ACT

By

ROBERT V. ST. DENIS
B.S. (Northeastern University) 2013

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical & Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Venkatesh Akella, Chair

Professor Rajeevan Amirtharajah

Adjunct Assistant Professor Christopher Nitta

Professor Matthew Farrens

Committee in Charge
2017

Contents

List of Figures

List of Tables

List of Acronyms

NTV	Near Threshold Voltage	iii
NTC	Near Threshold Computing	iii
VLSI	Very Large Scale Integration	1
IC	Integrated Circuit	1
FIFO	First-In First-Out	2
I/O	Input/Output	4
e1of2	Enable + 1 of 2	3
e1of4	Enable + 1 of 4	3
PCHB	Pre-Charged Half Buffer	4
PCFB	Pre-Charged Full Buffer	4
MSB	Most Significant Bit	6
LSB	Least Significant Bit	6
FSM	Finite State Machines	9
FA	Full Adder	7
CSA	Carry-Save Adder	13
STFB	Single-Track Full Buffer	13
PVT	Process, Voltage & Temperature	16
SRAM	Static Random Access Memory	16
CDP	Cached Dual-Ported	18
BIST	Built-In Self Test	19
MRU	Most Recently Used	18
ACT	Asynchronous Cache Tool	24
SECDED	Single-Error-Correcting Double-Error-Detecting	21

1

Introduction

"There's nothing remarkable about it. All one has to do is hit the right keys at the right time and the instrument plays itself."

– J.S. Bach

Although a myriad of highly-developed techniques exist in the field of Very Large Scale Integration (VLSI) Integrated Circuit (IC) design, two dominant paradigms are encountered universally throughout: synchronous systems, which leverage a shared clock to perform sequencing, and asynchronous systems, which do not [4][5][6]. Of the various forms of asynchronous design - all of which can be categorized in their use of timing assumptions to implement sequencing - the most robust form is that of delay-insensitive (DI) design [1][7]. In a fully DI system, the correct functionality of the circuitry remains independent of wire and gate delays, as by definition, all wires and composite gates can assume arbitrary and unbounded, yet finite and positive delays while maintaining error-free operation [7][4]. Despite this attractive quality however, [8] has formally proven that the set of circuits adhering to these requirements are not only limited in practicality, but are also Turing incomplete, inevitably giving rise to the notion of Quasi Delay insensitivity (QDI) [7]. Much like DI circuits, QDI circuits uphold identical delay requirements, with the single exception of the isochronic fork timing assumption [9], which requires the delay on one branch of a forked wire to be less than the delay through a transitional/gate sequence on the other end of the fork [10].

To place both ideologies in terms of analogy, designing a purely DI system would be equivalent to writing a song in a major scale, without using either of the scale's half-steps or notes beyond those remaining: while seemingly conceivable at first, one quickly realizes the practicality is ineffably limited and the scale remains incomplete. The isochronic fork is analogous to the scale's half steps, as additional timing assumptions are to leveraging notes beyond the scale's scope - both are essential to the musical system and inherent full-bodied creations, but improper placement or overuse of either results in dissonance and perceived noise. The analogy is further intended to be self-encompassing, as synchronous systems also lend themselves to rhythmic & harmonious metaphors.

In contrast to the synchronous paradigm however, which requires worst-case timing guard-bands, asynchronous QDI design delivers average-case performance to always operate as best it can for any set of inputs and actual case circuit parameters, rather than remaining limited to a constraint imposed by a shared worst-case clock or matched delay. As a result, this independence of shared global information produces increased performance and highly modular interfaces, in contrast to circuits requiring complex control logic often found in synchronous systems [5]. Furthermore, as each circuit can be customized to perform a specific function, the implementation style naturally lends itself to highly pipelined & multi-faceted designs along with omnipresent concurrency[11].

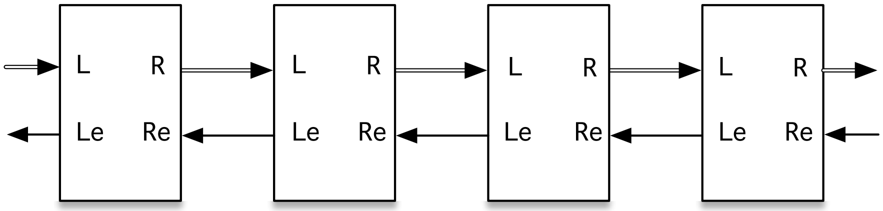


Figure 2.1: A generalized linear asynchronous QDI pipeline.
The request is encoded as a 1 of N signal, while the enables represent inverted acknowledges.

A linear pipeline is depicted in Figure 2.1 and can be described as a sequence of consecutively connected cells, with the right outputs of one cell connected to the left inputs of the next. As tokens are placed into the foremost left input, they sequentially propagate from left to right in a First-In First-Out (FIFO) fashion, under the condition that the next successive cell is vacantly available to receive the token; otherwise the propagation is stalled until the token can be received [11].

As there is no globally shared information between the circuits, cells require the use of enabling and acknowledgement for communication between one another, in order to convey the receiving availability as well as successful token transmission. This use of handshaking between cells allows the relevant sending & receiving state to be encoded within the token's propagation [12], giving rise to the notion of "self-timed" [5] circuits.

As each circuit must be available to receive a token prior to successful transmission, the concept of "slack" must be introduced. The token capacity of a buffer is defined as the *static slack* - buffers than can simultaneously send a and receive a token are said to have a static slack of 1, while those that can only send or receive at any given time are defined as having a static slack of 1-half, consequently defining the nomenclature of "half -buffers" and "full-buffers" [11]. Furthermore, while the majority of asynchronous literature focuses on depicting dual-rail or Enable + 1 of 2 (e1of2) implementations because they are the easiest to understand and most commonly utilized, asynchronous QDI prefers the use of Enable + 1 of 4 (e1of4) DI codes in order to mitigate the transitional energy penalty of the four-phase handshake [13]. Because an e1of4 code encodes 2 bits, the number of transitions is effectively halved compared to that of using e1of2 codes, as one would need two e1of2 codes to encode 2 bits.

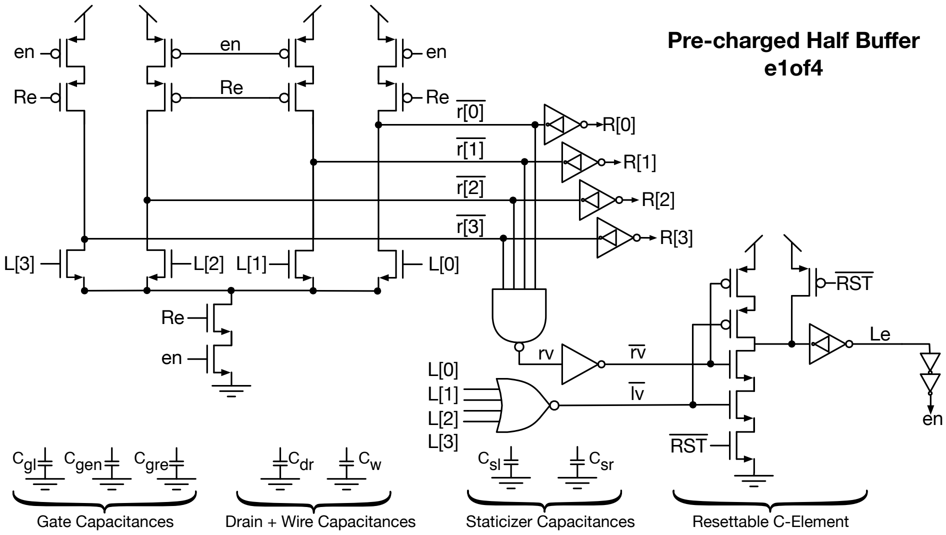


Figure 1.2: Detailed implementation of an e1of4 PCHB
Critical parasitic capacitances have also been labeled for clarity as follows: gate capacitances of the left input, internal and right enables; drain capacitances attached to $\bar{r}$, capacitances of the wires, and the capacitances internal to cross-coupled staticizers, left and right [14].

To visualize the contents of one type of half buffer, consider the e1of4 Pre-Charged Half Buffer (PCHB) shown in Figure 2.2. The circuit's e1of4 left Input/Output (I/O) is denoted as L[3:0] & *Le*, while the e1of4 right I/O is denoted as R[3:0] & *Re*. Signals beginning with capital letters are cell I/O signals, while lower case letters distinguish signals internal to the cell. Before the circuit can be of use, it must be placed into a valid state [15] by asserting the active low reset signal $\overline{RST}$. When asserted, the circuit's internal-enable *en* and left enable *Le* are brought low allowing the output rails $\overline{r[3:0]}$ to pre-charge to a logic '1', consequently setting the inverted outputs R[3:0] to logic '0'. In addition, it's assumed that the circuit attached to the cell's output brings its left enable low, as the cell's right enable *Re* must also hold the value of logic '0' for the pre-charge to take place [16]. Once the output rails have sufficiently charged, the right valid signal *rv* holds logic '0', while the inverted left and right valid signals $\overline{rv}$ and $\overline{lv}$ contain logic '1'. After the reset has succeeded, $\overline{RST}$ is released and set to the default value of '1' allowing the output of the C-element to return to the cell's idle state of '1'. During operation, 1of4 data arrives at the input L to pull one of the four output rails $\bar{r}$ low, while simultaneously causing the output of the NOR gate $\overline{lv}$ to cease firing. After $\bar{r}$ has fallen, *rv* fires, $\overline{rv}$ ceases to fire, and *Le* & *en* are pulled low to acknowledge Left input data. In parallel, the successive pipelined block acknowledges the received propagating data, thus bringing *Re* low, consequently allowing $\bar{r}$ to pre-charge once again, returning the PCHB to it's idle state for the process to repeat.

Figure 2.3 depicts an e1of4 Pre-Charged Full Buffer (PCFB). In order to simultaneously transmit while receiving, the full buffer requires an extra state-holding element, implemented using the asymmetric C-element. All pseudo-static implementations of C-elements and pre-charged, non-complementary outputs require the use of a cross-coupled staticizers, in order to prevent their pre-charges from leaking away when held for extended durations of time; [15] provides a comprehensive overview of methods for implementing staticized buffers with reset capability. This use of the extra state holding element allows the left and right handshakes to reset in parallel [7].

In addition to the static slack, metrics of *forward latency*, *throughput*, *cycle time*, *dynamic slack*, and *energy per cycle* are useful for quantifying the characteristics of asynchronous pipelines. The forward latency is defined as the time it takes a given token to travel the full length of the pipeline, while the throughput is defined as the rate a given pipeline produces a result. The cycle time is the inverse of the throughput, and the energy per cycle is the integral of the pipeline's

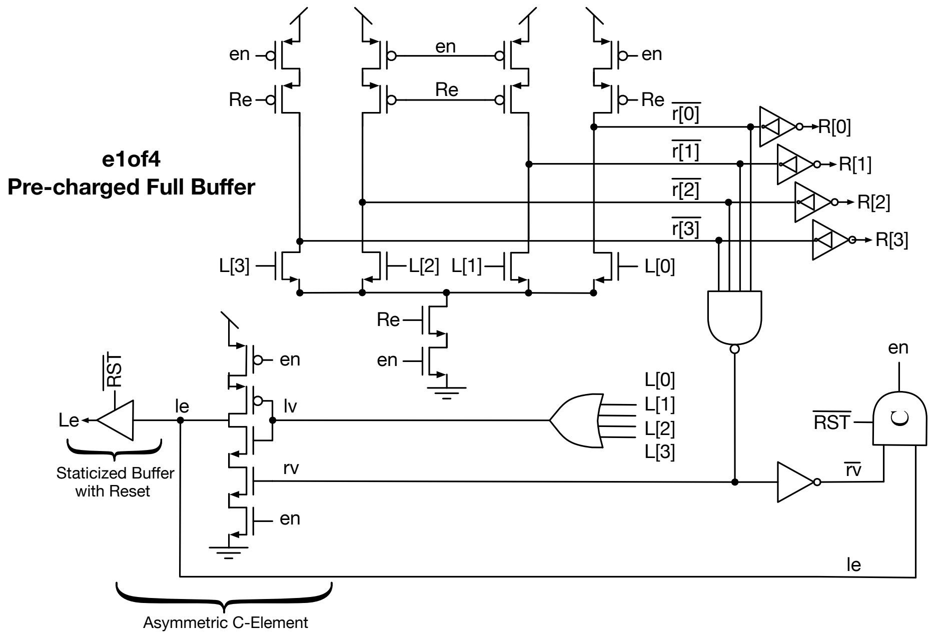


Figure 1.3: Detailed implementation of an e1of4 PCFB.
The implementation can be further improved by altering the internal and left enable computations.

power trace, beginning from the production of one result and ending with production of the next. Furthermore, the static slack defines the maximum token capacity of a given pipeline, while the dynamic slack defines the number of tokens present in the pipeline in order to operate at peak throughput.

2

Cell Design

"How can I help but see what is in front of my eyes? Two and two are four."

"Sometimes, Winston. Sometimes they are five. Sometimes they are three. Sometimes they are all of them at once. You must try harder."

– George Orwell, *1984*

Similar to many concepts in digital design, the first step to designing a cell by hand is simply to make a truth table. Table 2.1 provides the logic for implementing a 2-bit vertical XOR. In summary, the function performs the following: XOR the Most Significant Bit (MSB) of A with the MSB of B, XOR the Least Significant Bit (LSB) of A with the LSB of B, and concatenate the result. For example, in binary: A=10, B=11; XOR(A[1], B[1]) = 0; XOR(A[0], B[0]) = 1; Result = 01; Because QDI prefers the use of e1of4 codes however, the table is of maximal use after it has been re-written using quaternary logic, therefore 2.1 provides both binary and quaternary systems. From the quaternary logic table, each result is then grouped together to form the pulldown logic, which is shown in Table 2.2.

From the pulldown table, each pair of inputs is assigned a pair of NMOS transistors stacked in series to create the pulldown logic. As there are 16 total combinations and each combination has 2 inputs, the result initially requires 16 transistors stacked across 2 layers. However, because each combinational input of A and B is unique, either A or B can be shared in the second layer of pulldowns for further area savings. Figure 2.1 provides a completed set of output rails with A[3:0] chosen as the shared set of inputs. Array notation is used to maintain an efficient use

Table 2.1: Vertical XOR Truth Table

Binary Logic						1of4 Logic		
B		A		Result		B	A	Result
0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	1	1
0	0	1	0	1	0	0	2	2
0	0	1	1	1	1	0	3	3
0	1	0	0	0	1	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	1	1	1	2	3
0	1	1	1	1	0	1	3	2
1	0	0	0	1	0	2	0	2
1	0	0	1	1	1	2	1	3
1	0	1	0	0	0	2	2	0
1	0	1	1	0	1	2	3	1
1	1	0	0	1	1	3	0	3
1	1	0	1	1	0	3	1	2
1	1	1	0	0	1	3	2	1
1	1	1	1	0	0	3	3	0

Table 2.2: Vertical XOR 1of4 Pulldown Computation

0 Rail		1 Rail		2 Rail		3 Rail	
B	A	B	A	B	A	B	A
0	0	0	1	0	2	0	3
1	1	1	0	1	3	1	2
2	2	2	3	2	0	2	1
3	3	3	2	3	1	3	0

of drafting space; [17] provides a comprehensive overview of schematic array notation under the connectivity and naming conventions guide. When implementing the cell, the left valid computation can be performed with two 4-input NOR gates fed into an inverted C-element, while the right valid computation is implemented using a 4-input NAND gate attached to $\overline{r[3:0]}$. Depending on whether a half-buffer or full-buffer implementation is preferred, one can follow the usage of the inverter and C-elements provided in Figures 1.2 or 1.3 to complete the cell.

Figure 2.2 depicts the pulldown rails for use in a 3:2 compressor. The compressor sums the number of 1's present in the 1ofN codes' binary representation. The process for developing the pulldown rails follows the same procedure as the vertical XOR cell. Dissimilar to binary logic however, this compressor is not synonymous with a 3:2 Full Adder (FA), as an e1ofN adder would sum the values represented by the 1ofN codes, rather than the number of 1's present in the binary representation. Therefore in addition to requiring a unique set of 1of4 pulldown rails, an e1ofN

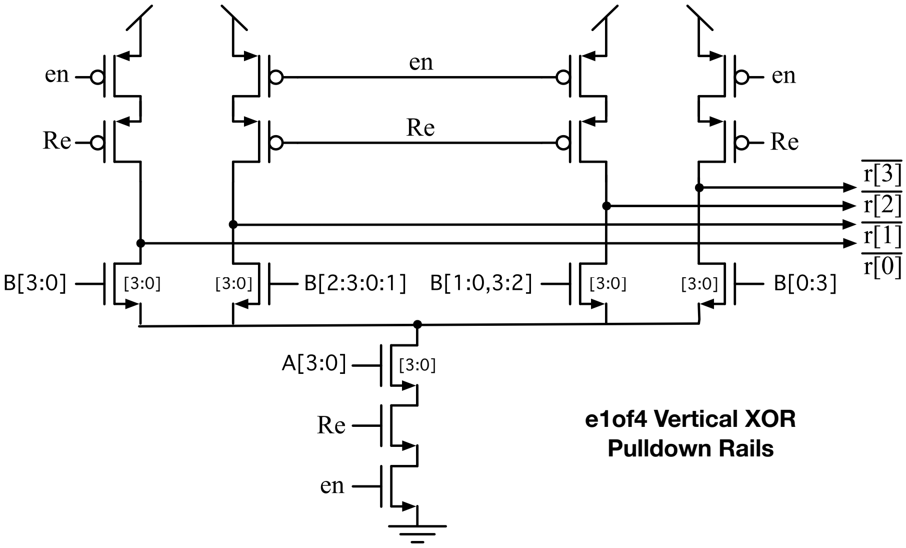


Figure 2.1: Detailed implementation of the output rails for an e1of4 Vertical XOR
Note that B[3:0] inherits a higher input capacitance than A[3:0].

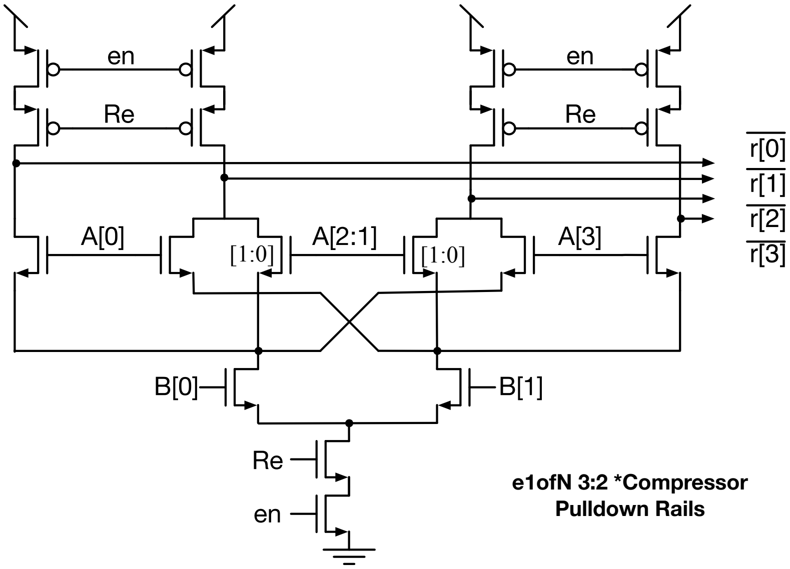


Figure 2.2: Detailed implementation of the output rails for an e1ofN Compressor.
In this case, compressor is used to mean a device that sums the number of 1's present in the 1ofN's binary representation.

adder also effectively demands a unique set of e1of2 pulldown rails to represent the maximum value of 3+1 = 4.

In addition to methods for implementing logical functions, another important aspect of digital design is the implementation of Finite State Machines (FSM)'s. FSM's maintain a given state and only change after all expected inputs have become available [7]. One method for doing so would be to build two separate sets of logic rails - one set provides the next-state logic while the other provides output logic. The output of the next state logic is then buffered and fed as an input into the output logic along with the main inputs to the FSM. The simplicity of the method enables synchronous techniques to be leveraged for generating the boolean state and next state logic, as the synchronous data flops can be easily swapped out for pipelined buffers [7]. However, as the state machine becomes large, the area efficiency of the technique begins to depreciate, inevitably motivating the use of state-holding circuits, rather than maintaining the current state by looping the next-state logic back via pipelined buffers. [16] & [7] provide one method for implementing QDI state-bits, while Figure 2.3 depicts a version with increased modularity.

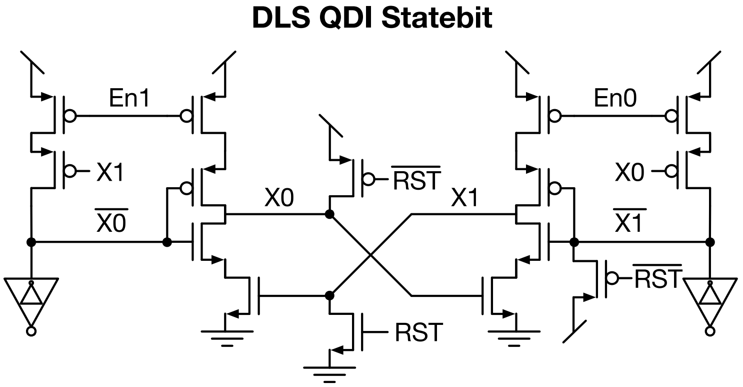


Figure 2.3: Davies, Lines, & Southworth QDI State-bit Template.
An alternate implementation of [16][7].

During the reset, both enable inputs are '0', and *X0* is pulled high while *X1* is pulled low, effectively setting the default state of the bit to '0'. Ideally, this would allow both $\overline{X0}$ and $\overline{X1}$ to pre-charge, but because *X0* is initialized to logic '1', $\overline{X1}$ necessitates the use of an extra transistor to pre-charge during the circuit reset. Alternatively, the three reset transistors can be mirrored to set the default state to '1' instead of defaulting to '0'. Once the reset has completed, $\overline{X0}$ & $\overline{X1}$ can approached similarly to a set of pre-charged output rails: each time the required set of inputs becomes valid, either $\overline{X0}$ or $\overline{X1}$ will be pulled low to indicate a valid transition has occurred, even if the transition does not require the currently held state to change. The transition on either of the rails is then detected using a state-validity circuit and subsequently combined with the other valid signals according to a half-buffer or full-buffer implementation. After the valid signals have been aggregated and the internal enable is lowered, the state-bit will subsequently change states if the corresponding $\overline{X}$ rail has lowered. For example, if the state-bit holds a '0', *X0* is initialized to '1'. When $\overline{X1}$ is pulled down and the enable is lowered, *X1* will be pre-charged to '1' and pull *X0* down to '0' subsequently allowing $\overline{X1}$ to pre-charge to the default state of '1'. If $\overline{X0}$ had been pulled down instead of $\overline{X1}$, when the enable is lowered the state is maintained and $\overline{X0}$ simply pre-charges back to the original state.

Table 2.3: 2-Bit Counter State & Pulldown Rail Table

Counter State Table				**State[1] - 1 Rail**		**State[1] - 0 Rail**	
State[1:0]		***State[1:0]**		0	1	0	0
0	0	0	1	1	0	1	1
0	1	1	0	**State[0] - 1 Rail**		**State[0] - 0 Rail**	
1	0	1	1	0	0	0	1
1	1	0	0	1	0	1	1

Table 2 provides a state table and pulldown logic for each of the state-bit rails for use in a 2-bit QDI counter, while Figure 2.4 provides the implementation of the pulldown rails and state-validity. Also note the use of 2-dimensional array notation that follows the form **S[statebit][rail]**. The state-bits could further be combined into a signal array but have been drawn separately for clarity. One method for detecting the validity of the state-bit transitions would be to use two 2-input NAND gates fed into a C-element. However, [5] provides a superior method of detecting the validity, which is also used in the weak-conditioned full adder [11][18].

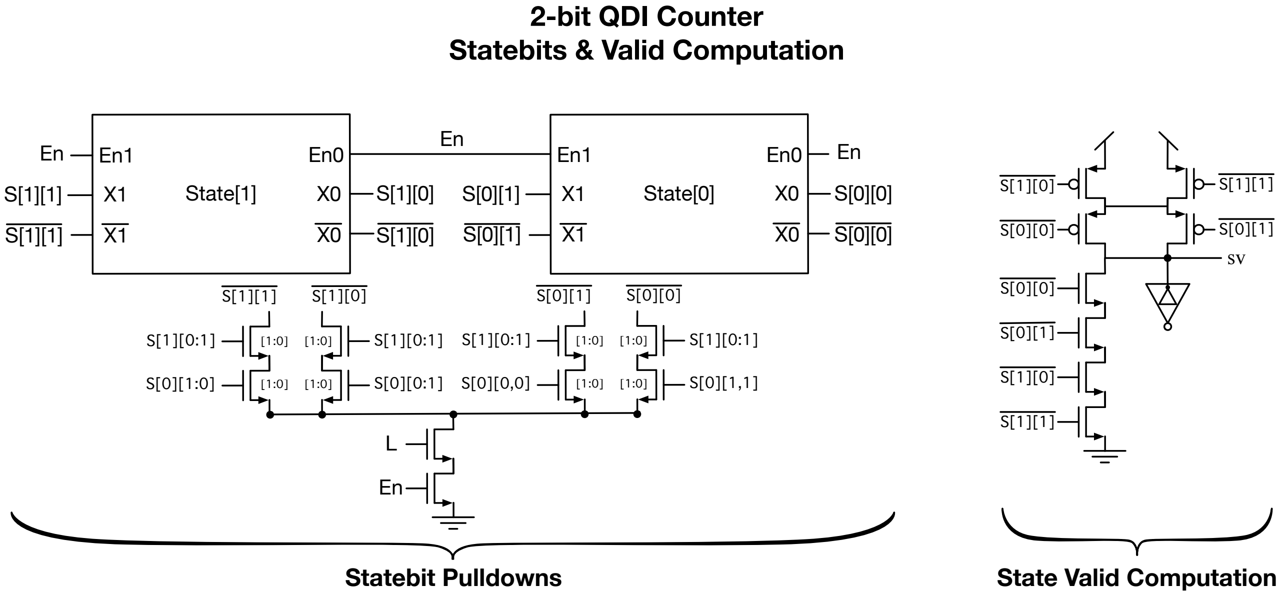


Figure 2.4: Statebit pulldowns and state validity for use in a 2-bit QDI counter. The state validity performs the equivalent of 2 NAND gates fed into a C-element [5][11][18].

Although the state-bit's separate enables remain tied together in the counter, their separation is of use in specific instances. First, consider the 4-way PCFB e1of4 Split in Figure 2.5. The split conditionally writes an e1of4 code onto one of four channels, annotated using 2D array notation as **R[channel][rail]**. Each of the 4-channels is provided with it's own set of pulldowns, only one of which will evaluate depending on the select line that is asserted. Regardless of the channel the data is written to however, S[3:0] and L[3:0] are acknowledged with a single wire, as they are both always acknowledged unconditionally. Next, consider the 2-way interleaved Split depicted in Figure 2.6. The output of a single state-bit is used to replace the S[1:0] inputs. When left data arrives, the channel currently selected by the state-bit receives the data, the opposite state rail is pulled low, and the corresponding *se* is lowered allowing the state-bit to flip. As *se* cannot be raised again until the bit has successfully flipped, it eliminates the need for an additional state-valid computation. The additional reset transistor is required to ensure se[0] successfully pre-charges.

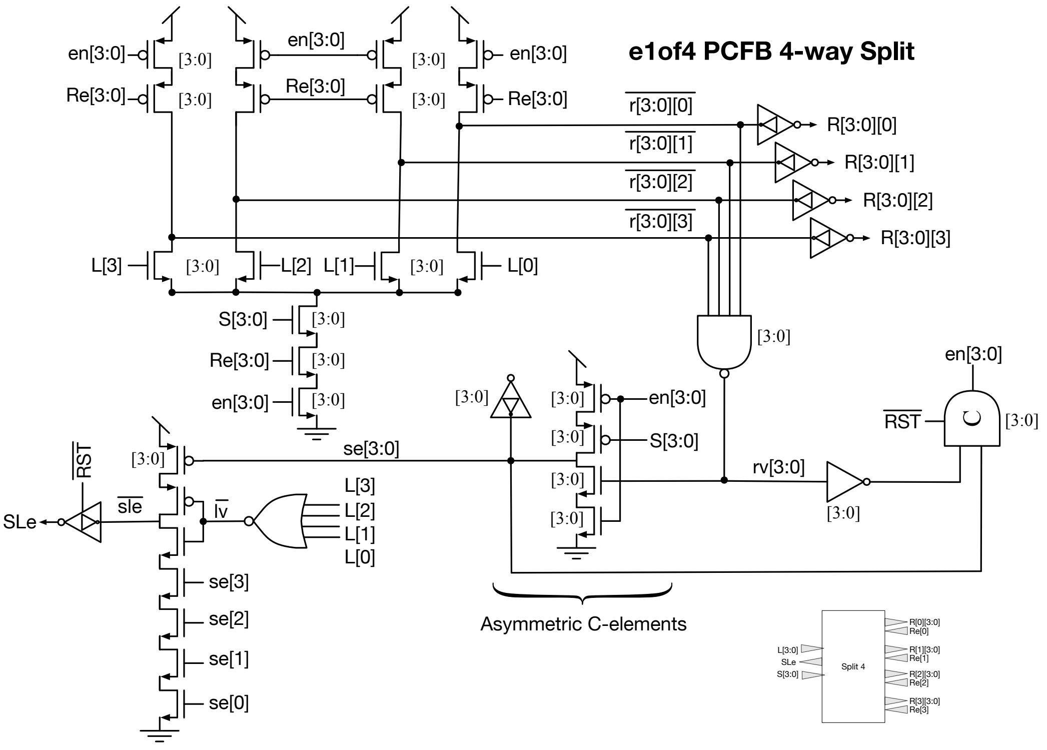


Figure 2.5: Detailed implementation of an e1of4 4-way PCFB Split.
An extension of [19].

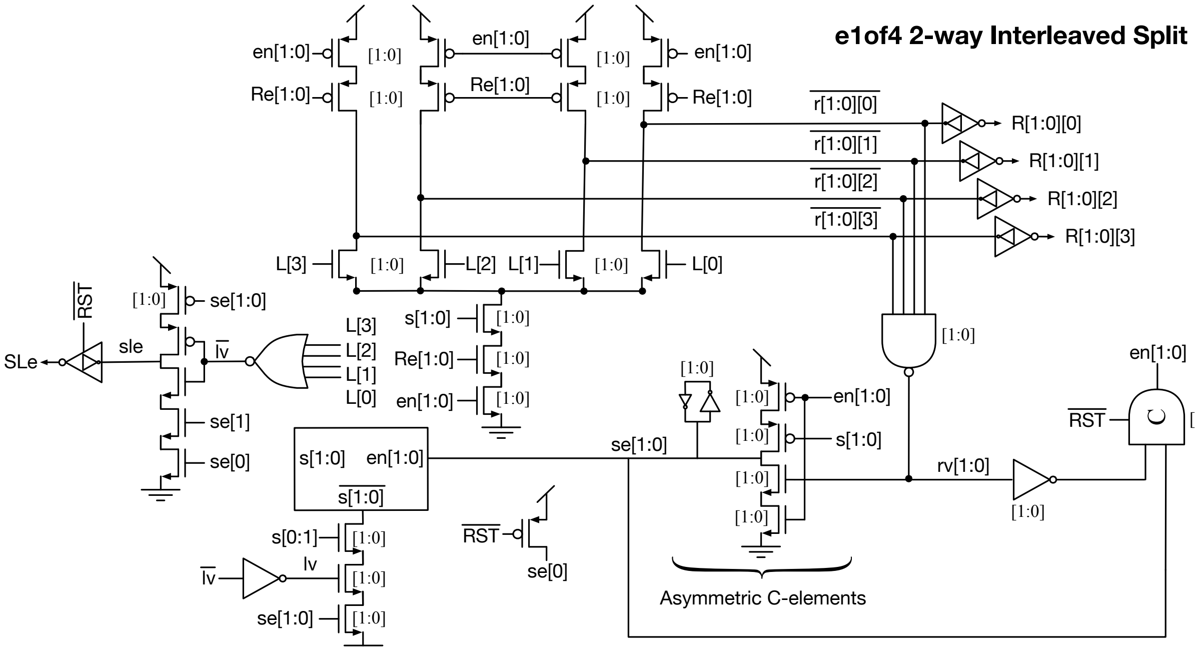


Figure 2.6: Detailed implementation of an e1of4 2-way interleaved PCFB Split.
Extensions of [16][19] using a single DLS QDI state-bit.

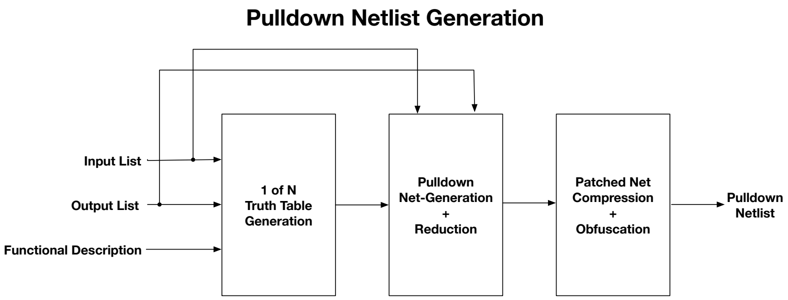

Pulldown Netlist Generation
Input List
Output List
Functional Description
1 of N Truth Table Generation
Pulldown Net-Generation + Reduction
Patched Net Compression + Obfuscation
Pulldown Netlist

Figure 2.7: High level view of a pulldown netlist generator.

In all cases encountered thus far, transistors in the pulldown logic have been drawn using wired connections. However in some cases, connections cannot always be drawn easily. First, consider the e1ofN Carry-Save Adder (CSA) carry-sum rails depicted in Figure 2.8. The carry save adder adds the value of two e1of4 codes + an e1of2 code. Carry-save addition is commonly used in high-performance Booth multiplication to quickly sum the results of partial product computations - [20] leverages an alternate Single-Track Full Buffer (STFB) implementation. As the maximum value produced by the CSA is 3+3+1=7, it necessitates the use of an additional set of e1of2 carry-out rails (not shown). In Figure 2.8, the carry-sum operation has been broken into 2-sages, and as such, the connections can be drawn in a similar fashion to the previous methods. The 2-stage CSA may be of use when the number of transistors in the pulldown logic is limited due to NTV operation, as [2] does not utilize implementations with more than 4 NMOS transistors in series. Next, consider the single-stage CSA shown in 2.9. In order to effectively illustrate the connections, a set of *patched nets* is used. Each of the nets represents 4 wired connections. Finally, consider the method for pulldown netlist generation shown in Figure 2.7. Although an automated tool for constructing a functional pulldown netlist is beyond the scope of this thesis, once cell implementation has been thoroughly understood, the process of automating logic for standard cell generation is the next step in constructing a unique toolset (if alternative methods are unavailable). As a result, one may choose to develop a program to automate pulldown netlist generation, rather than designing cells by hand.

While this chapter has aimed to educate the reader on techniques for designing e1ofN cells, there are still many additional topics that remain uncovered, such as, pipelined arbitration, pipelined completion, weak-conditioned logic, reduced stack pre-charged logic, single-track full

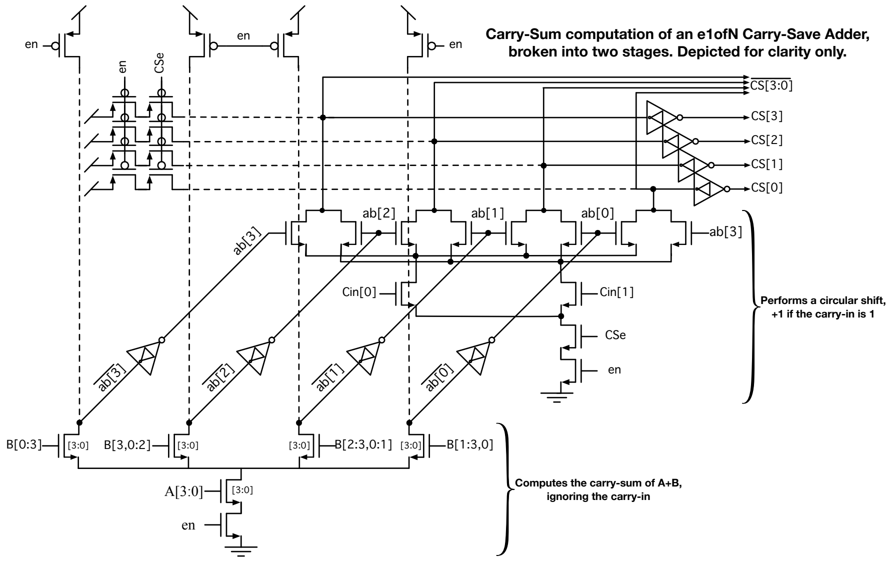


Figure 2.8: Detailed implementation of carry-sum rails in a 2-stage CSA.

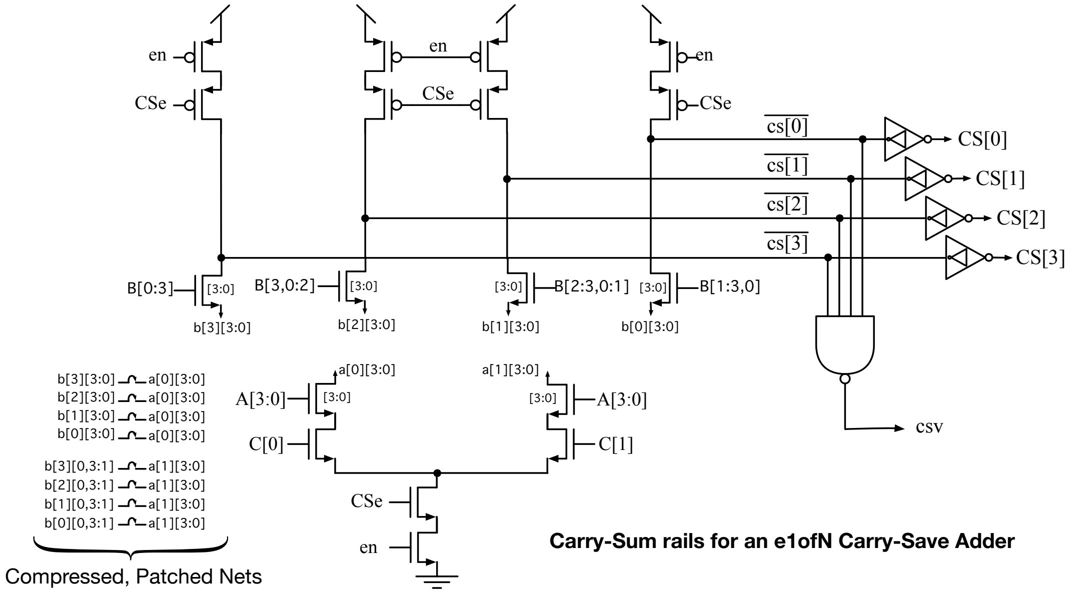
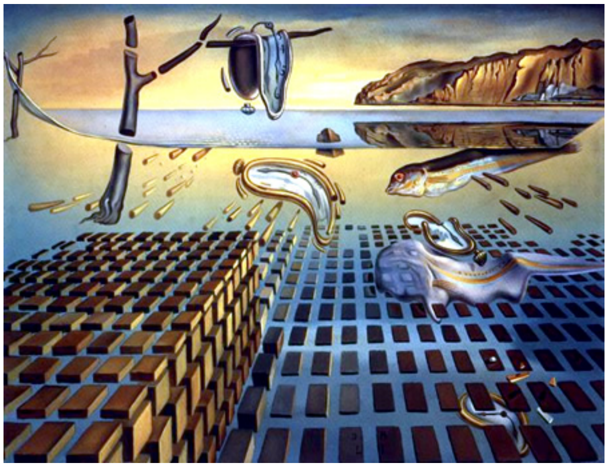


Figure 2.9: Detailed implementation of carry-sum computation in a single-stage CSA.

buffered logic, and sense-amp half buffered logic[21]. [7] is an excellent resource, as it provides a exhaustive overview of nearly all industrially-utilized asynchronous techniques, while [16] and [19] provide comprehensive implementation examples, in addition to other works output by pioneers in the field of asynchronous design.

3

Memory Design, Pipelining, & Energy Estimation



Salvador Dali, *Disintegration of the Persistence of Memory*

One element common to all computational systems is memory. In the majority of modern CPU's, memory exists as a hierarchy composed of several layers, with each layer increasing in size and consequently decreasing in speed [5]. This *cache hierarchy* occupies a significant fraction of the overall die size and thus may consume up to 50% of the relative dynamic power [22][23]. Moreover, as technology scaling nears the end of initial ascent to cruising altitude at the fundamental atomic limit, and improvements gained as a result of Moore's law draw to a close, alternate methods for scaling supply voltage must be sought out.

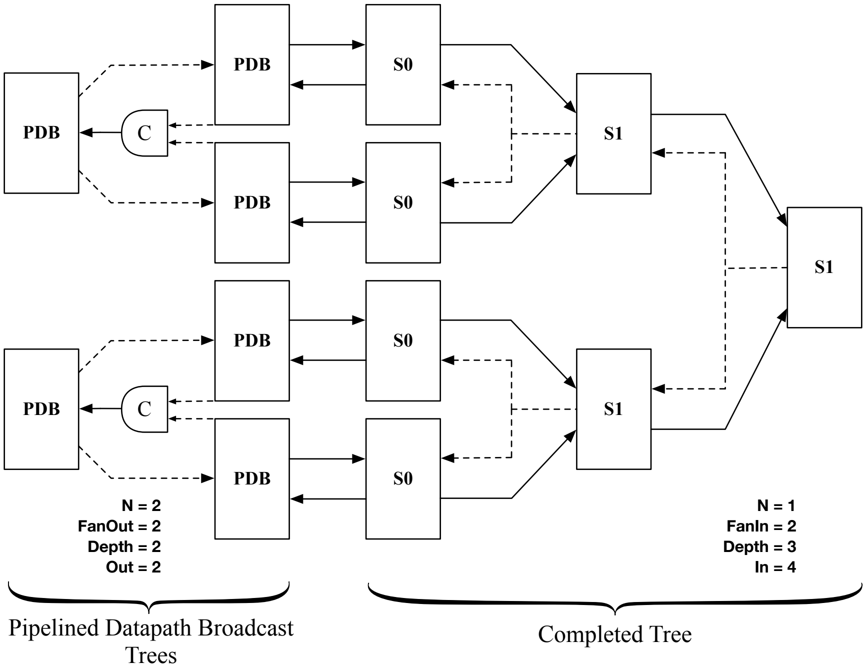


Figure 3.2: Two Broadcast / Copy Trees communicating with a single Completed Tree.

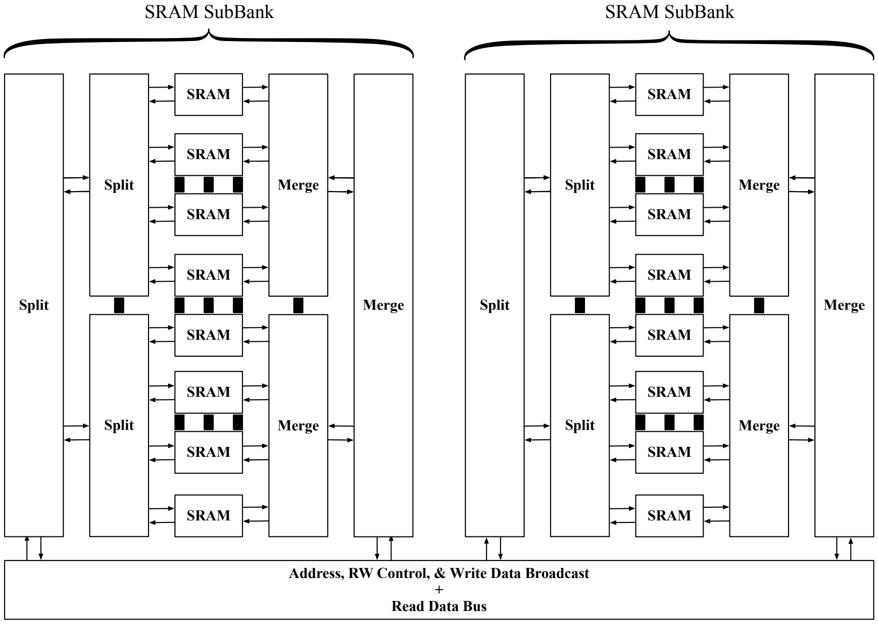
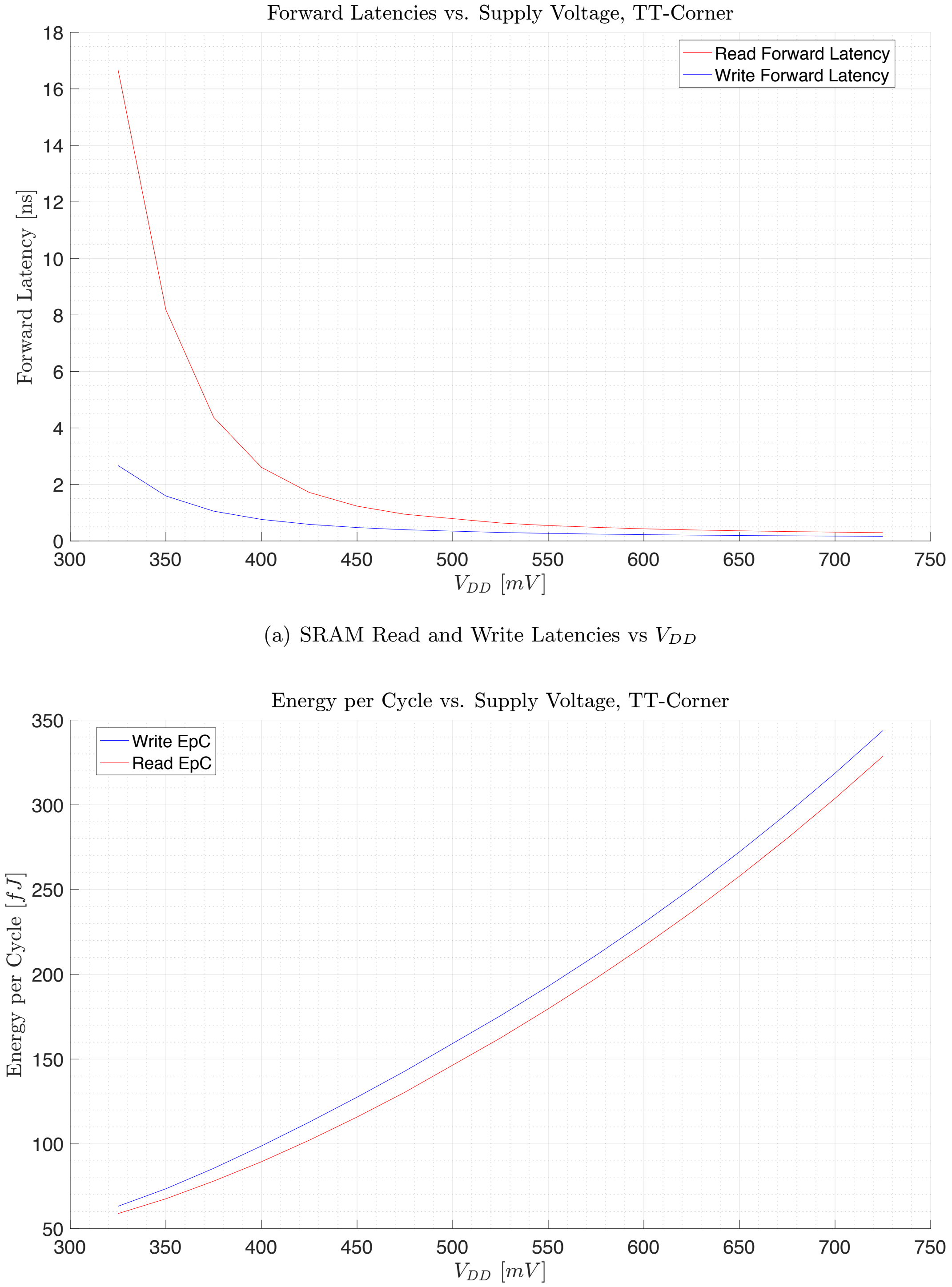


Figure 3.3: A Pipelined Split-Merge Tree utilized during SRAM access.

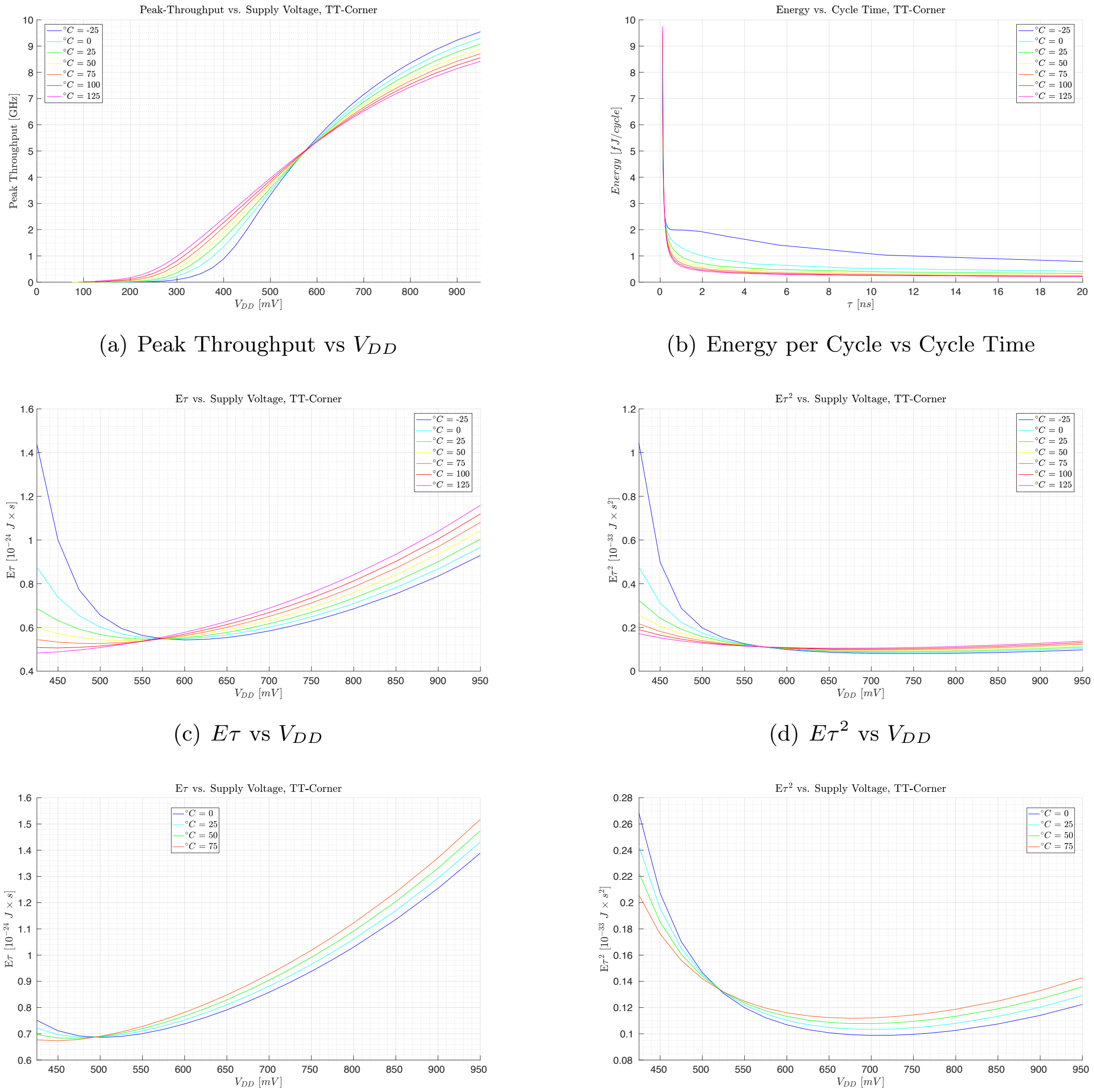
In order to to create wide SRAM subarrays, SRAM subarrays, tiles of width 12 and 16 bits are stacked vertically. Using these two widths, any width of 4 bits may be supported, except for widths of 4, 8 and 20. Commercial asynchronous systems have used this method in the past for purposes of minimizing the number of physical VLSI templates and the effective time to market



(a) SRAM Read and Write Latencies vs V_{DD}

(b) SRAM Read and Write Energy per Cycle vs V_{DD}

Figure 4.1: 1024 x 16 bit Pipelined SRAM Results



(a) Peak Throughput vs V_{DD}

(b) Energy per Cycle vs Cycle Time

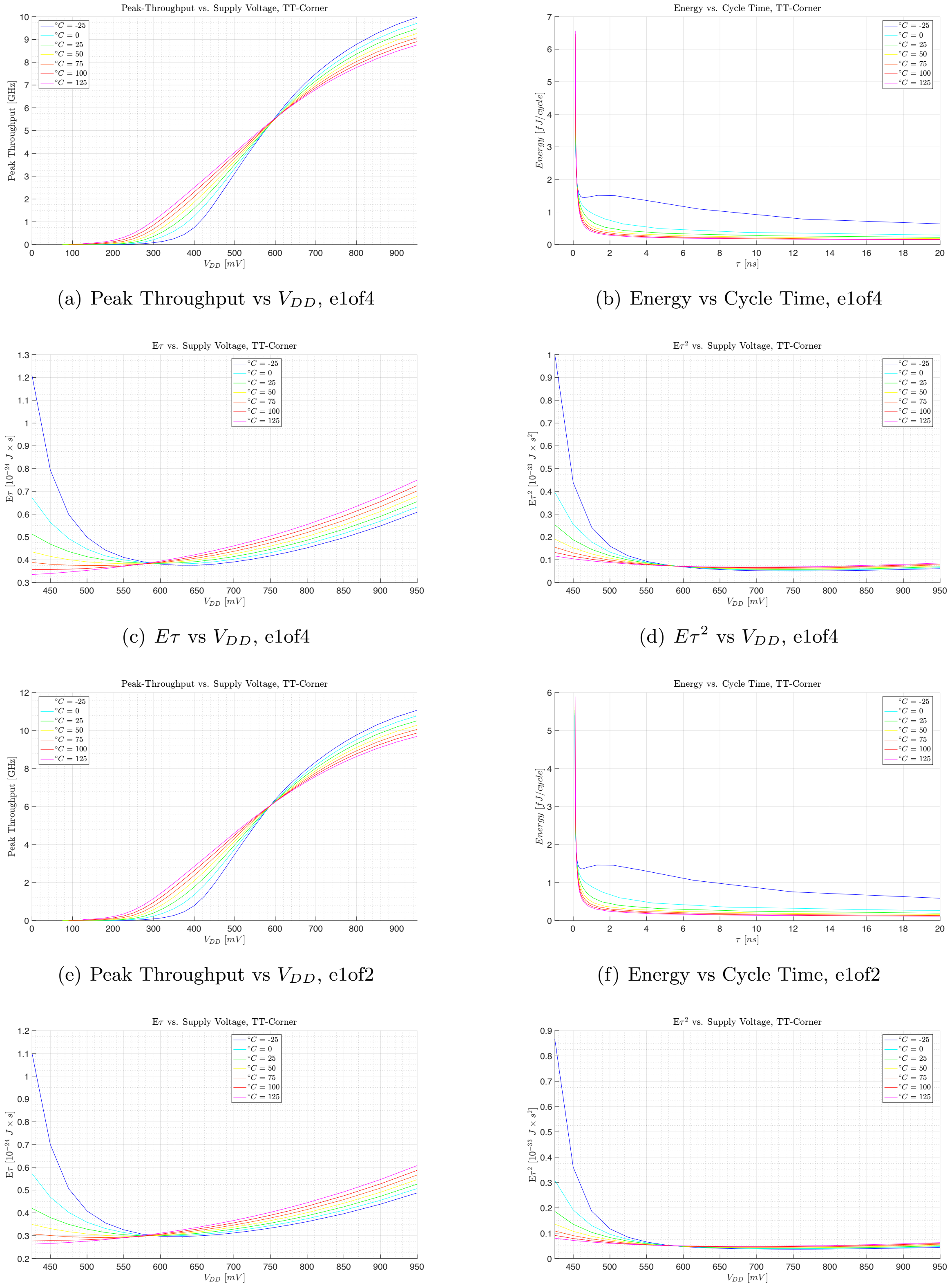
(c) $E\tau$ vs V_{DD}

(d) $E\tau^2$ vs V_{DD}

(e) $E\tau$ vs V_{DD} 3x

(f) $E\tau^2$ vs V_{DD} 3x

Figure 4.3: e1of4 PCFB 7nm TT-Corner Results
$E\tau$ and $E\tau^2$ are also shown for the same buffer, with all transistors sized 3x larger



(a) Peak Throughput vs V_{DD}, e1of4

(b) Energy vs Cycle Time, e1of4

(c) $E\tau$ vs V_{DD}, e1of4

(d) $E\tau^2$ vs V_{DD}, e1of4

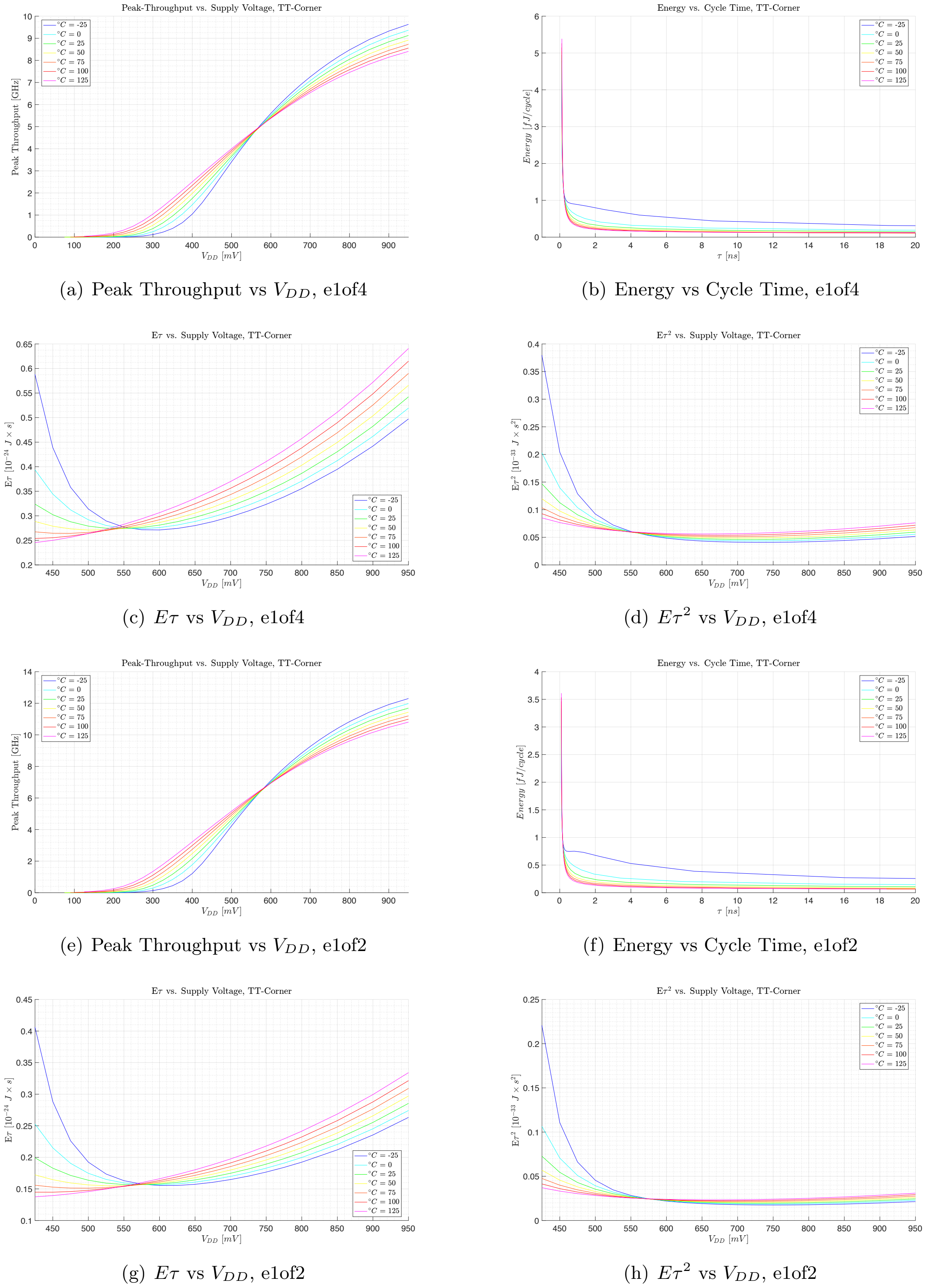
(e) Peak Throughput vs V_{DD}, e1of2

(f) Energy vs Cycle Time, e1of2

(g) $E\tau$ vs V_{DD}, e1of2

(h) $E\tau^2$ vs V_{DD}, e1of2

Figure 4.4: e1of4 & e1of2 PCHB 7nm TT-Corner Results



(a) Peak Throughput vs V_{DD}, e1of4

(b) Energy vs Cycle Time, e1of4

(c) $E\tau$ vs V_{DD}, e1of4

(d) $E\tau^2$ vs V_{DD}, e1of4

(e) Peak Throughput vs V_{DD}, e1of2

(f) Energy vs Cycle Time, e1of2

(g) $E\tau$ vs V_{DD}, e1of2

(h) $E\tau^2$ vs V_{DD}, e1of2

Figure 4.5: e1of4 & e1of2 WCHB 7nm TT-Corner Results

References

[1] Alain J Martin. 25 Years Ago: The First Asynchronous Microprocessor. 2014.

[2] S. Keller, A. J. Martin, and C. Moore. DD1: A QDI, Radiation-Hard-by-Design, Near-Threshold 18uW/MIPS Microcontroller in 40nm Bulk CMOS. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 37–44, May 2015.

[3] Sean Keller. *Robust Near-Threshold QDI Circuit Analysis and Design.* PhD thesis, 2013.

[4] Alain J Martin and Mika Nyström. Asynchronous techniques for system-on-chip design. *Proceedings of the IEEE*, 94(6):1089–1120, 2006.

[5] Mika Nyström. A Pipelined Asynchronous Cache System. Master's thesis, 1997.

[6] Wonjin Jang. Soft-error Tolerant QDI Circuits, 2008.

[7] Peter A. Beerel, Recep O. Ozdag, and Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[8] Alain J Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. 1989.

[9] Rajit Mohnar & Alain J. Martin. Quasi delay insensitive circuits are Turing complete. 1995.

[10] S. Keller, M. Katelman, and A. J. Martin. A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits. In *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, pages 65–76, 2009.

[11] Andrew M Lines. Pipelined Asynchronous Circuits. Technical report, Pasadena, CA, USA, 1998.

[12] Mika Nyström, Andrew Lines, and Alain J Martin. A Pipelined Asynchronous Cache System.

[13] Jonathan Dama and Andrew Lines. GHz asynchronous SRAM in 65nm. In *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on*, pages 85–94. IEEE, 2009.

[14] P. L. Penzes and A. J. Martin. An energy estimation method for asynchronous circuits with application to an asynchronous microprocessor. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 640–647, 2002.

[15] Xiaofei Chang. Resetting Asynchronous QDI systems. Master's thesis, 2014.

[16] M.I. Davies, A. Lines, and R. Southworth. Techniques for facilitating conversion between asynchronous and synchronous domains, November 1 2005. US Patent 6,961,863.

[17] Cadence Design Systems Inc. Icadv 12.3 connectivity and naming conventions guide. Technical report, 2017.

[18] U. V. Cummings, A. M. Lines, and A. J. Martin. An asynchronous pipelined lattice structure filter. In *Proceedings of 1994 IEEE Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, Nov 1994.

[19] U. Cummings and A. Lines. Asynchronous crossbar with deterministic or arbitrated control, September 25 2007. US Patent 7,274,709.

[20] B. R. Sheikh and R. Manohar. An Asynchronous Floating-Point Multiplier. In *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, pages 89–96, May 2012.

[21] K. S. Chong, W. G. Ho, T. Lin, B. H. Gwee, and J. S. Chang. Sense Amplifier Half-Buffer (SAHB) A Low-Power High-Performance Asynchronous Logic QDI Cell Template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):402–415, Feb 2017.

[22] S. Kaxiras, Zhigang Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 240–251, 2001.

[23] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 148–157, 2002.

[24] V. De. Energy efficient computing in nanoscale CMOS: Challenges and opportunities. In *Solid-State Circuits Conference (A-SSCC), 2014 IEEE Asian*, pages 121–124, Nov 2014.

[25] Wonjin Jang and Alain J Martin. Soft-error robustness in QDI circuits. 2005.

[26] Sean Jason Keller. *Robust near-threshold QDI circuit analysis and designs*. PhD thesis, Citeseer, 2014.

[27] Brian Zimmer. Resilient Design Methodology for Energy-Efficient SRAM. Master's thesis, University of California, Berkeley, May 2012.

[28] Brian Zimmer and Borivoje Nikolic. *Resilient Design Methodology for Energy-Efficient SRAM*. PhD thesis, Citeseer, 2013.

[29] L. Dolecek, M. Qazi, D. Shah, and A. Chandrakasan. Breaking the simulation barrier: SRAM evaluation through norm minimization. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 322–329, Nov 2008.

[30] J. Teifel, D. Fang, D. Biermann, C. Kelly, and R. Manohar. Energy-efficient pipelines. In *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*, pages 23–33, April 2002.

[31] Steven Morgan Burns. *Performance analysis and Optimization of Asynchronous Circuits*. PhD thesis, 1991.

[32] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, HP Laboratories, Palo Alto, CA, USA, Apr. 2009.

[33] C. Y. Lee and N. K. Jha. CACTI-FinFET: An integrated delay and power modeling framework for FinFET-based caches under process variations. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 866–871, June 2011.

[34] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105 – 115, 2016.

[35] R. Amirtharajah M. Farrens V. Akella R. St. Denis, C. Nitta. cdsAsync: An Asynchronous VLSI Toolset & Schematic Library. Technical report, 2017.