University of Alberta

Clock Logic Domino Circuits for High-Speed and Energy Efficient Microprocessor Pipelines

by

Raymond Jit-Hung Sung ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2003

Your file Votre référence
ISBN: 0-612-95894-9
Our file Notre référence
ISBN: 0-612-95894-9

I think and think for months and years.
Ninety-nine times, the conclusion is false.
The hundredth time I am right.
Albert Einstein

Praise be to our Lord Jesus Christ who sustained me throughout
the preparation of this dissertation and whose wisdom made possible all the
pages contained within. This thesis is also dedicated to my parents, who
have loved and provided for me throughout the countless late nights.
Thanks also to my brother who I had the privilege to work with in my graduate
classes and thanks to my sister who always made me laugh even after a long
day at school.

Acknowledgements

This work was sponsored by the National Sciences and Engineering Research Council of Canada (NSERC), the Alberta Informatics Circle of Research Excellence (iCORE), the Canadian Microelectronics Corporation (CMC), and by Micronet R & D. Special thanks goes out to Steve Dillen for introducing me to the spectreVerilog simulation environment and for comments concerning some of the ideas contained in this dissertation. Thanks also to John Koob who helped me with Latex and the VIM editor. Thanks to Raewadee Parnmukh for the completing the CL-domino ALU layout for future extracted layout with capacitance simulations. Last, a special word of thanks to my supervisor Duncan Elliott for the initial idea behind Clock Logic domino and for all of the comments, changes to the equations and edits.

Table of Contents

List of Figures

List of Tables

List of Nomenclature

Acronyms

A/D	Analog-to-Digital Converter
ALU	Arithmetic Logic Unit
$CD - Domino$	Clock-Delayed domino logic
$CL - Domino$	Clock Logic domino
CLA	Carry Lookahead Adder
$CNSA$	Conditional Sum Adder
CSG	Single-ended to domino-compatible Complementary Signal Generator
$CSG - Domino$	Single-rail domino logic with CSG generators
$CSLA$	Carry Select Adder
D/A	Digital-to-Analog Converter
$DDCVSL$	Domino Dynamic Cascode Voltage Switch Logic
DFT	Design-For-Testability
DLC	Dynamic Latch Converter
$DLC - Domino$	OTB domino with bolt-on DLC
DLL	Delay-Locked-Loop
$FIFO$	First-In-First-Out memory
$L1$	Level-1 Cache
LSB	Least Significant Bit
MCC	Manchester Carry Chain adder
mux	Multiplexer
$NFET$	N-Type Field Effect Transistor

$NMOS$ N-Type Metal Oxide Semiconductor transistor

OTB Opportunistic Time-Borrowing domino logic

PDN Pull-Down-Network. Logic function computed by NMOS transistors

$PFET$ P-Type Field Effect Transistor

PGK Propagate-Generate-Kill addition algorithm

PLA Programmable Logic Array

PLL Phase Lock Loop

PLL Phase-Locked-Loop

$PMOS$ P-Type Metal Oxide Semiconductor transistor

RAW Read-After-Write data hazard for microprocessor pipelines. Occurs when an instruction j tries to read a register before instruction i has written to it; where instruction i occurs before instruction j

$SIMD$ Single instruction, multiple data

SOI Silicon-On-Insulator

TLB Translation Lookaside Buffer

$TSMC$ Taiwan Semiconductor Manufacturing Company Ltd.

$VCDL$ Voltage-Controlled-Delay-Line

Abbreviations

wrt With respect to

Description

$Behavioral\ Verilog$ Verilog code that was not meant to be synthesized into actual logic gates but provide stimulus to or analyze results from a circuit under test

$clock\ skew$ The absolute difference between the nominal and actual interarrival times of a pair of physical clock edges

$Functional\ Verilog$ Verilog code that can be mapped directly to logic gates from a standard cell library

Figure Labels

$Dynamic$ A regular monotonic dynamic stage

$Dynamic^*$ An input complemented, non-monotonic or regular dynamic stage

$Dynamic^+$ An input complemented, non-monotonic or regular dynamic stage

Variables

$\alpha_{0\rightarrow 1}$	The transition probability or the probability that a logic gate will make an energy consuming transition in a given clock cycle
β	The effective transistor width of the PMOS transistor network divided by the effective width of the NMOS transistor network
a	The first input operand
b	The second input operand
f_{osc}	Oscillation frequency of a ring oscillator
$FO4$	Fanout-of-four inverter delay. The delay of an inverter driving four identical copies of itself while neglecting any wiring parasitics
g	Generate term in binary addition
$i(t)$	Instantaneous current
i_{ave_clock}	Average current sourced by a clock D/A interface element modeled by a boolean-controlled voltage source with zero output impedance
$I_{Leakage}$	The leakage current from the output node of a dynamic gate
k	A k-bit adder
N	Number of overlapping clock phases
n	Number of stages in a ring oscillator
p	Propagate term in binary addition
$p(t)$	Instantaneous power
$P:N$	Ratio of the widths of the PMOS transistors to the NMOS transistors in a static CMOS logic gate. Also referred to as Beta ratio
P_{ave_clock}	Average power dissipation for a single clock
$P_{ave\_transistors}$	Average power dissipated in the transistors during the period of interest
$psum$	Partial sum or sum computed without taking into account the carry-in
T	The period of interest during a transient analysis
$t_{AND\_eval\_cycle}$	AND Evaluate nominal logic evaluation window
$t_{borrow\_max\_OR\_pre\_OR\_eval}$	Time borrowing for OR-precharge/OR-evaluate
t_{borrow_max}	The maximum time that one phase can borrow from an adjacent phase, assuming all the clock overlap is used for time-borrowing

t_{borrow}	The actual overlap that is used for time borrowing while considering global and local clock domains
t_{cd}	The contamination delay time for dynamic logic
T_c	Cycle time as measured from the rising edge of the first phase clock until the next rising edge of the first phase clock
t_{delay}	The time that the overlapping clocks are delayed from the extra non-overlapping clocks in OTB domino logic
t_{eval_prev}	Logic evaluation window for the phase of logic preceding an input complemented or non-monotonic logic function
t_e	Period that the clock is high per cycle or clock duty-cycle (evaluate in the case of skew-tolerant domino clocks)
t_{hold}	The required overlap between two adjacent clock phases
$t_{min\_overlap}$	Nominal overlap between a first phase rising and a second phase falling
t_{PHL}	Propagation delay time for a high-to-low transition on the output of a logic gate
t_{PLH}	Propagation delay time for a low-to-high transition on the output of a logic gate
t_{prech}	The nominal time required for precharging a phase of dynamic logic when not accounting for clock skew
t_P	Overall propagation delay time for both high-to-low and low-to-high transitions on the output of a logic gate
t_p	Period that the clock is low per cycle (precharge in the case of skew-tolerant domino clocks)
$t_{skew\_max\_global\_OTB}$	The maximum tolerable skew for OTB domino
t_{skew_global}	The skew between two directly coupled domino gates from different local clock domains
t_{skew_local}	The skew between two directly coupled domino gates in a local clock domain
$t_{skew\_max\_global}$	The maximum global skew tolerable when global and local skews differ, assuming all clock overlap is used for skew tolerance
t_{skew_max}	The maximum skew tolerable between two adjacent clock phases if all the available overlap between the clock phases is used for skew tolerance
t_{skew}	The skew between adjacent clock phases
$timex$	Transit time through an A/D converter
V_{dd}	The positive supply rail
V_{dd}	The power supply voltage

V_{dyn}	The final value on the output of a dynamic gate after charge sharing
V_{ss}	The ground supply voltage
V_{ss}	The negative supply rail
V_{tn}	The threshold voltage of an NMOS transistor in a given process
V_t	The transistor threshold voltage
VIH	The accepted "high" voltage level where values above this level are interpreted as a logic '1'
VIL	The accepted "low" voltage level where values below this level are interpreted as a logic '0'
VOH	The "high" output voltage level
VOL	The "low" output voltage level
$t_{overlap}$	Nominal overlap between adjacent clocks

Chapter 1

Introduction

During the past three decades, the power consumption of integrated circuits, including microprocessors, has been increasing at an exponential rate with each new product generation [39]. This steady increase in power dissipation comes despite the use of advanced technologies and scaled power supply voltages. This is the result of several factors [40]: First, the number of transistors and the transistor density has doubled every 24 months [57]. This increase in the number of transistors on a die, commonly known as "Moore's Law", has enabled more complex architectural features to be incorporated into each new microprocessor generation. However, it is noteworthy that the power efficiency of microarchitectures, measured by MIPS/Watt, is considerably worse as more superscalar features are built into a design. Second, the use of more complex circuit techniques has allowed clock frequencies to increase faster than pure process scaling would suggest; often at the expense of increased power. Third, aggressive transistor technologies with higher current carrying capabilities ($I_{d_{sat}}$) and lower threshold voltages have increased switching speeds at the expense of significant subthreshold leakage current. Last, improvements in compilers and software applications have also increased the switching activity within a microprocessor.

If this trend continues, it is expected that the power consumption of microprocessors will be several thousand Watts by 2008 [39]. This presents an enormous challenge in the design of the power distribution networks needed to carry the large currents [40] and also in the verification of digital noise immunity. Furthermore, these predicted power numbers are prohibitively large from a reliability and system cost perspective. Last, from a system performance standpoint, high power dissipation limits the scalability in the number of processors that can be incorporated in a system [51] and the number of cores on a single die [16, 25]. From this discussion, it becomes clear that total power consumption will eventually become a limiting factor to increased chip integration.

Despite the power dilemma, designers are still most concerned about speed performance because, in most cases, that is what determines whether a system is successful. For most microprocessors incorporating advanced superscalar microarchitectures, this meant using dynamic domino logic. While, many of the design difficulties concerning noise and delay performance of dynamic logic have been addressed [8, 42, 43, 60, 61], practical power considerations have often been ignored.

This dissertation is concerned with developing domino logic circuits which minimize power consumption while maintaining equal delay performance when compared to existing techniques. The developed logic family, called Clock Logic (CL)-Domino, attains low power because it is a single-ended logic style that is functionally complete, unlike skew-tolerant domino or other domino logic styles that employ dual-ended gates for logic completeness. Second, CL-domino attains similar delay performance compared to dual-ended logic styles because it hides most of the clock skew overhead much like skew-tolerant domino while reducing wire delays. This is in contrast to Clock-Delayed (CD)-domino [72] which is another single-ended dynamic logic style that pays enormous skew penalties.

This thesis is organized as follows: Chapter 2 introduces the reader to domino circuits and how skew-tolerant domino circuits offer significant performance improvements in microprocessors. Chapter 3 introduces Clock-Logic domino and develops timing constraints and circuit design considerations related to the proposed logic style. Chapter 4 compares a representative microprocessor datapath in CL-domino versus one in dual-rail skew-tolerant domino and shows that CL-domino consumes less power while maintaining equal delay performance. Chapter 5 presents future work that could be taken to make CL-domino an even more viable design alternative. Finally, Chapter 6 summarizes the main results of the dissertation.

Chapter 2

Dynamic Logic Circuits

Dynamic logic is a digital circuit style used in high-performance integrated circuits, including microprocessors. In contrast to the more popular logic family of static CMOS, dynamic circuits are faster because they present much lower input capacitance for the same output current and have a lower switching threshold. Unfortunately, dynamic circuits are more susceptible to noise than static CMOS. While noise in static gates will lead to reduced performance, noise in dynamic circuits can cause functional failures which have to be addressed through significant analysis and verification. They also dissipate more power than their static counterparts because of their higher activity factors and significant clock loading. However, in many circumstances, they have proven to be the only circuit family able to meet the demands of reduced cycle times. Since domino circuits are synchronized by clocks, clock skew can have a significant impact on domino circuit performance. Skew-tolerant domino circuits have been shown to alleviate the effects of skew on the performance of traditional domino circuits.

This chapter is organized as follows: Section 2.1 discusses the basics of dynamic circuits and what general dynamic circuit families are preferred by many microprocessor manufacturers. Section 2.2 describes why dynamic circuit are faster than static circuits and explains their sources of power dissipation. Section 2.3 describes why dynamic circuits are often difficult to design because of their sensitivity to noise. Section 2.4 explains why circuits are pipelined and why skew can negatively affect pipelined circuits. Section 2.5 describes a method to pipeline dynamic domino logic circuits, called "skew-tolerant domino", to minimize the effects of skew while maximizing performance.

2.1 Dynamic Circuits

2.1.1 Dynamic Circuit Operation

In static logic, the output nodes of a logic gate are always actively driven to the power supply rails, V_{dd} or V_{ss}, through a network of transistors which provides at least one low resistance path. In dynamic logic, the output node is either driven to one of these rails, usually V_{ss}, or is left floating. When the output is in this floating/high impedance state, the logic value is stored as charge on the parasitic and load capacitors of the output node. Examples of simple NMOS dynamic gates are shown in Fig. 2.1.

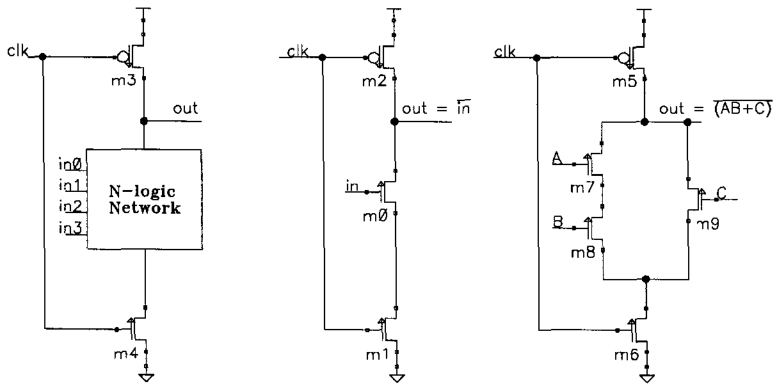

Figure 2.1: Simple N-type dynamic gates.

A dynamic logic gate operates in two phases: precharge and evaluate. The gate is preconditioned during the precharge phase. During this period, the clock is low and the PMOS clock transistor is on while the NMOS clock transistor is off. Node *Out* is thus precharged to V_{dd} and the logic gate maintains an output high value. During the evaluation period, the clock transitions from low to high and the PMOS clock transistor turns off while the NMOS clock transistor turns on. Thus, the output node can either remain at a logic high value or be conditionally discharged low to V_{ss} depending on the values of the inputs to the NMOS logic network. Dynamic gates are normally clocked by 50% duty cycle clocks so there is a half period for each of the precharge and evaluate operations [61].

2.1.2 Non-inverting and Monoticity Rules

Dynamic gates cannot be directly cascaded with a single clock. This is because the precharge level would cause the next stage to erroneously discharge. The inputs to an NMOS-logic network must be stable during the evaluation period or monotonically transition from a 0 $\rightarrow$ 1. If the inputs switch from a 1 $\rightarrow$ 0, there is no direct path from V_{dd} to the output node during evaluation and any lost charge cannot be replenished [68]. An example of a false evaluation for two directly coupled dynamic gates is shown in Fig. 2.2.

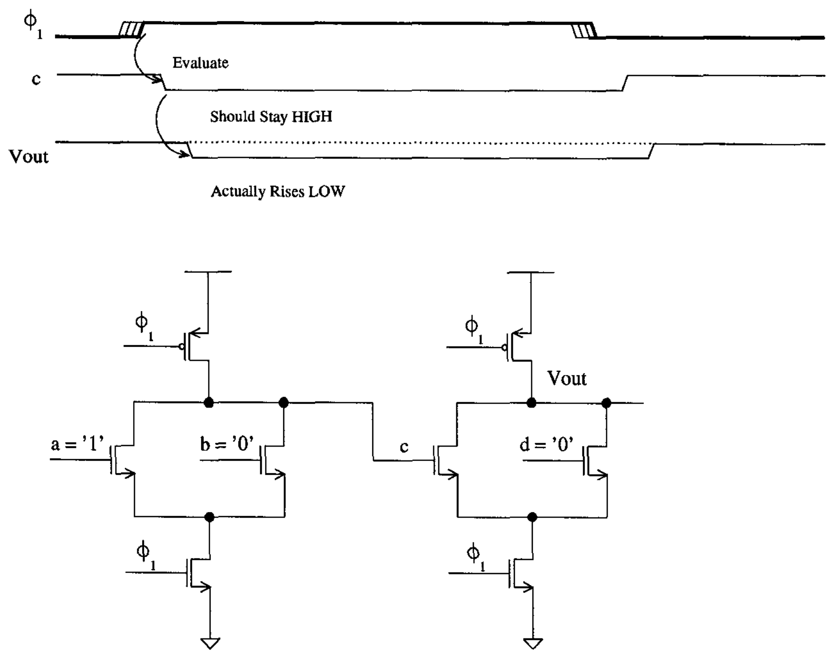

Figure 2.2: No direct cascading of dynamic gates with a single clock.

In this case the output of the two NOR gates in series should have produced a high result but instead produced a low result. This was because the second NOR gate could not pull the output node back high when the correct value of its C input arrived. Moreover, the first NOR gate could have discharged faster than the second NOR gate thus leaving that output at an indeterminate value between V_{dd} and V_{ss}. Thus the first NOR gate violated the monotonicity rule by making a 1 $\rightarrow$ 0 transition while the second NOR gate was in evaluation.

In the rare case of PMOS logic networks, as was popular in the early days of dynamic logic [38], the opposite condition holds true and the inputs can only make a monotonic 1 $\rightarrow$ 0 transition during evaluation as the gate cannot discharge a false charging of its output.

2.1.3 Domino Logic

Domino logic gates are a popular dynamic logic family that solves the monotonicity problem through inserting an inverting static gate between the dynamic gates [61]. Standard domino logic inserts an inverter between the dynamic gates while compound domino logic inserts multiple input complementary gates [8, 45]. The dynamic/static gate pair is known as a domino gate, although it is in fact constructed from two gates. A series of connected domino gates precharge simultaneously as if setting up a set of dominos. During evaluation, the first dynamic gate falls causing the static gate to rise which then causes the next dynamic gate to fall and its static gate to rise, much like a chain of toppling dominos. Examples of domino logic as applied to the previous example of two consecutive NOR gates is shown in Fig. 2.3. Unfortunately, to satisfy the monotonicity property we have constructed a pair of OR gates rather than a pair of NOR gates. In general, single-rail domino logic can only implement non-inverting logic functions. Furthermore, since non-monotonic logic functions do not have monotonic truth tables, non-monotonic XOR and XNOR functions cannot normally be implemented in single-rail domino logic either. An example of a compound domino gate is shown in Fig. 2.4. As is demonstrated by this figure, wide AND functions are usually well suited for compound domino since they would result in prohibitively high stack heights if a normal inverter is used. In fact compound domino gates often lead to reduced stack heights and enables increased logic complexity although they still cannot implement inverting or non-monotonic functions.

2.1.4 Dual-Rail Domino Logic

When true and complementary outputs are required or the function to be evaluated is non-monotonic, as in the case of XOR and XNOR functions, dual-rail differential output domino gates are often employed [43]. A dual-rail domino gate accepts true and complement versions of the inputs, usually from other differential domino gates, and produce true and complementary versions of its outputs. Both outputs of the dynamic gate are low during

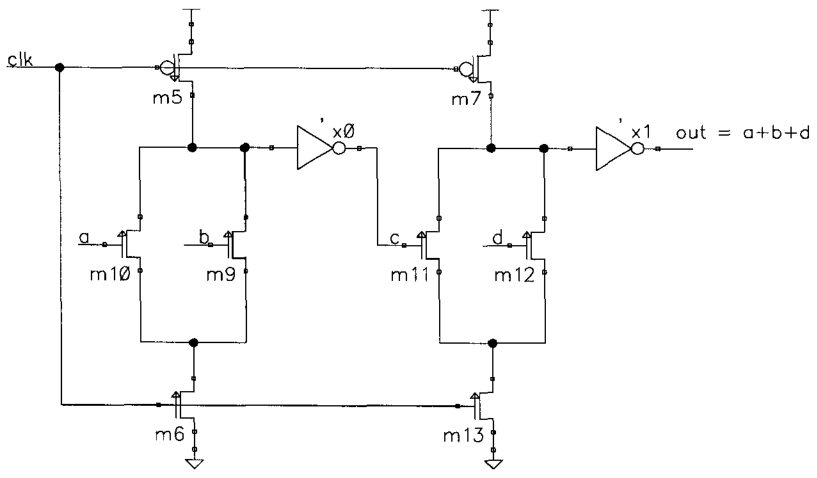

Figure 2.3: Two consecutive domino OR gates.

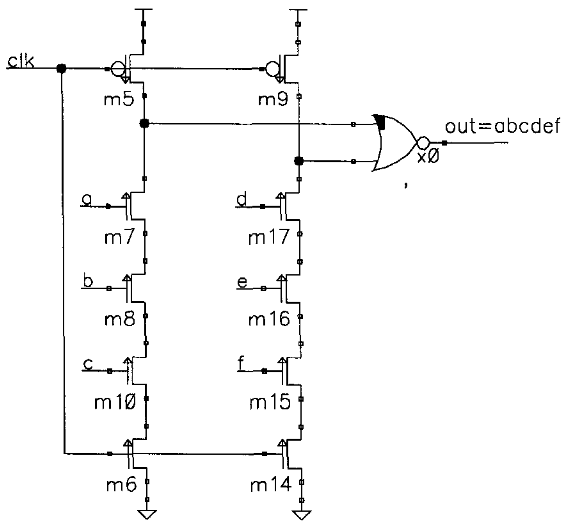

Figure 2.4: Compound domino 6-input AND gate.

the precharge phase and one of the pair makes a conditional $0 \rightarrow 1$ transition during evaluation. Because of its differential nature, one of the internal output nodes will always be discharged per cycle and bring the output of its static gate high. In order to implement a dual-rail gate, two separate pulldown stacks are required for the true and complementary versions of the function. However, the two pulldown stacks can often be shared to reduce the number of transistors. Thus a dual-rail domino gate is able to implement any arbitrary logic function since the monoticity rule is guaranteed for pulldown stacks that are complements. Two examples of dual-rail domino AND/NAND gates are shown in Fig. 2.5.

Dual-rail domino gates require dual-rail inputs. The gates producing these inputs must in turn receive dual-rail inputs. This means that entire blocks of logic must be implemented in dual-rail fashion [43]. This comes at a cost of twice as many wires to carry the dual rail signals, twice as many transistors (increased area), greater clock load, higher switching activity factors (higher power), and lower performance (when considering wire delays) compared to single-rail domino logic. Some designers have begun employing single-rail to dual-rail generators (Complementary Signal Generators (CSGs)) to convert single-rail domino to dual-rail domino whenever dual-rail gates to minimize the penalties inherent in dual-rail logic as described in Appendix B.

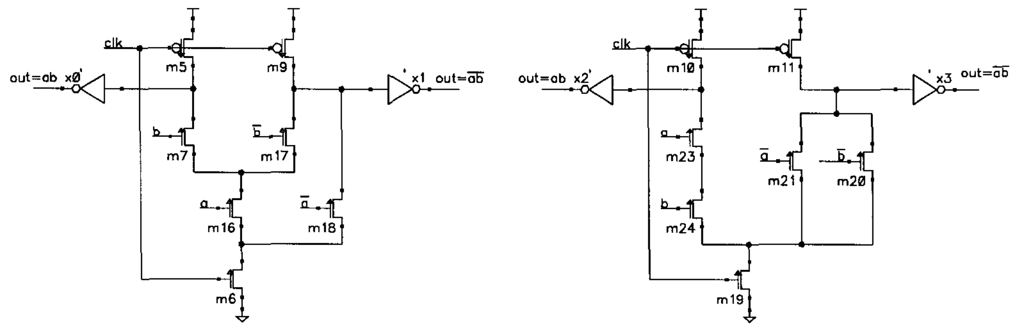

Figure 2.5: Two versions of dual-rail domino AND/NAND gates.

2.1.5 Footed Versus Unfooted Domino Gates

Since the inputs to domino gates are low during precharge, and evaluation of the gate cannot proceed until its inputs have evaluated, it is possible to eliminate the clocked evaluation transistor in order to decrease series resistance and increase circuit performance [8]. An example of an unfooted domino gate, interspersed with footed domino gates is shown in Fig. 2.6.

Unfooted domino gates are faster than footed domino gates as the stack height is reduced and the corresponding logic transistors can be made smaller to achieve the same output current [68]. However, unfooted domino circuits dissipate more power than footed domino gates since precharge cannot begin immediately after evaluate but instead has to ripple. Consider again the domino gates of Fig. 2.6. With the evaluation device removed, the unfooted gate (2), will not have its pull-down path disabled until all of its inputs go

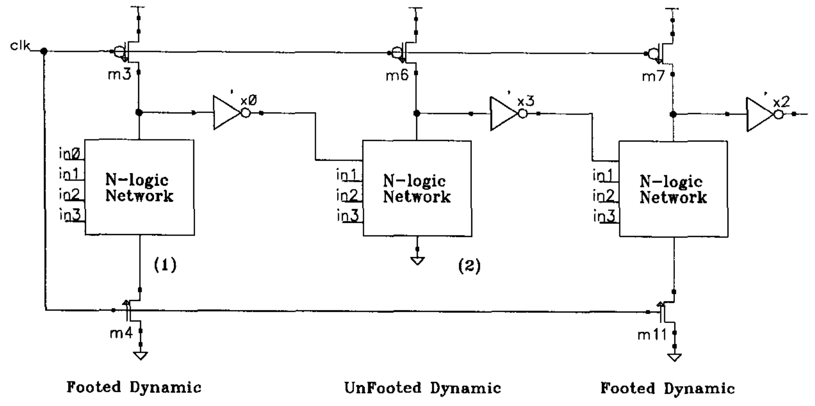

Figure 2.6: Unfooted domino gate.

low. This requires that all of the stage (1) dynamic gates complete their precharge and propagated the result through their static gates driving 0 to the input of gate (2). Before this happens, the PMOS and NMOS transistors of gate (2) are both on and form a voltage divider between V_{dd} and V_{ss} thus causing short circuit power dissipation. A method to reduce the short circuit current would be to add delay chains to delay the precharge of gate (2) by the estimated evaluate delay of gate (1) and its subsequent static gate as shown in Fig. 2.7 [8]. The precharge delay to (2) should not exceed the estimated evaluate delay to gate (1) since the unfooted gate would unnecessarily wait for the clock during the evaluate phase. This approach reduces short circuit current to some extent although not as much as it would seem initially since the logic gates are often favored to respond to the evaluation edge at the expense of the precharge edge for performance reasons. On the other hand, the removal of the evaluation transistor reduces clock load and the hence clock power of the system. Since the precharge must ripple through the unfooted domino gates, the precharge delay must also be carefully characterized. Some designs have been known to require a footed gate before every unfooted gate to reduce short circuit currents and meet precharge constraints [29].

2.2 Dynamic Circuit Performance

2.2.1 Delay Performance

Dynamic circuits are faster than equivalent static circuits for several reasons. First, the load capacitance is substantially lower since there is only a single NMOS transistor load per

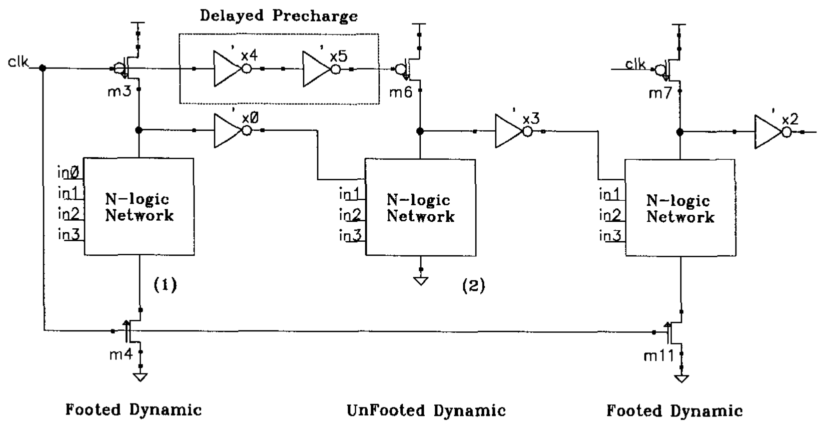

Figure 2.7: Delayed precharge of unfooted domino gate.

fan-in [61]. In contrast, inputs to static CMOS circuits must drive both NMOS and PMOS transistors. Since only one of the two transistors is on at any one time, the other transistor loads the input without contributing to the current drive of the gate [43]. It is the current drive of the gate that charges or discharges the output parasitic and transistor load capacitors that determine the state of a logic gate. Moreover, the PMOS transistors in a logic gate are often larger than the NMOS transistors because of their reduced carrier mobility and thus add much load capacitance. Figure. 2.8 compares typically sized static CMOS and dynamic 3-input NOR gates. The static CMOS NOR gate is sized for equal rise and fall delay while the dynamic NOR gate is sized to have the same fall delay as the static CMOS version. Note that the NMOS clocked evaluation and PMOS clocked evaluation transistors for the dynamic gate do not load the input. However, they do present load capacitance to the clock drivers which may affect their slew rates which then in turn directly affect the performance of a dynamic gate. The static CMOS NOR gate has seven units of gate capacitance per input while the dynamic NOR gate has only two units of gate capacitance. Thus dynamic gates present much lower input capacitance for the same output current. While NOR/OR functions are particularly poor in static CMOS, comparable NAND functions also present more input load for static gates. This analysis does not account for the increased internal parasitic capacitances that must be charged and discharged when more transistors are used to implement a logic function. Thus dynamic logic switches less total capacitance than does static logic.

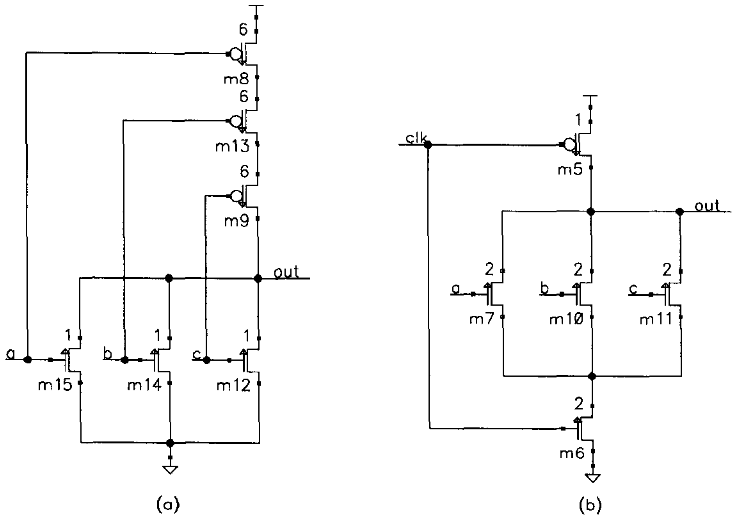

Figure 2.8: Three input NOR gates (a) static CMOS (b) dynamic logic.

Another reason that dynamic gates are faster than static gates is the reduced switching threshold. Whereas the outputs of static gates switch roughly when the input passes $V_{dd}/2$, for equal rise and fall times, the outputs of dynamic gates begin to switch when the input passes the NMOS transistor threshold voltage, V_{tn} [43].

The domino logic family uses intervening static gates after every dynamic gate to satisfy the monotonicity rule. The use of static gates somewhat slows down dynamic circuit performance. With the inclusion of clocked evaluation devices in footed gates, all the gates precharge in parallel and the precharge operation only requires two gate delays from charging the dynamic gate through bringing the output of its associated static gate low. Thus the critical path through the domino network is through the pull-down path of the dynamic gates and the PMOS transistors of the static gates. Therefore, to speed the critical rising output during evaluation, the static gates use wider than normal PMOS transistors and possibly smaller NMOS transistors in order to raise its switching threshold closer to V_{dd}. These high skew gates are often referred to as static gates with high beta ratios, where the beta ratio, β, is the effective transistor width of the PMOS transistor network divided by the effective width of the NMOS transistor network. The larger beta ratios affect the noise margin of the domino gate in that it is more susceptible to false switching if noise couples to its dynamic node. Therefore, in many designs β is kept within reasonable bounds such as 4:1 [43] or

5:1 [29].

The assumption of parallel precharging is slightly modified for unfooted gates, since those gates precharge in series. Thus, the high beta ratios would put a delay constraint on the precharge edge as well. However, most designs have a limitation on the length of unfooted domino paths so guaranteeing adequate precharging is usually not as much of an issue.

Finally, static circuits can exhibit spurious transitions or glitching due to finite propagation delays from one logic gate to the next which could lead to critical races and dynamic hazards [9]. In other words, an output node can have multiple transitions during the same clock cycle since inputs to a static logic gate can arrive at different times, thus reducing performance. Since the dynamic logic can only make a monotonic transitions during the same clock cycle, spurious transitions do not exist.

2.2.2 Power Performance

Dynamic logic dissipates more power than static logic mainly due to its increased switching activity resulting from periodic precharge and discharge operations [9, 61]. The switching activity of dynamic gates does not depend on the history of the inputs during the previous clock cycle. That is if a combination of inputs resulted in the discharging of the output node during the evaluate phase, then the output node will always be precharged during the precharge phase. Then the output node might be discharged again during the next cycle if the same or a different combination of inputs activates a pull-down path. In contrast, static CMOS logic would not have had its output pulled high if the same or a different combination of inputs in successive cycles resulted in activation of a pull-down path. The static output would have instead remained low between the clock cycles. Thus a dynamic gate would make a power consuming precharge transition from $0 \rightarrow 1$ if the output was discharged in the preceding evaluation phase. In other words, power is consumed every time the output equals 0, independent of the preceding or following values of the outputs [60, 61]. This signal transition probability can be expressed as:

$$\alpha_{0\rightarrow1\ single\_rail\_dynamic} = p_0. \tag{2.1}$$

On the other hand, the probability of an energy consuming $0 \rightarrow 1$ transition at the output of a static CMOS gate is equal to the probability that the gate is initially at the zero state multiplied by the probability that the next set of inputs will result in an output of 1. The product of two signal transition probabilities, which is defined as the transition probability, is therefore equal to:

$$\alpha_{0\rightarrow1\ static} = p_0 p_1 = p_0(1 - p_0). \tag{2.2}$$

This probability will always be lower than the signal transition probability of dynamic logic. This analysis is straightforward for single gates but becomes significantly more complex for logic gate networks. However, the same conclusion that dynamic gates have higher switching activity still holds true. The situation is worse for dual-rail domino logic since either the true or complementary output nodes are guaranteed to make a transition each cycle. Therefore, the switching probability is equal to:

$$\alpha_{0\rightarrow1\ dual\_rail\_dynamic} = 1. \tag{2.3}$$

Furthermore, the clock node for all dynamic logic gates has an activity factor of 1 and therefore significant power is consumed by the clock buffers in driving the clock transistors. In many cases, however, the clock is stopped when the functional unit containing the gate is not required [43], thus saving power to some extent.

If not for their high switching activity, the other metrics for power consumption show advantages for dynamic gates as opposed to static gates. For example the spurious transitions, where input signals arriving at different times can cause internal and output nodes in static CMOS to inadvertently switch, does not happen for dynamic logic. This is because dynamic nodes can only, by construction, undergo at most one power-consuming transition per cycle due to precharge [9]. Furthermore, short-circuit or crowbar current is reduced in dynamic logic compared to static logic since the pull-up path is never enabled when the gate is evaluating [61]. Therefore, the only causes of short circuit currents result when the clock switches or if the clock is skewed when between the precharge and evaluate clock transistors. These short circuit currents are small since the clock usually have very fast edge rates and the skew within a gate is often insignificant if good layout and clock distribution schemes are used. Short circuit currents do exist in domino logic since static gates are alternated between dynamic logic stages. However, the static gates are often much smaller in size and fewer in number than is the case for a static CMOS implementation of the same logic network. Furthermore, less physical capacitance is switched in dynamic logic which directly affects the power dissipation. It is worthwhile to note that weak keeper devices used to control leakage and charge sharing and also to maintain circuit state during clock power down will exhibit some non-negligible short circuit current. This occurs when the keepers function to replenish charge lost due to coupling noise and when they provide a brief period of contention against the pull-down network during evaluation. Furthermore, these keeper devices, which might have to be used in internal circuit nodes as well as on the output node, increase the transistor count thus increasing the switched capacitance. The method and use of keeper devices will be discussed in the following sections.

2.3 Dynamic Circuit Design Challenges

2.3.1 Charge Leakage

Pure dynamic gates have a minimum operating frequency because they rely on charge stored on the load and parasitic capacitors of the output node [61]. The output floats high (is in a high impedance state) during evaluation if the pulldown path is not activated and the dynamic gate evaluates to a 1. If the gate is left floating too long, the charge will eventually leak away and cause the logic gate to lose its evaluated value. As shown in Fig. 2.9, there are two main sources of leakage currents for each source/drain connection to the dynamic node, Dyn: (1) the subthreshold leakage current through the transistors and (2) the reverse biased diode current.

The subthreshold leakage current is the small current that flows through a device even when it is considered "off" ($V_{gs} < V_t$). The reverse-biased diode current is the result

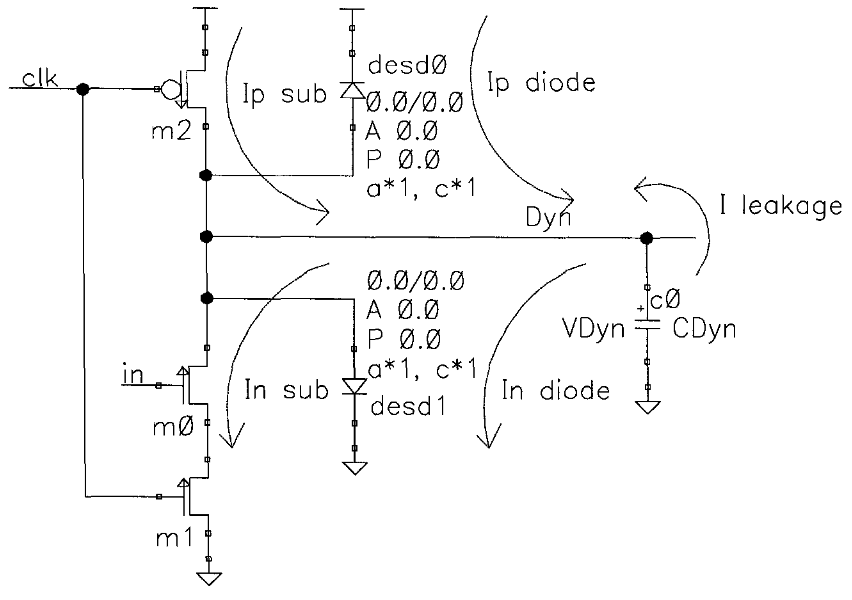

Figure 2.9: Sources of leakage current in a dynamic gate.

of parasitic p-n diodes formed by the n+ drain diffusion of the NMOS device and the p-substrate, and by the p+ diffusion of the PMOS device and the n-well. An expression for the sources of leakage in a dynamic gate is:

$$I_{Leakage} = (I_{N\_subthreshold} + I_{N\_diode}) - (I_{P\_subthreshold} + I_{P\_diode}). \quad (2.4)$$

From (2.4), it can be seen than the leakage currents from PMOS transistors add charge

$$I_{Leakage} = (I_{N\_subthreshold} + I_{N\_diode}) - (I_{P\_subthreshold} + I_{P\_diode}). \quad (2.4)$$

to the dynamic node while the leakage currents from the NMOS transistors subtract from it. If the expression is positive, then the charge on the dynamic node will eventually leak away. This is almost always the case in dynamic circuits since the effective width of the NMOS devices connected to the dynamic node is greater than the effective width of the PMOS devices. This is true in the case of wide NOR gates with many NMOS transistors connected to the dynamic node and is a serious design problem since designers often reduce critical path logic functions to wide fan-in dynamic NORs for maximum circuit performance.

A widely applicable method to keep the dynamic node from floating is to introduce weak feedback keepers [8]. A keeper is a weak PMOS transistor coupled back to the dynamic node through either an output inverter, as is the case of single-rail domino logic, or from another complementary dynamic node, as is the case for dual-rail domino logic. Addition of keeper transistors makes the dynamic circuit pseudo-static or bistable. In other words, they help to maintain the state of the dynamic node(s) when the clock frequency is reduced or the clocks are stopped altogether. Examples of typical keeper circuits for single-rail domino logic are shown in Fig. 2.10 while those for compound domino logic and dual-rail domino logic are shown in Fig. 2.11.

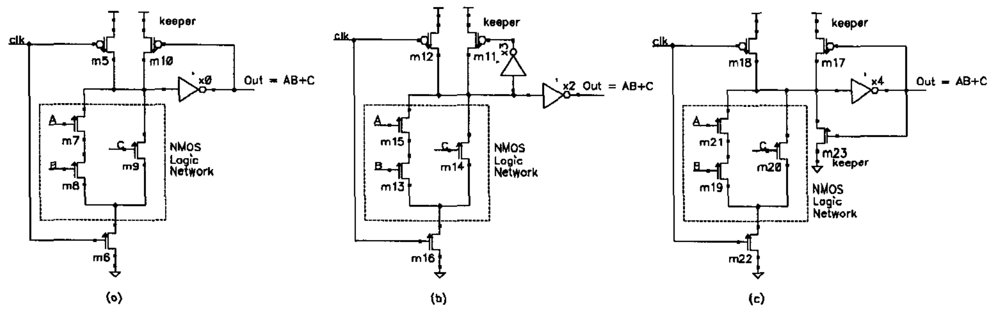

Figure 2.10: Dynamic gate keepers (a) feedback PMOS (b) isolated feedback PMOS (c) feedback PMOS/NMOS full keeper.

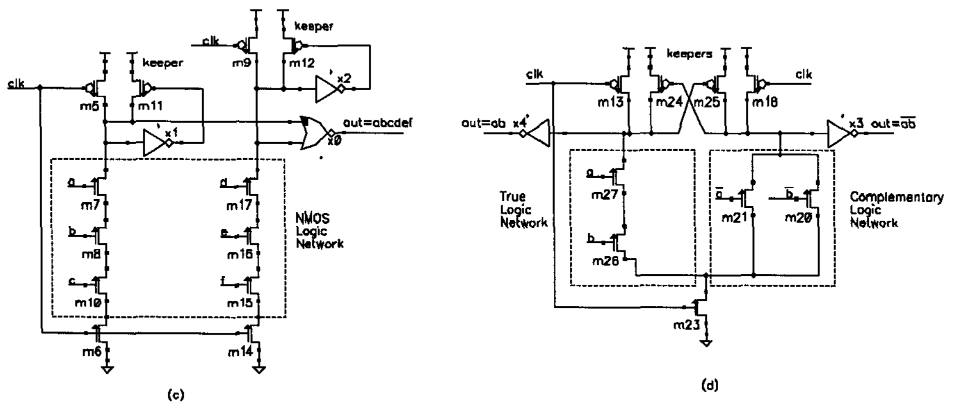

Figure 2.11: Dynamic gate keepers on (a) compound domino gate (b) dual-rail domino gate.

The keepers work by supplying a trickle of current to compensate for leakage on the dynamic node, thus keeping it high if that is the evaluated value of the logic gate. If the dynamic node is evaluated low, there is a brief period of contention as the keeper transistor and the pull-down network are on simultaneously. However, the pull-down network usually has much more conductance than the keeper and the keeper will eventually shut off when the trip point of the feedback inverter is reached. Nevertheless, this brief period of contention dissipates unwanted power and slows down the circuit. Furthermore, the presence of the keeper contributes capacitive load to the dynamic node. Therefore, keepers should be small to minimize fighting the dynamic gate when it switches and to minimize loading on the forward path. However, sizing the keeper just small enough to compensate for the leakage current, $I_{Leakage}$, does not take into account the very important use of keepers to alleviate charge sharing as discussed in the next section.

The keeper configuration of (b) isolates noise on the output from feeding back to the dynamic node although it also increases capacitive loading due to the extra inverter. The keeper configuration of (c) is required when the output could float either high or low during evaluation. This could happen for certain cases where the clock is stopped high.

As process technologies continue to shrink, the transistor subthreshold leakage is becoming increasingly high. The keepers must therefore be upsized to compensate for these leakage currents. One study [3] has shown that leakage currents have become so high in sub-130-nm technologies that conventional keepers must be too large. They instead propose a technique of using delay elements and logic gates called the "Conditional Keeper Technique" for combating the leakage.

2.3.2 Charge Sharing

As dynamic logic relies on charge storage on the output load and parasitic capacitors, this charge can sometimes be shared between the output node and an internal node when both capacitors are floating. Consider the domino gate of Fig. 2.12.

During the precharge phase, the inputs to the domino gate are all low and the output capacitor C_{Dyn} precharges high. If, during evaluation, the only input that makes a transition from a $0 \rightarrow 1$ is a, then there is no direct path to either supply rail and charge is shared between the floating capacitors C_{Dyn} and C_{Int}, assuming that capacitor C_{Int} has been discharged in a previous cycle. The magnitude of the charge sharing depends on the size of the internal capacitor C_{Int} compared with the output capacitor C_{Dyn}. If the value of C_{Int} is small enough, the voltage drop on Dyn will be less than the threshold voltage of an NMOS transistor. The final voltage on Dyn after charge sharing will then be equal to [8]:

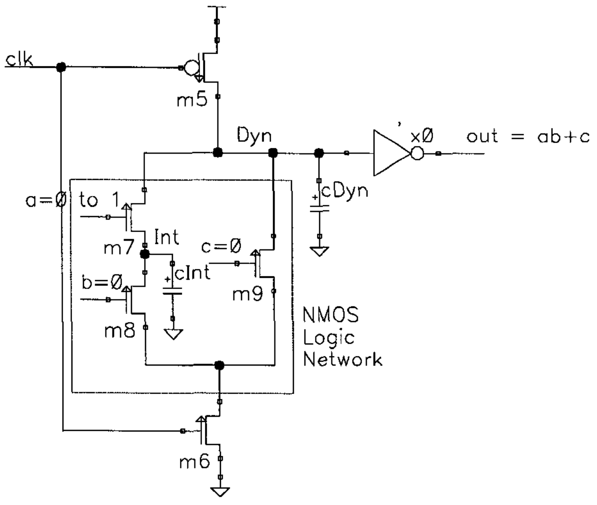

Figure 2.12: Charge sharing in dynamic gates.

$$V_{dyn} = V_{dd} - \frac{C_{Int}}{C_{Dyn}}(V_{dd} - V_{tn}). \tag{2.5}$$

If the value of C_{Int} was large enough, then the amount of charge sharing will be greater than a threshold voltage. The final voltage on Dyn after charge sharing would then be equal to:

$$V_{dyn} = V_{dd}\frac{C_{Dyn}}{C_{Dyn} + C_{Int}}. \tag{2.6}$$

Since Dyn is a dynamic node, the voltage drop cannot be recovered and might lead to logic failures if it drops past the switch point of the output static gate. Decreasing the β ratio of the output static gate would give the gate more noise tolerance although it will slow down the critical rising transition. Examining the timing of the input signals and reordering or duplicating the input transistors of the pull-down stack can sometimes be effective in reducing charge sharing [8, 32]. The solution of inserting keeper transistors as described in Section 2.3.1 to combat leakage is effective at reducing the effects of charge sharing as well. Here larger keeper transistors are more effective at preventing charge sharing at the cost of increasing the dynamic output load and the amount of contention during switching. Each noise event that activates the keepers will also dissipate power as well. In complex domino gates where many internal nodes can charge share with the output node, it may be required to precharge the internal nodes with secondary precharge transistors as shown in Fig. 2.13 [8]. Charge sharing is eliminated in this case since both nodes *Dyn* and *Int* have no potential difference when the input transistor *a* begins to conduct. Adding internal precharge transistors in this manner increases the clock load, circuit area and diffusion capacitance that slows the gate. Therefore, enough internal precharge transistors should only be added to meet prescribed noise budgets. Usually precharging every other node is sufficient [43]. One must also be careful when precharging internal nodes in unfooted domino gates since some of the input transistors might turn off slower and thus cause short circuit currents.

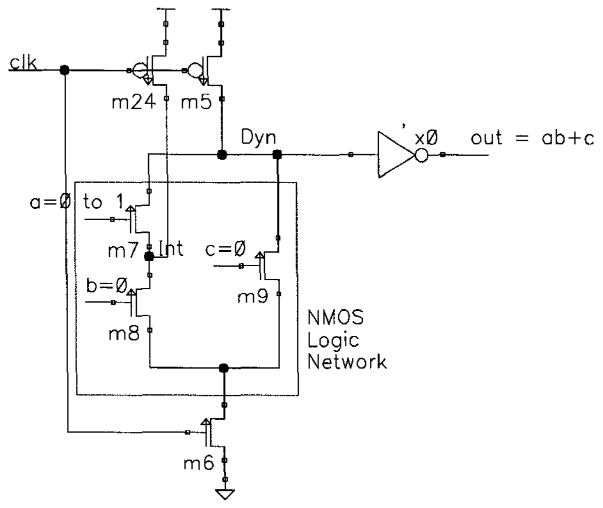

Figure 2.13: Precharging internal nodes.

2.3.3 Other Design Considerations

Other dynamic circuit design challenges will be discussed briefly since this dissertation does not improve upon previous work concerning these issues. For a more detailed discussion, the interested reader can consult available literature [8, 43, 61]. Dynamic nodes are more susceptible to failures caused by interconnect capacitive coupling or crosstalk. Furthermore, the inputs to dynamic gates also need special attention, since the noise margin is only an NMOS threshold voltage. However, these inputs are actively driven by static gates, thus reducing their risk. Crosstalk occurs when an unrelated signal wire running adjacent to the short dynamic node or dynamic input switches. The mutual capacitance between the "aggressor" and "victim" signal wires causes an addition or removal of charge from victim dynamic nodes or from the dynamic inputs. Methods to reduce interconnect coupling include: spacing the aggressor and victim signals further apart in the horizontal and vertical directions, shielding a victim laterally on one or both sides by supply rails [28], routing a victim node between two complementary aggressor nodes (effective for dual-rail domino logic) or through twisting and swapping routing tracks at regular intervals. Another form of coupling is the back-gate coupling that occurs for compound domino gates. When a dynamic gate drives a NAND structure, there is the possibility that the inputs further up in the NAND stack could switch while the lower inputs remain idle, causing a simultaneous voltage change on both sides of an input transistor. This will increase the capacitance of the dynamic node which in turn causes a voltage drop. Reordering the static gate input stack is sometimes helpful in resolving back-gate coupling. When a dynamic gate drives a NOR structure, output charge glitches can occur if the internal capacitance of the PMOS network is charged in a previous cycle and then the charge shared between the internal node and the static output. This glitch is problematic since the static output is immediately coupled to another dynamic input with a low noise threshold. A method that is effective at reducing this effect is gating the internal node with a transistor to ground where the transistor's discharge operation is controlled by the dynamic node [66]. Other dynamic circuit design challenges include the effects of power supply noise variation and IR drop, Miller coupling/clock feedthrough, residual noise, minority carrier charge injection and alpha particles. Because of all these noise sources, designers of dynamic circuits must adhere to strict noise budgets and put much effort into electrically verifying the circuits.

2.4 Clock Uncertainty

2.4.1 Circuit Pipelining

A synchronous digital system consists of cascaded banks of sequential memory elements with combinational logic between each set of memory elements. The functional requirements of the digital system are satisfied by the combinational logic. The performance of the system is determined by inserting clocked memory elements to segment the combinational logic so that the longest delays through each logic stage are constrained within the desired cycle time [37]. Furthermore, the shortest delays through the combinational logic must be considered since the inputs to memory elements should not change until a hold time after the sampling edge to avoid race conditions. Separating logic into stages with memory elements is called pipelining. The clock signal synchronizes the events in the synchronous system since the movement of data through various stages of the pipeline take place concurrently in response to the clock stimulus [8]. New sets of inputs are sampled by storage elements and new computations ensue in response to the clock thus changing the state of the sequential network. Once the computations are complete, the results must await the next clock transition in order to advance to the next pipeline stage. In other words, the next cycle of an operation cannot begin unless all the computations have completed. An alternative view to pipelining, that is useful in understanding latch-based and domino logic designs, is that memory elements between the logic stages enforce sequencing, distinguishing between $current$ from $previous$ and $next$, rather than remembering state [43]. If memory elements were not used, fast signals might race ahead and catch up with slow signals from a different operation resulting in meaningless results. Thus, an ideal memory element would slow down early signals while adding no delay to the signals that are already late. However, real memory elements generally delay the late signals as well, which is considered the sequencing overhead. It is desired that systems with the smallest possible sequencing overhead be built for maximum performance. Domino logic circuits can function as inherent memory elements when divided into pipeline stages and hence will enforce sequencing.

2.4.2 Skew

In a ideal synchronous system, all the clocks would arrive at all the memory elements or clocked logic gates at the same time. Unfortunately, because clocks are loaded with the greatest fanout, travel the longest distances, and operate at the highest speed of any signal in the system, ensuring that all the clocks arrive at the same time is often impossible. The interarrival time of two clock edges is the delay between the edges. Clock skew is the absolute difference between the nominal and actual interarrival times of a pair of physical clock edges [43], where a physical clock is the clock that is actually received by the memory element or clocked gate. Skew between a pair of clocks is shown Fig. 2.14, where the bold lines denote the nominal interarrival time of the clocks.

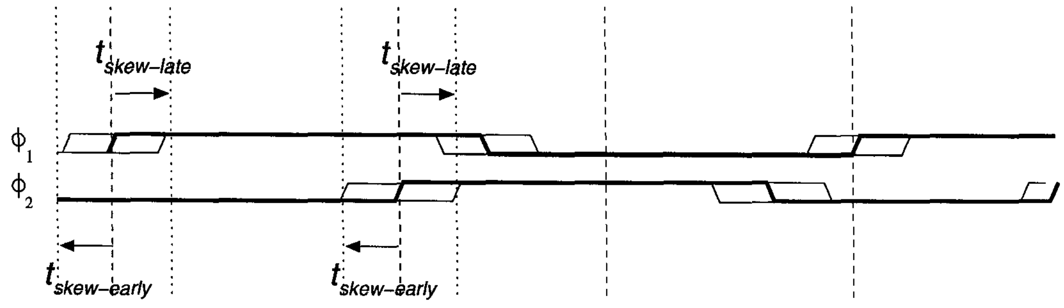

Figure 2.14: Definition of skew.

It is important to observe that only the skew between two sequentially adjacent memory elements or domino gates is relevant to the definition of skew. The skew between two non-sequentially adjacent memory elements or domino gates has no meaning since these two elements do not directly exchange data [37]. It should be noted that clock jitter, which is another type of clock uncertainty that affects microprocessor cycle times [27], is encompassed in our definition of skew. Normally jitter is defined as the time difference in the actual and expected transition of a single clock since the periodicity of a clock signal is affected by the deviation of its edges from their expected transition time [8]. Jitter is a characteristic of the clock generator and can be caused by supply voltage variations and mismatches in the clock phase-locked-loop (PLL) circuitry which generates most clocks in high-speed digital circuits. However, this definition of jitter and associated duty-cycle variations, which can be defined as the percentage of time that the clock is high as opposed to when it is low, is actually a source of skew and hence is a contributor rather than a separate variable. The causes of skew are many and can include [37, 42]:

1. Mismatches in path lengths from the clock source to the clocked memory elements or domino gates.

2. Mismatches in passive interconnect parameters; such as line resistivity of different metal layers, line capacitance of different width metals, interconnect coupling capac-

itance, contact resistance etc.

3. Differences in active device processing parameters; such as variations in transistor threshold voltages and channel mobilities which affect the delays of the repeaters or buffers within the clock distribution network.

4. Environmental variations that cause repeater and buffer delay variations such as voltage and temperature

For a well-designed and balanced clock distribution network, the distributed clock repeaters and buffers are the principal cause of clock skew.

2.5 Skew-Tolerant Domino Logic

2.5.1 Skew-tolerant Pipelines

Skew-tolerant domino circuits remove the three sources of sequencing overhead found in traditional latch-based domino pipelines: clock skew, latch overhead and pipeline imbalances. This is accomplished through using overlapping clock phases to different stages of domino logic [42, 43]. The use of overlapping clocks eliminates the need to budget clock skew in the cycle time since data can now arrive and depart from different pipeline stages irrespective of modest variations in the arrival time of the clocks. Furthermore, since the overlapping clocks allow time for the first gate of a phase to evaluate before the last gate of the previous phase precharges, latches are eliminated from the pipeline as domino gates function as inherent latches [70]. Finally, if the overlap between clock phases is larger than the worst-case clock skew, then domino gates can time borrow across stages. Gates in two adjacent phases can evaluate when their respective clocks are high and overlap, allowing gates that nominally evaluate during a first phase to run late into a second phase. Thus, removing all the sources of overhead allow the entire cycle time to be available for useful computation. The method of skew-tolerant domino logic will become more apparent in subsequent sections.

2.5.2 Definitions

We will follow the terminology as defined in [42, 43]. We consider N overlapping clocks. The clock cycle of period T_c is divided into N phases. Each phase rises T_c/N after the previous phase, and by symmetry all phases have the same duty cycle. Each phase is high for an evaluation period t_e and low for a precharge period t_p as shown in Fig. 2.15. Although we show a three-phase system in all of our examples, the results are applicable to N phase systems. Therefore, given these N clocks, it is possible to derive waveforms which maximize the tolerable skew and amount of time borrowing in an N-phase system, independent of the actual logic contained in the phases. In the prior art skew-tolerant pipeline, each dynamic gate in a phase receives one clock for both precharge and evaluate.

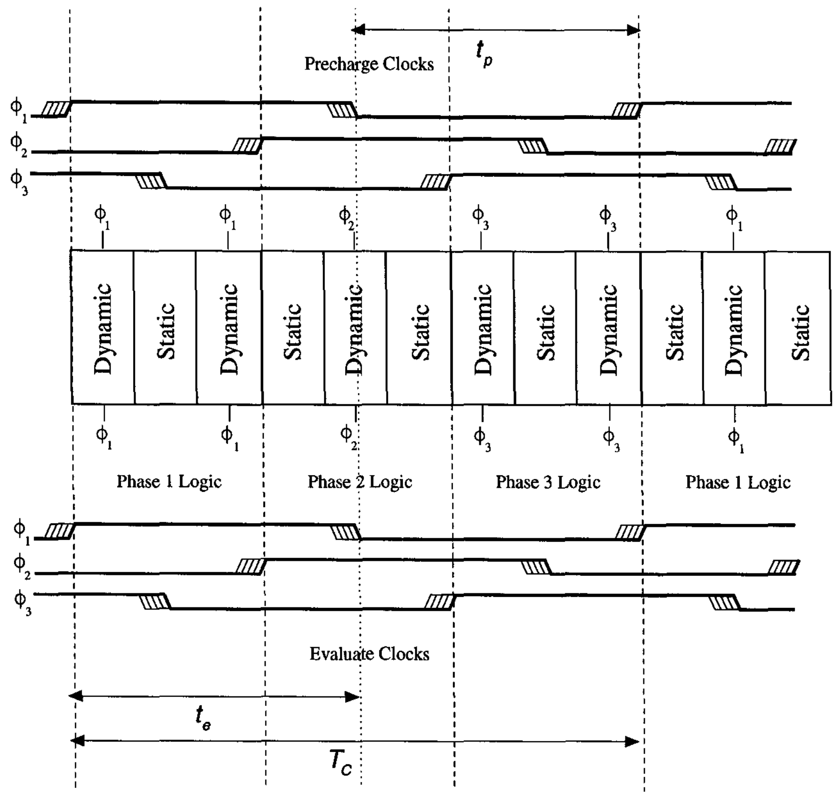

Figure 2.15: Skew-tolerant domino with same clocks for precharge and evaluate.

A domino gate consists of a dynamic gate followed by a static gate, where the static gate can be an inverter or a complex static CMOS logic gate as in [45]. From the timing waveforms, it is assumed that logic in a phase begins evaluating at the latest rising edge of its clock and continues for T_c/N until the next phase begins. The nominal logic evaluation time for a phase is therefore equal to T_c/N. When two consecutive clock phases overlap, the logic of the first phase may run late into the time nominally allocated to the second phase. The maximum amount of time that can be borrowed depends on the amount of nominal overlap between consecutive clock phases. This nominal overlap in turn is affected by the amount and type of clock skew between two consecutive clock phases. We will define skew in a multi-phase system as measured backward from the rising edge of a second clock phase to the falling edge of a first clock phase in the region where the two clock phases overlap.

A pair of clocks is positively skewed if the amount of overlap decreases from the nominal value, $t_{overlap}$ as shown in Fig. 2.16. In this case, the second clock phase arrives late by t_{skew} with respect to the first clock phase.

A pair of clocks is negatively skewed if the amount of overlap increases from the nominal value, $t_{overlap}$, as shown in Fig. 2.17. In this case, the second clock phase arrives early by t_{skew} with respect to the first clock phase.

In our example three phase skew-tolerant domino, positive skew between adjacent clocks Φ_2 wrt Φ_1 gives the Φ_2 logic less time to evaluate when Φ_1 logic is time-borrowing from Φ_2 while negative skew gives Φ_3 logic less time to evaluate when Φ_2 logic is time-borrowing from Φ_3. Assuming that the first clock arrives at the latest possible rising edge, both types of skew effectively reduce the amount of nominal overlap that can be used for skew tolerance and time borrowing.

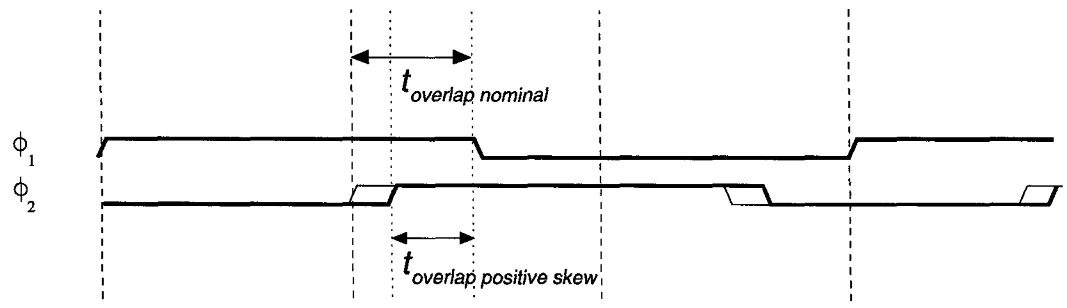

Figure 2.16: Positive skew in the context of multiple overlapping clocks.

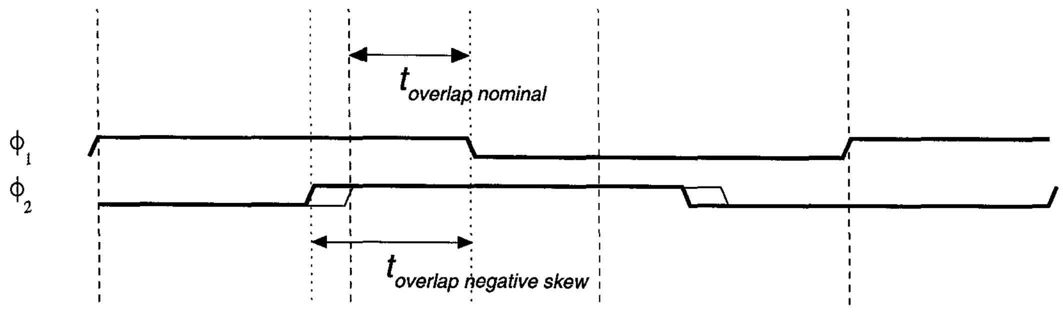

Figure 2.17: Negative skew in the context of multiple overlapping clocks.

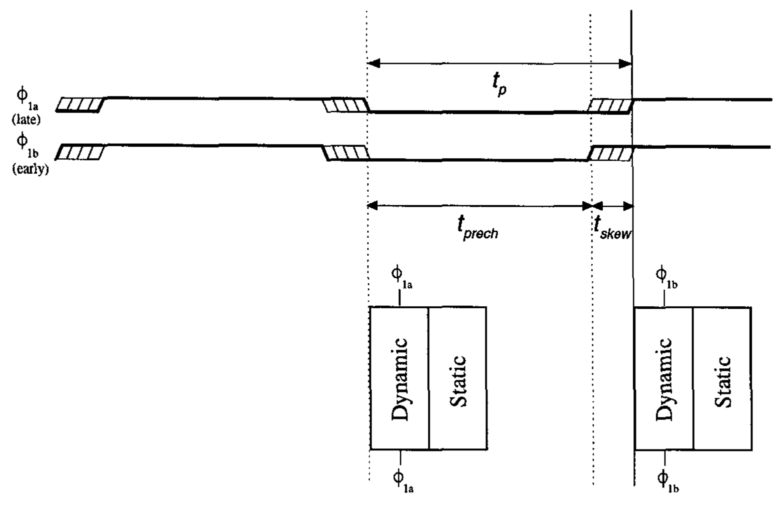

Figure 2.18: Precharge time constraint.

2.5.3 Precharge and Evaluate

It was shown in [42] that for skew-tolerant domino, the precharge time, t_p is limited by the rate at which two consecutive gates in the same phase precharge. Assuming that a first dynamic gate has its outputs coupled to the inputs of a first static gate and the first static gate has its outputs coupled to the inputs of a second dynamic gate as in Fig. 2.18, the constraint requires that the first dynamic gate precharge fully and discharge the output of the first static gate below V_t by some noise margin, before the second dynamic gate enters evaluation. This is so the first static gate does not cause the second dynamic gate to incorrectly evaluate old data. This time, denoted by t_{prech}, assumes its worst case when clock skew within a phase causes the clock to the first dynamic gate to arrive at the latest possible time and the clock to the second dynamic gate to arrive at the earliest possible time thus reducing the effective precharge window by t_{skew}. Therefore, the lower bound to guarantee proper precharge is equal to:

$$t_p \geq t_{prech} + t_{skew}. \tag{2.7}$$

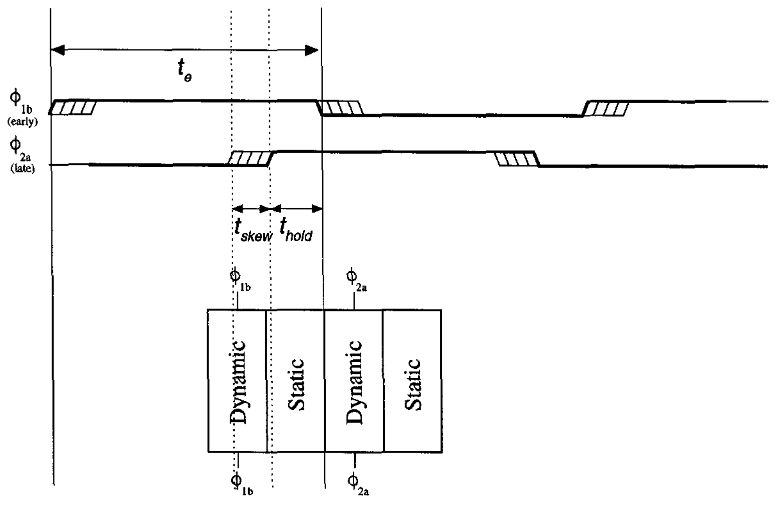

Figure 2.19: Evaluation time constraint.

Similarly, it was shown in [42] that, t_e, is set by required overlap between adjacent clock phases, since a phase must remain in evaluation until the following phase consumes the data. Thus, the worst case occurs when the clock to the last dynamic gate of a first phase arrives at the earliest possible time and the clock to the first dynamic gate of a second phase arrives at the latest possible time as shown in Fig. 2.19. This necessary overlap, t_{hold}, is usually a small negative time since the first dynamic gate of the second phase evaluates immediately after its rising clock edge while the precharge must ripple through both the last dynamic gate and last static gate of the first phase. The precharge might take even longer to ripple through if non-footed dynamic gates are used for the last several dynamic gates gates of a phase. Moreover, domino logic is made to favor the critical falling edges of the dynamic gates and the critical rising edges of the static gates at the expense of slower precharge times, thus decreasing the hold time even more. A conservative number to use for t_{hold} is usually 0. Note that this hold time is not the same as the hold time associated with latches of flip-flops, where that hold time denotes the time that the data must be stable following a sampling or enabling clock edge. Budgeting in the effect of clock skew and adding the time shift of T_c/N between phases gives the lower bound to guarantee proper evaluation as:

$$t_e \geq \frac{T_c}{N} + t_{hold} + t_{skew}. \tag{2.8}$$

2.5.4 Skew Tolerance

Assuming all the available overlap between clock phases is used for robustness against clock skew, the constraints on precharge (2.7) and evaluate (2.8) can be combined to yield a formula for maximum allowable skew equal to [42, 43]:

$$t_{skew\_max} = \frac{\frac{N-1}{N}T_c - t_{hold} - t_{prech}}{2}. \tag{2.9}$$

From (2.9), it is apparent that the precharge window and hold time must be guaranteed independent of logic evaluation delays.

Since t_{skew} must be budgeted at the interface between static and domino logic, as the static output must be stable before the latest skewed evaluation clock arrives, building entire critical paths and loops in skew-tolerant domino logic often requires the use of dual-rail circuits to avoid inversions and non-monotonic behavior for single-ended signals [42]. Each skew-tolerant domino path should also contain at least one gate per phase to avoid potential race-through problems. Therefore, the outputs of a first phase domino gate should not couple to the inputs of a third phase domino gate without at least passing through a second phase domino buffer. However, this restriction normally would not limit the cycle time of a circuit since a path that does no work will usually not be the critical path.

2.5.5 Exactly One Dynamic and Static Gate Per Phase

For some high frequency designs, it might be appropriate to use exactly one domino gate per phase. In this case, the precharge constraint of (2.7) can be relaxed since the output of the static gate in the first phase must fall low by the time the second phase, rather than the first phase, reenters evaluation. The available precharge time therefore increases by T_c/N to give [42, 43]:

$$t_{skew\_max} = \frac{T_c - t_{hold} - t_{prech}}{2}. \tag{2.10}$$

2.5.6 Global and Local Skew

Clock domains, as used in this dissertation, are lumped hierarchies of clocks where pairs of clocks within the same local clock domain experience less skew than the skew seen by clocks in different global clock domains [43]. If more than two levels of clock domains exist, the higher-level clock domains see progressively more skew as delay variations in the global clock distribution network appear as skew. Thus, reducing the local skew within local clock domains increases the time available for global skew through the observation that: (1) the precharge constraint (2.7) becomes dependent only on local skew since precharge must complete before the next gate in the same phase resumes evaluation, (2) the evaluation constraint (2.8) remains dependent on global skew because clocks from different clock domains must still be overlapping. Thus the maximum tolerable global skew when global and local skews differ is equal to [42, 43]:

$$t_{skew\_max\_global} = \frac{N-1}{N}T_c - t_{hold} - t_{prech} - t_{skew\_local}. \tag{2.11}$$

2.5.7 Time Borrowing

When two adjacent clocks overlap, that overlapping region $t_{overlap}$ can be used for skew tolerance as discussed above, or for time borrowing as shown in Fig. 2.20. Time borrowing happens when all the gates in a phase of logic take longer than T_c/N to evaluate. When that happens, evaluation for the current phase of logic can run into the time nominally allocated to the next phase of logic up until the end of the overlapping region. The gates of the next phase must wait for the gates of the current phase to finish evaluating but do not have to block the clock since domino logic makes a monotonic low-to-high transition during evaluation and therefore the gates of the next phase cannot be unintentionally corrupted.

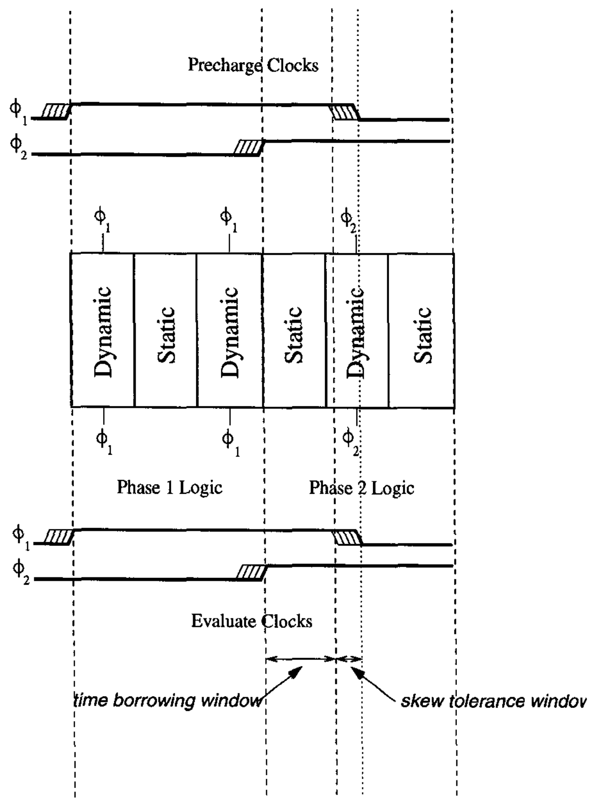

Figure 2.20: Time borrowing and skew tolerance windows for overlapping clocks.

The time actually borrowed from the next phase, denoted by t_{borrow}, will reduce the nominal time that the next phase has to evaluate by $T_c/N - t_{borrow}$. Time borrowing, however, may take place over several phases thus alleviating the time borrowing penalty. For example if a first phase evaluates for a duration of $T_c/N + t_{borrow}$, a second phase may actually evaluate for a time between $T_c/N - t_{borrow}$ and T_c/N, since the second phase can also borrow time from a third phase. This can go on indefinitely as long as the results of a current stage are consumed by the next stage before a current stage precharges. Thus balancing pipeline stages to increase clock frequency, which can be extremely difficult for other design styles, becomes easier. Furthermore, time-borrowing is useful because it automatically helps to compensate for environmental conditions and process variations across the die when positive and negative conditions/variations exist, and for inaccuracies in the modeling and simulation of the digital circuits. Last, domino circuits actually run faster with time borrowing into each stage, since the stage that is borrowed from would have its clock high and discharged the internal node capacitance at the drain of the evaluation transistor before the data arrives. This is called a flow-through condition for domino gates or transparent latches. It is interesting to note that if a phase of logic completes before the nominal logic evaluation time of T_c/N, that the time between the end of evaluation and the rise of the next phase clock is wasted. Thus, a hard edge is imposed which can only be removed through asynchronous means beyond the scope of this work [70]. If all the available overlap between phases is used for time-borrowing, the maximum amount of time that can be borrowed is:

$$t_{borrow\_max} = t_e - \frac{T_c}{N} - t_{skew}. \tag{2.12}$$

As time borrowing and skew tolerance trade off directly, the time allotted to time borrowing as derived from (2.11), for global and local clock domains is equal to [42, 43]:

$$t_{borrow} = \frac{N-1}{N} T_c - t_{hold} - t_{prech} - t_{skew\_local} - t_{skew\_global}. \tag{2.13}$$

In (2.13) t_{skew_local} is the skew between two directly coupled domino gates in a local clock domain while t_{skew_global} is the skew between two directly coupled domino gates from different local clock domains.

2.5.8 Racethrough or Min-Delay

Domino logic pipelines have constraints for minimum delay, that is data departing from one clocked element as early as possible must not arrive at the next clocked element until a contamination delay time, t_{cd}, after the sampling/falling edge of the next element. Otherwise, the second clocked element would have latched or evaluated the first clocked element's input rather than its own input. In skew-tolerant domino, min-delay failure occurs when a rising edge clock arrives early at the first phase dynamic gates and a falling edge clock arrives late at the second phase dynamic gates. If the first phase logic is very fast or the logic depth is small, data may arrive at the second phase gates a cycle early thereby contaminating the previous cycle results of those gates, since the second phase gates should not have received the data until after the next rising edge of the second phase clock. An example of this is shown in Fig. 2.21.

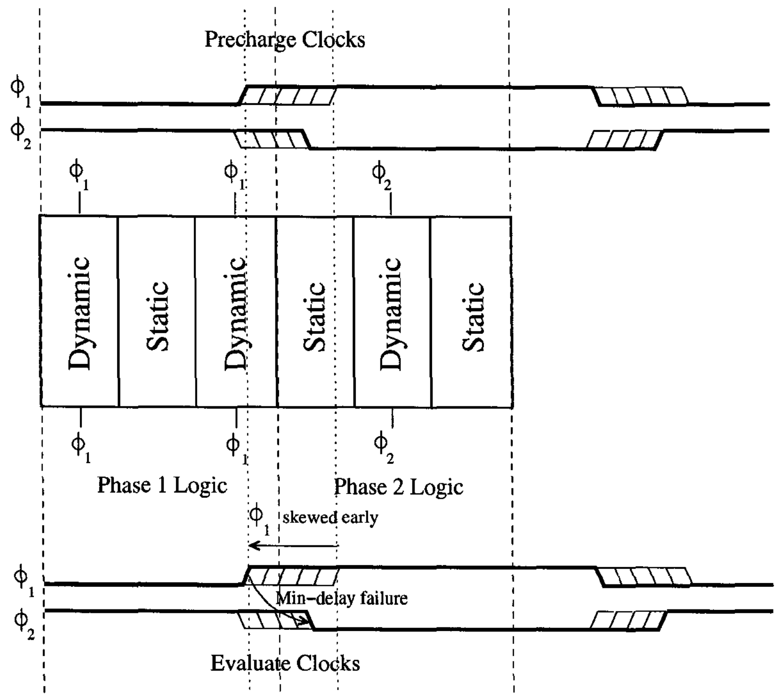

Figure 2.21: Min-delay failure for overlapping clocks.

In practice, the value of t_{cd} is usually a small positive number since the improper result must propagate through the first phase logic before corrupting the second phase logic. Since the nominal overlap between the first phase rising and the second phase falling is:

$$t_{min\_overlap} = \frac{N-1}{N}T_c - t_e, \tag{2.14}$$

the maximum global skew that can be tolerated, while ignoring the effects of the precharge time constraint (2.7) and the evaluation time constraint (2.8) is [42]:

$$t_{skew\_max\_global} < \frac{N-1}{N}T_c - t_e + t_{cd}. \tag{2.15}$$

If the expression (2.15) is equal to zero, the domino system cannot tolerate any skew. Furthermore, if the expression (2.15) is less than zero, the domino system cannot function. The selection of the number of clock phases and duty cycle therefore can affect the maximum tolerable skew. For example, a three phase system with 50% duty-cycle clocks can tolerate 1/6 cycle of skew assuming $t_{cd} = 0$ while not considering the effect of precharge or evaluate constraints. Two-phase skew-tolerant domino systems with 50% duty cycles can tolerate no skew, assuming again that $t_{cd} = 0$ while not considering the effect of precharge or evaluate constraints. To circumvent this problem, extra non-overlapping clocks may be used for the first domino gate in each phase as in [20]. However, there may exist min-delay problems between the extra non-overlapping clocks and the normal clocks. A solution, used in two commercial processors, was to delay the normal clocks relative to the non-overlapping clocks [29, 62] and ensure that at least one of the later gates of a phase is clocked by the delayed clock as shown in Fig. 2.22.

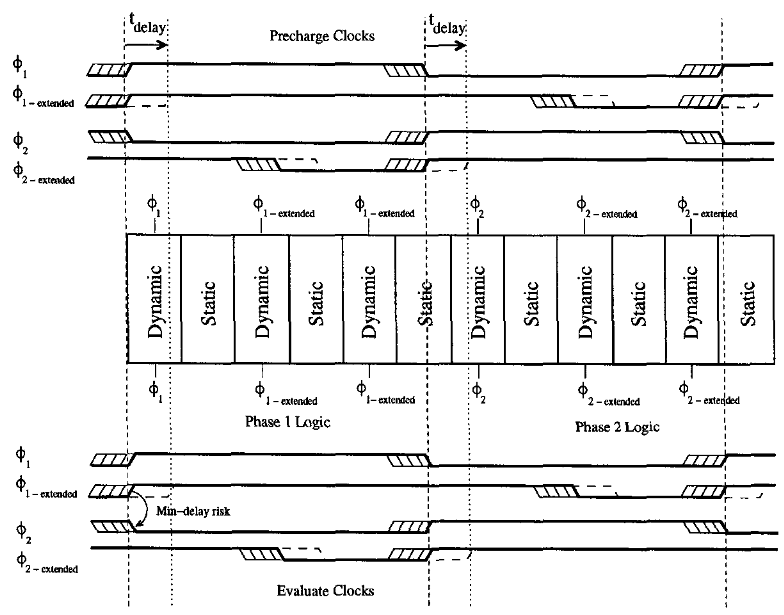

Figure 2.22: OTB domino logic min-delay risk and solution through extended clock delay.

The skew that can be tolerated for this opportunistic time-borrowing (OTB) domino is:

$$t_{skew\_max\_global\_OTB} < \frac{N-1}{N} T_c - t_e + t_{cd} + t_{delay}. \tag{2.16}$$

t_{delay} is defined as the time from the rising edge of the non-overlapping clock of the first gate of a phase to the rising edge of the overlapping clock of the remaining gates of a phase.

2.6 Summary

In this chapter we introduced dynamic logic circuits and explained why they are faster, but normally dissipate more power, than static CMOS circuits. We also showed that designing domino circuits is often more difficult since issues with leakage, charge sharing and other noise sources that affect floating dynamic nodes must be taken into consideration. Further, we have defined clock skew and explained why they pose a problem for dynamic circuit designs since domino logic is synchronized by clock signals. Last we have discussed how skew-tolerant domino logic, and its derivatives such as Opportunistic Time-Borrowing domino logic, can be pipelined to withstand fair amounts of clock skew, eliminate latching overhead and allow time borrowing to balance pipeline stages.

Chapter 3

Method for Single-Rail All-Domino Pipelines

3.1 Introduction

Domino logic circuits are often used in microprocessor critical paths because of their 1.5 to 2 times speed improvement over static CMOS gates [43]. Despite their wide application to microprocessor design, single-rail domino is not functionally complete because of its inability to perform inversions [61].

There are many situations, however, where inverting or non-monotonic logic needs to be used in conjunction with non-inverting/monotonic logic. These include multiplexers, parity circuits, and arithmetic units which depend heavily on XOR and XNOR functions. Normally, a designer must use slower logic styles such as static CMOS or transmission gates to implement inverting and non-monotonic functions with the additional cost of increased overhead to interface from dynamic to static logic and back. Some designs have also been known to use clock-blocking techniques that require the clock to be the last input signal to arrive at a dynamic gate after the data inputs so that non-inverting and monotonic functions are possible, such as clock-delayed (CD)-domino [71, 72]. However, these clock-blocking techniques require precise matching of data and clock delays which have to be accounted for under all possible process and environmental corners. Furthermore, clock skew must be budgeted at each clock-blocking gate, making this logic family skew-intolerant. Last, scaling of such designs would require complete re-verification of the data and clock delay paths. For designs where speed is the most critical design parameter, Domino Dynamic Cascode

Voltage Switch Logic (DCVSL)/dual-rail domino circuits [58] can be used to meet the requirements for inverting and non-monotonic functions. Such circuits require approximately double the number of transistors compared to single-rail domino logic, resulting in greatly increased routing complexity, circuit area and in many cases, decreased circuit speed due to longer differential routing lines. Furthermore, dual-rail domino circuits dissipate more power (approximately double) than single-rail domino because of their increased routing capacitance and unity activity factor.

This chapter proposes an alternative to these approaches which provide inverting or non-inverting outputs in a single-rail domino pipeline. This technique involves using logic functions of overlapping clocks so that dynamic gates do not receive the same clock for precharge and evaluate. This method, known as Clock-Logic (CL) domino, is robust against clock skew and allows time borrowing across pipeline stages as in skew-tolerant domino logic [42, 43]. This method allows complete data and control paths to be built from single-rail domino logic thereby improving speed over mixed domino/static techniques. This is accomplished while minimizing power consumption and area overhead and maintaining approximately equal speed compared to dual-rail differential domino.

This chapter is organized as follows: Section (3.2) derives the basic timing constraints and gives practical examples of Clock-Logic (CL)-domino. Section (3.3) shows how the extra clocks required for CL-domino do not have to be distributed but rather generated locally at the dynamic gates. Section (3.4) examines how locally generating the CL-domino clocks can sometimes slow a domino gate and practical methods to deal with the added delay as well as to prevent excessive charge sharing. Section (3.5) shows how a CL-domino can be used with normal skew-tolerant domino to yield a CL-domino pipeline with minimum delay. Finally, Section (3.6) shows that multiple phase clocks can easily be generated from a single distributed clock originating from a clock source such as a phase locked loop (PLL).

3.2 Clock Logic (CL)-Domino Timing

Clock logic (CL)-domino is built upon the observation that the clocks used to precharge and evaluate a dynamic gate need not be the same. Instead logic functions derived from multiple-phase clocks, can be used for separately precharging and evaluating domino logic stages. Through adherence to certain rules based on those derived previously for skew-tolerant domino circuits, entire microprocessor critical loops can be built in single-rail dynamic logic that supports skew-tolerance and time-borrowing thus saving circuit area and power while minimizing circuit delay due to reduced wiring parasitics. We will show that these asymmetric clocks can be generated at the transistor level at each domino gate.

3.2.1 Precharge Problem for Inverting and Non-Monotonic Domino Logic

If inverting functions (some inputs to the first dynamic gate of a phase are complemented) or non-monotonic functions are used inside a domino pipeline with multi-phase clocks, the inverting or non-monotonic functions will be corrupted when the the previous phase precharges. For example, in Fig. 3.1, the inverting stage in phase 2 will be corrupted when phase 1 precharges and likewise the inverting stage in phase 3 will be corrupted when phase 2 precharges. This is because an inverting function of the previous phase might cause a $0 \rightarrow 1$ transition on the input of the current phase in the middle of the evaluate cycle where the input to the current phase should have remained at 0 as it was at the start of the evaluate cycle. This is illustrated in Fig. 3.2 for the case of two AND gates in adjacent phases, where one of the inputs to the second AND gate is complemented. In the case of a non-monotonic function, the inputs to the dynamic gate will change before the end of the current evaluate cycle and the output might no longer maintain the correct result. Such a logic function where an inversion exists at the input of a dynamic gate or the gate implements non-monotonic logic will hereafter be referred to as an input complemented or non-monotonic dynamic logic function.

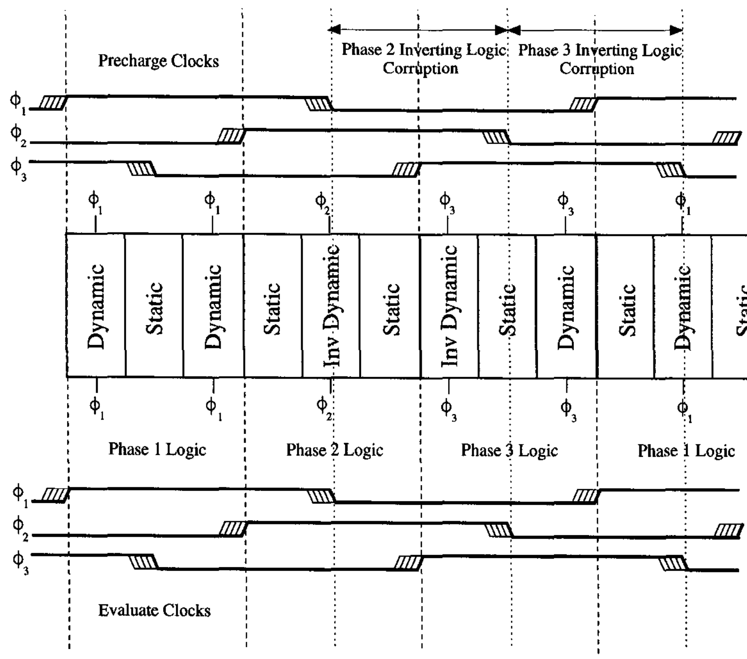

Figure 3.1: Inverting domino logic failure for overlapping clocks.

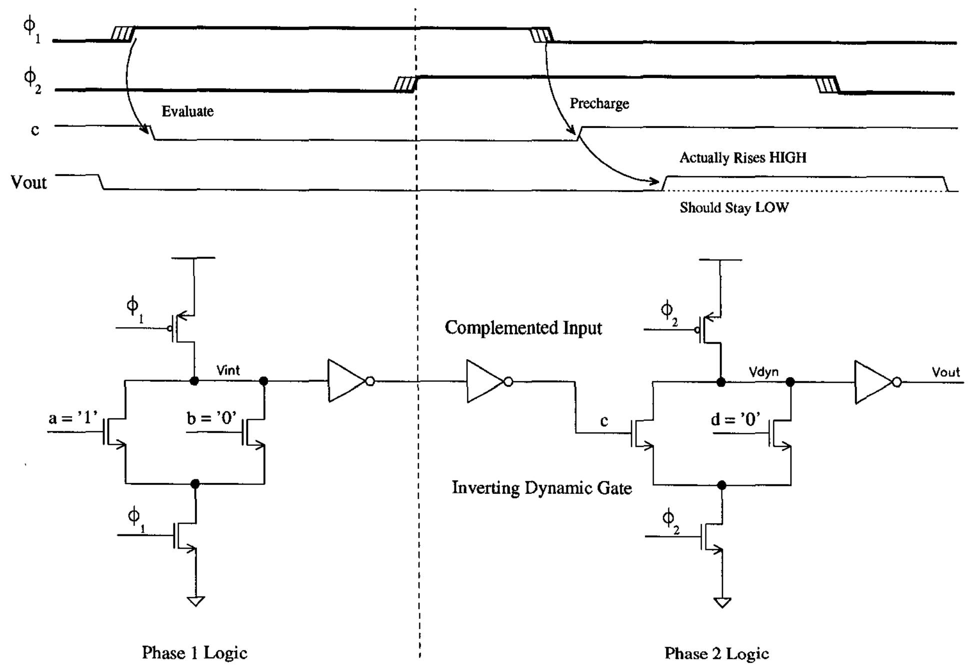

Figure 3.2: Example of inverting domino logic failure for overlapping clocks.

3.2.2 OR-Precharge/Domino-Evaluate

Our first solution for all single-rail domino pipelines is to delay the precharge of the previous phase until the end of the evaluation period of the current phase. This can be accomplished through extending the duty-cycle of the precharge clocks in the previous phase up until the time that the evaluation period for the current phase ends and its evaluation clock goes low as shown in Fig. 3.3, where *Dynamic** is an input complemented function, a non-monotonic function, or a regular dynamic logic function. *Dynamic* on the other hand is a regular dynamic logic function. We call this first embodiment of Clock-Logic domino "OR-Precharge / Domino-Evaluate" since the precharge clocks are simply a logical OR function of the precharge/evaluate clocks used in regular skew-tolerant domino logic. A proprietary design technique for multi-phase clocks using a form of OR-precharge was published in [1], although no systematic analysis or design considerations were disclosed or explored.

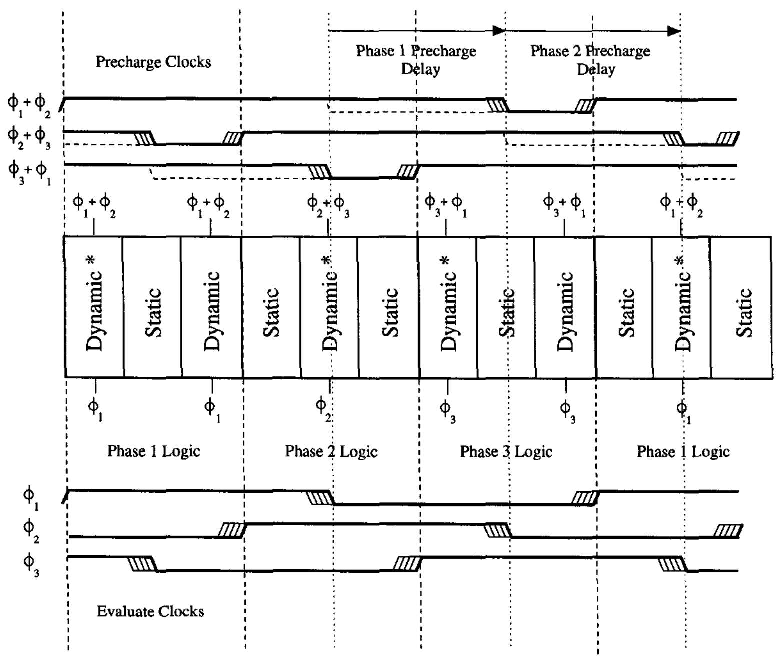

Figure 3.3: Clock-logic OR-precharge.

Inverting or non-monotonic functions can only be placed at the phase boundaries since the remaining domino gates in a phase must still only make a monotonic $0 \rightarrow 1$ transition during its evaluation period. Furthermore, if *Dynamic** implements an input complemented or non-monotonic function, the previous phase logic must now finish evaluation (be stable) by the time the *Dynamic** evaluation clock rises, thus imposing a hard edge on the data much like the setup time required for flip-flops. Therefore, when clock skew is accounted for, the previous phase of logic has a logic evaluation constraint of:

$$t_{eval\_prev} \leq \frac{T_c}{N} - t_{skew}. \tag{3.1}$$

A proprietary technique for domino logic establishing that (3.1) must hold true was published in [12]. It should be noted that only the phase before the input complemented *Dynamic** has this evaluation constraint. This is just another way of stating that the previous phase cannot borrow time from the phase that implements an input complemented function. Furthermore, a skew penalty must be paid by an input complemented or non-monotonic function at the phase boundary. Skew-tolerance and time borrowing, however, can occur normally, as in skew-tolerant domino, if the *Dynamic** gate implements a normal dynamic logic function.

Since the precharge clock duty-cycle has been increased from its nominal value, there is less time for the domino gates to precharge. The direction of the skew between two adjacent clock phases Φ_2 wrt Φ_1 affects precharge times as follows: negative skew gives the Φ_2 logic less time to precharge while giving the Φ_1 logic more time to precharge and positive skew gives the Φ_2 logic more time to precharge and the Φ_1 logic less time to precharge. For simplicity, both directions of skew effectively reduce the available precharge time by T_c/N. The maximum skew tolerable decreases to:

$$t_{skew\_max\_OR\_pre} = \frac{\frac{N-2}{N}T_c - t_{hold} - t_{prech}}{2}. \tag{3.2}$$

The minimum number of phases in CL OR-precharge domino logic is three so that the required number of precharge clocks are generated.

For the case of exactly one domino gate per phase the maximum tolerable skew is:

$$t_{skew\_max\_OR\_pre} = \frac{\frac{N-1}{N}T_c - t_{hold} - t_{prech}}{2}. \tag{3.3}$$

If we consider global and local clock domains, the maximum global skew tolerable decreases to:

$$t_{skew\_max\_global\_OR\_pre} = \frac{N-2}{N} T_c - t_{hold} - t_{prech} - t_{skew\_local}. \tag{3.4}$$

The maximum time available for time borrowing does not change from (2.12) since the evaluation is unaffected by the change in the precharge clocks. However, the time available for time-borrowing, while trading off skew-tolerance and taking into account global and local clock domains is reduced to:

$$t_{borrow\_OR\_pre} = \frac{N-2}{N} T_c - t_{hold} - t_{prech} - t_{skew\_local} - t_{skew\_global}. \tag{3.5}$$

In regards to min-delay failure (2.15) still holds since the evaluate clocks remain unchanged from normal skew-tolerant domino logic. However, for domino systems clocked strictly with OR-precharge, the system will fail due to a violated precharge time constraint, as described above, before it will fail due to any racethrough conditions.

3.2.3 Domino-Precharge/AND-Evaluate

Our second solution for all single-rail domino pipelines is to end the evaluation of a phase early. This can be accomplished through limiting the period of evaluation from the time that the current phase clock rises up until the time that the previous phase clock goes low as shown in Fig. 3.4, where *Dynamic** is an input complemented function, a non-monotonic function, or a regular dynamic logic function. *Dynamic* on the other hand is a regular dynamic logic function. We call this second embodiment of Clock-Logic domino "Domino-Precharge / AND-Evaluate" since the evaluate clocks are simply a logical AND function of the precharge/evaluate clocks used in regular skew-tolerant domino logic. A proprietary design technique which uses a form of AND-evaluate with a single-phase clock has been published in [7].

The constraint that input complemented or non-monotonic functions be placed only at phase boundaries, applies for AND-evaluate as it does for OR-precharge. The constraint that the previous phase complete evaluation before the rising edge of an input complemented or non-monotonic *Dynamic** is enforced automatically because evaluation ends early. The nominal evaluate time for a phase of domino logic is equal to:

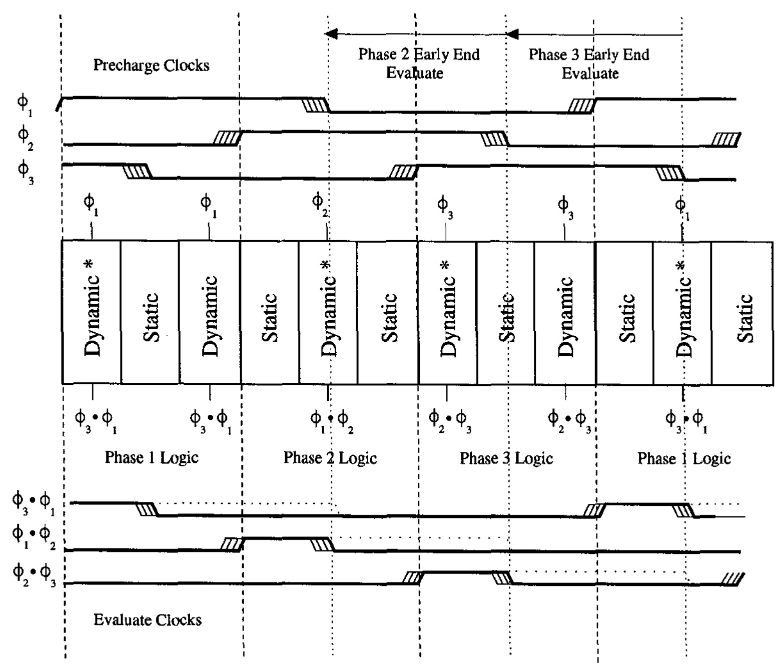

Figure 3.4: Clock-logic AND-evaluate.

$$t_{AND\_eval\_cycle} = t_e - \frac{T_c}{N} - 2t_{skew}. \tag{3.6}$$

This is less than the nominal evaluate time for a phase of skew-tolerant domino logic without time borrowing, $T_c/N - t_{skew}$. Furthermore, time-borrowing cannot occur for domino gates that are clocked strictly by AND-evaluation even if *Dynamic** implements a regular dynamic logic function. However, for practical designs, AND-evaluate will never be used exclusively without also incorporating other CL-domino or skew-tolerant domino thus alleviating the evaluation time and non-time borrowing penalties. Thus the only important constraint is that no time can be borrowed from AND-Evaluate *Dynamic** stages that implement input complemented or non-monotonic functions.

Since the evaluate clock duty-cycle has been decreased from its nominal value, there is less time for the domino gates to evaluate. The direction of the skew between two adjacent clock phases Φ_2 wrt Φ_1 affects evaluate times as follows: negative skew gives the Φ_2 logic more time to evaluate while giving the Φ_3 logic less time to evaluate and positive skew gives the Φ_2 logic less time to evaluate and the Φ_3 logic more time to evaluate. Clock skew will eventually cause a domino CL AND-evaluate stage to fail when there is no more overlap between adjacent clock phases. Therefore, the maximum tolerable skew is:

$$t_{skew\_max\_AND\_eval} = \frac{\frac{N-1}{N}T_c - t_{hold} - t_{prech}}{2}. \tag{3.7}$$

Note that this is the same equation as that derived for normal skew-tolerant domino logic.

The minimum number of phases in CL AND-evaluate domino logic is three so that the required number of evaluate clocks are generated and the results passed down through the logic stages.

For the case of exactly one domino gate per phase, the maximum tolerable skew is:

$$t_{skew\_max\_AND\_eval} = \frac{T_c - t_{hold} - t_{prech}}{2}. \tag{3.8}$$

This equation is again the same as that derived for normal skew-tolerant domino logic.

When comparing OR-precharge with AND-evaluate, AND-evaluate offers more skew tolerance because the precharge operation to the dynamic gates is unaffected. However,the amount of useful time per cycle is reduced from OR-precharge.

As is the case for normal skew-tolerant domino logic, reducing the local skew can be used to increase the maximum tolerable global skew according to:

$$t_{skew\_max\_global\_AND\_eval} = \frac{N-1}{N}T_c - t_{hold} - t_{prech} - t_{skew\_local}. \tag{3.9}$$

As mentioned previously, time borrowing cannot occur across phases for CL-domino gates clocked strictly by AND-evaluate. However, we will see that this constraint will be relaxed for more general CL-domino pipelines. A min-delay condition can occur if a rising edge of a first clock phase results in three consecutive clock phases being high simultaneously. This occurs under the same conditions as that for normal skew-tolerant domino logic (2.15). In the case of AND-evaluate, failure due to min-delay will occur first before a violated precharge time constraint.

3.2.4 OR-Precharge/AND-Evaluate

Our third embodiment of CL-domino combines OR-precharge with AND-evaluate to eliminate spurious transitions on the inputs of input complemented or non-monotonic functions, when that gate is holding the evaluated data and doing no useful work. Spurious input transitions can cause charge-sharing which may result in a change of the evaluated logic state. Unwanted input transitions can occur for domino-precharge/AND-evaluate when the previous phase is precharging and the current phase has finished evaluating thus leaving the output floating high. The implementation of "OR-precharge/AND-evaluate" is shown in Fig. 3.5.

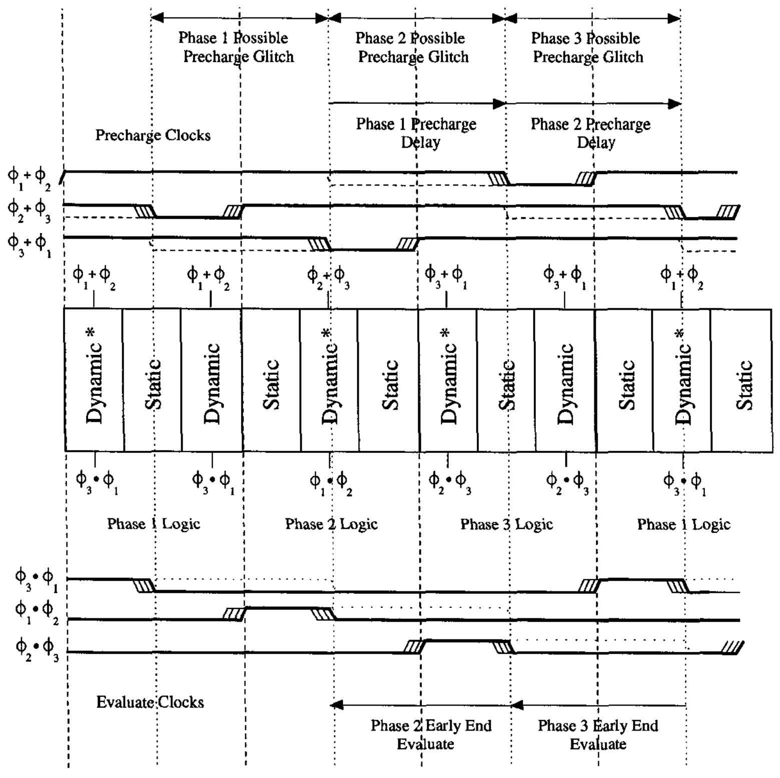

Figure 3.5: Clock-logic OR-precharge/AND-evaluate.

Since this style of CL-domino combines OR-precharge with AND-evaluate, the equations derived for the skew-tolerance of OR-precharge and AND-evaluate are applicable. We therefore, will take the worst case of the two CL-domino derivatives. Therefore the constraints on skew tolerance for OR-precharge / Domino-evaluate will apply.

Furthermore, as for AND-evaluate, time borrowing cannot occur across phases for CL-domino gates clocked strictly by OR-precharge/AND-evaluate. However, we will see that this constraint will be relaxed for more general CL-domino pipelines. A min-delay condition can occur if a rising edge of a first clock phase results in three consecutive clock phases being high simultaneously. This occurs under the same conditions as that for normal skew-tolerant domino logic (2.15).

3.2.5 OR-Precharge/OR-Evaluate

Our fourth solution for all single-rail domino pipelines takes advantage of the fact that the precharge operation for a phase of domino logic occurs for all gates in parallel while evaluation happens in series. Because series evaluation is usually the critical path, it is reasonable to allocate a larger portion of the cycle for evaluation. This means that the duty cycle of the evaluate clocks should be increased. However, from (2.15), we know that evaluate clocks with large duty cycles will often fail due to min-delay failures. Conversely, long duty cycle clocks will not be able to tolerate much skew or afford much time-borrowing. A technique similar to using extra non-overlapping clocks to the first domino gate of each phase as in [20] is shown for CL-domino in Fig. 3.6.

The extra clocks, denoted by $\Phi_1 \rightarrow \Phi 3$, are just the normal clocks used in skew-tolerant domino, and the extended evaluate clocks are just the logical OR of two consecutive clock phases. We call this fourth embodiment of Clock-Logic domino "OR-Precharge / OR-Evaluate", since the precharge/evaluate clocks, with the exception of the extra clocks, are a logical OR of the precharge/evaluate clocks used in regular skew-tolerant domino logic.

As in previous embodiments, *Dynamic** can implement an input complemented function, a non-monotonic function, or a regular dynamic logic function. *Dynamic*, on the other hand, is a regular dynamic logic function.

The constraint that input complemented or non-monotonic functions be placed only at the phase boundaries applies as before. Any phase before an an input complemented of non-monotonic *Dynamic** cannot borrow time from that phase. Unlike the solutions proposed for skew-tolerant OTB domino logic as described in [29, 62] and in Section 2.5.8, there is

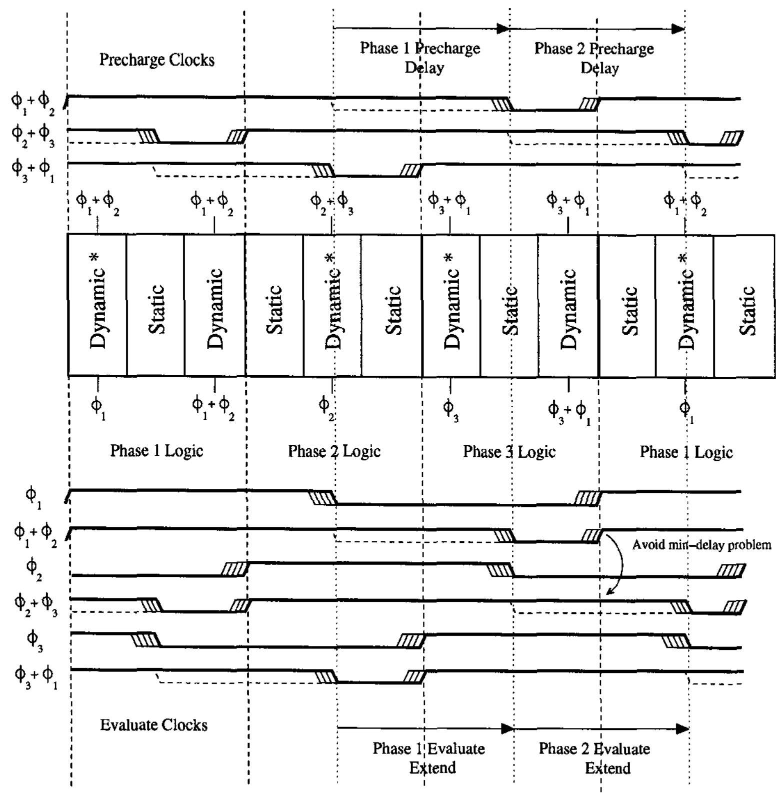

Figure 3.6: Clock-logic OR-precharge/OR-evaluate.

no need to delay the extended (OR-evaluate) clocks from the extra clocks after the first gates since the extended clocks are directly derived from the extra clocks using the transistors at each gate. This means that min-delay problems cannot happen between the extra clocks and the extended clocks. Furthermore, only the extra clocks have to be distributed from a local clock generator, thus reducing the complexity of the clock generator and distribution network. Most importantly, this CL-domino method eliminates the dead space between the first gate of the phase and the later gates, where logic might possibly wait for the delayed clock [29, 62]. It is also important to note that single-rail OTB domino, cannot by itself incorporate input complemented or non-monotonic functions.

Since the evaluate duty-cycle has been increased, there is more time for the domino gates to evaluate or equivalently the current phase can allocate more slack time to the previous phase so that the previous phase can borrow more time from the current phase. The direction of the skew between two adjacent clock phases , Φ_2 wrt Φ_1, affects evaluate times as follows: negative skew gives the Φ_1 logic less time to evaluate while giving the Φ_2 logic more time to evaluate and positive skew gives the Φ_1 logic more time to evaluate and the Φ_2 logic less time to evaluate.

Since the precharge duty-cycle has also been increased, there is less time for the domino gates to precharge. The direction of the skew between two adjacent clock phases Φ_2 wrt Φ_1 affects precharge times as follows: negative skew gives the Φ_2 logic less time to precharge while giving the Φ_1 logic more time to precharge and positive skew gives the Φ_2 logic more time to precharge and the Φ_1 logic less time to precharge.

In regards to precharge and evaluate constraints, the equations derived for OR-precharge / Domino-evaluate will still apply, since the precharge of the domino gates are uncharged from that embodiment and the OR of the evaluation clocks do not affect those constraints.

However, for phases where *Dynamic** implements a regular dynamic logic function, the maximum time that can be borrowed from next next phase is increased to:

$$t_{borrow\_max\_OR\_pre\_OR\_eval} = t_e - t_{skew}. \tag{3.10}$$

The extra time available for time-borrowing helps to alleviate the design difficulties and reduced cycle times associated with imbalanced pipeline stages in high-speed designs.

The min-delay constraint of (2.15) still holds even though the duty-cycle of the clocks are effectively increased. This is a result of the extended clocks being directly derived from the normal clocks. However, a system clocked strictly by OR-precharge/OR-evaluate will fail due to a violated precharge time constraint before it will fail due to any racethrough conditions.

3.2.6 Dynamic Cascaded OR-Precharge/Domino-Evaluate

Dynamic stages that are directly coupled to each other with no intervening static logic predate the popularity of CMOS circuits [55]. A fifth method for CL-domino which uses OR-precharge for all single-rail domino pipelines, is called "Dynamic Cascaded OR-Precharge / Domino-Evaluate" and is shown in Fig. 3.7.

Dynamic gates can be placed back-to-back at phase boundaries without an intervening

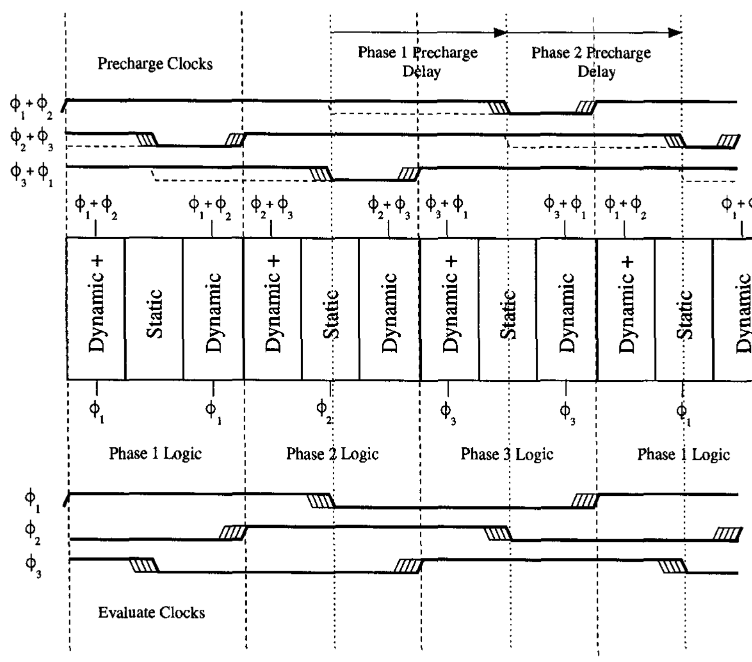

Figure 3.7: Clock-logic dynamic cascaded OR-precharge/normal-evaluate.

static gate as long as the logic of the current phase finishes before the next phase begins. Since it has been established in previous embodiments of CL-domino, that time cannot be borrowed from a phase that implements an input complemented or non-monotonic function in any case, cascading dynamic gates directly will result in better performance in some situations since the pipeline will contain more dynamic gates in the critical paths.

$Dynamic^+$ is an input complemented function, a non-monotonic function, or a regular dynamic logic function. *Dynamic* on the other hand is a regular dynamic logic function.

Since this embodiment is based on OR-precharge/domino-evaluate, the equations derived for Section 3.2.2 apply. However, a system with strictly back-to-back dynamic gates across phases, will support no time-borrowing, as mentioned previously, and all phases will have a restricted logic evaluation time as derived in (3.1). However, for practical CL-domino pipelines, this logic style will often be used in combination with other CL-domino or skew-tolerant domino, that does support time-borrowing across phases that require it.

3.2.7 Dynamic Cascaded Domino-Precharge/AND-Evaluate

A directly cascaded version of AND-evaluate, where dynamic gates are directly coupled back-to-back at phase boundaries without an intervening static gate, can also be derived. Our sixth embodiment of CL-domino, which uses AND-evaluate for all single-rail domino pipelines, is called "Dynamic Cascaded Domino-Precharge / AND-Evaluate" and is implemented as in Fig. 3.8.

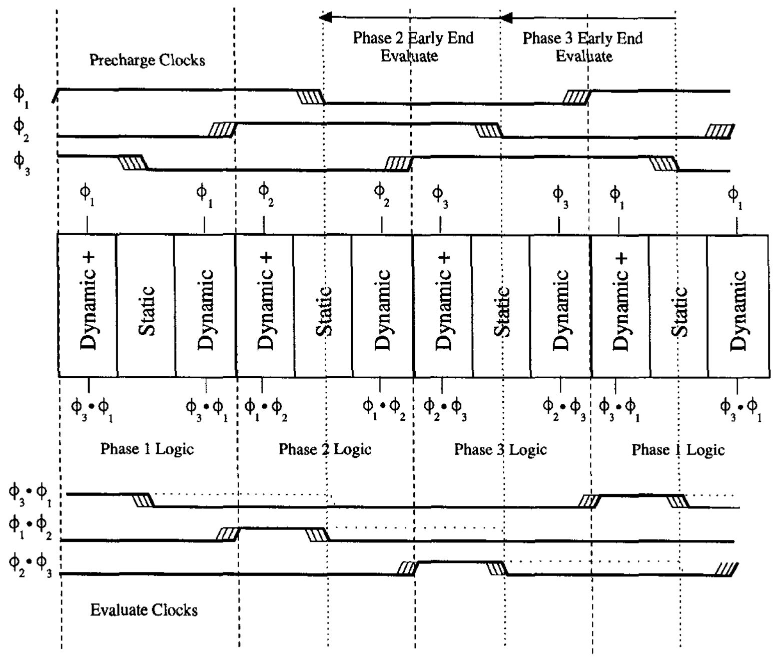

Figure 3.8: Clock-logic dynamic cascaded normal-precharge/AND-evaluate.

Dynamic$^+$ is an input complemented function, a non-monotonic function, or a regular dynamic logic function. *Dynamic* on the other hand is a regular dynamic logic function.

Since this embodiment is based on normal-precharge/AND-evaluate, the equations derived for Section 3.2.3 apply. A system with strictly back-to-back dynamic gates across phases, will support no time-borrowing and all phases will have a restricted logic evaluation time as determined by the amount of guaranteed overlap between adjacent phases (3.6). This is less than the evaluation time as derived for dynamic cascaded OR-precharge/domino-evaluate. However, for practical CL-domino pipelines, this logic style will often be used in combination with other CL-domino or skew-tolerant domino, that does support time-borrowing across phases and longer evaluation duty cycles.

3.3 Local Clock Generation at the Dynamic Gates

In this section we shall describe how logic functions of overlapping clocks can be easily implemented at the transistors of the dynamic gates in CL-domino. Generating the required clocks within the dynamic gates simplifies the clock distribution network since fewer phases have to be distributed, and hence, less inter-phase skew introduced. Local clock functions are skew tolerant and also allow performance scaling of traditional skew-tolerant domino designs, and its variants such as OTB domino, without much design modification since the new clocks are implemented through an addition of a minimum number of transistors to a dynamic gate.

3.3.1 Clock Logic Formulas

Clock-Logic Domino Equations

An ordinary N-type dynamic gate, as in Fig. 3.9, implements the logic function [61]:

$$Out = \overline{CLK} + \overline{(BOOL)} \cdot CLK. \tag{3.11}$$

$BOOL$ is a non-inverting monotonic logic function. For example, the dynamic gate of 3.10 implements:

$$Bool = (A \cdot B + C). \tag{3.12}$$

$$Out = \overline{CLK} + \overline{(A \cdot B + C)} \cdot CLK. \tag{3.13}$$

Through using multiple clocks at a dynamic gate, different logical functions of the clocks can be used for precharge and evaluate operations. A Clock-Logic dynamic gate therefore implements the generalized function:

$$Out = \text{precharge condition} + \text{evaluate condition} + \text{state}. \tag{3.14}$$

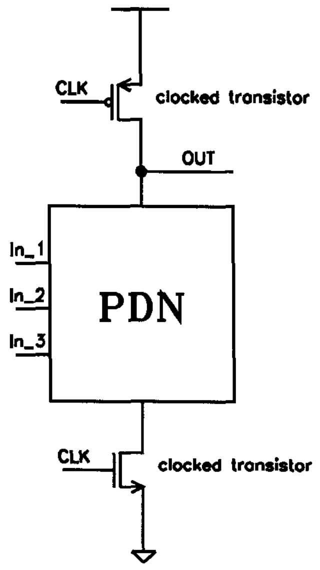

Figure 3.9: N-type dynamic gate.

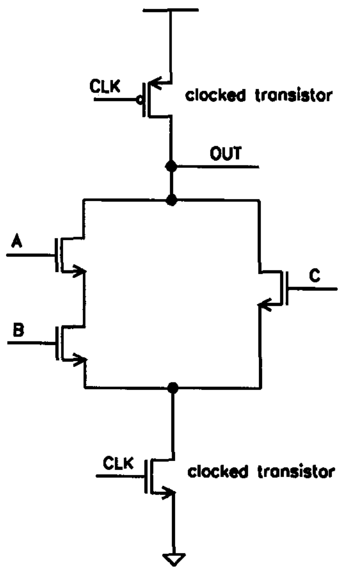

Figure 3.10: Example N-type dynamic gate.

The *state* exists for dynamic logic when the clocks for precharge and evaluate are different and the clocked transistors fully disconnect the gate from V_{dd} and V_{ss}. In this mode of operation, the gate is neither precharging nor evaluating but instead holding its previous state much like an opaque latch.

For the Clock-Logic implementations as described in Section 3.2, the dynamic stages implement the following logic functions:

$$Out_{or\_pre} = \overline{CLK1} \cdot \overline{CLK2} + \overline{(BOOL)} \cdot CLK1 + state \cdot \overline{CLK1} \cdot CLK2. \quad (3.15)$$

$$Out_{and\_eval} = \overline{CLK2} + \overline{(BOOL)} \cdot CLK1 \cdot CLK2 + state \cdot \overline{CLK1} \cdot CLK2. \quad (3.16)$$

$$Out_{or\_pre\_and\_eval} = \overline{CLK2} \cdot \overline{CLK3} + \overline{(BOOL)} \cdot CLK1 \cdot CLK2 + state \cdot \overline{CLK1} \cdot CLK2. \quad (3.17)$$

$$Out_{or\_pre\_or\_eval} = \begin{cases} \overline{CLK1} \cdot \overline{CLK2} + \overline{(BOOL)} \cdot CLK1 + state \cdot \overline{CLK1} \cdot CLK2 : & \text{first gate of phase.} \\ \overline{CLK1} \cdot \overline{CLK2} + \overline{(BOOL)} \cdot (CLK1 + CLK2) : & \text{other gates of phase.} \end{cases} \quad (3.18)$$

$$Out_{dyn\_cascade\_or\_pre} = Out_{or\_pre}. \quad (3.19)$$

$$Out_{dyn\_cascade\_and\_eval} = Out_{and\_eval}. \tag{3.20}$$

$CLK1$ is a first clock phase, $CLK2$ is a next clock phase and $CLK3$ follows $CLK2$.

Clock Logic Domino Transistor Level Implementation

The transistor level implementations of equations (3.15), (3.16), and (3.17) are shown as Fig. 3.11, Fig. 3.12, and Fig. 3.13 respectively. Here series PMOS clock transistors represent ORed precharge clocks while series NMOS clock transistors represent ANDed evaluate clocks.

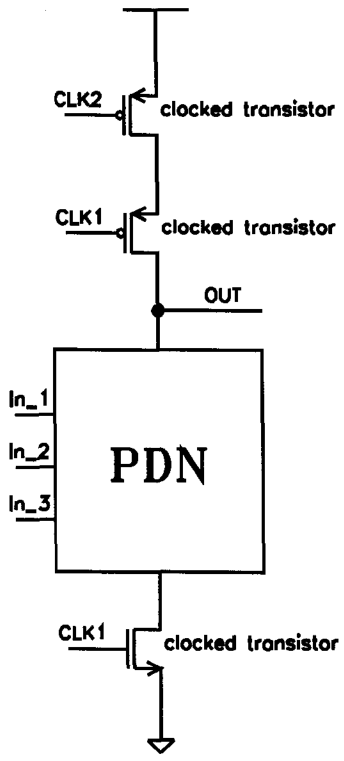

Figure 3.11: OR-precharge.

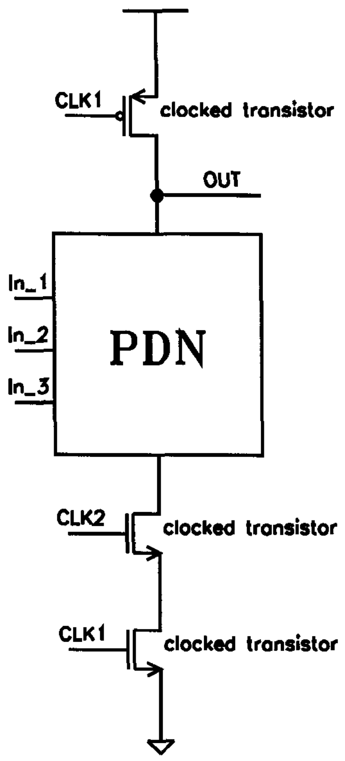

Figure 3.12: AND-evaluate.

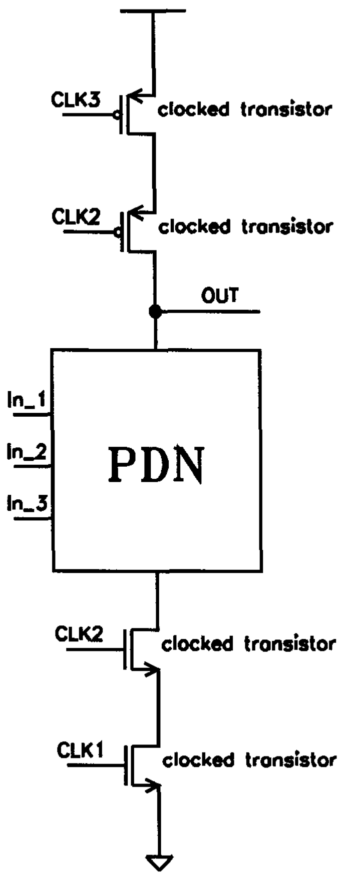

Figure 3.13: OR-precharge / AND-evaluate.

The transistor level implementation of equations (3.18) is shown in Fig. 3.14 for the first gate of each logic phase, while Fig. 3.15 shows the configuration used for the other remaining gates in the phase. Here series PMOS clock transistors represent ORed precharge clocks while parallel NMOS clock transistors represent ORed evaluate clocks.

In the other degenerate case, parallel PMOS clock transistors can be used to represent ANDed precharge clocks. The arrangement of transistors for ORing and ANDing clocks is similar in form to that for normal static CMOS logic gates with data signals as the inputs to the gate terminals of the transistors [5].

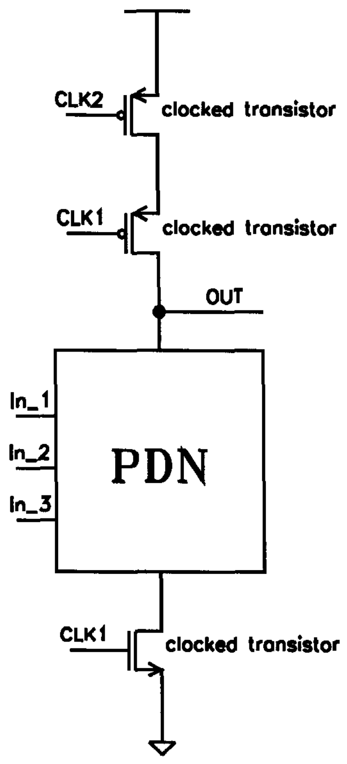

Figure 3.14: First gate of phase in OR-precharge / OR-evaluate.

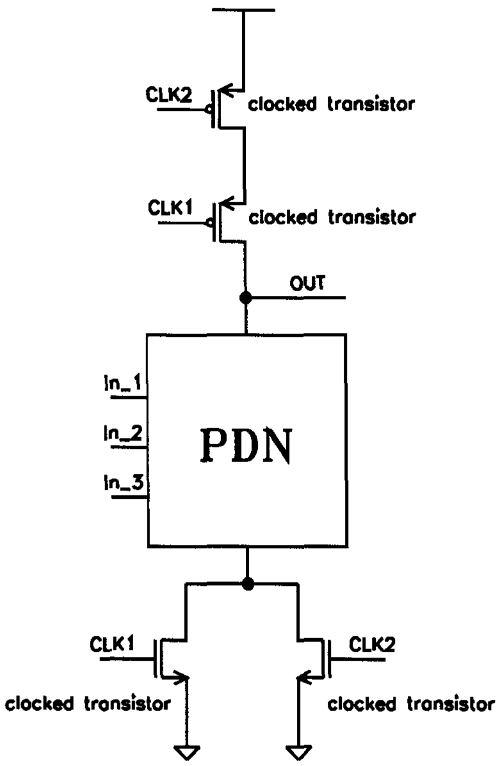

Figure 3.15: Remaining gates of phase in OR-precharge / OR-evaluate.

3.4 Transistor Level Design Considerations

3.4.1 Design Considerations for Series and Parallel Connections of Clock Transistors

Series connections of clocked transistors increases the resistance, while decreasing the conductance, from the V_{dd} supply-rail to the output node for series PMOS and from the bottom of the logic network to the V_{ss} supply-rail for series NMOS. Parallel connections of clocked transistors decreases the resistance, while increasing the conductance, when both transistors are on, while the resistance and conductance are unchanged from a single clocked transistor, when only one transistor is on. This is commonly the case when these circuits switch.

Series PMOS Clock Transistors

In the case of series PMOS clock transistors, the increased resistance will negatively affect the precharge time of the logic gate unless the channel width of the PMOS transistors are increased. This is made worse by the fact that OR-precharge allows less time for the precharge operation as discussed previously in Section 3.2. Increasing the channel width, however, is detrimental to the clock load and hence the clock power of an OR-Precharge CL-domino gate. Furthermore, the parasitic load capacitance to the output node is also increased, thus reducing the gate's evaluate switching speed. Last, since PMOS transistors have one-half to one-third the mobility of NMOS transistors, they must be sized even larger.

These design issues can be managed through selective placement of series PMOS transistors which will be discussed in detail. So long as the dynamic gate can precharge within the shortened precharge cycle time, the overhead incurred through larger PMOS transistors is small compared to the power savings and delay performance obtained for Clock-Logic domino compared to dual-rail skew-tolerant domino.

Series NMOS Clock Transistors

In the case of series NMOS clock transistors, the increased resistance will negatively affect the time it takes the logic gate to pull-down the output node (logic evaluate time) unless the channel width of the NMOS transistors are increased. This is compounded by the fact that AND-evaluate allows less time for the evaluate operation as derived previously in Section 3.2. Increasing the channel width, however, is detrimental to the clock load and hence the clock power of an AND-evaluate CL-domino gate. Since the height of the NMOS Pull-Down-Network (PDN) has now been effectively increased by one transistor, all the transistors of the NMOS stack should be increased to minimize circuit delay as is common practice in digital CMOS circuits [61]. This will negatively affect the circuit area, increase the effects of charge sharing, while increasing the output load of the logic gates that drive it.

These design issues can be managed through selective placement of stacked NMOS transistors which will be discussed in detail. As long as the dynamic gate can evaluate within the shortened evaluate cycle time, the overhead incurred through larger NMOS transistors is small compared to the power savings and delay performance obtained for Clock-Logic domino compared to dual-rail skew-tolerant domino.

Parallel PMOS Clock Transistors

Parallel PMOS clocked transistors have the effect of increasing the cycle time for precharge compared to that for evaluate. Since precharge for a stage of dynamic logic often occurs in parallel and evaluation occurs in series, increasing the precharge period is not normally required for CL-domino logic.

Parallel NMOS Clock Transistors

In the case of parallel NMOS clock transistors, the decreased resistance, when both transistors are on, will increase the conductance from the bottom of the PDN to ground. Increasing the channel widths of the transistors will increase the PDN conductance even more while trading off increased clock load and clock power. However, the increased parasitic capacitance of the additional clock transistor cannot be ignored. Thus the increase in pull-down current is to some extent offset by the increase in parasitic capacitance of the the clock transistor drains. When only one clock transistor is on, the pull-down current is the same as when only clocked transistor is used. However, the drain node of the clock transistors will still be capacitively loaded by the off clock transistor, thus decreasing the switching speed of the logic gate. The decreased logic evaluate time is usually not appreciable.

3.4.2 Clock Input Ordering

Logic gates often have internal node capacitances that must be (dis)charged in addition to the output load. Because of this, the ordering of the clock transistors can affect the transient performance of CL-domino gates. In the sections that follow, we limit our discussion to Clock-Logic functions of two inputs, although more inputs are possible and the results extend to those gates as well.

Series PMOS Clock Transistors

The ordering of the clock inputs for OR-precharge, can affect both the precharge and evaluate times of a CL-domino gate. The two options for the series connection of two PMOS clock transistors are shown in Fig. 3.16 and Fig. 3.17, where $CLK1$ is a current phase clock, which rises T_c/N before a later phase clock, $CLK2$.

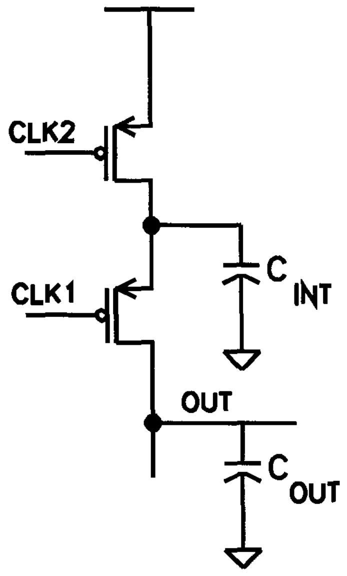

Figure 3.16: Placing current phase precharge clock nearest to the output.

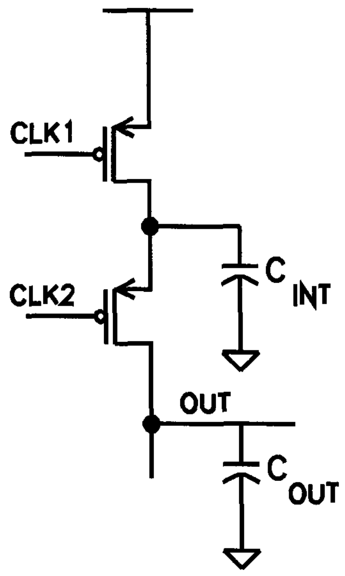

Figure 3.17: Placing current phase precharge clock nearest to the supply rail.

The internal node capacitance, C_{INT}, has to be discharged along with the output capacitance C_{OUT} during evaluation when the PMOS clock transistor closer to the output is enabled. Since the size of the PMOS clock transistors are larger than normal skew-tolerant domino, gates that are located early in a phase (after $CLK1$ rises and $CLK2$ is still low) can evaluate faster if the clock transistor of the current phase is placed closer to the output node as in Fig. 3.16. The capacitance that needs to discharged is equal to C_{OUT}, resulting in faster operation.

For $CLK1$ phase gates that evaluate while $CLK2$ is high, or alternatively when $CLK1$ logic is borrowing time from $CLK2$, the ordering of the clock transistors does not affect the evaluate time. However, during the precharge period, the capacitance to charge can either be C_{OUT} or $C_{OUT} + C_{INT}$. Therefore, when time borrowing into the next phase is expected, a gate can precharge faster through placing the current phase clock transistor closer to the supply rail as in Fig. 3.17. This is because the internal node capacitance, C_{INT}, is already precharged by the time the next phase clock goes low and precharge of the output node begins. Hence the capacitance to be precharged is equal to C_{OUT} during that period.

Series NMOS Clock Transistors

The ordering of the clock inputs for AND-evaluate, can affect the evaluate time, precharge time and noise tolerance of a CL-domino gate. The two options for the series connection of two NMOS clock transistors are shown in Fig. 3.18 and Fig. 3.19, where $CLK1$ is a previous phase clock which rises T_c/N before a current phase clock, $CLK2$.

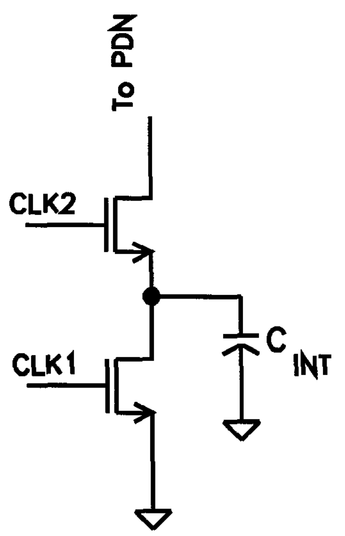

Figure 3.18: Placing current phase precharge clock nearest to the PDN.

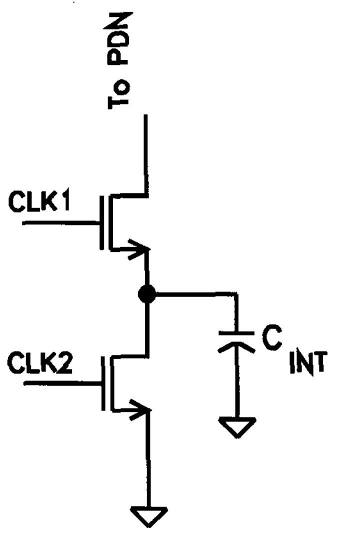

Figure 3.19: Placing current phase precharge clock nearest to the ground rail.

The internal node capacitance, C_{INT}, has to be discharged in either case although it is possible to discharge C_{INT} early. Since the size of the NMOS clock transistors are larger than in normal skew-tolerant domino, CL-Domino AND-Evaluate gates can evaluate faster if the clock transistor of the current phase is placed nearest to the PDN as in Fig. 3.18. This is because the internal node capacitance, C_{INT}, would have already been predischarged when the current phase clock rises. This is similar to the method of transistor re-ordering to reduce delay as is common practice in digital VLSI design [61], with $CLK2$ synonymous with a late arriving input signal.

Since gates clocked with AND-Evaluate can have charge sharing problems when the previous phase precharges as, described in Section 3.2.4, a gate can be made more robust to this form of charge sharing through placing the current phase clock transistor nearest to the ground-rail as in 3.19. This is because the internal node capacitance, C_{INT}, is disconnected from the PDN during the period that charge sharing can occur, while $CLK1$ is low and $CLK2$ is high. C_{INT} is reasonably large even when compared to the output capacitance because of the increased height of the NMOS stacks and the increased sizes of the transistors thus making the gate more susceptible to charge sharing.

3.4.3 Charge Sharing

Both OR-Precharge and AND-Evaluate have a time during the clock cycle where the output node is floating either high or low as shown in Fig. 3.20. It is during this time that the gate is susceptible to charge sharing.

Keeper Design

In order to alleviate charge sharing, leakage currents and staticize the dynamic circuits during power saving clock stop, full keepers should be used on the outputs of Clock-Logic dynamic gates. That is, if keepers are used, they must contain both PMOS and NMOS feedback devices since the output node can float either high or low. For more generalized CL-domino pipelines, as described further, full keepers should be used on every gate that is clocked by OR-Precharge or AND-Evaluate. If maintaining circuit state during clock stop is the only concern, then only the dynamic circuits of a chosen phase need contain keepers.

The use of weak keepers slightly increases the diffusion capacitance to the output nodes of these dynamic gates. Furthermore, precharge and evaluate operations have to overcome a brief period of contention between the keeper transistors and the precharge and logic evaluation networks, respectively. Note that CL-domino systems clocked by OR-Precharge / OR-Evaluate as described in Section 3.2.5, do not require full keepers since the output can only float high. A half PMOS feedback device will suffice in this case.

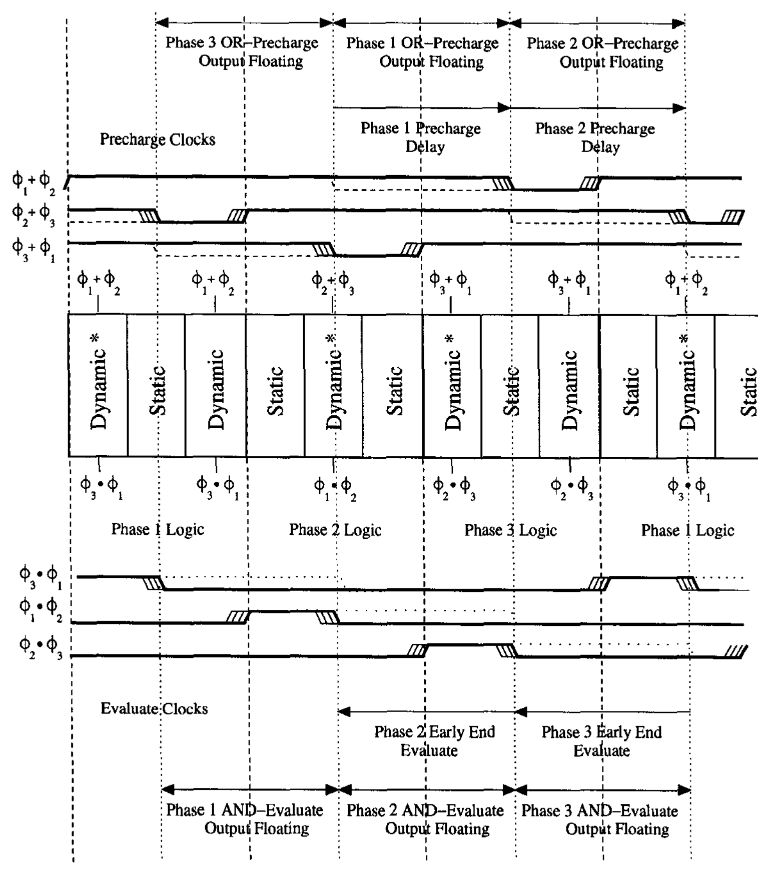

Figure 3.20: Clock-logic Output Floating.

Precharging Internal Nodes

Precharging internal stack nodes is an effective method of dealing with charge sharing in dynamic circuits. For CL-domino logic, the secondary precharge network, used for precharging internal stack nodes, must have the same form as that used for the primary precharge network as shown in Fig. 3.21 and Fig. 3.22. This is because precharging the internal stacks in the same manner as in traditional dynamic logic may lead to the corruption of the output values since the gate needs to maintain its state for a larger portion of the cycle than is the case with a single clock.

The method of AND-evaluate also presents problems with charge sharing when the previous phase precharges as described previously for gates that implement input complemented or non-monotonic logic. This problem cannot be solved with the precharging circuitry of Fig. 3.22 since the secondary precharge network would not have been previously

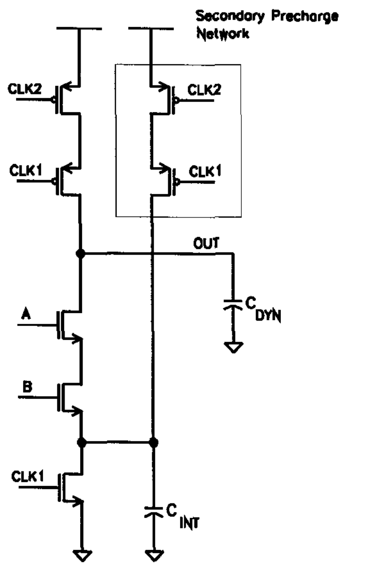

Figure 3.21: OR-precharge of internal nodes.

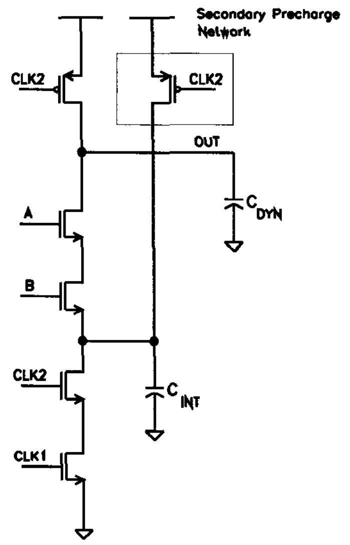

Figure 3.22: AND-precharge of internal nodes.

enabled prior to the time where potential charge sharing occurs. A circuit that precharges internal nodes during this glitching period is shown in Fig. 3.23.

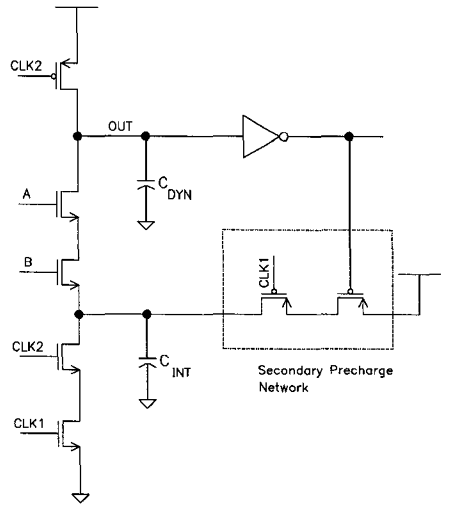

Figure 3.23: AND-precharge of internal nodes for charge sharing suppression.

The proposed circuit will not corrupt an evaluated low result on *Out* since the secondary precharge network is conditionally activated only when the value of *Out* was evaluated high. In this case, precharge of the internal node is desirable to avoid charge sharing. Furthermore, the circuit will initiate precharge of the internal node during the normal precharge period to circumvent charge sharing during evaluate, and thus can be used in place of the secondary precharge network of Fig. 3.22.

3.5 Optimized Placement for OR-Precharge/AND-Evaluate

3.5.1 CL-Domino and Skew-Tolerant Domino

It was established previously that exclusive use of CL-domino clocking for gates that do not implement input complemented or non-monotonic functions, is detrimental to the switching speed and power dissipation of a dynamic pipeline. Therefore, a more general CL-domino methodology might integrate CL-domino OR-Precharge, AND-Evaluate, Dynamic Cascaded OR-Precharge and Dynamic Cascaded AND-Evaluate with normal skew-tolerant domino logic to achieve the fastest speed, lowest power dissipation and lowest area.

Note that OR-Precharge / OR-Evaluate is a CL-domino logic style that cannot be incorporated with normal skew-tolerant domino. However, the extended evaluate duty cycle and increased opportunity for time borrowing for that logic style may offset any potential performance and power penalties incurred for that logic style.

For generalized CL-domino, only those gates that implement input complemented or non-monotonic functions, at a phase boundary, require more than one clock to be distributed to the dynamic gate. If OR-precharge is used, the last gate of the previous phase requires two series PMOS clock transistors, each driven by successive clock phases. The first gate of the current phase, that implements the input complemented/non-monotonic function, and the subsequent gates of the phase can be clocked with single PMOS precharge transistors and single NMOS evaluate transistors as in skew-tolerant domino logic. If AND-evaluate is used, the first gate of the current phase, that implements the input complemented/non-monotonic function, requires two series NMOS clock transistors, each driven by successive clock phases. The subsequent gates of the current phase can be clocked with single PMOS precharge and single NMOS evaluate transistors as with skew-tolerant domino logic.

When an input complemented or non-monotonic function occurs in a domino pipeline, only one of either OR-precharge or AND-evaluate is required. The remaining dynamic gates of the input complemented/non-monotonic phase and the gates of phases that implement normal monotonic output logic can be clocked with single PMOS precharge transistors and single NMOS evaluate transistors, thus increasing the switching speed of those gates and reducing the clock power. One very important property of this arrangement is that only the gates that use series transistors for precharge or evaluation need to follow the more stringent timing constraints for CL-domino logic while the remaining gates need to adhere to the less stringent timing constraints for skew-tolerant domino logic. This generalized method applies particularly well for non-footed dynamic gates to increase switching speed with no short circuit current as will be discussed later.

Examples of cases where CL-domino and skew-tolerant domino gates adjacent within the same pipeline are presented in Table 3.1. Here, a *monotonic* gate type describes a skew-tolerant dynamic gate with a single precharge transistor and a single evaluate transistor.

3.5.2 Input Complemented or Non-Monotonic Gates in Two or More Consecutive Phases

When two or more consecutive clock phases contain input complemented or non-monotonic logic, the connections of the clocked transistors in the pipeline demonstrates a unique pattern.

If OR-precharge is used for the consecutive non-monotonic logic phases, the clocking scheme employed will be that shown in Fig. 3.24.

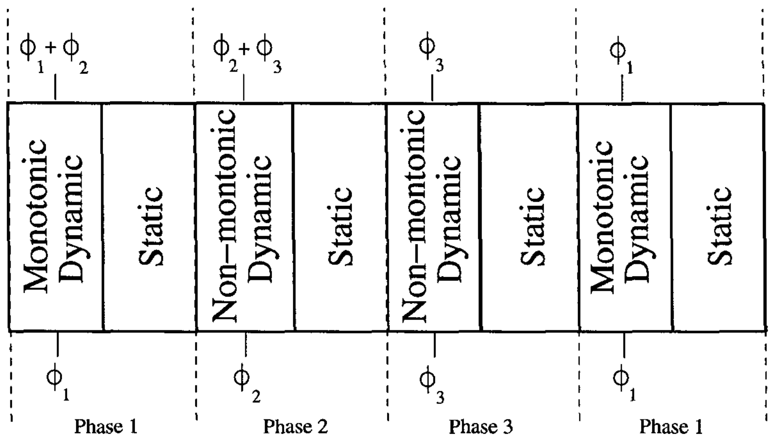

Figure 3.24: OR-precharge in consecutive phases of non-monotonic logic.

Table 3.1: Examples of CL-domino and skew-tolerant domino in the same pipeline.

Phase Position	Previous Gate Type	Current Gate Type	Next Gate Type
first	monotonic	monotonic	monotonic
	monotonic OR-pre	complemented domino eval	monotonic
	monotonic	complemented AND-eval	monotonic
	complemented AND-eval	monotonic	monotonic
	complemented domino-eval	monotonic	monotonic
middle	monotonic	monotonic	monotonic
	complemented domino eval	monotonic	monotonic
	complemented AND-eval	monotonic	monotonic
last	monotonic	monotonic	monotonic
	monotonic	monotonic	complemented AND-eval
	monotonic	monotonic OR-pre	complemented domino eval
first = last	monotonic	monotonic	monotonic
	monotonic OR-pre	complemented domino eval	complemented AND-eval
	monotonic	complemented OR-pre and AND-eval	complemented domino eval
	monotonic OR-pre	complemented OR-pre	complemented domino eval
	monotonic	complemented AND-eval	complemented AND-eval

If AND-evaluate is used for the consecutive non-monotonic logic phases, the clocking scheme employed will be that shown in Fig. 3.25.

Furthermore, if OR-precharge is followed by AND-evaluate in consecutive non-monotonic logic phases, the clocking scheme will be that as shown in Fig. 3.26.

Last, if AND-Evaluate is followed by OR-precharge in consecutive non-monotonic logic phases, the clocking scheme will be that as shown in Fig. 3.27.

3.5.3 Unfooted Gates

Unfooted gates can cause short circuit currents during the precharge operation since precharge to those gates occur in series rather than in parallel as footed domino logic. An example of

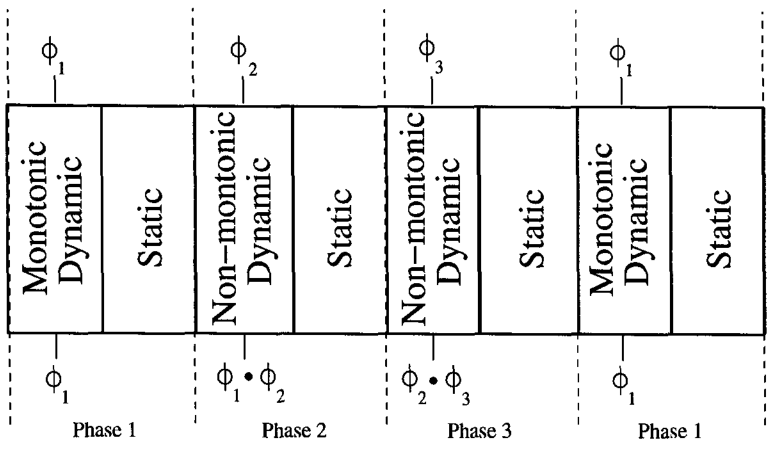

Figure 3.25: AND-evaluate in consecutive phases of non-monotonic logic.

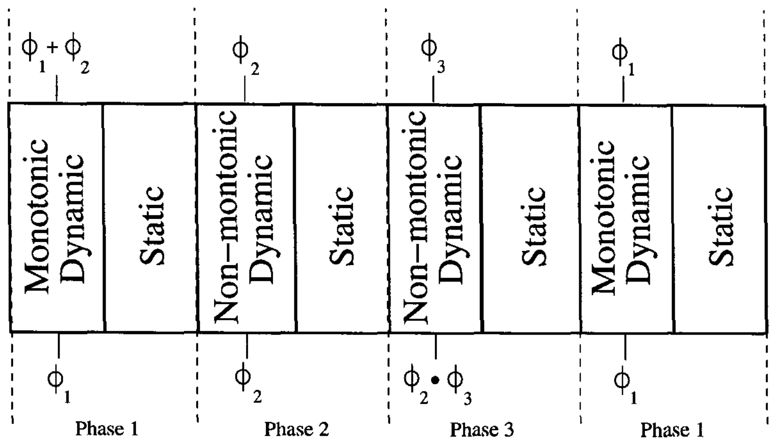

Figure 3.26: OR-Precharge followed by AND-evaluate in consecutive phases of non-monotonic logic.

an unfooted gate in a CL-domino pipeline for high-speed operation is shown in Fig. 3.28.

Normally the precharge delay of the footed dynamic gate (1) affects the short circuit current through the unfooted dynamic gate (2) since gate (1) must have flipped the value of its static gate from 1 $\longrightarrow$ 0 before gate (2) can fully precharge to the rail voltage. Furthermore, since precharge now ripples, the precharge delay path must be considered as well since all the dynamic gates must have a high voltage on its output node when the evaluation phase begins. Using series PMOS transistors for the unfooted gate delays the onset of precharge so that gate (1) is fully precharged before gate (2) starts precharging thus eliminating short circuit current. Note that while gate (2) has to obey CL-domino precharge constraints, gate (1) only has to obey the relaxed precharge constraints of skew-tolerant

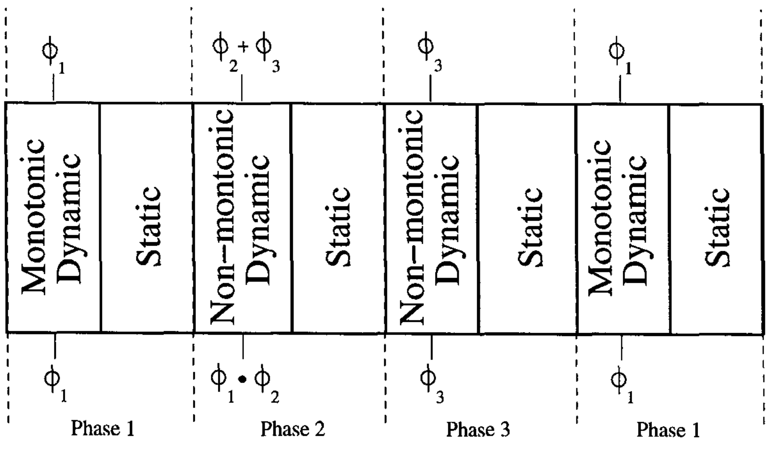

Figure 3.27: AND-evaluate followed by OR-precharge in consecutive phases of non-monotonic logic.

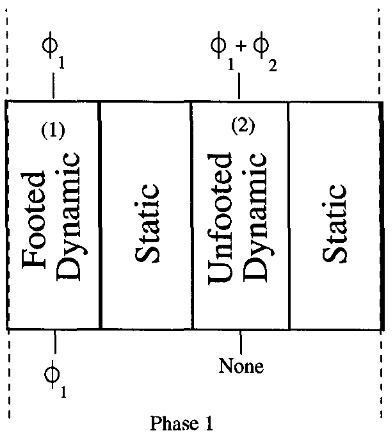

Figure 3.28: Unfooted dynamic gates in a CL-domino pipeline.

domino. However, logic gate (1) should not take too long to precharge since it must finish precharge by the time $\Phi_1 + \Phi_2 = 0$, which starts the precharge of gate (2) in order to avoid short circuit current. A proprietary circuit technique which produces a similar effect with a delayed clock and series PMOS transistors can be found in [53].

In unfooted domino pipelines with more unfooted gates per phase as shown in 3.29, the later unfooted gate, (3), will draw short circuit current when its prior dynamic gate, (2), precharges. One method to reduce short circuit current in this case would be to delay the precharge clock to gate (3) by the sum of the nominal evaluate delays of gate (2) and its subsequent static gate as described in [8]. However, a better design alternative would be to require a footed gate before any OR-precharge unfooted gate as shown in Fig. 3.30. In this way all of the footed gates would precharge when Φ_1 is low while the unfooted gates would precharge when $\Phi_1 + \Phi_2$ is low, hence eliminating short circuit current altogether while maintaining high speed.

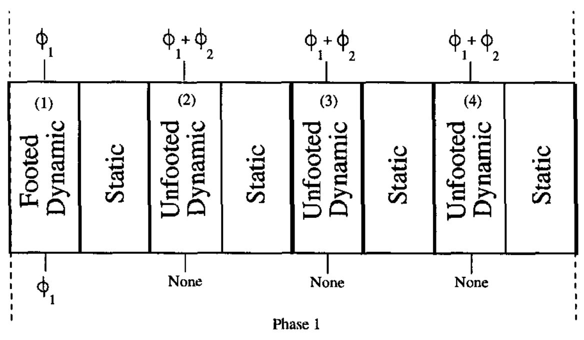

Figure 3.29: Unfooted dynamic gates in a CL-domino pipeline with more gates per phase.

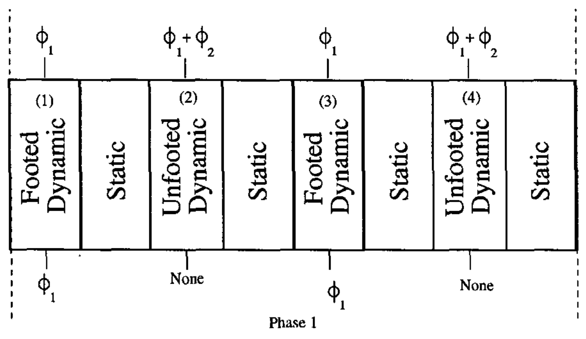

Figure 3.30: Alternating footed and unfooted gates in a CL-domino pipeline.

3.5.4 Generalized CL-Domino Pipeline

A generalized CL-domino pipeline with selective placement of series connected PMOS and NMOS clock transistors is illustrated in Fig. 3.31.

It is important to note that although there are nine set of clocks shown for this pipeline, that only three clocks are actually distributed. All of the other clocks are locally generated through series PMOS and NMOS transistors at the dynamic gates. This simplifies the clock distribution network considerably and makes the network less prone to skew or other clock uncertainties.

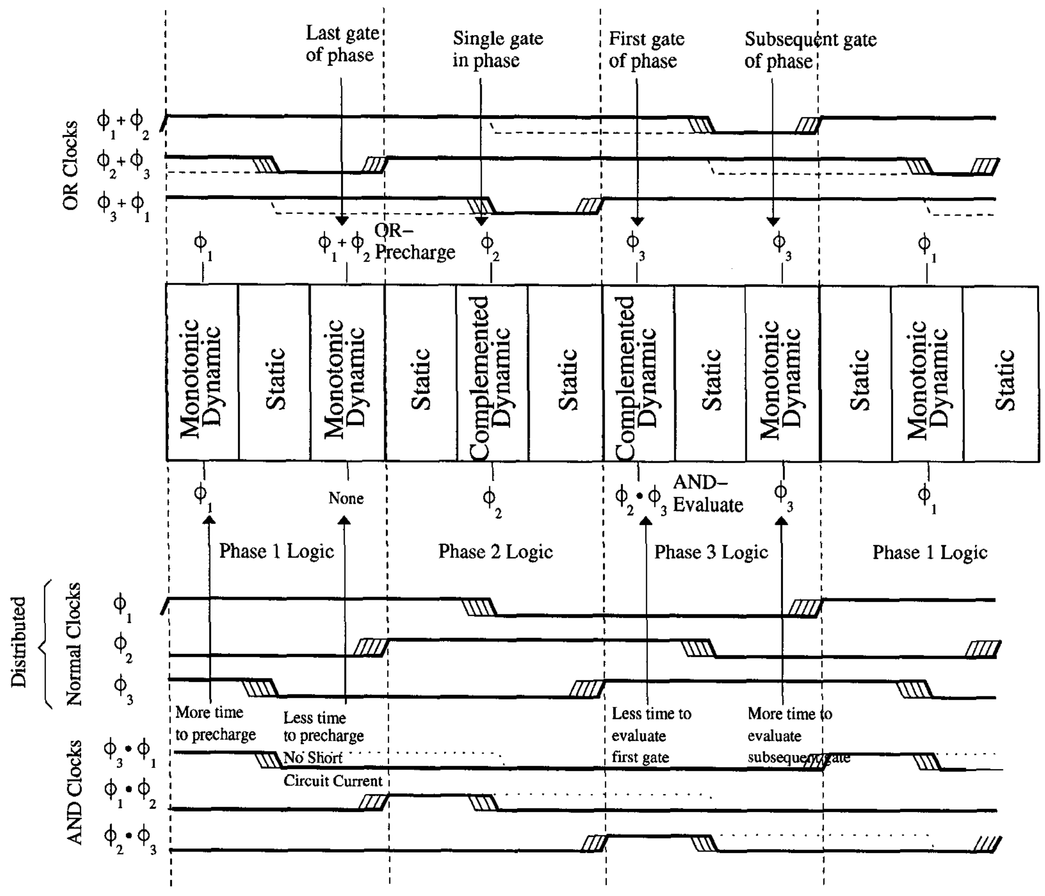

Figure 3.31: Generalized CL-domino pipeline.

3.6 Local Clock Generators

In most high frequency digital systems, including microprocessors, a single global clock is distributed using either RC-matched trees or grids to minimize skew. Elements of tree networks include one-dimensional binary trees, H-trees, X-trees, geometrically matched trees, arbitrary trees, tapered trees, trunks, spines, and meshes [4, 8, 29, 31, 35, 37, 52]. Moreover, pseudogrid-spine networks have been shown which combine trees and grids [4, 18, 31]. Clock-logic domino, much like skew-tolerant domino, can use this same clock distribution scheme with a single global clock. Within each unit or functional block, local clock generators utilizing either delay elements or feedback clock generators, such as Delay-Locked-Loops (DLLs), could produce the required overlapping clock phases for CL-domino.

3.6.1 Clock Logic Generation Using Delay Lines

The simplest method to generate multiple overlapping clocks involve delay lines or delay chains. This method is adequate for most applications since the generated clocks are distributed only within local clock domains and therefore skew can be controlled to a high degree. Fig. 3.32 and Fig. 3.33 [71] show simple three phase clock generators that can be used for CL-domino. The 1/3 cycle delay can be implemented with any even number of inverters, thus forming a non-inverting delay chain. In the general case, the amount of delay required is T_c/N and N phases can be produced by delaying the clock with delay chains. Note that low-skew complement generators for complemented clocks are not required and hence no additional skew is introduced for those components.

The delay line will closely track the speed of critical paths to which it feeds. This is because the delay line will be located in close proximity to the clocked circuits and any variations in voltage, temperature, transistor orientation and processing will affect both the clock generator and clocked logic circuits equally, to a first order. Thus we can say that Clock Logic domino is only sensitive to relative delays rather than absolute delays.

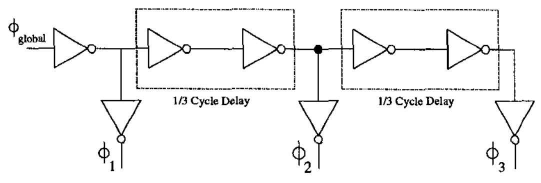

Figure 3.32: Three phase clock generator with inverting buffers.

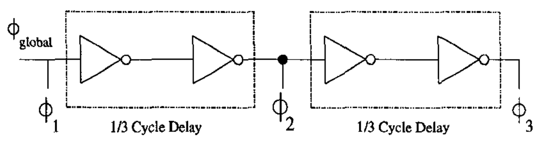

Figure 3.33: Three phase clock generator with non-inverting buffers.

3.6.2 Clock Logic Generation Using Delay Locked Loops

Lower skew and less duty cycle uncertainty can be achieved for CL-domino clocks using feedback systems that track delays over process and environmental changes. There are two common types of feedback systems which precisely generate local clocks from a globally distributed reference. The simpler of the two are delay-locked-loops (DLLs) while the more complicated are, from a loop architecture perspective, the phase-locked-loops (PLLs) [8] . An example of a local three-phase DLL CL-domino generator is shown in Fig. 3.34. It is important to note that this is just a abstracted view of a DLL system whose design is beyond the scope of this work.

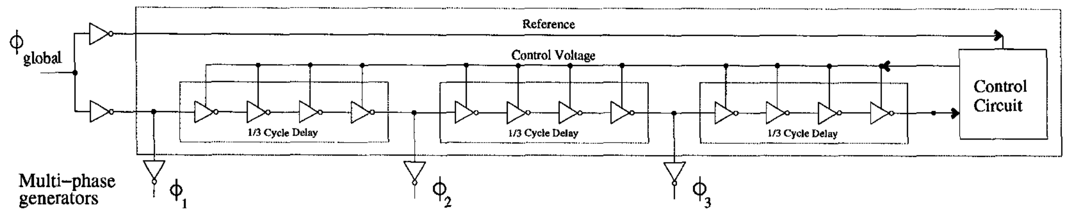

Figure 3.34: Three-phase DLL CL-domino clock generator.

A global PLL distributes a single-phase global clock to multiple DLLs at different CL-domino functional units [54]. This distribution scheme requires less area for clock wiring and is free of inter-phase skew at the global level, where clocks could run for several tens of millimeters. Each DLL loop receives the global clock and delays it by T_c/N, or 1/3 in this case, through adjusting the control voltage to the inverters so that the delay line has a full cycle delay. The delay line in this case is called a Voltage-Controlled-Delay-Line (VCDL). The feedback controller; nominally containing a phase detector, a charge pump and a loop filter [47]; compensates for process and environmentally dependent frequency variations through modulating the delay line voltage. Normally the time required to perform the compensation, or lock time in DLLs, is very short [8].

Recently, improved DLLs have been introduced which overcome, to a certain degree, some limitations of classical DLLs. These include limited delay ranges [10, 34, 47, 56], loop-to-loop jitter [36, 47, 56], power consumption [36, 47], and area penalty [47]. However, there still exists a relatively large area penalty for CL-domino units that employ DLLs over those that employ delay lines. In addition, power requirements, which have recently become a problem in high frequency integrated circuits, and design complexity have to be balanced against any potential skew improvement and duty cycle invariability that DLLs offer. In future microprocessor designs, however, where skew might account for a larger portion of the clock cycle, DLLs might inevitably have to be used.

3.7 Summary

Domino logic circuits provide a raw gate delay advantage over static circuits to achieve multi-GHz performance in high speed designs. Skew-tolerant domino logic uses overlapping clocks to remove the three sources of sequencing overhead in traditional single-ended domino and dual rail domino: clock skew, latch delay and unbalanced logic. However, single-rail domino only allows non-inverting, monotonic logic in the pipeline. Dual-rail domino, with true and complemented versions of the inputs and outputs are usually required. Clock-logic domino, is an alternative to dual-rail domino for computing inverting and non-monotonic logic in a single-rail dynamic pipeline. This is accomplished through adding between one to four clocked transistors to selected dynamic gates. Many of the benefits of skew-tolerant domino such as immunity against clock skew and time borrowing to balance pipeline stages are inherited for normal monotonic logic functions although the windows for skew tolerance are narrowed and non-time borrowing penalties exist for any complemented or non-monotonic function. If the timing guidelines for CL-domino are adhered to, a single rail domino pipeline can accomplish the same function as a dual-rail domino pipeline while minimizing power consumption, RC delays and area.

Chapter 4

Example High-Speed Microprocessor Datapath: 64-bit ALU

4.1 Introduction

To evaluate the delay performance and power benefits of Clock-Logic domino in the context of current generation high-speed microprocessors, we compared three 64-bit Arithmetic Logic Unit (ALU) datapaths: the first built in CL-domino, the second built in dual-rail skew-tolerant domino and the third built in static CMOS. The ALUs were simulated in TSMC's 0.18 μm general purpose logic process, with a typical inverter delay of 29.5-ps, an FO4 inverter delay of 71.6-ps, and a nominal operating voltage of 1.8-V. We assumed an ALU microarchitecture similar to that proposed by Intel [30, 32], by IBM [26], and by Sun Microsystems [28]. For the comparison study, each of the dynamic ALUs, CL-domino and dual-rail domino, was modeled with contacted diffusion parasitics on each transistor source/drain and with parasitic capacitors on long signal bus lines. We did not, however, model the capacitive or resistive parasitics of the remaining signal wires. For the static CMOS ALU, the transistor parasitics were modeled using extracted layout data from a standard cell library and the effect of wiring was minimized using a very optimistic wire load model for fair comparison.

This chapter is organized as follows: Section (4.2) describes the architecture and circuit design of a 64-bit ALU in three different CMOS logic families: CL-domino, dual-rail skew-tolerant domino and static CMOS. Section (4.3) describes the simulation methodology and gives the simulated power dissipation and delay performance results of the three

ALU designs.

4.2 ALU Design

For current generation, out-of-order superscalar microprocessors, 64-bit ALUs that can execute integer instructions and effective address calculations with single-cycle latency and throughput in order to avoid unnecessary stalls are desirable [30, 32]. Multiple integer ALUs, working in parallel, typically occupy a stage of a microprocessor pipeline in the portion of the instruction cycle called "Execute" [44]. The cycle time of the ALUs, in some cases, can determine the maximum clock frequency of a microprocessor and so must be designed for maximum speed. However, since the ALU is also one of the most utilized components, they must also be energy-efficient to conserve power and improve reliability.

ALUs typically consist of a fast binary adder core, a set of inverted inputs to support subtraction, input multiplexers, a variable shifter, and possibly output multiplexers to select between the adder and other simpler functional units. All of these components will be incorporated into our ALU test architecture.

4.2.1 ALU Architecture

The ALU structure as proposed in [22, 30] was used to design all three ALUs. This particular architecture contains wide input multiplexers; to support a combination of forwarding from different microprocessor instruction pipeline stages and data from register files, first-in-first-out (FIFO) buffers and other ALUs; variable 4-b left shift operations for one operand as discussed in [22], and negation/subtraction operations for the second operand. It can also select among the results of the adder and two other simpler logical units for the output result bus. The simpler logical units are not on the critical path of the ALU and hence were not implemented for these experiments. The ALU has a feedback path which allows the previous results to be used directly in the following cycle. This allows fast parallel out-of-order execution for superscalar microprocessors.

Although microprocessor pipelining often occurs across instruction stages, micropipelining within an instruction stage is also possible. The CL-domino ALU and the dual-rail skew-tolerant domino ALU are micropipelined to operate from a locally generated five-phase clock with no intervening latches between blocks of domino logic to reduce the skew penalty, manage logic imbalances and reduce the micropipeline overhead for latch delays. Each stage of the ALU micropipeline is distinct from the other stages according to the clocks that control its precharge and evaluate operations. Studies of assigning clocks to skew-tolerant micropipelines in high speed functional units, including a 64-bit adder, have been previously discussed [42, 46, 48]. Those micropipelines were concentrated on efficient time-borrowing between stages to minimize the delay for stages of imbalanced logic. We will concentrate not only on skew tolerance and time borrowing, but also on assigning clocks to reduce the performance penalty at the interface between monotonic and non-monotonic logic since time cannot be borrowed across this boundary.

CL-domino ALU

A generalized CL-domino pipeline using a combination of OR-precharge, AND-evaluate, domino precharge, and domino evaluate was implemented in the first ALU design. Only five clocks were distributed to the ALU and all other ORed and ANDed clocks were generated directly at the dynamic gates. Unfooted dynamic gates and compound static gates were interspersed with footed dynamic gates and inverters. Furthermore, unfooted gates used a delayed phase clock, matched to approximate the evaluation delay of the previous gates in the same phase [8], in order to minimize short circuit currents without impacting performance. The method of staggering footed and unfooted dynamic gates as discussed in Section 3.5.3 was not used. Dynamic inverters and buffers were used to pass data through phases that perform no logic to avoid race-conditions. An odd bitslice of the CL-domino ALU with the clocks assigned to each stage is shown in Fig. 4.1. The dynamic gates that implement input complemented and/or non-monotonic logic are denoted in the figure. The critical path through the adder core is from the 2:1 mux through the carry merge tree to the XOR gate that produces the final sum. Thus nine dynamic and static gate stages must execute in $3 \cdot T_c/N \ ps$ before the rising edge of Φ_5.

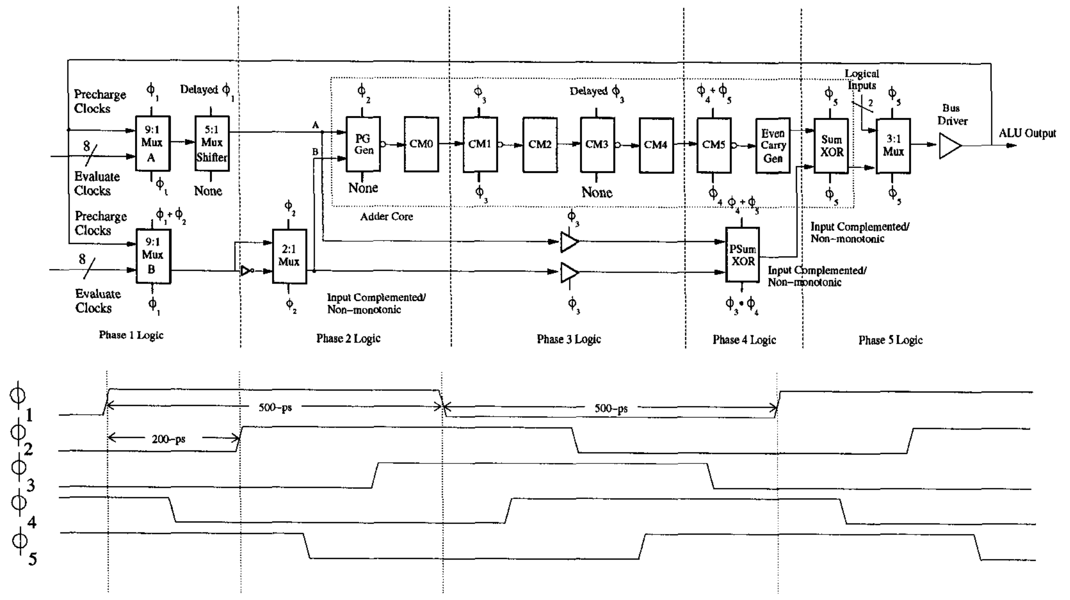

Figure 4.1: CL-domino ALU odd bitslice.

Dual-Rail Skew-Tolerant Domino ALU

A dual-rail skew-tolerant domino pipeline was implemented in the second ALU design. A dual-rail ALU is used in most commercial microprocessors due to the non-monotonic nature of the XOR functions needed to implement addition [19, 22, 28, 42, 66]. In the case of the Itanium 2, dual-rail signals are only generated before the adder core through a dual-rail pulsed entry latch.

Although many differential domino logic families exist [58, 64, 65], Domino Dynamic Cascode Voltage Switch Logic (DDCVSL) is still the simplest, most robust, and is among the fastest. As in the CL-domino ALU, footed and unfooted dynamic gates and compound static gates were used extensively in the dual-rail skew-tolerant ALU. An odd bitslice of the dual-rail DDCVSL ALU utilizing five-phase clocks is shown in Fig. 4.2.

Static CMOS ALU

The static CMOS logic style was used for the third ALU design. Although static CMOS is usually 1.5-2.0X slower than dynamic logic [43], it still is important for many high-performance microprocessor designs [17, 28, 25] because of its better noise margins, design simplicity and support by synthesis tools. Furthermore, static circuits dissipate less power

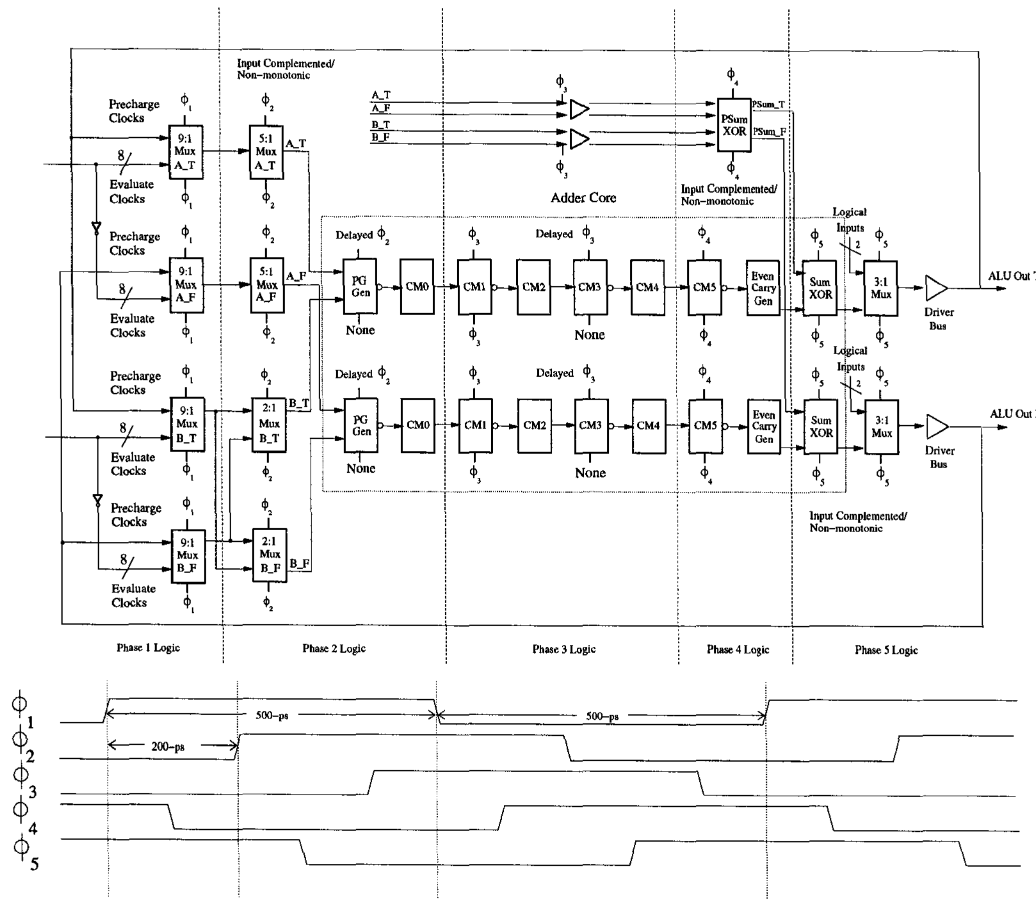

Figure 4.2: Dual-rail skew-tolerant ALU odd bitslice.

than dynamic circuits, assuming a switching activity factor of 0.5, because they switch only when their inputs change while dynamic circuits continuously cycle through precharge and evaluate operations. Furthermore, no clock power is consumed. This ALU was not micropipelined since static CMOS has no implicit latching and inserting latches in between static circuits would incur too much overhead. Therefore, the static CMOS ALU is a purely combinational circuit. A single bitslice of a static CMOS ALU is shown in Fig. 4.3.

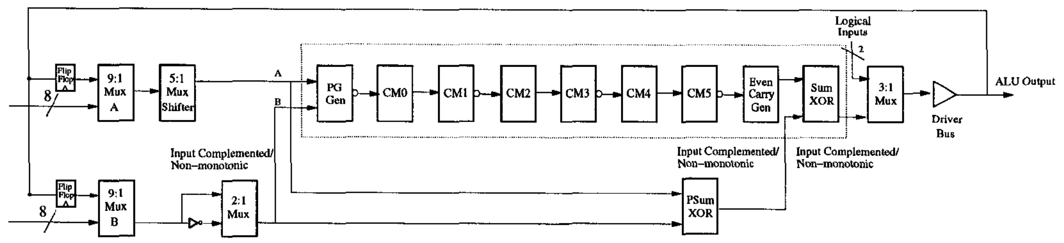

Figure 4.3: Static CMOS ALU bitslice.

4.2.2 Adder Core

Background

The 64-bit adder core is the primary component of an ALU and its architecture is critical in determining its performance and power requirements. Although many adder architectures exist [6, 8, 60], only a few are predominantly used inside modern microprocessors.

In the first generation Alpha design by DEC/Compaq (21064) [21], the 64-bit adder used Manchester Carry Chains (MCC) at the eight-bit level with Carry-Look-ahead Addition (CLA) and Conditional-Sum Addition (CNSA) on the least and most significant 32-bits respectively. Furthermore, a Carry-Select-Adder (CSLA) was used to produce the results in the upper 32-bits. In more recent Alpha implementations (21164) [42], a two level carry select scheme is used.

For recent IBM 64-bit PowerPC and the Power4 microprocessors [19, 25], the 64-bit ALU is implemented so that the lower 24-bits are calculated faster than the higher order 40-bits because the lower order bits are required to access the translation-lookaside-buffer (TLB) and the cache memory. In the Northstar PowerPC architecture, a CSLA architecture is used where carry selection is performed at the eight-bit level. These eight bit slices were built from simple-function dynamic circuits. In later designs, such as the Pulsar PowerPC, the Istar PowerPC [2], and the Sstar PowerPC [33, 66], the low order 24-bits were generated with a three-level CLA and the remaining high-order 40-bits generated by the same CSLA logic as the Northstar PowerPC. The Power4 uses a carry-lookahead Ling adder for its low order 24-bits. It should be noted that the Istar, Sstar, and Power4 processors are built in partially-depleted silicon-on-insulator (SOI), which boasts higher logic speeds at the expense of increased design complexity and decreased circuit yield [11].

A recent Hewlett-Packard/Intel IA-64 processor (Itanium 2) implements 64-bit and 32-bit ALUs which use a combination of Carry-Save Adders (CSA) and carry-look-ahead adders with simplified 4-bit carry Ling term [22].

The Intel Pentium IV incorporates a 32-bit adder core using a simpler propagate-generate-kill addition algorithm (PGK) on 16-bit slices at twice the core clock frequency [23]. The low-order 16-bits are needed at one time to begin access of the $L1$ data cache when used to calculate an address. Because the low-order and high-order 16-bits are computed at twice the rate of the core clock and can immediately feed a dependent operation in the second half of the core clock cycle, read-after-write (RAW) hazards between two simultaneously issued and dependent instructions are avoided in this superscalar architecture.

Some processors have adopted an alternative approach for addition based on recurrence solvers of which the Kogge-Stone [50] and Han-Carlson [41, 67] adders are the most popular. For example, the 32-bit floating point adder on the Intel IA-32 (Pentium II and Pentium III) uses a Kogge-Stone architecture to support single instruction, multiple data (SIMD) floating-point data types [63]. An IBM 64-bit PowerPC processor also uses a Kogge-Stone architecture in its fixed-point execution unit [26]. Sun Microsystems' third-generation SPARC V9 processor (UltraSPARC III) uses a modified Kogge-Stone carry chain for its high performance single-cycle ALU [28]. Intel has demonstrated 32-bit and 64-bit versions of single-cycle latency and throughput ALUs based on Han-Carlson adder cores [30, 32] for possible use in future microprocessors.

Implemented 64-Bit Han-Carlson Adder Core

It was assumed in our implementation, that all bits of the ALU are equally critical and therefore the results from the higher order bits are required as fast as the results of the lower order bits. A recurrence solver-based adder was chosen based on its highly regular layout properties, good control of fan-out and fan-in to the carry generation gates, and its popularity in recent microprocessors [8].

Logarithmic look-ahead adders are often referred to as recurrence solver-based adders or parallel-prefix adders [8, 60]. They function in a way such that the carries are produced in $O(log_2k)$ stages, where k denotes an k-bit adder. In essence, the "recurrence solver" are simply a variation of the many possible CLA topologies, where $Cin = 0$ to the least significant bit, or LSB, is assumed. The carry-look-ahead equations rewritten in the form of a recurrence is:

$$(g,p) \bullet (g,p) = (g + p\overline{g}, p\overline{p}). \tag{4.1}$$

The main disadvantage of the recurrence solver-based adders, is their inability to handle subtraction in a single ALU cycle if the result is required in two's complement form. An exception to this is when the ALU core is double clocked with-respect-to the rest of the logic as is the case in some Intel architectures [30, 32]. ALU subtract operations therefore must execute in two clock cycles through adding the true value of a first input operand to the complemented value of a second input operand in the first cycle, to obtain a one's complement result, and then adding '1' to the one's complement result in the second cycle to obtain the two's complement result. The inability to execute single cycle subtract operations is usually not detrimental to processor performance since a 64-bit ALU covers the entire range of possible memory addresses (i.e. all addresses can be calculated through additions) and most integer subtract operations can be easily recoded into additions through the compiler or in hardware. Furthermore, subtracted data in one's complement form can be recomputed into two's complement form where necessary, possibly using a simple incrementor.

A recurrence solver-based adder computes the sum in three steps:

(1) A preprocessing step where the propagate and generate terms are calculated for the input operands according to:

$$\begin{aligned} g &= a \cdot b \\ p &= \begin{cases} a + b & : \text{ if psum used} \\ a \oplus b & : \text{ if psum not used.} \end{cases} \end{aligned} \tag{4.2}$$

The propagate term (4.2) can be obtained through the OR operation since carry generation does not require the use of an XOR term. An OR operation is good since it is a monotonic, non-inverting function which can be implemented in conventional single-rail domino logic as opposed to the XOR operation.

(2) A processing step that generates the carry bits using a parallel prefix algorithm. This step is often referred to as carry merging.

The partial sum (psum) could also be obtained in parallel during this step, if the OR operation is used for calculation of carry propagate (4.2). This is appropriate since the partial sum can be calculated much faster than the carry bits and therefore the partial sum XOR operation is not on the critical path. The partial sum can be calculated from the following equation:

$$psum = a \oplus b. \tag{4.3}$$

(3) A postprocessing step where the final sum is calculated from the carry bits:

$$sum = \begin{cases} psum \oplus c_{in-1} & : \text{ if psum used} \\ p \oplus c_{in-1} & : \text{ if psum not used.} \end{cases} \tag{4.4}$$

While the Kogge-Stone and Han-Carlson adder architectures are similar, a recent study of both has shown that a radix-2 Han-Carlson adder dissipates around 57% of the energy of a Kogge-Stone adder [30]. This is because the carry-merge (carry generation) is performed on alternating bitslices instead of on every bitslice. This reduces the number of logic gates by approximately one-half at the cost of an extra stage delay in the carry merge tree. This results in the carries being generated in $O(log_2k + 1)$ stages. The interested reader can find more information about recurrence solver addition in the available literature [59].

The chosen Han-Carlson adder core comprises of only nine primary types of cells. Logic gate variants include footed dynamic, unfooted dynamic and static CMOS so that the carry merge is accomplished in both the dynamic and static stages using compound domino logic [45]. Static and dynamic inverters are further added to the paths to pass data between stages on bitslices where carry merging does not take place. The dynamic inverters functioned as latches which fed into compound domino logic. The logic equations for the primary types of cells are:

1. PG Gen

$$\begin{aligned} \overline{p_{i,1}} &= \overline{a_i + b_i} \\ \overline{g_{i,1}} &= \overline{a_i \cdot b_i}. \end{aligned} \tag{4.5}$$

2. Black Cells (Negative Input)

$$\begin{aligned} p_{j,2k} &= \overline{\overline{p_{i,2k-1}} + \overline{p_{j,2k-1}}} \\ g_{j,2k} &= \overline{(\overline{p_{j,2k-1}} + \overline{g_{i,2k-1}}) \cdot \overline{g_{j,2k-1}}} \end{aligned} \tag{4.6}$$

3. Black Cells (Positive Input)

$$\begin{aligned}\overline{p_{j,2k+1}} &= \overline{p_{j,2k} \cdot g_{i,2k} + g_{j,2k}} \\ \overline{g_{j,2k+1}} &= \overline{p_{i,2k} \cdot p_{j,2k}}\end{aligned} \tag{4.7}$$

4. White Cells

$$\begin{aligned}\overline{p_{i,k}} &= \overline{p_{i,k-1}} \\ \overline{g_{i,k}} &= \overline{g_{i,k-1}}\end{aligned} \tag{4.8}$$

5. Sum Cells

$$psum_i \mid sum_i = a_i \oplus b_i \mid psum_i \oplus c_{i-1} \tag{4.9}$$

The adder core was constructed using carry merging on odd bitslices, skipping even carries, as in [41], instead of carry merging on even bitslices as in [30, 32]. This reduces the number of carry merge bitslices by one. Minimum sized static and dynamic inverters pass the p and g signals from the even bitslices to the next logic stage. This implementation assumes that the inputs to the adder are available in true form. Following the propagate/generate stage, 64-bit carries are generated in six logic stages with an extra carry merge logic stage required to generate the missing even carries at the end of the carry merge tree. These carries can then be combined with the precomputed partial sums to obtain the final sum in a last stage. Therefore, all of the sum bits are computed in nine gate stages. In the PG Gen, the fanout of $\overline{p_{i,1}}$ is limited to two gates while the fanout of $\overline{g_{i,1}}$ is limited to one gate. For the other carry merge stages, p is limited to a fanout of three gates while g is limited to a fanout of two gates. Since the p gates have more fanout than the g gates, the precharge transistors for these dynamic gates are upsized appropriately. The switched capacitance is negatively affected by the relatively long wires that route across the carry merge tree. These wires can span 32-bit slices in the worst case. However, careful routing and shielding of these signals can limit stray wire capacitance and coupling effects. A dynamic node spans no more than 16-bit slices in the worst case and the precharge transistors are upsized to account for these long wires.

The parallel prefix graph of the 64-bit adder core is shown in Fig. 4.4. In order to reduce the number of transistors used, the propagate term for each bit was only computed where necessary. For example, bit 59 does not require the propagate term after CM1. All the generate terms, however, are required to compute the carry out for that particular bit position. The carry out for the most significant bit 63 is discarded.

4.2.3 Adder Core Cells

From the equations in Section 4.2.2, the largest dynamic or static gates are only 2-wide in the adder core for minimum stray capacitance on the output nodes of the logic gates. Furthermore, to increase speed at the expense of robustness to noise, keepers were omitted from the dynamic gates of the carry merge tree.

CL-Domino Propagate/Generate Cells

The CL-domino propagate/generate cells implemented in dynamic logic are shown in Fig. 4.5.

Dual-Rail Skew-Tolerant Domino: Propagate/Generate Cells

The dual-rail skew-tolerant domino propagate/generate cells implemented in dynamic logic are shown in Fig. 4.6.

CL-Domino: Dynamic and Static Carry Merge Cells

In order to increase the speed of the adder core in CL-domino, series clock transistors were removed from the critical path as much as possible. An example of this occurs when considering the use of OR-precharge in the 2:1 multiplexer before the non-monotonic PSUM gate. The series transistors were moved from the 2:1 multiplexer to the PSUM gate, which was clocked with AND-evaluate, since that XOR gate is not on the critical path. The CL-domino compound static carry-merge cells are shown in Fig. 4.7. The CL-domino footed and unfooted dynamic carry-merge cells are shown in Fig. 4.8, while the carry-merge cell that uses OR-precharge before the final sum is shown in Fig. 4.9.

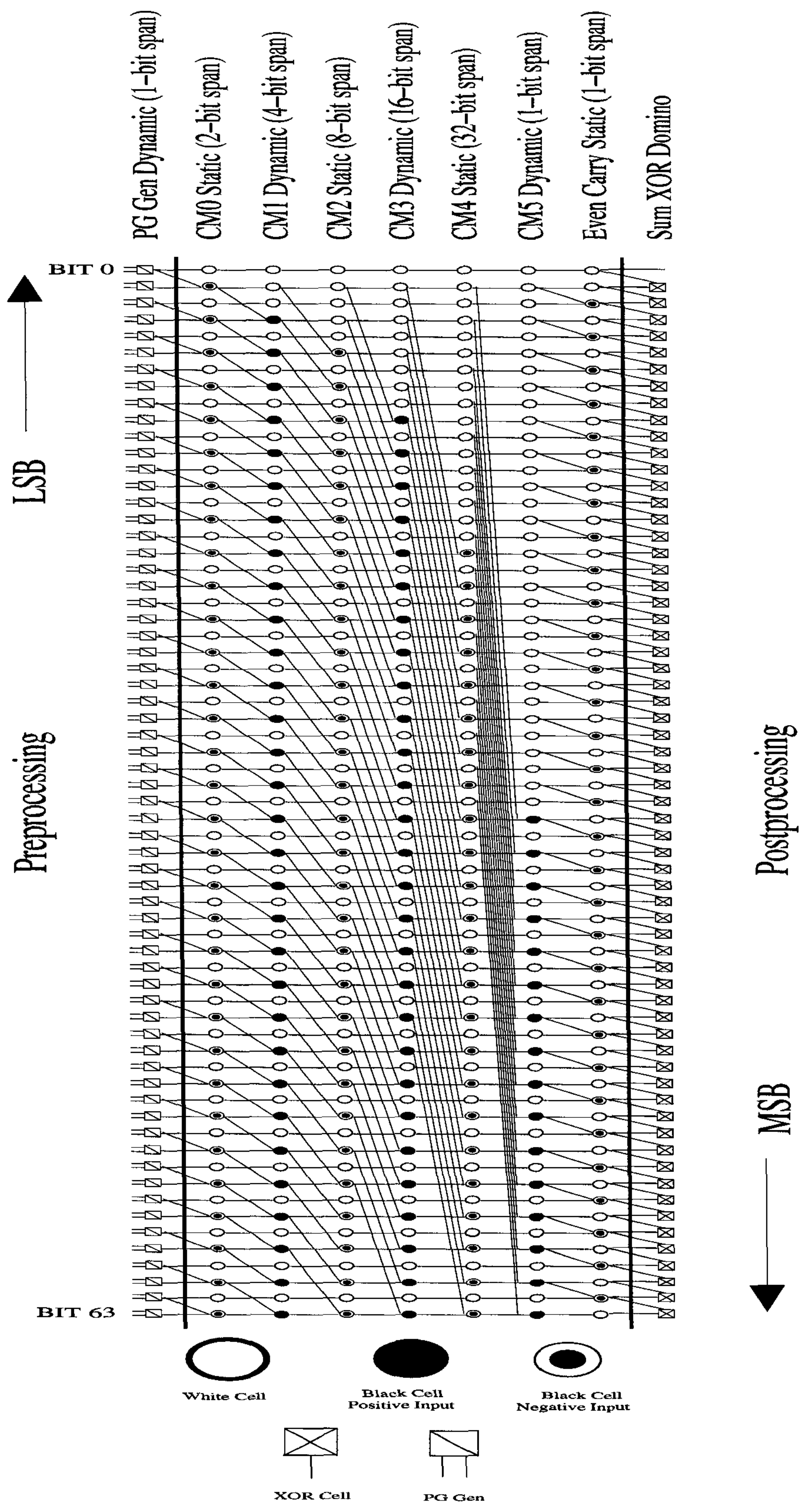

Figure 4.4: 64-bit Han-Carlson adder core architecture.

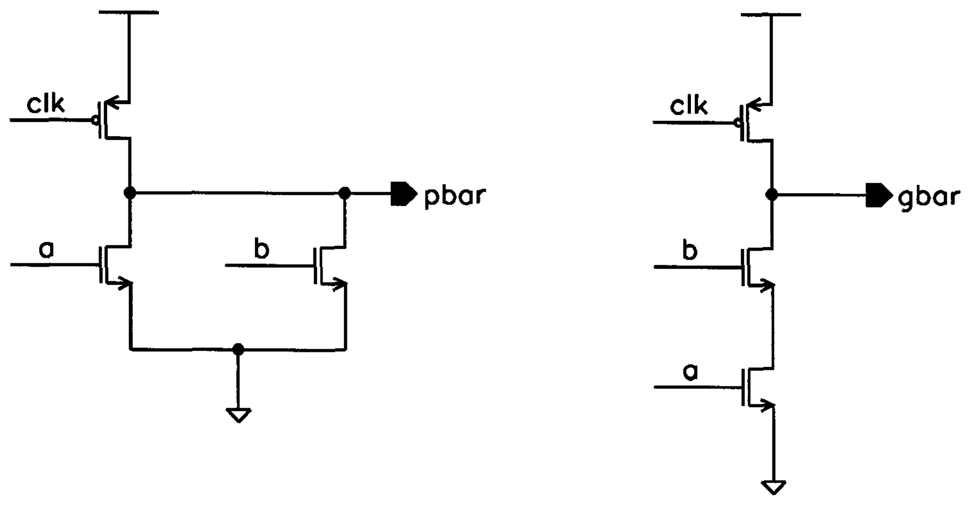

Figure 4.5: Unfooted CL-domino: propagate/generate cells.

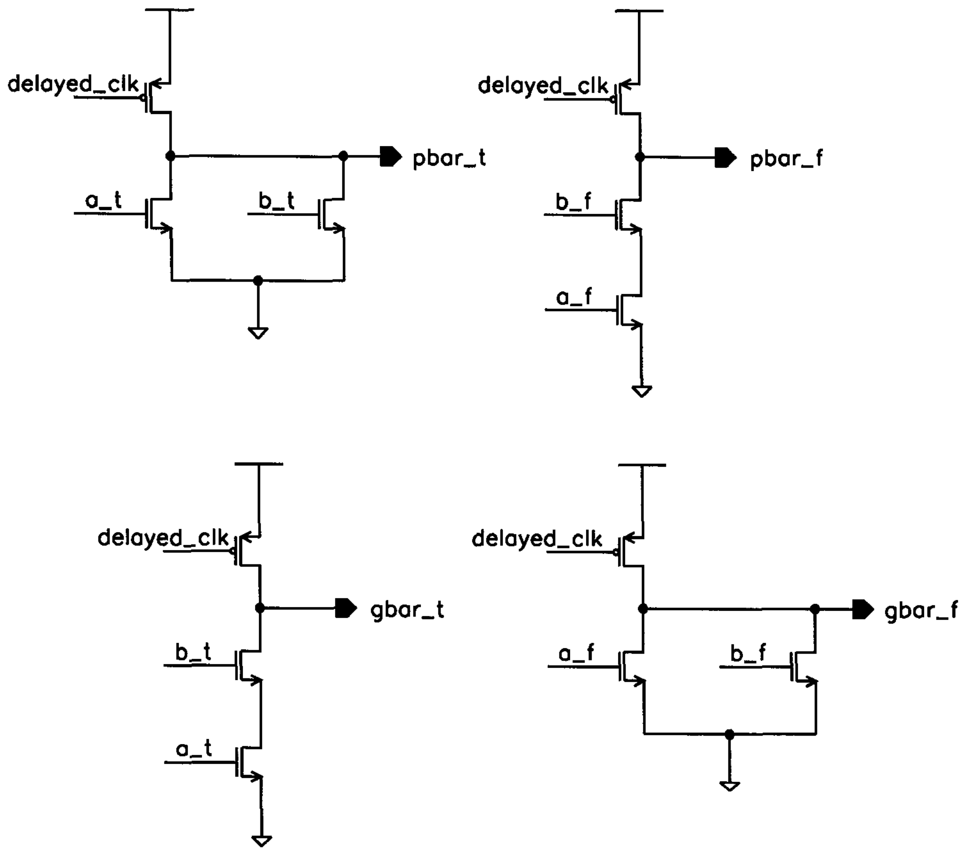

Figure 4.6: Dual-rail skew-tolerant domino: propagate/generate cells.

Dual-Rail Skew-Tolerant Domino: Dynamic and Static Carry Merge Cells

The dual-rail compound static carry-merge cells are shown in Fig. 4.10.

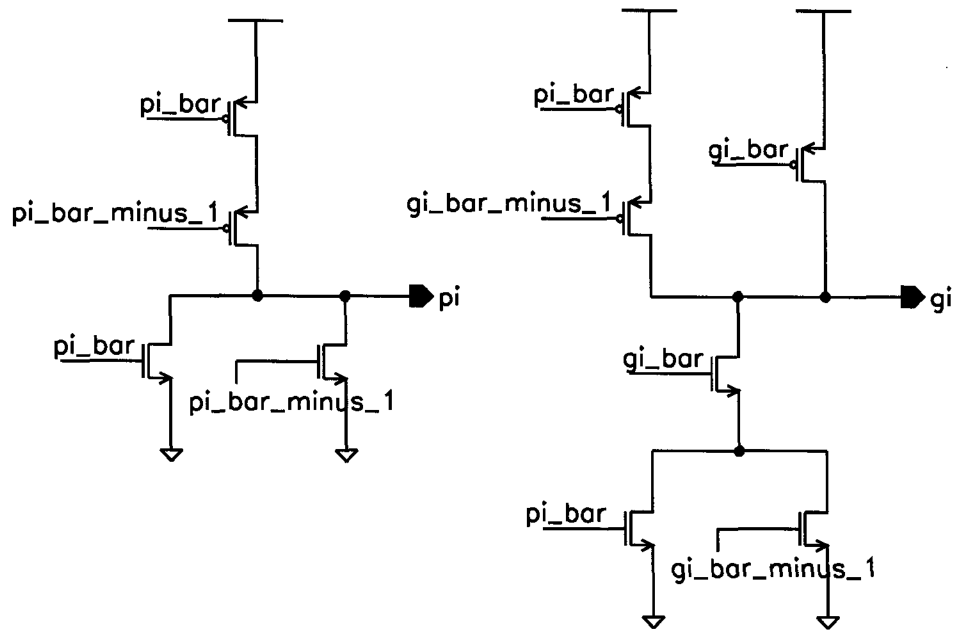

Figure 4.7: CL-domino: static carry-merge cells.

The dual-rail footed dynamic carry-merge cells are shown in Fig. 4.11, while the unfooted dynamic carry-merge cells are shown in Fig. 4.12.

CL-Domino: XOR Cells

The CL-Domino XOR gate is the true logic network of the high-performance XOR gate proposed in [49]. This XOR gate was used for both the calculation of the partial sum and final sum. The partial sum XOR gate uses OR-Precharge / AND-evaluate with internal node precharging as shown in Fig. 4.13, while the final sum used domino-precharge/domino-evaluate as shown in Fig. 4.14.

Although more complicated XOR circuits boasting higher performance using precharged pass-gates and precharged Wang's XOR [73] have been recently proposed [30, 32], we decided to keep our design simple and robust.

Dual-Rail Skew Tolerant Domino: XOR Cells

The dual-rail skew tolerant domino XOR gate is the commonly known dual-rail XOR with shared evaluation terms [43] This circuit is shown in Fig. 4.15.

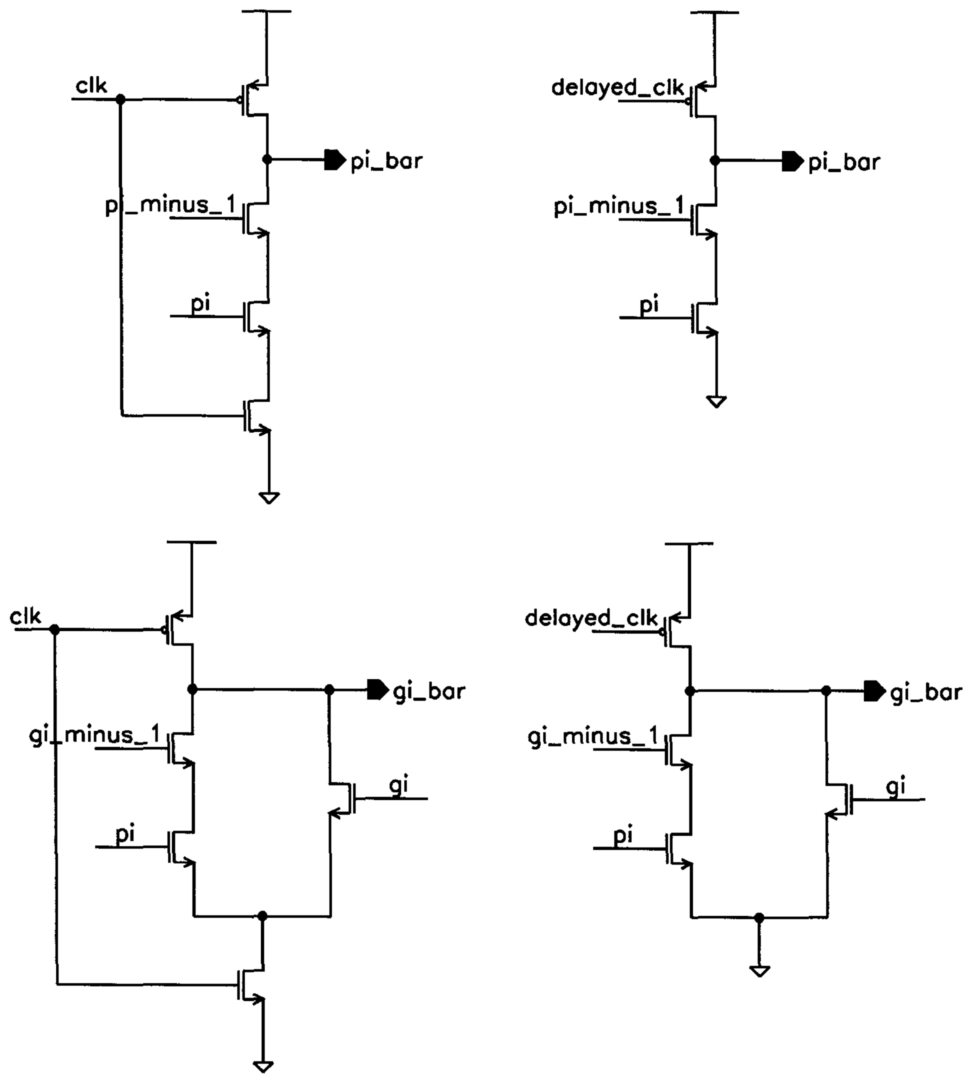

Figure 4.8: CL-domino: dynamic footed and unfooted carry-merge cells.

4.2.4 Multiplexers

The multiplexers are on the critical path of the ALU and must therefore be designed for minimum delay. While transmission gate multiplexers have been used in other ALU designs [30], it is more beneficial to implement dynamic multiplexers since they can help balance the pipeline through time borrowing. The multiplexers are domino logic with one-hot encoded select inputs; meaning only one select bit goes high during the evaluate period as in [3, 22, 29]. The select inputs are at the top of the NMOS stacks and the data inputs are at

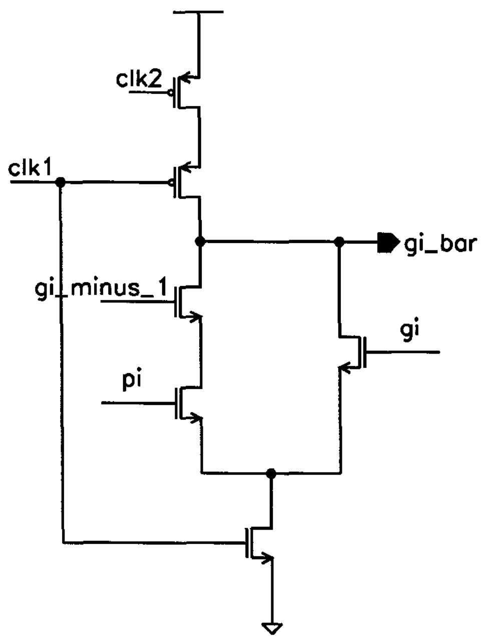

Figure 4.9: CL-domino: dynamic OR-precharge carry-merge cell.

the bottom to minimize charge sharing. Most variants of the dynamic multiplexers contain standard PMOS half keepers to increase noise robustness while CL-domino multiplexers with locally generated OR/AND clocks used full keepers. Each leg of the dynamic multiplexer used separate evaluation transistors thus improving evaluate and precharge delays and reduced charge sharing at the expense of extra clock load.

9:1 Input Multiplexers

The 9:1 input multiplexers, used for forwarding and selecting data from register files and FIFOs, are fairly wide OR-type structures, well suited for dynamic logic. The two variants used for all circuit designs are footed domino with domino precharge as shown in Fig. 4.16 and footed domino with OR-Precharge as shown in Fig. 4.17. The OR-Precharge variant is only used in the CL-domino ALU while the other one is used in both CL-domino and dual-rail skew tolerant domino ALUs.

In deep sub-micron processes, increased leakage currents and greater noise sensitivity of wide fan-in gates can often become problematic. To reduce the number of parallel pull-down paths, a technique of splitting the storage node and then combining the storage nodes

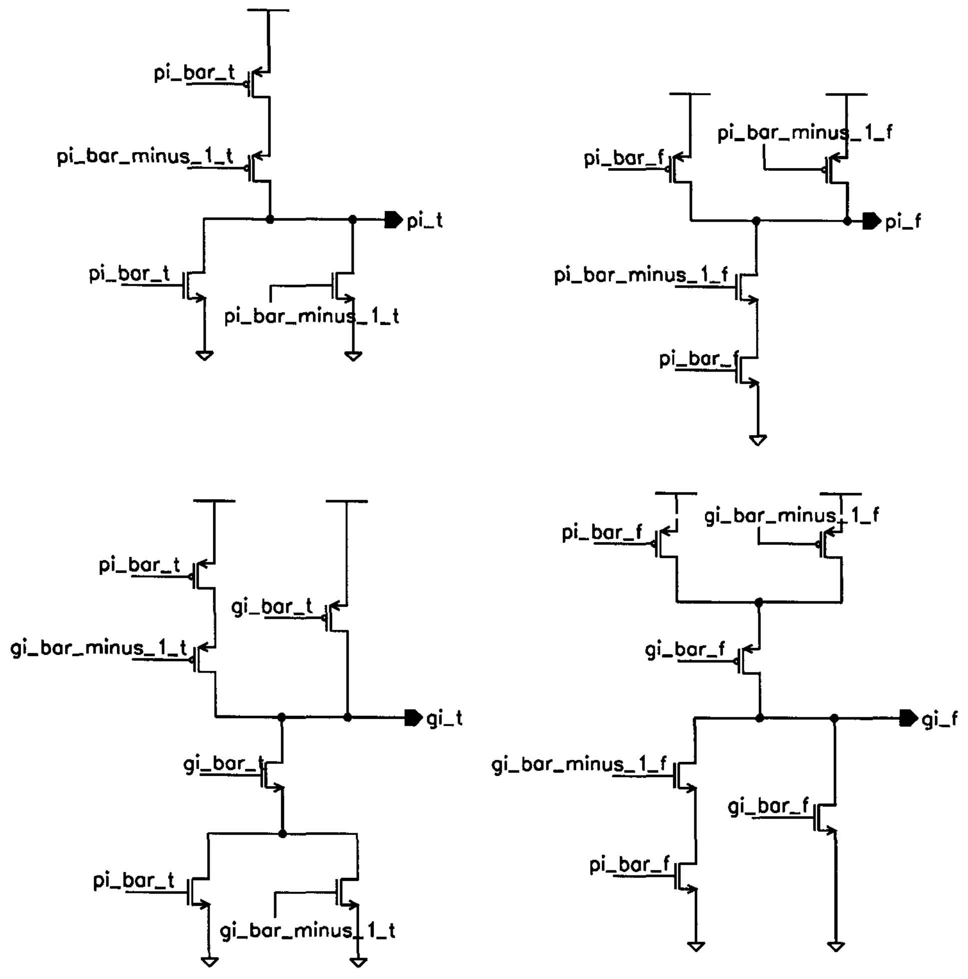

Figure 4.10: Dual-rail skew-tolerant domino: static carry-merge cells.

with an output static NAND gate, much like compound domino, is commonly used [22, 29]. However, we did not implement wide multiplexers in this manner since they consume more area and are more difficult to layout.

2:1 Inverting Multiplexer

The 2:1 multiplexer, used for implementing subtraction, is a 2-wide domino circuit that accepts true and complemented values of the 9:1 'B' multiplexer output. In single-rail CL-domino, one of its inputs is fed by an even number of inverters after the preceding dynamic gate and therefore is input complemented. This imposes a non-time-borrowing penalty through that 2:1 multiplexer. In dual-rail domino, the true and complemented outputs of

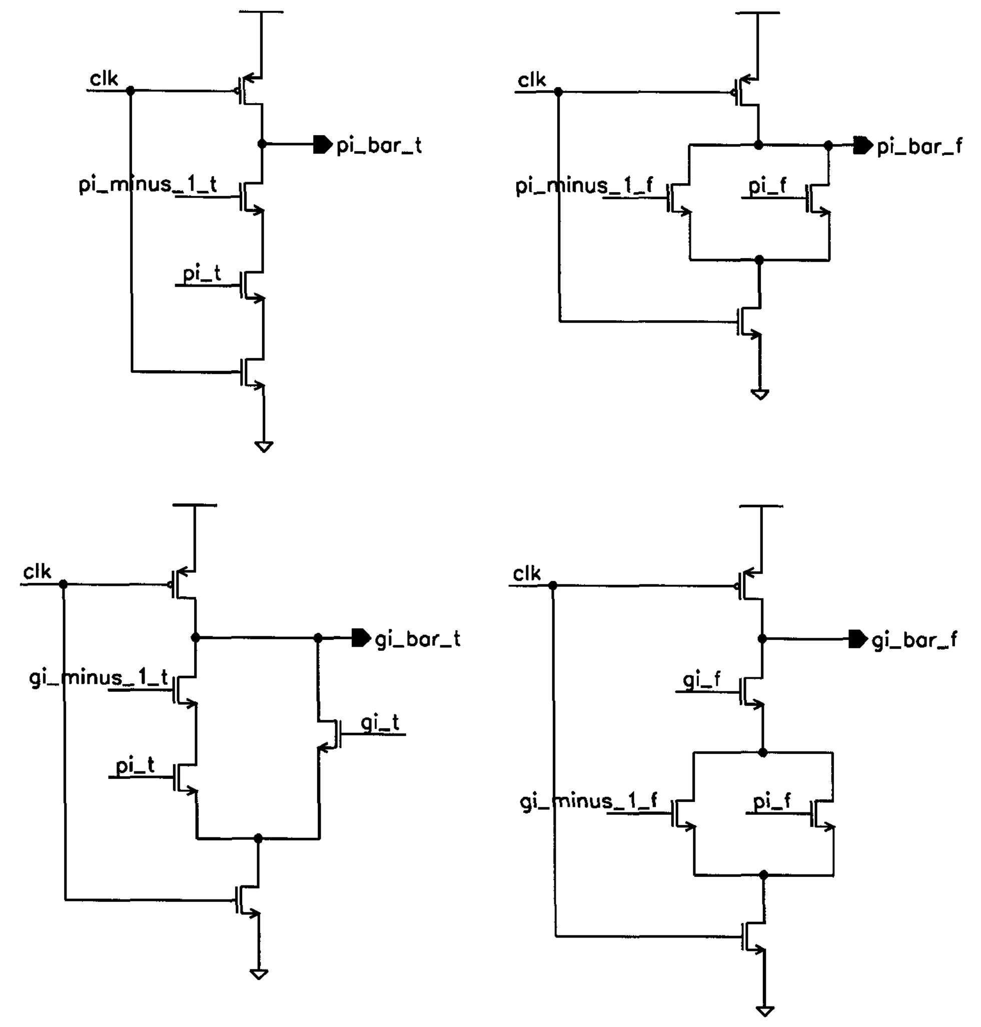

Figure 4.11: Dual-rail skew-tolerant domino: dynamic footed carry-merge cells.

the 9:1 'B' multiplexer are available and no time borrowing penalty exists at the expense of almost twice the number of transistors and correspondingly increased power dissipation and performance degradation due to increased routing complexity. The 2:1 multiplexer is shown in Fig. 4.18.

4.2.5 Variable Shifter with 5:1 multiplexer

The 5:1 multiplexer is a 5-wide domino circuit that accomplishes shifting of the 9:1 'A' multiplexer output operand through wiring. Each output bit of the 'A' multiplexer is wired to the inputs of data transistors in 5 adjacent 5:1 multiplexer bit slices to accomplish left

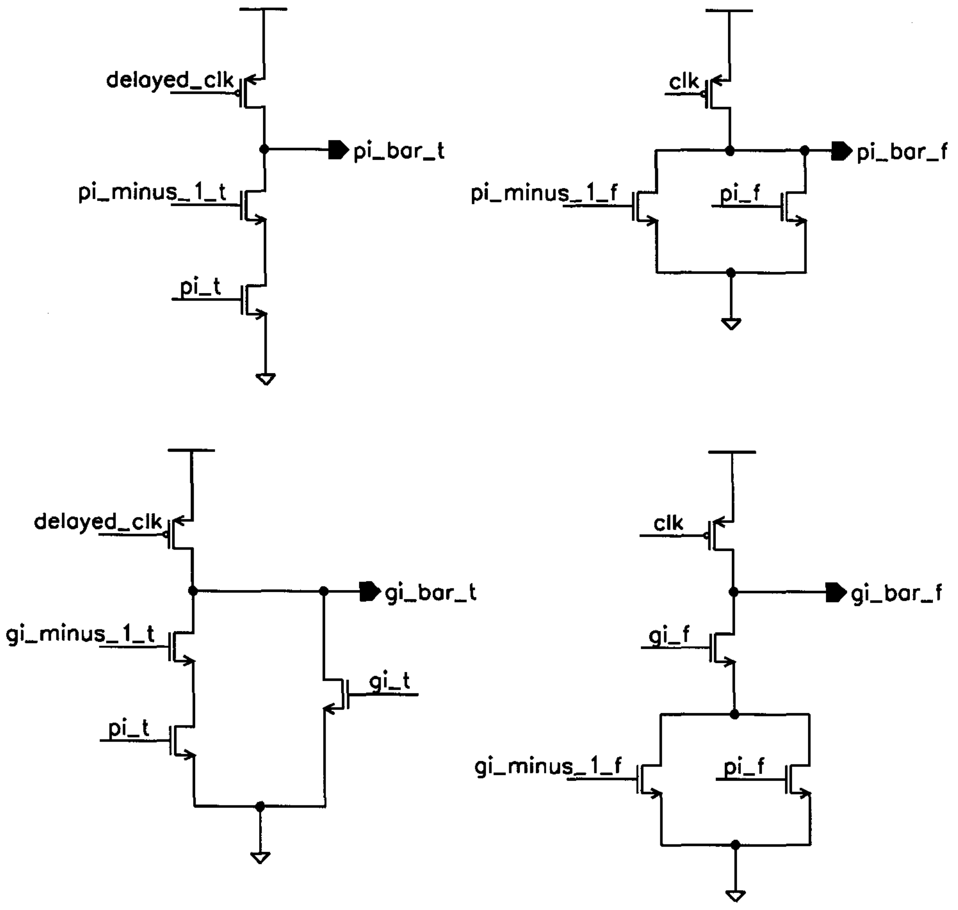

Figure 4.12: Dual-rail skew-tolerant domino: dynamic unfooted carry-merge cells.

shifting by 0-4 bits. The most significant bits that are shifted out are discarded. While other types of shifters exist, such as barrel shifters or logarithmic shifters [60], they are based on pass-transistor techniques and thus do not have the benefits of time borrowing to balance pipeline stages as in domino logic. The unfooted domino version of the 5:1 multiplexer as used in the CL-domino implementation as shown in Fig. 4.19 and the footed domino version as used in the dual-rail skew tolerant domino implementation is shown in Fig. 4.20.

4.2.6 3:1 Output Multiplexer

The 3:1 multiplexer is a 3-wide domino circuit that selects the output of the ALU from the adder or two other simpler functional units. These other functional units were not designed for these experiments since they are not on the ALU critical path. Examples of simple

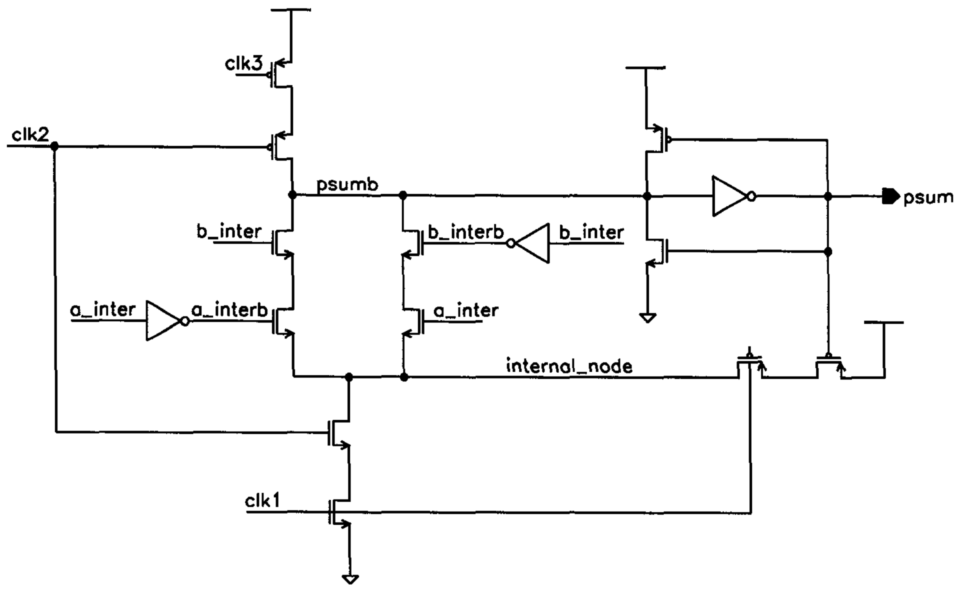

Figure 4.13: CL-domino: PSUM cell.

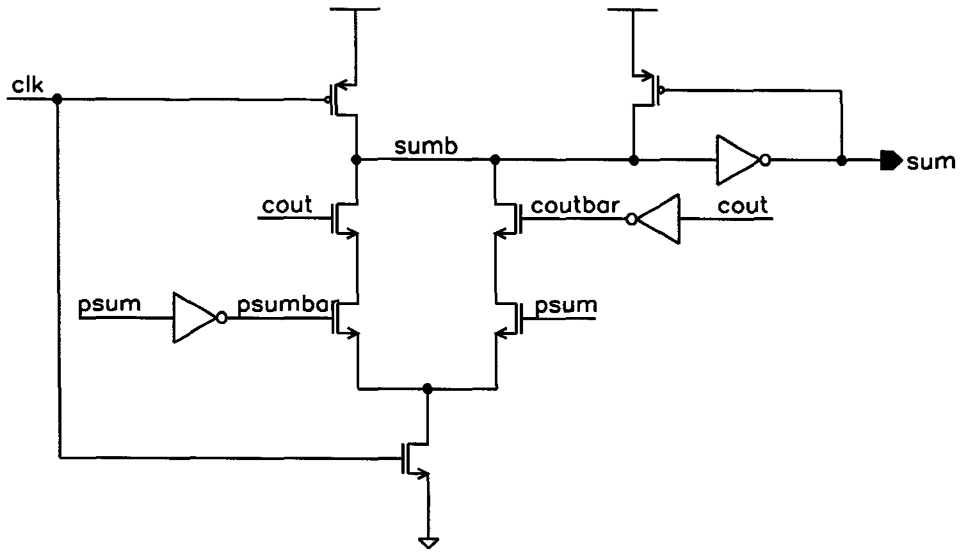

Figure 4.14: CL-domino: SUM cell.

functional units include zero detectors, counting leading zeros or rotators. The 3:1 output mux is shown in Fig. 4.21.

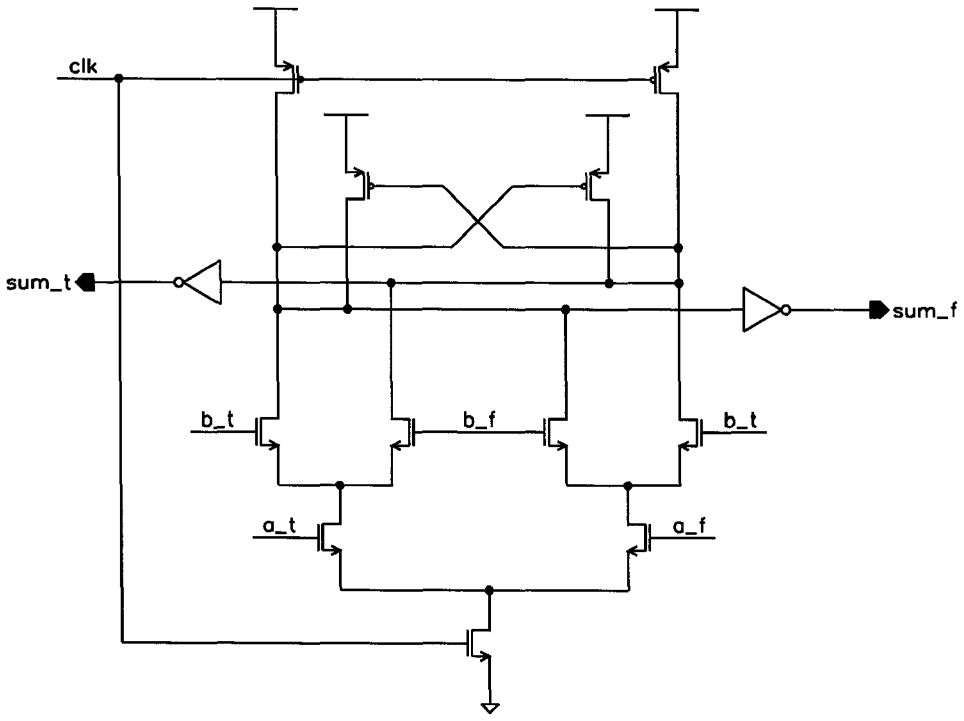

Figure 4.15: Dual-rail skew-tolerant domino: PSUM and SUM cell.

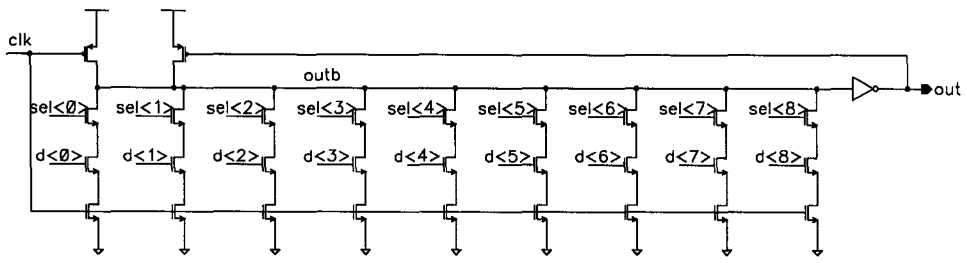
clk
outb
out
sel<0> sel<1> sel<2> sel<3> sel<4> sel<5> sel<6> sel<7> sel<8>
d<0> d<1> d<2> d<3> d<4> d<5> d<6> d<7> d<8>

Figure 4.16: 9:1 input multiplexer.

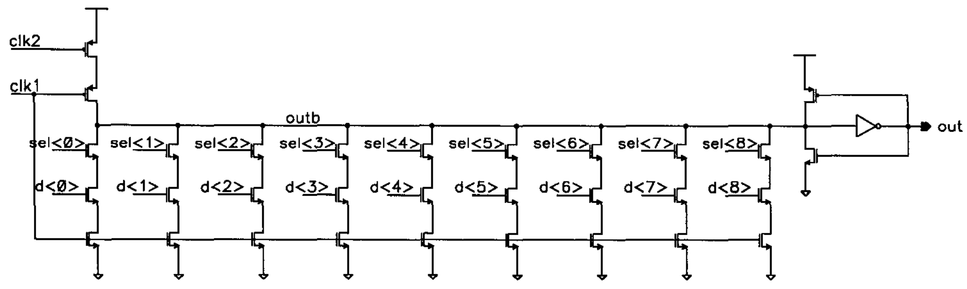
clk2
clk1
outb
out
sel<0> sel<1> sel<2> sel<3> sel<4> sel<5> sel<6> sel<7> sel<8>
d<0> d<1> d<2> d<3> d<4> d<5> d<6> d<7> d<8>

Figure 4.17: 9:1 input multiplexer with OR-precharge.

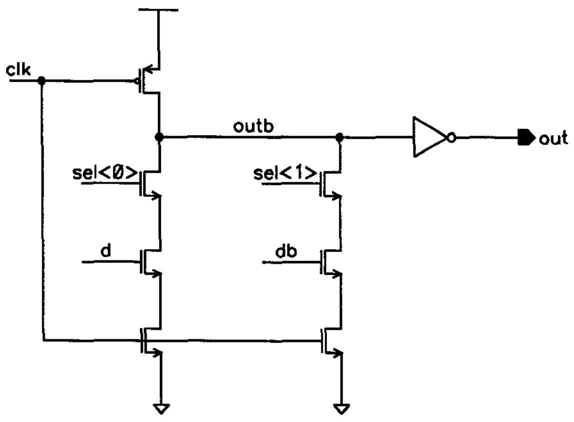

Figure 4.18: 2:1 multiplexer.

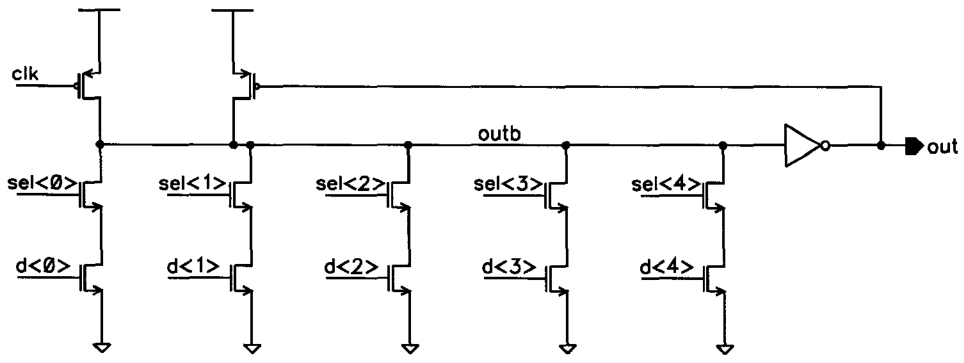

Figure 4.19: 5:1 unfooted variable shifter/multiplexer.

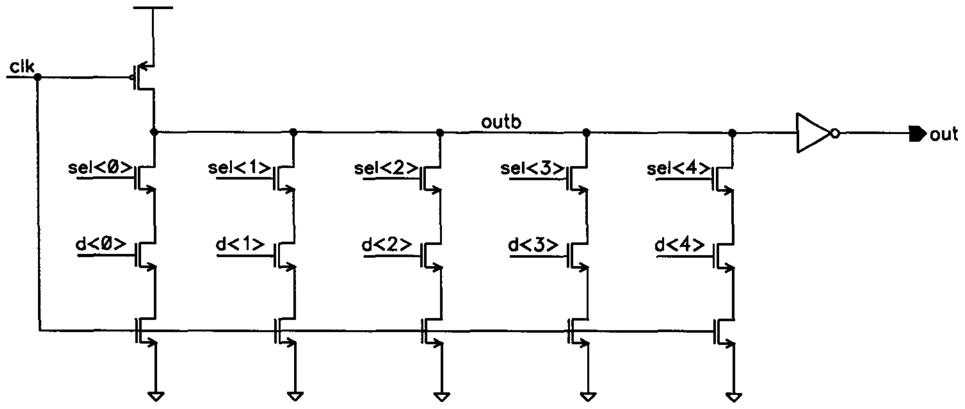

Figure 4.20: 5:1 footed variable shifter/multiplexer.

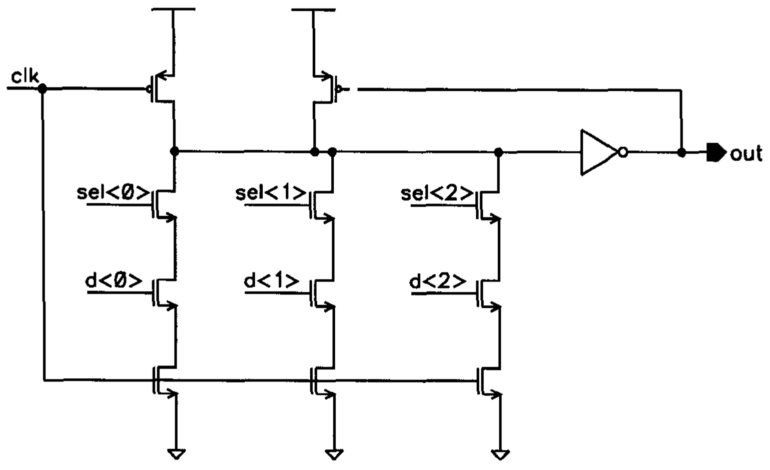

Figure 4.21: 3:1 output multiplexer.

4.2.7 Bus Driver

The ALU has to drive the inputs of other units used for integer execution such as register files, FIFOs, data caches and loopback buses [30, 32, 42]. This presents a fairly large capacitive load to the ALU outputs and hence a string of progressively larger inverters was used to drive the load. For our design, the string of inverters was folded into the static output inverter of the 3:1 multiplexer for domino logic compatibility. Thus any odd number of bus driver inverters would maintain the correct polarity. Since it was assumed that the load capacitance is approximately 50-fF for each bit, a three stage inverter design, with two inverters comprising the bus driver, was required. The scaling factor was 3.75 as suggested in [68] instead of the exponential factor as suggested in other literature [5]. Using a folded driver in this manner allowed the 3:1 output multiplexer to borrow evaluation time from the 9:1 input multiplexers when the loopback bus was enabled, thus mitigating timing constraints. The falling transition of the bus driver is now counted toward the precharge delay of the output 3:1 multiplexer since an odd number of inverters exist between the dynamic gate and the chain of static gates.

4.2.8 Other Circuit Design Considerations

Other design considerations included the extensive use of compound static gates inside the adder core, thus enabling carry merging to be accomplished in the static as well as the dynamic logic stages. This substantially improves circuit performance. The compound static gates inside the adder core favor rising outputs for evaluation at the expense of falling outputs for precharge. This was accomplished through sizing all the NMOS network transistors minimum width and sizing the PMOS network transistors for equal drive strength as a normally sized NMOS network.

The use of unfooted dynamic gates increases logic speed since the pull-down stack height is reduced and the unfooted gates can use smaller logic transistors thus resulting in smaller input loading. Unfooted dynamic gates, furthermore, reduce clock load thus saving clock power. Care was taken to size the precharge transistors of the unfooted gates to meet precharge timing, where some unfooted gates have less precharge time because they use OR-precharge. Since it is imperative that the inputs to the unfooted gates discharge quickly to minimize short circuit current and reduce precharge time, an unfooted dynamic gate must be preceded by a footed dynamic gate [29]. Moreover, most unfooted domino gates were clocked by a delayed version of the clock phase, as generated through passing the original phase clock through two simulated inverters. The propagation delay of this delay buffer is intended to match the delay of the dynamic and static gate of the previous footed domino gate thus effectively delaying the precharge clock edge to reduce power without impacting performance [8]. In our ALU designs, delayed clocks were slowed by the equivalent of two back-to-back inverters. The output of the inverters have ideal drive strength, as determined through the Cadence Mixed-Signal Interface, to minimize the design effort required for the clock drivers. The delays of these inverters will compare favorably against the logic delays for different process and environmental conditions since the precharge delay circuit is placed in close proximity to the logic gates and therefore both will be affected by the conditions equally. The only unfooted gate that was not clocked with a delayed phase clock was PG Gen. This was so that the adder performed better for best case data as the expense of extra short circuit power dissipation.

Tapered pull-down NFET stacks were used extensively in the ALU designs to reduce junction capacitance loading and increase speed [61, 62]. Tapering refers to increasing the size of each NMOS transistor that is one level deeper in the stack because those transistors must discharge more internal node diffusion capacitance than transistors higher up in the stack. Last, if an inverter was used as the static gate at the output of a dynamic gate, it was high-skewed with a ratio of 4:1 (P:N) to maintain a reasonable trip point for speed while maintaining improved noise immunity [43]. The inverters on non-timing critical paths were skewed with a 2:1 Beta ratio.

4.3 Simulation Study of 64-Bit ALUs

4.3.1 ALU and Adder Core Functional Verification

The ALUs were designed with a typical top-down microprocessor design flow [13]. The starting point of the design was a functional ALU specification, followed by a behavioral level model written in 'C' and then a functional model written in Verilog. The functional Verilog model of the adder core, which implemented the Han-Carlson carry merge algorithm, was verified for correctness against a Verilog model of the adder core implemented with the '+' operator. Once all of the above steps were satisfied for the ALU design, the CL-domino and dual-rail skew-tolerant domino ALUs were implemented at the transistor circuit level while the static CMOS ALU was synthesized into a netlist comprising of gates from a standard cell library.

Before CPU and memory intensive transient analysis of the dynamic ALUs was attempted, the adder core and ALU were checked for functional correctness at the Verilog switch-level. The two dynamic ALU schematics were extracted and each transistor was considered a primitive element modeled by one of four zero-delay switches: NMOS, PMOS, weak NMOS, weak PMOS. The "weak" switches were used when the transistors were configured as weak feedback elements so that the logic simulator (NC-Verilog) can resolve signal contention on nets that have multiple drivers. This occurred frequently in the adder and ALU for keepers that fed back to the outputs of dynamic gates. The Verilog switch-level models also included capacitance on the transistor gate terminals. This allowed nets coupled to transistor gates to store previously driven logic values to accommodate simulation of dynamic logic.

The functionality of all three adder cores was simulated for 100 random test vectors as generated by a 'C' program. The results of the schematic adders were compared to the functional Verilog model of the 64-bit adder. The Cadence Comparescan utility was used to compare the results of the schematic adder cores with the results of the functional model. This program was a waveform viewer which flagged differences between digital signal traces. All of the adder cores were completely verified for the generated input stimulus.

The functionality of the ALUs were verified for the first 28 random test vectors as generated by the aforementioned 'C' program. A behavioral Verilog stimulus file provided control for the ALU through issuing new data operands to the ALUs every global clock cycle and controlling the flow of data through the ALUs. This was mainly accomplished by asserting various multiplexer select lines at different points in the global clock cycle. Every path through the ALUs was tested through asserting each multiplexer control signal at least once during the course of 28 ALU cycles. All the multiplexer select lines, except for the output multiplexer, are asserted on the rising edge of Φ_1, while the output multiplexer is asserted on the rising edge of Φ_4. The schematic ALUs were compared against the results generated by a separate 'C' program that modeled the behavior of the ALU. The Cadence Comparescan utility, in conjunction with the stimulus file, was used for comparing the simulation results between the ALU and the 'C' program. All the ALUs were completely verified for the generated input stimulus.

4.3.2 ALU Functional Delay and Power Measurements

Methodology

Circuit entry was performed using the Cadence IC design tools version 4.4.6. The dynamic ALUs were simulated using the Cadence spectreVerilog mixed-signal simulator. A simulation run consists of a DC analysis followed by a transient analysis where a digital simulator (NC-Verilog) and an analog simulator (Spectre) fork different processes on a UNIX workstation and communicate results. Furthermore, mixed-signal simulation parameters were required to obtain the initial solution of a mixed-signal design. The accuracy of the transient analysis correspond to a selectable "Moderate" accuracy setting inside the Cadence Analog Environment. The Cadence Analog Environment was used to netlist the dynamic ALU designs. The Verilog functional description of the static CMOS ALU was exported to Synopsys, which in turn generated the static CMOS ALU netlist, using the Cadence to Synopsys interface.

The transient analysis results of the CL-domino and dual-rail skew-tolerant domino ALUs were analyzed using a combination of the Cadence Analog Waveform Viewer, Analog Waveform Calculator, SignalScan and the Results Browser. The results of the static CMOS ALU on the other hand was reported using static timing and power analysis in Synopsys Design Analyzer.

The target operating frequency for the dynamic ALUs, both CL-domino and dual-rail skew-tolerant domino, was 1-GHz under high skew conditions. The latency and throughput for a single add instruction was set to the clock frequency, meaning that the rate of operations completed and the number of new operations issued to the ALU is equal to the cycle time, 1-ns . Although both ALUs can function at higher frequencies, proper operation at the target frequency was sufficient to characterize it for intended use in a modern 64-bit microprocessor built in $0.18\ \mu m$ technology with aluminum interconnect.

This clock frequency may seem low compared with recent 32-bit microprocessors which have reached speeds of over 3-GHz [14]. It is, however, representative of 64-bit processors in comparable or smaller technologies with comparable or faster interconnects and/or low threshold voltage transistors for critical paths as shown in Table. 4.1.

Table 4.1: 64-bit microprocessors.

Microprocessor	Technology	Interconnect	Frequency	Reference
Intel Itanium	$0.18\ \mu m$ bulk-CMOS	6-layer aluminum	800-MHz	[62]
Intel Itanium 2	$0.18\ \mu m$ bulk-CMOS	6-layer aluminum	1.0-GHz	[29]
Sun UltraSPARC III	$0.13\ \mu m$ bulk-CMOS	7-layer copper	1.1-GHz	[24]
Fujitsu SPARC64 V	$0.13\ \mu m$ bulk-CMOS	8-layer copper	1.35-GHz	[51]
IBM POWER4	$0.18\ \mu m$ SOI-CMOS	7-layer copper	1.3-GHz	[25]
HP Alpha 21364	$0.18\ \mu m$ bulk-CMOS	7-layer copper	1.2-GHz	[15]
HP PA-8700	$0.18\ \mu m$ SOI-CMOS	7-layer copper	1.0-GHz	[69]

The target operating frequency of the static CMOS ALU, on the other hand, was not fixed at 1-GHz since we could not achieve that speed with the provided tools and standard cell library. In all likelihood, even a custom static CMOS implementation would not be able to achieve 1-GHz. Instead, constraints were set in the synthesis tool to minimize the delay through the ALU at the expense of extra area and power.

For the dynamic ALUs, the clock generator, developed in behavioral Verilog, was able to independently skew each clock by arbitrary positive and negative increments in relation to adjacent clocks. Since it was determined that the FO4 inverter delay in our process is 71.6-ps at 25°C, the skew between selected clocks was skewed by 72-ps. The worst-case skew was chosen to be 1 FO4 inverter delay since a well-designed local clock distribution network can bound the skew to within 1 to 2 FO4 inverter delays [42]. Moreover, recent publications have shown that the local skew inside microprocessors (incorporating an ALU) can be bounded even more tightly. Simulated and/or measured maximum local skew inside a 0.18 μm aluminum interconnect Pentium IV microprocessor is 38-ps [52], while the maximum local skew inside a 0.18 μm aluminum interconnect first generation Itanium microprocessor is 28-ps [31] and is 20-ps on a second generation Itanium 2 microprocessor [29]. The FO4 inverter delay in these processes was 70-ps [27] and therefore the maximum local skew is: 0.54 FO4 inverter delays in the Pentium IV, 0.40 FO4 inverter delays in the Itanium, and 0.29 FO4 inverter delays in the Itanium 2. The Pentium IV manages to reduce skew through delay-matched taps while the Itanium reduces the skew through active deskew compensation and feedback control. Last, the Itanium 2 uses carefully matched load and resistance-capacitance (RC) delays, special gater clock circuits and short local routes of less than 1000 μm to minimize skew. Thus the 1 FO4 skew between adjacent clocks in our ALU comparison is conservative.

The clocks were skewed according to Table. 4.2 for the CL-domino and dual-rail skew-tolerant domino ALUs. For our simulation studies, the clock generator produced 50% duty-cycle clocks and clocks with longer or shorter duty-cycles were generated directly at the dynamic gates. This is because it is common to generate 50% duty-cycle clocks from a PLL and in most cases the overlap provided by 50% duty-cycle clocks is sufficient to compensate for any skew and required time-borrowing [42, 43].

Table 4.2: Worst case skew parameters for ALU simulation.

Reference Clock	**Skewed Clock**	**Skew Direction**	**Primary Effects**
Φ_1	Φ_2	Positive	Reduces precharge time of 9:1 Mux B which uses OR-precharge
Φ_3	Φ_4	Positive	Reduces evaluate time of Psum which uses AND-evaluate
Φ_4	Φ_5	Positive	Reduces precharge time of Psum which uses OR-precharge reduces available time borrowing of Φ_5 from Φ_1

Both the ALU stimulus, that provided the ALU control signals and data operands, and the multi-phase clock generator were written in behavioral Verilog. These behavioral components exist in a digital simulation partition and therefore must be interfaced to the ALU which is modeled in an analog simulation partition. Furthermore, analysis of large binary numbers in the analog simulation domain is difficult and thus the ALU outputs must be interfaced back into the digital simulation domain for use in a logic analysis tool. Cadence provides this ability through its Mixed-Signal Hierarchy editor and D/A and A/D simulation interface elements.

The simulation D/A interface primitive performs the most basic digital-to-analog conversion step by converting a logic state to a voltage and timing relationship. The chosen D/A interface element was implemented as a boolean-controlled voltage source and an output resistor. The primary purpose of the resistor is to model the non-ideal output impedance of the digital input pin driving the digital-to-analog interface net. However, we chose to model the driving clock, control and data signal drivers as ideal voltage sources with approximately zero output impedance to simplify analysis. The input edge rate for clock and data inputs follows the example of [30]. The parameters for the D/A interface elements are summarized in Table. 4.3.

The simulation A/D interface primitive performs the most basic analog-to-digital conversion step by sensing voltage and converting it to a logic state. The chosen A/D element

Table 4.3: D/A simulation interface element parameters.

Parameter	Clock Outputs	Control and Data Outputs
Output Impedance	1 $a\Omega$	1 $a\Omega$
VOH	V_{dd}	V_{dd}
VOL	V_{ss}	V_{ss}
Rise Time $(VOL \rightarrow VOH)$	10 ps	50 ps
Fall Time $(VOH \rightarrow VOL)$	10 ps	50 ps

was implemented as an open circuit voltmeter. The interface element determines the voltage level across its positive and negative terminals and sends out a logic '0' if the voltage value is below VIL and sends out a logic '1' if its value is above VIH. If the value is above VIL and below VIH for a period longer than the transit time, $timex$, the interface element sends out a logic 'X'. We chose to approximate $timex$ by a value very close to zero so that the value read by the logic simulator would be the stable high or low output values of the analog circuit. The parameters for the A/D interface elements are summarized in Table. 4.4.

Table 4.4: A/D simulation interface element parameters.

Parameter	Analog Input Values
$timex$	1 as
VIH	2/3 V_{dd}
VIL	1/3 V_{dd}

Results

The comparison study involved three ALU designs while largely neglecting the parasitic effects introduced by wiring. Contacted diffusion parasitics were estimated for the transistor terminals and the electrical behavior of all transistors was described by industry standard BSIM3 physical device models made available by the foundry. For the static CMOS ALU, the effects of wiring were mostly neglected through specifying an optimistic, "TSMC 8K aggressive", wire load model since a zero wire load model could not be selected. Each output bit of the ALU, can be coupled to the inputs of register files, FIFOs and data caches, which might comprise the next stage in a microprocessor pipeline, in addition to a loopback bus for both 9:1 ALU input multiplexers. Excluding the input gate capacitance of the ALU multiplexers, a 50-fF load capacitor was used to approximate the capacitance of each bit of the loopback bus and that of the inputs to the next stage execution units. This value was estimated through comparing two ALU loopback buses, one for a 64-bit ALU of $1200\mu m$ length in a $0.18\ \mu m$ process and another optimized one for a 32-bit ALU of length $84\mu m$ in $0.13\ \mu m$ [30, 32]. All three ALUs were characterized under typical (NMOS/PMOS=TT/TT) transistor processing, a nominal 1.8-V operating voltage and a temperature of 25°C. The conditions under which each ALU was found to be functional are summarized in Table. 4.5. The row in the table for "Static CMOS 1" refers to a synthesized static CMOS design where optimization constraints were set to obtain the minimum delay through the ALU critical paths. The row in the table for "Static CMOS 2" refers to a synthesized design where optimization constraints were set so that 400-MHz operation was achieved and power is saved by not optimizing the design any further.

Table 4.5: Confirmed functional performance of different ALUs with no wiring parasitics except for output load.

ALU Design	Clock Skew	Operating Frequency	Result
CL-Domino	0 FO4	1-GHz	functional
CL-Domino	Worst-Case 1 FO4	1-GHz	functional
Static CMOS 1	N/A	427-MHz	functional
Static CMOS 2	N/A	400-MHz	functional
Dual-Rail Domino	0 FO4	1-GHz	functional
Dual-Rail Domino	Worst-Case 1 FO4	1-GHz	functional

Bits 62 and 63 of the adder core is on the critical path of the three simulated ALUs. The correct operation of these bits for the output of the CL-domino ALU is shown in Fig. 4.22. Correct operation of the CL-domino ALU under worst-case skew conditions is shown in Fig. 4.23

Power dissipation of the transistors due to: (a) the charging and discharging of capacitors (b) short-circuit currents (c) noise currents and (d) static leakage was measured separately from the switching power dissipation of the multi-phase clocks.

In the case of the CL-domino and dual-rail skew-tolerant domino ALUs, spectreVerilog saves the results of the power dissipation of the transistors as a waveform, representing

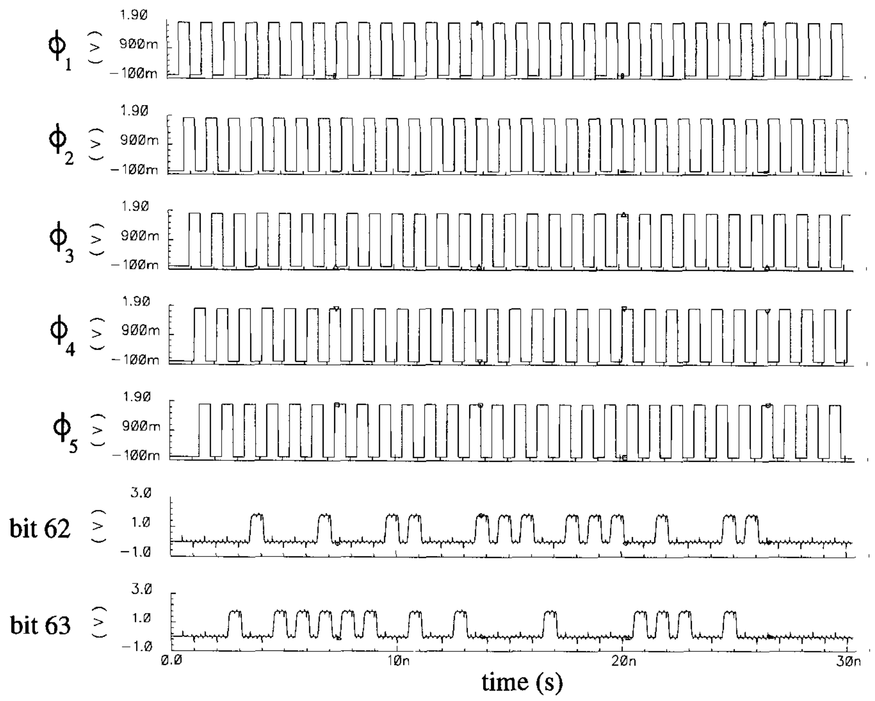

Figure 4.22: Clock-logic domino ALU critical path operation at 1-GHz operation and no clock-skew.

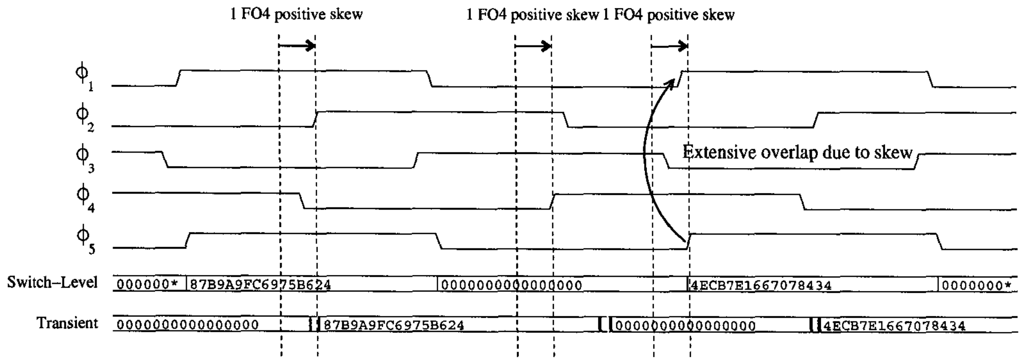

Figure 4.23: Clock-logic domino ALU at 1 GHz operation and worst-case 1 FO4 skew.

the instantaneous power dissipated in the circuit during the transient analysis. The average power was therefore computed as:

$$P_{ave\_transistors} = \frac{1}{T}\int_0^T p(t)\,\mathrm{d}t. \tag{4.10}$$

$p(t)$ represents the instantaneous power and T is the period of interest.

In the case of the static CMOS ALU, the power dissipation was measured from the static power analysis tool found inside the Synopsys Design Analyzer.

The switching power dissipation of the ALU operands was not measured, although the energy of those signals is small compared to the other sources of power dissipation.

The switching power dissipation of each clock phase and each delayed clock phase was measured from the waveforms of instantaneous current through each D/A element that served as a clock source. First the instantaneous current for each clock source was averaged over the period of interest and compared to zero to determine whether a purely capacitive load was being switched according to:

$$i_{test\_clock} = \frac{1}{T}\int_0^T i(t)\,\mathrm{d}t \cong 0. \tag{4.11}$$

The approximate equality in (4.11) was to account for rounding errors in the simulator. If (4.11) holds true, then the average clock current can be found from:

$$i_{ave\_clock} = \frac{1}{2}\frac{1}{T}\int_0^T |i(t)|\,\mathrm{d}t. \tag{4.12}$$

The 1/2 in (4.12), is necessary since $|i(t)|$ is the measure of current flowing into and out of the clock source, where only one or the other should be counted.

Finally the average power dissipated for a single clock can be calculated from:

$$P_{ave\_clock} = i_{ave\_clock} V_{dd}.$$

$$P_{ave\_clock} = i_{ave\_clock} V_{dd}. \tag{4.13}$$

For the 1-ns cycle time of the dynamic ALUs, the period of interest occurs between 1.45-ns to 30.3-ns, as the ALU input stimulus and clock generator must initialize at the beginning of the transient analysis. It should be noted that the static CMOS ALU dissipates no clock power since it is a purely combinational circuit except for the input flip-flops which act as hard synchronization points. The average transistor power is added to the average clock power to obtain the total power dissipation of the dynamic ALUs while the power dissipation of the static CMOS ALU was found through static power analysis. The results are summarized in Table. 4.6, while Fig. 4.24 shows an example plot of the instantaneous transistor power dissipation in the CL-domino ALU over time.

Table 4.6: Power dissipation of different ALUs with no wiring parasitics except for output load.

ALU Design	Clock Skew (FO4)	Operating Frequency (MHz)	Transistor Power (mW)	Clock Power (mW)	Total Power (mW)	Total Energy (pJ)	Normal
CL-Domino	0	1-GHz	61.61	52.08	113.69	113.69	1.00
CL-Domino	1	1-GHz	67.14	47.44	114.58	114.58	1.00
Static CMOS 1	N/A	427-MHz	249.69	N/A	249.68	584.74	5.10
Static CMOS 2	N/A	400-MHz	174.08	N/A	174.08	435.19	3.80
Dual-Rail Domino	0	1-GHz	113.83	88.46	202.29	202.29	1.78
Dual-Rail Domino	1	1-GHz	111.55	82.11	193.65	193.65	1.69

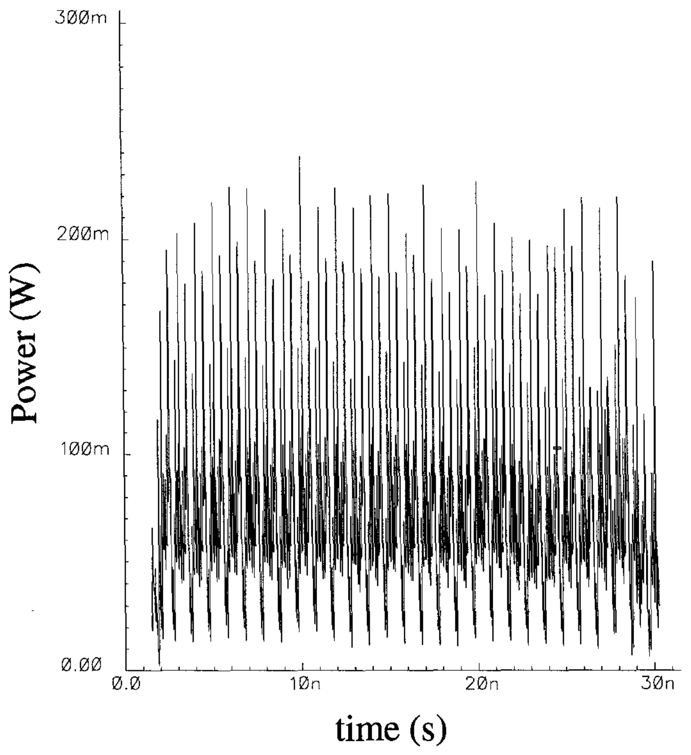

Figure 4.24: Clock-logic domino ALU transistor power dissipation at 1 GHz operation and no skew.

While the results for the clock power dissipation may seem high, they are typical for modern commercial microprocessors. For example, in the first and second generation Alpha microprocessors, the clocks represented 40% of the total power consumption [40]. In the third generation Alpha microprocessor (21264), the clock power represented 44% of the total power consumption even though this design used a conditioned clock hierarchy [4, 37]. In the latest IA-64 processor (Itanium 2), the clocks represent 1/3 of the total power [29]. In general clocks in local clock domains, can be conditioned or gated to save power [4, 52]. Gating a clock usually means disabling the clock feeding a module when that module is idle. Thus, the heavily loaded capacitance of the clocks in the module are not switched in this idle state, thus saving power. The overall power dissipation of the ALUs in this experiment are high due to the use of unfooted domino gates which were clocked with delayed clocks rather than the CL-domino method using OR-precharge for the footless gates as described in Section 3.5.3. This was done to simplify the comparisons between CL-domino and dual-rail skew-tolerant domino.

4.3.3 Discussion

Power Dissipation

As expected the power dissipation of the CL-domino ALU is much lower than that of the dual-rail skew-tolerant domino ALU at the same operating frequency as shown in Fig. 4.25. Without skew, the CL-domino ALU dissipates 56% of the power of the dual-rail domino ALU and at 1 FO4 skew, it dissipates 59% of the power. The discrepancy in the power dissipation for skewed and unskewed clocks can be attributed to slightly different transistor switching characteristics that result from the clocks arriving at different times.

The greater power dissipation of the dual-rail skew-tolerant domino ALU over the CL-domino ALU can first be attributed to the unity activity factor of dual-rail gates; as one of the true or complementary output node of each gate is cycled through precharge and evaluate operations every cycle. The activity factor of single-rail domino is dependent on the input signal probability instead. This is reflected in the transistor power dissipation for each design as in Table. 4.6. The second source of increased power dissipation in the dual-rail domino ALU is due to approximately double the number of clock transistors required for dual-rail gates. This is reflected in the clock power dissipation for each design as in Table. 4.6. The CL-domino ALU does not achieve a 50% reduction in clock power over the

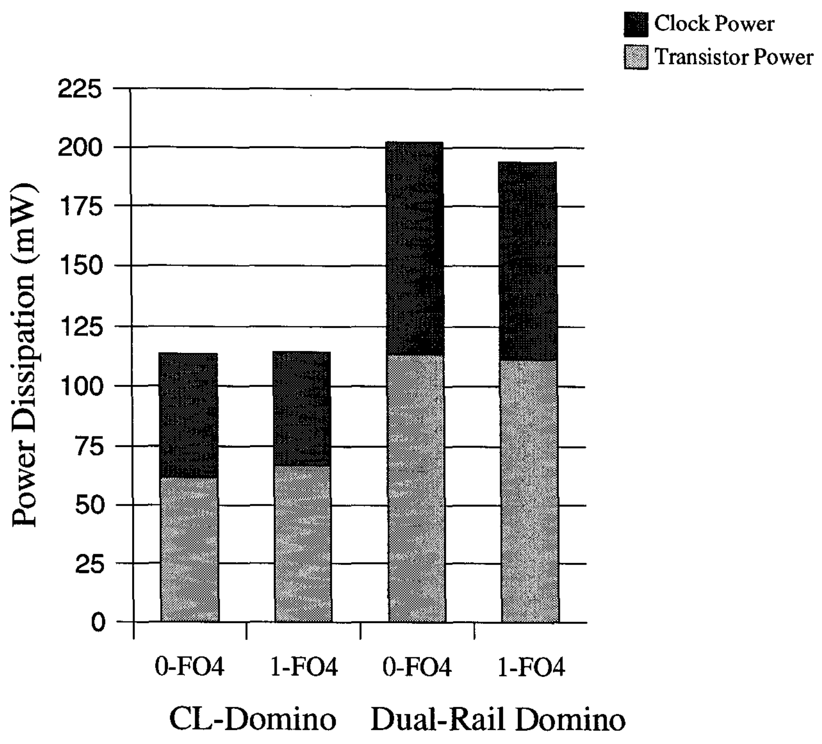

Figure 4.25: Power dissipation of ALUs at 1 GHz operation.

dual-rail domino ALU as extra clock transistors are required for OR-precharge and AND-evaluate. Furthermore, since the clock transistors are larger for the gates in CL-domino that implement OR-precharge and/or AND-evaluate, more capacitance is switched and the transistor power dissipation for CL-domino is higher than 50% of dual-rail skew-tolerant domino. Had wiring contributions been taken into account, the increased wire lengths and associated wiring capacitance of the dual-rail domino ALU would have resulted in even higher power dissipation than the CL-domino ALU.

The results for the synthesized static CMOS ALU are much higher than expected since static CMOS logic should dissipate less power than dynamic logic [8, 9]. Dynamic logic eliminates spurious transitions, or glitches, due to finite propagation delays, and switches less capacitance due to the elimination of the PMOS networks as compared to static CMOS. However, the switching probability of a dynamic gate is equal to signal probabilities that do not depend on the history of the inputs [61]. This signal probability, which will always be higher than the transition probability of static CMOS, makes dynamic logic switch more frequently, thus increasing power dissipation [60]. Furthermore, the clock node has a guaranteed transition each cycle and is a significant source of power dissipation. Although the total power dissipation of the second static CMOS design, "Static CMOS 2", lies between the results for CL-domino and dual-rail domino, we cannot safely conclude that CL-domino dissipates less power than static CMOS. This is because a static timing analysis tool was used to obtain the results for static CMOS while the dynamic ALUs used the results from a SPICE simulation. In all likelihood, the reason for the high power dissipation in the static CMOS designs is due to the course granularity of the drive strengths and associated transistor sizes in the standard cell library when comparing against the dynamic ALUs, which used custom transistor sizes.

Delay Performance

Schematic simulations of the CL-domino versus the dual-rail skew-tolerant domino ALU would most likely result in the dual-rail ALU obtaining similar latency and cycle times. This is because the dual-rail ALU can time borrow across the phase boundaries to input complemented and/or non-monotonic logic while the CL-Domino ALU must meet setup times and pay a skew penalty at these boundaries. However, the dual-rail ALU will experience much greater RC delays due to wiring. A study of dual-rail versus single-rail domino logic has shown that a 50% increase in transistors will result in approximately a 100% increase in layout area which adversely affects delay [32]. This increase in area means longer wire lengths and much greater difficulty in routing differential signals, as all true signals and all complement signals should be routed together to prevent coupling [43]. Thus we could tentatively conclude the CL-domino and dual-rail skew-tolerant domino would achieve almost equal delay performance. The CL-domino ALU was found to be fully functional at 1.12-GHz at 25°C. The performance would have been substantially higher if more care was taken to size the precharge transistors, since the internal nodes do not precharge fully during the precharge portion of the cycle when attempting to use the CL-domino ALU at higher frequencies. The static CMOS ALU, on the other hand, was only able to achieve 427-MHz performance with all the optimization constraints enabled.

4.4 Summary

Many high-speed datapaths require inverting or non-monotonic logic in small proportions to normal non-inverting monotonic logic. Traditionally, any dynamic logic datapath requiring these inversions have been built in a dual-rail fashion. In this chapter we investigated a representative ALU datapath of a modern 64-bit microprocessor and found that the single-rail techniques of CL-domino reduced power to 56% - 59% of a dual-rail skew-tolerant domino design while maintaining similar delay performance and smaller area.

Chapter 5

Future Work

5.1 Mask Layout and Extracted Capacitance Simulations

While schematic simulations, which neglect the effects of wiring, provide a good base comparison for different logic styles, a mask layout for each of the CL-domino and dual-rail skew-tolerant domino ALUs should be produced and simulations run using the extracted capacitance values. This would give designers more confidence that the CL-domino and dual-rail skew-tolerant domino ALU have equal delays and robustness while the CL-domino ALU dissipates much less power. A mask layout of the CL-domino ALU is included in the Appendix F.1.

5.2 CL-domino Interfaces

CL-domino methodologies should be developed for integrating CL-domino with static logic, as these interfaces almost always exist in real designs. The CL-domino/static interface should be investigated to develop techniques for simple transitions between CL-domino to static logic and from static logic to CL-domino. This involves developing timing types and clocking strategies. Furthermore pipeline placement of flip-flops, transparent latches and pulsed latches should be defined. Last, CL-domino should be investigated for gated clock buffers that implement clock stop (qualified clocks) during power saving modes and pipeline stalls.

In a real system, CL-domino circuits must also interface to special structures such as register files, cache memories and programmable logic arrays (PLAs). These interfaces should be adequately defined. Moreover, CL-domino may be used with other domino logic families such as traditional domino with transparent latches, dual-rail skew tolerant domino, CD-domino, CSG-domino, OTB domino, or OTB domino with Dynamic Latch Converters (DLCs) [29]. An understanding of how to incorporate CL-domino with these other dynamic circuit families, which are already used in commercial integrated circuits, would make CL-domino more appealing to the average design engineer.

5.3 Testability

As integrated circuits continue to pack more transistors and wiring layers onto a single silicon die, debug and functional production testing become more difficult. Design-For-Testability (DFT) techniques trade area, and in some cases performance, to make the testing task easier. The most often used testability technique is scan in which memory elements; such as flip-flops, latches, or the outputs of domino circuits; are made externally observable and controllable through a scan chain. This generally involves modifying the flip-flop, latch or dynamic gate to add scan signals and scan logic.

Conceptually simple DFT techniques should be developed for CL-domino which minimize extra cell area, extra wiring, performance impact and testing time. Furthermore, the scan should not introduce critical paths or require analysis and timing verification of the scan logic since it does not add any direct value to the customer. Because of the constraints for scan, simple "bolt-on" scan logic should be devised.

5.4 Leakage Energy

The use of wide dynamic gates is strongly impacted by reduced noise margins and increasing leakage currents in sub 130-nm technologies [3]. Another advantage to single-rail domino logic over dual-rail domino is the reduced leakage current due to fewer transistors and dynamic nodes needed to implement the logic. In a recent CSG-domino scheduler in 0.13 μm dual-V_T CMOS, the active leakage energy dissipation was 50% lower than a dual-rail domino design. As processes continue to scale to the $130 - nm$ node and beyond, leakage power dissipation will be an even greater concern and the benefit of CL-domino for leakage power dissipation should be quantified.

5.5 Clocking Issues

In our analysis, we derived timing constraints for the number of phases and the duty cycle of the CL-domino clocks. From our work so far, we did not recommend an optimal number of clock phases to use nor an optimal clock duty cycle. Although, these choices are design specific, guidelines should be devised to make this decision easier.

Chapter 6

Conclusion

As microprocessors integrate more and more transistors on a die, decreasing the energy consumption becomes as important as decreasing the cycle time. While skew-tolerant domino logic has been effective at providing the necessary performance in a high-speed system, providing it with logic completeness usually meant doubling the area and power dissipation. Clock-logic domino is a single-rail dynamic logic family that attempts to reduce the power consumption and area of domino gates through using logic functions of overlapping clocks so that dynamic gates do not receive the same clock for precharge and evaluate. Because of the reduced wire delays inherent in single-rail gates, Clock-logic domino can implement any boolean function while maintaining equal performance compared to the fastest dual-rail logic styles.

Chapter 2 explored the design of dynamic circuits and skew-tolerant domino that minimizes the effects of skew while affording time-borrowing to balance pipeline stages. Skew-tolerant domino provides a substantial performance improvement since it hides the overhead of traditional domino systems with latches. Chapter 3 provided a systematic method of incorporating inversions and non-monotonic logic into a domino pipeline. This method, called CL-domino, provided many of the advantages of skew-tolerant domino in regards to skew tolerance and time borrowing while eliminating short circuit currents. It was shown that selective partitioning of CL-domino logic allowed inverting gates to suffer a small skew penalty while providing high-speed and low-power performance. Chapter 4 compared a representative 64-bit microprocessor datapath in CL-domino and dual-rail skew-tolerant domino logic. It was shown that while both designs functioned at very high frequencies, that the CL-domino ALU dissipated 41% of the power of the dual-rail domino ALU. Finally, Chapter 5 explored several outstanding issues that would require solutions to make CL-domino a preferred industry alternative. In particular, the issues of interfacing with other logic families and methods to incorporate testability could be addressed.

In the near future, building faster microprocessors will mean meeting both power and cycle time budgets. Without a method to address the former, systems with billions of transistors operating at many gigahertz might never become a reality.

Bibliography

[1] S. Abdel-Hafeez and N. Ranjan. Single rail domino logic for four-phase clocking scheme. US Patent 6,265,899, July 24 2001. S3 Incorporated.

[2] A. Aipperspach, D. Allen, D. Cox, N. Phan, and S. Storino. A 0.2-μm, 1.8-V, SOI, 550-MHz, 64-b PowerPC microprocessor with copper interconnects. *IEEE J. of Solid-State Circuits*, 34(11):1430–35, November 1999.

[3] A. Alvandpour, R. K. Krishnamurthy, K. Soumyanath, and S. Y. Borkar. A sub-130-nm conditional keeper technique. *IEEE J. of Solid-State Circuits*, 37(5):633–38, May 2002.

[4] D. Bailey and B. Benschneider. Clocking design and analysis for a 600-MHz alpha microprocessor. *IEEE J. of Solid-State Circuits*, 33(11):1627–33, November 1998.

[5] R. Baker, H. Li, and D. Boyce. *CMOS - circuit design, layout, and simulation*, chapter Static logic gates, pages 231–52. IEEE Press, 1998.

[6] A. Bellaouar and M. I. Elmasry. *Low-Power Digital VLSI Design - Circuits and Systems*. Kluwer Academic Publishers, 1995.

[7] P. Bosshart and P. Landman. Inverting hold time latch circuits, systems, and methods. US Patent 6,242,952, June 5 2001. Texas Instruments Incorporated.

[8] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of high performance microprocessor circuits*. IEEE Press, 2001.

[9] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE J. of Solid-State Circuits*, 27(4):473–84, April 1992.

[10] H.-H. Chang, J.-W. Lin, C.-Y. Yang, and S.-I. Liu. A wide-range delay-locked loop with a fixed latency of one clock cycle. *IEEE J. of Solid-State Circuits*, 37(8):1021–27, August 2002.

[11] C.T. Chuang, P. F. Lu, and C. J. Anderson. SOI for digital CMOS VLSI: design considerations and advances. *Proc. of the IEEE*, 86(4):689–720, April 1998.

[12] M. Ciraula, G. Lattimore, R. Masleid, and D. Mikan Jr. Creating inversions in ripple domino logic. US Patent 5,892,372, April 6 1999. International Business Machines Corporation.

[13] D. Clein. *CMOS IC layout: concepts, methodologies, and tools*. Newnes, Woburn, MA, 2000.

[14] D. Deleganes, J. Douglas, B. Kommandur, and M. Patyra. Designing a 3GHz, 130nm, Intel Pentium 4 processor. In *Proc. IEEE Int. Symposium on VLSI Circuits*, pages 130–33, June 2002.

[15] A. Jain et al. A 1.2-GHz Alpha microprocessor with 44.8 GB/s chip pin bandwidth. In *Proc. IEEE Int. Solid-State Circuits Conf.*, pages 240–41, February 2001.

[16] A. Kowalczyk et al. the first MAJC microprocessor: a dual CPU system-on-chip. *IEEE J. of Solid-State Circuits*, 36(11):1609–16, November 2001.

[17] B. W. Curran et al. IBM eServer z900 high-frequency microprocessor technology, circuits and design methodology. *IBM J. of Res. Develop.*, 46(4/5):631–44, July/September 2002.

[18] C. Akrout et al. A 480-MHz RISC microprocessor in a 0.12-μm l_{eff} CMOS technology with copper interconnects. *IEEE J. of Solid-State Circuits*, 33(11):1609–16, November 1998.

[19] D. Allen et al. Custom circuit design as a driver of microprocessor performance. *IBM J. of Res. Develop.*, 44(6):799–822, November 2000.

[20] D. Harris et al. Opportunistic time-borrowing domino logic. US Patent 5,517,136, March 3 1996.

[21] D. W. Dobberpuhl et al. A 200-MHz 64-b dual-issue CMOS microprocessor. *IEEE J. of Solid-State Circuits*, 27(11):1555–67, November 1992.

[22] E. S. Fetzer et al. A fully bypassed six-issue integer datapath and register file on the Itanium 2 microprocessor. *IEEE J. of Solid-State Circuits*, 37(11):1433–40, November 2002.

[23] G. Hinton et al. A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit. *IEEE J. of Solid-State Circuits*, 36(11):1617–27, November 2001.

[24] G. K. Konstadinidis et al. Implementation of a third-generation 1.1-GHz 64-bit microprocessor. *IEEE J. of Solid-State Circuits*, 37(11):1461–69, November 2002.

[25] J. D. Warnock et al. the circuit and physical design of the POWER4 microprocessor. *IBM J. of Res. Develop.*, 46(1):27–51, January 2002.

[26] J. Silberman et al. A 1-GHz single-issue 64-bit PowerPC integer processor. *IEEE J. of Solid-State Circuits*, 33(11):1600–08, November 1998.

[27] M. S. Hrishikesh et al. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proc. 29th Annual Int. Symposium on Computer Architecture*, pages 14–24, May 2002.

[28] R. Heald et al. A third-generation SPARC V9 64-b microprocessor. *IEEE J. of Solid-State Circuits*, 35(11):1526–38, November 2000.

[29] S. D. Naffizger et al. The implementation of the Itanium 2 microprocessor. *IEEE J. of Solid-State Circuits*, 37(11):1488–60, November 2002.

[30] S. Matthew et al. Sub-500-ps 64-b ALUs in 0.18-μm SOI/Bulk CMOS: design and scaling trends. *IEEE J. of Solid-State Circuits*, 36(11):1636–46, November 2001.

[31] S. Tam et al. Clock generation and distribution for the first IA-64 microprocessor. *IEEE J. of Solid-State Circuits*, 35(11):1545–52, November 2000.

[32] S. Vangal et al. 5-GHz 32-bit integer execution core in 130-nm dual-V_T CMOS. *IEEE J. of Solid-State Circuits*, 37(11):1421–32, November 2002.

[33] T. Buchholtz et al. A 0.18-μm, 1.5-V, SOI, 660-MHz, 64-bit PowerPC microprocessor with copper interconnects. In *Proc. IEEE Int. Solid-State Circuits Conf.*, pages 88–99, February 2000.

[34] Y.-J. Jung et al. A dual-loop delay-locked loop using multiple voltage-controlled delay lines. *IEEE J. of Solid-State Circuits*, 36(5):784–91, May 2001.

[35] H. Fair and D. Bailey. Clocking design and analysis for a 600 MHz Alpha microprocessor. In *Proc. IEEE Int. Solid-State Circuits Conf.*, pages 398–99, 473, February 1998.

[36] D. Foley and M. Flynn. CMOS DLL-based 2-V 3.2-ps jitter 1-GHz clock synthesizer and temperature-compensated tunable oscillator. *IEEE J. of Solid-State Circuits*, 36(3):417–23, March 2001.

[37] E. G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–92, May 2001.

[38] V. Friedman and S. Liu. Dynamic logic CMOS circuits. *IEEE J. of Solid-State Circuits*, 19(2):263–66, April 1984.

[39] P. P. Gelsinger. Microprocessors for the new millennium: challenges, opportunities, and new frontiers. In *Proc. IEEE Int. Solid-State Circuits Conf.*, pages 22–25, February 2001.

[40] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon. High-performance microprocessor design. *IEEE J. of Solid-State Circuits*, 33(5):1488–60, May 1998.

[41] T. Han and D. A. Carlson. Fast area-efficient VLSI adders. In *8th Symp. Computer Arithmetic*, pages 49–56, September 1987.

[42] D. Harris. Skew-tolerant domino circuits. *IEEE J. of Solid-State Circuits*, 32(11):1702–11, November 1997.

[43] D. Harris. *Skew-tolerant circuit design.* Morgan Kaufmann Publishers, 2001.

[44] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers, 2002.

[45] T. W. Houston, P. W. Bosshart, and C. Shaw. Compound domino CMOS circuit. US Patent 5,015,882, May 14 1990. Texas Instruments Incorporated.

[46] G. Jung, V. Perepelitsa, and G. E. Sobelman. Time borrowing in high-speed funtional units using skew-tolerant domino circuits. In *Proc. IEEE Int. Symp. on Circuits and Systems*, volume 6, pages 41–44, May 2000.

[47] C. Kim, I.-C. Hwang, and S.-M. Kang. A low-power small-area ± 7.28-ps-jitter 1-GHz DLL-based clock generator. *IEEE J. of Solid-State Circuits*, 37(11):1414–20, November 2002.

[48] S. Kim and G. E. Sobelman. Efficient digit-serial FIR filters with skew-tolerant domino. In *Proc. IEEE Int. Symp. on Circuits and Systems*, volume 6, pages 369–72, May 2002.

[49] U. Ko, P. T. Balsara, and W. Lee. Low-power design technique for high-performance CMOS adders. *IEEE Trans. on VLSI Systems*, 3(2):327–33, June 1995.

[50] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):783–91, August 1973.

[51] K. Krewell. Fujitsu's SPARC64 V is real deal. *Microprocessor report*, October 2002.

[52] N. A. Kurd, J. S. Barkatullah, R. O. Dizon, T. D. Fletcher, and P.D. Madland. A multigigahertz clocking scheme for the Pentium 4 microprocessor. *IEEE J. of Solid-State Circuits*, 36(11):1647–53, November 2001.

[53] H. L. Levy and S. A. Shah. Method and apparatus for fast evaluation of dynamic cmos logic circuits. US Patent 5,825,208, October 20 1998.

[54] W. Dally M.-J. E. Lee and P. Chiang. A 90 mW 4 Gb/s equalized I/O circuit with input offset cancellation. In *Proc. IEEE Int. Solid-State Circuits Conf.*, pages 252–253,463, February 2000.

[55] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley Pub Co, 1980.

[56] Y. Moon, J. Choi, K. Lee, D.-K. Jeong, and M.-K. Kim. An all-analog multiphase delay-locked loop using a replica delay line for wide-range operation and low-jitter performance. *IEEE J. of Solid-State Circuits*, 35(3):377–84, March 2000.

[57] G. Moore. Progress in digital integrated electronics. *IEDM*, 1975.

[58] P. Ng, P. T. Balsara, and D. Steiss. Performance of CMOS differential circuits. *IEEE J. of Solid-State Circuits*, 31(6):841–46, June 1996.

[59] B. Parhami. *Computer arithmetic : algorithms and hardware designs.* Oxford University Press, 2000.

[60] J. M. Rabaey. *Digital integrated circuits - a design perspective.* Prentice Hall, 1st edition, 1996.

[61] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital integrated circuits - a design perspective.* Prentice Hall, 2nd edition, 2002. Preprint.

[62] S. Rusu and G. Singer. The first IA-64 microprocessor. *IEEE J. of Solid-State Circuits*, 35(11):1539–44, November 2000.

[63] R. Senthinathan, S. Fischer, H. Rangchi, and H. Yazdanmehr. A 650-MHz, IA-32 microprocessor with enhanced data streaming for graphics and video. *IEEE J. of Solid-State Circuits*, 34(11):1454–65, November 1999.

[64] D. Somasekhar and K. Roy. Differential current switch logic: a low power DCVS logic family. *IEEE J. of Solid-State Circuits*, 31(7):981–91, July 1996.

[65] D. Somasekhar and K. Roy. LVDCSL: a high fan-in, high-performance, low-voltage differential current switch logic family. *IEEE Trans. on VLSI Systems*, 6(4):573–77, December 1998.

[66] D. Stasiak, R. Mounes-Toussi, and S. Storino. A 440-ps 64-bit adder in 1.5-V/0.18-μm partially depleted SOI technology. *IEEE J. of Solid-State Circuits*, 36(10):1546–52, October 2001.

[67] B. Sugla and D. A. Carlson. Extreme area-time tradeoffs in VLSI. *IEEE Transactions on Computers*, 39(2):251–57, February 1990.

[68] I. Sutherland, B. Sproull, and D. Harris. *Logical effort.* Morgan Kaufmann Publishers, Inc., 1st edition, 1999.

[69] L. C. Tsai. A 1 GHz PA-RISC processor. In *Proc. IEEE Int. Solid-State Circuits Conf.*, pages 322–23, February 2001.

[70] T. Williams and M. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE J. of Solid-State Circuits*, 26(11):1651–61, November 1991.

[71] G. Yee. *Dynamic logic design and synthesis using clock-delayed domino.* Department of electrical engineering, University of Washington, 1999.

[72] G. Yee and C. Sechen. Clock-delayed domino for dynamic circuit design. *IEEE Trans. on VLSI Systems*, 8(4):425–30, August 2000.

[73] R. Zimmermann and W. Fichtner. Low-power logic styles: CMOS versus pass-transistor logic. *IEEE J. of Solid-State Circuits*, 32(7):1079–90, July 1997.

Appendix A

Process Characterization

A.1 Process Parameters

The process technology characteristics for Taiwan Semiconductor Manufacturing Company Ltd.'s (TSMC) 0.18 μm general purpose logic process are listed in Table. A.1. All the implemented ALUs were designed in this process technology.

Table A.1: Process technology characteristics.

Parameter	Value
Wafer Type	bulk CMOS
Wells	single N-type
L(drawn)	0.18-μm
L(effective-NMOS)	0.135-μm
L(effective-PMOS)	0.160-μm
T_{ox}	32 angstroms
Polysilicon Layers	single layer
Poly 1 Pitch	0.46-μm
Local Interconnect	None
Wiring Layers	6 Aluminum
Metal 1 Pitch	0.46-μm
Metal 2-5 Pitch	0.56-μm
Metal 6 Pitch	0.90-μm
Nominal Supply Voltage	1.8-V

A.2 Inverter and FO4 Inverter Delays

Delay measurements are usually performed using a ring oscillator [Rabaey96], which consists of an odd number of inverters coupled together in a circular chain as in Fig. A.1. Due to the odd number of inversions, the circuit does not have a stable operating point and oscillates between a logic 0 and a logic 1. The equation that governs the oscillation frequency of a ring oscillator is [Baker98]:

$$f_{osc} = \frac{1}{n \cdot (t_{PHL} + t_{PLH})}. \tag{A.1}$$

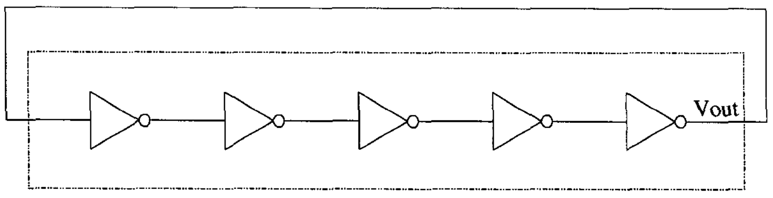

Figure A.1: Ring oscillator for inverter delay measurement.

Assuming that $t_{PHL} = t_{PLH}$, which is a good approximation when the PMOS transistors are two times the width of the NMOS transistors for equal rise and fall times, the frequency of oscillation is equal to:

$$f_{osc} = \frac{1}{2 \cdot n \cdot t_P}. \tag{A.2}$$

Re-arranging (A.2), the delay time through an inverter is equal to:

$$t_{P-Inverter} = \frac{1}{2 \cdot n \cdot f_{osc}}. \tag{A.3}$$

When measuring delays, it is often beneficial to use a process-independent unit of delay so that intuition about delay can be carried over from one process to another [Harris01]. A unit of delay common to custom designs is the Fanout-of-4 (FO4) inverter delay, measured as the the delay of an inverter driving four identical copies of itself. The FO4 delay is useful because it is easy to determine and because the theory of logical effort [Sutherland99] predicts that cascaded logic drive a load fastest when each gate has a fanout of about 4.

The FO4 inverter delay is also measured using a ring oscillator configuration as shown in Fig. A.2. However, the output of each inverter, (a), is loaded with three additional copies of itself, (b), to give the required fanout of four. Within a stage in the ring oscillator, the load inverters, (b), must also have their outputs coupled to the inputs of four other inverters, (c). This is because removing (c) would make (b) switch very rapidly. Because of the Miller effect, this would increase the effective input capacitance of (b) and hence affect the output load on (a). This would result in higher delays in the ring oscillator [Sutherland99] than is the case of inverters in an integrated circuit. The FO4 inverter delay is determined from the following equation:

$$t_{P-FO4} = \frac{1}{2 \cdot n \cdot f_{osc}}. \tag{A.4}$$

The results for rings of minimum sized inverters with PMOS transistors $W/L = 0.84\mu m/0.18\mu m$ and NMOS transistors $W/L = 0.42\mu m/0.18\mu m$ for different environmental conditions and process corners are summarized in Table. A.2. Since a good estimate of the FO4 inverter delay in a short-channel sub-micron process is approximately $400 \cdot$ feature size-ps [Harris01], our simulation results closely match the predicted value, which is equal to 72-ps in a $0.18\ \mu m$ process.

Table A.2: Single inverter and FO4 inverter delay times for different environmental conditions and process corners.

Component	**TT, 1.8V, 25°C (ps)**	**TT, 1.8V, 105°C (ps)**	**FF, 2.0V, 0°C (ps)**	**SS, 1.6V, 105°C (ps)**
Inverter	29.46	32.49	25.91	36.73
FO4	71.56	78.54	63.4	88.18

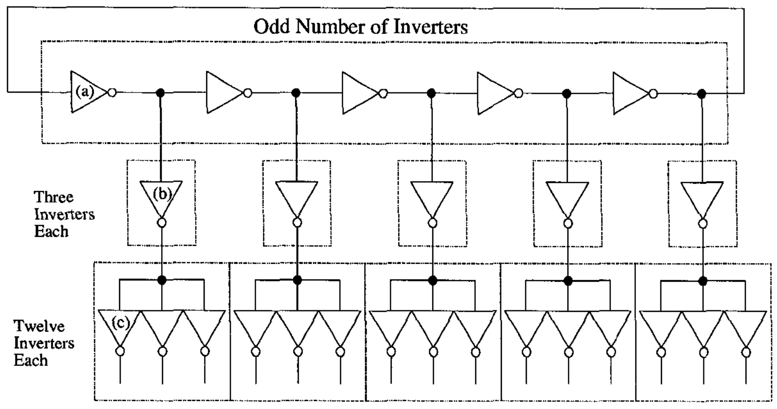

Figure A.2: Ring oscillator for FO4 delay measurement.

A.3 Wire and Transistor Parasitics

In our 0.18 μm technology, a 45 μm long M1, minimum width (0.23 μm) wire running over substrate with no coupling to adjacent signal wires is equivalent in load capacitance to a minimum sized inverter. The load capacitance value of the minimum sized inverter is approximately 2.94 fF. This was measured in simulations through matching the delays when driving an inverter without interconnect parasitics to driving a wire with interconnect parasitics. While this method abstracts the non-linear capacitance contribution of the inverter and does not take into account wiring resistance, it is a reasonable "back-of-envelope" approximation.

Appendix B

Single-Rail Domino Design Alternative

B.1 CSG-Domino

Other attempts at single-rail domino designs for input complemented and non-monotonic logic have focused on using complementary signal generators (CSGs) to generate dual-rail signals [Mathew01,Vangal02]. These techniques are usually protected by patent law [Kanetani01,Krishnamurthy01]. A schematic of a single-rail input to domino-compatible dual-rail output generator is shown in Fig. B.1.

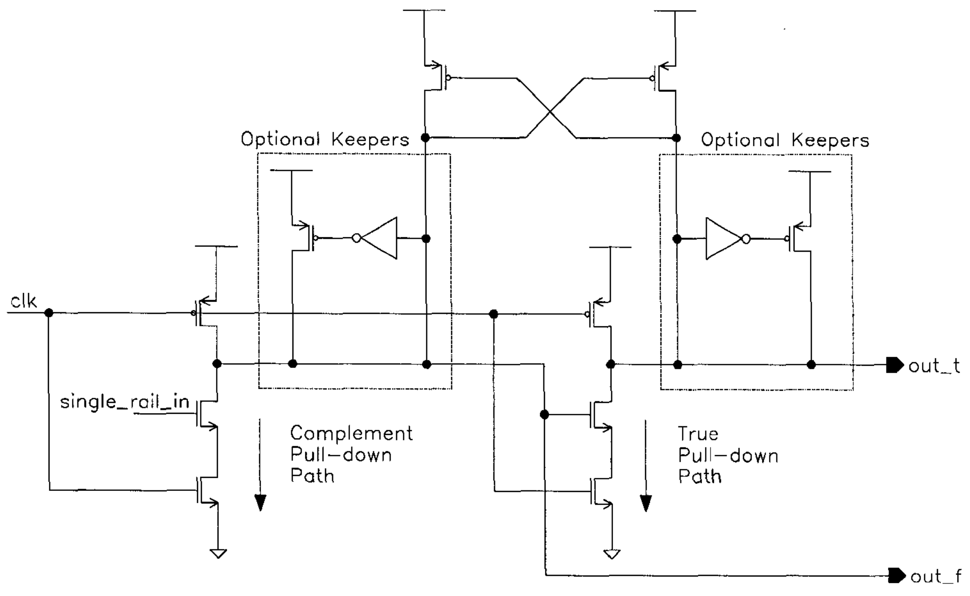

Figure B.1: CSG generator.

The CSG circuit contains two dynamic nodes, a true dynamic node and a complementary dynamic node, where both nodes are cycled from precharge to evaluate using the same clock. The complementary pull-down path is gated with the single-rail input signal while the true pull-down path is gated by the complementary dynamic node. The circuit has a race condition where the complementary pull-down path will discharge its dynamic node faster than the true pull-down path can discharge its dynamic node if the input is high, while only the true pull-down path will discharge its dynamic node if the input is low. The cross-coupled PMOS transistors provide weak feedback for noise immunity and statically holds the true node when the complementary node discharges. This is especially important since the true dynamic node will experience a noise droop due to the finite discharge time of the complementary dynamic node. Furthermore, the circuit will function only when the complementary pull-down path has a higher transconductance than the true pull-down path. This can be accommodated if the transistors of the complementary path are wider than the true path or if the transistors of the complementary path have longer length. Both of these solutions to transconductance decrease the switching speed of the CSG circuit since wider transistors present more input loading while longer transistors decrease transistor current. The half-keepers on the dynamic nodes are used to further increase noise immunity.

Like CL-domino, a CSG circuit and the input complemented or non-monotonic logic that follows it, must be placed at a clock phase boundary [Mathew01]. Data must setup at the inputs of the CSG before the rising edge of the clock and therefore the evaluation imposes a hard edge on the data, much like the setup time required for flip-flops. Therefore, when clock skew is accounted for, the previous phase of logic has a logic evaluation constraint equal to that previously derived for CL-domino (3.1). Therefore, time cannot be borrowed from an input complemented or non-monotonic logic function in CSG-domino pipelines much like the non-time borrowing penalty in CL-domino pipelines. Modified CSGs have been shown to fold monotonic functions at the inputs and non-monotonic functions at the outputs for increased speed as shown in Fig. B.2 [Vangal02]. When the CSG technique was applied to a scheduler for an integer execution unit, the layout area and loopback interconnect length was reduced by 67% and 25%, respectively, over a dual-rail design. For this particular design in the literature, the number of gate stages required for a single-rail scheduler was reduced over a dual-rail scheduler by two. This reduction in area and in gate stages resulted in a 23% delay improvement over a comparable dual-rail domino design.

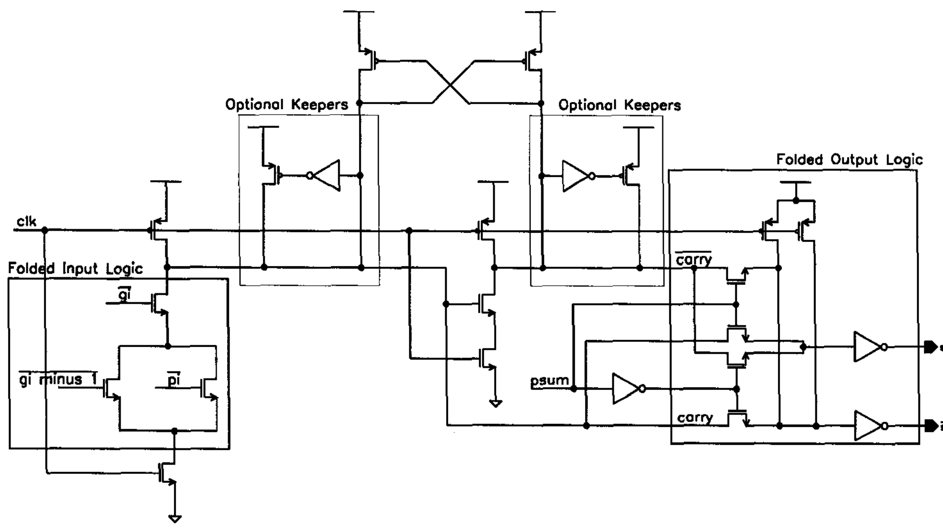

Figure B.2: CSG generator with folded input and output logic.

B.2 CSG-Domino Versus CL-Domino

The CSG generators have a unity activity factor since either the true or complement node is discharged and then must be precharged every cycle. This is unlike any circuit in CL-domino, which only have activity factors equal to the input signal probabilities. Because the CSG circuits have a small race condition, the cross-coupled keeper has to replenish charge lost on the true node when the complementary node discharges, thus increasing power dissipation. Moreover, dual-rail domino gates are used to implement input complemented or non-monotonic logic functions in CSG-domino. This is in contrast to CL-domino which uses single-rail gates throughout, thus reducing circuit area and power dissipation. Clock power dissipation is also higher since the CSG circuit, with precharge and evaluate transistors, must be clocked. There are more clock transistors in the CSG-domino than in CL-domino even though a generalized CL-domino pipeline utilizes some series PMOS and/or series NMOS clock transistors.

The CSG circuit might be considered a dynamic gate with a stack height of two for the most optimal implementation where logic is folded into the circuit [Vangal02]. A CSG-domino design of the ALU used for our comparison studies (Section 4.2), would have added two gate delays to the critical path compared to the CL-domino ALU of Section 4.2.1. However, the CSG generators removes two sets of series CL-domino clock transistors in the critical path, thus closing the performance gap between CSG-domino and CL-domino. One must also consider that the CSG must be ratioed so that the complementary pull-down path is faster than the true pull-down path, thus increasing the gate load of the CSG and retarding performance.

Appendix C

Verilog Code

C.1 Functional Verilog Code for ALU (Synthesizable)

The flip-flops found in the ALU verilog code were required for synthesis and can be removed from the functional model.

```
// Verilog netlist of
// "alu"

// HDL models

// HDL file - CSG_alu, thru, behavioral.

// Verilog HDL for "CSG_alu", "thru" "functional"

module thru(in,out);

input in;
output out;

assign out = in;

endmodule

// HDL file - CSG_alu, back_to_back_inv, behavioral.

// Verilog HDL for "CSG_alu", "back_to_back_inv" "behavioral"

module back_to_back_inv(in,out);

parameter MIN_DELAYS = 26;
parameter TYP_DELAYS = 29;
parameter MAX_DELAYS = 37;

input in;
output out;

wire inv_out;

not #(MIN_DELAYS:TYP_DELAYS:MAX_DELAYS) i0(inv_out,in);
not #(MIN_DELAYS:TYP_DELAYS:MAX_DELAYS) i1(out,inv_out);

endmodule

// HDL file - CSG_alu_syn, mux91_a, functional.

// Verilog HDL for "CSG_alu_syn", "mux91_a" "functional"

module mux91_a (out, clka, a0, a1, a2, a3, a4, a5, a6, a7, a8, sel);

parameter NWORDS = 9;
parameter NBITS = 64;

output [NBITS-1:0] out;
reg [NBITS-1:0] out;
    input clka;
    input [NBITS-1:0] a0, a1, a2, a3, a4, a5, a6, a7, a8;
    input [NWORDS-1:0] sel;

always@(sel or a0 or a1 or a2 or a3 or a4 or a5 or a6 or a7 or a8)

case(sel)
```

```
9'b000000001: out = a0;
9'b000000010: out = a1;
9'b000000100: out = a2;
9'b000001000: out = a3;
9'b000010000: out = a4;
9'b000100000: out = a5;
9'b001000000: out = a6;
9'b010000000: out = a7;
9'b100000000: out = a8;
    default: out = a0;
endcase

endmodule

// HDL file - CSG_alu_syn, mux91_b, functional.

// Verilog HDL for "CSG_alu_syn", "mux91_b" "functional"

module mux91_b (out, clka, clkb, b0, b1, b2, b3, b4, b5, b6, b7, b8, sel);

parameter NWORDS = 9;
parameter NBITS = 64;

output [NBITS-1:0] out;
reg [NBITS-1:0] out;
    input clka, clkb;
    input [NBITS-1:0] b0, b1, b2, b3, b4, b5, b6, b7, b8;
    input [NWORDS-1:0] sel;

always@(sel or b0 or b1 or b2 or b3 or b4 or b5 or b6 or b7 or b8)

case(sel)
9'b000000001: out = b0;
9'b000000010: out = b1;
9'b000000100: out = b2;
9'b000001000: out = b3;
9'b000010000: out = b4;
9'b000100000: out = b5;
9'b001000000: out = b6;
9'b010000000: out = b7;
9'b100000000: out = b8;
    default: out = b0;
endcase

endmodule

// HDL file - CSG_alu_syn, mux51, functional.

// Verilog HDL for "CSG_alu_syn", "mux51" "functional"

module mux51 (out, clka, d0, d1, d2, d3, d4, sel);

parameter NWORDS = 5;
parameter NBITS = 64;

output [NBITS-1:0] out;
reg [NBITS-1:0] out;
    input clka;
    input [NBITS-1:0] d0, d1, d2, d3, d4;
    input [NWORDS-1:0] sel;

always@(sel or d0 or d1 or d2 or d3 or d4)

case(sel)
5'b00001: out = d0;
5'b00010: out = d1;
5'b00100: out = d2;
5'b01000: out = d3;
5'b10000: out = d4;
    default: out = d0;
endcase

endmodule

// HDL file - CSG_alu_syn, mux21, functional.

// Verilog HDL for "CSG_alu_syn", "mux21" "functional"

module mux21 (out, clka, d, sel);

parameter NWORDS = 2;
parameter NBITS = 64;

output [NBITS-1:0] out;
reg [NBITS-1:0] out;
    input clka;
    input [NBITS-1:0] d;
    input [NWORDS-1:0] sel;
```

```
always@(sel or d)

case(sel)
2'b01: out = d;
2'b10: out = ~d;
    default: out = d;
endcase

endmodule

// HDL file - CSG_alu_syn, flip_flop, functional.

// Verilog HDL for "CSG_alu_syn", "flip_flop" "functional"

module flip_flop (clka, d, q);

parameter NBITS = 64;

output [NBITS-1:0] q;
reg [NBITS-1:0] q;
    input clka;
    input [NBITS-1:0] d;

always@(posedge clka)
q <= d;

endmodule

// HDL file - CSG_alu, pbar_gbar_gen_64, functional.

// Verilog HDL for "CSG_alu", "pbar_gbar_gen_64" "functional"

module pbar_gbar_gen_64(clkb,clkc,clkd,clke,a,b,pbar,gbar,psum) ;

parameter NUMBITS = 64;

input clkb,clkc,clkd,clke;
input [NUMBITS-1:0] a,b;
output [NUMBITS-1:0] pbar,gbar;
output [NUMBITS-1:0] psum;

assign pbar = ~(a | b);
assign gbar = ~(a & b);
assign psum = a ^ b;

endmodule

// HDL file - CSG_alu, carry_merge_64, functional.

// Verilog HDL for "CSG_alu", "carry_merge_64" "functional"

module carry_merge_64(pbar,gbar,clk,p,g) ;

parameter NUMBITS = 64;

input [NUMBITS-1:0] pbar,gbar;
input [3:5] clk;
output [NUMBITS-1:0] p,g;

integer i;
integer j;

reg [NUMBITS-1:0] gcm0, pcm0;
reg [NUMBITS-1:0] gcm1, pcm1;
reg [NUMBITS-1:0] gcm2, pcm2;
reg [NUMBITS-1:0] gcm3, pcm3;
reg [NUMBITS-1:0] gcm4, pcm4;
reg [NUMBITS-1:0] gcm5, pcm5;

always@(pbar or gbar)
begin
for(i = 0; i < NUMBITS; i = i + 1)
begin
if (i % 2 == 0)
begin

//gcm0[i] = ~gbar[i];
//pcm0[i] = ~pbar[i];

//gcm1[i] = ~gcm0[i];
//pcm1[i] = ~pcm0[i];

//gcm2[i] = ~gcm1[i];
//pcm2[i] = ~pcm1[i];

//gcm3[i] = ~gcm2[i];
//pcm3[i] = ~pcm2[i];

//gcm4[i] = ~gcm3[i];
```

```
//pcm4[i] = ~pcm3[i];

//gcm5[i] = ~gcm4[i];
//pcm5[i] = ~pcm4[i];


pcm5[i] = pbar[i];
gcm5[i] = gbar[i];
end
else
begin
j = 1;

gcm0[i] = ~((pbar[i] | gbar[i-j]) & gbar[i]);
pcm0[i] = ~(pbar[i] | pbar[i-j]);

j = 2;

if (i - j < 0)
begin
gcm1[i] = ~gcm0[i];
pcm1[i] = ~pcm0[i];
end
else
begin
gcm1[i] = ~((pcm0[i] & gcm0[i-j]) | gcm0[i]);
pcm1[i] = ~(pcm0[i] & pcm0[i-j]);
end

j = 4;

if (i - j < 0)
begin
gcm2[i] = ~gcm1[i];
pcm2[i] = ~pcm1[i];
end
else
begin
gcm2[i] = ~((pcm1[i] | gcm1[i-j]) & gcm1[i]);
pcm2[i] = ~(pcm1[i] | pcm1[i-j]);
end

j = 8;

if (i - j < 0)
begin
gcm3[i] = ~gcm2[i];
pcm3[i] = ~pcm2[i];
end
else
begin
gcm3[i] = ~((pcm2[i] & gcm2[i-j]) | gcm2[i]);
pcm3[i] = ~(pcm2[i] & pcm2[i-j]);
end

j = 16;

if (i - j < 0)
begin
gcm4[i] = ~gcm3[i];
pcm4[i] = ~pcm3[i];
end
else
begin
gcm4[i] = ~((pcm3[i] | gcm3[i-j]) & gcm3[i]);
pcm4[i] = ~(pcm3[i] | pcm3[i-j]);
end

j = 32;

if (i - j < 0)
begin
gcm5[i] = ~gcm4[i];
pcm5[i] = ~pcm4[i];
end
else
begin
gcm5[i] = ~((pcm4[i] & gcm4[i-j]) | gcm4[i]);
pcm5[i] = ~(pcm4[i] & pcm4[i-j]);
end
end
end
end

assign g = gcm5;
assign p = pcm5;

endmodule
```

```
// HDL file - CSG_alu, even_carry_gen_64, functional.

// Verilog HDL for "CSG_alu", "even_carry_gen_64" "functional"

module even_carry_gen_64(pbar,gbar,cout) ;

parameter NUMBITS = 64;

input [NUMBITS-1:0] pbar,gbar;
output [NUMBITS-1:0] cout;

reg [NUMBITS-1:0] cout;

integer i;

always@(pbar or gbar)
begin
    cout[0] = ~(gbar[0]);
for(i = 1; i < NUMBITS; i = i + 1)
begin
if ((i % 2) == 0)
cout[i] = ~((pbar[i] | gbar[i-1]) & gbar[i]);
else
cout[i] = ~gbar[i];
end
end
endmodule

// HDL file - CSG_alu, sum_xor_64, functional.

// Verilog HDL for "CSG_alu", "sum_xor_64" "functional"

module sum_xor_64(cout,psum,clka,sum) ;

parameter NUMBITS = 64;

input [NUMBITS-1:0] psum, cout;
input clka;
output [NUMBITS-1:0] sum;

assign sum = psum ^ cout << 1;

endmodule

// HDL file - CSG_alu_syn, bus_driver_bit, functional.

// Verilog HDL for "CSG_alu", "bus_driver_bit" "functional"

module bus_driver_bit (out, in);
    output out;
    input in;

assign #1 out = in;

endmodule

// HDL file - CSG_alu_syn, mux31, functional.

// Verilog HDL for "CSG_alu_syn", "mux31" "functional"

module mux31 (out, clka, d0, d1, d2, sel);

parameter NWORDS = 3;
parameter NBITS = 64;

output [NBITS-1:0] out;
reg [NBITS-1:0] out;
    input clka;
    input [NBITS-1:0] d0, d1, d2;
    input [NWORDS-1:0] sel;

always@(sel or d0 or d1 or d2)

case(sel)
3'b001: out = d0;
3'b010: out = d1;
3'b100: out = d2;
    default: out = d0;
endcase

endmodule
// End HDL models

// Library - CSG_alu_syn, Cell - bus_driver, View - schematic
// LAST TIME SAVED: Nov 28 18:21:26 2002
// NETLIST TIME: Nov 28 22:57:43 2002
`timescale 1ps / 1ps

module bus_driver ( out, {in[63], in[62], in[61], in[60], in[59],
     in[58], in[57], in[56], in[55], in[54], in[53], in[52], in[51],
     in[50], in[49], in[48], in[47], in[46], in[45], in[44], in[43],
     in[42], in[41], in[40], in[39], in[38], in[37], in[36], in[35],
     in[34], in[33], in[32], in[31], in[30], in[29], in[28], in[27],
     in[26], in[25], in[24], in[23], in[22], in[21], in[20], in[19],
     in[18], in[17], in[16], in[15], in[14], in[13], in[12], in[11],
     in[10], in[9], in[8], in[7], in[6], in[5], in[4], in[3], in[2],
     in[1], in[0]} );

output [63:0]  out;

input [0:63]  in;
supply0 VSS_;

// List of primary aliased buses



bus_driver_bit I0_0_ ( out[0], in[0]);
bus_driver_bit I0_1_ ( out[1], in[1]);
bus_driver_bit I0_2_ ( out[2], in[2]);
bus_driver_bit I0_3_ ( out[3], in[3]);
bus_driver_bit I0_4_ ( out[4], in[4]);
bus_driver_bit I0_5_ ( out[5], in[5]);
bus_driver_bit I0_6_ ( out[6], in[6]);
bus_driver_bit I0_7_ ( out[7], in[7]);
bus_driver_bit I0_8_ ( out[8], in[8]);
bus_driver_bit I0_9_ ( out[9], in[9]);
bus_driver_bit I0_10_ ( out[10], in[10]);
bus_driver_bit I0_11_ ( out[11], in[11]);
bus_driver_bit I0_12_ ( out[12], in[12]);
bus_driver_bit I0_13_ ( out[13], in[13]);
bus_driver_bit I0_14_ ( out[14], in[14]);
bus_driver_bit I0_15_ ( out[15], in[15]);
bus_driver_bit I0_16_ ( out[16], in[16]);
bus_driver_bit I0_17_ ( out[17], in[17]);
bus_driver_bit I0_18_ ( out[18], in[18]);
bus_driver_bit I0_19_ ( out[19], in[19]);
bus_driver_bit I0_20_ ( out[20], in[20]);
bus_driver_bit I0_21_ ( out[21], in[21]);
bus_driver_bit I0_22_ ( out[22], in[22]);
bus_driver_bit I0_23_ ( out[23], in[23]);
bus_driver_bit I0_24_ ( out[24], in[24]);
bus_driver_bit I0_25_ ( out[25], in[25]);
bus_driver_bit I0_26_ ( out[26], in[26]);
bus_driver_bit I0_27_ ( out[27], in[27]);
bus_driver_bit I0_28_ ( out[28], in[28]);
bus_driver_bit I0_29_ ( out[29], in[29]);
bus_driver_bit I0_30_ ( out[30], in[30]);
bus_driver_bit I0_31_ ( out[31], in[31]);
bus_driver_bit I0_32_ ( out[32], in[32]);
bus_driver_bit I0_33_ ( out[33], in[33]);
bus_driver_bit I0_34_ ( out[34], in[34]);
bus_driver_bit I0_35_ ( out[35], in[35]);
bus_driver_bit I0_36_ ( out[36], in[36]);
bus_driver_bit I0_37_ ( out[37], in[37]);
bus_driver_bit I0_38_ ( out[38], in[38]);
bus_driver_bit I0_39_ ( out[39], in[39]);
bus_driver_bit I0_40_ ( out[40], in[40]);
bus_driver_bit I0_41_ ( out[41], in[41]);
bus_driver_bit I0_42_ ( out[42], in[42]);
bus_driver_bit I0_43_ ( out[43], in[43]);
bus_driver_bit I0_44_ ( out[44], in[44]);
bus_driver_bit I0_45_ ( out[45], in[45]);
bus_driver_bit I0_46_ ( out[46], in[46]);
bus_driver_bit I0_47_ ( out[47], in[47]);
bus_driver_bit I0_48_ ( out[48], in[48]);
bus_driver_bit I0_49_ ( out[49], in[49]);
bus_driver_bit I0_50_ ( out[50], in[50]);
bus_driver_bit I0_51_ ( out[51], in[51]);
bus_driver_bit I0_52_ ( out[52], in[52]);
bus_driver_bit I0_53_ ( out[53], in[53]);
bus_driver_bit I0_54_ ( out[54], in[54]);
bus_driver_bit I0_55_ ( out[55], in[55]);
bus_driver_bit I0_56_ ( out[56], in[56]);
bus_driver_bit I0_57_ ( out[57], in[57]);
bus_driver_bit I0_58_ ( out[58], in[58]);
bus_driver_bit I0_59_ ( out[59], in[59]);
bus_driver_bit I0_60_ ( out[60], in[60]);
bus_driver_bit I0_61_ ( out[61], in[61]);
bus_driver_bit I0_62_ ( out[62], in[62]);
bus_driver_bit I0_63_ ( out[63], in[63]);

endmodule
// Library - CSG_alu, Cell - adder_64, View - schematic
// LAST TIME SAVED: Nov 27 21:21:09 2002
// NETLIST TIME: Nov 28 22:57:43 2002
`timescale 1ps / 1ps
```

```
module adder_64 ( sum, a, b, clk );


output [63:0]  sum;

input [63:0]  a;
input [63:0]  b;
input [2:5]  clk;
supply0 VSS_;

// Buses in the design

wire  [63:0]  cout;

wire  [63:0]  gout;

wire  [63:0]  pout;

wire  [63:0]  gbar;

wire  [63:0]  pbar;

wire  [63:0]  psum;

// List of primary aliased buses



sum_xor_64 I3 ( cout[63:0], psum[63:0], clk[5], sum[63:0]);
even_carry_gen_64 I2 ( pout[63:0], gout[63:0], cout[63:0]);
carry_merge_64 I1 ( pbar[63:0], gbar[63:0], clk[3:5], pout[63:0],
     gout[63:0]);
pbar_gbar_gen_64 I0 ( clk[2], clk[3], clk[4], clk[5], a[63:0], b[63:0],
     pbar[63:0], gbar[63:0], psum[63:0]);

endmodule
// Library - CSG_alu_syn, Cell - alu@sheet002, View - schematic
// LAST TIME SAVED: Nov 28 22:55:43 2002
// NETLIST TIME: Nov 28 22:57:43 2002
`timescale 1ps / 1ps

module cdsModule_8 ( bus_out[63:0], a[63:0], b[63:0], clk[2], clk[3],
     clk[4], clk[5], logical_control[2:0], logical_in0[63:0],
     logical_in1[63:0] );


output [63:0]  bus_out;

input [63:0]  a;
input [63:0]  b;
input [63:0]  logical_in0;
input [63:0]  logical_in1;
input [2:0]  logical_control;
input [2:5]  clk;
supply0 VSS_;

// Buses in the design

wire  [63:0]  sum;

wire  [63:0]  sum_out;

// List of primary aliased buses



mux31 I6 ( sum_out[63:0], clk[5], sum[63:0], logical_in0[63:0],
     logical_in1[63:0], logical_control[2:0]);
bus_driver I7 ( bus_out[63:0], sum_out[63:0]);
adder_64 I4 ( sum[63:0], a[63:0], b[63:0], clk[2:5]);

endmodule
// Library - CSG_alu_syn, Cell - alu@sheet001, View - schematic
// LAST TIME SAVED: Nov 28 22:48:30 2002
// NETLIST TIME: Nov 28 22:57:43 2002
`timescale 1ps / 1ps

module cdsModule_9 ( a[63:0], b[63:0], a0[63:0], a1[63:0], a2[63:0],
     a3[63:0], a4[63:0], a5[63:0], a6[63:0], a7[63:0], b0[63:0],
     b1[63:0], b2[63:0], b3[63:0], b4[63:0], b5[63:0], b6[63:0],
     b7[63:0], bus_out[63:0], clk[1], clk[2], mux_control_0[8:0],
     mux_control_1[8:0], shift_control[4:0], sign_control[1:0] );


output [63:0]  a;
output [63:0]  b;

input [63:0]  a7;
input [63:0]  a6;
input [63:0]  a5;
input [63:0]  a4;
input [63:0]  a3;
input [63:0]  a2;
input [63:0]  a1;
input [63:0]  a0;
input [63:0]  b7;
input [63:0]  b6;
input [63:0]  b5;
input [63:0]  b4;
input [63:0]  b3;
input [63:0]  b2;
input [63:0]  b1;
input [63:0]  b0;
input [1:0]  sign_control;
input [4:0]  shift_control;
input [1:2]  clk;
input [63:0]  bus_out;
input [8:0]  mux_control_1;
input [8:0]  mux_control_0;
supply0 VSS_;

// Buses in the design

wire  [63:0]  b_mux;

wire  [319:0]  a_mux;

wire  [0:2]  delayed_clk1;

wire  [63:0]  bus_out_latch;

// List of primary aliased buses

// List of all aliases


assign delayed_clk1[0] = clk[1];
assign a_mux[124:65] = a_mux[319:260];
assign a_mux[124:65] = a_mux[59:0];
assign a_mux[124:65] = a_mux[254:195];
assign a_mux[124:65] = a_mux[189:130];
assign a_mux[125] = a_mux[255];
assign a_mux[125] = a_mux[60];
assign a_mux[125] = a_mux[190];
assign a_mux[126] = a_mux[191];
assign a_mux[126] = a_mux[61];
assign a_mux[127] = a_mux[62];



flip_flop I23 ( delayed_clk1[0], bus_out[63:0], bus_out_latch[63:0]);
mux21 I3 ( b[63:0], clk[2], b_mux[63:0], sign_control[1:0]);
mux51 I2 ( a[63:0], delayed_clk1[2], a_mux[63:0], a_mux[127:64],
     a_mux[191:128], a_mux[255:192], a_mux[319:256],
     shift_control[4:0]);
mux91_b I1 ( b_mux[63:0], delayed_clk1[0], clk[2], b0[63:0], b1[63:0],
     b2[63:0], b3[63:0], b4[63:0], b5[63:0], b6[63:0], b7[63:0],
     bus_out_latch[63:0], mux_control_1[8:0]);
mux91_a I0 ( a_mux[63:0], delayed_clk1[0], a0[63:0], a1[63:0],
     a2[63:0], a3[63:0], a4[63:0], a5[63:0], a6[63:0], a7[63:0],
     bus_out_latch[63:0], mux_control_0[8:0]);
back_to_back_inv I5_0_ ( delayed_clk1[0], delayed_clk1[1]);
back_to_back_inv I5_1_ ( delayed_clk1[1], delayed_clk1[2]);
thru I19_0_ ( VSS_, a_mux[259]);
thru I19_1_ ( VSS_, a_mux[258]);
thru I19_2_ ( VSS_, a_mux[257]);
thru I19_3_ ( VSS_, a_mux[256]);
thru I18_0_ ( VSS_, a_mux[194]);
thru I18_1_ ( VSS_, a_mux[193]);
thru I18_2_ ( VSS_, a_mux[192]);
thru I17_0_ ( VSS_, a_mux[129]);
thru I17_1_ ( VSS_, a_mux[128]);
thru I16 ( VSS_, a_mux[64]);

endmodule
// Library - CSG_alu_syn, Cell - alu, View - schematic
// LAST TIME SAVED: Nov 28 22:55:49 2002
// NETLIST TIME: Nov 28 22:57:43 2002
`timescale 1ps / 1ps

module alu ( bus_out[63:0], a0[63:0], a1[63:0], a2[63:0], a3[63:0],
     a4[63:0], a5[63:0], a6[63:0], a7[63:0], b0[63:0], b1[63:0],
     b2[63:0], b3[63:0], b4[63:0], b5[63:0], b6[63:0], b7[63:0],
     clk[1], clk[2], clk[3], clk[4], clk[5], logical_control[2:0],
     logical_in0[63:0], logical_in1[63:0], mux_control_0[8:0],
     mux_control_1[8:0], shift_control[4:0], sign_control[1:0] );
```

```
output [63:0]  bus_out;

input [63:0]  a0;
input [63:0]  a1;
input [63:0]  a2;
input [63:0]  a3;
input [63:0]  a4;
input [63:0]  a5;
input [63:0]  a6;
input [63:0]  a7;
input [63:0]  b0;
input [63:0]  b1;
input [63:0]  b2;
input [63:0]  b3;
input [63:0]  b4;
input [63:0]  b5;
input [63:0]  b6;
input [63:0]  b7;
input [4:0]  shift_control;
input [1:5]  clk;
input [1:0]  sign_control;
input [2:0]  logical_control;
input [63:0]  logical_in0;
input [63:0]  logical_in1;
input [8:0]  mux_control_0;
input [8:0]  mux_control_1;
supply0 VSS_;

// Buses in the design

wire  [63:0]  a;

wire  [63:0]  b;

// List of primary aliased buses


cdsModule_8 SH2 ( bus_out[63:0], a[63:0], b[63:0], clk[2], clk[3],
     clk[4], clk[5], logical_control[2:0], logical_in0[63:0],
     logical_in1[63:0]);
cdsModule_9 SH1 ( a[63:0], b[63:0], a0[63:0], a1[63:0], a2[63:0],
     a3[63:0], a4[63:0], a5[63:0], a6[63:0], a7[63:0], b0[63:0],
     b1[63:0], b2[63:0], b3[63:0], b4[63:0], b5[63:0], b6[63:0],
     b7[63:0], bus_out[63:0], clk[1], clk[2], mux_control_0[8:0],
     mux_control_1[8:0], shift_control[4:0], sign_control[1:0]);

endmodule
```

C.2 Behavioral Verilog Transistor Models

For the Verilog transistor models used for switch-level simulations, the normal NMOS/PMOS can overdrive the resistive NMOS/PMOS in the case when there are multiple drivers on a net. The "trireg" statements enable the simulation of dynamic logic, since the previously driven logic value is retained when there is no direct path to V_{dd} or V_{ss}.

```
// Verilog HDL for "cmosp18Local", "nfet3" "verilog"

module nfet3 (D, G, S);
    inout D;
    inout G;
    inout S;

//    trireg D;
//    trireg S;
    trireg G;

    nmos ( D, S, G );

endmodule

// Verilog HDL for "cmosp18Local", "nfet3_res" "verilog"

module nfet3_res (D, G, S);
    inout D;
    inout G;
    inout S;

//    trireg D;
//    trireg S;
    trireg G;
    rnmos ( D, S, G );

endmodule

// Verilog HDL for "cmosp18Local", "pfet3" "verilog"

module pfet3 (D, G, S);
    inout D;
    inout G;
    inout S;

//    trireg D;
//    trireg S;
    trireg G;

    pmos ( D, S, G );

endmodule

// Verilog HDL for "cmosp18Local", "pfet3_res" "verilog"

module pfet3_res (D, G, S);
    inout D;
    inout G;
    inout S;

//    trireg D;
//    trireg S;
    trireg G;

    rpmos ( D, S, G );

endmodule
```

C.3 Behavioral Verilog Multi-phase Clock Generator

This clock generator, can skew adjacent clocks arbitrarily according to the values of the input parameters.

```
// Verilog HDL for "CSG_testbench", "clock_gen_four_phase" "behavioral"

// Testbench does not work for greater than 75% duty cycle clocks

//module clock_gen_five_phase(phi,phiand,phior,phidelay) ;
module clock_gen_five_phase(phi,phiand,phior,phidelay,phigate) ;

parameter period = 32;
parameter pulse_width = 16;
parameter num_phases = 5;
parameter delayed_clock_pulse_width = 24;

// skew parameters
// must be positive

parameter phi1_skew = 0;
parameter phi2_skew = 0;
parameter phi3_skew = 0;
parameter phi4_skew = 0;
parameter phi5_skew = 0;

parameter phidelay1_skew = 0;
parameter phidelay2_skew = 0;
parameter phidelay3_skew = 0;
parameter phidelay4_skew = 0;
parameter phidelay5_skew = 0;

parameter half_period = period/2;
parameter Tc_over_N = period/num_phases;

//define clock outputs
output[1:5] phi;
output[1:5] phiand;
output[1:5] phior;
output[1:5] phidelay;
output[1:5] phigate;

//clock outputs must hold their values
reg[1:5] phi;
reg[1:5] phiand;
reg[1:5] phior;
reg[1:5] phidelay;
reg[1:5] phigate;
```

```
//internal variables
reg clk, clk1, clk2, clk3, clk4, clk5;

initial
begin
    clk = 1'b0;
    clk1 = 1'b0;
    clk2 = 1'b0;
    clk3 = 1'b0;
    clk4 = 1'b0;
    clk5 = 1'b0;
end

// generate internal reference clocks
initial
begin
    forever
begin
#(half_period) clk = ~clk;
end
end

always@(posedge clk)
begin
    #0 clk1 = ~clk1;
    #pulse_width clk1 = ~clk1;
end

always@(posedge clk1)
begin
    #(Tc_over_N) clk2 = ~clk2;
    #pulse_width clk2 = ~clk2;
end

always@(posedge clk2)
begin
    #(Tc_over_N) clk3 = ~clk3;
    #pulse_width clk3 = ~clk3;
end

always@(posedge clk3)
begin
    #(Tc_over_N) clk4 = ~clk4;
    #pulse_width clk4 = ~clk4;
end

always@(posedge clk4)
begin
    #(Tc_over_N) clk5 = ~clk5;
    #pulse_width clk5 = ~clk5;
end

// set all skew-enabled clocks to initial values
initial
begin
    phi[1:5] = 5'h0;
    phiand[1:5] = 5'h0;
phior[1:5] = 5'h0;
phidelay[1:5] = 5'h0;
phigate[1:5] = 5'h0;
end

// generate different delayed phases of the clock
always@(posedge clk1)
begin
    #phi1_skew phi[1] = ~phi[1];
    #pulse_width phi[1] = ~phi[1];
end

always@(posedge clk2)
begin
#phi2_skew phi[2] = ~phi[2];
#pulse_width phi[2] = ~phi[2];
end

always@(posedge clk3)
begin
#phi3_skew phi[3] = ~phi[3];
#pulse_width phi[3] = ~phi[3];
end

always@(posedge clk4)
begin
#phi4_skew phi[4] = ~phi[4];
#pulse_width phi[4] = ~phi[4];
end

always@(posedge clk5)
begin
```

```
#phi5_skew phi[5] = ~phi[5];
#pulse_width phi[5] = ~phi[5];
end

// generate anded versions of adjacent clocks

always@(phi[1] or phi[2])
begin
phior[1] = phi[1] | phi[2];
end

always@(phi[2] or phi[3])
begin
phior[2] = phi[2] | phi[3];
end

always@(phi[3] or phi[4])
begin
phior[3] = phi[3] | phi[4];
end

always@(phi[4] or phi[5])
begin
phior[4] = phi[4] | phi[5];
end

always@(phi[5] or phi[1])
begin
phior[5] = phi[5] | phi[1];
end


// generate or versions of adjacent clocks

always@(phi[1] or phi[2])
begin
phiand[1] = phi[1] & phi[2];
end

always@(phi[2] or phi[3])
begin
phiand[2] = phi[2] & phi[3];
end

always@(phi[3] or phi[4])
begin
phiand[3] = phi[3] & phi[4];
end

always@(phi[4] or phi[5])
begin
phiand[4] = phi[4] & phi[5];
end

always@(phi[5] or phi[1])
begin
phiand[5] = phi[5] & phi[1];
end


// generate delayed clocks for OTB domino

always@(posedge clk1)
begin
#phidelay1_skew phidelay[1] = ~phidelay[1];
#delayed_clock_pulse_width phidelay[1] = ~phidelay[1];
end

always@(posedge clk2)
begin
#phidelay2_skew phidelay[2] = ~phidelay[2];
#delayed_clock_pulse_width phidelay[2] = ~phidelay[2];
end

always@(posedge clk3)
begin
#phidelay3_skew phidelay[3] = ~phidelay[3];
#delayed_clock_pulse_width phidelay[3] = ~phidelay[3];
end

always@(posedge clk4)
begin
#phidelay4_skew phidelay[4] = ~phidelay[4];
#delayed_clock_pulse_width phidelay[4] = ~phidelay[4];
end

always@(posedge clk5)
begin
#phidelay5_skew phidelay[5] = ~phidelay[5];
```

```
#delayed_clock_pulse_width phidelay[5] = ~phidelay[5];
end

// generate gated evaluations

always@(phi[1] or phi[2])
begin
phigate[1] = phi[1] & ~phi[2];
end

always@(phi[2] or phi[3])
begin
phigate[2] = phi[2] & ~phi[3];
end

always@(phi[3] or phi[4])
begin
phigate[3] = phi[3] & ~phi[4];
end

always@(phi[4] or phi[5])
begin
phigate[4] = phi[4] & ~phi[5];
end

always@(phi[5] or phi[1])
begin
phigate[5] = phi[5] & ~phi[1];
end

//always@(phidelay[1] or phidelay[3])
//begin
// phigate[1] = phidelay[1] & ~phidelay[3];
//end

//always@(phidelay[2] or phidelay[4])
//begin
// phigate[2] = phidelay[2] & ~phidelay[4];
//end

//always@(phidelay[1] or phidelay[3])
//begin
// phigate[3] = ~phidelay[1] & phidelay[3];
//end

//always@(phidelay[2] or phidelay[4])
//begin
// phigate[4] = ~phidelay[2] & phidelay[4];
//end

endmodule
```

C.4 Behavioral Verilog ALU Stimulus

This Verilog testbench stimulus, provided test vectors and control signals for the 64-bit ALUs under test.

```
// Verilog HDL for "CSG_alu", "alu_stim" "behavioral"

module alu_stim(a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7,
log0,log1,mux_control_0,mux_control_1,shift_control,sign_control,logical_control);

parameter NUMBITS = 64;
parameter PERIOD = 1000;
parameter HALF_PERIOD = PERIOD/2;
parameter FIFTH_PERIOD = PERIOD/5;

// ALU stimulus

output [NUMBITS-1:0] a0,a1,a2,a3,a4,a5,a6,a7;
output [NUMBITS-1:0] b0,b1,b2,b3,b4,b5,b6,b7;
output [NUMBITS-1:0] log0,log1;

output [8:0] mux_control_0;
output [8:0] mux_control_1;
output [4:0] shift_control;
output [1:0] sign_control;
output [2:0] logical_control;

reg [NUMBITS-1:0] a0,a1,a2,a3,a4,a5,a6,a7;
reg [NUMBITS-1:0] b0,b1,b2,b3,b4,b5,b6,b7;
reg [NUMBITS-1:0] log0,log1;

reg [8:0] mux_control_0;
reg [8:0] mux_control_1;
reg [4:0] shift_control;
reg [1:0] sign_control;
reg [2:0] logical_control;

// Internal Memories for reading in text data

reg [NUMBITS-1:0] a[0:7];
reg [NUMBITS-1:0] b[0:7];
reg [NUMBITS-1:0] log[0:1];

reg [NUMBITS-1:0] alu_vec[0:27];
reg [NUMBITS-1:0] alu_out;

// Local variables

integer i;

initial
begin
$readmemh("/homes/sung/asp/asp_cmosp18_rs.Work/CSG_alu/adder_64_stim/behavioral/adder_64_vec_a.txt",a);
$readmemh("/homes/sung/asp/asp_cmosp18_rs.Work/CSG_alu/adder_64_stim/behavioral/adder_64_vec_b.txt",b);
$readmemh("/homes/sung/asp/asp_cmosp18_rs.Work/CSG_alu/adder_64_stim/behavioral/logical_vec.txt",log);
$readmemh("/homes/sung/asp/asp_cmosp18_rs.Work/CSG_alu/adder_64_stim/behavioral/alu_vec_outputs.txt",alu_vec);
end

initial
begin
a0 = a[0];
a1 = a[1];
a2 = a[2];
a3 = a[3];
a4 = a[4];
a5 = a[5];
a6 = a[6];
a7 = a[7];

b0 = b[0];
b1 = b[1];
b2 = b[2];
b3 = b[3];
b4 = b[4];
b5 = b[5];
b6 = b[6];
b7 = b[7];

log0 = log[0];
log1 = log[1];

alu_out = 64'b0;

mux_control_0 = 9'b000000000;
mux_control_1 = 9'b000000000;
shift_control = 5'b00001;
sign_control = 2'b01;
logical_control = 3'b001;
end

initial
begin
#PERIOD;
#HALF_PERIOD;

mux_control_0 = 9'b000000001;
mux_control_1 = 9'b000000001;

// test first data set a0 + b0

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[0];

// test functionality of first 9:1 multiplexer

for (i = 1; i < 9; i = i + 1)
begin
mux_control_0 = mux_control_0 << 1;  // b0 + {a1:a7,sum(b0+a7)}
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[i];
end

// test loopback second time
```

```
    #FIFTH_PERIOD;
    #FIFTH_PERIOD;
    #FIFTH_PERIOD;
    #FIFTH_PERIOD;
    #FIFTH_PERIOD;
alu_out = alu_vec[9];  // b0 + (sum(b0+a7) + b0)

mux_control_0 = 9'b000100000;
mux_control_1 = 9'b000000001;

// a5 + b0

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[10];  // a5 + b0

// test functionality of second 9:1 multiplexer

for (i = 11; i < 19; i = i + 1)
begin
mux_control_1 = mux_control_1 << 1;  // a5 + {b1:b7,sum(a5+b7)}
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[i];
end

// test loopback second time

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[19];  // a5 + (sum(a5+b7) + a5)

// select a3 and b6

mux_control_0 = 9'b000001000;
mux_control_1 = 9'b001000000;

// test shift control

for (i = 20; i < 24; i = i + 1)
begin
shift_control = shift_control << 1;  // b6 + {a5<<1,a5<<2,a5<<3,a5<<4}
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[i];
end

shift_control = 5'b00001;  // reset shift to shift << 0

// test logical control

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
logical_control = 3'b010;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[24];

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
logical_control = 3'b100;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[25];

// test subtraction a6 - b1
mux_control_0 = 9'b001000000;
mux_control_1 = 9'b000000010;
a0 = 64'b1;  // load constant 1
sign_control = 2'b10; // select inverted values

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
```

```
logical_control = 3'b001;  // deselect logical outputs
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[26];

mux_control_0 = 9'b000000001;  // select constant 1
mux_control_1 = 9'b100000000;  // select loop back
sign_control = 2'b01;  // select non-inverted values again

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
alu_out = alu_vec[27];

#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;
#FIFTH_PERIOD;

$finish;
end
endmodule
```

Appendix D

Test Vectors

D.1 C Program for generating random input vectors for Adder and ALU

```
#include <stdio.h>
#include <stdlib.h>

main()
{
int i, numbers, seed;

printf("// Enter the number of input values to generate");
scanf("%d", &numbers);
printf(" %d\n", numbers);

printf("// Enter seed value to start random number generation");
scanf("%d", &seed);
printf(" %d\n", seed);

srand(seed);

    for (i = 0; i < numbers; i++)  {
printf("%04x%04x%04x%04x\n", rand(), rand(), rand(), rand());
}
}
```

D.2 Input Vectors for Adder and ALU

```
// A-input
//
// Enter the number of input values to generate 100
// Enter seed value to start random number generation 50
58bb579e3f9353a9
132a23280f176673
4e3154646e1403ac
32f267c651c609d4
70cf0c3f4dde4cea
3fcc2c6a64435173
2ca15e7c148d2c4a
0aa266e8673e132b
274f39d570be69fe
1bbb2740081d58e3
579d1df12d873701
4237311f5c86560e
0ceb4ce245fb7a9e
1396455d2dda0d2d
66b32ca768a84feb
5b311b7e34b6254a
3c0f58aa6b1b3348
79bd0b3344726f7c
733363422d585fe3
34cd65d766a41c68
48ed47306d655d35
764f7ff5669d21bc
082b09240abe3c83
7c0c6abf4ce84fb4
3173093739e00660
772b5d964ad01697
```

```
126a047027c15828
1bac4f8715d87c38
4d4f46410d543185
1fef40ca33434178
507f0a09330771ec
2c2c3444338c37c1
3df20c926446061f
39fa59045ded468a
42ba4d9252f845aa
63cb63c74bdb20d8
5e89012c6aff0068
246c187674852ab9
1b29316f15bb3bfd
068803a4285a0ec9
34040fd46d86215c
442464137c8303af
2d9c76c161361a40
5623442f2663419f
5d121b0a47a01eb5
73c1438f471c27d7
5fa16d6d6312328f
02191079270a1224
02212c1454d612ef
73a1115c61485e5d
1887241500b413c3
17aa3e567f2921e5
456222f3606e2d44
59e42aae03be0f2b
315e181c47433978
71d63e5967540b2b
32c3666b156f61b0
02681f7702f572ec
64f465a65ba83c09
295b4bd56be51102
2ff213ff6ea118ed
7ecd626a3f24430c
3fd877946a880092
04b81cdc5cf04f35
6c5f2ff126896077
067178fb144269b4
105a378b184b68ec
682c12433e4f6a4e
5139554527701a89
66742b5c4a0631d2
56883ded711803b3
45b6679c37963f9a
716f65bd7b5c2e1d
71b5348103e7770e
343201dd58d53c43
0d156339041e1522
79b039de3c1536ef
70d2721c311c55d5
5ce6093e7b2a0ab5
1bc8062f62fe49b2
206b50e903203579
122b14a17c9c5b51
05f318b43b7e23e6
2d0e3c4f070c04d5
15ce00e159c63ef6
59df4f423d1e356d
566863a024f82970
49e70c2d06e066d5
73c9268a290d2d63
11cc07f2651962d5
57153c275a7e59ad
3910471d4cd038bf
2e0c556729bd4f78
3992076572c462f3
6288432b06b83fba
6f0a393006f41c5c
020407bb545b18b3
768f290d601665f2
1511184f4c0d6371
7e13593a17223c38

// B-input
//
// Enter the number of input values to generate 100
// Enter seed value to start random number generation 200
62ed62e4093f4b78
23ab464300563274
2ab66f0428a351aa
09752cbf4ffa7819
47ed7d92053264b1
0eff51ac02c432c1
3a657b9827282124
0e6b594a16f0202b
0d6d01ea64a15249
291404645a546b81
```

```
4b4f02ac41111b90
731335d241e66eda
60d326c93ccb5215
0ebe27f7313624b7
70bc5b7c147c0620
3826524231ee4987
17470fc970af3ca6
40977eaf1b572d25
1db12202054958c8
7a190f4629d43457
1db07d45210c3551
12f3739906660a4f
14f766fa2d166e39
61254e483e29022c
68b57c571c615a27
1dec4a7e29d42878
59145fde4b4333e9
11402173453b04a9
64bc16dc4eee73fc
2d564fea76cf0aa2
1c5016ea34ef5a09
1a207bb3471a3c32
65ee036e32b02663
30ca092808457a91
30b11b1944f8038c
673e60b26d9407e8
183055683f681faf
2b9f644312e638c8
77fd30254bfe7626
2fcf14db743855b0
6f292ce55a9448f3
24eb680555212c8a
53be7e8a005f4a49
66cc4d591856522c
164166c047737602
178663fb072513d9
153843826e3c1b28
2aea601808fc18bf
60a04c941704156f
6206206e4ae13379
6d730109677136b7
36a273c256f9638c
392a44bd1806608d
36c30e6a1c0406ec
5d372dd9739f4da8
50292fc364dd3b77
1289025446f80b6e
0bd377b93ffd7076
250a656e4025236f
39786c4556f62821
572235353e1974e7
0b0d2ee835fa691a
3534180210223e3e
2044623303545fef
591d39fb25695689
02093f4026eb309a
1dba3a912c741709
0e065b3861c57803
42614a0222a574a9
521e1ccd55807d7a
7e6576d653bb4982
57f361bf7bd81440
04952b63494e4e58
6062545a0e262cfd
76bb5b4b2a55441c
0f0370f45cdc1749
491f60fa0aa24d6a
05ac3b72170c7326
16030b2b2a62780b
6fee24c101e15282
61275a6325a07673
7e94205f20230ab7
4b456e7213c00941
532969d249b6360c
359323f7170718c4
5b717a2d33196534
554661db5c107e72
1cec2d9164e800cc
370c16d309567e71
705918a7235c5adf
328f66e01eae7110
2d2f0e2945c87064
4df708d144cc1c4d
452354587e1d5abc
116653bd0cbc3d4f
4fa70a8510657137
4a7c7c8b1f6841eb
05673e8b7a491a8e
3fd3036c5d173221
```

```
2bcc32510a8c4e68

// Input from Logical Units
//
// Enter the number of input values to generate 2
// Enter seed value to start random number generation 80
0df826ac4e4f386c
301176fa3f574273
```

D.3 Expected Results of ALU operations

The following numbers are in hexadecimal format.

```
//Output =  b0 + {a0:a7,sum(b0+a7)} then b0 + {b0 + sum(b0+a7)}
//Output =  a5 + {b1:b7,sum(a5+b7)} then a5 + {a5 + sum(a5+b7)}
//Output = b[6] + {a5<<1,a5<<2,a5<<3,a5<<4}
//Output = log0,log1
//Output = {a6 + (~b1)} + 1} = a6 - b1
bba8ba8248d29f21
7617860c1856b1eb
b11eb74877534f24
95dfcaaa5b05554c
d3bc6f23571d9862
a2b98f4e6d829ceb
8f8ec1601dcc77c2
6d8fc9cc707d5ea3
d07d2cb079bcaa1b
336a8f9482fbf593
a2b98f4e6d829ceb
637772ad649983e7
6a829b6e8ce6a31d
49415929b43dc98c
87b9a9fc6975b624
4ecb7e1667078434
7a31a8028b6b7297
4e3785b47b33719e
8e03b21edf76c311
cdcfde8943ba1484
a04a4b24cab434cc
062f1ab16e404874
d1f8b9cab5586fc4
698bf7fd4388be64
0df826ac4e4f386c
301176fa3f574273
08f618391436f9d5
08f618391436f9d6
08f618391436f9d6
```

D.4 C Program that models ALU operations

```
#include <stdio.h>

#define FILE1 "adder_64_vec_a.txt"
#define FILE2 "adder_64_vec_b.txt"
#define FILE3 "logical_vec.txt"

main()
{
long long a[8],b[8],sum[10],temp,log[2];  //define 64 bit numbers
FILE *fp1,*fp2,*fp3;
short i;
char buf[128];  //character buffer to read away the comments

//get rid of first and second line of comments for file 1
fp1 = fopen(FILE1, "r");
fgets(buf,128,fp1);
fgets(buf,128,fp1);

//get rid of first and second line of comments for file 2
fp2 = fopen(FILE2, "r");
fgets(buf,128,fp2);
fgets(buf,128,fp2);

//get rid of first and second line of comments for file 2
fp3 = fopen(FILE3, "r");
fgets(buf,128,fp3);
fgets(buf,128,fp3);

//scan in eight numbers for input to the 9:1 muxes
for(i = 0; i < 8; i++)  {
```

```
fscanf(fp1, "%llx", &a[i]);
fscanf(fp2, "%llx", &b[i]);
}

//scan in logical inputs
for(i = 0; i < 2; i++)  {
fscanf(fp3, "%llx", &log[i]);
}

printf("//Output =  b0 + {a0:a7,sum(b0+a7)} then b0 + {b0 + sum(b0+a7)}\n");

printf("//Output =  a5 + {b1:b7,sum(a5+b7)} then a5 + {a5 + sum(a5+b7)}\n");
printf("//Output = b[6] + {a5<<1,a5<<2,a5<<3,a5<<4}\n");
printf("//Output = log0,log1\n");
printf("//Output = {a6 + (~b1)} + 1} = a6 - b1\n");

//test all select lines on first multiplexer
for(i = 0; i < 8; i++)  {
sum[i] = a[i] + b[0];
printf("%016llx\n",sum[i]);
}
//test loopback to first multiplexer
sum[8] = sum[7] + b[0];
printf("%016llx\n",sum[8]);

//test loopback to first multiplexer again
sum[9] = sum[8] + b[0];
printf("%016llx\n",sum[9]);

//test all select lines on second multiplexer
for(i = 0; i < 8; i++)  {
sum[i] = b[i] + a[5];
printf("%016llx\n",sum[i]);
}
//test loopback to second multiplexer
sum[8] = sum[7] + a[5];
printf("%016llx\n",sum[8]);

//test loopback to second multiplexer again
sum[9] = sum[8] + a[5];
printf("%016llx\n",sum[9]);

temp = a[3];

//test shift control
for(i = 0; i < 4; i++)  {
temp = temp << 1;
sum[i] = temp  + b[6];
printf("%016llx\n",sum[i]);
}

//test logical control
for(i = 0; i < 2; i++)  {
printf("%016llx\n",log[i]);
}

//test subtraction
temp = ~b[1];
sum[0] = a[6] + temp; // a6 + (~b1)
printf("%016llx\n",sum[0]);
sum[0] = sum[0] + 1;  // a6 - b1
printf("%016llx\n",sum[0]);

printf("%016llx\n",a[6]-b[1]);
}
```

Appendix E

SpectreVerilog Environment

E.1 Example SpectreVerilog Run script

```
spectre  -env artist4.4.6 +log ../psf/spectre.out -format psfbin
-raw ../psf   -mixmod -slvhost v880
'-slave"/scratch/sung/trans_CL_TT_25_Nov16_1ns/netlist/digital
verilog.vmx  -a +sxl_keep_minimum  +typdelays -y ./hdlFilesDir
+libext+.v+ +incdir+hdlFilesDir +sdf_verbose +sdf_nocheck_celltype
-l verilog.log +COSIM_WARNING_OFF +vmxwavedir../../psf -f
mmenvOpts testfixture.template -f verilog.inpfiles"' analog/input.scs
```

Appendix F

Mask Layouts

F.1 Mask Layout Plot of CL-Domino ALU

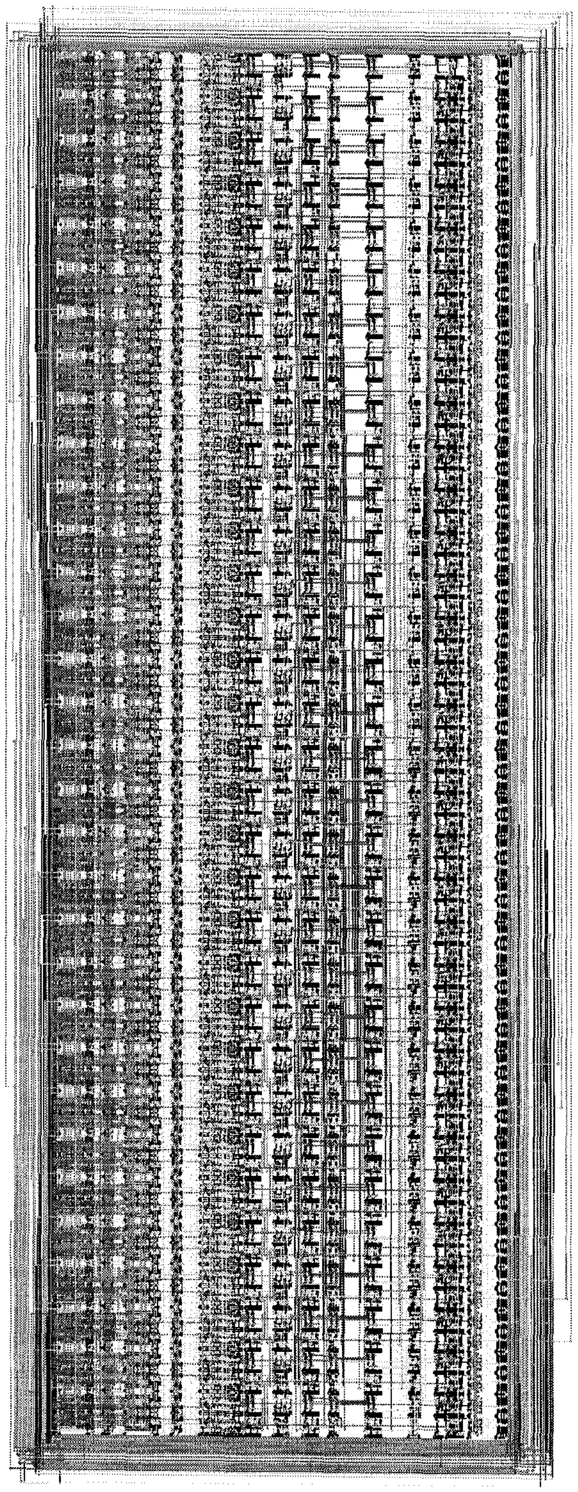

Figure F.1: Layout plot of CL-domino ALU.