Designing Asynchronous Circuits using NULL Convention Logic (NCL)

Synthesis Lectures on Digital Circuits and Systems

Atmel AVR Microcontroller Primer: Programming and Interfacing
Steven F. Barrett, Daniel J. Pack
2007

Pragmatic Logic
William J. Eccles
2007

PSpice for Filters and Transmission Lines
Paul Tobin
2007

PSpice for Digital Signal Processing
Paul Tobin
2007

PSpice for Analog Communications Engineering
Paul Tobin
2007

PSpice for Digital Communications Engineering
Paul Tobin
2007

PSpice for Circuit Theory and Electronic Devices
Paul Tobin
2007

Pragmatic Circuits: DC and Time Domain
William J. Eccles
2006

Pragmatic Circuits: Frequency Domain
William J. Eccles
2006

Pragmatic Circuits: Signals and Filters
William J. Eccles
2006

High-Speed Digital System Design
Justin Davis
2006

Introduction to Logic Synthesis using Verilog HDL
Robert B.Reese, Mitchell A.Thornton
2006

Microcontrollers Fundamentals for Engineers and Scientists
Steven F. Barrett, Daniel J. Pack
2006

Designing Asynchronous Circuits using NULL Convention Logic (NCL)

Scott C. Smith and Jia Di

www.morganclaypool.com

ISBN: 9781598299816 paperback
ISBN: 9781598299823 ebook

DOI 10.2200/S00202ED1V01Y200907DCS023

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Lecture #23
Series Editor: Mitchell A. Thornton, *Southern Methodist University*

Series ISSN
Synthesis Lectures on Digital Circuits and Systems
Print 1930-1743 Electronic 1930-1751

Designing Asynchronous Circuits using NULL Convention Logic (NCL)

Scott C. Smith and Jia Di
University of Arkansas

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #23

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Designing Asynchronous Circuits using NULL Convention Logic (NCL) begins with an introduction to asynchronous (clockless) logic in general, and then focuses on delay-insensitive asynchronous logic design using the NCL paradigm. The book details design of input-complete and observable dual-rail and quad-rail combinational circuits, and then discusses implementation of sequential circuits, which require datapath feedback. Next, throughput optimization techniques are presented, including pipelining, embedding registration, early completion, and NULL cycle reduction. Subsequently, low-power design techniques, such as wavefront steering and Multi-Threshold CMOS (MTCMOS) for NCL, are discussed. The book culminates with a comprehensive design example of an optimized Greatest Common Divisor circuit.

Readers should have prior knowledge of basic logic design concepts, such as Boolean algebra and Karnaugh maps. After studying this book, readers should have a good understanding of the differences between asynchronous and synchronous circuits, and should be able to design arbitrary NCL circuits, optimized for area, throughput, and power.

KEYWORDS

computer engineering, digital design, asynchronous logic, delay-insensitive logic, combinational logic, sequential logic, NULL Convention Logic, NCL, input-completeness, observability, dual-rail, quad-rail, pipelining, embedded registration, early completion, NULL cycle reduction, wavefront steering, MTCMOS

Contents

CHAPTER 1

Introduction to Asynchronous Logic

For the last three decades, the focus of digital design has been primarily on synchronous, clocked architectures. However, as clock rates have significantly increased while feature size has decreased, clock skew has become a major problem. High performance chips must dedicate increasingly larger portions of their area for clock drivers to achieve acceptable skew, causing these chips to dissipate increasingly higher power, especially at the clock edge, when switching is most prevalent. As these trends continue, the clock is becoming more and more difficult to manage, while clocked circuits' inherent power inefficiencies are emerging as the dominant factor hindering increased performance. These issues have caused renewed interest in asynchronous digital design. Asynchronous, clockless circuits require less power, generate less noise, and produce less electro-magnetic interference (EMI), compared to their synchronous counterparts, without degrading performance. Furthermore, ***delay-insensitive (DI)*** asynchronous paradigms have a number of additional advantages, especially when designing complex circuits, like Systems-on-a-Chip (SoCs), including substantially reduced crosstalk between analog and digital circuits, ease of integrating multi-rate circuits, and facilitation of component reuse. Asynchronous circuits can even utilize a synchronous wrapper, such that the end user does not know that the internal circuitry is actually asynchronous in nature. Currently, companies such as ARM, Phillips, Intel, and others are incorporating asynchronous logic into some of their products using their own proprietary tools.

As demand increases for designs with higher performance, greater complexity, and decreased feature size, asynchronous paradigms will become more prevalent in the multi-billion dollar semiconductor industry, as predicted by the International Technology Roadmap for Semiconductors (ITRS), which envisions a likely shift from synchronous to asynchronous design styles in order to increase circuit robustness, decrease power, and alleviate many clock-related issues. ITRS shows that asynchronous circuits accounted for 11% of chip area in 2008, compared to 7% in 2007, and estimates they will account for 23% of chip area by 2014, and 35% of chip area by 2019.

Asynchronous circuits can be grouped into two main categories: *bounded-delay* and *delay-insensitive* models. Bounded-delay models, such as *micropipelines* [1], assume that delays in both gates and wires are bounded. Delays are added based on worse-case scenarios to avoid hazard conditions. This leads to extensive timing analysis of worse-case behavior to ensure correct circuit operation. On the other hand, delay-insensitive circuits assume delays in both logic elements and interconnects to be unbounded. Although they assume that wire forks within basic components, such as a full adder, are isochronic. This means that the wire delays within a component are much

less than the logic element delays within the component, which is a valid assumption even in future nanometer technologies. Wires connecting components do not have to adhere to the isochronic fork assumption. This implies the ability to operate in the presence of indefinite arrival times for the reception of inputs. Completion detection of the output signals allows for handshaking to control input wavefronts. Delay-insensitive design styles, therefore, require very little, if any, timing analysis to ensure correct operation (i.e., they are correct-by-construction), and also yield average-case performance rather than the worse-case performance of bounded-delay and traditional synchronous paradigms.

Most delay-insensitive methods combine *C-elements* with Boolean gates for circuit construction. A C-element behaves as follows: when all inputs assume the same value then the output assumes this value, otherwise the output does not change. Seitz's [2], DIMS [3], Anantharaman's [4], Singh's [5], and David's [6] methods are examples of DI paradigms that only use C-elements to achieve delay-insensitivity. On the other hand, both Phased Logic [7] and NULL Convention Logic (NCL) [8] target a library of multiple gates with *hysteresis* state-holding functionality. Phased Logic converts a traditional synchronous gate-level circuit into a delay-insensitive circuit by replacing each conventional synchronous gate with its corresponding Phased Logic gate, and then augmenting the new network with additional signals. NCL circuits are realized using 27 fundamental gates implementing the set of all functions of four or fewer variables, each with *hysteresis* state-holding functionality.

Seitz's method, Anantharaman's approach, and DIMS require the generation of all minterms to implement a function, where a minterm is defined as the logical AND, or product, containing all input signals in either complemented or non-complemented form. While Singh's and David's methods do not require full minterm generation, they rely solely on C-elements for speed-independence. NCL also does not require full minterm generation and, furthermore, includes 27 fundamental state-holding gates for circuit design, rather than only C-elements, thus yielding a greater potential for optimization than other delay-insensitive paradigms. Phased Logic also does not require full minterm generation and does not rely solely on C-elements for speed-independence; however, Phased Logic circuitry is derived directly from its equivalent synchronous design, not created independently, thus it does not have the same potential for optimization as does NCL. Furthermore, the Phased Logic paradigm has been developed mainly for easing the timing constraints of synchronous designs, not for obtaining speed and power benefits, whereas these are main concerns of other asynchronous paradigms.

Self-timed circuits can also be designed at the transistor level as demonstrated by Martin [9]. However, automation of this method would be vastly different than that of the standard synchronous approach, since it optimizes designs at the transistor level instead of targeting a predefined set of gates, as do the previously mentioned methods. Overall, NULL Convention Logic offers the best opportunity for integrating asynchronous digital design into the predominantly synchronous semiconductor design industry for the following reasons:

1) The framework for NCL systems consists of DI combinational logic sandwiched between DI registers, which is very similar to synchronous systems, such that the automated design of NCL circuits can follow the same fundamental steps as synchronous circuit design automation. This will enable the developed DI design flow to be more easily incorporated into the chip design industry, since the tools and design process will already be familiar to designers, such that the learning curve is relatively flat.

2) NCL systems are delay-insensitive, making the design process much easier to automate than other non-DI asynchronous paradigms, since minimal delay analysis is necessary to ensure correct circuit operation.

3) NCL systems have power, noise, and EMI advantages compared to synchronous circuits, performance and design reuse advantages compared to synchronous and non-DI asynchronous paradigms, area and performance advantages compared to other DI paradigms, and have a number of advantages for designing complex systems, like SoCs, including substantially reduced crosstalk between analog and digital circuits, ease of integrating multi-rate circuits, and facilitation of component reuse and technology migration.

As the trend towards higher clock frequency and smaller feature size continues, power consumption, noise, and EMI of synchronous designs increase significantly. With the absence of a clock, DI systems aim to reduce power consumption, noise, and EMI. DI circuits designed using CMOS exhibit an inherent idle behavior since they only switch when useful work is being performed, unlike clocked Boolean circuits that switch every clock pulse, unless specifically disabled through specialized circuitry, which itself requires additional area and power. DI circuits adhere to monotonic transitions between DATA and NULL, so there is no glitching, unlike clocked Boolean circuits that produce substantial glitch power. DI systems better distribute switching over time and area, reducing the occurrence of hot spots, peak power demand, and system noise, unlike clocked Boolean circuits where much of the circuitry switches simultaneously at the clock edge. Furthermore, DI systems are very tolerant of power supply variations, allowing cheaper power supplies to be used and voltage to be dramatically reduced to meet desired performance while decreasing power consumption. Therefore, a very fast DI circuit can be run at a lower voltage to reduce power consumption when high performance is not required. Other DI advantages include tolerance of vast temperature differences, making these circuits well suited for operation in harsh environments, like outer space, and easing the difficulty of integrating designs with non-harmonically related clock frequencies. Their main disadvantage is increased area, which is approximately 1.5 – 2 times as much as an equivalent synchronous design when using static CMOS gates, but less for semi-static CMOS gates. However, for large designs, such as SoCs, the processor core(s) normally require(s) less than 1/2 of the chip's total area, while the rest of the chip area consists of flash, cache, RAM, peripherals, etc., which are the same in both DI and synchronous implementations. Therefore, the increased area for the DI implementation of the processor core(s) is less significant, especially considering the increased robustness and numerous other advantages.

BIBLIOGRAPHY

[1] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, Vol. 32/6, pp. 720–738, 1989. DOI: 10.1145/63526.63532

[2] C. L. Seitz, "System Timing," in *Introduction to VLSI Systems*, Addison-Wesley, pp. 218–262, 1980.

[3] J. Sparso, J. Staunstrup, M. Dantzer-Sorensen, "Design of Delay Insensitive Circuits using Multi-Ring Structures," *Proceedings of the European Design Automation Conference*, pp. 15–20, 1992. DOI: 10.1109/EURDAC.1992.246271

[4] T. S. Anantharaman, "A Delay Insensitive Regular Expression Recognizer," *IEEE VLSI Technical Bulletin*, Sept. 1986.

[5] N. P. Singh, "A Design Methodology for Self-Timed Systems," *Master's Thesis*, MIT/LCS/TR-258, Laboratory for Computer Science, MIT, 1981.

[6] I. David, R. Ginosar, and M. Yoeli, "An Efficient Implementation of Boolean Functions as Self-Timed Circuits," *IEEE Transactions on Computers*, Vol. 41/1, pp. 2–10, 1992. DOI: 10.1109/12.123377

[7] D. H. Linder and J. H. Harden, "Phased Logic: Supporting the Synchronous Design Paradigm with Delay-Insensitive Circuitry," *IEEE Transactions on Computers*, Vol. 45/9, pp. 1031–1044, 1996. DOI: 10.1109/12.537126

[8] K. M. Fant and S. A. Brandt, "NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis," *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261–273, 1996. DOI: 10.1109/ASAP.1996.542821

[9] A. J. Martin, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits," *Distributed Computing*, Vol. 1/4, pp. 226–234, 1986. DOI: 10.1007/BF01660034

CHAPTER 2

Overview of NULL Convention Logic (NCL)

2.1 NCL SYSTEM FRAMEWORK AND FUNDAMENTAL COMPONENTS

NCL is a ***delay-insensitive (DI)*** asynchronous (i.e., clockless) paradigm, which means that NCL circuits will operate correctly regardless of when circuit inputs become available; therefore, NCL circuits are said to be correct-by-construction (i.e., no timing analysis is necessary for correct operation). NCL circuits utilize dual-rail or quad-rail logic to achieve delay-insensitivity. A dual-rail signal, D, consists of two wires or rails, D^0 and D^1, which may assume any value from the set {DATA0, DATA1, NULL}, as depicted in Table 2.1. The DATA0 state corresponds to a Boolean

Table 2.1: Dual-Rail signal.

	DATA0	DATA1	NULL	Illegal
D^0	1	0	0	1
D^1	0	1	0	1

logic 0, the DATA1 state corresponds to a Boolean logic 1, and the NULL state corresponds to the empty set (meaning that the value of D is not yet available). The two rails are mutually exclusive, such that both rails can never be asserted simultaneously; this state is defined as an illegal state. A quad-rail signal, Q, consists of four wires, Q^0, Q^1, Q^2, and Q^3, which may assume any value from the set {DATA0, DATA1, DATA2, DATA3, NULL}, as depicted in Table 2.2. The DATA0 state corresponds to two Boolean logic signals, X and Y, where $X = 0$ and $Y = 0$; the DATA1 state corresponds to $X = 0$ and $Y = 1$; the DATA2 state corresponds to $X = 1$ and $Y = 0$; the DATA3 state corresponds to $X = 1$ and $Y = 1$; and the NULL state corresponds to the empty set meaning that the result is not yet available. The four rails of a quad-rail NCL signal are mutually exclusive, such that no two rails can ever be asserted simultaneously; these states are defined as illegal states. Both dual-rail and quad-rail signals are space optimal 1-hot delay-insensitive codes, requiring two wires per bit. Other 1-hot encodings may be used for delay-insensitive signaling; however, these may not be space optimal (e.g., an 8-rail MEAG (Mutually Exclusive Assertion Group) can be used to represent 3 bits, but requires 2.67 wires per bit).

The framework for NCL systems consist of DI combinational logic sandwiched between DI registers, as shown in Fig. 2.1, which is very similar to synchronous systems.

Table 2.2: Quad-Rail signal.

	DATA0	DATA1	DATA2	DATA3	NULL
Q^0	1	0	0	0	0
Q^1	0	1	0	0	0
Q^2	0	0	1	0	0
Q^3	0	0	0	1	0

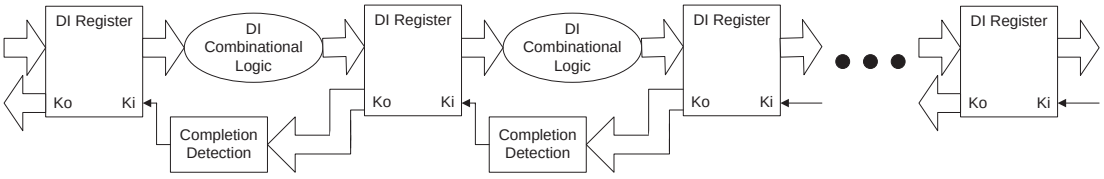


Figure 2.1: NCL system framework: input wavefronts are controlled by local handshaking signals and Completion Detection instead of by a global clock signal. Feedback requires at least three DI registers in the feedback loop to prevent deadlock.

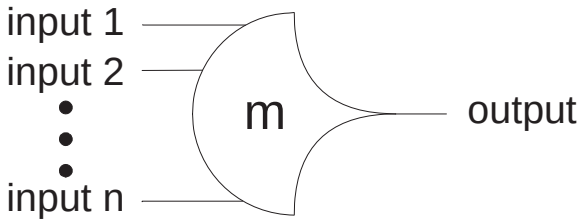


Figure 2.2: THmn threshold gate.

NCL circuits are comprised of 27 fundamental gates, as shown in Table 2.3, which constitute the set of all functions consisting of four or fewer variables. Since each rail of an NCL signal is considered a separate variable, a four variable function is not the same as a function of four literals, which would consist of eight variables for dual-rail logic (e.g., a literal includes both a variable and its complement, F and F', whereas NCL rails are never complemented, such that a dual-rail NCL signal, F, consists of two variables, F^1 and F^0, where F^0 is equivalent to F'). The primary type of threshold gate, shown in Fig. 2.2, is the *THmn gate*, where $1 \leq m \leq n$. THmn gates have n inputs. At least m of the n inputs must be asserted before the output will become asserted. In a THmn gate,

Table 2.3: 27 fundamental NCL gates.

NCL Gate	Boolean Function
TH12	$A + B$
TH22	AB
TH13	$A + B + C$
TH23	$AB + AC + BC$
TH33	ABC
TH23w2	$A + BC$
TH33w2	$AB + AC$
TH14	$A + B + C + D$
TH24	$AB + AC + AD + BC + BD + CD$
TH34	$ABC + ABD + ACD + BCD$
TH44	$ABCD$
TH24w2	$A + BC + BD + CD$
TH34w2	$AB + AC + AD + BCD$
TH44w2	$ABC + ABD + ACD$
TH34w3	$A + BCD$
TH44w3	$AB + AC + AD$
TH24w22	$A + B + CD$
TH34w22	$AB + AC + AD + BC + BD$
TH44w22	$AB + ACD + BCD$
TH54w22	$ABC + ABD$
TH34w32	$A + BC + BD$
TH54w32	$AB + ACD$
TH44w322	$AB + AC + AD + BC$
TH54w322	$AB + AC + BCD$
THxor0	$AB + CD$
THand0	$AB + BC + AD$
TH24comp	$AC + BC + AD + BD$

each of the n inputs is connected to the rounded portion of the gate; the output emanates from the pointed end of the gate; and the gate's threshold value, m, is written inside of the gate.

Another type of threshold gate is referred to as a weighted threshold gate, denoted as THmnW$w_1 w_2 \ldots w_R$. Weighted threshold gates have an integer value, $m \geq w_R > 1$, applied to *input* R. Here $1 \leq R < n$; where n is the number of inputs; m is the gate's threshold; and $w_1, w_2, \ldots w_R$, each > 1, are the integer weights of *input1, input2, …input* R, respectively. For example, consider the TH34W2 gate, whose $n = 4$ inputs are labeled A, B, C, and D, shown in Fig. 2.3. The weight

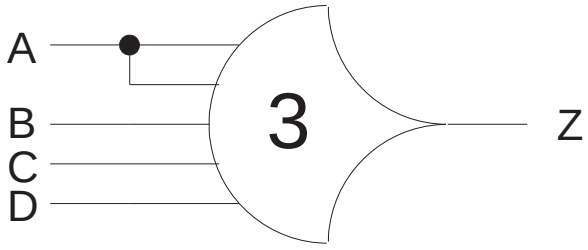


Figure 2.3: TH34w2 threshold gate: $Z = AB + AC + AD + BCD$.

of input A, $W(A)$, is, therefore, 2. Since the gate's threshold, m, is 3, this implies that in order for the output to be asserted, either inputs B, C, and D must all be asserted, or input A must be asserted along with any other input, B, C, or D.

NCL threshold gates are designed with *hysteresis* state-holding capability, such that after the output is asserted, all inputs must be deasserted before the output will be deasserted. Hysteresis ensures a complete transition of inputs back to NULL before asserting the output associated with the next wavefront of input data. Therefore, a THnn gate is equivalent to an n-input C-element (i.e., when all inputs are asserted the output is asserted; the output then remains asserted until all inputs are deasserted, at which time the output becomes deasserted); and a TH1n gate is equivalent to an n-input OR gate. NCL threshold gates may also include a *reset* input to initialize the output. Circuit diagrams designate resettable gates by either a d or an n appearing inside the gate, along with the gate's threshold. d denotes the gate as being reset to logic 1; n, to logic 0. These resettable gates are used in the design of DI registers.

NCL systems contain at least two DI registers, one at both the input and at the output. Two adjacent register stages interact through their request and acknowledge signals, K_i and K_o, respectively, to prevent the current DATA wavefront from overwriting the previous DATA wavefront, by ensuring that the two DATA wavefronts are always separated by a NULL wavefront. The acknowledge signals are combined in the Completion Detection circuitry to produce the request signal(s) to the previous register stage. NCL registration is realized through cascaded arrangements of single-bit dual-rail registers or single-signal quad-rail registers, depicted in Figs. 2.4 and 2.5, respectively. These registers consist of TH22 gates that pass a DATA value at the input only when K_i is *request for data* (rfd) (i.e., logic 1) and likewise pass NULL only when K_i is *request for null* (rfn) (i.e., logic 0). They also contain a NOR gate to generate K_o, which is *rfn* when the register output is DATA and *rfd* when the register output is NULL. The registers shown below are reset to NULL, since all TH22 gates are reset to logic 0. However, either register could be instead reset to a DATA value by replacing exactly one of the TH22n gates with a TH22d gate.

An N-bit register stage, comprised of N single-bit dual-rail NCL registers, requires N completion signals, one for each bit. The NCL completion component, shown in Fig. 2.6, uses these

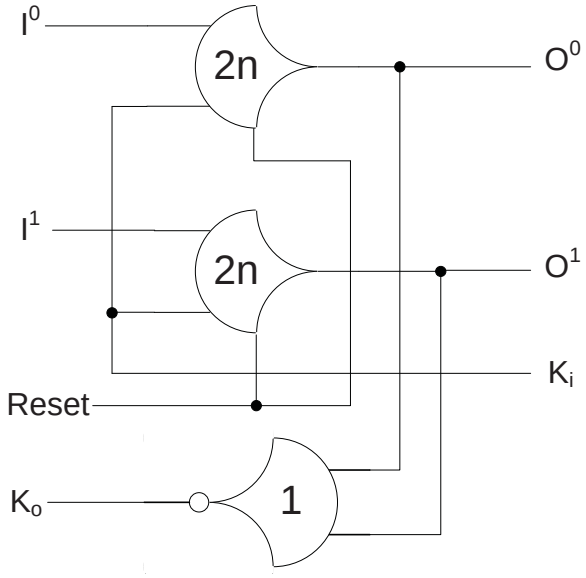


Figure 2.4: Single-bit dual-rail register.

NK_o lines to detect complete DATA and NULL sets at the output of every register stage and request the next NULL and DATA set, respectively. In full-word completion, the single-bit output of the completion component is connected to all K_i lines of the previous register stage. Since the maximum input threshold gate is the TH44 gate, the number of logic levels in the completion component for an N-bit register is given by $\lceil \log_4 N \rceil$. Likewise, the completion component for an N-bit quad-rail registration stage requires $\frac{N}{2}$ inputs, and can be realized in a similar fashion using TH44 gates. Figs. 2.7 and 2.8 show the flow of DATA and NULL wavefronts through an NCL combinational circuit (i.e., an AND function) and an arbitrary pipeline stage, respectively. The average DATA/NULL cycle time, referred to as T_{DD}, is comparable to the clock frequency of a synchronous circuit.

2.2 TRANSISTOR-LEVEL NCL GATE DESIGN

As explained in Section 2.1, NCL threshold gates are designed with *hysteresis* state-holding capability, such that after the output is asserted, all inputs must be deasserted before the output will be deasserted. Therefore, NCL gates have both *set* and *hold* equations, where the *set* equation determines when the gate will become asserted and the *hold* equation determines when the gate will remain asserted once it has been asserted. The *set* equation determines the gate's functionality as one of the 27 NCL gates, as listed in Table 2.4, whereas the *hold1* equation is simply all inputs ORed together. The general equation for an NCL gate with output Z is: $Z = \text{set} + (Z^- \bullet \textit{hold1})$, where Z^- is the previous output value and Z is the new value.

Table 2.4: 27 fundamental NCL gates.

NCL Gate	Boolean Function	Transistor Count (static)	Transistor Count (semi-static)
TH12	$A + B$	6	6
TH22	AB	12	8
TH13	$A + B + C$	8	8
TH23	$AB + AC + BC$	18	12
TH33	ABC	16	10
TH23w2	$A + BC$	14	10
TH33w2	$AB + AC$	14	10
TH14	$A + B + C + D$	10	10
TH24	$AB + AC + AD + BC + BD + CD$	26	16
TH34	$ABC + ABD + ACD + BCD$	24	16
TH44	$ABCD$	20	12
TH24w2	$A + BC + BD + CD$	20	14
TH34w2	$AB + AC + AD + BCD$	22	15
TH44w2	$ABC + ABD + ACD$	23	15
TH34w3	$A + BCD$	18	12
TH44w3	$AB + AC + AD$	16	12
TH24w22	$A + B + CD$	16	12
TH34w22	$AB + AC + AD + BC + BD$	22	14
TH44w22	$AB + ACD + BCD$	22	14
TH54w22	$ABC + ABD$	18	12
TH34w32	$A + BC + BD$	17	12
TH54w32	$AB + ACD$	20	12
TH44w322	$AB + AC + AD + BC$	20	14
TH54w322	$AB + AC + BCD$	21	14
THxor0	$AB + CD$	20	12
THand0	$AB + BC + AD$	19	13
TH24comp	$AC + BC + AD + BD$	18	12

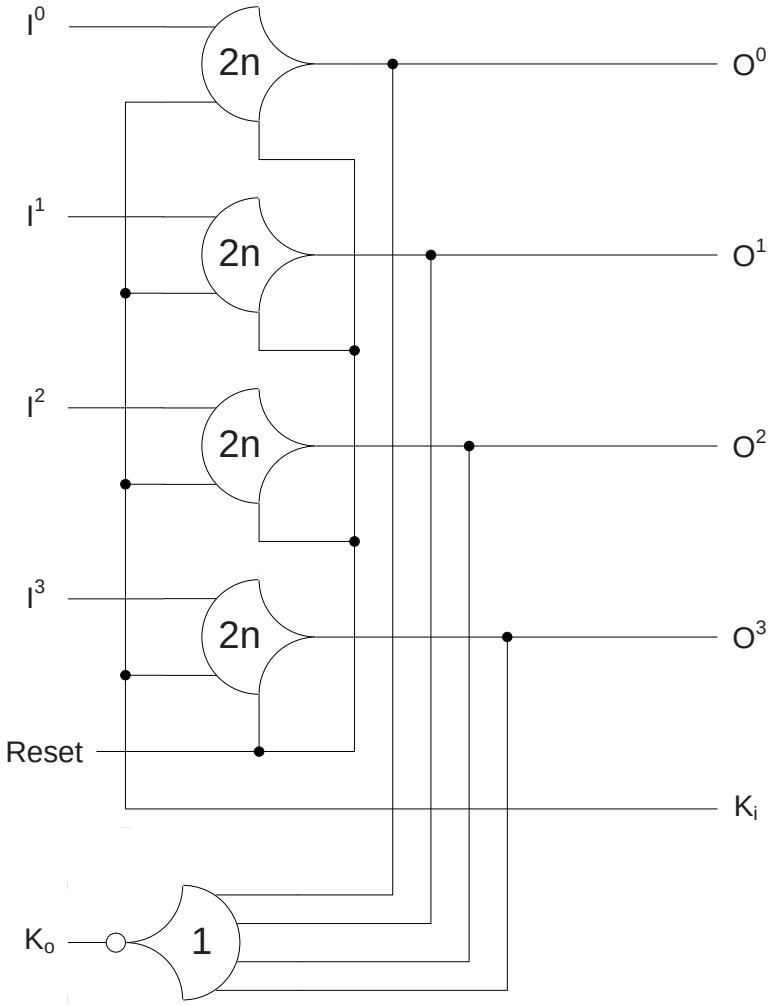


Figure 2.5: Single-signal quad-rail register.

To implement an NCL gate using CMOS technology, an equation for the complement of Z is also required, which in general form is: $Z' = \text{reset} + (Z^{-'} \bullet \textit{hold0})$, where *reset* is the complement of *hold1* (i.e., the complement of each input, ANDed together) and *hold0* is the complement of *set*, such that the gate output is deasserted when all inputs are deasserted, and then remains deasserted while the gate's *set* condition is false. To achieve hysteresis state-holding behavior, the new output value, Z, depends on the previous output value, Z^-, which requires internal gate feedback, as shown in Fig. 2.9. For the static realization, the equations for Z and Z', given above, are directly implemented in the

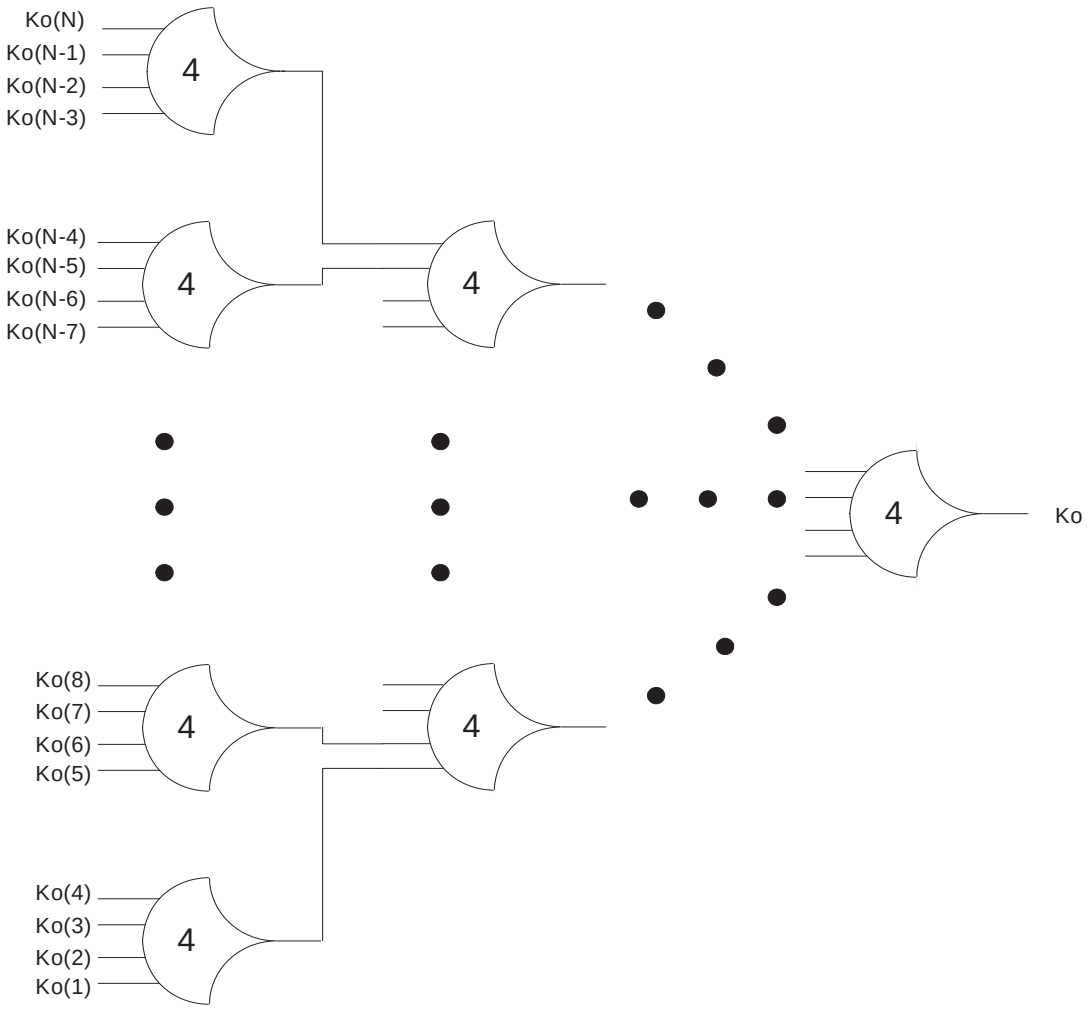


Figure 2.6: *N*-bit completion component.

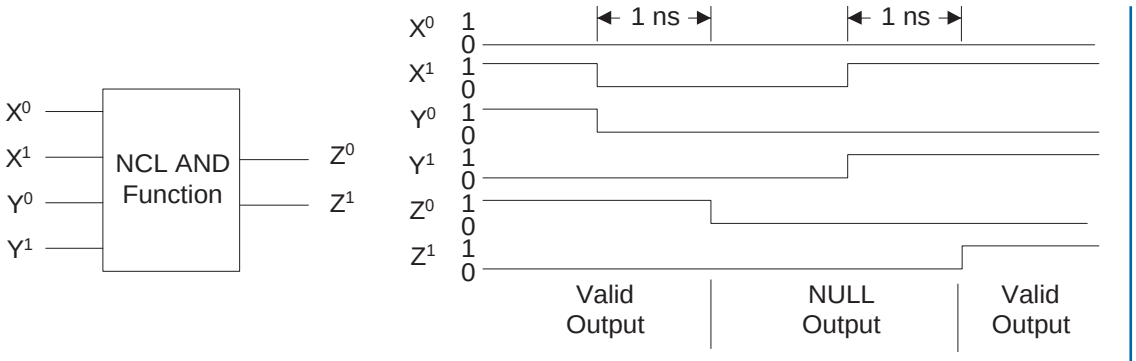
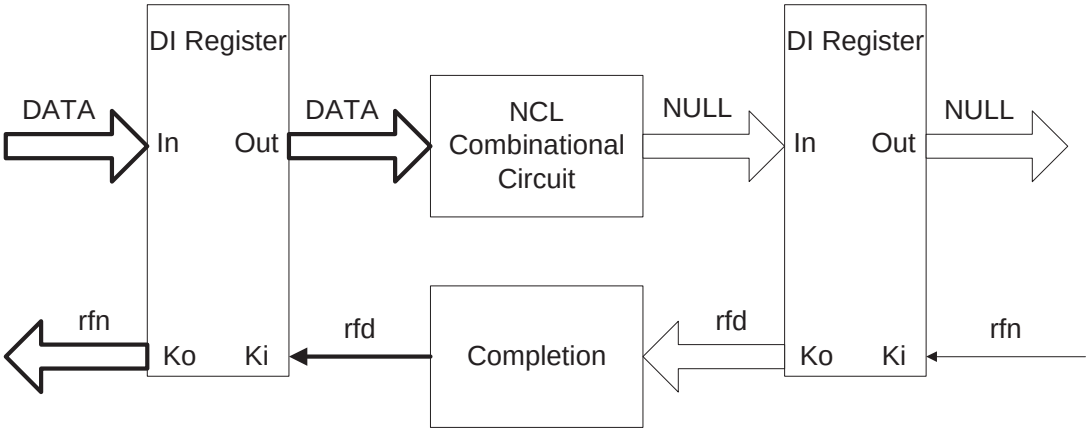


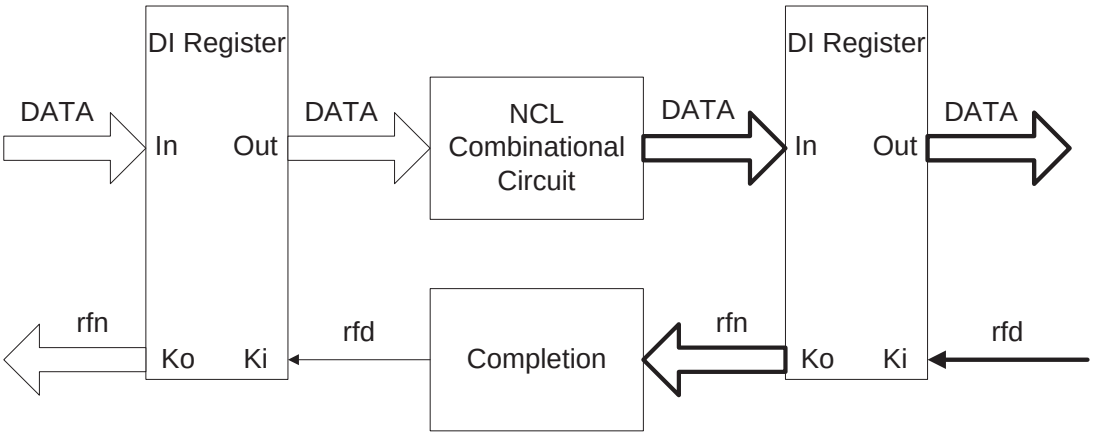
Figure 2.7: NCL AND function: $Z = X \bullet Y$: initially X =DATA1 and Y =DATA0, so Z =DATA0; next X and Y both transition to NULL, so Z transitions to NULL; then X and Y both transition to DATA1, so Z transitions to DATA1.

NMOS and PMOS logic, respectively, after simplifying; whereas, the semi-static realization only requires the *set* and *reset* equations to be implemented in the NMOS and PMOS logic, respectively, and *hold0* and *hold1* are implemented using a weak inverter.

For example, the *set* equation for the TH23 gate is $AB + AC + BC$, as given in Table 2.4, and the *hold* equation is $A + B + C$; therefore, the gate is asserted when at least 2 inputs are asserted and it then remains asserted until all inputs are deasserted. The *reset* equation is $A'B'C'$ and the simplified *set'* equation is $A'B' + B'C' + A'C'$. Directly implementing these equations for Z and Z', after simplification, yields the static transistor-level realization, shown in Fig. 2.10(a). The semi-static TH23 gate is shown in Fig. 2.10(b). In general, the semi-static implementation requires fewer transistors, but is slightly slower because of the weak inverter. Note that TH1n gates are simply OR gates and do not require any feedback, such that their static and semi-static implementations are exactly the same.

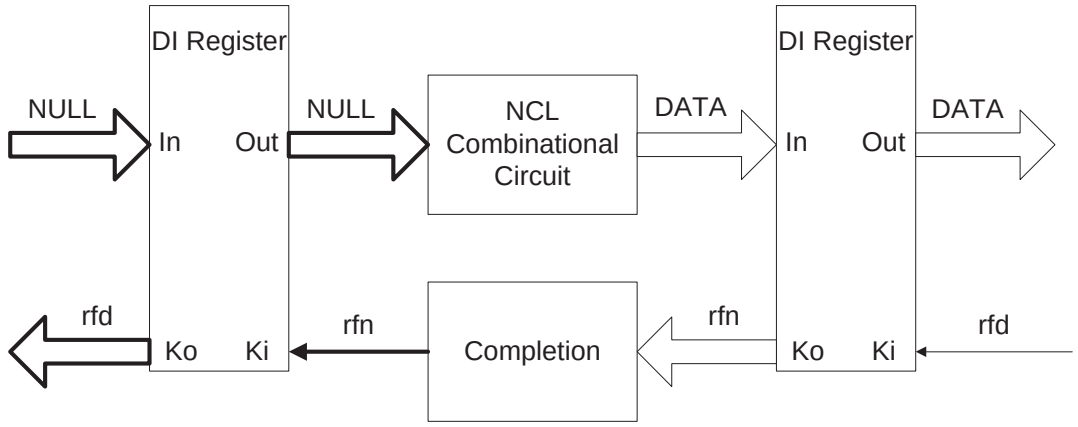


a) DATA flows through input register and combinational circuit

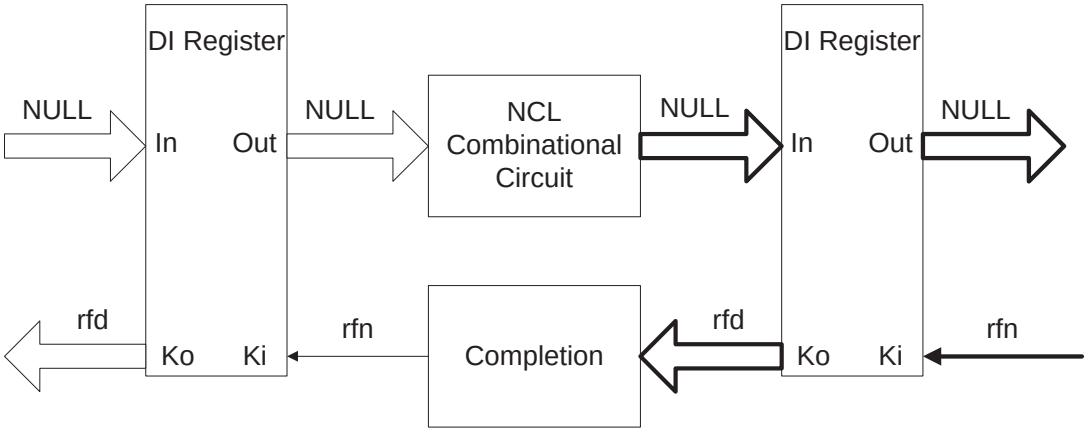


b) DATA flows through output register and *rfn* flows through completion circuit

Figure 2.8: NCL DATA/NULL cycle. a)DATA flows through input register and combinational circuit; b)DATA flows through output register and *rfn* flows through completion circuit;



c) NULL flows through input register and combinational circuit



d) NULL flows through output register and *rfd* flows through completion circuit

Figure 2.8: NCL DATA/NULL cycle. c)NULL flows through input register and combinational circuit; d)NULL flows through output register and *rfd* flows through completion circuit.

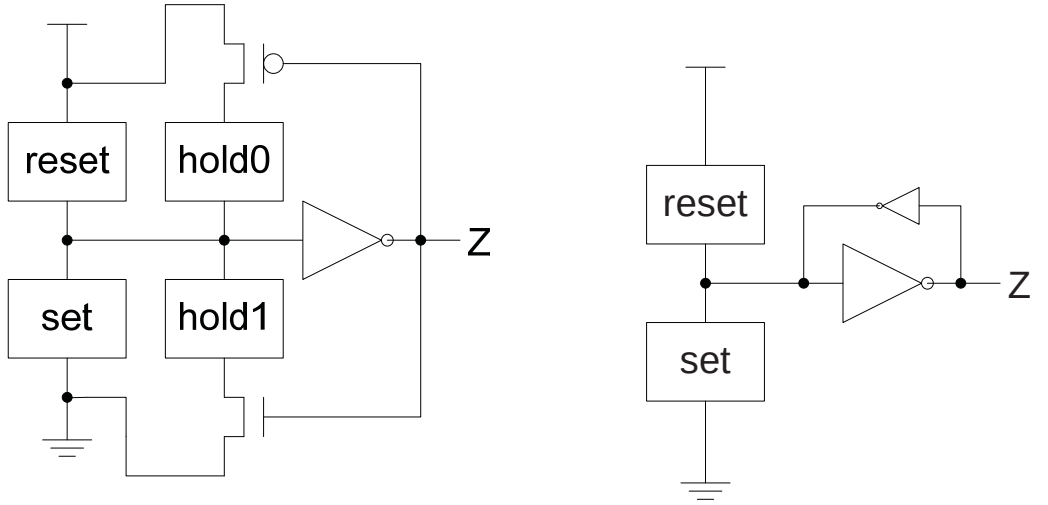


Figure 2.9: NCL gate realizations. a) static implementation; b) semi-static implementation.

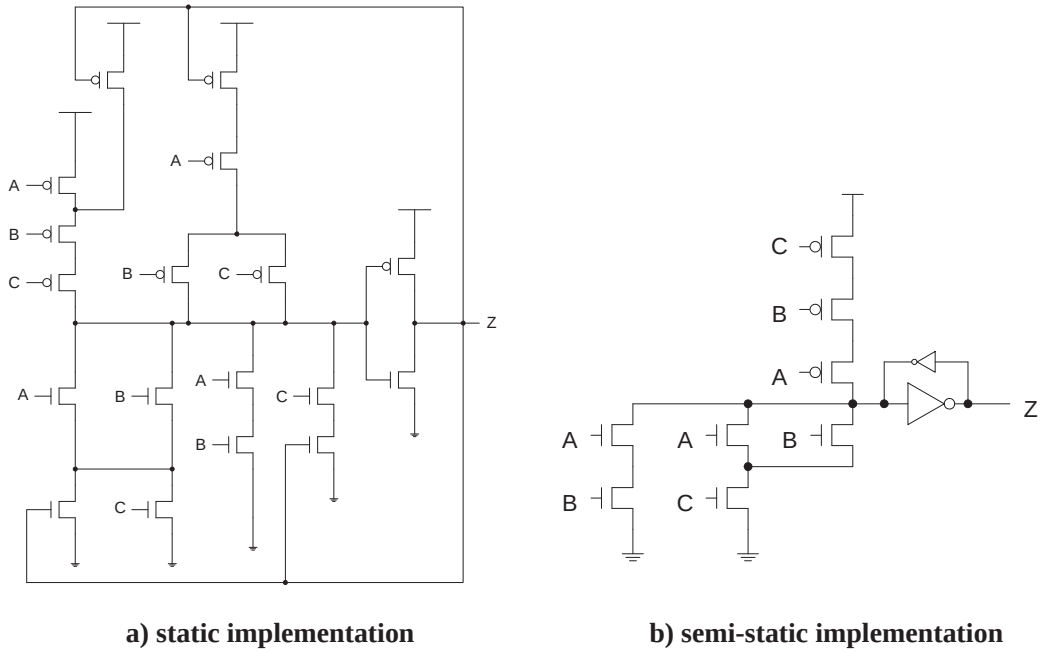


Figure 2.10: TH23 gate realizations. a) static implementation; b) semi-static implementation.

CHAPTER 3

Combinational NCL Circuit Design

NCL circuit design is similar to synchronous Boolean design, where minimized equations are generated and then mapped to a set of gates; however, NCL circuits must be both input-complete and observable in order to achieve delay-insensitivity.

3.1 INPUT-COMPLETENESS AND OBSERVABILITY

Input-Completeness requires that all outputs of a combinational circuit may not transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and that all outputs of a combinational circuit may not transition from DATA to NULL until all inputs have transitioned from DATA to NULL. In circuits with multiple outputs, it is acceptable according to Seitz's "weak conditions" of delay-insensitive signaling, for some of the outputs to transition without having a complete input set present, as long as all outputs cannot transition before all inputs arrive. For example, the NCL AND function in Fig. 3.1 is not input-complete because the output, Z, will transition to DATA0 if either input is DATA0, even if the other input is NULL. However, the half-

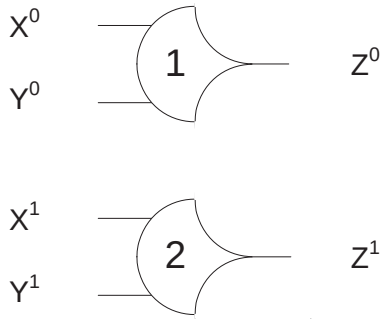


Figure 3.1: Input-incomplete NCL AND function.

adder in Fig. 3.2 is input-complete, even though C_{out} is not input-complete, because both inputs must be DATA in order for S to transition to DATA, such that the entire output set, $\{S, C_{out}\}$, cannot transition to DATA until both inputs transition to DATA. The hysteresis within each NCL gate ensures that all inputs must transition to NULL before a combinational circuit's output will

transition to NULL, making the circuit input-complete with respect to NULL, assuming that the circuit is input-complete with respect to DATA.

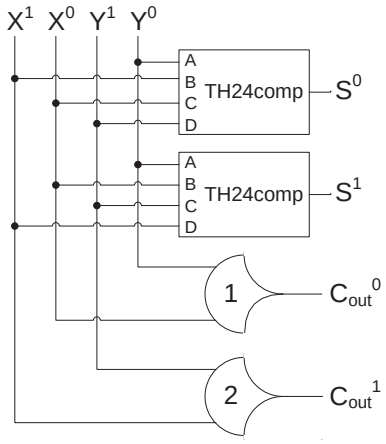


Figure 3.2: NCL half-adder.

To determine if a circuit is input complete, one must analyze the equation for each output. An output is input-complete with respect to a particular input if and only if every *non-don't care* product term in the output's equation (i.e., equations for all rails of the output) contains any of the rails of the particular input. Take Fig. 3.3, for example. The output equations are as follows: $X^0 = B^0C^1 + B^0B^1 + C^0C^1 + C^0B^1$; $X^1 = A^1B^1A^0 + A^0B^0 + A^1B^1C^1$; $Y^0 = A^0C^0 + A^1C^1$; $Y^1 = A^1B^1C^0 + A^1B^1B^0 + C^1C^0 + C^1B^0$. Removing the *don't care* terms, where two rails of the same signal are both asserted (i.e., both rails can never be simultaneously asserted; they are mutually exclusive), yields the following equations: $X^0 = B^0C^1 + C^0B^1$; $X^1 = A^0B^0 + A^1B^1C^1$; $Y^0 = A^0C^0 + A^1C^1$; $Y^1 = A^1B^1C^0 + C^1B^0$. X has a B in each product term, so it is input-complete with respect to B. Y has a C in each product term, so it is input-complete with respect to C. To make the circuit input-complete with respect to A, A must be added to all product terms in which it is missing in either X or Y, but not both. Since A is only missing in one of the Y product terms, it is added here, by ANDing the product term with logic 1, formed by ORing both rails of A, resulting in the following equation: $Y^1 = A^1B^1C^0 + C^1B^0(A^1 + A^0) = A^1B^1C^0 + C^1B^0A^1 + C^1B^0A^0$. However, since Y^0 contains an A^1C^1 product term, the new $C^1B^0A^1$ product term in Y^1 must have been a *don't care* in the original expression, since both Y^0 and Y^1 cannot be simultaneously asserted; therefore, the $C^1B^0A^1$ *don't care* term can be removed to simplify Y^1 as: $Y^1 = A^1B^1C^0 + C^1B^0A^0$. The input-complete circuit can then be redrawn as shown in Fig. 3.4.

Observability requires that no *orphans* may propagate through a gate. An orphan is defined as a wire that transitions during the current DATA wavefront but is not used in the determination

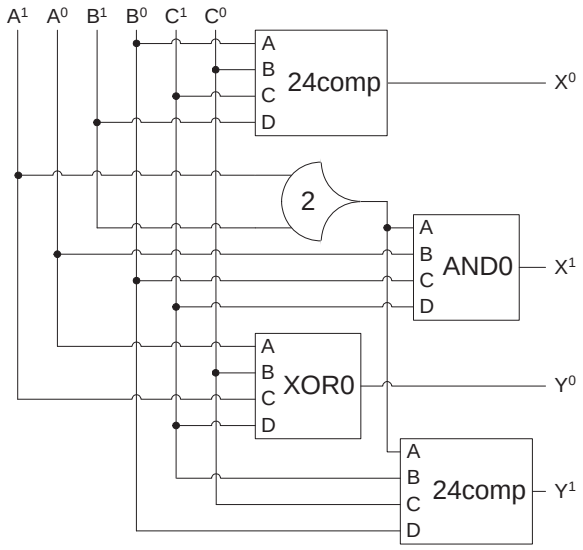


Figure 3.3: NCL circuit that's input-incomplete with respect to A.

of the output. Orphans are caused by wire forks and can be neglected through the isochronic fork assumption (i.e., gate delays are much longer than wire delays within a component, such as a full adder, which is a valid assumption even in future nanometer technologies), as long as they are not allowed to cross a gate boundary. This *observability* condition, also referred to as indicatability or stability, ensures that every gate transition is observable at the output, which means that every gate that transitions is necessary to transition at least one of the outputs. Consider an unobservable version of an XOR function, shown in Fig. 3.5, where an orphan is allowed to pass through the TH12 gate. For instance, when X =DATA0 and Y =DATA0, the TH12 gate is asserted, but does not take part in the determination of the output, Z =DATA0. This orphan path is shown in boldface in Fig. 3.5. The equation for Z^1 can be repartitioned to obtain a fully observable version of the XOR function, as shown in Fig. 3.6. Here, the two internal TH22 gates are each connected to a TH23W2 output gate with a weight of 2, which is the same as the threshold, such that if either internal gate is asserted, its corresponding output gate will always become asserted. Note that this circuit is for example only, since the XOR function can be simplified to two TH24comp gates.

The best way to ensure that a circuit is observable is to not divide product terms when mapping equations to their corresponding gate-level circuits. This, however, is not required for a circuit to be observable, and is not always possible, for example when a product term contains more than four variables. The circuit in Fig. 3.4 is observable even though a product term has been divided. The TH33 gate is observable because its output has the same weight as the output gate's threshold, similar to the previous example. This is not the case for the TH22 gate, so it must be analyzed more

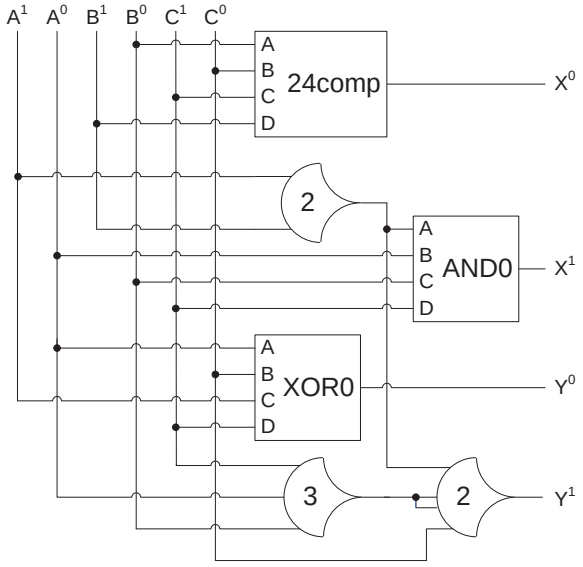


Figure 3.4: Input-complete NCL circuit.

closely. The equation for the TH22 gate is A^1B^1; and its output is used in the X^1 and Y^1 product terms: $A^1B^1C^1$ and $A^1B^1C^0$, respectively. Therefore, if the TH22 gate is asserted, it will always cause either X^1 or Y^1 to become asserted because C must either be DATA0 or DATA1; hence the TH22 gate is observable.

3.2 DUAL-RAIL NCL DESIGN

The design process for NCL combinational circuits is similar to Boolean circuits, where a Karnaugh map, or other simplification technique, can be utilized to determine the simplified sum-of-products (SOP) expressions for each output. However, SOP expressions for both the function's 1 and 0 outputs are needed. **The *0s* refer to a signal's *rail*^{0} and the *1s* refer to a signal's *rail*^{1}**. After expressions for the outputs have been obtained, an assessment must be made to ensure that the circuit is input-complete. If not, the missing input(s) must be added to the appropriate product term(s), as explained in Section 3.1. The output equations must then be partitioned into sets of four or fewer variables to be mapped to the 27 NCL gates, while ensuring that the resulting circuit is observable. To minimize area and delay, partitioning should be performed such that the minimal number of sets is obtained, which will occur when the maximum number of product terms are grouped into each set.

Take, for example, the design of a partial product (*PP*) generation component for the most significant bit of an unsigned Booth2 multiplier, which is only required to be input-complete with respect to input, MR_1. Fig. 3.7 shows the Karnaugh map for this component, along with the optimal

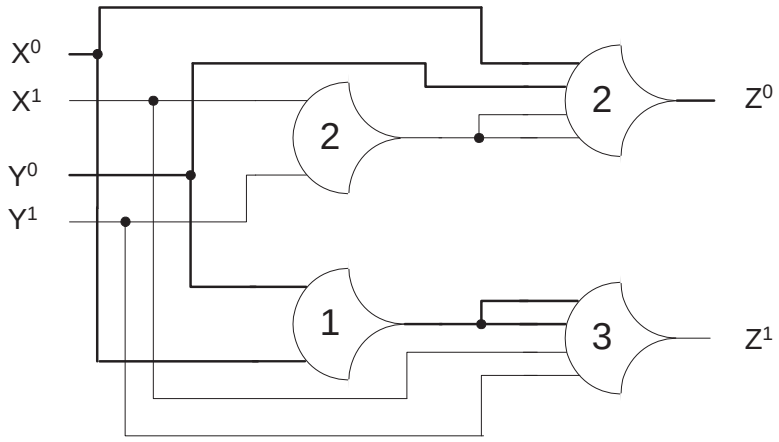


Figure 3.5: Unobservable NCL XOR function.

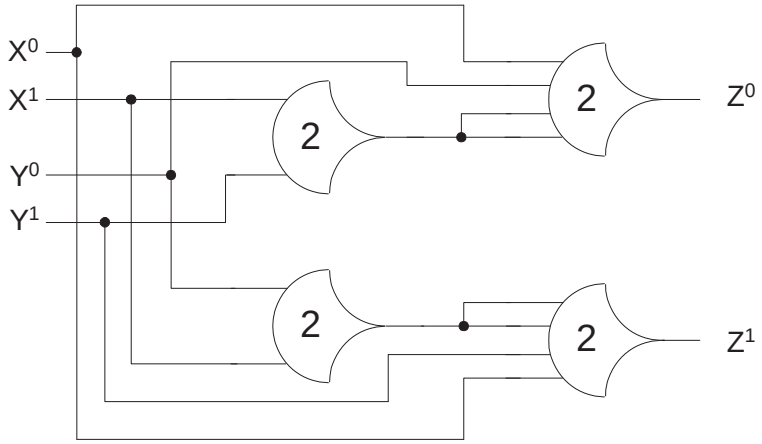


Figure 3.6: Observable NCL XOR function.

coverings. Since this design must be input-complete with respect to MR_1, the coverings should not eliminate MR_1 from the corresponding product term; hence some of the coverings are 2-coverings instead of 4-coverings. The SOP equations are derived directly from the K-map coverings (as shown in Fig. 3.8).

Since each product term contains MR_1, the circuit is input-complete with respect to MR_1. The equations can be partitioned into four sets of 4 variables as shown in Fig. 3.8, resulting in the optimized circuit as shown in Fig. 3.9. The first circled terms in both PP^0 and PP^1 each map to a

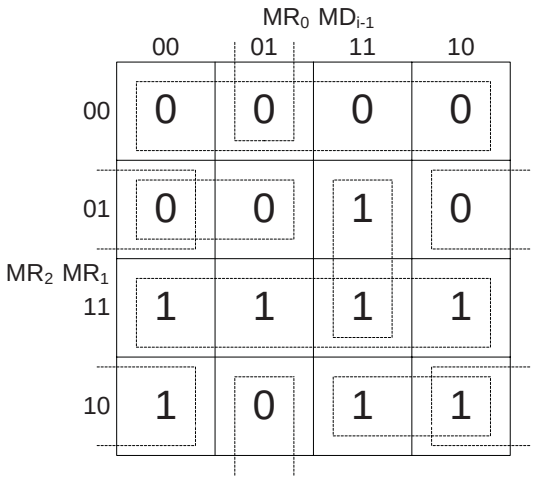


Figure 3.7: K-map for Booth2 *PP* generation component.

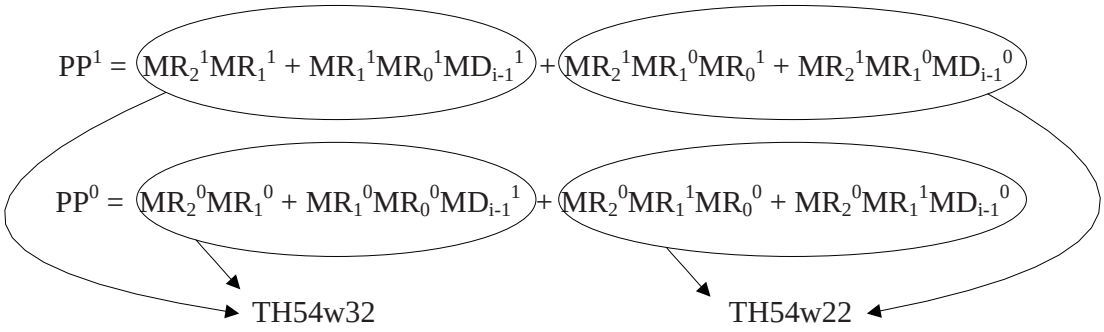


Figure 3.8: Booth2 *PP* equations and groupings.

TH54w32 gate; and the second circled PP^0 and PP^1 terms each map to a TH54w22 gate. The two terms for each output rail are then ORed together with TH12 gates. Since no product terms were divided, the resulting circuit is observable.

Now let's consider the design of input-complete optimized 2-input fundamental Boolean logic functions. Since each of these functions requires 2 dual-rail inputs, each output rail equation will consist of at most 4 input variables, such that the combinational logic for each rail will require only 1 NCL gate. The canonical SOP expressions (i.e., all inputs are contained in every product term) for an OR function, $Z = X + Y$, are $Z^0 = X^0Y^0$ and $Z^1 = X^1Y^1 + X^0Y^1 + X^1Y^0$. Z^0 directly maps to a TH22 gate and Z^1 directly maps to a THand0 gate. Similarly, canonical SOP expressions for an AND function, $Z = X \bullet Y$, are $Z^0 = X^0Y^0 + X^0Y^1 + X^1Y^0$ and $Z^1 = X^1Y^1$. Z^0 directly

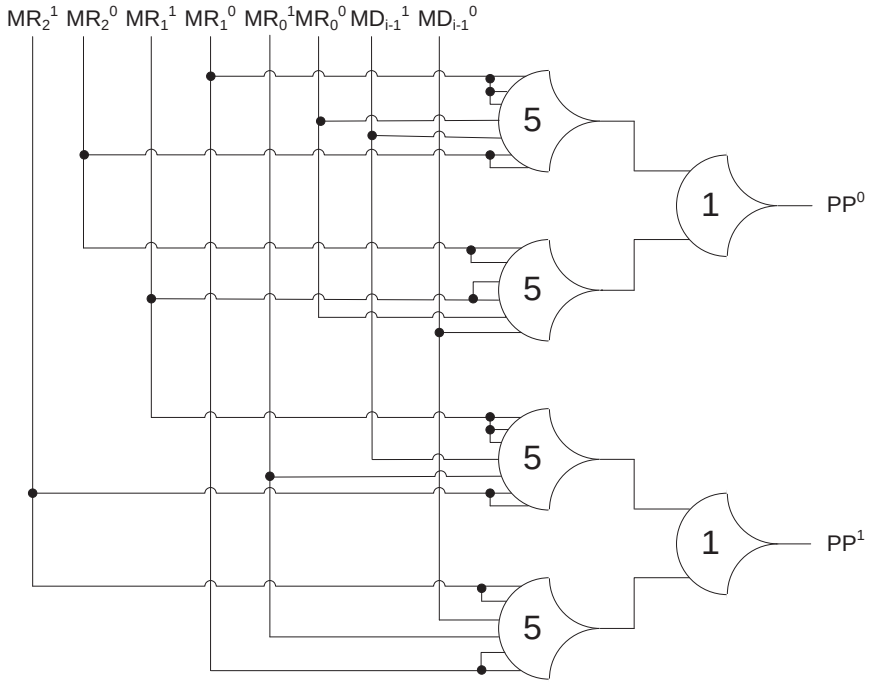


Figure 3.9: NCL Booth2 *PP* generation circuit.

maps to a THand0 gate and Z^1 directly maps to a TH22 gate. The optimized XOR function, $Z = X \oplus Y$, is a bit more complex. The canonical SOP expressions are $Z^0 = X^0Y^0 + X^1Y^1$ and $Z^1 = X^1Y^0 + X^0Y^1$, which both directly map to a THxor0 gate. However, two transistors can be eliminated for each rail of Z (when using static gates) by adding two *don't care* terms, representing the cases when both rails of either X or Y are simultaneously asserted. The new equations are: $Z^0 = X^0Y^0 + X^1Y^1 + X^0X^1 + Y^0Y^1$ and $Z^1 = X^1Y^0 + X^0Y^1 + X^0X^1 + Y^0Y^1$, both of which now map to a TH24comp gate. An NCL inverter, $F = Z'$, is realized by simply swapping rails: $F^0 = Z^1$ and $F^1 = Z^0$; therefore, the inverse logic functions, NAND, NOR, and NXOR, are obtained by exchanging the output rails of the AND, OR, and XOR functions, respectively.

Finally, let's design an optimized NCL full adder, whose truth table is shown in Fig. 3.10, where X and Y denote the input addends and C_i denotes the carry input. S and C_o denote the *sum* and *carry* outputs, respectively. The K-map for the C_o output is shown in Fig. 3.11, yielding: $C_o^0 = X^0Y^0 + C_i^0X^0 + C_i^0Y^0$ and $C_o^1 = X^1Y^1 + C_i^1X^1 + C_i^1Y^1$, both of which directly map to a TH23 gate.

Since C_o is not input-complete with respect to any inputs, S must be input-complete with respect to all inputs, which means that the equations for S in terms of the inputs, X, Y, and C_i,

X	Y	Ci	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3.10: Truth table for full adder.

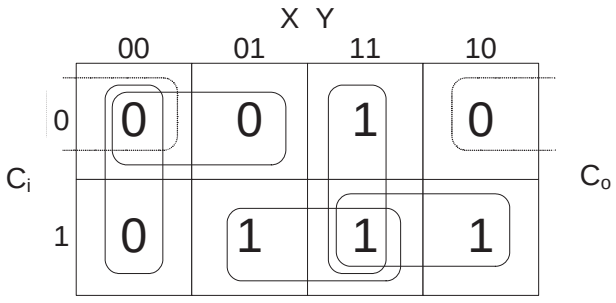


Figure 3.11: K-map for C_o output of full adder.

must be in canonical form: $S^0 = X^0Y^0C_i^0 + X^0Y^1C_i^1 + X^1Y^0C_i^1 + X^1Y^1C_i^0$ and $S^1 = X^0Y^0C_i^1 + X^0Y^1C_i^0 + X^1Y^0C_i^0 + X^1Y^1C_i^1$, which if implemented directly would require two gate delays and four TH33 gates and one TH14 gate for each rail. The equation for S could also be written as $S = (X \oplus Y) \oplus C_i$, which would require two gate delays and four TH24 comp gates. However, since S requires two gate delays and C_o is generated in only one gate delay, C_o could be utilized as a fourth input to generate S, possibly reducing the number of gates without increasing delay.

The K-map for S, based on X, Y, C_i, and C_o, is shown in Fig. 3.12, with essential prime implicants covered. This covering yields: $S^0 = C_o^1X^0 + C_o^1Y^0 + C_o^1C_i^0 + X^0Y^0C_i^0$ and $S^1 = C_o^0X^1 + C_o^0Y^1 + C_o^0C_i^1 + X^1Y^1C_i^1$, both of which directly map to a TH34W2 gate. Checking input-completeness, the *carry* output requires at least two inputs to be generated and the *sum* output requires either the carry output and the third input, or all three inputs to be generated; so all three

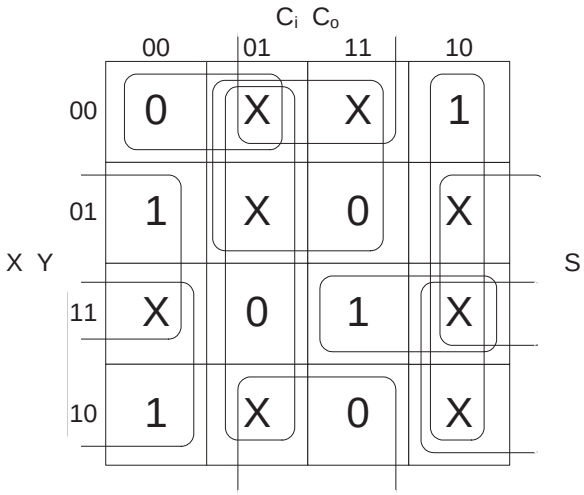


Figure 3.12: K-map for *S* output of full adder.

inputs are needed to generate the *sum* output. Therefore, the circuit as a whole is input-complete even though C_o is not. Furthermore, the *sum* output and, therefore, this circuit, is inherently input-complete since it is impossible to determine the value of *S* without knowing the value of all three inputs, *X*, *Y*, and C_i. The resulting optimized NCL full adder circuit is shown in Fig. 3.13.

3.3 QUAD-RAIL NCL DESIGN

The design process for NCL quad-rail circuits is similar to dual-rail circuits, where a Karnaugh map, or other simplification technique, can be utilized to determine the simplified SOP expressions for each output rail. However, instead of only *0s* and *1s*, corresponding to a signal's $rail^0$ and $rail^1$, respectively, the K-map also contains *2s* and *3s*, which correspond to a signal's $rail^2$ and $rail^3$, respectively. The *0* outputs are then grouped together to obtain a minimized expression for $rail^0$; the *1* outputs are grouped together to obtain a minimized expression for $rail^1$; the *2* outputs are grouped together to obtain a minimized expression for $rail^2$; and the *3* outputs are grouped together to obtain a minimized expression for $rail^3$. After expressions for the outputs have been obtained, an assessment must be made to ensure that the circuit is input-complete; and if not, the missing input(s) must be added to the appropriate product term(s), as explained in Section 3.1. The output equations must then be partitioned into sets of four or fewer variables to be mapped to the 27 NCL gates, while ensuring that the resulting circuit is observable. To minimize area and delay, partitioning should be performed such that the minimal number of sets is obtained, which will occur when the maximum number of product terms are grouped into each set, the same as for dual-rail NCL design.

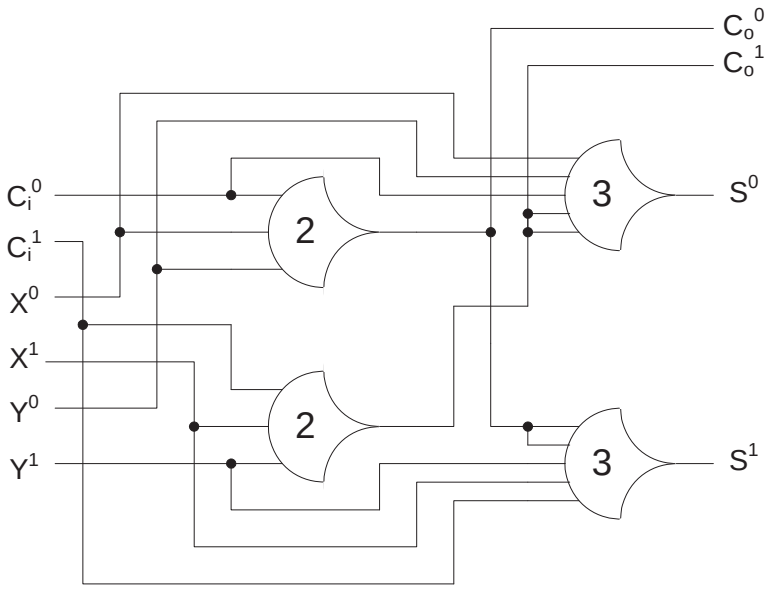


Figure 3.13: Optimized NCL full adder.

Take, for example, the design of a quad-rail partial product (*PP*) generation component, depicted in Fig. 3.16, for use in an unsigned quad-rail multiplier. Remember that each quad-rail signal corresponds to 2 bits, such that the quad-rail partial product (*PP*) generation component is equivalent to 2 bits × 2 bits, which yields a 4-bit result, and hence 2 quad-rail signals, *PPH* and *PPL*. Fig. 3.15 shows the Karnaugh maps for this component, along with the optimal coverings. Note that only linear 4-coverings, which contain all four rails of a quad-rail signal, can be utilized to eliminate a quad-rail signal from the corresponding product term; 2-coverings will not eliminate a quad-rail signal, and are, therefore, not used. Because of this, the input order does not need to be rearranged like required for Boolean and dual-rail K-maps (i.e., 0, 1, 2, 3 for quad-rail vs. 00, 01, 11, 10 for Boolean and dual-rail). Also note that 3 does not appear as an output in the *PPH* K-map. This is because the maximum value of *PPH* is 2, resulting when *A* and *B* are both 3 (i.e., $3 \times 3 = 9_{10} = 1001_2 = 21_4$); therefore, PPH^3 is always 0, and can be treated as a *don't care* in subsequent circuits that use *PPH* as an input. The minimal SOP equations are derived directly from the K-map coverings as shown in Fig. 3.16.

Since all product terms in either *PPL* or *PPH* do not contain either an *A* or *B*, the resulting circuit is not input-complete with respect to either input; therefore, additional terms must be added. Specifically, additional terms must be added to either PPL^0 or PPH^0, since these are the input-incomplete rails of *PPL* and *PPH*, respectively. Making PPL^0 input-complete requires fewer additional terms and does not increase *PPL*'s worse-case delay, whereas making PPH^0 input-

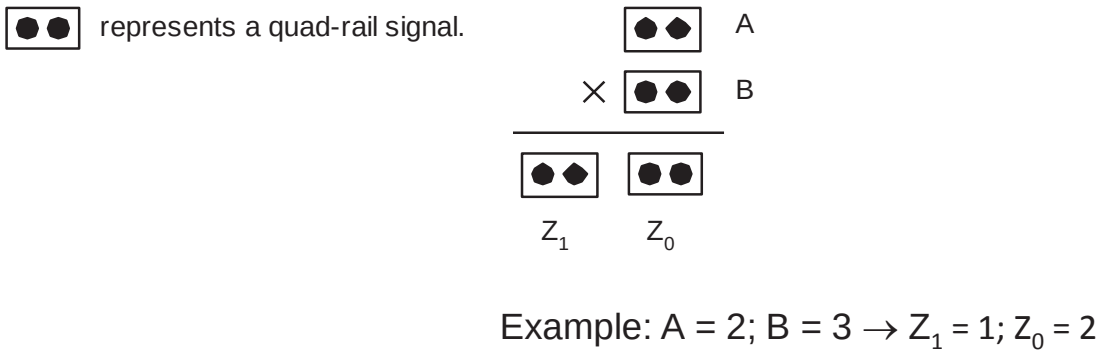


Figure 3.14: Quad-rail *PP* generation component.

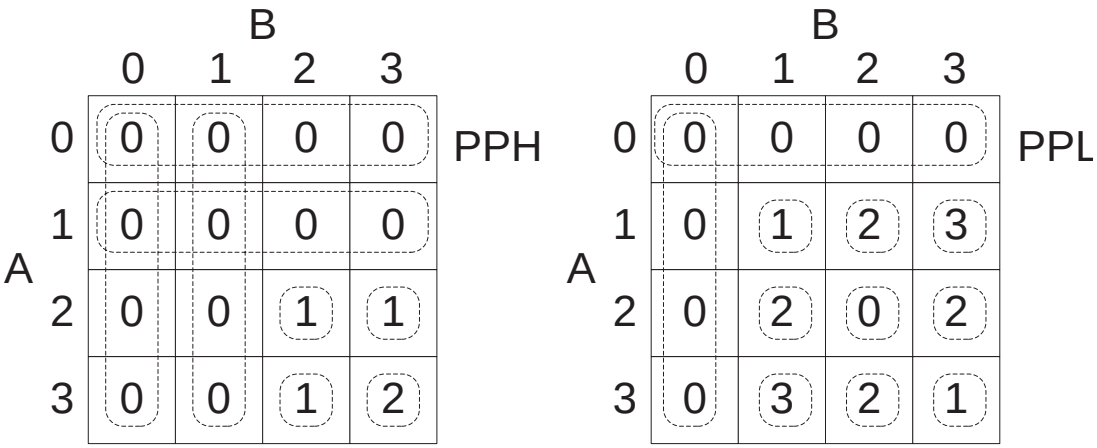


Figure 3.15: K-maps for quad-rail *PP* generation component.

complete would increase *PPH*'s worse-case delay from 1 gate to 2 gates; therefore, additional terms are added to *PPL* to make it input-complete with respect to both *A* and *B*, as shown in the PPL^0 equations. Since the first product term, A^0, is missing *B*, *B* is added to the product term by ANDing it with logic 1, formed by ORing all rails of *B* together. Likewise, the second product term, B^0, is missing *A*, so *A* is added to the product term by ANDing it with all rails of *A* ORed together. After distributing AND over OR and removing the redundant A^0B^0 term, the minimal input-complete equation for PPL^0 is obtained. Now all product terms in all rails of *PPL* contain both an *A* and *B*; so *PPL*, and, therefore, the entire circuit, is input-complete. Note that the minimal input-complete

$PPH^0 = A^0 + A^1 + B^0 + B^1 \rightarrow TH14$

$PPH^1 = A^2B^2 + A^2B^3 + A^3B^2 \rightarrow THand0$

$PPH^2 = A^3B^3 \rightarrow TH22$

$PPH^3 = 0$

$PPL^0 = A^0 + B^0 + A^2B^2 = A^0(B^0 + B^1 + B^2 + B^3) + B^0(A^0 + A^1 + A^2 + A^3) + A^2B^2$

$= (A^0B^3 + A^0B^1) + (A^3B^0 + A^1B^0) + (A^2B^2 + A^2B^0 + A^0B^2 + A^0B^0)$

TH33w2

$PPL^1 = A^1B^1 + A^3B^3 = A^1B^1 + A^3B^3 + A^1A^3 + B^1B^3 \rightarrow TH24comp$

$PPL^2 = (A^2B^1 + A^2B^3) + A^1B^2 + A^3B^2 \rightarrow TH34w32$

TH33w2

$PPL^3 = A^1B^3 + A^3B^1 = A^1B^3 + A^3B^1 + A^1A^3 + B^1B^3 \rightarrow TH24comp$

Figure 3.16: Quad-rail *PP* generation equations and groupings.

equation for PPL^0 can also be obtained directly from the K-map by only utilizing 1-coverings for the *0* outputs, which directly yields the canonical expression for PPL^0.

The equations for each rail of *PPH* directly map to one NCL gate, as shown in Fig. 3.16. The equations for PPL^1 and PPL^3 each map to TH24comp gates after adding two *don't care* terms, representing the cases when two rails of either *A* or *B* are simultaneously asserted, as shown in Fig. 3.16, resulting in two fewer transistors for implementing each rail (when using static gates). The equation for PPL^0 can be partitioned into one set of 4 variables and two sets of 3 variables, as shown in Fig. 3.16. The first and second circled terms each map to a TH33w2 gate, and the third circled term maps to a TH24comp gate. These three terms are then ORed together with a TH13 gate. The equation for PPL^2 can be partitioned into one set of 3 variables and a second set of 4 variables that contains the output of the first set as one input, as shown in Fig. 3.16. The inner circled term maps to a TH33w2 gate, and the outer circled term (third and fourth product terms along with the output of the TH33w2 gate) maps to a TH34w32 gate. Since no product terms were divided, the circuit is observable. The resulting optimized circuit is shown in Fig. 3.17.

Now let's consider the design of the increment circuitry for a 4-bit counter shown in Fig. 3.18. The specifications for this counter included a full NCL interface with request and acknowledge signals labeled K_i and K_o, respectively. Functionality was specified to reset *count* to 00_4 (i.e., 0000_2)

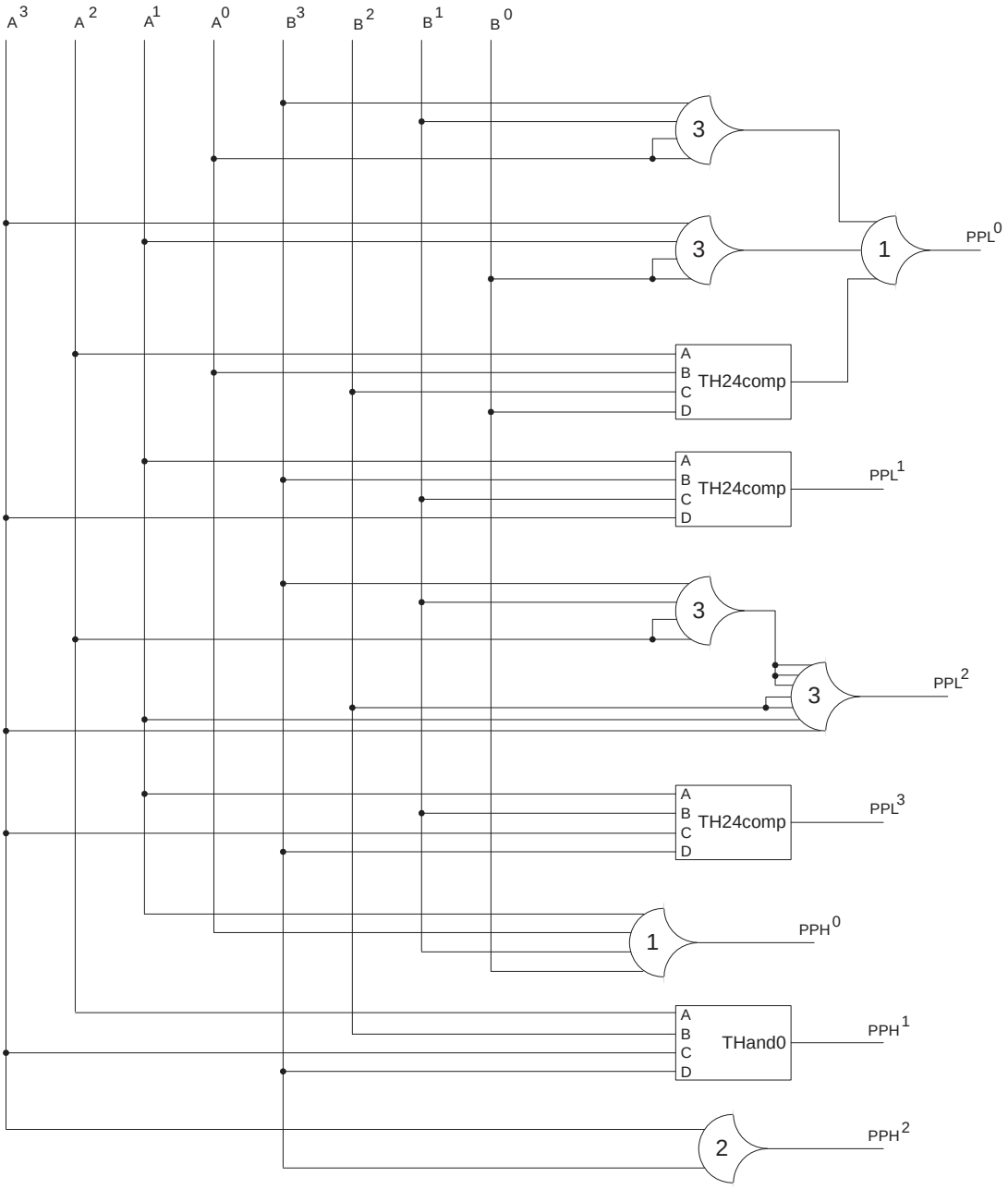


Figure 3.17: Optimized quad-rail *PP* generation component.

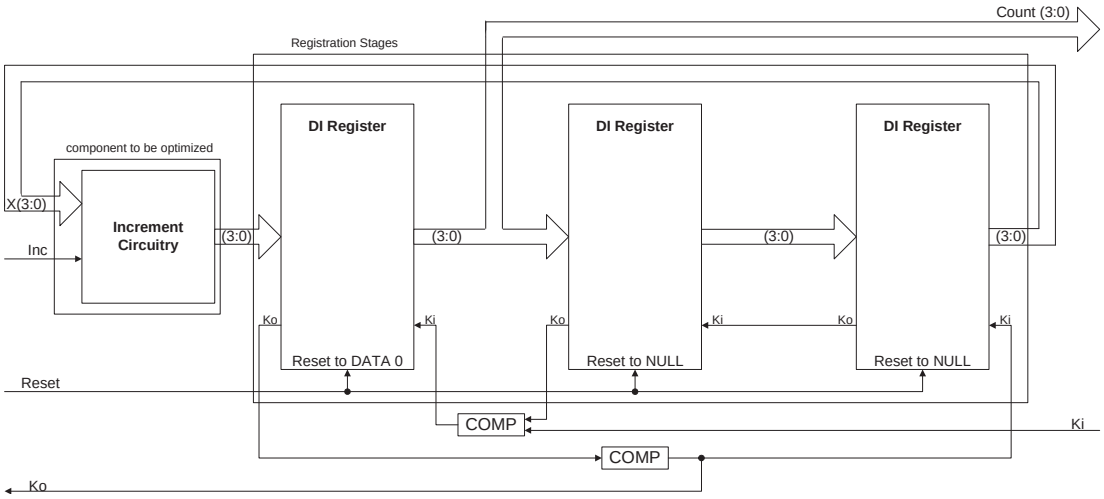


Figure 3.18: NCL up-counter with three-register feedback.

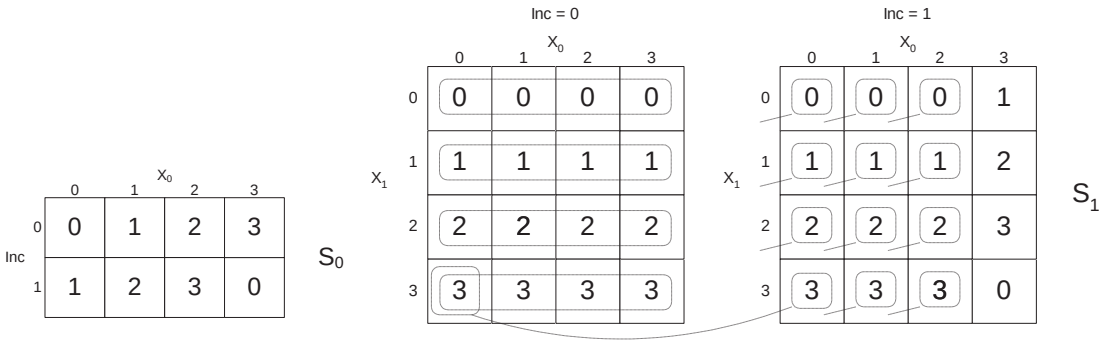


Figure 3.19: K-maps for quad-rail increment circuitry.

when the *reset* signal is applied, to increment *count* by 1 when $inc = 1$, and to keep *count* the same when $inc = 0$. The counter will rollover to 00_4 (i.e., 0000_2) when $count = 33_4$ (i.e., 1111_2) and $inc = 1$.

To design the increment circuitry using quad-rail logic requires a dual-rail *Inc* input and two quad-rail inputs, X_1 and X_0, and two quad-rail outputs, S_1 and S_0. Fig. 3.19 shows the Karnaugh maps for the increment circuitry, along with the optimal coverings. Note that not all of the coverings that eliminate the dual-rail input, *Inc*, are fully shown, so as not to clutter the drawing. The minimal SOP equations are derived directly from the K-map coverings as shown in Fig. 3.20.

$$
\begin{aligned}
S_0^0 &= Inc^0X_0^0 + Inc^1X_0^3 = Inc^0X_0^0 + Inc^1X_0^3 + Inc^0Inc^1 + X^0X^3 \rightarrow \text{TH24comp} \\
S_0^1 &= Inc^0X_0^1 + Inc^1X_0^0 = Inc^0X_0^1 + Inc^1X_0^0 + Inc^0Inc^1 + X^0X^1 \rightarrow \text{TH24comp} \\
S_0^2 &= Inc^0X_0^2 + Inc^1X_0^1 = Inc^0X_0^2 + Inc^1X_0^1 + Inc^0Inc^1 + X^1X^2 \rightarrow \text{TH24comp} \\
S_0^3 &= Inc^0X_0^3 + Inc^1X_0^2 = Inc^0X_0^3 + Inc^1X_0^2 + Inc^0Inc^1 + X^2X^3 \rightarrow \text{TH24comp} \\
S_1^0 &= Inc^0X_1^0 + X_0^0X_1^0 + X_0^1X_1^0 + X_0^2X_1^0 + Inc^1X_0^3X_1^3 \\
&= X_1^0 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^3 \bullet (Inc^1X_0^3) \\
&= X_1^0 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^3 \bullet (Inc^1X_0^3) \\
&\quad + (Inc^0 + X_0^0 + X_0^1 + X_0^2) \bullet (Inc^1X_0^3) + X_1^0X_1^3 \\
S_1^1 &= Inc^0X_1^1 + X_0^0X_1^1 + X_0^1X_1^1 + X_0^2X_1^1 + Inc^1X_0^3X_1^0 \\
&= X_1^1 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^0 \bullet (Inc^1X_0^3) \\
&= X_1^1 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^0 \bullet (Inc^1X_0^3) \\
&\quad + (Inc^0 + X_0^0 + X_0^1 + X_0^2) \bullet (Inc^1X_0^3) + X_1^0X_1^1 \\
S_1^2 &= Inc^0X_1^2 + X_0^0X_1^2 + X_0^1X_1^2 + X_0^2X_1^2 + Inc^1X_0^3X_1^1 \\
&= X_1^2 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^1 \bullet (Inc^1X_0^3) \\
&= X_1^2 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^1 \bullet (Inc^1X_0^3) \\
&\quad + (Inc^0 + X_0^0 + X_0^1 + X_0^2) \bullet (Inc^1X_0^3) + X_1^1X_1^2 \\
S_1^3 &= Inc^0X_1^3 + X_0^0X_1^3 + X_0^1X_1^3 + X_0^2X_1^3 + Inc^1X_0^3X_1^2 \\
&= X_1^3 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^2 \bullet (Inc^1X_0^3) \\
&= X_1^3 \bullet (Inc^0 + X_0^0 + X_0^1 + X_0^2) + X_1^2 \bullet (Inc^1X_0^3) \\
&\quad + (Inc^0 + X_0^0 + X_0^1 + X_0^2) \bullet (Inc^1X_0^3) + X_1^2X_1^3
\end{aligned}
$$

Figure 3.20: Quad-rail increment circuitry equations.

S_0 is input-complete with respect to Inc and X_0 since these appear in all S_0 product terms, and S_1 is input-complete with respect to X_1 since it appears in all S_1 product terms. Therefore, the circuit is input-complete with respect to all inputs. Furthermore, this circuit is inherently input-complete since it is impossible to determine the value of S without knowing the value of both X and Inc. The equations for all rails of S_0 each directly map to a TH24comp gate, after adding two *don't care* terms. The equations for all S_1 rails contain two like terms, $(Inc^0 + X_0^0 + X_0^1 + X_0^2)$ and $(Inc^1X_0^3)$, such that these can each be implemented using a single gate, and reused for all S_1 rails, as shown in the resulting optimized circuit in Fig. 3.21. Note that these two terms are mutually exclusive, such that their product can be added as a *don't care* term, along with the product of two X_1 rails, as shown in the modified S_1 equations, such that each rail of S_1 maps to a TH24comp gate.

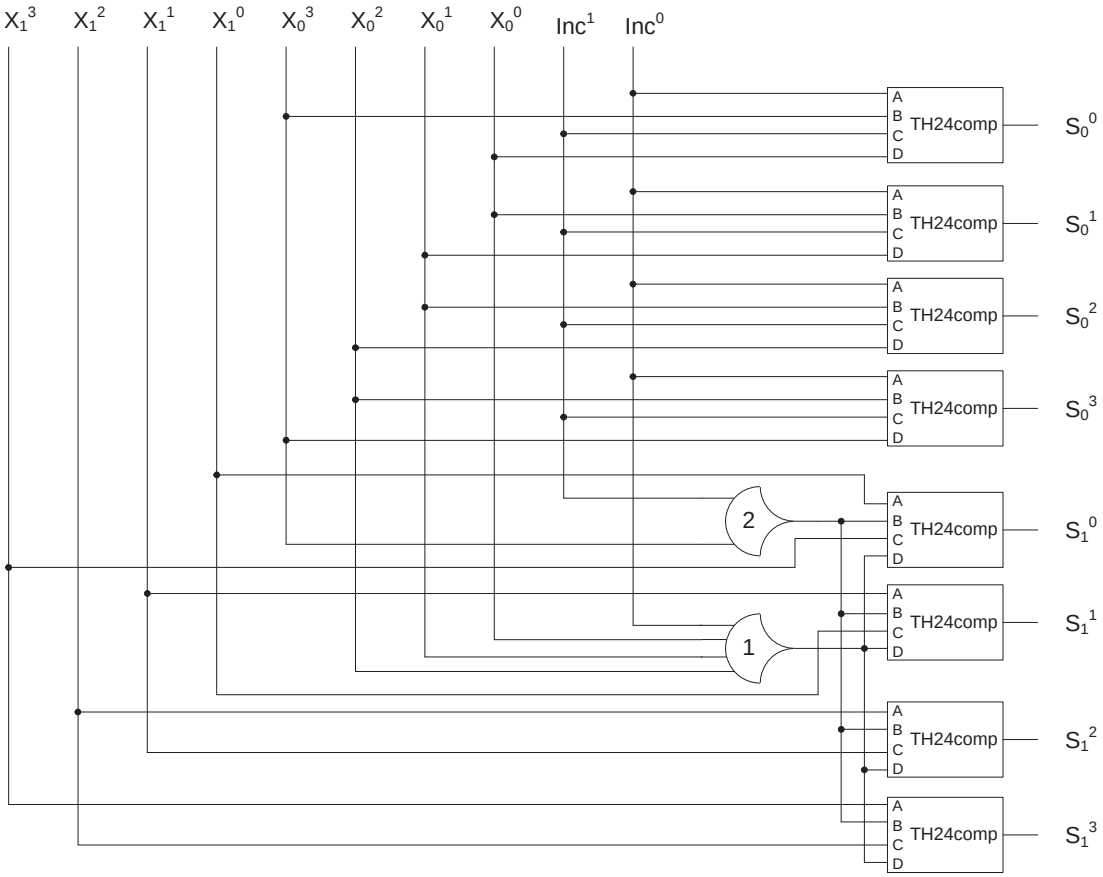


Figure 3.21: Optimized quad-rail increment circuitry.

CHAPTER 4

Sequential NCL Circuit Design

The outputs of sequential circuits depend not only on the current inputs, but also on the past inputs; therefore, they utilize memory to store the current *state*, which is used to generate the current outputs, and is fed back to generate the next state. Sequential circuits can be represented as Finite State Machines (FSMs), depicted in Fig. 4.1, where the State Memory is implemented using registers and the Next-State and Output circuits are combinational logic. FSMs include Mealy and Moore machines for simple circuits, and Algorithmic State Machines (ASMs) for more complex circuits. FSM design and optimization is beyond the scope of this book, but is detailed in [1, 2].

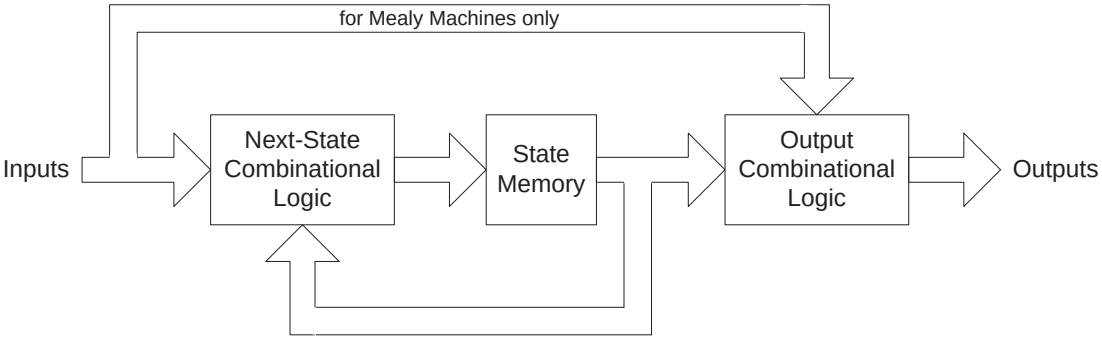


Figure 4.1: Finite State Machine block diagram.

4.1 NCL IMPLEMENTATION OF MEALY AND MOORE MACHINES

Take, for example, the design of a non-resetting sequence detector to identify an active high or active low pulse (i.e., 010 or 101, respectively) on input X and assert output Z when detected. Note that resetting in this context means that after a sequence is detected, the subsequent input is treated like the very first input; whereas for a non-resetting design, after a sequence is detected, the previous inputs are retained and used along with the subsequent input to potentially detect another sequence. Therefore, the sequence 0010101110100 generates an output of 0001111000110 for the non-resetting sequence detector and 0001001000100 for a resetting version. Fig. 4.2 shows the minimal state diagram, designed using the methods in [1]. After utilizing the Armstrong-Humphrey rules to generate a good state assignment, $Q_A Q_B Q_C$ (i.e., $S0 = 100$, $S1 = 000$, $S2 = 011$, $S3 = 010$, and $S4 = 001$),

the following minimal next-state and output equations can be derived from the K-maps shown in Fig. 4.3, as explained in [1]: $D_A = 0$; $D_B = Q_A' \bullet (X \oplus Q_C)$; $D_C = X$; $Z = Q_B \bullet (X \oplus Q_C)$. From these equations, the synchronous state machine can be directly implemented as shown in Fig. 4.4.

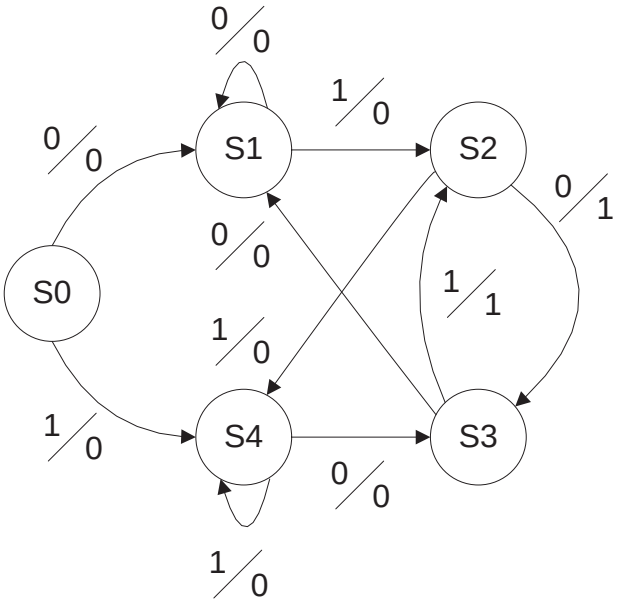


Figure 4.2: Minimal state diagram for non-resetting Mealy machine to detect 010 or 101.

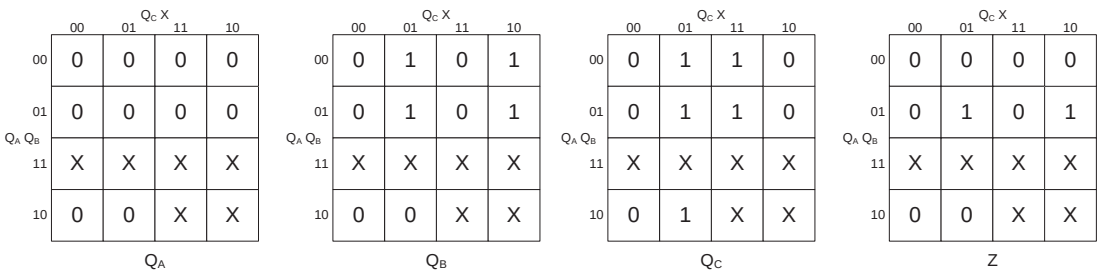


Figure 4.3: Mealy machine K-maps.

To convert the design to NCL, the XOR and AND Boolean gates can be directly replaced with their respective input-complete NCL functions, as given in Section 3.2 and the D-type flip-flops replaced with a three-stage NCL register, as shown in Fig. 4.5. Note that a minimum of three NCL

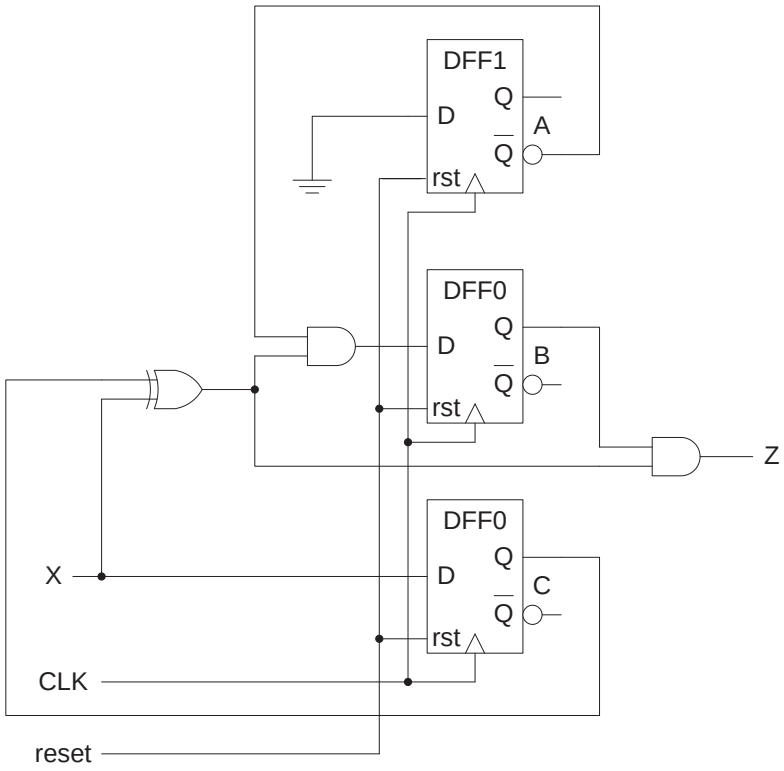


Figure 4.4: Synchronous Mealy machine implementation.

registers are required in a feedback loop to allow the DATA and NULL wavefronts to propagate; however, adding a fourth register increases throughput (4% in this case for static CMOS gates implemented with a 1.8V 180nm process), since this allows the DATA and NULL wavefronts to propagate more independently. To optimize the design, input-incomplete AND/NAND/OR/NOR functions should be used whenever possible. These input-incomplete functions only require 2 gates and 1 gate delay for up to 4-input functions (i.e., for an n-input function, one THnn and one TH1n gate are required). Whereas input-complete Boolean functions requiring more than 2 inputs must be decomposed into input-complete 2-input Boolean functions in order to retain input-completeness and, therefore, delay-insensitivity (e.g., an input-complete 4-input AND function will require 3 input-complete 2-input AND functions, totaling 6 gates and 2 gate delays).

In this case, the inherently input-complete XOR function is input-complete with respect to X and Q_C, such that using an input-complete AND function to generate Z makes Z input-complete with respect to X, Q_B, and Q_C. D_A is a constant zero; however, NCL circuits alternate between DATA and NULL, so a DATA0 must be generated for D_A every DATA cycle. This can be done by

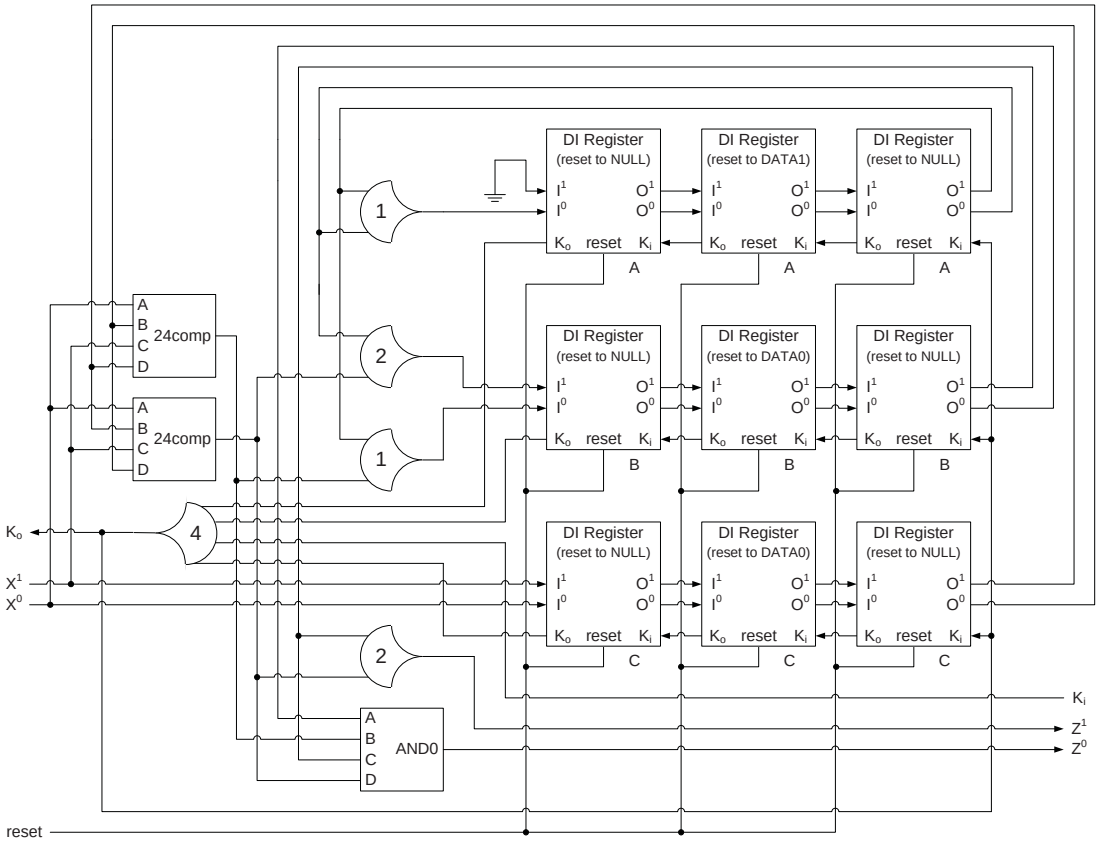


Figure 4.5: NCL Mealy machine implementation using Boolean functions.

making $D_A^1 = 0$ and $D_A^0 = Q_A^0 + Q_A^1$, such that D_A is input-complete with respect to Q_A; hence, the circuit as a whole is already input-complete, such that an input-incomplete AND function can be used to generate D_B, as shown in Fig. 4.5. Using this input-incomplete AND function increases throughput by 5.3% (comparing 4-register versions) and requires 13 fewer transistors.

Alternatively, the next-state and output equations can be derived directly in dual-rail form from the K-maps, as detailed in Section 3.2. The dual-rail equations are: $D_A^0 = 1 = Q_A^0 + Q_A^1$, $D_A^1 = 0$; $D_B^0 = X^0 Q_C^0 + X^1 Q_C^1 + Q_A^1$, $D_B^1 = X^1 Q_A^0 Q_C^0 + X^0 Q_C^1$; $D_C^0 = X^0$, $D_C^1 = X^1$; $Z^0 = Q_B^0 + X^0 Q_C^0 + X^1 Q_C^1$, $Z^1 = X^1 Q_B^1 Q_C^0 + X^0 Q_B^1 Q_C^1$. D_A is input-complete with respect to Q_A; D_C is input-complete with respect to X; D_B can be modified to be input-complete with respect to Q_C by making $D_B^0 = X^0 Q_C^0 + X^1 Q_C^1 + Q_A^1 Q_C^0$; and Z can be modified to be input-complete with respect to Q_B by making $Z^0 = Q_B^0 + X^0 Q_B^1 Q_C^0 + X^1 Q_B^1 Q_C^1$. The resulting input-complete design is shown in Fig. 4.6.

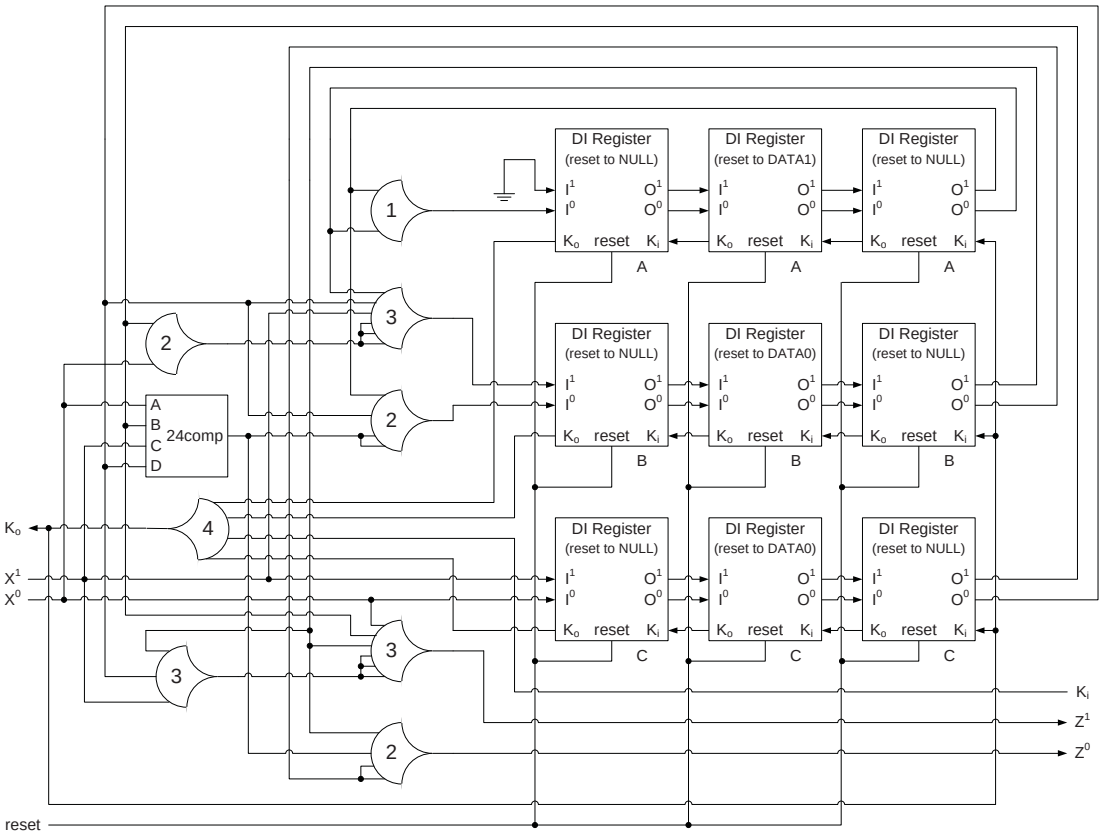


Figure 4.6: NCL Mealy machine implementation using dual-rail equations.

The next-state and output equations can also be derived in quad-rail form, as detailed in Section 3.3. This circuit consists of a single bit input, X, a single bit output, Z, and a 3-bit state variable $Q_A Q_B Q_C$; hence, quad-rail signals cannot be used for the input and output, but the internal state variable can be represented as a dual-rail signal and a quad-rail signal. Since D_A is a constant zero, Q_B and Q_C are combined into a single quad-rail signal, Q_{BC}. The quad-rail equations derived from the Karnaugh maps shown in Fig. 4.7 are: $D_A^0 = 1 = Q_A^0 + Q_A^1$, $D_A^1 = 0$; $D_{BC}^0 = X^0 Q_{BC}^0 + X^0 Q_{BC}^2$, $D_{BC}^1 = X^1 Q_A^1 + X^1 Q_{BC}^1 + X^1 Q_{BC}^3$; $D_{BC}^2 = X^0 Q_{BC}^1 + X^0 Q_{BC}^3$, $D_{BC}^3 = X^1 Q_A^0 Q_{BC}^0 + X^1 Q_{BC}^2$; $Z^0 = Q_{BC}^0 + Q_{BC}^1 + X^0 Q_{BC}^2 + X^1 Q_{BC}^3$, $Z^1 = X^0 Q_{BC}^3 + X^1 Q_{BC}^2$. D_A is input-complete with respect to Q_A; D_{BC} is input-complete with respect to X; and Z is input-complete with respect to Q_{BC}. The resulting input-complete design is shown in Fig. 4.8.

Table 4.1 summarizes the Mealy machine results for the different 4-register versions using static CMOS gates implemented with a 1.8V 180nm process. Note that the dual-rail and quad-rail

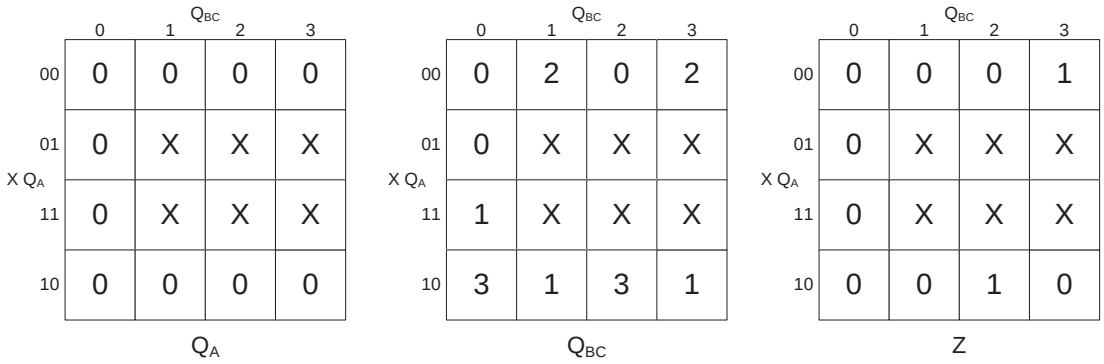


Figure 4.7: Mealy machine quad-rail K-maps.

registration and completion logic require approximately the same area (i.e., 446 transistors for dual-rail vs. 444 transistors for quad-rail). The Optimized Boolean Function version required the least area, whereas the Quad-Rail Optimized version was fastest, even though all designs had a worse-case combinational delay of 2 gates. The Quad-Rail Optimized version will also utilize the least amount of energy per operation because it only requires 3-4 gates to switch in the combinational logic (C/L) and 17 in the registration and completion logic, per operation; whereas the Optimized Boolean Function version always requires 4 gates to switch in the combinational logic, the Dual-Rail Optimized version requires 3-5 gates to switch in the combinational logic, and both dual-rail versions require 25 gates to switch in the registration and completion logic, per operation.

Table 4.1: Mealy machine design comparison.

Design	# C/L Gates	# C/L Transistors	C/L Delay (gates)	T_{DD} (ns)
Optimized Boolean Function	7	91	2	2.27
Dual-Rail Optimized	8	116	2	2.31
Quad-Rail Optimized	8	111	2	1.87

In general, a dual-rail optimized design will usually outperform its optimized Boolean function version in all aspects (i.e., area, speed, and power), especially when many input-complete Boolean functions requiring more than 2 inputs are needed. However, there is no clear winner for dual-rail vs. quad-rail since design differences between the two are highly circuit dependant. Quad-rail circuits usually require less energy per operation because a quad-rail signal only requires one wire to switch per transition from NULL to DATA and vice versa, whereas 2 wires must switch for the equivalent

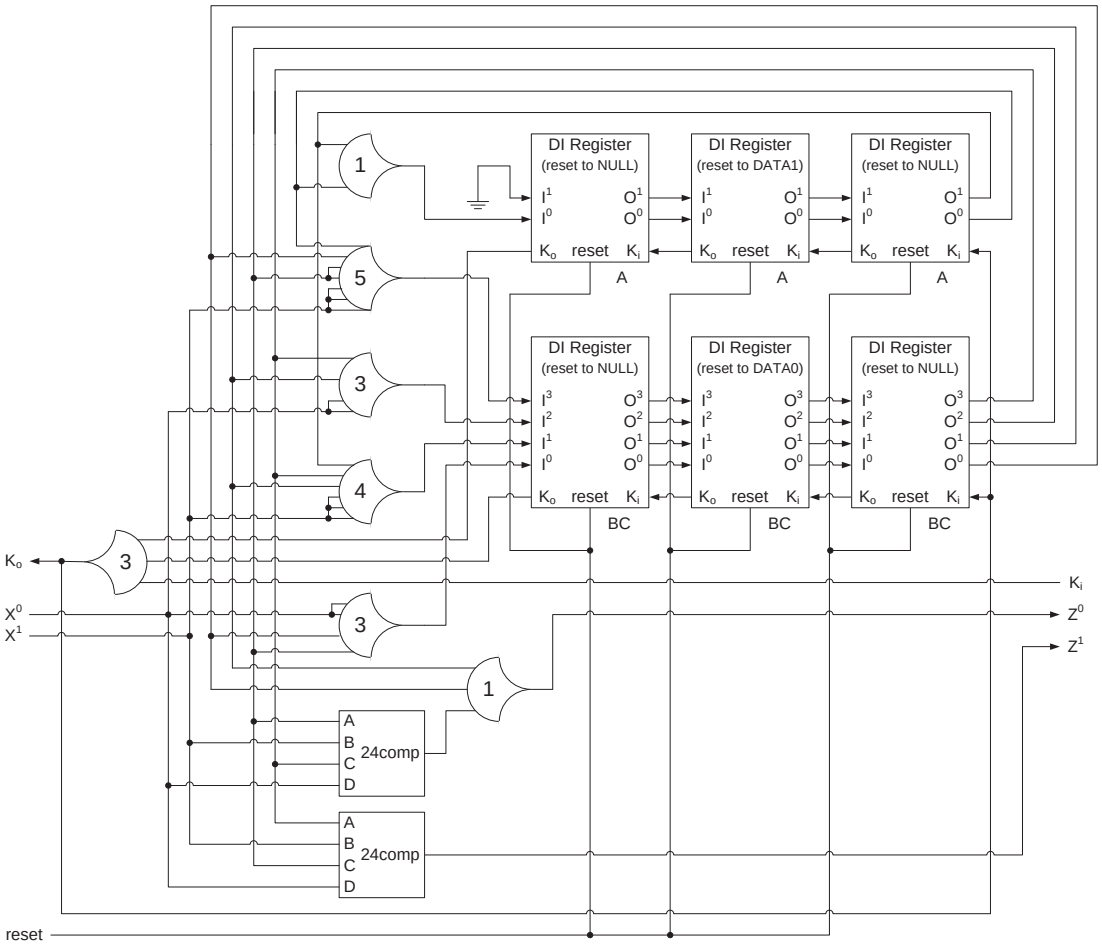


Figure 4.8: NCL Mealy machine implementation using quad-rail equations.

2 dual-rail signals. However, if the quad-rail implementation requires substantially more logic, the dual-rail version could utilize less energy.

4.2 NCL IMPLEMENTATION OF ALGORITHMIC STATE MACHINES

Algorithmic State Machines (ASMs) are used to design complex sequential circuits, and normally consist of a datapath controlled by an ASM. ASMs utilize complex state transition conditions, such as $A(7:0) > B(7:0)$, whereas Mealy and Moore machines only use the value of one or few bits

to determine state transition since Mealy and Moore machine complexity grows exponentially with number of state transition bits. Fig. 4.9 shows the interface, ASM, and corresponding datapath for a Greatest Common Divisor circuit. The numerical inputs are two 8-bit unsigned numbers, A and B; the numerical output is the 8-bit Greatest Common Divisor (GCD) of A and B, Y. The circuit also has a *reset* and *clk* input and input/output handshaking signals, following the One Cycle Demand Driven Convention (OCDDC), as shown in Fig. 4.9(a). The OCDDC uses *rqst* and *dat* bits along with an input or output to ensure that the input/output is valid before loading/outputting the corresponding data. *rqst* is asserted to signify that the receiver is ready for new data, after which *dat* is asserted (signifying valid data) for one rising *clk* edge (either the immediately following edge or any subsequent edge), where the data is latched. *rqst* is then deasserted, or can remain asserted to request another data. Additionally, at the rising edge of *clk* when *reset* is asserted, the circuit should reset to its initial state.

The GCD algorithm continually subtracts the smaller of A or B from the larger, storing the result in the larger, until both are the same. This is the GCD.

Example:	A:	15	5	5
	B:	10	10	5
Calculation:		$A - B$	$B - A$	$A = B \rightarrow 5$ is GCD

ASM design and optimization is beyond the scope of this book, but is detailed in [2]. The C/L datapath components can be designed using the techniques presented in Chapter 3. Any sequential datapath components, such as registers and counters, can be designed as C/L with 3- or 4-register feedback, as demonstrated for the quad-rail counter in Section 3.3. Minimal next-state and output equations can then be derived for the ASM, following the methods detailed in [2], and implemented as explained for the Mealy machine in Section 4.1. The optimized GCD circuit will be designed as a comprehensive example in Chapter 7.

BIBLIOGRAPHY

[1] Allen Dewey, *Analysis and Design of Digital Systems with VHDL*, PWS Publishing Company, 1997.

[2] Justin Davis and Robert Reese, *Finite State Machine Datapath Design, Optimization, and Implementation*, Morgan & Claypool Publishers, 2008.
DOI: 10.2200/S00087ED1V01Y200702DCS014

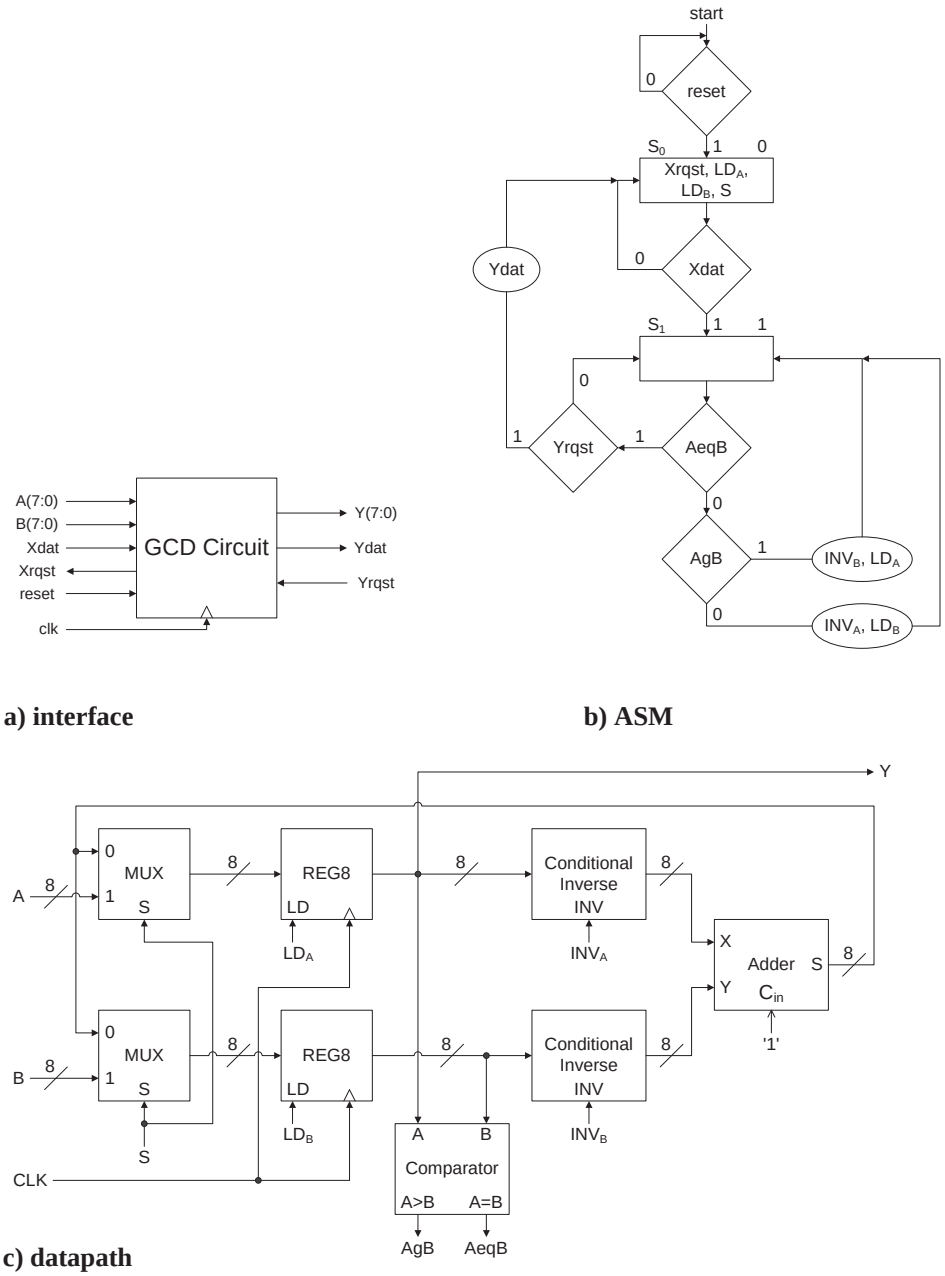


Figure 4.9: GCD circuit. (a) interface; (b) ASM; (c) datapath.

CHAPTER 5

NCL Throughput Optimization

There are a number of techniques that can be used to increase throughput of NCL systems, such as Pipelining, Embedded Registration, Early Completion, and NULL Cycle Reduction.

5.1 PIPELINING

NCL systems can be optimized for speed by partitioning the combinational circuitry and inserting additional NCL registers and corresponding completion components. However, NCL circuits cannot be partitioned arbitrarily; they can only be divided at component boundaries in order to preserve delay-insensitivity. The average cycle time for an NCL system, T_{DD}, can be estimated as the worse-case stage delay of any stage in the pipeline, where the delay of one stage is equal to twice the sum of the stage's worse-case combinational delay and completion delay, to account for both the DATA and NULL wavefronts. Algorithm 1 depicts this calculation for an N-stage pipeline, where $Dcomb_i$ and $Dcomp_i$ are stage$_i$'s combinational and completion delays, respectively.

```
T_DDmax = 2 × (Dcomb_1 + Dcomp_1)
for (i = 2 to N) loop
     T_DDtemp = 2 × (Dcomb_i + Dcomp_i)
     T_DDmax = MAX(T_DDtemp, T_DDmax)
end loop
```

Algorithm 5.1: NCL T_{DD} estimation.

NCL pipelining can utilize either of two completion strategies: full-word or bit-wise completion. Full-word completion, as shown in Fig. 5.1, requires that the acknowledge signals from each bit in register$_i$ be conjoined together by the completion component, whose single-bit output is connected to all request lines of register$_{i-1}$. On the other hand, bit-wise completion, as shown in Fig. 5.2, only sends the completion signal from bit b in register$_i$ back to the bits in register$_{i-1}$ that took part in the calculation of bit b. This method may, therefore, require fewer logic levels than that of full-word completion, thus increasing throughput. In this example, bit-wise completion is faster (i.e., 1 gate delay vs. 2 gate delays), but it requires more area (i.e., 4 gates vs. 2 gates).

To maximize throughput while minimizing latency and area, the following algorithm should be used to optimally partition an NCL circuit. Steps 1 and 2 initially partition an NCL circuit into stages of *primary components*, where a primary component is defined as a component whose inputs only consist of the circuit's inputs, or outputs of components that have already been added to a

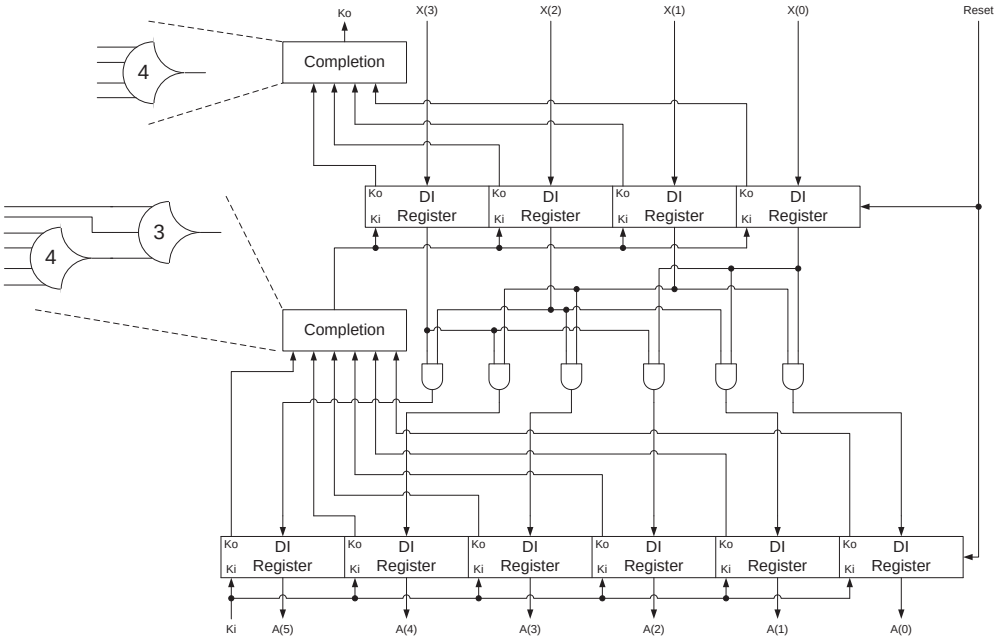


Figure 5.1: Full-word completion.

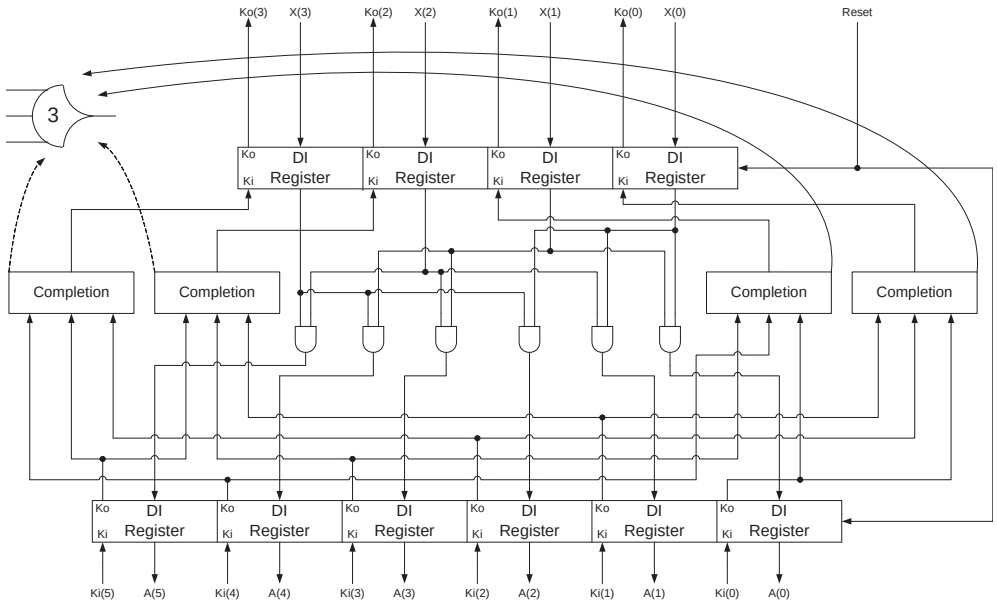


Figure 5.2: Bit-wise completion.

previous stage. Steps 3 and 4 then calculate the combinational delay (i.e., *Dcomb*) and completion delay (i.e., *Dcomp*) for each stage and the maximum delay for the entire pipeline (i.e., *max_delay*), utilizing both full-word and bit-wise completion strategies. Finally, Step 5 merges stages to reduce latency and area, as long as doing so does not decrease throughput. Note that when merging stages the new merged combinational delay (i.e., *merged_comb*) is not necessarily $Dcomb_i + Dcomb_{i+1}$. Take, for example, two full adders in a ripple-carry adder: $Dcomb_i = 2$ and $Dcomb_{i+1} = 2$, but *merged_comb* = 3, since the *carry* output of a full adder has only 1 gate delay.

As an example, the non-pipelined quad-rail multiplier in Fig. 5.3 has a worse-case combinational delay of 8 and a completion delay of 1, such that $T_{DD} = 18$. Applying Steps 1-4 of the pipelining algorithm to the quad-rail multiplier yields the results shown in Tables 5.1 and 5.2 for full-word and bit-wise completion, respectively. These tables show that the full-word pipelined design has a T_{DD} (i.e. $2 \times$ *max_delay*, to account for both the DATA and NULL wavefronts) of 10 gate delays, while the bit-wise pipelined design has a T_{DD} of 8 gate delays; hence, the bit-wise pipelined design is preferred, since it maximizes throughput. Applying Step 5 of the algorithm to merge stages for both full-word and bit-wise completion results in both pipelined designs merging Stages 3 and 4, such that both designs only require 3 stages. The new D_{comb} is 3 and the new stage delay for both designs is 4. Note that *max_outputs* for the bit-wise design changes to 2 for the merged stage, such that D_{comp} becomes 1.

5.2 EMBEDDED REGISTRATION

Embedded registration merges delay-insensitive registers into the combinational logic, when possible, to increase throughput and decrease latency and area. Take, for example, an input-complete 2:1 MUX with output register, as shown in Fig. 5.4(a). The DI register can be integrated into the combinational logic by making K_i a third input to the TH22 MUX output gates, and adjusting these gates' thresholds and types, accordingly, and making them resettable, as shown in Fig. 5.4(b).

Embedded registration can also be applied to the input-incomplete 2:1 MUX, shown in Fig. 5.5(a), in either of two ways, as shown in Figs. 5.5(b) and 5.5(c). Method 1, as shown in Fig. 5.5(b), is the same as used for the input-complete 2:1 MUX, where the DI register is integrated into the combinational logic by making K_i another input to the output gates, and adjusting the output gates' thresholds and types accordingly, and making them resettable. Alternatively, when the output gates are TH1n gates, it is often advantageous to make K_i another input to all gates preceding the TH1n gates, and adjusting the preceding gates' thresholds and types accordingly, and making them resettable, as shown in Fig. 5.5(c).

Method 2 often necessitates a smaller gate library, as in this case, requiring standard resettable TH33 gates, whereas Method 1 requires non-standard resettable TH33w2 gates. The standard NCL gate library includes up to 4-input gates, as provided in Table 2.4, as well as resettable TH22 and TH33 gates, and inverting TH12, TH13, and TH14 gates. Hence, any NCL gate in the standard library has a maximum of 4 transistors in series (i.e., all 4-input gates and resettable TH33 gates).

```
1) i = 1
2) loop until all components are part of a stage
                        -- initially partition into stages
     add all primary components to stage_i
     i = i + 1
   end loop
3) N = i -1             -- stage_N is final stage
   max_delay_FW = 0
   max_delay_BW = 0
4) for j in 1 to N loop -- calculate worse-case cycle times
     Dcomb = max delay of stage_i's components
                        -- for both full-word and bit-wise completion
     B = # of outputs from stage_j
     Dcomp_j = ⌈Log_4 B⌉
     if ((Dcomb + Dcomp_j) > max_delay_FW) then
        max_delay_FW = (Dcomb + Dcomp_j)
     end if
     B = # of inputs to stage_j
     max_outputs = 1
     for i in 1 to B loop
          num_outputs = number of outputs of stage_j generated by input_i
          if (num_outputs > max_outputs) then
              max_outputs = num_outputs
          end if
     end loop
     Dcomp = ⌈Log_4 max_outputs⌉
     if ((Dcomb + Dcomp) > max_delay_BW) then
          max_delay_BW = (Dcomb + Dcomp)
     end if
   end loop
```

Algorithm 5.2: NCL pipelining algorithm (*continues*).

```
5) if (max_delay_FW > max_delay_BW) then
                        -- bit-wise design is faster
     num_stages = call merge_BW function
     output bit-wise pipelined design
   elsif (max_delay_BW > max_delay_FW) then
                        -- full-word design is faster
     num_stages = call merge_FW function
     output full-word pipelined design
   else
     num_stages_BW = call merge_BW function
     num_stages_FW = call merge_FW function
     if (num_stages_BW > num_stages_FW) then
                        -- full-word design has less latency
        output full-word pipelined design
     elsif (num_stages_FW > num_stages_BW) then
                        -- bit-wise design has less latency
        output bit-wise pipelined design
     elsif (area of full-word design
                        > area of bit-wise design) then
        output bit-wise pipelined design
                        -- bit-wise design is smaller
     else
        output full-word pipelined design
                        -- full-word design is smaller
     end if
   end if
                         merge_FW function
   num_stages = N
   for k in 1 to N-1 loop    -- merge stages to decrease latency
     merged_comb = max combinational delay of stage_k
                        and stage_k+1 merged into a single stage
     if ((merged_comb + comp_k+1) = max_delay_FW) then
       merge stage_k into stage_k+1
       delete stage_k
       num_stages = num_stages - 1
     end if
   end loop
   return num_stages
```

Algorithm 5.2: (*continued*) NCL pipelining algorithm (*continues*).

```
                        merge_BW function
num_stages = N
for k in 1 to N-1 loop    -- merge stages to decrease latency
  merged_comb = max combinational delay of stage_k and
                    stage_k+1 merged into a single stage
  B = # of inputs to stage_k
  max_outputs = 1
  for i in 1 to B loop
    num_outputs = number of outputs of stage_k+1 generated by input_i
    if (num_outputs > max_outputs) then
       max_outputs = num_outputs
    end if
  end loop
  merged_comp = ⌈Log4 max_outputs⌉
  if ((merged_comb + merged_comp) = max_delay_BW) then
    merge stage_k into stage_k+1
    delete stage_k
    num_stages = num_stages − 1
  end if
end loop
return num_stages
```

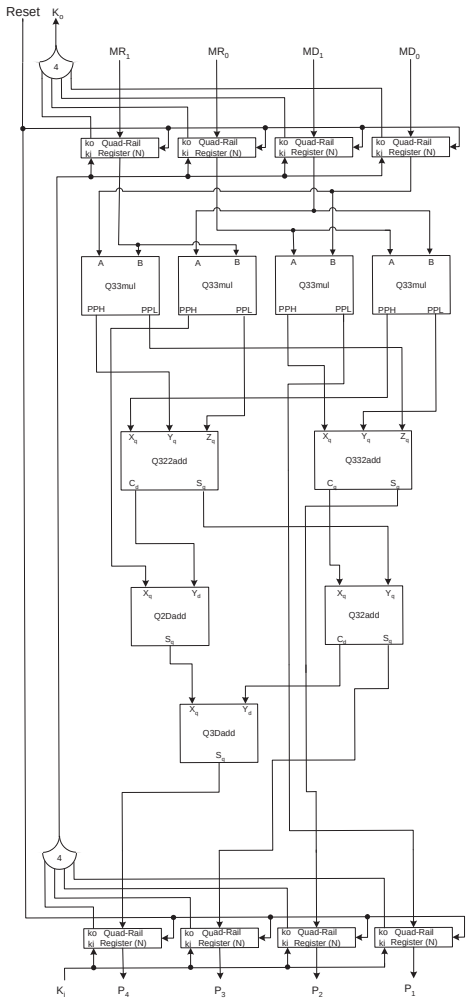
Algorithm 5.2: (*continued*) NCL pipelining algorithm.

Table 5.1: Full-word completion pipelining.

Stage	D_{comb}	# Outputs	D_{comp}	delay
1	2	8	2	4
2	3	6	2	5
3	2	5	2	4
4	1	4	1	2
			max_delay	5

5.3 EARLY COMPLETION

Early Completion performs the completion detection for a register at its input, instead of at the register output as in standard NCL, in order to significantly increase the throughput of NCL systems without impacting latency or compromising delay-insensitivity. An Early Completion register is shown in Fig. 5.6, compared to the regular DI register shown in Fig. 2.4. Early Completion requires that the inverted completion signal from Stage$_{i+1}$, Ko_{i+1}, be used as an additional input to the



Component Type	Output Gate Delays	
	Carry / PPH	Sum / PPL
Q33mul	1	2
Q332add	3	3
Q322add	2	3
Q32add	2	2
Q2Dadd	N/A	1
Q3Dadd	N/A	1

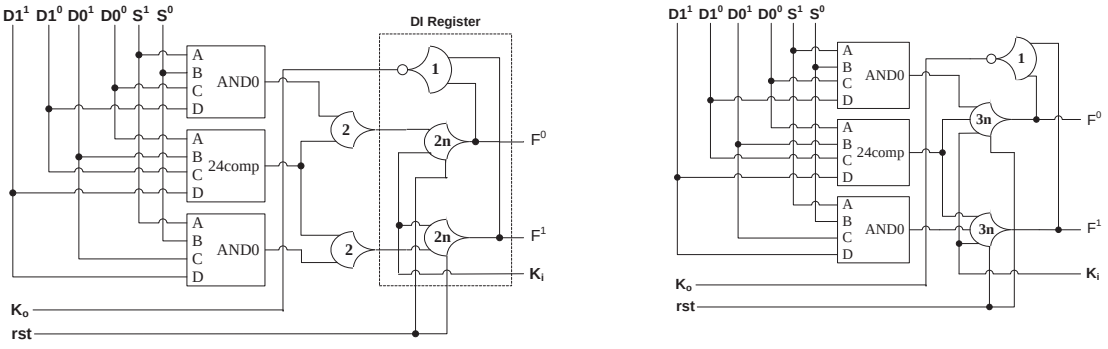
Figure 5.3: 4-bit × 4-bit unsigned quad-rail multiplier.

Completion tree for Stage$_i$, to maintain delay-insensitivity. The Early Completion component and pipeline are shown in Figs. 5.7 and 5.8, respectively, compared to the regular versions shown in Figs. 2.6 and 2.1, respectively. Note that the final gate of an Early Completion component must be resettable for proper initialization. If the stage's register is reset to NULL/DATA, its Early Completion component must be reset to *rfd/rfn* (i.e., logic 1 / logic 0), respectively.

The Early Completion component for Stage$_i$ requests DATA/NULL when all inputs to Register$_i$ are NULL/DATA and Ko_{i+1} is *rfn/rfd*, respectively. Note that the Early Completion

Table 5.2: Bit-wise completion pipelining.

Stage	D_{comb}	*max_outputs*	D_{comp}	delay
1	2	4	1	3
2	3	2	1	4
3	2	2	1	3
4	1	1	0	1
			max_delay	4



a) with output register **b) with embedded registration**

Figure 5.4: Input-complete 2:1 MUX. (a) with output register; (b) with embedded registration.

component for the final stage, Stage_M, is slightly different, requiring the inverter for the external Ko_{M+1} signal (i.e., K_i) to be removed. This causes the Stage_M Early Completion component to request DATA/NULL when the input to Register_M is NULL/DATA and K_i is *rfd/rfn*, respectively. This variation in the Early Completion component for the last stage is required since K_i may change to *rfn/rfd* as soon as the output is DATA/NULL, respectively, assuming a zero delay external interface. An alternative is to use the standard completion component, shown in Fig. 2.6, for Stage_M. However, this later approach produces a system with reduced throughput compared to that when the modified Early Completion component is used for the last stage. In a real system, there will be some delay between when the outputs change and when K_i subsequently changes, such that modifying the Early Completion component for the last stage may not be necessary.

Early Completion reduces handshaking overhead by allowing the Completion evaluation for Stage_i to begin before the DATA/NULL wavefront has been latched by Register_i. Early Completion does not impact latency since the forward path is unchanged. However, delay-insensitivity must be reanalyzed. In the most delay-sensitive case, Ko_i and Ko_{i+1} are both *rfd/rfn* and all bits at the input of Register_{i-1} change to DATA/NULL, respectively, within a very short period of time.

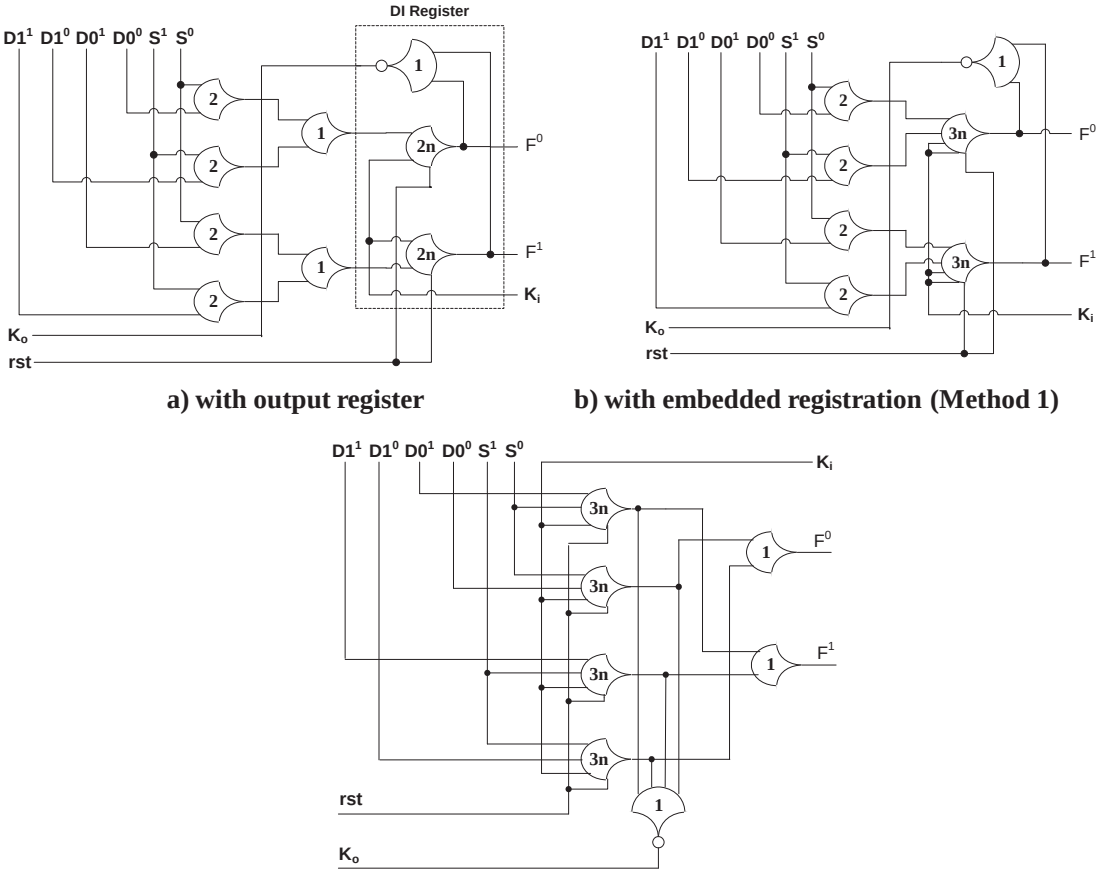


Figure 5.5: Input-incomplete 2:1 MUX. (a) with output register; (b) with embedded registration (Method 1); (c) with embedded registration (Method 2).

The DATA/NULL wavefront at the input of Register$_{i-1}$ flows through Register$_{i-1}$, followed by Combinational Circuit$_i$, and finally Early Completion Component$_i$, in order to transition Ko_i to *rfn/rfd*, respectively. Simultaneously, the DATA/NULL wavefront at the input of Register$_{i-1}$ flows through Early Completion Component$_{i-1}$ in order to transition Ko_{i-1} to *rfn/rfd*, respectively. Therefore, in order for the system to function incorrectly, the DATA/NULL wavefront would have to travel through a set of TH22 gates (i.e., Register$_{i-1}$), Combinational Circuit$_i$, and Early Completion Component$_i$, before the same signal traveled through only Early Completion Component$_{i-1}$. Since the first path is normally much longer, the delay is well known and the system remains self-timed.

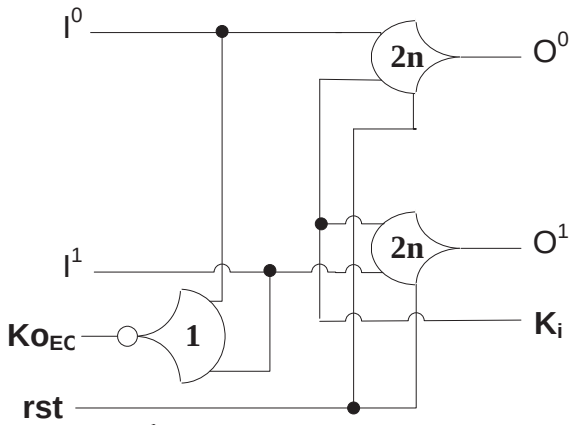


Figure 5.6: Early Completion DI register.

For the special case of a FIFO, the combinational logic delay would be zero, but the delay through Early Completion Component$_i$ and Early Completion Component$_{i-1}$ would be identical, so the above argument would still hold. For the generalized case, Early Completion Component$_i$ and Early Completion Component$_{i-1}$ normally have about the same delay, within one or two gate delays, such that the above analysis holds true.

The other delay-sensitive scenario introduced by Early Completion is when Ko_{i+1} changes to *rfd/rfn* when all inputs to Register$_i$ are already DATA/NULL and all inputs to Register$_{i-1}$ are NULL/DATA, respectively. In this case the *rfd/rfn* must pass through an inverter and one TH22 gate in order to transition Ko_i to *rfn/rfd*, respectively. Once Ko_i is *rfn/rfd*, the NULL/DATA wavefront at the input of Register$_{i-1}$ can flow through the register's TH22 gates and overwrite the previous DATA/NULL wavefront at the input of Register$_i$, respectively. Simultaneously, the DATA/NULL wavefront at the input of Register$_i$ must only pass through one TH22 gate to be latched at the output of Register$_i$. Therefore, in order for the system to function incorrectly, a signal would have to travel through both an inverter and two TH22 gates before the same signal travels through only a single TH22 gate. Since the path through the three gates is obviously longer than the path through a single gate, the delays are well known and the system remains self-timed. Note that this example assumes that there is no combinational logic delay, as would be the case for a FIFO. For the generalized case, the delay-sensitivity would be even less, since the path through an inverter, two TH22 gates, and combinational logic would have to be faster than the path through a single TH22 gate in order to adversely affect self-timed operation.

As an example, Early Completion was applied to a full-word pipelined 4-bit×4-bit unsigned multiplier, yielding a 21% increase in throughput [1, 2]. Additionally, Early Completion must be

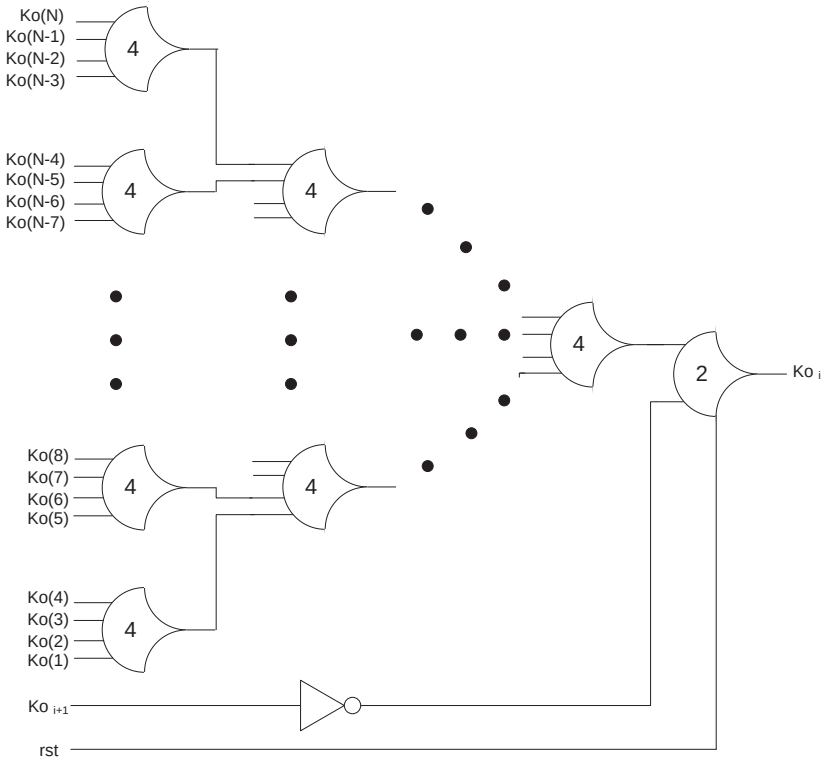


Figure 5.7: Early Completion component.

utilized when applying the ultra-low power NCL MTCMOS technique, explained in Section 6.2, in order to maintain delay-insensitivity.

5.4 NULL CYCLE REDUCTION

NCL system throughput can also be increased by applying the NULL Cycle Reduction (NCR) technique, depicted in Fig. 5.9, which increases the throughput of an NCL system by decreasing the circuit's NULL cycle time, without affecting its DATA cycle time. Successive input wavefronts are partitioned so that one circuit processes a DATA wavefront, while its duplicate processes a NULL wavefront. The first DATA/NULL cycle flows through the original circuit, while the next DATA/NULL cycle flows through the duplicate circuit. The outputs of the two circuits are then multiplexed to form a single output stream. NCR can be used to speedup slow stages in an NCL pipeline that cannot be further divided (e.g., Stage 2 in the quad-rail multiplier shown in Fig. 5.3). The application of NCR to only the slow stages in a pipeline increases the throughput for the entire pipeline [3]. NCR can also be used to increase the throughput of a feedback loop, which

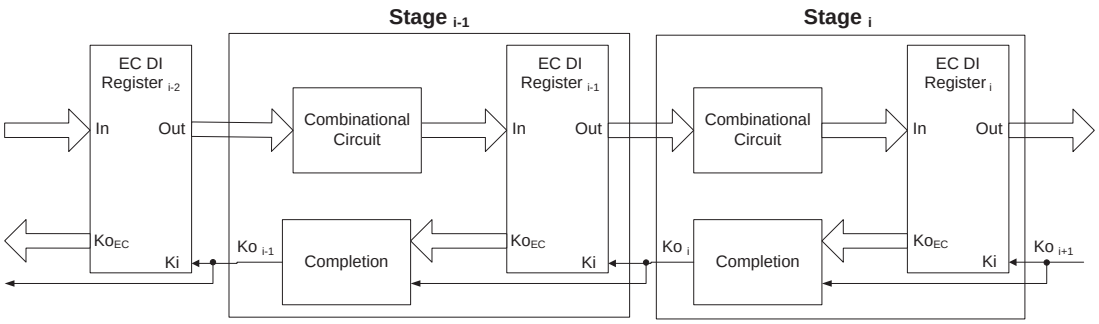


Figure 5.8: Early Completion pipeline.

cannot be increased by any other means, again increasing throughput for the entire pipeline [4]. Fig. 5.9 depicts the NCR architecture for a dual-rail logic circuit utilizing full-word completion; however, NCR is also applicable to quad-rail circuits and bit-wise completion. Quad-rail logic only requires a redesign of the Demultiplexer and Multiplexer circuits to handle quad-rail signals, whereas bit-wise completion requires removal of the Completion Detection component and replication of the Sequencer components, such that each input/output bit has its own Sequencer#1/Sequencer#2 component, respectively.

A *Sequencer* is an N-stage (at least 3 stages) single-rail ring structure consisting of resettable TH33 gates, used to generate an N-bit sequence that changes based on K_i. Here, $S1$ and $S2$ are taken directly from one of the sequencer taps to generate the desired sequences, 1000 and 0010, respectively; however, various sequencer taps may be ORed together to generate any arbitrary N-bit sequence. Sequencers can be used in lieu of state machines to generate the same repeated control signals, or in lieu of a counter, to repeatedly count a fixed number of events [5]. An N-stage sequencer contains $\lfloor (N-1)/2 \rfloor$ tokens, where a token is defined as a DATA wavefront with corresponding NULL wavefront, and one bubble for an odd N and two bubbles for an even N, where a bubble is defined as either a DATA or NULL wavefront occupying more than one neighboring stage [6]. When K_i becomes *rfd/rfn* the DATA/NULL wavefront moves through the one or two NULL/DATA bubbles ahead of it, creating one or two DATA/NULL bubbles in its wake, respectively. The DATA/NULL wavefront restricts the forward propagation of the NULL/DATA wavefront, respectively, for each change of K_i, limiting the forward propagation to only the one or two bubbles.

For the NCR architecture, *Sequencer #1* is controlled by the output of the Completion circuitry and is used to select either output A or B of the Demultiplexer. Upon reset, it selects output A to receive the first DATA/NULL cycle, after its K_i becomes *rfd*. It then selects output B to receive the second DATA/NULL cycle, and continuously alternates the DATA/NULL cycles between outputs A and B. *Sequencer #2* is controlled by the external request, K_i, and is used to allow DATA and NULL wavefronts to flow through the output register of Circuit #1 and Circuit #2. Upon reset,

it selects Circuit #1 to output the first DATA/NULL cycle, after K_i becomes *rfd*. It then selects Circuit #2 to output the second DATA/NULL cycle, and continuously alternates the DATA/NULL cycles between Circuit #1 and Circuit #2.

BIBLIOGRAPHY

[1] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "Delay-Insensitive Gate-Level Pipelining," *Elsevier's Integration, the VLSI Journal*, Vol. 30/2, pp. 103–131, October 2001. DOI: 10.1016/S0167-9260(01)00013-X

[2] S. C. Smith, "Speedup of Self-Timed Digital Systems Using Early Completion," *IEEE Computer Society Annual Symposium on VLSI*, pp. 107–113, April 2002. DOI: 10.1109/ISVLSI.2002.1016884

[3] S. C. Smith, "Speedup of NULL Convention Digital Circuits Using NULL Cycle Reduction," *Elsevier's Journal of Systems Architecture*, Vol. 52/7, pp. 411–422, July 2006. DOI: 10.1016/j.sysarc.2005.12.002

[4] S. C. Smith, "Development of a Large Word-Width High-Speed Asynchronous Multiply and Accumulate Unit," *Elsevier's Integration, the VLSI Journal*, Vol. 39/1, pp. 12–28, September 2005. DOI: 10.1016/j.vlsi.2004.11.001

[5] S. C. Smith, "Design of a NULL Convention Self-Timed Divider," *International Conference on VLSI*, pp. 447–453, June 2004.

[6] J. Sparso and J. Staunstrup, "Design and Performance Analysis of Delay Insensitive Multi-Ring Structures," *Twenty-Sixth Hawaii International Conference on System Sciences*, Vol. 1, pp. 349–358, 1993. DOI: 10.1016/0167-9260(93)90035-B

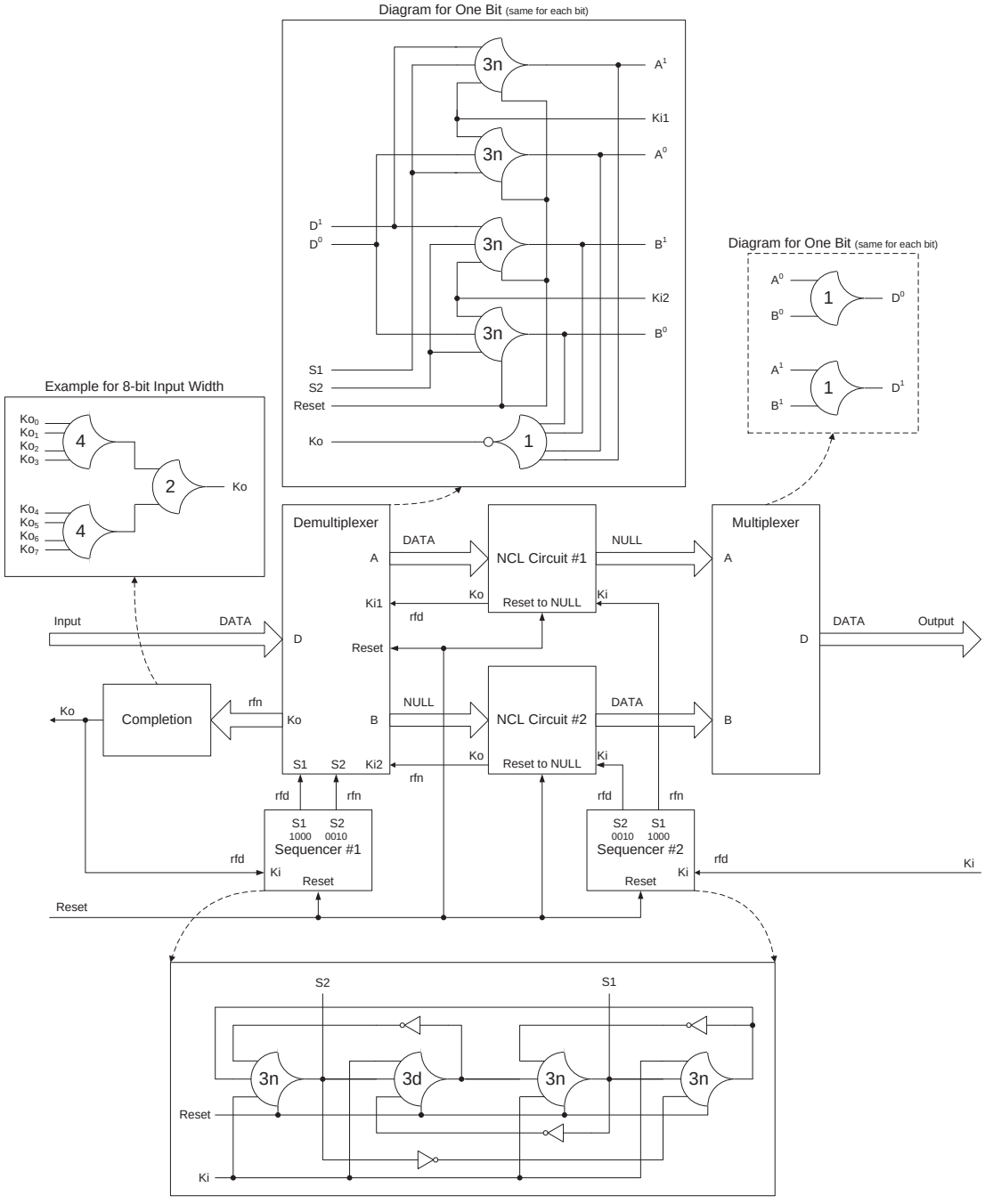


Figure 5.9: NCR architecture for a dual-rail circuit utilizing full-word completion.

CHAPTER 6

Low-Power NCL Design

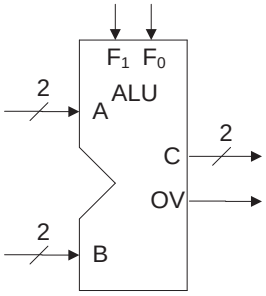
Delay-insensitive NCL circuits designed using CMOS exhibit an inherent idle behavior since they only switch when useful work is being performed, unlike clocked Boolean circuits that switch every clock pulse, unless specifically disabled through specialized circuitry, which itself requires additional area and power. Therefore, NCL systems inherently utilize significantly less power than their synchronous counterparts. Additionally, techniques such as wavefront steering, Multi-Threshold CMOS (MTCMOS), and supply voltage reduction can be applied to NCL systems in order to substantially further reduce energy usage.

6.1 WAVEFRONT STEERING

Wavefront steering is used to direct a DATA/NULL wavefront to flow through only the specific path needed for the selected operation, such that the alternative paths remain idle, and, therefore, utilize minimal energy. Wavefront steering also increases throughput by maintaining average-case delay.

Take, for example, the 4-operation Arithmetic Logic Unit (ALU), depicted in Fig. 6.1. The Boolean implementation, shown in Fig. 6.2, sends the input wavefront through all four functions, and utilizes a multiplexer at the output to select the desired function; whereas the NCL implementation, shown in Fig. 6.3, utilizes a Demultiplexer at the input to direct the input wavefront to only flow through the selected function. Since only a single function is switching, and the other three remain NULL, each rail of the four functions' outputs can simply be ORed together to generate the desired function output. Hence, all four functions switch every operation for the Boolean implementation; whereas only the selected function switches for the NCL implementation. Additionally, the clock period for the synchronous Boolean implementation, assuming input and output registers, will be derived from the worse-case delay through the ALU, which for N-bit operands is O(Log N) for a Carry-Lookahead Adder (CLA) implementation of the ADD operation. The other three operations (i.e., XOR, AND, and OR), which only require a delay of O(1) (i.e., 1 gate delay), must still wait the same amount of time as for the ADD operation before latching in the next set of operands to be processed, even though their result will be ready much sooner. The NCL implementation has an average-case delay of O(Log N) for a Ripple-Carry Adder (RCA) implementation of the ADD operation, and 1 gate delay for the XOR, AND, and OR operations. Therefore, when an XOR, AND, or OR operation is selected, the NCL implementation will produce the output and request and start processing the next input set much faster than for an ADD operation. Hence, the NCL implementation will be faster and require less energy than the Boolean implementation. The speed

advantage will increase proportionally to the operand size, N; and the energy advantage will increase proportionally to the number of ALU functions.



F_1	F_0	Function	Result
0	0	AND	C = A∧B, OV = 0
0	1	OR	C = A∨B, OV = 0
1	0	XOR	C = A⊕B, OV = 0
1	1	ADD	C = A + B, OV = 1 if A+B = 1XX else OV = 0

Figure 6.1: ALU block diagram and function table.

This wavefront steering NCL implementation is input-complete because the Demultiplexer output is input-complete with respect to all of the ALU inputs, A, B, and the function select bits, F_1 and F_0; and each of the four functions are input-complete with respect to the Demultiplexer output. The alternative NCL implementation, similar to the Boolean implementation, would be to send each input wavefront through all four functions and then utilize an input-complete 4:1 MUX at the output to select the desired function. This would require significantly more energy per operation since all four functions would switch every operation, instead of only the selected function. Additionally, speed would be substantially reduced since every operation would wait until all four functions completed before outputting the selected result and starting the next operation.

NCL circuits that utilize wavefront steering can be pipelined by: 1) embedding registration within the Demultiplexer, as explained in Section 5.2; 2) separately pipelining each function, as explained in Section 5.1; 3) adding a multi-stage pipeline for the function select bits, which are normally converted into a single MEAG, used to select the desired function pipeline to pass its output to the OR gates; and 4) adding an output register, which can be embedded within the output OR gates. Reference [1] details the design and subsequent pipelining of a 4-bit 8-operation ALU.

6.2 MULTI-THRESHOLD CMOS (MTCMOS) FOR NCL (MTNCL)

With the current trend of semiconductor devices scaling into the deep submicron region, design challenges that were previously minor issues have now become increasingly important. Where in the past, dynamic, switching power has been the predominant factor in CMOS digital circuit power dissipation, recently, with the dramatic decrease of supply and threshold voltages, a significant growth in leakage power demands new design methodologies for digital integrated circuits (ICs). The main component of leakage power is sub-threshold leakage, caused by current flowing through a transistor

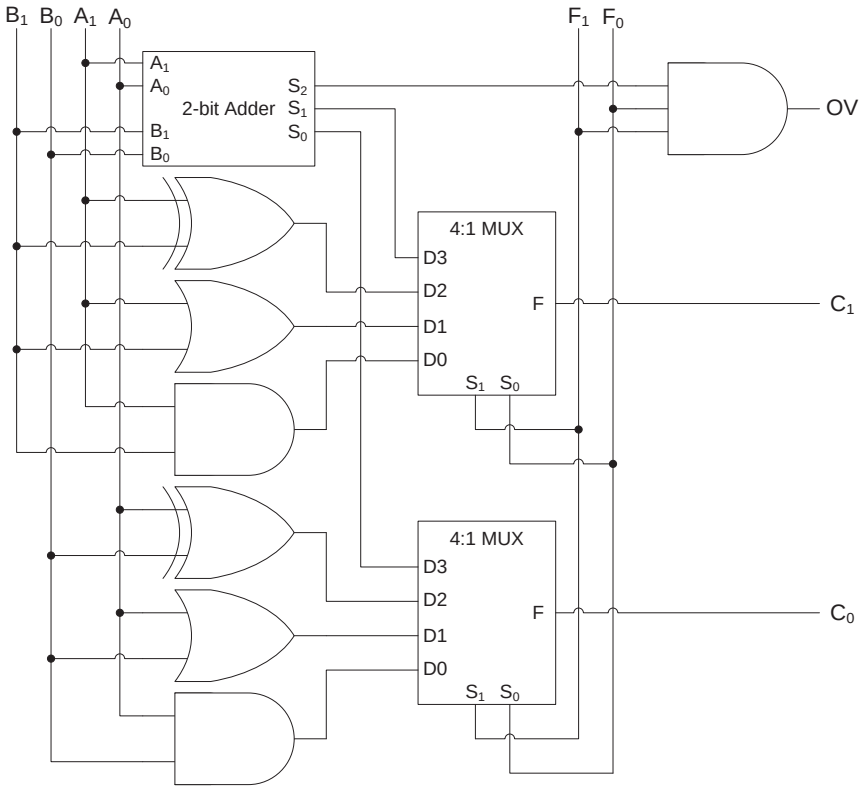


Figure 6.2: Boolean ALU implementation.

even if it is supposedly turned off. Sub-threshold leakage increases exponentially with decreasing transistor feature size.

Among the many techniques proposed to control or minimize leakage power in deep sub-micron technology, Multi-Threshold CMOS (MTCMOS) [2], which reduces leakage power by disconnecting the power supply from the circuit during idle (or sleep) mode while maintaining high performance in active mode, is very promising. MTCMOS incorporates transistors with two or more different threshold voltages (V_t) in a circuit. Low-V_t transistors offer fast speed but have high leakage, whereas high-V_t transistors have reduced speed but far less leakage current. MTCMOS combines these two types of transistors by utilizing low-V_t transistors for circuit switching to preserve performance and high-V_t transistors to gate the circuit power supply to significantly decrease sub-threshold leakage.

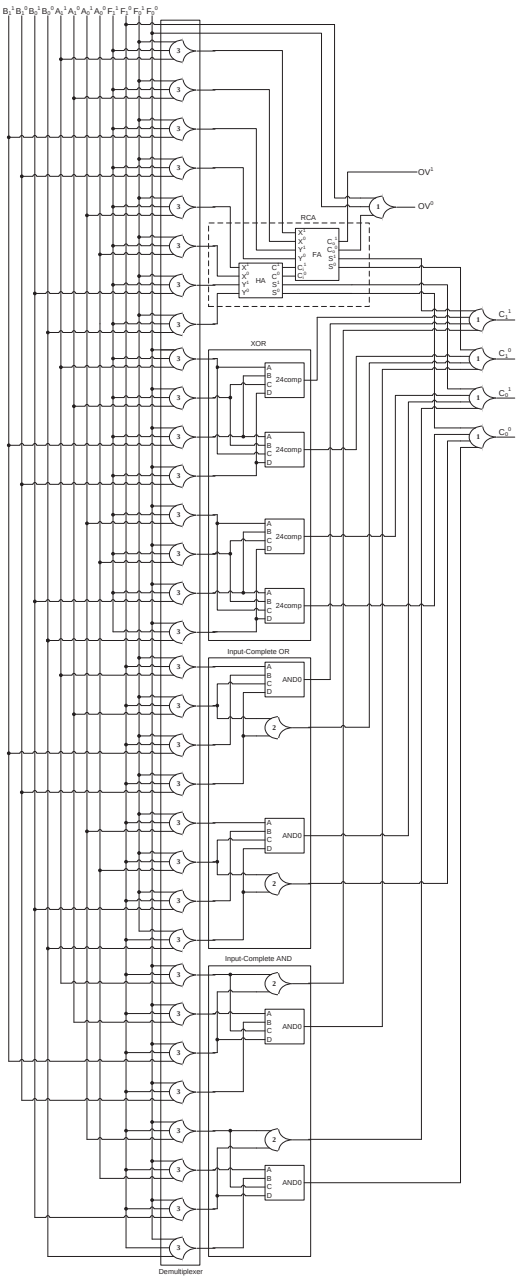


Figure 6.3: NCL ALU implementation.

6.2.1 MTCMOS FOR SYNCHRONOUS CIRCUITS

There are multiple ways to implement MTCMOS in synchronous circuits. One method is to use low-V_t transistors for critical paths to maintain high performance, while using slower high-V_t transistors for the non-critical paths to reduce leakage. Besides this path replacement methodology, there are two other architectures for implementing MTCMOS. A course-grained technique investigated in [3] uses low-V_t logic for all circuit functions and gates the power to entire logic blocks with high-V_t sleep transistors, as shown in Fig. 6.4. The sleep transistors are controlled by a *Sleep* signal. During

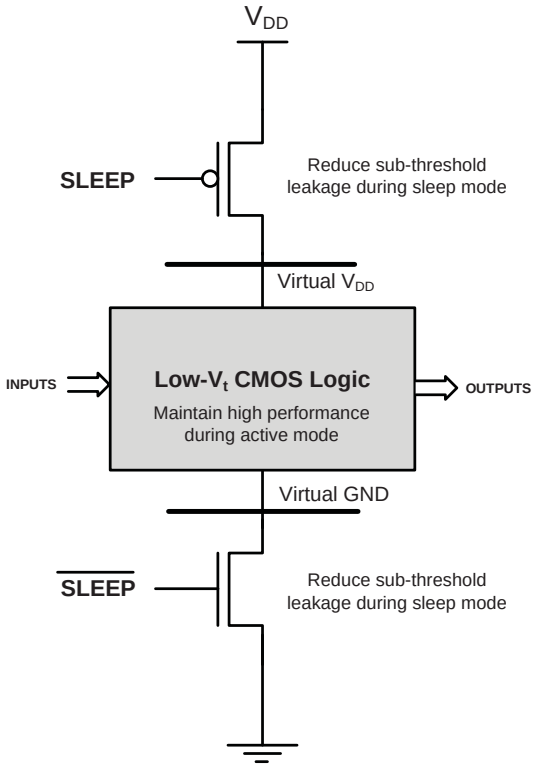


Figure 6.4: General MTCMOS circuit architecture.

active mode, the *Sleep* signal is deasserted, causing both high-V_t transistors to turn on and provide a virtual power and ground to the low-V_t logic. When the circuit is idle, the *Sleep* signal is asserted, forcing both high-V_t transistors to turn off and disconnect power from the low-V_t logic, resulting in a very low sub-threshold leakage current. One major drawback of this method is that partitioning the circuit into appropriate logic blocks and sleep transistor sizing is difficult for large circuits. An alternative fine-grained architecture, shown in Fig. 6.5, incorporates the MTCMOS technique within every gate [4], using low-V_t transistors for the Pull-Up Network (PUN) and Pull-Down

Network (PDN) and a high-V_t transistor to gate the leakage current between the two networks. Two additional low-V_t transistors are included in parallel with the PUN and PDN to maintain nearly

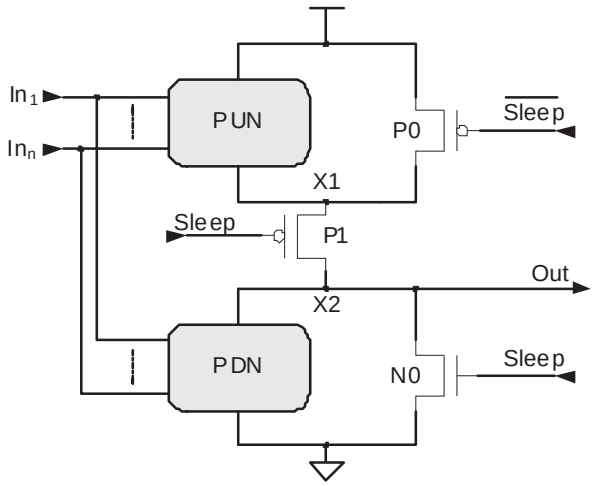


Figure 6.5: MTCMOS applied to a Boolean gate.

equivalent voltage potential across these networks during sleep mode. Implementing MTCMOS within each gate solves the problems of logic block partitioning and sleep transistor sizing; however, this results in a large area overhead.

In general, three serious drawbacks hinder the widespread usage of MTCMOS in synchronous circuits [3]: 1) the generation of *Sleep* signals is timing critical, often requiring complex logic circuits; 2) synchronous storage elements lose data when the power transistors are turned off during sleep mode; and 3) logic block partitioning and transistor sizing is very difficult for the course-grained approach, which is critical for correct circuit operation, and the fine-grained approach requires a large area overhead. However, all three of these drawbacks are eliminated by utilizing NCL in conjunction with the MTCMOS technique.

6.2.2 IMPLEMENTING MTCMOS IN NCL CIRCUITS

6.2.2.1 Early-Completion Input-Incomplete (ECII) MTNCL Architecture

NCL threshold gates are larger and implement more complicated functions than basic Boolean gates, such that fewer threshold gates are normally needed to implement an arbitrary function compared to the number of Boolean gates; however, the NCL implementation often requires more transistors. Therefore, incorporating MTCMOS inside each threshold gate will facilitate easy sleep transistor sizing without requiring as large of an area overhead. Since floating nodes may result in substantial short circuit power consumption at the following stage, an MTCMOS structure similar to the one shown in Fig. 6.5 is used to pull the output node to ground during sleep mode. When all MTNCL

gates in a pipeline stage are in sleep mode, such that all gate outputs are logic 0, this condition is equivalent to the pipeline stage being in the NULL state. Hence, after each DATA cycle, all MTNCL gates in a pipeline stage can be forced to output logic 0 by asserting the sleep control signal instead of propagating a NULL wavefront through the stage, such that data is not lost during sleep mode.

Since the completion detection signal, K_o, indicates whether the corresponding pipeline stage is ready to undergo a DATA or NULL cycle, K_o can be naturally used as the sleep control signal, without requiring any additional hardware, in contrast to the complex *Sleep* signal generation circuitry needed for synchronous MTCMOS circuits. Unfortunately, the direct implementation of this idea using regular NCL completion compromises delay-insensitivity, as shown in Fig. 6.6.

In Fig. 6.6, each inverted completion signal is used as the sleep signal for all MTNCL gates in the corresponding pipeline stage. Looking at the left stage, after a DATA (D) cycle, the completion signal becomes *rfn* (i.e., logic 0), which forces all threshold gates in the stage to enter sleep mode since the next cycle will be NULL (N). When this sleep generated NULL wavefront is latched by the subsequent register, the stage's completion signal will switch back to *rfd* (i.e., logic 1). If this occurs before all bits of the preceding DATA wavefront become NULL, the non-NULL preceding wavefront bits will be retained and utilized in the subsequent operation, thereby compromising delay-insensitivity.

To solve this problem, Early Completion, as detailed in Section 5.3, can be used in lieu of regular completion, as shown in Fig. 6.7, where each completion signal is used as the sleep signal for all threshold gates in the subsequent pipeline stage. Now the combinational logic won't be put to sleep until all inputs are NULL and the stage is requesting NULL; therefore, the NULL wavefront is ready to propagate through the stage, so the stage can instead be put to sleep without compromising delay-insensitivity. The stage will then remain in sleep mode until all inputs are DATA and the stage is requesting DATA, and is, therefore, ready to evaluate. This Early Completion MTNCL architecture, denoted as *ECII*, ensures input-completeness through the sleep mechanism (i.e., the circuit is only put to sleep after all inputs are NULL, and only evaluates after all inputs are DATA), such that input-incomplete logic functions can be used to design the circuit, which decreases area and power and increases speed.

6.2.2.2 MTNCL Threshold Gate Design for ECII Architecture

The MTCMOS structure is incorporated inside each NCL threshold gate, and actually results in a number of the original transistors no longer being needed. As shown in Fig. 6.8(a), the *reset* circuitry is no longer needed since the gate output will now be forced to NULL by the MTCMOS sleep mechanism, instead of by all inputs becoming logic 0. *hold1* is used to ensure that the gate remains asserted, once it has become asserted, until all inputs are deasserted, in order to guarantee input-completeness with respect to the NULL wavefront; however, since the ECII architecture guarantees input-completeness through the sleep mechanism, as explained in Section 6.2.2.1, NCL gate hysteresis is no longer required. Hence, the *hold1* circuitry and corresponding NMOS transistor are removed, and the PMOS transistor is removed to maintain the complementary nature of CMOS

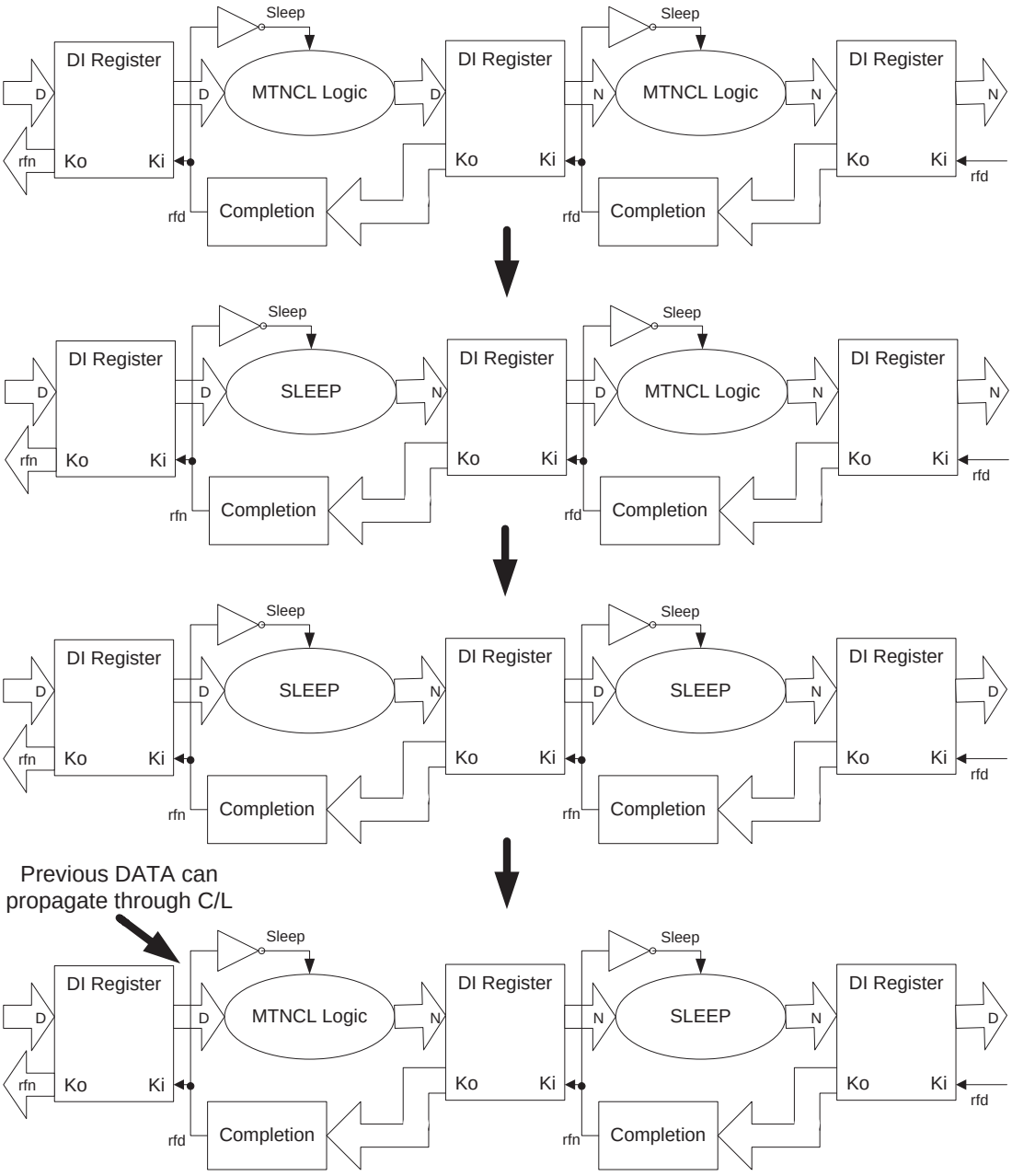


Figure 6.6: MTNCL pipeline architecture using regular completion.

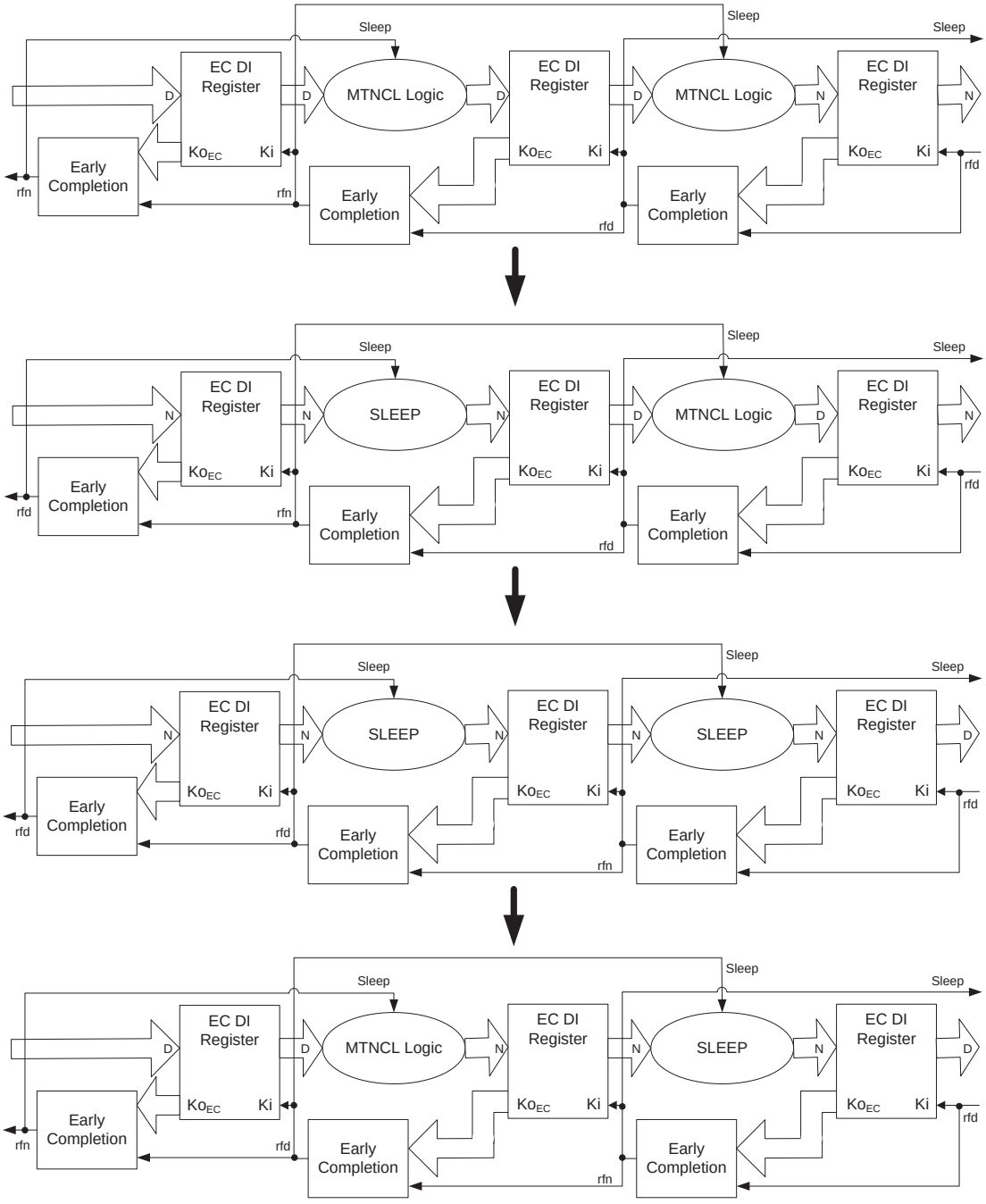


Figure 6.7: MTNCL pipeline architecture using Early Completion.

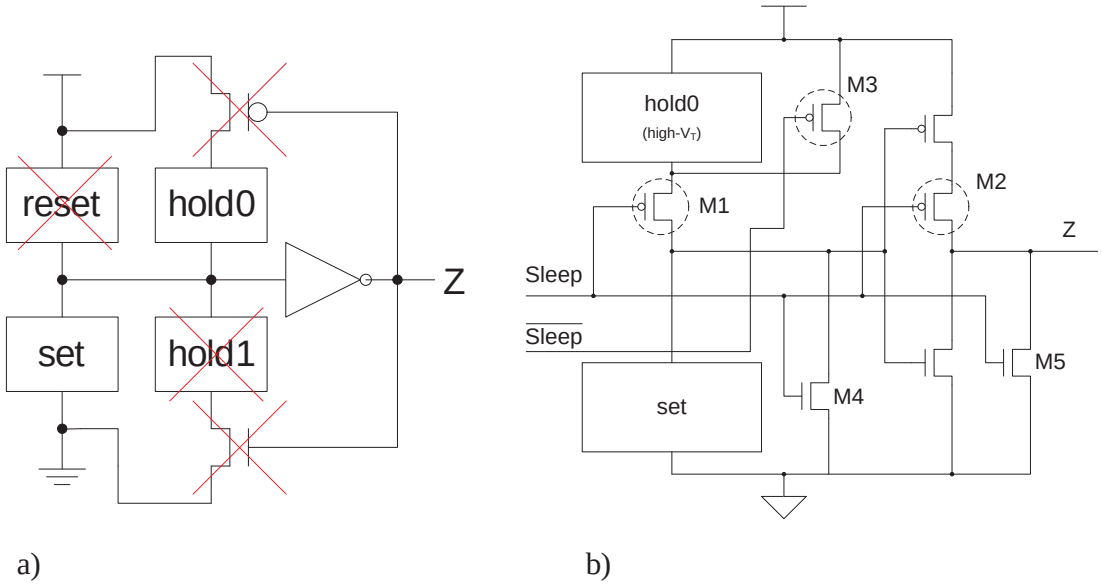


Figure 6.8: Incorporating MTCMOS into NCL threshold gates.

logic (i.e., *set* and *hold0* are complements of each other, as explained in Section 2.2), such that the gate is never floating.

A direct MTCMOS NCL threshold gate implementation, similar to the structure shown in Fig. 6.5, is shown in Fig 6.8(b). All PMOS transistors except the inverter are high-V_t, denoted by a dotted circle because they are only turned on when the gate enters sleep mode and the inputs become logic 0, and remain on when the gate exits sleep mode until the gate's *set* condition becomes true. In both cases, the gate output is already logic 0; therefore, the speed of these PMOS transistors does not affect performance, so high-V_t transistors are used to reduce leakage current. During active mode, the *Sleep* signal is logic 0 and $\overline{Sleep}$ is logic 1, such that sleep transistors $M1$ and $M2$ are turned on, bypass transistors $M3$ and $M4$ are turned off, and the output pull-down transistor, $M5$, is also turned off, such that the gate functions as normal. During sleep mode, *Sleep* is logic 1 and $\overline{Sleep}$ is logic 0, such that $M5$, which is a low-V_t transistor, is turned on to quickly pull the output to logic 0, while $M3$ and $M4$ are turned on to minimize the voltage potential across the *hold0* and *set* blocks, respectively, and high-V_t gating transistors, $M1$ and $M2$, are turned off to reduce leakage. As an example, this MTNCL implementation of the static TH23 gate is shown in Fig. 6.9, whereas the original static TH23 gate is shown in Fig. 2.10(a).

Note that the MTNCL TH23 gate is actually smaller than the original TH23 gate (i.e., 17 vs. 18 transistors). Although five transistors are added to each gate for the MTNCL structure, this only increases total number of transistors for 3 out of the 27 threshold gates since the *reset* and *hold1*

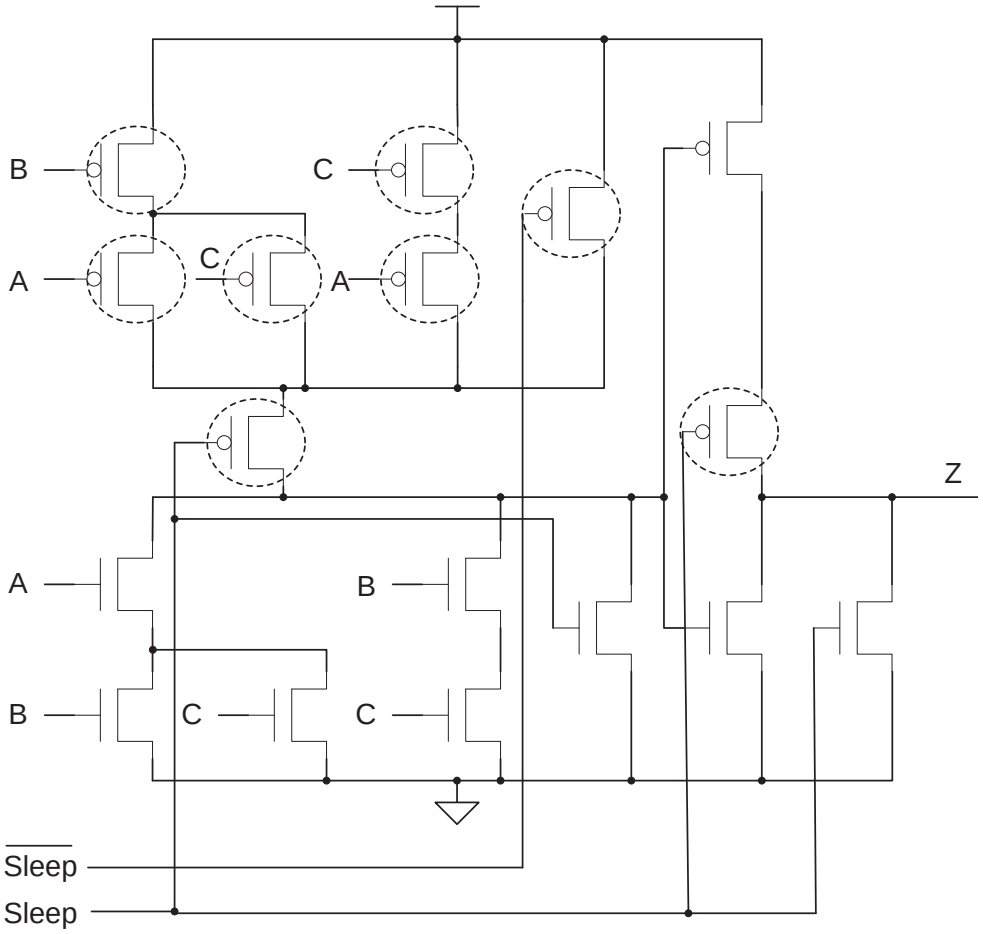


Figure 6.9: Original MTNCL static TH23 gate.

blocks are removed, such that the total number of transistors in the MTNCL version is normally less than the original version. The exceptions are the three TH1n gates, which are OR gates and, therefore, do not have extra hysteresis circuitry (i.e., *reset* = *hold0* and *set* = *hold1*), such that no transistors can be removed for the MTNCL versions, and, therefore, these three MTNCL gates each require 5 additional transistors.

This initial MTNCL static threshold gate structure has been used to implement an 8-bit×8-bit pipelined array multiplier with the 1.2V 130nm IBM 8RF CMOS process, yielding a 150× leakage power reduction and 1.8× active energy savings compared to the regular NCL low-V_t counterpart [5]. However, this structure produces unwanted glitches at the gate outputs, as shown

in Fig. 6.10(b). Referring to Fig. 6.10(a), during sleep mode, *Sleep* is logic 1 and $\overline{Sleep}$ is logic 0. $Q1$ and $Q5$ are off, while $Q1$, $Q2$, and $Q4$ are on. The internal parasitic capacitance, C_p, is discharged through $Q4$, making the internal node, p, logic 0. When the gate is taken out of sleep mode, *Sleep* is logic 0 and $\overline{Sleep}$ is logic 1. $Q1$ and $Q5$ are on, while $Q1$, $Q2$, and $Q4$ are off. Since all inputs are logic 0 at this moment, due to the preceding NULL/sleep cycle, C_p begins charging through the PMOS network and $Q1$. However, before the voltage on p rises to $V_{DD} - |V_{TP}|$, where V_{TP} is the threshold voltage of the PMOS transistor in the output inverter, the gate output will start to rise since the input signal to the inverter, which is the voltage on C_p, momentarily turns on the PMOS transistor, causing the glitch shown in Fig. 6.10(b). With a supply voltage of 1.2V, these glitches

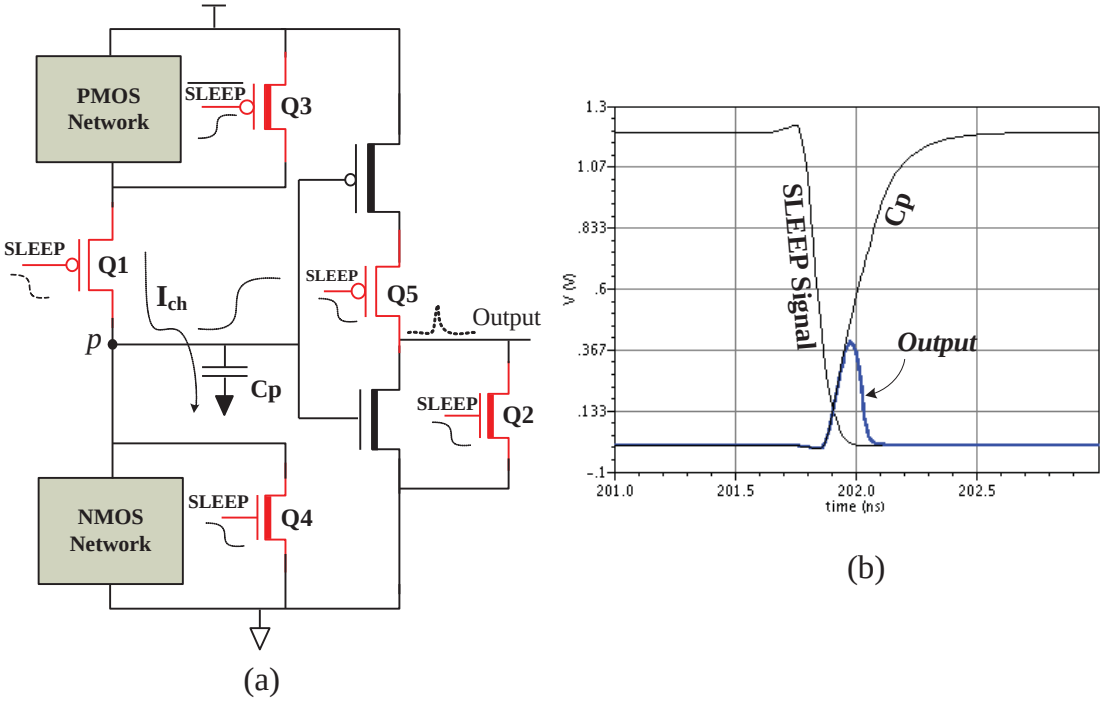


Figure 6.10: (a) Original MTNCL static threshold gate structure, and (b) output glitch.

can be as high as 400mV, and are able to propagate through logic gates. Although the multiplier test circuit still functioned correctly, these glitches need to be removed to ensure reliable operation and eliminate glitch power. Additionally, the two bypass transistors, $Q3$ and $Q4$, were found to only have very minimal contribution to leakage savings; therefore, they can be removed to reduce area.

To eliminate the glitch, the MTNCL threshold gate structure was modified, as shown in Fig. 6.11, by moving the power gating high-V_t transistor to the PDN, and removing the two bypass transistors, as discussed previously, such that during sleep mode the internal node will be charged

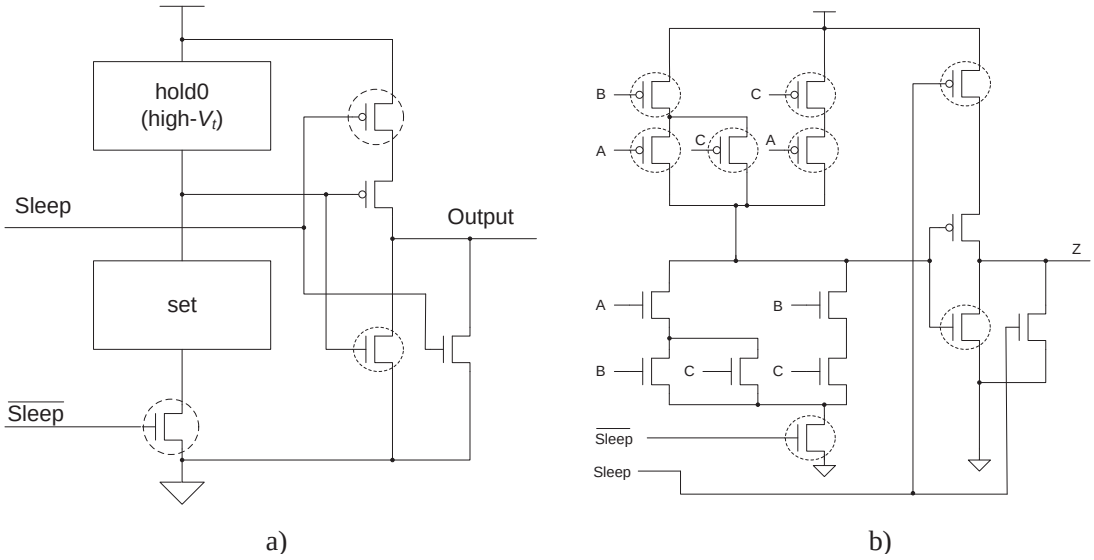


Figure 6.11: (a) SMTNCL gate structure, and (b) TH23 implementation.

to logic 1. Therefore, when the gate is taken out of sleep mode, the output will remain at logic 0 without any glitch, due to the internal logic 1 flowing through the output inverter, until the DATA wavefront arrives. Note that since the internal node is logic 1 during sleep mode and the output is logic 0, the NMOS transistor in the output inverter is no longer on the critical path and therefore can be a high-V_t transistor. This modified Static MTNCL threshold gate structure is referred to as SMTNCL.

6.2.2.3 Delay-Insensitivity Analysis

Combining the ECII architecture with the SMTNCL gate structure, results in a delay-sensitivity problem, as shown in Fig. 6.12. After a DATA cycle, if most, but not all, inputs become NULL, this Partial NULL (PN) wavefront can pass through the stage's input register, because the subsequent stage is requesting NULL, and cause all stage outputs to become NULL, before all inputs are NULL and the stage is put to sleep because the *hold1* logic has been removed from the SMTNCL gates. This violates the input-completeness criteria, discussed in Section 3.1 and can cause the subsequent stage to request the next DATA while the previous stage input is still a partial NULL, such that the preceding wavefront bits that are still DATA will be retained and utilized in the subsequent operation, thereby compromising delay-insensitivity, similar to the problem when using regular completion, as explained in Section 6.2.2.1.

There are two solutions to this problem, one at the architecture level and the other at the gate level. Since the problem is caused by a partial NULL passing through the register, this can be fixed at the architecture-level by ensuring that the NULL wavefront is only allowed to pass through the

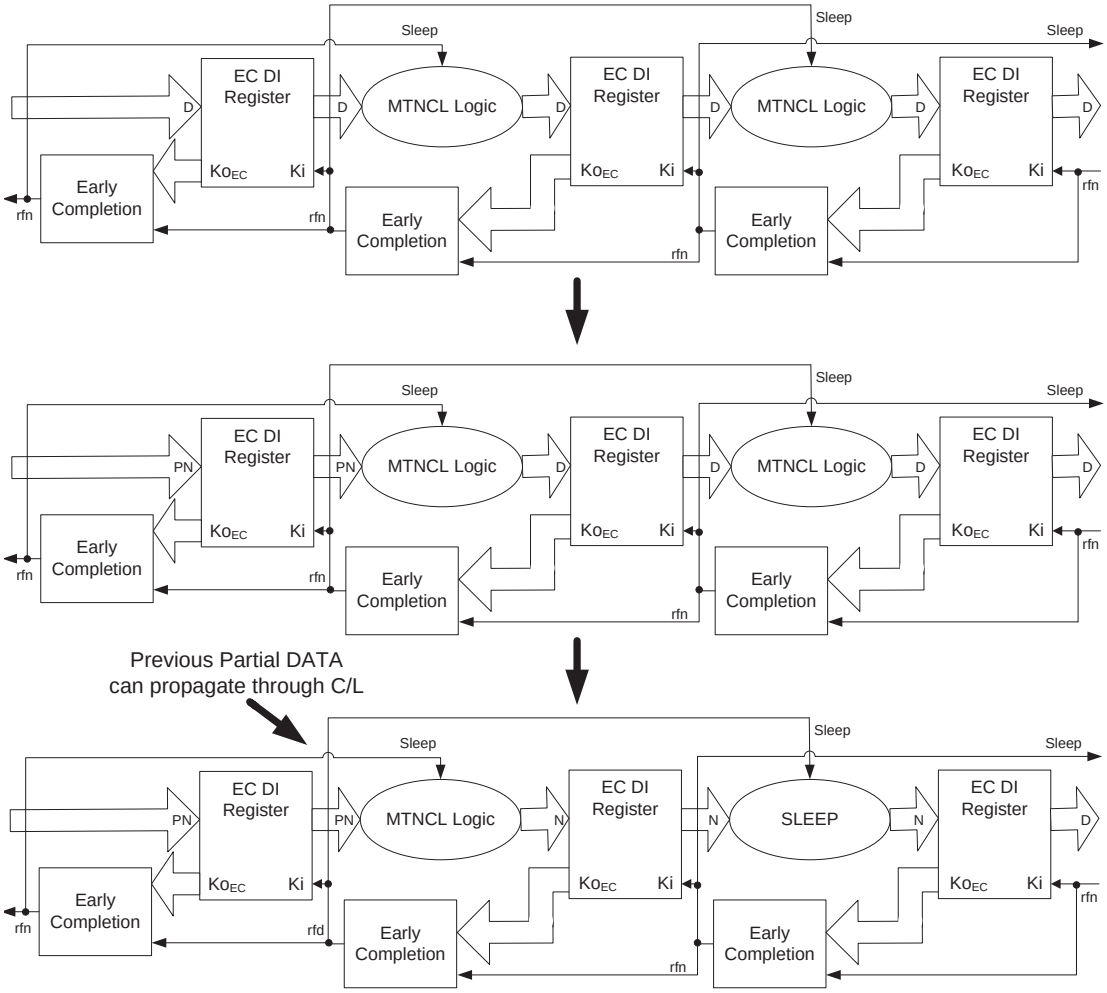


Figure 6.12: Delay-sensitivity problem combining ECII architecture with SMTNCL gates.

register after all register inputs are NULL, which is easily achievable by using the stage's inverted sleep signal as its input register's K_i signal. This Fixed Early Completion Input-Incomplete (FECII) architecture is shown in Fig. 6.13. Compared to ECII, FECII is slower because the registers must wait until all inputs become DATA/NULL before they are latched. Note that a partial DATA wavefront passing through the register does not pose a problem, because the stage will remain in sleep mode until all inputs are DATA, thereby ensuring that all stage outputs will remain NULL until all inputs are DATA.

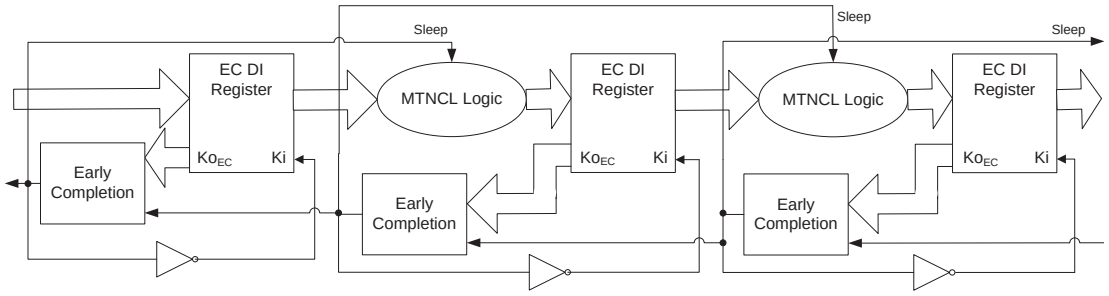


Figure 6.13: Fixed Early Completion Input-Incomplete (FECII) architecture.

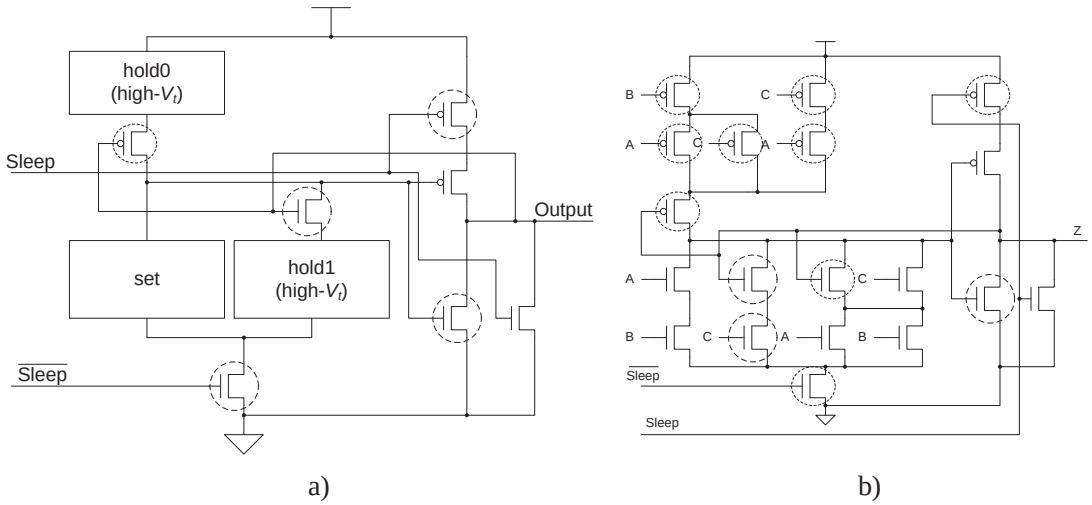


Figure 6.14: (a) SMTNCL1 gate structure, and (b) TH23 implementation.

This problem can be solved at the gate level by adding the *hold1* logic back into each SMTNCL gate, to ensure input-completeness with respect to NULL, such that a partial NULL wavefront cannot cause all outputs to become NULL. Note that this requires the PMOS transistor between *hold0* and V_{DD} to be added back to prevent a direct path from V_{DD} to ground when both *hold1* and *hold0* are simultaneously asserted. Also note that the *hold1* transistors not shared with the *set* condition can be high-V_t transistors, since they are not on the critical path. This Static MTNCL implementation with *hold1* is shown in Fig. 6.14, and is denoted as SMTNCL1.

Since SMTNCL1 increases transistor count, the MTCMOS structure can be applied to semi-static NCL gates, which utilize a weak feedback inverter to implement the *hold1* and *hold0*

functions. This Semi-Static MTNCL design with *hold1* is denoted as SSMTNCL1, and is shown in Fig. 6.15. Note that a 1 at the end of an MTNCL gate name denotes that the gate includes

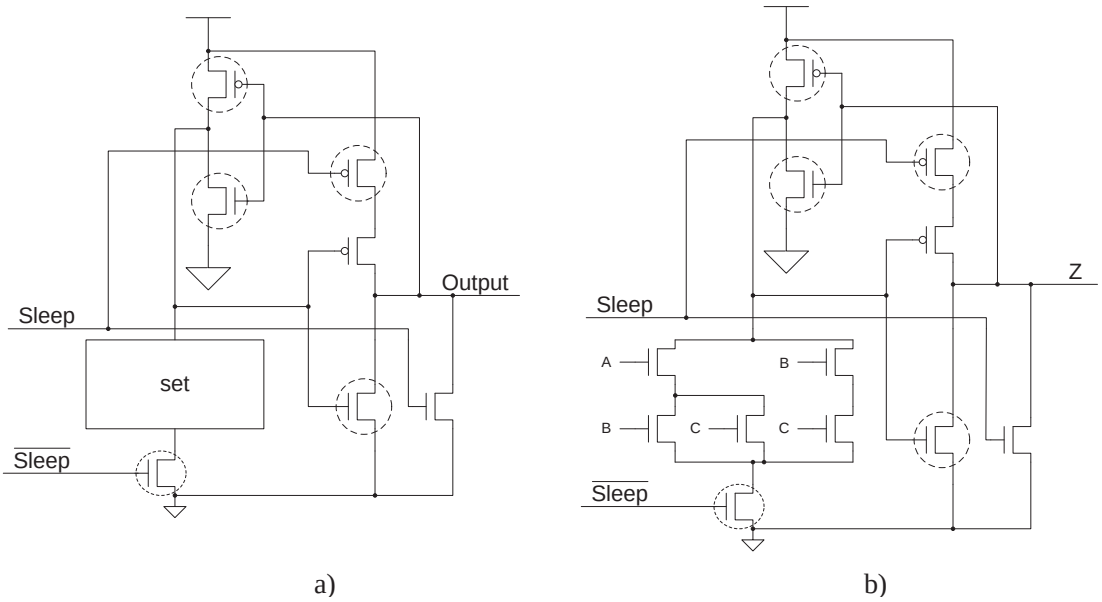


Figure 6.15: (a) SSMTNCL1 gate structure, and (b) TH23 implementation.

hold1 circuitry. The NMOS transistor in the weak inverter serves as the *hold1* function, which is not needed for the FECII architecture; hence, this transistor can be removed to save area when using the slower FECII architecture. This modified Semi-Static MTNCL design is denoted as SSMTNCL, and is shown in Fig. 6.16.

To summarize, the ECII architecture only works with SMTNCL1 or SSMTNCL1 gates, which both include the *hold1* function. The FECII architecture works with all four MTNCL gate designs (i.e., SMTNCL, SMTNCL1, SSMTNCL1, and SSMTNCL). However, the SMTNCL and SSMTNCL gates require fewer transistors than their equivalent SMTNCL1 and SSMTNCL1 gates, respectively, such that the FECII architecture would normally use either the SMTNCL or SSMTNCL gates. Additionally, the ECII architecture is faster than FECII; and the static NCL gates (i.e., SMTNCL and SMTNCL1) perform better than their semi-static counterparts (i.e., SSMTNCL and SSMTNCL1) at reduced supply voltages, since the static implementations do not utilize a weak inverter, which ceases to operate properly with a substantially reduced supply voltage, for state-holding. Preliminary simulation results can be found in [5]–[7]; and additional results will be posted on the authors' websites [8]–[9], as they become available. Note that MTNCL is also known as DIMLOG (i.e., Delay-Insensitive Multi-threshold LOGic).

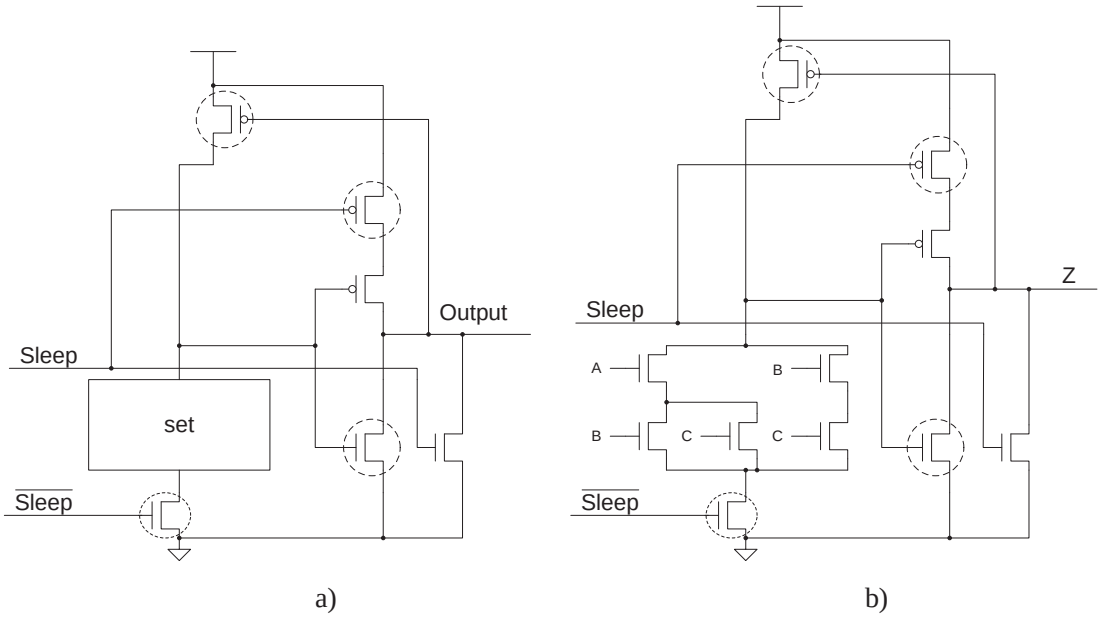


Figure 6.16: (a) SSMTNCL gate structure, and (b) TH23 implementation.

BIBLIOGRAPHY

[1] S. K. Bandapati and S. C. Smith, "Design and Characterization of NULL Convention Arithmetic Logic Units," *Elsevier's Microelectronic Engineering: Special Issue on VLSI Design and Test*, Vol. 84/2, pp. 280–287, February 2007. DOI: 10.1016/j.mee.2006.02.012

[2] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-V Power Supply High-Speed Digital Circuit Technology with Multithreshold-Voltage CMOS," *IEEE Journal of Solid-State Circuits*, Vol. 30/8, pp. 847–854, August 1995. DOI: 10.1109/4.400426

[3] J. T. Kao and A. P. Chandrakasan, "Dual-Threshold Voltage Techniques for Low-Power Digital Circuits," *IEEE Journal of Solid-State Circuits*, Vol. 35/7, pp. 1009–1018, July 2000. DOI: 10.1109/4.848210

[4] P. Lakshmikanthan, K. Sahni, and A. Nunez, "Design of Ultra-Low Power Combinational Standard Library Cells Using a Novel Leakage Reduction Methodology," *IEEE International SoC Conference*, 2006. DOI: 10.1109/SOCC.2006.283854

[5] A. D. Bailey, A. Al Zahrani, G. Fu, J. Di, and S. C. Smith, "Multi-Threshold Asynchronous Circuit Design for Ultra-Low Power," *Journal of Low Power Electronics*, Vol. 4/3, pp. 337–348, December 2008. DOI: 10.1166/jolpe.2008.181

[6] A. Alzahrani, A. D. Bailey, G. Fu, and J. Di, "Glitch-Free Design for Multi-Threshold CMOS NCL Circuits," *2009 Great Lakes Symposium on VLSI*, May 2009. DOI: 10.1145/1531542.1531596

[7] A. D. Bailey, J. Di, S. C. Smith, and H. A. Mantooth, "Ultra-Low Power Delay-Insensitive Circuit Design," *IEEE Midwest Symposium on Circuits and Systems*, pp. 503–506, August 2008. DOI: 10.1109/MWSCAS.2008.4616846

[8] http://comp.uark.edu/~smithsco

[9] http://comp.uark.edu/~jdi

CHAPTER 7

Comprehensive NCL Design Example

This chapter details the NCL implementation of an optimized version of the Greatest Common Divisor (GCD) circuit, discussed in Section 4.2 and shown in Fig. 4.9. The ThroughPut Capability (TPC) diagram for the original GCD circuit is shown in Fig. 7.1. TPC is a measure of the number of new inputs loaded divided by the number of clock cycles to process the data. A * denotes when a new

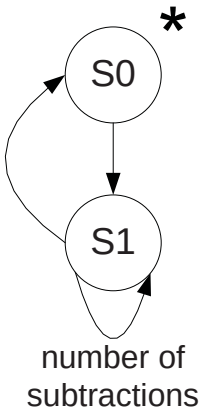


Figure 7.1: TPC diagram for original GCD circuit.

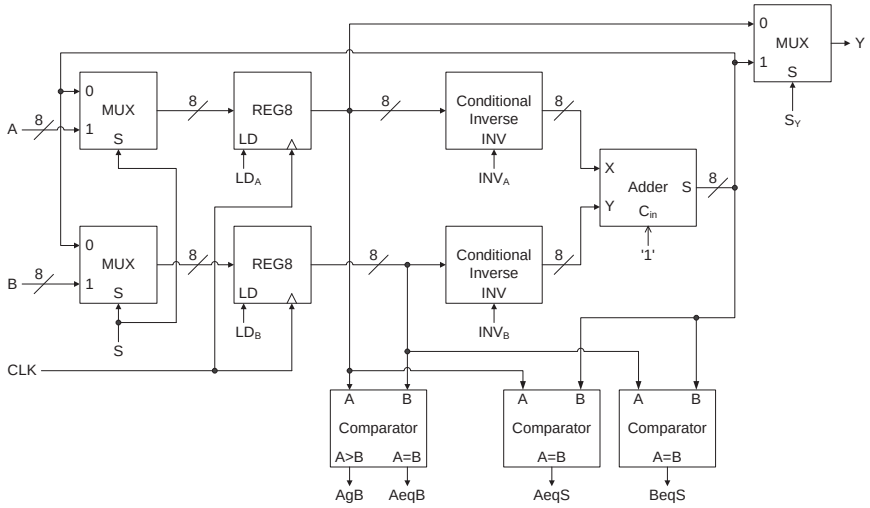
data is loaded, which occurs on the transition from one state to another. Note that the following three assumptions are used when calculating TPC: 1) steady state operation (i.e., initial states are ignored); 2) data is supplied as fast as requested (i.e., X_{dat} is always asserted); and 3) data is requested at the output as fast as supplied (i.e., Y_{rqst} is always asserted). This TPC diagram shows that TPC is 1/2 for the special case where $A = B$ (i.e., $S0 \rightarrow S1 \rightarrow S0$, loading a data on the $S0 \rightarrow S1$ transition), and 1/(number of subtractions + 2) for the general case when $A \neq B$ (i.e., $S0 \rightarrow S1 + S1 \rightarrow S1$ for each subtraction $+S1 \rightarrow S0$, loading a data on the $S0 \rightarrow S1$ transition). Additionally, overall TPC is calculated as the worse-case TPC for any operation (e.g., for the original GCD circuit, the worse case operation is when one input is 1 and the other is 255, which requires 254 subtractions, yielding a worse-case TPC of 1/256).

By analyzing the GCD algorithm, the maximum attainable TPC, TPC_{MAX}, is expected to be 1 for the special case where $A = B$, and 1/(number of subtractions) for the general case when $A \neq B$. To achieve TPC_{MAX}, the datapath and corresponding ASM have been redesigned to load new operands at the same time the previous result is output, and to output the result one cycle sooner, by taking it from the output of the adder rather than loading it back into the register, as shown in Figs. 7.2(a) and 7.2(b), respectively. The new TPC diagram, shown in Fig. 7.2(c), indeed shows that TPC is 1 when $A = B$ (i.e., $S1 \rightarrow S1$ loading new operands each transition), and 1/(number of subtractions) when $A \neq B$ (i.e., $S1 \rightarrow S1$, which outputs the previous result after the final subtraction and loads the next operands simultaneously, $+S1 \rightarrow S1$ number of subtractions – 1 times). Note that $S0$ is an initial state since it is never returned to as long as X_{dat} is asserted; hence, $S0$ is not shown in the TPC diagram. Also note that this GCD optimization increases TPC at the expense of increased datapath delay, such that if the GCD datapath has the longest delay in the system, it may be better to utilize a less optimized GCD circuit, such that a faster clock can be used for the entire system.

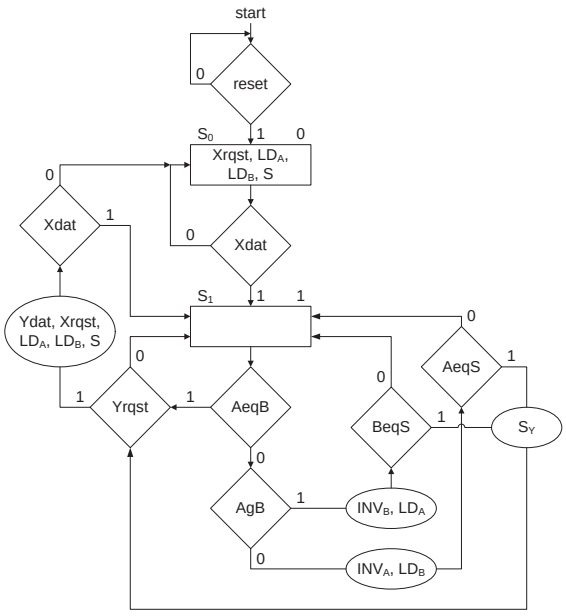
When implementing the GCD circuit using NCL, the OCDDC handshaking is no longer needed since NCL circuits already utilize input and output handshaking. Therefore, the interface and ASM must be modified, as shown in Fig. 7.3(b) and 7.3(c). The datapath, shown in Fig. 7.3(a), must be modified by replacing the A and B input/feedback registers with 3-register NCL feedback. This requires adding two MUXes to feedback data to the input MUXes each iteration (i.e., either A or the RCA *sum* for the top MUX and either B or the RCA *sum* for the bottom MUX). To reduce area, registration was embedded within all four MUXes, and a third DI Register was added on both of these feedback paths to complete the required 3-register feedback. Additionally, Y is only output when the GCD operation is completed, not every iteration; hence, a Select Register is required, controlled by Y_{dat}, which is now an internal signal. Two additional single-bit DI Registers were added, one to generate the select signal, S, for the input MUXes, which was required to complete the 3-register feedback, and a second to latch Y_{dat}.

MUX Reg is the input-incomplete 2:1 MUX with embedded registration, shown in Fig. 5.5(c). Note that the value in parenthesis in the registers denotes their reset condition (i.e., N refers to NULL, $D0$ to DATA0, and $D1$ to DATA1). MUX Reg Comp0 is a 2:1 MUX with embedded registration that is input-complete with respect to its $D0$ input, but not $D1$. This is equivalent to the input-complete 2:1 MUX with embedded registration, shown in Fig. 5.4(b), with the TH24comp gate replaced by a TH12 gate with inputs $D0^0$ and $D0^1$. The Select Reg is shown in Fig. 7.4, and only passes data to the output when both K_i and S are asserted.

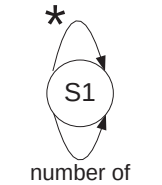
The input MUXes are both designed to be input-incomplete with respect to their $D1$ inputs because A and B are only DATA at the beginning of each new operation. The B input MUX must be input-complete with respect to its $D0$ input to ensure that the output of the B/*sum* feedback MUX is observable when a new operand is loaded. This is not required for the A input MUX since the output of the A/*sum* feedback MUX is observable through the Select Reg when a new operand is loaded. The feedback MUXes are both input-incomplete because one of the two will always pass



a) datapath

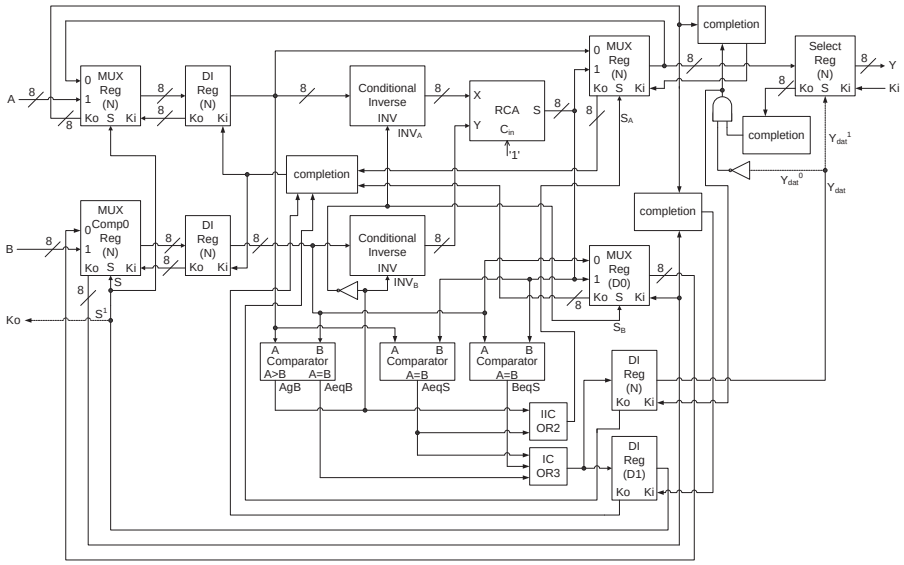


b) ASM

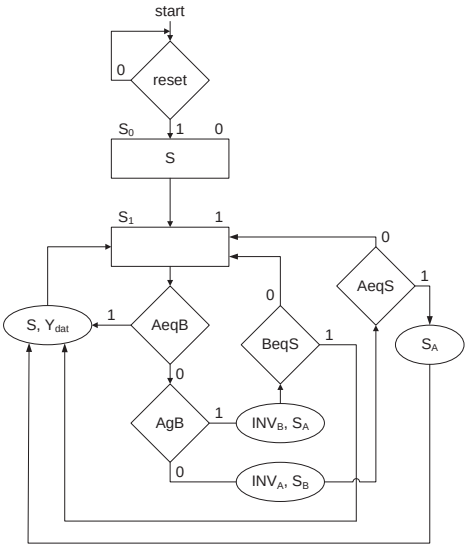


c) TPC diagram

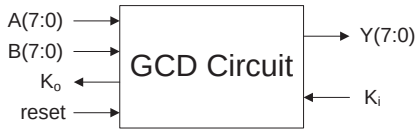
Figure 7.2: Optimized GCD circuit. (a) datapath; (b) ASM; (c) TPC diagram.



a) datapath



b) ASM



c) interface

Figure 7.3: Optimized NCL GCD circuit. (a) datapath; (b) ASM; (c) interface.

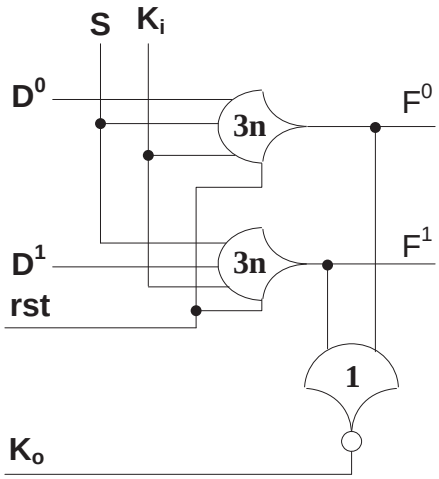


Figure 7.4: Select Reg.

the RCA *sum*, which is input-complete with respect to both the internal *A* and *B* signals, such that the C/L is input-complete, and the RCA *sum* is observable. Since Select Reg only outputs a DATA at the end of a complete GCD operation, not every iteration, its K_o signal will remain *rfd* during all internal GCD iterations, such that it needs to be masked during these iterations to allow the DATA/NULL cycles to flow through the feedback loops. This is done by ANDing its completion signal with the inverse of Y^0_{dat}, such that when Y_{dat} is DATA0 (i.e., the result is not ready and will not be output through the Select Reg) the completion signal becomes *rfn*. Likewise, when Y_{dat} is DATA1 or NULL, the completion signal of Select Reg is passed. Additionally, since new *A* and *B* operands are only loaded at the beginning of each new GCD operation, not each iteration, K_o cannot be generated from the input MUX Regs' completion signals; instead, K_o is tied to the input MUXes' select input, S^1, such that K_o is only asserted when the new operands are ready to be passed through the input MUXes.

NCL versions of the C/L datapath components can be designed as follows. The Conditional Inverse circuit is simply an array of NCL XOR functions, each consisting of two TH24comp gates, as explained in Section 3.2, with INV as one XOR input and one of the 8-bit data inputs as the second XOR input, generating the corresponding bit of the 8-bit data output. The RCA utilizes a FA1 component (i.e., full adder with constant 1 input) with inputs A_0 and B_0 to generate the least significant bit, which consists of an NCL XNOR function to generate the *sum* output and an input-incomplete OR function to generate the *carry* output. These functions are described in Sections 3.2 and 4.1, respectively. Full adders, as shown in Fig. 3.13, are used to generate bits 1-6; and a FAS component (i.e., full adder with *sum* output only), which consists of two NCL XOR functions, is used to generate the Most Significant Bit (MSB). Note that a regular full adder cannot be used for

the MSB because the internal gates are not observable when *carry* is not an output. The $A = B$ comparators are designed as shown in Fig. 7.5, where the XNOR and AND gates are both NCL functions; and the AND gates are input-complete to ensure observability of the XNOR outputs. The $A > B$ output of the third comparator is designed by partitioning the inputs into groups of

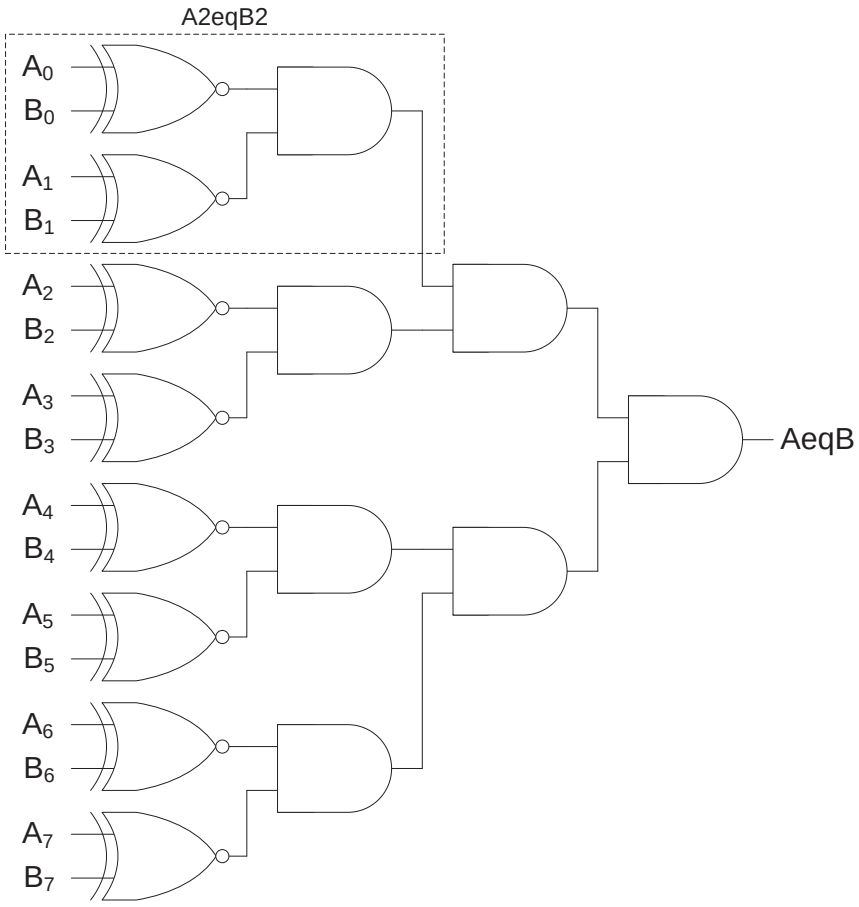


Figure 7.5: *AeqB* comparator.

two, as shown in the following equation:

$$\begin{aligned} AgB = {} & A2gB2(7:6) + A2eqB2(7:6) \bullet A2gB2(5:4) \\ & + A2eqB2(7:6) \bullet A2eqB2(5:4) \bullet A2gB2(3:2) \\ & + A2eqB2(7:6) \bullet A2eqB2(5:4) \bullet A2eqB2(3:2) \bullet A2gB2(1:0) \end{aligned}$$

The *A2eqB2* circuit is shown in the dotted box in Fig. 7.5; and the *A2gB2* circuit is shown in Fig. 7.6, designed using the method described in Section 3.2. The complete diagram of the *AgB*/*AeqB*

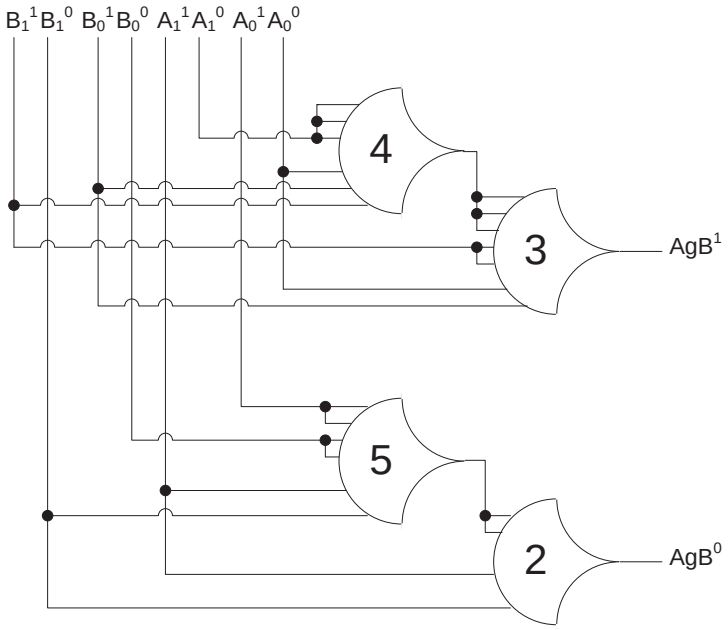


Figure 7.6: *A2gB2* circuit.

comparator is shown in Fig. 7.7, where all AND and OR gates are input-complete NCL functions, to ensure observability. Alternatively, the *AgB* output could have been designed by partitioning the inputs into groups of three, also resulting in a circuit with 6 gate delays; however, the implementation shown in Fig. 7.7 requires fewer gates.

The minimal next-state and output equations are derived directly from the ASM and then optimized as follows utilizing Boolean algebra and mutually exclusive conditions:

$$
\begin{aligned}
S &= S_0 + S_1 \bullet AeqB + S_1 \bullet AeqB' \bullet AgB \bullet BeqS + S_1 \bullet AeqB' \bullet AgB' \bullet AeqS \\
&= S_0 + AeqB + BeqS + AeqS \\
S_B &= INV_A = S_1 \bullet AeqB' \bullet AgB' = S_1 \bullet AgB' \\
INV_B &= S_1 \bullet AeqB' \bullet AgB = S_1 \bullet AgB \\
S_A &= S_1 \bullet AeqB' \bullet AgB + S_1 \bullet AeqB' \bullet AgB' \bullet AeqS = S_1 \bullet AgB + S_1 AeqS \\
Y_{\text{dat}} &= S_1 \bullet AeqB + S_1 \bullet AeqB' \bullet AgB \bullet BeqS + S_1 \bullet AeqB' \bullet AgB' \bullet AeqS \\
&= S_1 \bullet (AeqB + BeqS + AeqS) \\
D &= 1
\end{aligned}
$$

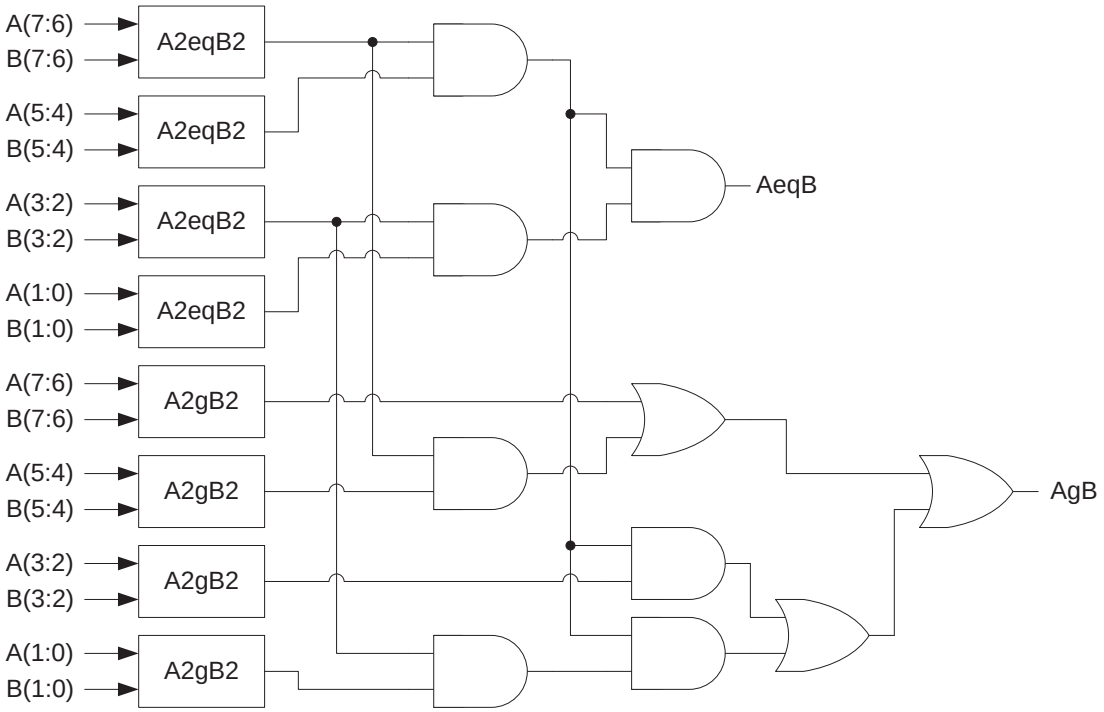


Figure 7.7: *AgB*/*AeqB* comparator.

These equations could now be implemented in a 3-register feedback state machine, as detailed in Section 4.1. However, since S_0 is only an initial state that is never returned to, the initial setup condition (i.e., S asserted and Y_{dat} not asserted) can be realized by using separate registers for S and Y_{dat}, as shown in Fig. 7.3(a), to reset S to DATA1 to pass the first operands and Y_{dat} to NULL to not output Y upon reset. Hence, the state is no longer needed, and can be removed from the equations as follows:

$$\begin{aligned} S = Y_{\text{dat}} &= AeqB + BeqS + AeqS \\ SB &= AgB' \\ INV_B &= AgB \\ S_A &= AgB + AeqS \end{aligned}$$

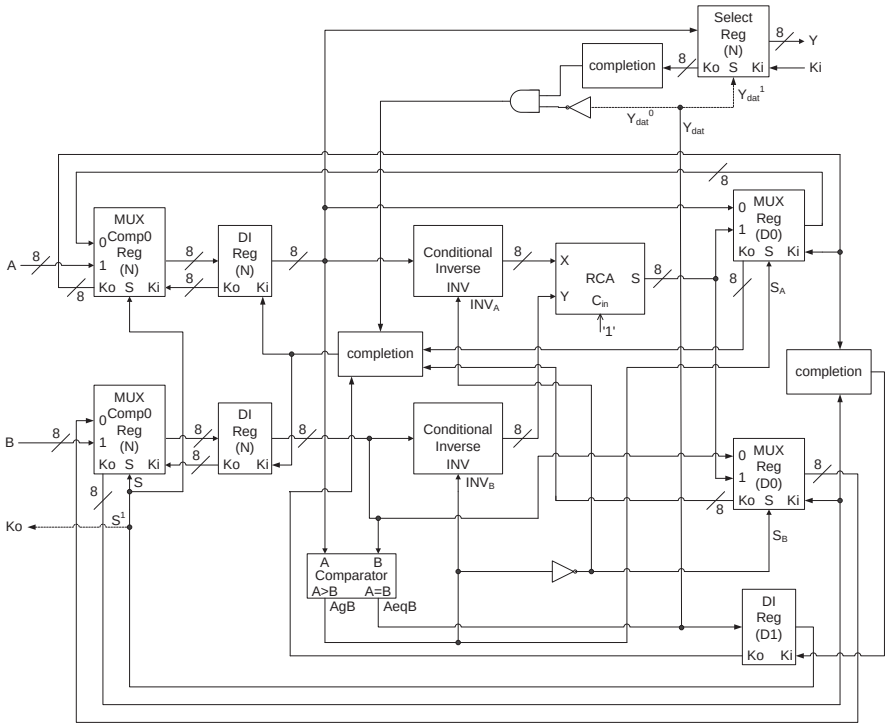
To make the comparators' outputs observable, S/Y_{dat} must utilize an input-complete NCL OR3 function, consisting of two input-complete NCL OR2 functions, while S_A can be implemented with an input-incomplete OR function. S_B is implemented by simply swapping the rails of *AgB*, as discussed in Section 3.2. This ensures that all comparator outputs return to NULL before the next data wavefront is latched to flow through the C/L.

As mentioned above, using the fully optimized GCD circuit for a synchronous design depends on the clock period of the entire system, due to the tradeoff of increased datapath delay vs. increased TPC. The NCL GCD circuit is self-timed and therefore its throughput is independent of other system components. Hence, the design with the smaller datapath delay should also be considered to ensure optimal throughput. This design is shown in Fig. 7.8, where the new operands are still loaded at the same time the previous result is output, but the result is output one cycle later when $A \neq B$, resulting in a TPC of 1 when $A = B$ and 1/(number of subtractions + 1) when $A \neq B$.

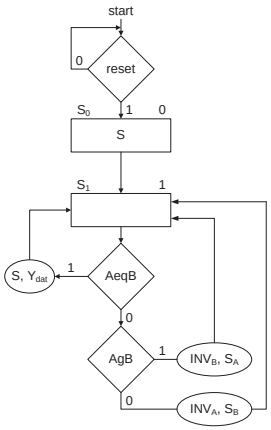
Both systems were exhaustively simulated (i.e., 65025 non-zero operand input combinations) using a VHDL library with timing extracted from physical-level simulations of 1.8V 180nm static NCL gates [1], showing that the version with the smaller datapath delay decreased average operation time by 12% (i.e., 212ns for Fig. 7.3 vs. 190ns for Fig. 7.8). Additionally, utilizing bit-wise completion between Select Reg and MUX Reg (for Fig. 7.3) and 4-register feedback were explored; however, neither decreased average operation time.

BIBLIOGRAPHY

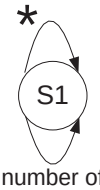
[1] http://comp.uark.edu/~smithsco/VHDL.html.



a) reduced delay NCL datapath



b) ASM



number of subtractions

c) TPC diagram

Figure 7.8: Reduced delay NCL GCD circuit. (a) reduced delay NCL datapath; (b) ASM; (c) TPC diagram.

Biography

SCOTT C. SMITH

Scott C. Smith received B.S. degrees in Electrical Engineering and Computer Engineering and an M.S.E.E., with emphasis in Computers and Digital Systems, from the University of Missouri—Columbia in 1996 and 1998, respectively, and a Ph.D. in Computer Engineering, with an emphasis in Computer Architecture and Digital Systems, from the University of Central Florida in 2001. He started as an Assistant Professor in the Electrical and Computer Engineering department at University of Missouri—Rolla in 2001, was promoted to Associate Professor and tenured in 2007, and is currently at the University of Arkansas as a Tenured Associate Professor in the Department of Electrical Engineering, an Adjunct Associate Professor in the Department of Computer Science & Computer Engineering, and Director of the Asynchronous Digital Design (ADD) Laboratory. Dr. Smith is an expert in asynchronous digital design, specifically focusing on NULL Convention Logic (NCL). He wrote his Ph.D. dissertation on design and optimization of NCL circuits, and has authored numerous publications on asynchronous logic, which can be viewed from his website: http://comp.uark.edu/~smithsco/. Dr. Smith is the recipient of a Phase I NSF Course Curriculum and Laboratory Improvement (CCLI) Grant to develop educational modules for incorporating asynchronous design and testing into the undergraduate Computer Engineering curriculum, and Phase II CCLI grant, along with Dr. Jia Di, to significantly expand the developed curricular content and broadly disseminate it to universities and colleges throughout the nation. The developed materials include lecture notes, example problems, group projects, and asynchronous digital design libraries, consisting of a VHDL library of asynchronous gates, components, and functions, and transistor-level and physical-level libraries of fundamental asynchronous gates, which can be downloaded from the project website: http://comp.uark.edu/~smithsco/CCLI_async.html. Dr. Smith's research interests include CAD Tool Development for Asynchronous Circuits, Asynchronous FPGA Design, VHDL, VLSI, Computer Architecture, Embedded System Design, Evolvable Hardware, Secure/Trustable Hardware, and Wireless Sensor Networks.

JIA DI

Jia Di received B.S. and M.S. degrees from Tsinghua University, China, in 1997 and 2000, respectively, and received his Ph.D. in Electrical Engineering from the University of Central Florida in 2004, where he focused on Energy Aware Design and Analysis for Synchronous and Asynchronous Circuits. He is currently a Tenured Associate Professor in the Department of Computer Science and Computer Engineering, an Adjunct Associate Professor in the Department of Electrical Engineer-

ing, and Director of the Trustable Logic Circuit Design (TruLogic) laboratory, at the University of Arkansas. Dr. Di's research interests include asynchronous logic design and applications in extreme temperature environments, ultra-low power, radiation hardening by design, 3-dimensional IC design, and hardware security. He has authored numerous publications in these areas, which can be viewed from his website: http://comp.uark.edu/~jdi/.