiple clock domains; we need to develop the supporting theory.

References

1. ARM Limited. *AMBA Specification (Rev 2.0).* 13 May 1999.
2. Jason Baumgartner, Tamir Heyman, Vigyan Singhal, and Adnan Aziz. Model checking the IBM gigahertz processor: An abstraction algorithm for high-performance netlists. In *Computer-Aided Verification: 11th International Conference*, pages 72–83. Springer, 1999. LNCS Number 1633.
3. Gianpiero Cabodi, Stefano Quer, and Fabio Somenzi. Optimizing sequential verification by re-timing transformations. In *37th Design Automation Conference*, pages 601–606. ACM/IEEE, 2000.
4. Hoon Choi, Byeong-Whee Yun, and Yun-Tae Lee. Simulation strategy after model checking: Experience in industrial SOC design. In *International High-Level Design, Validation, and Test Workshop*, pages 77–79. IEEE, 2000.
5. William J. Dally and John W. Poulton. *Digital Systems Engineering.* Cambridge University Press, 1998.
6. Klaus Eckl and Christian Legl. Retiming sequential circuits with multiple register classes. In *Design, Automation and Test in Europe*, pages 650–656. IEEE, March 1999.
7. Klaus Eckl, Jean Christophe Madre, Peter Zepter, and Christian Legl. A practical approach to multiple-class retiming. In *36th Design Automation Conference*, pages 237–242. ACM/IEEE, 1999.
8. Guy Even, Ilan Y. Spillinger, and Leon Stok. Retiming revisited and reversed. *IEEE Transactions on CAD*, 15(3):348–357, March 1996.

9. Soha Hassoun and Carl Ebeling. Architectural retiming: Pipelining latency-constrained circuits. In *33rd Design Automation Conference*, pages 708–713. ACM/IEEE, 1996.
10. Gagan Hasteer, Anmol Mathur, and Prithviraj Banerjee. Efficient equivalence checking of multi-phase designs using retiming. In *International Conference on Computer-Aided Design*, pages 557–562. IEEE/ACM, 1998.
11. Alexander T. Ishii, Charles E. Leiserson, and Marios C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. *Journal of the ACM*, 44(1):148–199, January 1997. An earlier version of this paper appear in *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, March 1992.
12. Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
13. Bill Lin, Steven Vercauteren, and Hugo De Man. Embedded architecture co-synthesis and system integration. In *4th International Workshop on Hardware/Software Codesign*, pages 2–9. IEEE, 1996.
14. Brian Lockyear and Carl Ebeling. Optimal retiming of multi-phase, level-clocked circuits. Technical Report TR-91-10-01, University of Washington, Department of Computer Science and Engineering, 1991.
15. T. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.
16. Christos A. Papachristou, Mehrdad Nourani, and Mark Spining. A multiple clocking scheme for low-power RTL design. *IEEE Transactions on VLSI Systems*, 7(2):266–276, June 1999.
17. Vigyan Singhal, Sharad Malik, and Robert K. Brayton. The case for retiming with explicit reset circuitry. In *International Conference on Computer-Aided Design*, pages 618–625. IEEE/ACM, 1996.
18. Hervé J. Touati and Robert K. Brayton. Computing the initial states of retimed circuits. *IEEE Transactions on CAD*, 12(1):157–162, January 1993.