2. BACKGROUND

2.1 Circuit Synthesis

The first digital solid-state circuits were designed structurally by hand through directly instantiating individual logic gates and establishing the connectivity between them. The designer was responsible for manually optimizing the circuit to achieve high performance, low power, and low area. Naturally, this process was time-consuming and error-prone, requiring extensive knowledge and experience to perform proficiently. Decades ago, this approach was replaced with logic synthesis, permitting engineers to describe circuits at a far higher abstraction level in behavioral form using an RTL language like Verilog or VHDL [3]. This behavioral specification allows engineers to describe what the circuit should do as opposed to how the circuit should do it, resulting in numerous advantages.

The first advantage is faster design cycles. Using a standard 64-bit adder as an example, an engineer can describe the adder in just a few minutes with a few lines of RTL code. The synthesis tool can then take the RTL and select from among many architectures—e.g., ripple-carry adder (RCA), carry-save adder (CSA), and carry-lookahead adder (CLA)—to best meet the performance specifications while minimizing power and area [4]. Conversely, if a designer were required to specify the same highly optimized 64-bit adder in structural form, substantially more time would be required. In fact, creating a comparable, highly optimized adder directly with a netlist of gates and interconnections would be practically infeasible—the amount of design time required would be insurmountable, and the opportunities for making mistakes would increase exponentially. To overcome these disadvantages, a designer could instead make use of higher-level constructs like conditional and looping generate statements. For larger designs, these constructs would reduce the turnaround time and the amount of HDL needed for describing the

circuit. However, for less structurally repetitive designs, such as an arbitrary-width CLA, implementation using these higher-level constructs would be quite complicated. Given the increased complexity and insufficient time, a designer may be forced to settle for a less optimal adder architecture like an RCA [4].

The second benefit is the greater opportunity for optimization. If an engineer determines that the current implementation is too slow for the target application, manual adjustments may be needed, such as resizing gates, adjusting the threshold voltage of individual gates, or inserting additional buffers. If these improvements alone are insufficient, the designer might be forced to completely discard the current implementation and redesign the circuit with a different architecture.

In contrast, modern synthesis programs handle this process automatically. For example, if a designer is attempting to synthesize an adder, the tool might initially produce a simple RCA. Should the tool be unable to meet the specified speed requirements, it might switch to a CLA architecture instead. For such a simple circuit, the synthesis utility can perform this analysis and optimization within minutes. Alternatively, suppose that a designer needs an adder with an operating frequency of 1 GHz. The synthesis program might begin with a CLA implementation and realize that it can operate at 2 GHz. With positive setup slack, the tool can then optimize the circuit for lower active power, lower static power, or both, by downsizing the gates or transitioning toward a lower-speed, lower-power implementation like an RCA.

Modern synthesis programs support a feature called register retiming, where registers are repositioned within the combinational logic to balance pipeline stages and achieve higher throughput [5]. Figure 1 and Figure 2 showcase an example of a pipelined circuit before and after register retiming, respectively. With the assistance of this feature, inserting additional

pipeline stages becomes trivial. Naturally, structural circuit designers may also need to insert additional pipeline stages to meet their target performance; however, manually partitioning the logic evenly across the stages is a non-trivial task, especially if the circuit was implemented with higher-level conditional or looping constructs. As a result, designers are often forced to insert the registers at natural separations in the logic, leading to unbalanced pipeline stages and significantly diminished speedup when inserting additional pipeline registers.

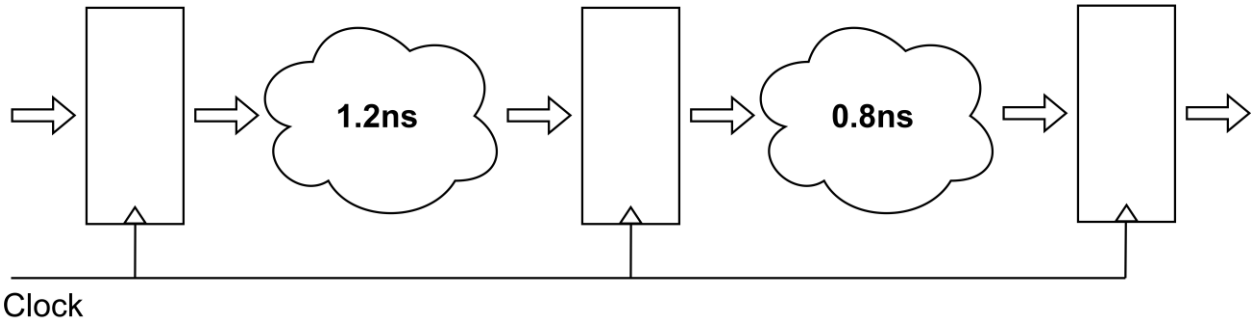

Figure 1: Unbalanced Pipeline

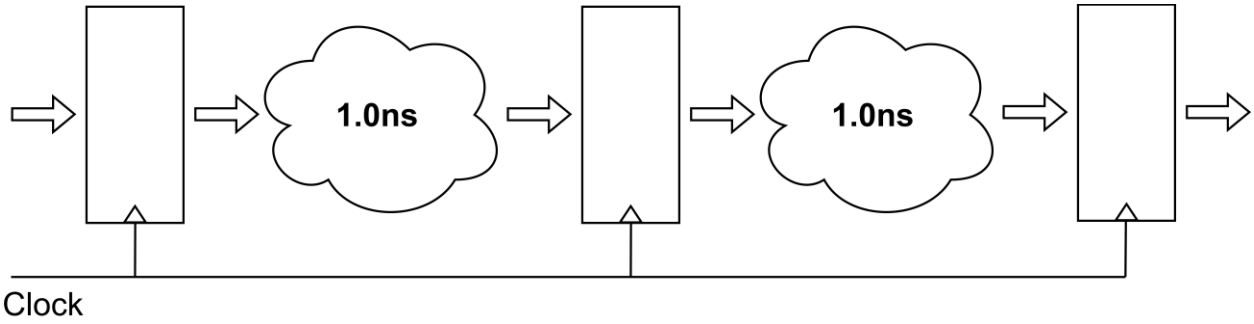

Figure 2: Balanced Pipeline after Register Retiming

In older technologies, circuit performance was bottlenecked by the delay through logic gates, while routing delays were relatively inconsequential [6] [7]. During this period, the performance of the resultant design could be estimated from the logical netlist alone before undergoing placement and routing. However, as feature sizes shrank, this trend started to reverse. Front-end designers would generate a netlist that met the target performance, but after placement and routing, the performance would be significantly degraded. This new discrepancy led to the

necessity of multiple iterations between the front-end and back-end design teams. Modern synthesis tools mitigate this issue through a process called physical synthesis, whereby the tool performs preliminary placement and routing throughout the synthesis flow [5]. Having accounted for both the gate and routing delay, the synthesis utility can generate designs that converge more quickly after full placement and routing. Without synthesis utilities, the structural circuit designer would have to predict the routing delays or launch full placement and routing to accurately measure the resultant circuit's performance. The alternative is to ignore the contribution of wire delay during design, resulting in poor quality netlists while complicating timing closure.

Given these considerations, structurally designing highly optimized and complex circuits, especially in deep submicron technologies, is practically impossible and often results in suboptimal outcomes. When assessing the quality of results (QoR) of structurally designed circuits multiplicatively—including speed, active energy, static power, and area—the QoR can be one or more orders of magnitude worse than functionally equivalent synthesized circuits.

2.2 Placement and Routing

The logical netlist produced from synthesis is not the final deliverable used in the tapeout of solid-state circuits, as the synthesized netlist needs to be physically implemented. The placement and routing step has major impacts upon the performance of the resultant circuit; thus, the industry standard for synchronous designs is to provide timing constraints to the tool to guide the circuit's implementation. The designer must specify which signals are clocks and how fast these clocks will be switching, along with input and output delays for the primary ports. The program will then use this information to place the gates, synthesize the clock tree, and route the interconnects in the circuit. Throughout the flow, the implementation utility will optimize the

circuit to meet the specified timing requirements while minimizing active power, static power, and area.

If timing analysis reveals that there is negative setup slack, a variety of optimizations will be employed to achieve timing closure. The utility may move cells on a failing setup path closer together, shortening the interconnect and reducing both resistance and capacitance. Higher layers in the metal stack or larger vias may also be employed to reduce resistance. If coupling capacitance, especially crosstalk, significantly contributes to the delay of the failing path, the implementation tool might increase the spacing between the failing net and its neighbors. Another optimization would be to replace gates along the path with alternatives that have higher drive strength, lower threshold voltage devices, or both. If upsizing gates proves insufficient, additional buffers can be added to split long routes or drive a large load capacitance more effectively. Conversely, the opposite transformations can be applied to eliminate negative hold slack.

If timing closure is achieved, the utility can also use these transformations to reduce active power, static power, and area. Shortening routes and reducing coupling capacitance help reduce dynamic power consumption, while downsizing cells saves dynamic power, static power, and area. Additionally, opting for higher threshold voltage (V_T) cells reduces static power dissipation. Without timing constraints, the tool lacks awareness of the critical paths, treating all nets as equally important, resulting in pseudorandom delays in the resultant circuit. This can lead to diminished performance and, especially critical in synchronous circuits, hold violations that render the circuit non-functional. Clearly, fully leveraging the capabilities of modern synthesis and physical implementation tools through application of timing constraints is paramount to producing high quality, highly reliable solid-state digital circuits.

2.3 NULL Convention Logic (NCL)

NULL Convention Logic (NCL) is a quasi-delay-insensitive (QDI) asynchronous circuit design style [1]. QDI circuits are sometimes referred to as *correct by construction*, i.e., they function correctly with no dependence on the delays through the transistors and interconnects. NCL circuits achieve their delay-insensitivity partly through their method of data representation. Boolean circuits utilize single-rail logic where each signal consists of a single wire, expressing a maximum of two values, *logic 1* and *logic 0*, both of which are used to express meaningful data. NCL circuits, on the other hand, utilize multi-rail logic, where each signal is composed of two or more wires. Table 1 illustrates a dual-rail encoding scheme, the most commonly employed multi-rail approach. Just as single-rail signals can express *logic 1* and *logic 0*, dual-rail signals can express *DATA1* and *DATA0*, respectively. However, the extra wire in each signal enables the expression of an additional value, NULL, which indicates an absence of useful data.

Table 1: Dual-rail Encoding

	DATA0	**DATA1**	**NULL**	**Illegal**
Rail 1	0	1	0	1
Rail 0	1	0	0	1

NCL circuits are composed using the 27 fundamental NCL threshold gates, implementing every threshold logic function with four or fewer variables. Table 2 lists the 27 threshold gates alongside their corresponding logic function and the required transistor count. With the exception of the last three gates, the naming convention of the threshold gates follows the pattern TH*mn*, where *m* is the threshold value and *n* is the number of inputs. For cells with weighted inputs, denoted by the inclusion of *w*, the weights from left to right correspond to the inputs in alphabetical order.

Table 2: Threshold Gates and Transistor Count [8]

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

As their names imply, THxor0 and THand0 implement the Rail0 logic for dual-rail, two-input XOR and AND functions, respectively, while TH24comp is utilized in completion detection logic. For reference in subsequent sections, TH*nn* cells are sometimes referred to as *C-elements*. At first glance, many of the logic functions in the table closely resemble the logic

function of common Boolean gates. The principal difference is that each threshold gate with a threshold of two or more has hysteresis, meaning that once the threshold is met, the output will remain high until *all* the inputs fall. Figure 3 presents an example of an NCL threshold gate with hysteresis, where the devices controlled by the feedback of the output node ensure that the output does not fall while one or more inputs are still high.

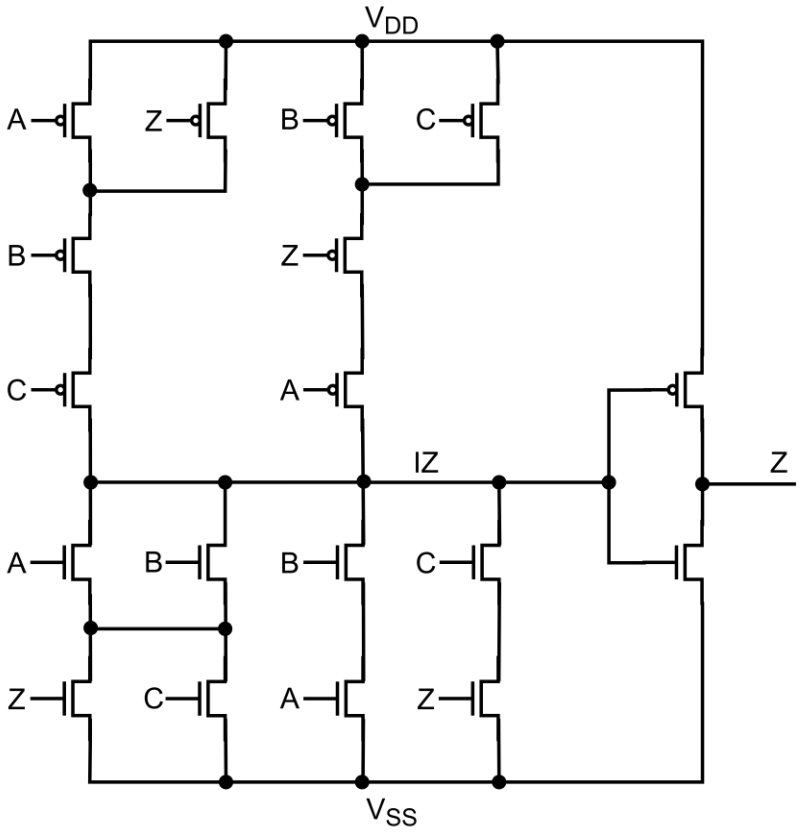

Figure 3: NCL TH23 Gate

Figure 4 demonstrates the structure of NCL circuits. Instead of synchronous registers that are controlled globally by a single clock signal, NCL circuits are pipelined with delay-insensitive (DI) registers that are controlled locally with handshaking signals. The DI registers establish delay-insensitivity by separating each pair of DATA waves with one or more NULL waves, as depicted in Figure 4.

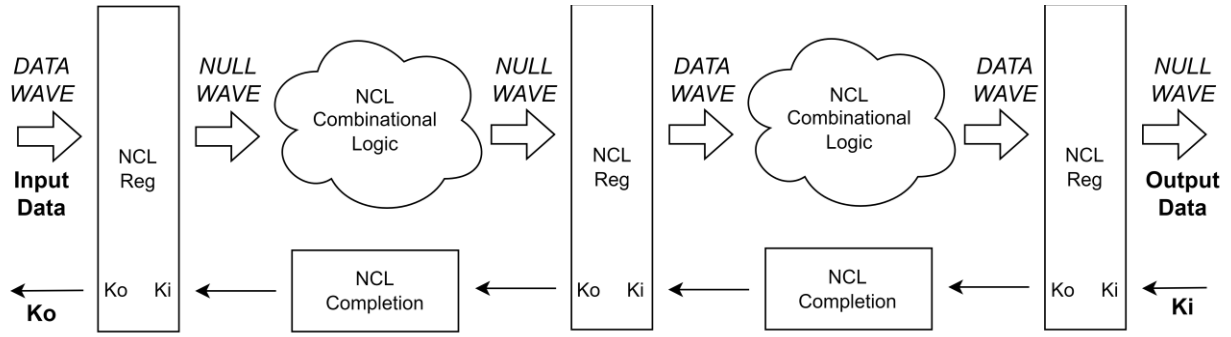

Figure 4: NCL Pipeline Architecture

This alternating pattern of DATA and NULL is accomplished through the usage of completion detection logic, which checks for the presence of complete DATA waves and complete NULL waves at the output of a given DI register. When each dual-rail signal at the output of a register is NULL, the completion unit detects the complete NULL wave and issues a request for DATA to the previous logic stage, allowing the upstream register to latch DATA. Similarly, the completion detection logic will request for NULL when there is a complete DATA wave at the register output, enabling the upstream register to latch NULL. In contrast to synchronous circuits, which assume that the data at the input of a register is stable and correct after a certain clock period, NCL circuits guarantee that the operation has completed through multi-rail encoding and completion detection while making no assumptions about the delay through the logic and routing. Thus, QDI circuit styles like NCL operate with average-case performance instead of worst-case performance. Moreover, no timing margins are imposed to account for delay variation caused by factors such as process corners, temperature, supply voltage, worst-case data patterns, routing resistance, capacitance, and aging.

Like Boolean synchronous circuits, NCL circuits can be throttled by both the rise and fall of gate outputs. The critical path in a synchronous circuit is often composed of rise propagation delay through some gates and fall propagation delay through other gates. To minimize the worst-

case delay across all data patterns, the devices implementing Boolean logic gates are sized to balance the rise and fall propagation delays. Each operation in an NCL circuit involves DATA wave propagation and NULL wave propagation, corresponding to rise propagation delay and fall propagation delay, respectively. Thus, akin to Boolean gates in synchronous circuits, NCL gates are sized to balance the rise and fall transitions.

One of the significant drawbacks of NCL is that the combinational logic must be designed according to input-completeness and observability. A circuit is input-complete if the outputs of the circuit cannot completely transition to DATA or NULL until all the inputs have transitioned to DATA or NULL, respectively. While hysteresis within NCL gates ensures input-completeness with respect to NULL, achieving input-completeness with respect to DATA is more cumbersome. The direct logical implementation of the desired functionality is insufficient and must be supplemented with additional logic to meet this requirement. Observability ensures that every DATA signal in a DATA wave is cleared to NULL before the next DATA wave arrives, preventing data corruption. When NCL circuits are designed with input-completeness and observability, they are effectively delay-insensitive.

2.4 Multi-Threshold NULL Convention Logic (MTNCL)

Multi-Threshold NULL Convention logic (MTNCL), sometimes referred to as Sleep Convention Logic (SCL), is another QDI asynchronous circuit design style that was developed to combine NCL with multi-threshold CMOS (MTCMOS) power gating [2]. This power gating is incorporated into every MTNCL threshold gate through the inclusion of two *sleep* transistors. The V_T of the devices is chosen carefully to strike a balance between performance and low static power dissipation. An example MTNCL gate is provided in Figure 5, where the higher V_T devices are enclosed in dashed rectangles. With the addition of the *sleep* transistors, NULL

waves can be generated at a single logical step with the assertion of the *sleep* signal instead of propagating through the logic as is the case for NCL circuits. Thus, the hysteresis transistors can be removed, simplifying the gate structure. Table 2 lists the number of transistors necessary to implement each MTNCL gate, which are notably fewer than the number required for the corresponding NCL implementation.

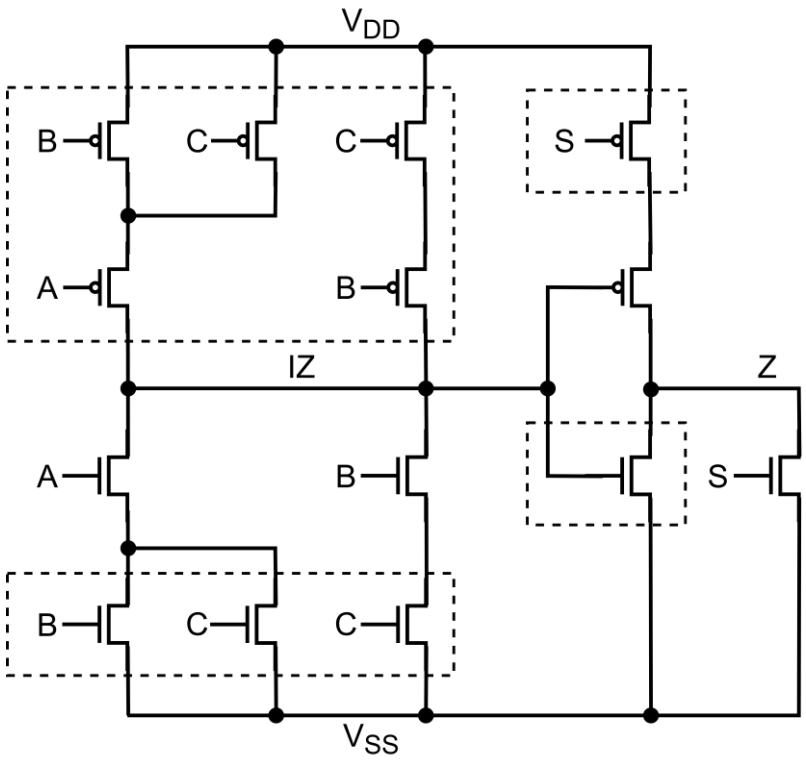

Figure 5: MTNCL TH23 Gate

Figure 6 illustrates the structure of an MTNCL pipeline stage. Similar to NCL circuits, MTNCL circuits include a completion detection unit with each register; however, the handshaking scheme employed by MTNCL circuits has some subtle yet important differences. When an MTNCL completion detection unit detects a complete DATA wave and the stage downstream is requesting for DATA (with a *Ko* of '1'), the current completion detection unit will produce a *Ko* of '0.' This transition has two effects: first, the *sleep* of the current register,

current combinational logic, and subsequent completion detection unit will fall, allowing DATA to latch in the current register and propagate through the current stage of combinational logic. Second, it serves as a request for NULL to the previous stage. Conversely, if a completion detection unit is put to sleep and the register downstream is requesting for NULL, the completion component will output a *Ko* of '1.' This initiates the opposite effects, generating a NULL wave by putting the current register, current combinational logic stage, and next completion detection unit to sleep while serving as a request for DATA to the previous logic stage.

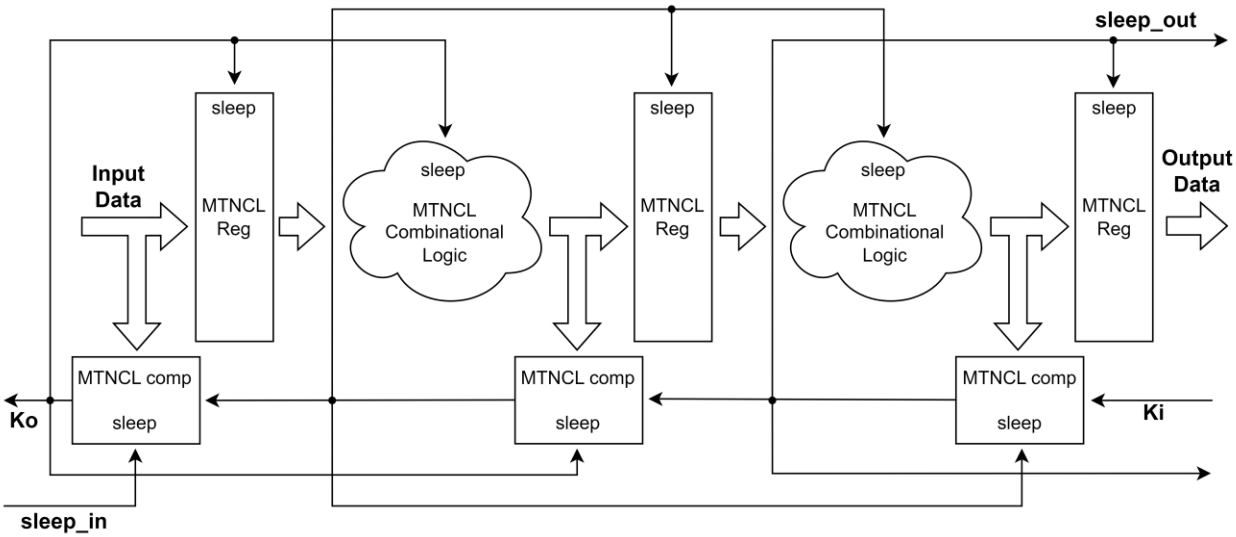

Figure 6: MTNCL Pipeline Architecture

Figure 7 depicts the structure of a 16-bit MTNCL completion detection unit. The first layer of logic consists of *TH24comp* cells, ensuring that one rail in each of the two dual-rail bits is high (i.e., both dual-rail bits are DATA). Following the first layer is an *AND tree* which validates that the entire multi-bit dual-rail input signal is in the DATA state. All but the last two gates in the tree are MTNCL cells, indicated in blue. The last threshold gate, in green, is an NCL *TH22n*. The hysteresis provided by the NCL implementation along with the final inverter are required to correctly implement the handshaking protocol. The *n* indicates that the threshold gate resets low and puts the associated stage to sleep during reset.

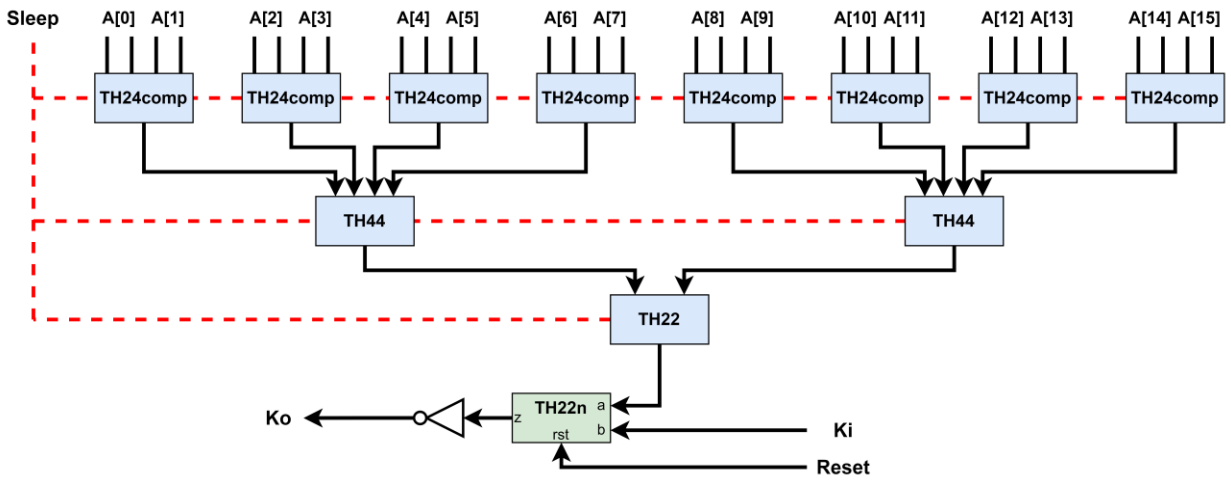

Figure 7: MTNCL Completion Detection Structure

The structure of an MTNCL register is illustrated in Figure 8, composed of one *TH12* cell per rail. The feedback ensures that DATA waves remain latched until the downstream stage requests for NULL, prompting the current completion component to assert sleep. This resets the register and generates the NULL wave in the next stage. The feedback can be implemented externally to a standard *TH12* cell, or a custom cell can be implemented to encompass the feedback path. As depicted in these figures, when an MTNCL circuit is idle, the combinational logic, completion detection, and registration are all put to sleep, resulting in very-low static power dissipation.

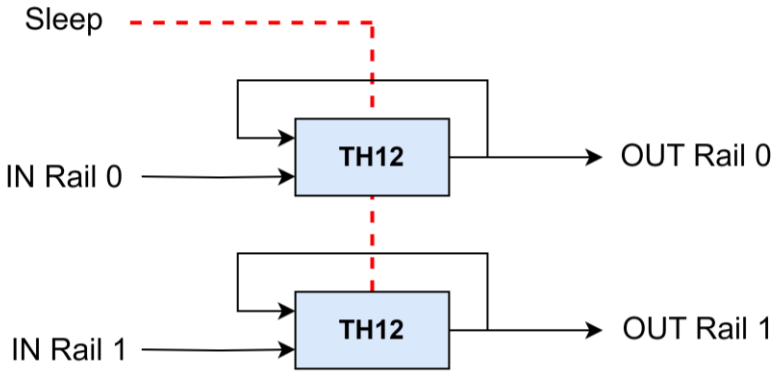

Figure 8: MTNCL Register Cell

One of the key challenges in MTNCL circuit design is the implementation of the *sleep* trees, which share certain characteristics with clock trees. In synchronous circuits, the clock tree is a

significant contributor to the power consumption of synchronous circuits, as it drives all flip-flops in the design and thus a substantial load capacitance [9]. Further, the clock signal switches with every operation, resulting in an activity factor of 1.0. Similarly, each *Ko* signal in an MTNCL circuit drives a *sleep* pin for every gate in the associated stage, resulting in very large load capacitances [10]. The *sleep* signal switches with each operation, resulting in the same activity factor of 1.0 [8]. While synchronous circuits utilize clock gating to reduce active power consumption, the *Ko* signals in MTNCL circuits only switch for the logic currently performing an operation. Due to its complexity and criticality, synchronous physical implementation flows dedicate an entire step to constructing the clock tree, i.e., clock tree synthesis (CTS). Due to the distributed nature of the clock, the implementation tool must use numerous buffers to minimize skew. Although *sleep* trees are more localized and skew is comparably less concerning, their implementation and buffering are nonetheless complex. An inadequately implemented *sleep* tree will result in diminished performance and unnecessarily high power consumption and area.

The replacement of NULL propagation with the parallelized *sleep* mechanism results in DATA wave propagation being almost always slower than NULL wave generation. For this reason, as noted in [11], the speed of MTNCL circuits depends solely on the speed of DATA wave propagation. In that work, skewing is employed in MTNCL gates to reduce rise propagation delay at the expense of fall propagation delay, leading to substantial improvements in performance, active energy per operation, static power dissipation, and area. As expected, when optimal MTNCL gates are deliberately skewed, they exhibit divergence in propagation delay across transition types. Table 3 demonstrates this discrepancy, showing the average propagation delays for a skewed, high drive strength MTNCL *TH12* cell in the TSMC 65nm technology. Notably, the fall propagation delay corresponding to the data inputs is approximately

2.6 times greater than the other delays. Despite the considerable delay, this transition type is irrelevant as the *sleep* signal drives the gate low, not the data inputs.

Table 3: Propagation Delays for an MTNCL TH12 Cell

Timing Arc	Average Propagation Delay (ps)
a, b ↑ → z ↑	58
s ↓ → z ↑	56
a, b ↓ → z ↓	160
s ↑ → z ↓	60

The performance of synchronous circuits is determined by the clock cycle delay, defined as the delay between two adjacent rising edges of the clock. Similarly, the performance of MTNCL circuits can be derived from the *Ko* cycle delay. Unlike synchronous circuits, the latency of each stage varies, so the performance of MTNCL circuits is constrained by the two adjacent stages in the design with the greatest combined delay. This equation holds true for designs lacking loops and implemented with slack matching [12], though the same approach can be adapted to evaluate the performance of these other circuits.

The *Ko* cycle time refers to the interval between two rising edges of a stage's *Ko* signal. This cycle can be further decomposed into a pair of smaller subcycles spanning two adjacent pipeline stages. The path tracing the first subcycle is presented in Figure 9, beginning with the completion component issuing a request for DATA and ending with the completion component issuing a request for NULL. Blue indicates the initial conditions of the circuit. This path can be deconstructed into four distinct components, itemized in the correct sequence below [13].

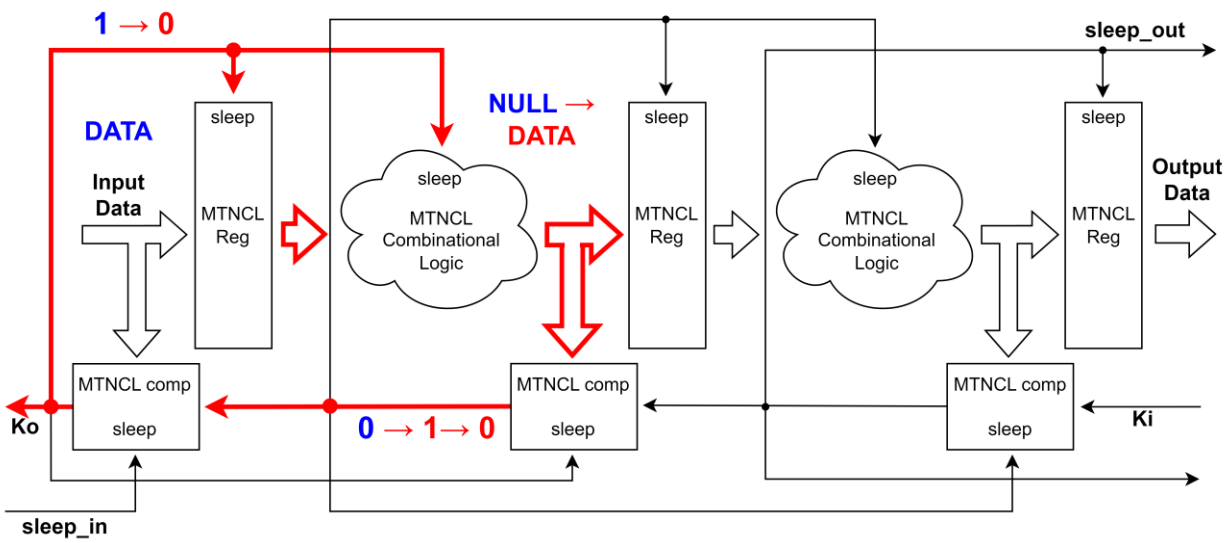

Figure 9: First *Ko* Subcycle Path

1. Back propagation of the rise of *Ko* from the output of *completion component (i)* to *completion component (i-1)*
2. Fall propagation delay through the final *TH22* of *completion component (i-1)*
3. Propagation of the fall of *sleep*, and the DATA wave propagating through the upstream register and combinational logic
4. Propagation delay for *completion component (i)* to validate the new DATA wave and issue a request for NULL, including the delay through the final *TH22* gate

The second subcycle, illustrated in Figure 10, contains the same four components, with a slight variation in sequence: the order is shifted forward by two positions, resulting in the sequence 3, 4, 1, then 2.

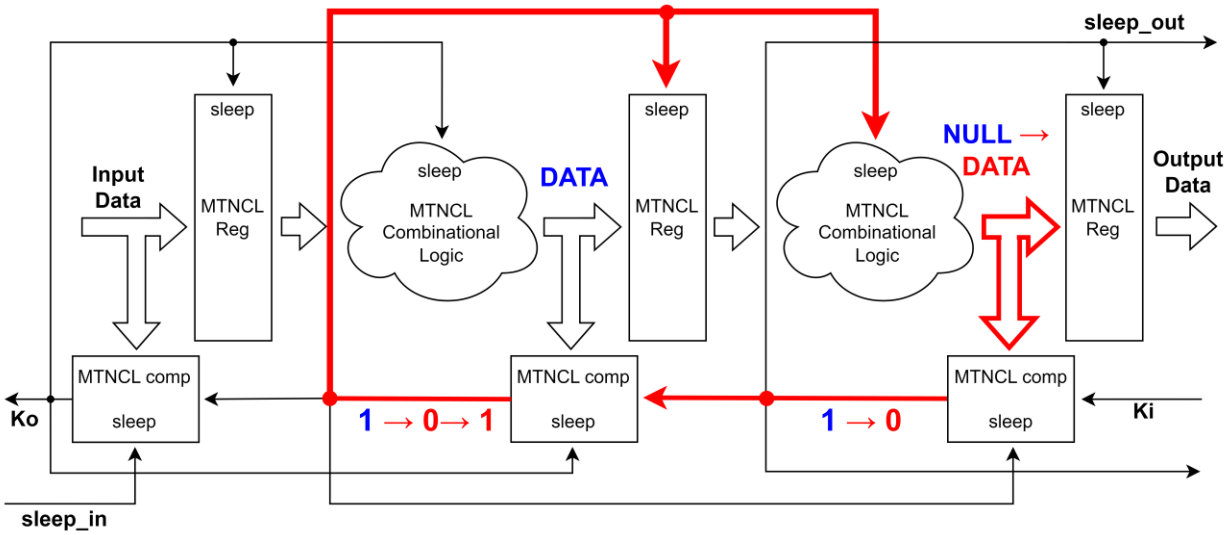

Figure 10: Second *Ko* Subcycle Path

1. Propagation of *sleep* and the new DATA wave through the adjacent register and combinational logic
2. Propagation delay for *completion component (i+1)* to validate the new DATA wave and issue a request for NULL, including the delay through the final *TH22*
3. Back propagation of the fall of *Ko* from the output of *completion component (i+1)* to *completion component (i)*
4. Rise propagation delay through the final *TH22* of *completion component (i)*

The differences between the two subcycles are limited to the direction of the transition in a single *TH22* gate and its associated *Ko*, specifically in components 1, 2 and 3, 4 in each subcycle, respectively. Assuming *TH22* and any applicable buffers are sized for balanced rise and fall propagation delay—a standard approach in most circuits—the two subcycles can be considered approximately identical. As mentioned earlier, NULL wave generation is faster than DATA wave propagation, so the performance of an MTNCL circuit can be determined by the DATA wave propagation delay alone. Thus, after factoring in the above two subcycles, the cycle time T_{cycle} of an MTNCL design can be expressed by Equation 1. T_{DD1} and T_{DD2} correspond to

the delay from issuing a request for DATA to receiving and validating a full DATA wave across two adjacent stages in the design, specifically the stages with the highest cumulative delay [13].

$$T_{cycle} = T_{DD1} + T_{DD2} \quad (1)$$

When MTNCL circuits are idle, the combinational logic, registration, and completion detection are put to sleep, resulting in very-low static power dissipation. MTNCL circuits require neither input-completeness nor observability, greatly simplifying their design. On average, MTNCL circuits are faster, consume less active and static power, and are smaller than functionally equivalent NCL circuits. However, these improvements come with a trade-off: MTNCL circuits are more timing-sensitive than NCL circuits.

2.5 MTNCL Timing Sensitivity

The gate-level and architectural changes made to NCL that improve power, performance, and area (PPA) also introduce timing sensitivity. While these relative timing assumptions have always existed in MTNCL, the increased impact of routing delays in modern process nodes has made them more pronounced. The first work identifying the timing sensitivity was [13], with further discussion in other works [8, 14, 15]. However, the summation of existing literature still fails to adequately cover all the relative timing assumptions, and only high-level, conceptual mitigation strategies are discussed. This work aims to fill this gap by providing an in-depth analysis of timing sensitivity in MTNCL. Quantitative and actionable solutions to these issues are presented in Section 3.3.1.

There are a total of four distinct relative timing assumptions in MTNCL's architecture, of which only two have been identified in existing literature. These assumptions and the consequences of their violation can be conceptualized as a 2×2 matrix of the associated wave

type (DATA or NULL) and either the introduction of early completion or the dual purposes of MTNCL's handshaking signals, as presented in Table 4.

Table 4: Classification and Implication of MTNCL Timing Assumptions

	DATA	**NULL**
Early Completion	DATA Loss	DATA Corruption
Dual-Purpose Handshaking	DATA Loss	DATA Corruption

The first assumption, presented in Figure 11, relates to DATA waves and the introduction of early completion and can be triggered by routing delays. Each register has a completion component that checks the register's input for complete DATA waves. However, since the signal splits to drive both the register input and the completion logic, there is no guarantee that the DATA wave will reach the register. In modern technologies characterized by substantial routing delays, especially if the register cells are placed too far from the fork, this assumption might not hold. Albeit very unlikely, since the completion component has already acknowledged a complete DATA wave, the next NULL wave might clear the current DATA wave before it arrives and successfully latches in the current register. Although this assumption is the least significant of those discussed, it is not documented in existing literature on MTNCL circuit design, to the author's knowledge.

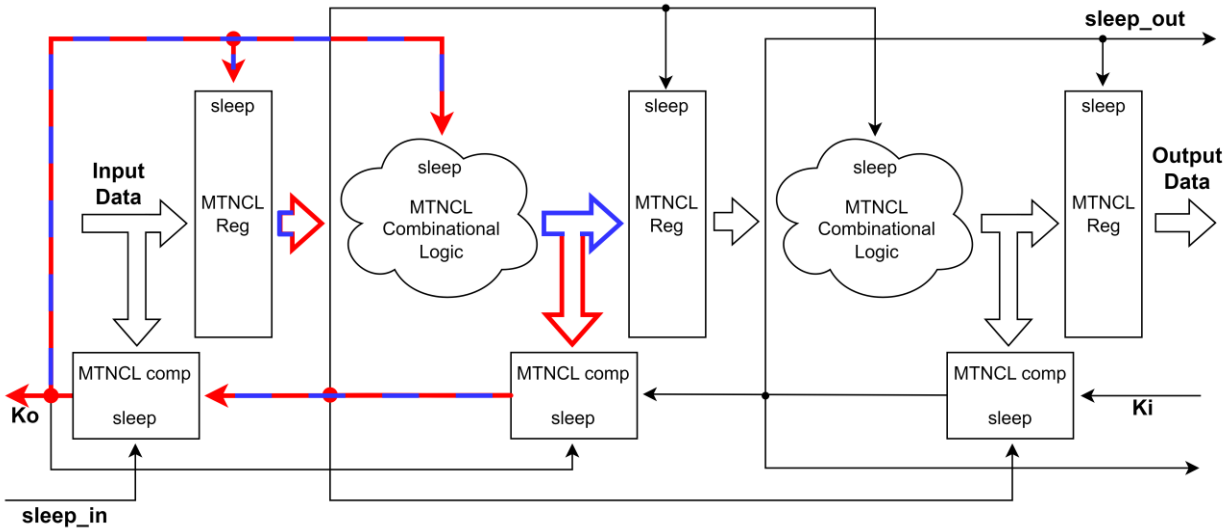

Figure 11: DATA Completion Race Condition

The second DATA-related relative timing assumption, as identified and discussed in [8, 13, 14, 15], pertains to the register's ability to latch the DATA at its input and is caused by the dual nature of the handshaking signals. Figure 12 illustrates the two paths involved in the race condition, while Figure 13 displays a plot of the signals involved in the race condition following a transistor-level simulation in the TSMC 65nm technology using a 1.0V supply voltage. For simplicity, only the five signals most germane to the race condition are plotted: the input of the register, the output of the register, the *Ko* from the register's completion component, and the *sleep* signals controlling the register and the datapath logic that drives the register input. Note that the scale of the y-axis for *reg_out* differs from the other signals.

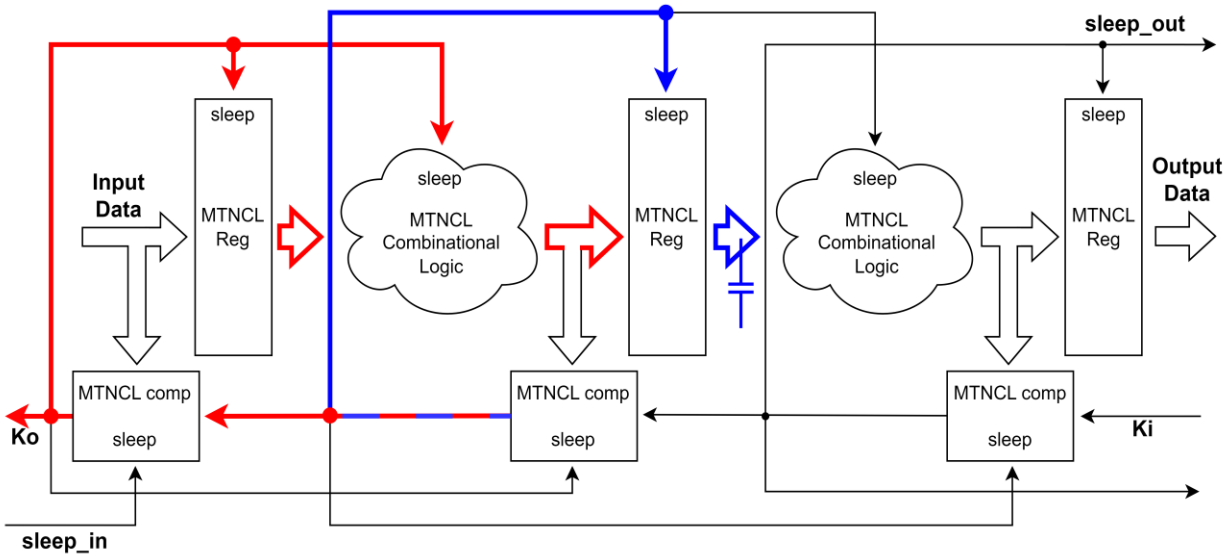

Figure 12: DATA Handshaking Race Condition

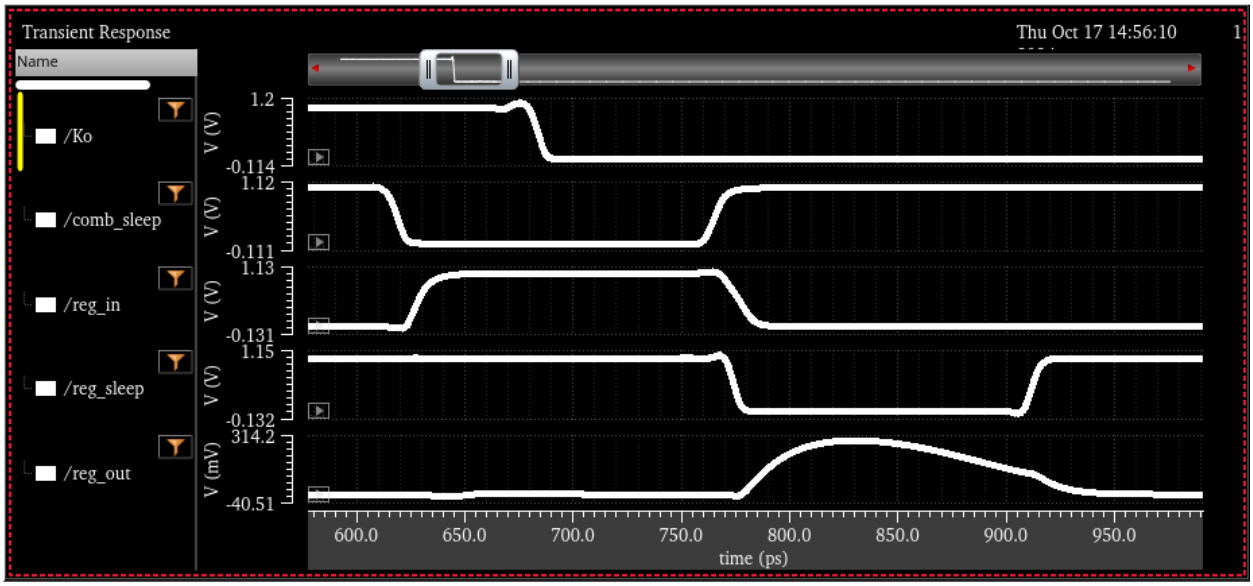

Figure 13: Transistor-level Demonstration of DATA Handshaking Race Condition

At the beginning of the waveform, the completion detection unit requests for DATA with a *Ko* of '1.' After some delay, the upstream completion detection unsleeps the combinational logic ahead of the register, enabling DATA propagation. When the operation on the DATA completes, the data input to the register rises. The completion detection unit preceding the register detects the valid DATA and requests for NULL by deasserting *Ko*, initiating the race. This transition

serves as a request for NULL to the previous stage (depicted in red) and unsleeps the current register so that it can latch the DATA at its input (shown in blue). The upstream stage responds to the request by asserting the *sleep* signal for the combinational logic preceding the register. However, due to the skew of the *Ko* signal, the register's *sleep* arrives too late. As a result, the register input falls before the previous DATA wave is latched, as the register output only reaches *284 mV*. This voltage is insufficient to engage the hysteresis in the register cell, preventing successful latching of the DATA value. Consequently, the downstream register will continue waiting for DATA indefinitely as the DATA wave was overwritten with NULL, stalling the entire circuit.

This phenomenon can be attributed to two sources. The first is the skew of the *Ko* signal, which means that the request for NULL to the previous stage cannot arrive significantly earlier than the *sleep* signal to the current register. The second factor is the load capacitance that the register drives, depicted by the blue capacitor in Figure 12. Naturally, a large load capacitance at the output of the register will degrade the slew of the output's transition, delaying the precise time the register latches the DATA at its input. This timing assumption is analogous to a hold-time violation in synchronous circuits, where the data value is not stable for the required amount of time after the active edge of the control signal. Moreover, rectifying this error might be impossible post-fabrication. The race condition was discussed and quantitatively analyzed in [8, 13, 14]. [8] and [14] advocate for using timing analysis with a commercial utility to ensure that the assumption is met but provide no details on an implementation of this timing analysis. [8] and [15] recommend adding a buffer to the output of each register to isolate the hysteresis mechanism from any potentially large load capacitance resulting from the register's fanout. While this method can certainly improve the register output slew, it does not entirely solve the

issue and would impose notable penalties on the performance, active power, static power, and area of the circuit. Ultimately, the only viable solution to this race condition is to conduct timing analysis with a capable tool.

The third of the four timing assumptions inherent to MTNCL circuits, only identified in [13], relates to NULL wave generation and arises from early completion detection. Similar to the first timing assumption, the completion detection does not ensure that the previous DATA wave at the input of the register has been cleared; however, the NULL-related assumption possesses some unique and concerning factors. For reference, a properly designed NCL circuit obeys input-completeness and observability, which ensures that every signal within a respective stage is guaranteed to have transitioned to NULL when the output of the stage is NULL. This behavior effectively *clears* the previous DATA wave. However, as previously mentioned, MTNCL replaces NCL's NULL wave propagation with the parallel generation of NULL through *sleep*, thereby creating the risk of a previous DATA wave not fully clearing. Figure 14 elucidates the paths involved in the timing assumption.

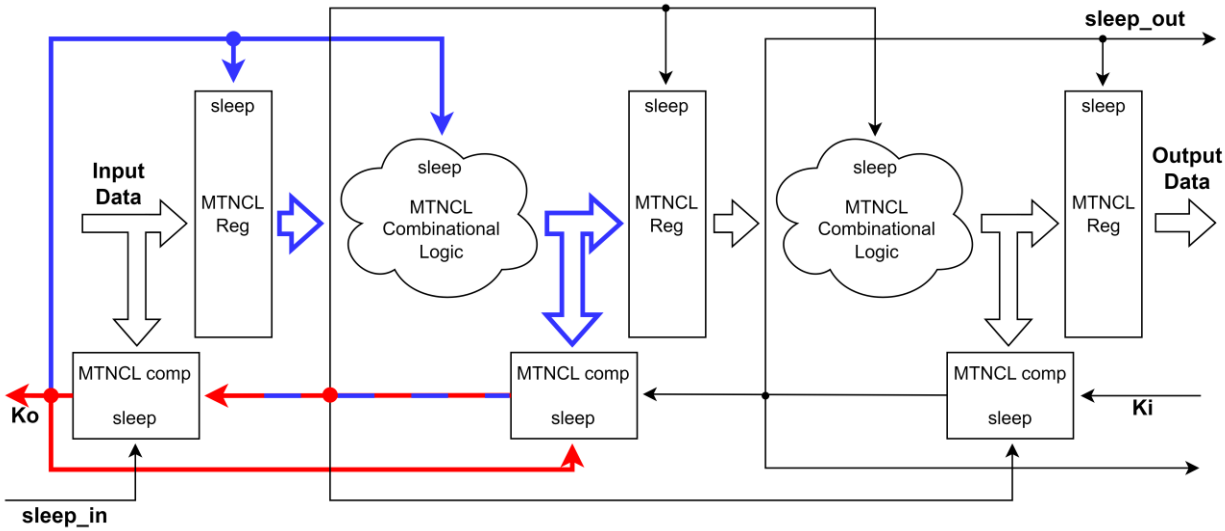

Figure 14: NULL Completion Race Condition

If the *sleep* tree is improperly designed, the completion detection logic could be put to sleep before the register and combinational logic, indicated by the red and blue paths, respectively. In such cases, when the completion detection unit subsequently requests for DATA, the register and combinational logic could be unslept before the previous DATA wave is completely cleared. This phenomenon violates the requirement that each pair of DATA waves be separated by a NULL wave, potentially corrupting the DATA and causing the circuit to produce incorrect outputs. Of the four MTNCL timing assumptions, this is less documented in existing literature. [13] indicates that it can be overcome by throttling the speed at which DATA waves are provided to the design. While this is a viable solution post-tapeout for simple circuits, where the speed of every stage is controllable off-chip, it would be difficult—if not impossible—to implement in a highly complex system. In addition, the MTNCL circuit would inherit one of the downsides of synchronous circuits, i.e., it would only function correctly if a specific, synchronous minimum period is obeyed. Furthermore, determining this minimum period would necessitate complex timing analysis, and additional margins would be required to account for delay variability.

The final MTNCL assumption, a novel contribution of this work, concerns NULL wave propagation and the dual-purpose handshaking scheme. This subtle race condition is illustrated in Figure 15. To aid in understanding, the initial conditions prior to the race condition are green, the first path is indicated in blue, and the second path—along with the transitions composing the race condition—are labeled with red. At the start of the race, two DATA waves are separated by a NULL wave. In response to the complete NULL wave, the middle completion component issues a request for DATA. However, due to the skew along this handshaking signal, it arrives at the upstream completion component before sleeping the middle register. Subsequently, the first completion component unsleeps its adjacent register and combinational logic. At this moment, both the first and second registers are transparent, so the DATA wave in the input stage propagates into the third stage, corrupting the downstream DATA wave.

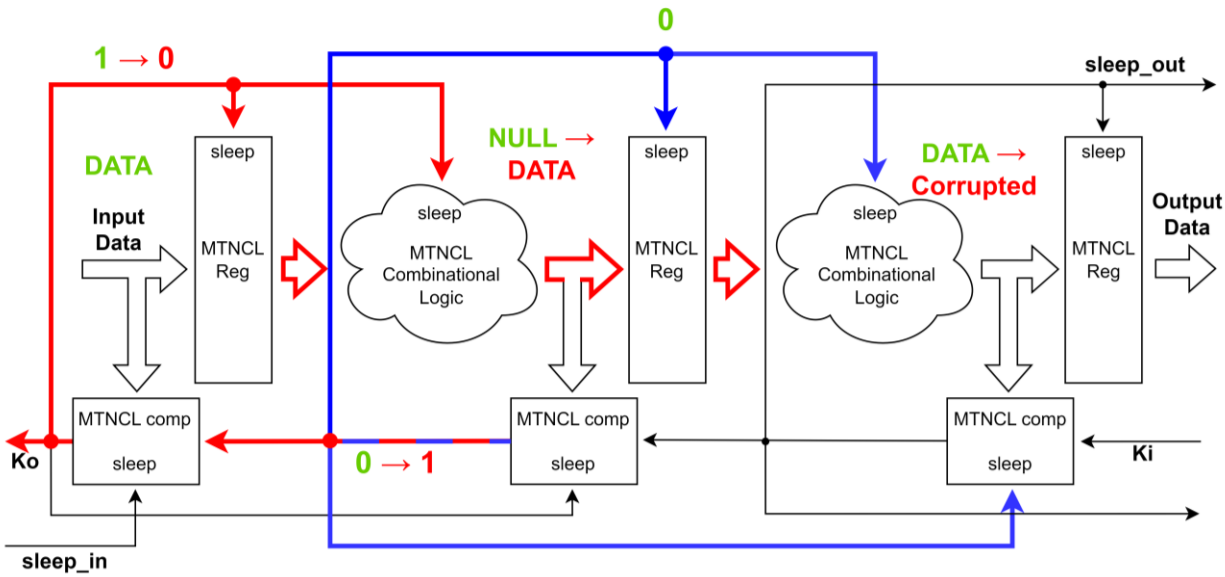

Figure 15: NULL Handshaking Race Condition

These four timing assumptions vary in likelihood from very improbable to very plausible. The author has encountered race conditions two through four in post-layout, transistor-level simulations. In fact, the DATA handshaking race condition is common enough at the input of

MTNCL circuits to necessitate adding a small delay between a request for NULL and the assignment of NULL to the input of the circuit. The absence of logic—and thus delay—between the testbench and input register yields a very fast feedback path, exacerbating the race condition. While these four race conditions are somewhat improbable, they must be properly mitigated to achieve a trustable, highly reliable MTNCL circuit.

2.6 Previous Work on Synthesizing NCL

Although there is ample literature covering the synthesis of other asynchronous circuit styles [12, 16-26], this work focuses on the synthesis of MTNCL circuits and how the process differs from that of NCL circuit synthesis. Historically, NCL circuits were designed manually using the aforementioned structural design approach because industry-standard EDA programs did not directly support NCL. Synchronous designs are ubiquitous, so commercial EDA software has been developed to support this design style. The tools expect that one or more global clocks coordinate the operation of the circuit, where each clock controls a given set of registers simultaneously. These registers, implemented with sequential cells, depend on the current inputs and the previous outputs, while the logic between registers is entirely combinational. Thus, EDA utilities assume that every sequential cell in the circuit is the beginning or end of a timing path.

In an NCL circuit, however, every gate is sequential due to hysteresis, so commercial EDA tools mistakenly identify each gate as a timing endpoint, prohibiting a direct approach to meaningful timing analysis. Similar to how flip-flops have constraints on the delay between transitions of the clock and data pins, NCL gates have similar requirements on the delay between a given input rising and another input falling. This attribute of NCL cells does not present any issues in NCL circuits since the signals switch high and low monotonically. However, the

synchronous-oriented tools do not understand NCL and are unaware that NCL circuits are not subject to those constraints.

Before meaningful synthesis and timing analysis can be attempted, Liberty models must be generated to describe the delay characteristics of the cells. The standard approach to generating these models is characterization using a program like Synopsys PrimeLib. This utility supports a feature known as functional recognition, where the tool autonomously extracts the logic function from the cell and automatically determines which tests are necessary to thoroughly model the delay, power, and noise characteristics. While functional recognition works well with standard Boolean and synchronous gates, it does not support NCL cells due to their complex functions [27]. Instead, the user must write an instance file for each cell, describing its sequential behavior.

Although timing analysis of NCL circuits using industry-standard utilities might be theoretically possible, modification to the Liberty models generated from characterization would be required to mask the sequential nature of NCL cells. This modified cell library, designed to be a compatible stand-in for the final cell library in commercial EDA tools, is sometimes referred to as an *image* or *virtual* library [23, 28]. For each NCL threshold gate, there exists a Boolean *image* gate that has the same function as the threshold gate but lacks hysteresis, such as an *AND2* cell for a *TH22*. In addition, commercial synthesis tools lack support for input-completeness and observability. Thus, if a designer attempts to construct NCL circuits using logic output from a commercial synthesis program after direct equation-level dual-rail expansion, the delay-insensitivity of the resultant NCL circuit would be compromised.

Despite these difficulties, multiple flows have been developed in an attempt to replace structural design of NCL circuits with automated synthesis. From a high level, most NCL synthesis flows follow the same 4-step process, conveyed by Figure 16. First, a single-rail

Boolean description of the circuit is generated. Next, dual-rail expansion is performed to transform the single-rail design into dual-rail. In the third step, the dual-rail description is mapped to the target cell library. Finally, completion detection and registers are added, and the handshaking signals are connected.

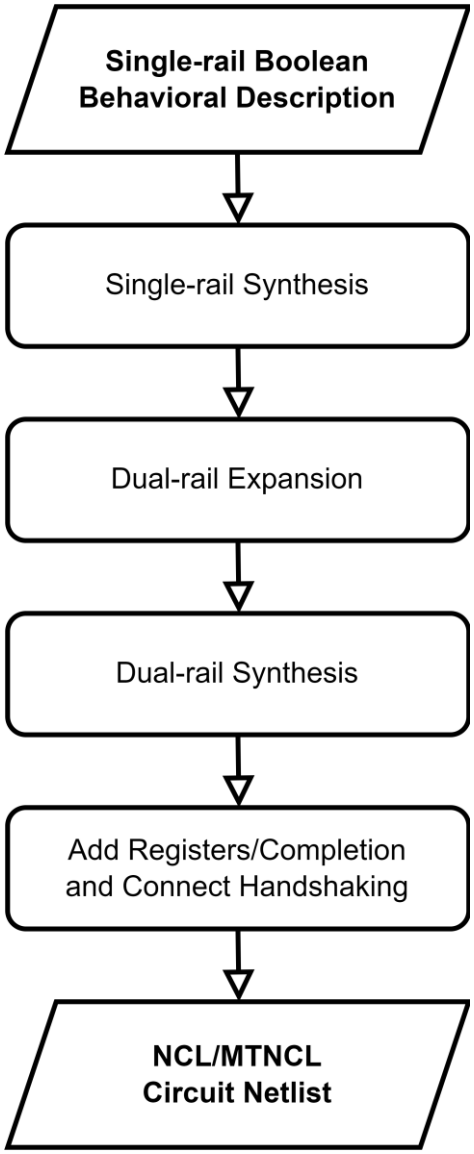

Figure 16: General NCL and MTNCL Synthesis Flow

Dual-rail expansion is an intuitive and straightforward process. Figure 17 shows a behavioral Verilog description of a single-rail Boolean full adder circuit, and Figure 18 presents the dual-

rail equivalent after expansion. The dual-rail version retains all the same ports and most of the original design's internal signals. Each signal is specified as dual-rail using a 2-bit array, where *bit 0* corresponds to *Rail 0*, and *bit 1* corresponds to *Rail 1*. The *Rail 1* assignments are conveniently identical to the assignments in the single-rail version once each signal is indexed to the proper rail, while the *Rail 0* equations are the logical complement after De Morgan's laws are applied. Where inverters are used in single-rail Boolean circuits, the *Rail 0* of the corresponding signal is used in dual-rail circuits, hence the inverter assignment is unnecessary and commented out. The existing NCL synthesis flows primarily differ in their implementation of the first, third, and fourth steps.

```
module full_adder(a, b, cin, sum, cout);
   input a, b, cin;
   output sum, cout;
   wire a, b, cin;
   wire sum, cout;

   wire cout_temp;
   wire cout_temp_inv;
   wire sum_temp;

   assign cout_temp = (a & b) | (a & cin) | (b & cin);
   assign cout_temp_inv = ~(cout_temp);
   assign sum_temp = cout_temp_inv & (a | b | cin);
   assign sum = sum_temp | (a & b & cin);
   assign cout = cout_temp;
endmodule
```

Figure 17: Boolean Single-rail Full Adder

```
module full_adder(a, b, cin, sum, cout);
   input a, b, cin;
   output sum, cout;
   wire [1:0] a, b, cin;
   wire [1:0] sum, cout;

   wire [1:0] cout_temp;
   // wire cout_temp_inv;
   wire [1:0] sum_temp;

   assign cout_temp[1] = (a[1] & b[1]) | (a[1] & cin[1]) | (b[1] & cin[1]);
   assign cout_temp[0] = (a[0] | b[0]) & (a[0] | cin[0]) & (b[0] | cin[0]);
   // assign cout_temp_inv = ~(cout_temp);
   assign sum_temp[1] = cout_temp[0] & (a[1] | b[1] | cin[1]);
   assign sum_temp[0] = cout_temp[1] | (a[0] & b[0] & cin[0]);
   assign sum[1] = sum_temp[1] | (a[1] & b[1] & cin[1]);
   assign sum[0] = sum_temp[0] & (a[0] | b[0] | cin[0]);
   assign cout = cout_temp;
endmodule
```

Figure 18: Boolean Dual-rail Full Adder

The flow described in [29] begins with synthesizing the single-rail description of the target circuit to a technology-independent GTECH library using a commercial synthesis tool. After dual-rail expansion, each single-rail cell is replaced by a dual-rail cell implementing the same logic function. Instead of ensuring the combinational logic obeys input-completeness and observability, completion detection is performed on every dual-rail signal in the design. These completion signals are then aggregated by a tree of C-elements to generate the handshaking signal to the previous stage. While this fine-grained completion detection frees the designer from the burden of input-completeness and observability, it imposes significant penalties to power, performance, and area. Mapping to GTECH cells generates a functionally correct design, but the optimization potential is limited to purely logical transformations since the synthesis utility lacks delay, power, and noise data for the cells composing the final circuit. Moreover, there is no way to approximate these attributes for the circuit until after synthesis.

The authors in [30] also utilize a commercial synthesis program to transform a behavioral single-rail description into generic logic. However, the input RTL must be written using a custom datatype called 3NCL. After dual-rail expansion, the circuit is optimized and mapped to threshold gates using a commercial synthesis tool through the usage of an *image* library. The additional optimization after dual-rail expansion provides an advantage over [29], but it is unclear whether the mapping utilizes cell-level delay, power, and area information. If so, it is also unclear whether the cell-level information of the *image* gates matches the characteristics of the real threshold gates used in the final circuit. While this flow supports pipelined designs, the registers must be manually added, leading to unbalanced stages and limited performance gains.

Another NCL synthesis flow, named UNCLE, is introduced in [31]. UNCLE supports commercial synthesis software, accepting Verilog as the input language; however, the user must instantiate the registers and completion logic in the RTL manually. This flow shares the same dual-rail expansion described in [29], where Boolean cells are replaced with their dual-rail equivalents. After flattening the netlist, a functional NCL circuit is ready, at which point UNCLE offers a few optimizations. Notably, it supports net buffering using non-linear delay model (NLDM) Liberty models to achieve a target transition time. While NLDM models are human-readable and intuitive, their accuracy is insufficient for deep submicron technologies. Consequently, NLDM was replaced by the far more complex and accurate Composite Current Source (CCS) and Effective Current Source Model (ECSM) models. Additionally, this buffering is applied to the entire circuit rather than targeting critical paths, leading to PPA degradation. While UNCLE supports latch balancing, the resultant pipeline stages would remain somewhat unbalanced as layout-level effects are not considered. UNCLE also includes custom scripts to perform the NCL-specific optimizations of relaxation and cell merging.

The flow presented in [32] requires the user to manually write a multi-rail, fully combinational, input-complete, behavioral description or to utilize another asynchronous synthesis utility for this initial step. After overcoming this barrier, [32] provides some unique and intriguing features. The behavioral description is physically synthesized with a commercial utility to *image* cells like [30]. Unique to this flow is the support of typical zero spacers, i.e., NULL, and one spacers within the same circuit. The flow utilizes positive unate return-to-zero cells, negative unate return-to-zero cells, positive unate return-to-one cells, and negative unate return-to-one cells. After synthesis, a custom script switches a subset of the cells to ensure correct operation in the presence of hysteresis. While the lack of register retiming will lead to unbalanced pipeline stages, access to this rich set of cells, typically not permitted in NCL circuits, can lead to significant PPA improvements over standard NCL.

2.7 Previous Work on Synthesizing MTNCL

To reiterate, unlike NCL, MTNCL gates do not employ hysteresis, and MTNCL circuits require neither input-completeness nor observability. For these reasons, despite the lack of direct support, synthesis of MTNCL circuits using commercial software is far more feasible than synthesis of NCL. Naturally, there exist some works attempting to synthesize MTNCL circuits using commercial EDA software.

The first synthesis flow for MTNCL circuits was documented in [13]. Synopsys DC is utilized to synthesize a single-rail version of the target circuit, which can be written in VHDL or Verilog. An *image* library with the *sleep* pins removed, which accurately models the delay, power, and area information of the target threshold gate library, is used for mapping. Then, the design goes through dual-rail expansion, followed by the replacement of each image cell with its associated threshold gate. Next, the synchronous registers are replaced with MTNCL registers,

and a commercial synthesis utility is used to further optimize the MTNCL combinational logic since MTNCL does not require input-completeness and observability. Finally, completion detection logic is structurally instantiated, and the *sleep* and handshaking signals are connected. However, without the use of timing constraints, the synthesis utility cannot adequately prioritize power, performance, or area.

The resultant circuit is complete and functionally correct, but this flow does not address the implementation and buffering of the *sleep* networks, a non-trivial necessity for MTNCL. The work also describes a method for estimating the performance of the resultant MTNCL circuit using marked graphs. However, the assumption that all gate delays are equal, coupled with the exclusion of routing delays, greatly limits the accuracy of its estimation. Additionally, the flow replaces every synchronous register with a pair of MTNCL registers—one reset-to-NULL register and one reset-to-DATA register—imposing an unnecessary overhead. Reset-to-DATA registers are only required in sequential circuits to generate an initial state at reset. Furthermore, while sequential designs require a minimum of three registers to maintain the current state value, each pipeline stage only needs a single register. Ordinary pipeline registers also do not require a reset; the standard completion detection unit puts the register to sleep during reset, generating a NULL wave by default. Non-resettable register cells have better power, performance, and area metrics. Consequently, the almost doubling of registers and ubiquitous application of resettable registers exact a significant PPA penalty. The most unique and potentially significant aspect of this work is its formal specification for how to correctly connect the handshaking signals in an arbitrary, complex MTNCL circuit.

The first step of [33] involves synthesis using Synopsys DC on an exclusively combinational single-rail design to generate a netlist of GTECH gates. Dual-rail expansion is performed, and

the GTECH cells are mapped to their MTNCL threshold gate equivalents. Unfortunately, this flow only supports logical optimization, with no consideration of the MTNCL cell library's delay, power, area characteristics, or routing delays. Similar to [13], buffering of the *sleep* network is also not addressed.

The flow covered in [8] also uses a commercial synthesis program to generate a single-rail circuit. Following this, it performs dual-rail expansion and executes a second round of synthesis using the same commercial tool to map to threshold gates. However, this flow is limited to combinational circuits only. If multiple blocks were synthesized and used in a pipeline, the stages would be unbalanced, sacrificing performance. Additionally, this flow structurally instantiates the completion detection logic and uses a custom python script to buffer the circuit based on the load capacitance of each net, leaving much room for optimization.

Like the other existing MTNCL synthesis flows, [15] utilizes an industry-standard synthesis utility to generate the initial single-rail circuit. Unlike other flows, however, [15] is limited to VHDL for design entry and requires substantial modification to the source synchronous RTL before the first round of synthesis. The first set of modifications ensures the synthesis utility infers multiplexers in place of flip-flops and latches. Next, to maintain the name of each multiplexer output, the signals are added as ports of the design. Naturally, these requirements introduce extra opportunities for errors. In addition, preventing the synthesis utility from inferring synchronous elements precludes register retiming, leading to unbalanced pipeline stages. With the necessary modifications to the input RTL complete, single-rail synthesis and dual-rail expansion proceed. Although the dual-rail expansion technique employed by this flow is logically correct, its performance does not scale well. Each stage's logic is written into a separate file and synthesized in isolation, meaning the synthesis utility is unaware of the overall

circuit's critical path, thereby forgoing many of the potential optimizations detailed in Section 2.1. Like many other flows, the dual-rail synthesis step utilizes an *image* library that lacks the *sleep* input. Following dual-rail synthesis, a register and completion detection unit are structurally implemented in each stage, and a buffer is added to the output of every register cell in an attempt to solve the DATA handshaking timing assumption described in Section 2.5. However, similar to most other flows, this work does not handle buffering of the *sleep* networks.