3. MTNCL RTL-TO-GDS FLOW

3.1 Library Preparation

While NCL gates require hysteresis, MTNCL gates implement MTCMOS power-gating. Boolean and synchronous standard cell libraries incorporate neither of these features within the combinational cells, so the user must develop one or more custom cell libraries. Custom IC EDA tools, such as Cadence Virtuoso, can be used to develop the schematics, verify the designs at transistor level, and generate the layouts for the MTNCL gates. Following export, a layout verification utility, such as Siemens Calibre, can be used to complete Design Rule Checking (DRC), Layout Versus Schematic (LVS), and Parasitic extraction (PEX). Subsequently, a characterization program like Synopsys PrimeLib characterizes the timing, power, and noise attributes of the MTNCL cells, outputting this information in the form of a Liberty model. To enable physical synthesis and accurate placement and routing of MTNCL circuits, an abstract must be generated for each gate. This abstract encompasses only the geometry of the layers leveraged during placement and routing, such as input/output pins, routing obstructions on metal layers, and the shape of the V_T implant layers. Virtuoso facilitates this abstraction process and can export the abstracts in Library Exchange Format (LEF) files.

The previously described steps are not unique to the development of MTNCL cells, as these steps are necessary for Boolean and synchronous cell design. However, additional steps are necessary to generate correct and optimal MTNCL circuit layouts. If an unmodified set of collateral is provided to a synthesis utility, the tool would interpret the *sleep* pin logically and attempt to use it to implement the logic equations of the target circuit. In this scenario, many of the *sleep* pins would be grounded, while others would be driven by datapath signals. Therefore,

attempting to simply replace the *sleep* pin connections in the post-synthesis netlist would corrupt the functionality of the circuit.

To prevent such issues, the existing MTNCL synthesis flows—and the one presented in this work—utilize an *image* library in which the *sleep* pin is removed from each MTNCL gate. Accomplishing this modification in the LEF file is straightforward; the pin section for *sleep* in each macro is deleted. However, the metal shapes still exist in the layout, so the *sleep* pin shapes should ideally be relocated to the obstructions section to accurately model the routability of the MTNCL cell.

Removing the *sleep* pin from the Liberty models is more involved and can be approached in various ways. The most straightforward method would be to directly modify the Liberty models after characterization; however, this can be labor-intensive and prone to error. The first modification required for this method is the removal of the pin section for *sleep* in each cell. Next, *sleep* would need to be removed from the function statement of the cell output pin. High-quality Liberty models employ an approach known as full state partitioning where the timing, power, and noise measurements are performed for every relevant combination of inputs, resulting in several values for each measurement. Consequently, removing the *sleep* pin would entail careful deletion of specific arcs and modification to the remaining arcs. Moreover, the characterization tool runtime would be unnecessarily wasted on the additional arcs.

An alternative approach would be to delete the *sleep* transistor from the gate prior to characterization. Although this approach produces a logically correct *image* cell, it leads to an inaccurate model of the real MTNCL cell's timing, power, and noise characteristics.

This work adopts a simple yet accurate approach by modifying the parasitic-extracted netlists prior to characterization. Specifically, the *sleep* port is removed from the port list in the Circuit Design Language (CDL) netlist, and every direct connection to *sleep* is tied to *VSS*. This modification is straightforward and scripted with basic find and replace operations. With *sleep* effectively disabled, the functionality of the desired *image* cell is achieved while the Liberty model accurately accounts for the additional resistance and capacitance associated with the *sleep* devices.

As previously outlined, MTNCL is unique in that the overall circuit throughput is not bottlenecked by the fall transition time of its constituent cells, unlike Boolean and NCL circuits. Thus, optimal MTNCL gates are skewed for faster rise propagation. If the synthesis utility has access to the inconsequential falling arcs, illustrated in Table 3, it would focus on optimizing non-existent critical paths. This misguided approach would hinder meaningful optimization of the MTNCL circuit.

To address this challenge, this work introduces a simple yet novel and highly effective transformation to the *image* cell Liberty models. The timing, power, and noise attributes for all arcs corresponding to fall transitions are replaced with the associated rise transition values, revealing the actual critical paths in the MTNCL circuit—namely, the DATA wave propagation—thereby enabling meaningful optimization.

Finally, as was the case for NCL cells with hysteresis, the Liberty models for any sequential cells that comprise the final MTNCL circuit must be modified so that the cells appear combinational to the EDA tools. Any sequential *preset* and *clear* timing arcs must be changed to *combinational_rise* and *combinational_fall*, respectively. Since all data signals in NCL and MTNCL circuits switch monotonically, all constraints between data pins—e.g.,

non_seq_setup_rising—are superfluous and should be removed to ease timing analysis. The functionality of non-synchronous sequential cells is specified in *statetable* statements. While not necessary for Genus and Innovus, other EDA tools may require that these statements be removed as well. Lastly, to more closely reflect the actual behavior of the underlying cell, a combinational *function* statement can be added to the output pin.

3.2 MTNCL Synthesis Flow

The MTNCL synthesis flow developed in this work consists of 14 steps organized under a top-level Makefile, as showcased in Figure 19. The flow takes standard behavioral synchronous Register-Transfer Level (RTL) as input. Notably, unlike some previous flows, no modifications are necessary, and the target design can be specified with any HDL supported by the chosen commercial synthesis utility. While Cadence Genus is employed for all synthesis operations in this work, any synthesis tool with equivalent functionality can also be utilized. Although this flow is split into 14 discrete steps, it follows the same general methodology as existing NCL and MTNCL synthesis flows, depicted in Figure 16.

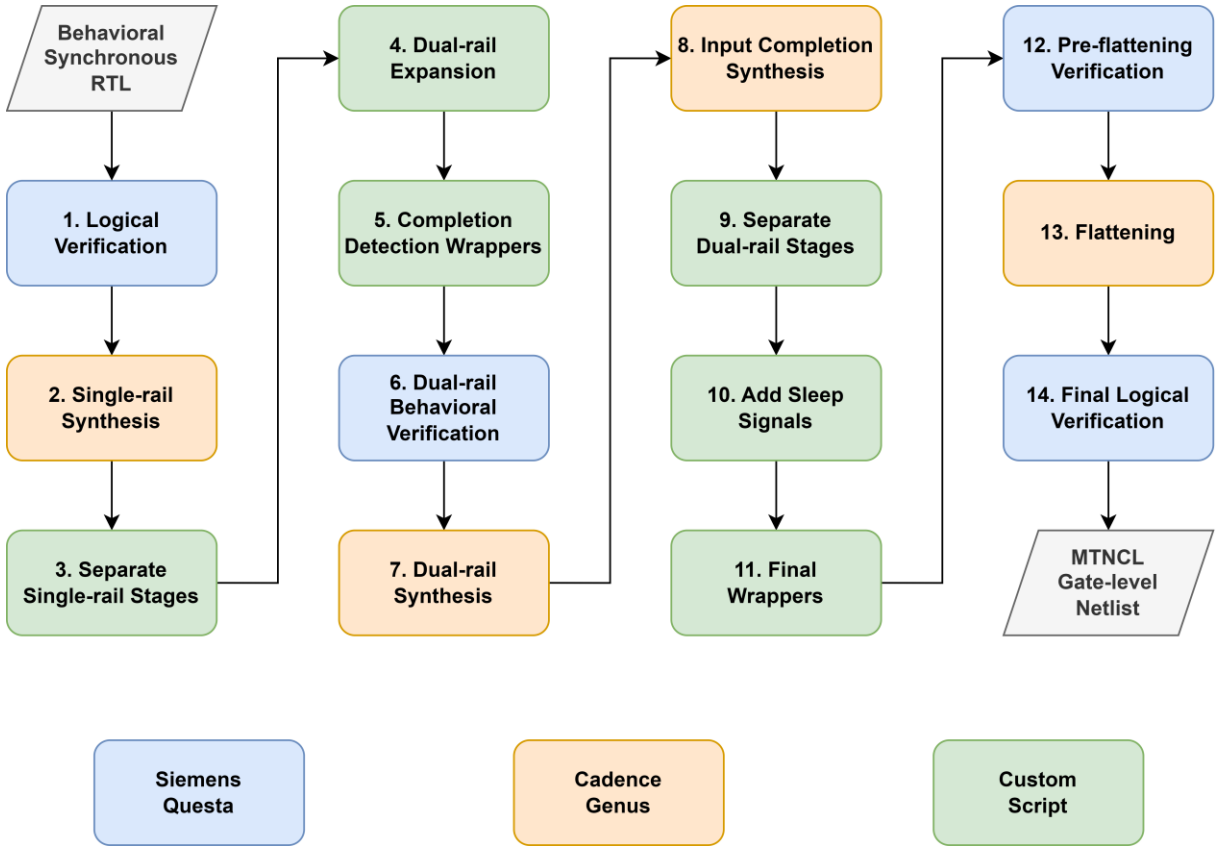

Figure 19: MTNCL Synthesis Flow

3.2.1 Single-rail Synthesis

The first step in the flow is to write a testbench for the design and logically verify the input RTL. This work leverages Siemens Questa and utilizes DO files to automate all logical verification steps. Naturally, the flow can be easily adapted to use an alternative simulator of the user's choice, such as Synopsys VCS.

The verified RTL is then provided to Genus for the first round of logic synthesis in step two. Since the flow accepts standard, unmodified synchronous RTL, combinational designs can be pipelined with an arbitrary number of stages through the application of register retiming and constrained to a user-selected clock frequency. This flow utilizes Boolean cell libraries from ARM for single-rail synthesis. Although somewhat trivial, it performs physical synthesis using

the library's LEF files, which enables quicker convergence in advanced technologies. After standard generic mapping, technology-specific mapping, and optimization, the Verilog netlist composed of Boolean and synchronous cells is generated.

3.2.2 Dual-rail Expansion

In the third step, the single-rail Verilog file is split into separate files for each logic stage using a short Java program. Step four entails dual-rail expansion of the single-rail logic of each stage, utilizing the same approach as many existing NCL and MTNCL synthesis flows. Specifically, an additional Java script doubles the width of each signal in the design to accommodate for the introduction of Rail 0, and a supplemental HDL file is incorporated into the design files, defining each Boolean cell as a functionally equivalent structural implementation composed of MTNCL gates.

Following dual-rail expansion, a third Java program generates a wrapper for each pipeline stage which encompasses the dual-rail netlists and behavioral completion detection logic in step five. These block-level wrappers are then instantiated between synchronous registers in a top-level wrapper and logically verified using Questa prior to the next round of logic synthesis.

3.2.3 Dual-rail Synthesis

Although the netlists provided to step seven are composed of Boolean cells, flattening the design using their structural descriptions yields logically correct dual-rail circuits composed of MTNCL gates. After reading in the design collateral and prior to initiating generic synthesis, all MTNCL cells are assigned the *dont_touch* attribute, ensuring that *only* the behavioral completion detection gets mapped to generic gates. Counterintuitively, disabling Genus's optimization attempts during the generic step yields circuits with significantly better PPA following dual-rail synthesis. More information regarding this observation and subsequent decision in the synthesis

flow can be found in Section 6.2. Upon completion of generic synthesis, the *dont_touch* attributes are rolled back to allow Genus to remap and optimize the circuit for enhanced performance and reduced power consumption.

The user can specify a target clock frequency for the synchronous registers in the top-level wrapper to optimize the MTNCL logic for the intended application. All existing MTNCL synthesis flows implement the completion detection logic structurally following dual-rail synthesis, as depicted in Figure 20, in which the critical path flows from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). Since completion detection is appended to the end of a logic stage, the critical path invariably flows through the completion detection, indicated by the red path.

Alternatively, synthesizing the completion detection with the combinational logic provides greater opportunity for optimization. When the completion detection output is tied to the synchronous registers alongside the data outputs, Genus becomes aware of the overall critical paths within the MTNCL circuit and can optimize them more effectively. As demonstrated in Figure 21, Genus will naturally interleave the combinational logic and the completion detection, tying the combinational logic output bit with the greatest delay toward the end of the completion detection tree. In the example circuit depicted in these figures, this approach reduced the critical path by two gate delays with no drawbacks. Furthermore, the shorter paths through the completion detection that are closer to the LSB, and which exhibit positive slack, can be optimized for lower power, reduced area, or both.

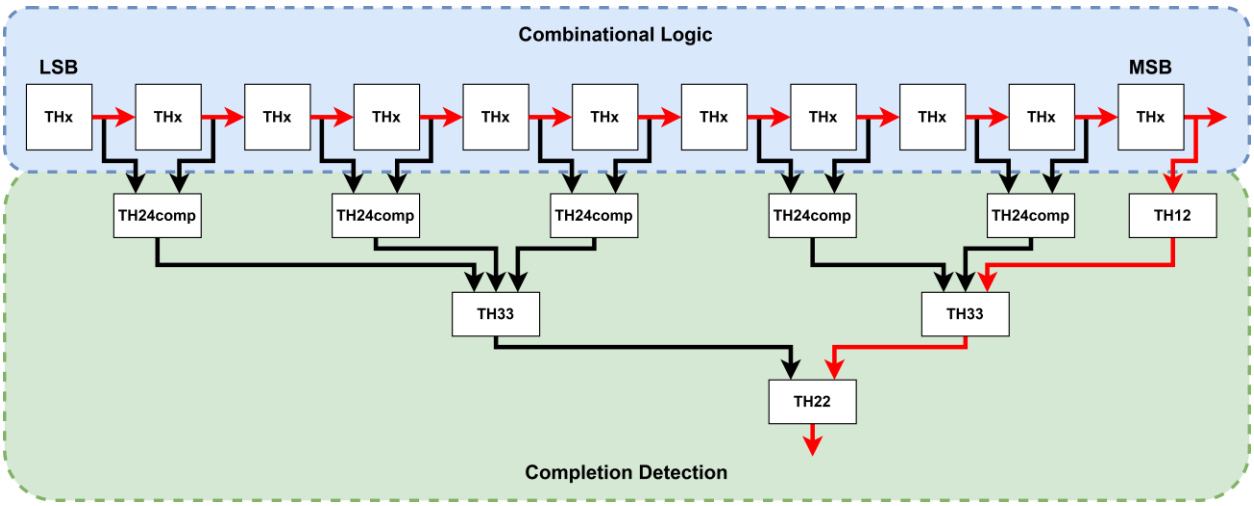

Figure 20: Structural Completion Detection

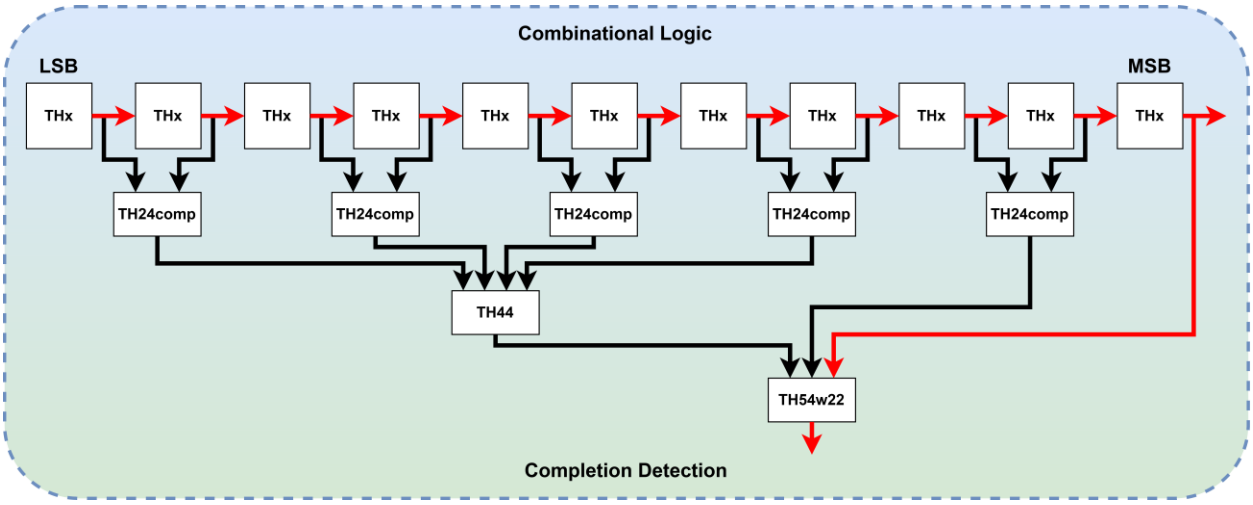

Figure 21: Synthesized Completion Detection

Although the first round of synthesis in step two can leverage all available features and optimization techniques offered by the chosen synthesis tool, the second round of synthesis on the dual-rail netlist must be carefully constrained to avoid introducing logical errors. First, register retiming must be disabled. Commercial synthesis programs like Genus understand neither dual-rail logic nor completion detection. If register retiming is enabled, Genus might repartition the combinational and completion detection logic, thereby corrupting the functionality of the resultant MTNCL circuit.