5-2019

Asynchronous Circuit Synthesis Using Multi-Threshold NULL Convention Logic

Nicholas Renoudet Mize
University of Arkansas, Fayetteville

Asynchronous Circuit Synthesis Using Multi-Threshold NULL Convention Logic

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Nicholas Mize
Bachelor of Science in Computer Engineering, 2017
University of Arkansas

May 2019
University of Arkansas

This thesis is approved for recommendation

Jia Di, Ph.D.
Thesis Director

Dale Thompson, Ph.D
Committee Member

Patrick Parkerson, Ph.D.
Committee Member

Abstract

As the demand for an energy-efficient alternative to traditional synchronous circuit design grows, hardware designers must reconsider the traditional clock tree. By doing away with the constrains of a clock, asynchronous sequential circuit designs can achieve a much greater level of efficiency. The utilization of asynchronous logic synthesis flows has enabled researchers to better implement asynchronous circuit designs which have been optimized using the same industry standard tools that are already used in sequential synchronous designs. This thesis offers a new flow for such tools which implements the MTNCL asynchronous circuit architecture.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Since the invention of Very High-Speed Hardware Design Language, or VHDL, and Verilog in the 1980s, logic synthesis has played a key role in the creation of gate-level design representations from high-level descriptions. Such logic synthesis tools have become standard in the creation of synchronous circuit designs that implement an oscillating clock signal to synchronize data flow. Unfortunately, these tools have become so focused on supporting synchronous architectures that asynchronous designs tend to have no well-known flow that can be used to create gate-level hardware from a high-level description. This leads to the design of an asynchronous logic circuit being described in a structural way which is not ideal due to the lack of optimizations that can be performed by synthesis tools. By developing a flow for asynchronous synthesis, designers can be less strict in their methods of designing components in an asynchronous circuit and allow the synthesis tools to make the gate level mapping on their behalf. This leads to less leakage and dynamic power being used for a design that performs identically to the behavioral model.

Behavioral level models are traditionally the source of synthesized circuit designs. These models are created using a hardware description language to behaviorally describe the way data should flow from input to output and to describe the Boolean logic operations that should be applied to the inputs. Boolean algebra can be used with individual data bits to create components which are instantiated in a design to create a hierarchy of components that are configured to perform computations on data inputs. In VHDL, components are referred to as entities and in Verilog they are specified as modules.

For the execution of asynchronous logic code to be implemented in behavioral code, modifications need to be made. For example, the flow of data between clock cycles requires a special synchronous logic cell known as a Flip Flop or Latch to hold the values of data bits in a design for a certain amount of time. Since there is no clock signal to synchronize with in asynchronous circuits, these synchronous logic cells cannot be mapped to the custom asynchronous library cells. Therefore, for any asynchronous architecture to become synthesizable, a flow is created which can be automated to perform the various modifications needed.

This thesis provides a unique flow for the construction of a gate-level netlist from a VHDL behavioral model. Chapter 2 will provide the necessary background for behavioral synthesis, MTNCL architecture, and asynchronous synthesis. The step-by-step implementation of MTNCL synthesis is defined in Chapter 3. Results of the MTNCL synthesis flow will are provided in Chapter 4. A 4-bit ripple carry adder (RCA), pipelined oscillator, 4-bit arithmetic logic unit (ALU), 8-bit array multiplier, and finite a state machine from the ISCAS'99/ITC'99 benchmarks library is presented [1]. A conclusion to the thesis is given in Chapter 5.

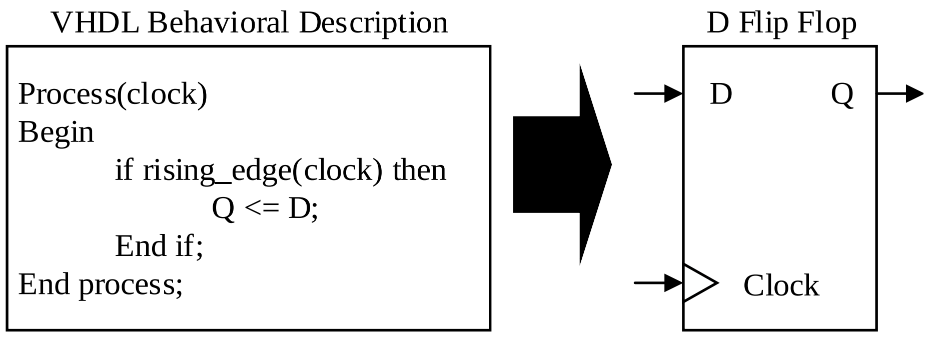
2. Background

This thesis is based on various previous works. Section 2.1 addresses behavioral code synthesis from a hardware design language like VHDL or Verilog. The asynchronous circuit design paradigm known as Multi-Threshold NULL Convention Logic (MTNCL), is introduced in section 2.2. An overview of previously developed asynchronous synthesis software tools are described in section 2.3.

2.1. Synchronous Behavioral Code Synthesis

Often, the functionality of digital circuits is described in a human readable hardware description language such as VHDL or Verilog. From these languages, the semiconductor industry has developed tools to transform high-level descriptions into low-level models describing the hardware to be implemented for a certain functionality. Like any other programming language, there are syntax rules that must be followed for the behavioral description to be compiled into a circuit or functional simulation. For example, VHDL and Verilog code is separated into sections that run sequentially and in parallel. Sequential code sections are described within a specific section known as process in VHDL. Code that is written in the process is performed from top to bottom of the entire process, like other popular programming languages like C++. Verilog has a similar label for sequential code operations which are performed in an always section. Code that is not included in either of these sections is assumed to execute in parallel.

Low-level models, typically generated as a gate-level netlist, are produced through a process known as logic synthesis. For synchronous designs, a clock signal is used to synchronize the data as it flows from input to output. Logic gate inference for synchronous circuits uses the clock signal to place D Flip Flops, D Latches, and other Boolean gates into a design depending on a behavioral description. Figure 1 provides an example of an inferred rising edge triggered D Flip Flop at the boundary of signals D and Q.



```
Process(clock)
Begin
        if rising_edge(clock) then
                Q <= D;
        End if;
End process;
```

Figure 1: Rising edge triggered D Flip Flop inference

For any circuit component, or construct, defined with a high-level description, there should exist a set of Boolean logic gates able to achieve the described functionality. This set of gates is optimized during synthesis to use the least amount of logic gates possible depending on the Boolean algebra reduction algorithms used in the synthesis tool while meeting the design constraints. Not all design constructs can be optimized using synchronous logic synthesis and software tool developers need to handle unsupported constructs such as setting the value of a signal outside of a system reset.

According to the IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, the three main categories for VHDL constructs are supported, ignored, or not supported [2]. For supported constructs, synthesis can correctly map to a hardware representation without issue. Ignored constructs are simply left out of the generated low-level model or gate-level description. Not supported constructs are left up to the discretion of the synthesis tool designer. Typically, synthesis tools make the decision to either ignore or throw out an error depending on whether the functionality of the circuit is dependent on the not supported construct. A construct relevant to this thesis is the combinational loop shown in Figure 2 by the red wire which is not gated by a clock signal.

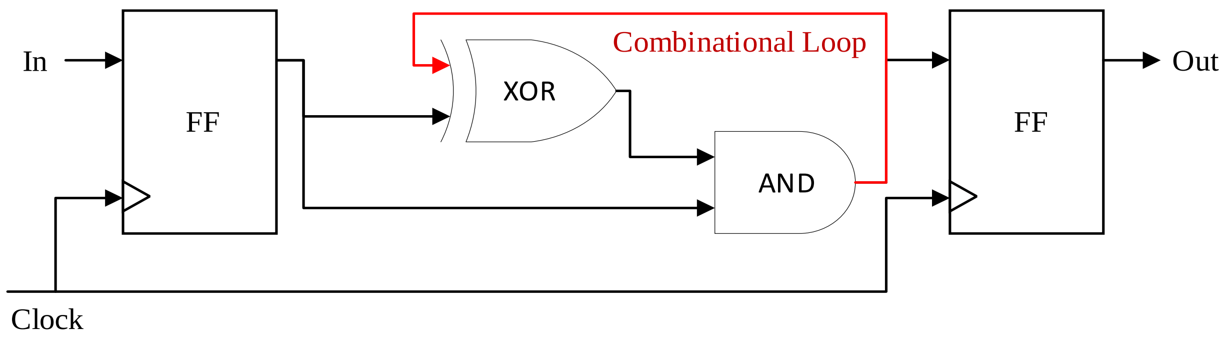


Figure 2: Combinational loop with D Flip Flops placed at input and output

Combinational loops cause issues in synchronous behavioral synthesis due to the possible unknown value presented on the red wire of Figure 2 under certain circumstances. Concerning the circuit in Figure 2, if the *In* signal is a value of logic ‘0’, then the combinational logic, consisting of a single XOR and a single AND gate, will output a value of logic ‘0’ on the red wire. This value will be stored by the *Out* rising edge triggered D Flip Flop (FF) on the next clock cycle. Conversely, if the *In* wire is a value of logic ‘1’, the combinational loop on the red wire between the two FFs will oscillate between logic ‘0’ and logic ‘1’. This oscillation will cause a value of either logic ‘1’ or logic ‘0’ to be stored by the *Out* FF on the next clock cycle. Ideally, every signal looping back from the output of a gate to the input of a previous gate needs an FF or D Latch to hold the value for at least one clock cycle to prevent oscillation. Regarding a synchronous design, the combinational logic of Figure 2 will need to be redesigned before behavioral synthesis can properly occur.

2.2. Multi-Threshold NULL Convention Logic (MTNCL)

Compared to their synchronous counterparts, circuits designed in an asynchronous way tend to benefit from high energy efficiency, more reliable operation, and simplified architecture due to the lack of a clock signal and corresponding clock tree. Multi-Threshold NULL Convention Logic (MTNCL) features quasi delay-insensitive (QDI) operation and ultra-low power consumption [3]. MTNCL itself is a derivative of NULL Convention Logic (NCL), a foundational design paradigm for component handshaking and propagation of data throughout the circuit [4].

Both MTNCL and NCL use custom gate libraries mapped from behavioral-level designs. There are 27 gates total for NCL and MTNCL alike. Gate functionality is similar regarding NCL and MTNCL with a slight naming difference. The letter "m" is appended to the end of MTNCL gate names to denote their multiple threshold voltage MTCMOS transistor design [3]. Table 1 names and describes the functionality of these gates using Boolean algebra.

Table 1: Basic MTNCL Threshold Gates [5]

MTNCL Gate	Boolean Function
TH12m	A+B
TH22m	AB
TH13m	A+B+C
TH23m	AB+AC+BC
TH33m	ABC
TH23w2m	A+BC
TH33w2m	AB+AC
TH14m	A+B+C+D
TH24m	AB+AC+AD+BC+BD+CD
TH34m	ABC+ABD+ACD+BCD
TH44m	ABCD
TH24w2m	A+BC+BD+CD
TH34w2m	AB+AC+AD+BCD

Table 1: (Cont.)

TH44w2m	ABC+ABD+ACD
TH34w3m	A+BCD
TH44w3m	AB+AC+AD
TH24w22m	A+B+CD
TH34w22m	AB+AC+AD+BC+BD
TH44w22m	AB+ACD+BCD
TH54w22m	ABC+ABD
TH34w32m	A+BC+BD
TH54w32m	AB+ACD
TH44w322m	AB+AC+AD+BC
TH54w322m	AB+AC+BCD
THxor0m	AB+CD
THand0m	AB+BC+AD
TH24compm	AC+BC+AD+BD

Like NCL, MTNCL threshold gates contain an M, N, and weight value following the "TH" notation. The first value after the "TH" is M, followed by N, and the weight or w. The M value specifies the number of inputs required to drive the output to Logic '1' while the value of N specifies the number of inputs the threshold gate supports. The weights specify the threshold value contributed by each input letter, A through D, with A being the first leftmost weight value. For example, the TH54w322m gate shown in Figure 3 needs the threshold value of each wire to sum to 5 before output Z becomes Logic '1'. Given the weight values in the naming convention, the weight of each input is A=3, B=2, C=2, and D=1. Unlike their NCL counterparts, MTNCL gates cannot hold their values due to the lack of internal feedback. Because of this, every MTNCL gate requires a *sleep* signal to be created by separate MTNCL completion logic. In this way, the outputs of the MTNCL gate can be controlled. The symbol of the MTNCL TH54w322m gate is shown in Figure 3.

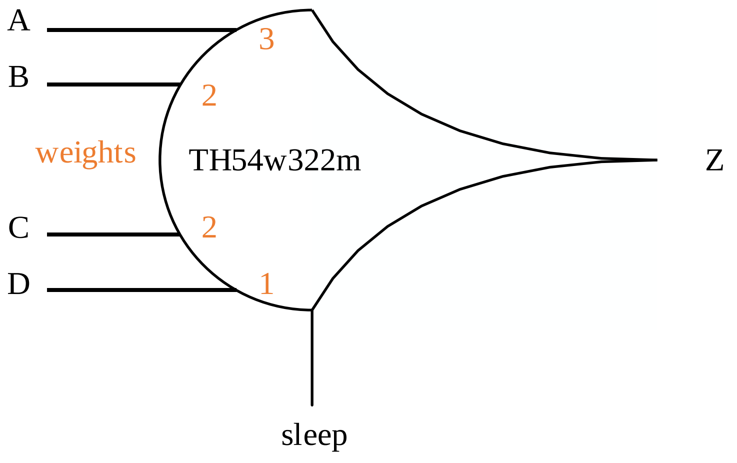


Figure 3: MTNCL threshold gate with weighted input

MTNCL makes use of a multi-rail logic encoding for every bit used in a design. The incorporation of multiple rails allows a dedicated MTNCL register and its corresponding completion logic to determine when the combinational circuit operation is complete. For this thesis, dual-rail logic encoding is chosen to encode each bit. When converting from a single-bit, the dual-rail terms DATA0 and DATA1 are used to represent the single-bit values of Logic '0' and Logic '1,' respectively. An example of this dual-rail encoding would be a signal, *A*, having two wires representing a valid DATA0 or DATA1 state, so that wire *A.Rail0* has a value of '1' when signal *A* is DATA0. Conversely, *A.Rail1* has a value of '0' when signal *A* is DATA0. Table 2 identifies all possible combinations of the dual-rail MTNCL encoding and the possible wavefronts of DATA0, DATA1, NULL, and INVALID.

Table 2: MTNCL Dual-rail State Encodings

	DATA0	DATA1	NULL	INVALID
Rail0	1	0	0	1
Rail1	0	1	0	1

Of the four possible dual-rail wavefronts, only three are expected to occur in normal circuit operation due to the mutual exclusivity of each rail [5]. An INVALID state is generally not expected and is indicative of a design flaw. The NULL state is a spacer in MTNCL designs and is only used to determine when a set of output values from the combinational logic has finished calculating and is ready to be stored. These wavefronts flow from input to output in MTNCL designs over time as in Figure 4.

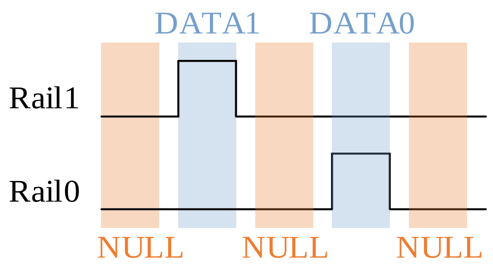


Figure 4: DATA and NULL propagation

Each dual-rail encoded state is used for handshaking between the MTNCL completion components and MTNCL register components. The MTNCL registers hold either DATA or NULL wavefronts until the corresponding combinational portion of the circuit is finished with its calculation. Due to the uncertainty of DATA and NULL arrival time at the input of every MTNCL register in a pipelined design, dedicated logic in the form of MTNCL completion is needed. MTNCL completion checks every input to the MTNCL registers to ensure either both Rail1 and Rail0 values are ‘0’ for a NULL state, or only one Rail1 or Rail0 value is asserted to ‘1’ for a DATA state. The state of the entire stage in an MTNCL pipeline will only transition when all wires in a stage are either DATA or NULL. In this way, the MTNCL completion logic for a stage can alert preceding stages in a pipeline when it expects to receive DATA or NULL.

The request-for-DATA (rfd) and request-for-NULL (rfn) signal is sent to previous stages via a *Kout* handshake signal. When the value of *Kout* equals Logic ‘1’, it is considered a rfd. When the value is Logic ‘0’, it is a rfn. For preceding stages, the rfd or rfn signal is read by that stage’s completion logic as a *Kin* handshake signal. MTNCL gates are configured to control whether a gate in the design is active or inactive via a *sleep* signal input. Gating the output thus lowers power consumption and leakage current. The *sleep* signal allows the completion logic of a stage to turn off components which are inactive during DATA and NULL wavefront propagation. For example, the MTNCL registers of a pipeline stage currently in a NULL wavefront as well as the combinational and completion logic of the stage after can be disabled. The pipelined MTNCL architecture is shown in Figure 5.

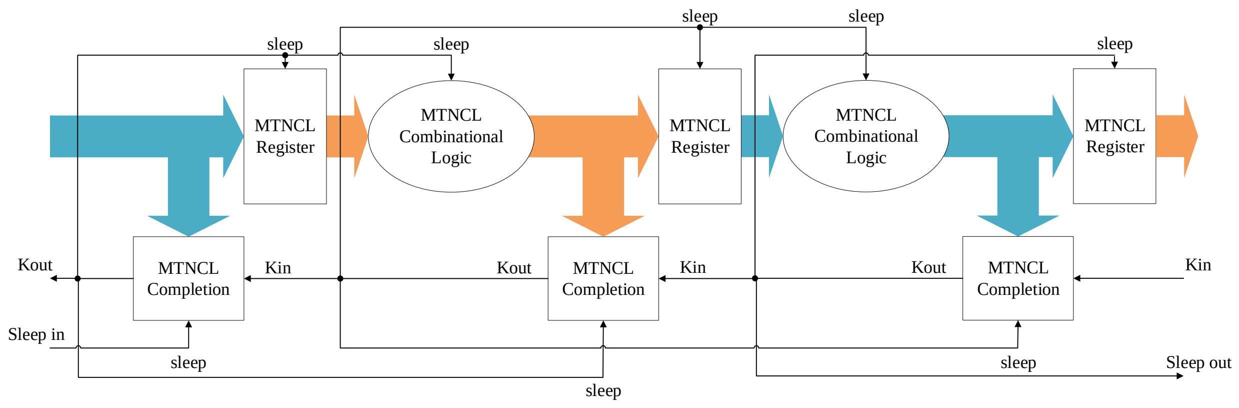


Figure 5: MTNCL Pipeline architecture with sleep completions and registers [3]

MTNCL benefits greatly from the creation of an asynchronous synthesis flow which allows hardware designers to create a gate-level design from a high-level description. Structural VHDL has been used in the past to create MTNCL circuits by instantiating various components manually in a hierarchy. While offering the best flexibility, this approach is not ideal for hardware designers with limited knowledge of the MTNCL architecture, and not scalable for large designs. Additionally, the optimizations implemented in modern synthesis tools cannot be used in designs which were created in a structural way. Using the MTNCL architecture, this thesis brings to light the possibility of mapping behavioral hardware design language code to an asynchronous MTNCL gate-level netlist.

2.3. Asynchronous Behavioral Code Synthesis

Asynchronous behavioral code synthesis tools can be designed to support the same hardware design languages as the synchronous synthesis tools. The main difference between the two methods is the lack of a clock signal to synchronize data flow for asynchronous designs. Since asynchronous designs are purely combinational, loops are supported and are expected in logic synthesis. There have been many attempts to create logic synthesis tools with the ability to synthesize from VHDL or Verilog behavioral descriptions into a circuit design implementing some form of handshaking for data flow as opposed to a clock signal and clock tree.

Previously developed tools vary in the way they handle the hardware design language descriptions and in the type of asynchronous netlist they produce. Because of the tremendous effort required to develop a new synthesis tool, researchers often try to use current industry standards instead. Likewise, this thesis uses the GENUS Synthesis Solution by Cadence to optimize and map a behavioral design to an MTNCL gate library [6]. For asynchronous synthesis tools using industry standards, modifications to the VHDL and generated gate-level netlist are applied at various steps in the flow. Various other asynchronous logic synthesis tools were explored which support complete or partial conversion from behavioral hardware description languages to a custom asynchronous cell library while simultaneously using industry standard synthesis tools whenever possible.

While varying in terms of complexity and ease of use, Phased-Logic [7], De-Synchronization [8], Proteus [9], Weaver [10], Unified NULL Convention Logic Environment (UNCLE) [11], and NCL-X [12] are examples of previously developed asynchronous synthesis tools. Phased-Logic implements a unique encoding scheme for each while doing away with the need for a NULL wavefront like NCL and MTNCL at the cost of increased complexity. De-Synchronization introduces the possibility of mapping standard cell library gates, rather than custom asynchronous library gates, to a design in an asynchronous way. However, this implementation is not QDI due to the usage of a combinational delay employed in each control logic. Proteus and Weaver tools both show similar results. They create asynchronous hardware from synchronous descriptions which can be faster than their synchronous counterparts. Both also exhibit larger area overhead due to the extra control logic needed. UNCLE and NCL-X were created to offer a flow for generating NCL designs given an input RTL behavioral design. Due to the similarities of NCL and MTNCL, these two tools offer the most insight into how an asynchronous MTNCL synthesis flow might be created.

The UNCLE tool, while not end-to-end in its RTL-netlist conversion methods, offers a great deal of flexibility to hardware designers who are familiar with the NCL architecture [11]. The main drawback of UNCLE is the need for designers to manually add NCL registers at the boundary of each combinational logic block. For designs implementing Finite State Machines (FSM) or feedback looping, this step is often complicated and can lead to design errors while also requiring designer familiarity with NCL. Because of this, the NCL-X tool is created to provide direct conversion of RTL to a gate-level netlist using several noteworthy steps. One such step is the grouping of a complementary logic wire (Rail0) with its original logic value (Rail1) through a method known as Dual-Rail expansion [12]. The now dual-rail design can then be checked for completion whenever the output of a logic gate branches to multiple gates using a completion logic handshaking signal. Multiple handshaking signals can be grouped together, typically with a Boolean AND function, to check whether all gates in a section of NCL combinational logic has completed its computation. With these techniques in mind, an automated synthesis flow for the generation of MTNCL designs is proposed in this thesis which inputs a high-level hardware design language description and outputs a gate-level MTNCL circuit.

3. MTNCL Synthesis

This thesis implements an asynchronous synthesis flow to make modifications to RTL, generic map, and gate-level netlists. Whenever possible, the GENUS Synthesis Solution by Cadence is used to optimize and map the design to the custom MTNCL gate library. For the purposes of automating the MTNCL synthesis flow, a python3 script, MTNCLSyn, was created. Using MTNCLSyn, any limitations which prevent GENUS from mapping to the MTNCL gate library can be resolved. This script automates as much of the MTNCL synthesis flow as possible while simultaneously allowing as much designer influence over the output gate-level netlist as possible. An overview of this flow is shown in Figure 6.

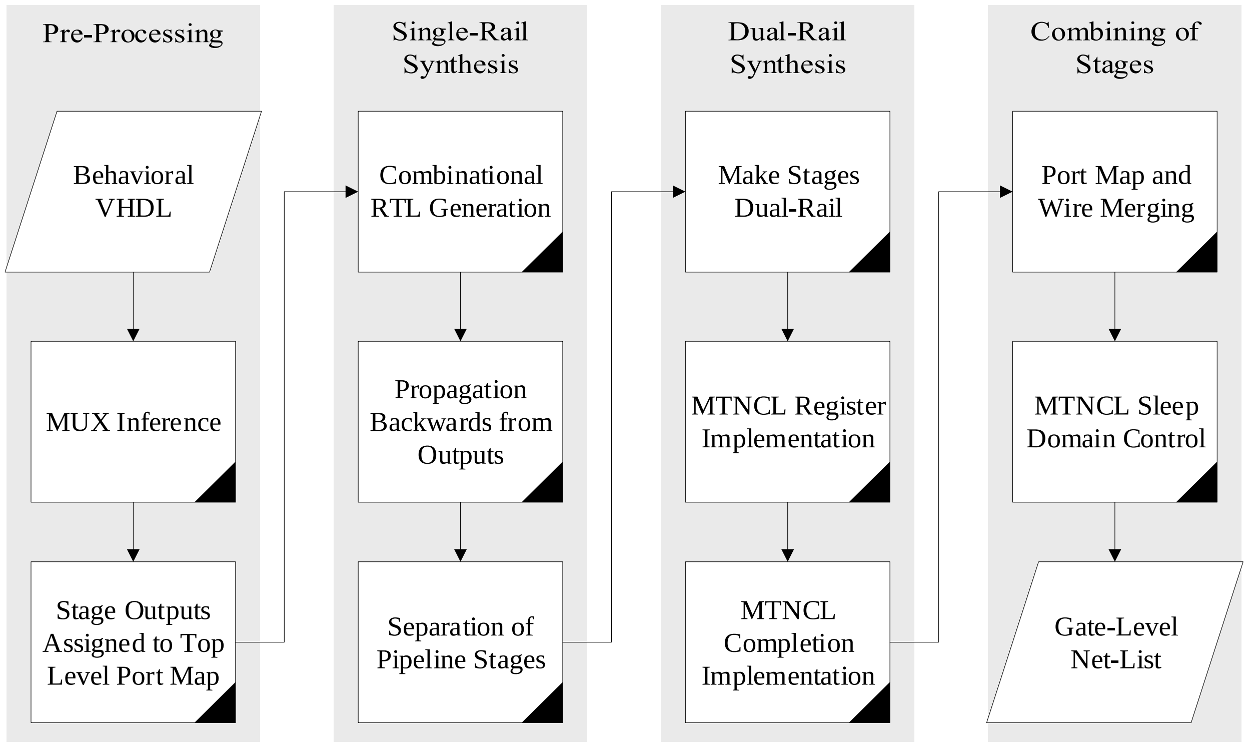

Pre-Processing
Behavioral VHDL
MUX Inference
Stage Outputs Assigned to Top Level Port Map
Single-Rail Synthesis
Combinational RTL Generation
Propagation Backwards from Outputs
Separation of Pipeline Stages
Dual-Rail Synthesis
Make Stages Dual-Rail
MTNCL Register Implementation
MTNCL Completion Implementation
Combining of Stages
Port Map and Wire Merging
MTNCL Sleep Domain Control
Gate-Level Net-List

Figure 6: MTNCL Synthesis flow from VHDL to gate-level netlist

Following are several sections covering the individual steps of MTNCL synthesis in detail. Section 3.1 addresses the initial setup required for the various tools and libraries. Modifications applied to the behavioral VHDL during pre-processing are explained in section 3.2. Single-rail synthesis is introduced in section 3.3. The steps pertaining to dual-rail synthesis are elaborated in section 3.4. Finally, the combination of all pipeline stages into a single gate-level netlist is explained in section 3.5.

3.1. Software Setup

The GENUS synthesis tool used in this thesis requires a custom MTNCL cell library to be input to GENUS as a Liberty library. This Liberty library was created by the SiliconSmart tool from Synopsys [13] to characterizes each cell for IBM's 45nm silicon on insulator (SOI) process. With this data, GENUS can buffer MTNCL cell output wires to ensure target rise and fall times are achieved. These vary depending on the capacitive load due to the number of MTNCL cell inputs driven by each wire. The functionality of each MTNCL cell is also provided to GENUS, so the RTL can be mapped to a gate-level netlist efficiently. It is possible to revert the buffering on each cell to obtain a generic gate-level netlist to use for manual configuration in simulations.

The libraries provided to GENUS only describe the combinational logic used by MTNCL. Due to their complexity, MTNCL registers and MTNCL completion components are handled separately from the GENUS synthesis tool. Additionally, D Flip Flops, D Latches, and other synchronous components are not synthesizable when the custom MTNCL cell library is used. To resolve this, all synchronous components are replaced with a logically equivalent MTNCL cell gate. Typically, the logical equivalent is either a TH22m, TH33m, or TH44m

MTNCL gate to implement an AND Boolean function. In this way, GENUS synthesizes the behavioral code by treating it as combinational logic.

3.2. Pre-Processing

Before the first round of single-rail synthesis can occur, several pre-processing steps must be applied to the input VHDL behavioral code. The first step is to remove any synchronous logic described by VHDL processes through a process called multiplexer inference. Multiplexer inference, or MUX inference, occurs when a VHDL process in a hierarchical design is modified to infer a MUX instead of a synchronous clock triggered component like a D Flip Flop or D Latch. For MUX inference to occur in the VHDL behavioral code, each read signal for the entire VHDL process must be placed into the sensitivity list. Additionally, each signal being written to must be assigned an initial value of Logic ‘0’. Because VHDL processes are read sequentially from the top, the initial value of Logic ‘0’ assigned to every written signal can be changed later in the process block. For example, an if-then statement using the clock as a conditional can reassign the value of an initially assigned signal if the clock signal is Logic ‘1’.

Each MUX has two inputs and one output with a single-bit select. The first input is the same input of the synchronous component being replaced, and the second input is a constant Logic ‘0’. The select signal is assigned to the clock. Figure 7 shows the replacement of a D Latch with a MUX and Figure 8 shows the before and after modification of the VHDL process for an 8-bit array multiplier.

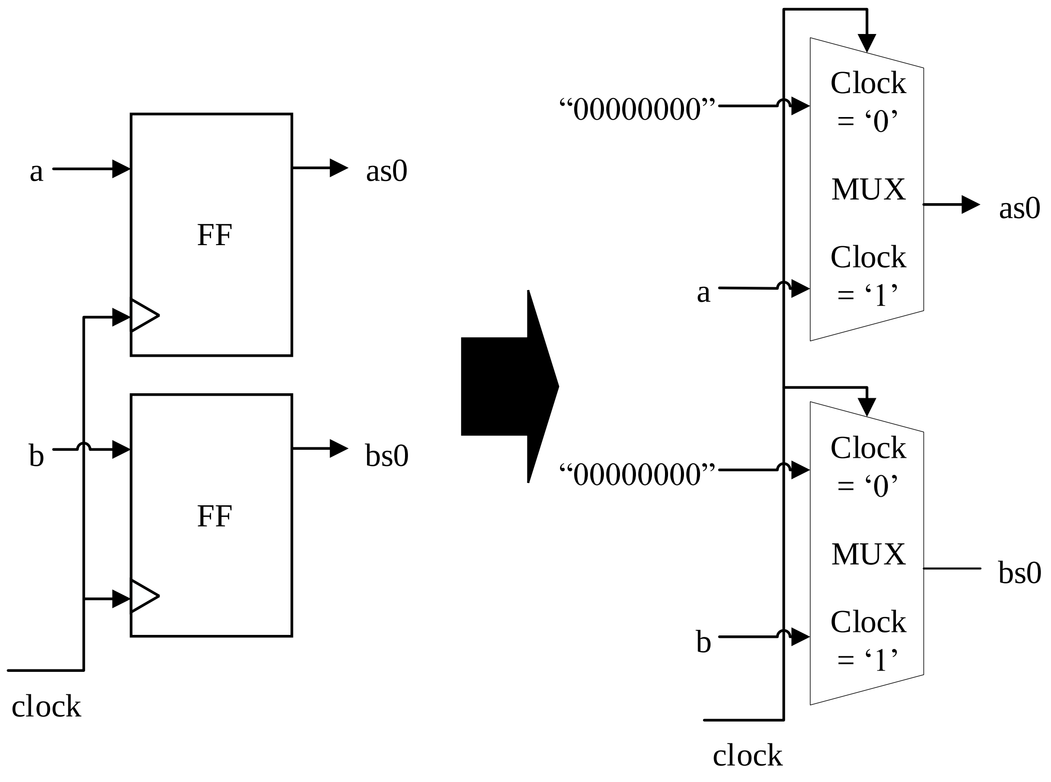


Figure 7: D Latch replacement and MUX inference

```
process(reset,clock)
begin
   if reset = '1' then
      as0 <= "00000000";
      bs0 <= "00000000";
   elsif clock = '1' then
      as0 <= a;
      bs0 <= b;
   end if;
end process;
```

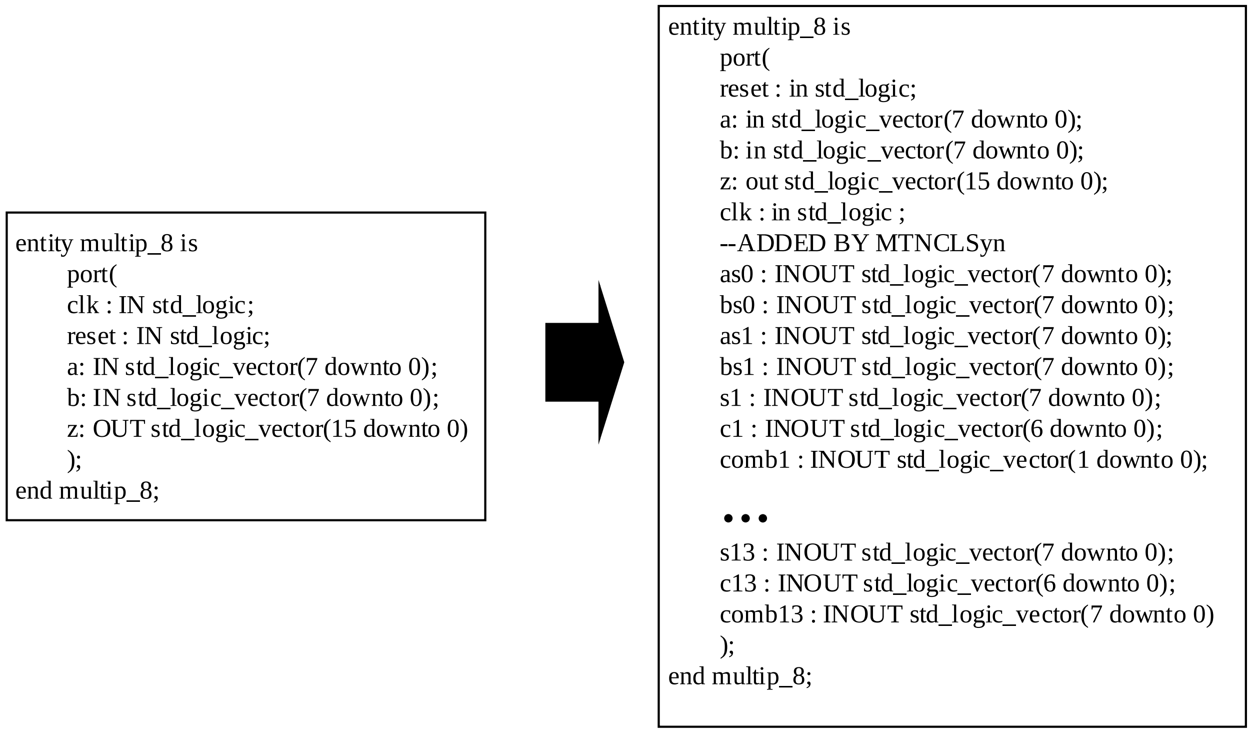
```
process(clock,reset,a,b)
begin
   as0 <= "00000000";
   bs0 <= "00000000";

   if  clock = '1'  then
      as0 <= a;
      bs0 <= b;
   end if;
end process;
```

Figure 8: D Latch replacement with a MUX in the VHDL process

The modified VHDL process, shown in Figure 8, will infer a MUX rather than a D Latch. Due to this replacement, the clock signal is no longer interpreted by GENUS as a clock. Rather, it enables the flow of data from *a* to *as0* and *b* and *bs0* depending on the value of the *clock* signal. In most cases, the inferred MUX is represented by a gate which implements a Boolean AND function for its inputs. From the custom MTNCL cell library, a TH44m, TH33m, or TH22m gate is used to AND the clock signal with inputs to the VHDL process depending on however many inputs are needed. For each VHDL process with a clock signal, the if-then reset condition is removed and added back after MTNCL gate-level netlist generation since it is no longer valid for MUX inference. Later in the MTNCL synthesis flow, these MUXs will be replaced with MTNCL registers after gate-level netlist generation.

While a MUX is enough to convert the flow of data in a synchronous VHDL behavioral model to an asynchronous flow, additional steps ensure the unique names of each signal assigned to every MUX output is not lost during synthesis. Valuable information, which can be used to determine from which MUX a signal was derived, is usually lost in the generated RTL from the first round of single-rail synthesis. Without knowing this information, it is difficult to determine where one pipeline stage begins and ends. Thus, it would be impossible to assign correct *sleep* inputs and outputs for each pipeline stage in the generated MTNCL gate-level netlist to perform the handshaking of DATA and NULL wave-fronts. To prevent this, every MUX output signal is re-assigned to the port map of the top-level entity. By converting the internal MUX outputs into primary outputs, the names of signals at the boundary between each pipeline stage can be preserved. Concerning components instantiated in a hierarchy with clocked VHDL processes, port maps are modified to append the additional pipeline stage output signals if they do not already exist in the port map as outputs. A unique identifier is also appended to each component’s signal name if more than one of the same components are instantiated by a VHDL entity. Figure 9 depicts the pipeline stage signals being appended to the top-level VHDL port map of an 8-bit multiplier.



```
entity multip_8 is
      port(
      clk : IN std_logic;
      reset : IN std_logic;
      a: IN std_logic_vector(7 downto 0);
      b: IN std_logic_vector(7 downto 0);
      z: OUT std_logic_vector(15 downto 0)
      );
end multip_8;
```

```
entity multip_8 is
      port(
      reset : in std_logic;
      a: in std_logic_vector(7 downto 0);
      b: in std_logic_vector(7 downto 0);
      z: out std_logic_vector(15 downto 0);
      clk : in std_logic ;
      --ADDED BY MTNCLSyn
      as0 : INOUT std_logic_vector(7 downto 0);
      bs0 : INOUT std_logic_vector(7 downto 0);
      as1 : INOUT std_logic_vector(7 downto 0);
      bs1 : INOUT std_logic_vector(7 downto 0);
      s1 : INOUT std_logic_vector(7 downto 0);
      c1 : INOUT std_logic_vector(6 downto 0);
      comb1 : INOUT std_logic_vector(1 downto 0);

      •••
      s13 : INOUT std_logic_vector(7 downto 0);
      c13 : INOUT std_logic_vector(6 downto 0);
      comb13 : INOUT std_logic_vector(7 downto 0)
      );
end multip_8;
```

Figure 9: Appending pipeline stage outputs to the VHDL port map

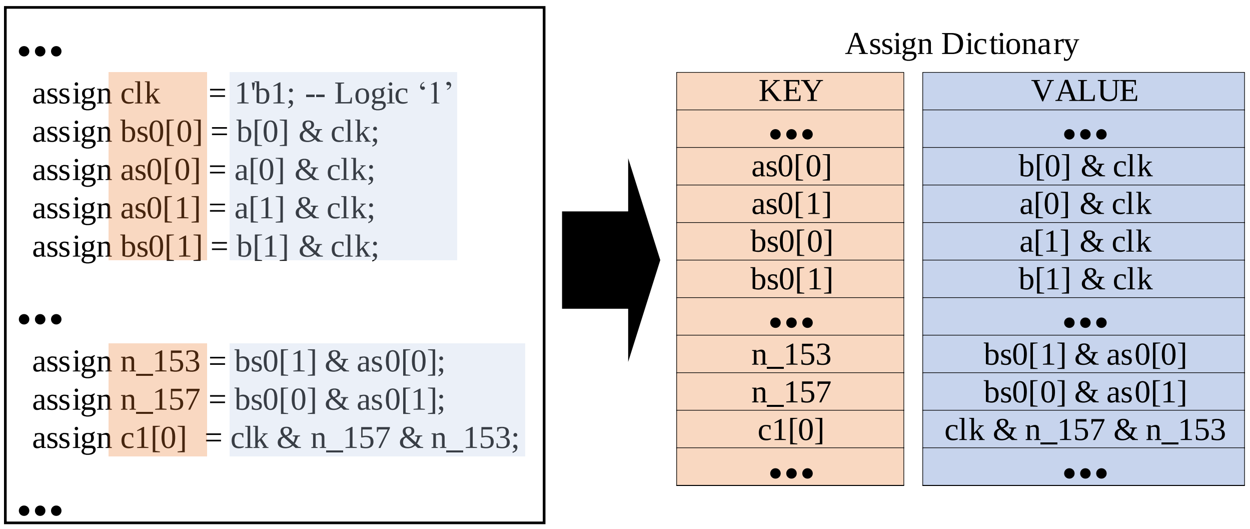
Regarding pipeline stage outputs appended to the port map, a port direction of INOUT is preferred to prevent the synthesis tool from simplifying crucial pipeline stage names out of the RTL. Once the port map modifications are complete for every VHDL entity in the design hierarchy, single-rail synthesis can begin.

3.3. Single-Rail Synthesis

The first round of synthesis, called single-rail synthesis, converts the pre-processed VHDL behavioral code into a list of combinational wire assignments. These wire assignments can be easily parsed by the MTNCLSyn script to group each wire assignment by pipeline stage. After every wire has been assigned to a pipeline stage, the stages are separated into Verilog RTL netlist files with the corresponding inputs, outputs, and Boolean logic.

The combinational RTL generated by GENUS consists of the flattened design hierarchy described in Verilog. To ensure there was no hierarchy created by GENUS during elaboration, the “ungroup -all” command from the GENUS command reference documentation [14] was passed to the tool. Every internal wire and output wire in the RTL are assigned to a Boolean logic function to be mapped to a function of the basic MTNCL threshold gates in Table 1. Because synchronous components are replaced with MUXs in the VHDL behavioral code, the generated RTL will contain MUX output signals assigned to Boolean logic functions containing a clock input. This clock input is not used in MTNCL handshaking, so it can be assigned to a constant value of Logic ‘1’. In the steps following single-rail synthesis, this constant Logic ‘1’ will be simplified away using Boolean logic to allow the flow of data to occur. For example, if an assigned output wire *z* is assigned to the Boolean logic *a & B & clock*, where “&” is a Boolean AND operation in Verilog, the resulting Boolean logic can be simplified to *a & B* when *clock* is substituted with a constant Logic ‘1’.

The MTNCLSyn script uses each wire to create a dictionary of assigned values. The wire being assigned is a key to the dictionary and the calculated Boolean function is the corresponding value. This is shown in Figure 10 below; Orange wire names are keys to the dictionary and blue Boolean functions are values for each key.



```
•••
  assign clk    = 1'b1; -- Logic '1'
  assign bs0[0] = b[0] & clk;
  assign as0[0] = a[0] & clk;
  assign as0[1] = a[1] & clk;
  assign bs0[1] = b[1] & clk;

•••
  assign n_153 = bs0[1] & as0[0];
  assign n_157 = bs0[0] & as0[1];
  assign c1[0]  = clk & n_157 & n_153;

•••
```

Assign Dictionary

KEY	VALUE
•••	•••
as0[0]	b[0] & clk
as0[1]	a[0] & clk
bs0[0]	a[1] & clk
bs0[1]	b[1] & clk
•••	•••
n_153	bs0[1] & as0[0]
n_157	bs0[0] & as0[1]
c1[0]	clk & n_157 & n_153
•••	•••

Figure 10: The generated RTL from single-rail synthesis placed into an assign dictionary

The RTL of Figure 10 shows the multi-bit pipeline stage inputs *as0* and *bs0* are assigned to the result of a Boolean AND function using *a* and *b*. Then, as0 and *bs0* are assigned to intermediary wires *n_153* and *n_*157. These intermediary values were created by GENUS and have arbitrary names. Finally, the intermediary wires can be assigned to the Boolean AND of a clock wire named *clk*. In this example, wires *as0* and bs0 will become primary inputs to the pipeline stage and the wire *c1[0]* will become a primary output to the pipeline stage. Any intermediate wires occurring within the pipeline stage, such as *n_153* and *n_157*, are also assigned to the stage group. Once the entirety of the generated RTL has been placed into the assign dictionary, propagation backwards from primary outputs can begin.

Beginning from every primary output bit of the design, backwards propagation iterates through each Boolean logic function in the assign dictionary. For example, given a primary output in the assign dictionary with a key value of *sout* and Boolean logic value of *a* | *b*, *a* and *b* are evaluated to determine if they belong to the same pipeline stage group responsible for the value of *sout*. By propagating through the assign dictionary for each path in the combinational logic, a pipeline stage output, primary input, or UNCONNECTED value is reached. Figure 11 provides an example design with multiple, colored outputs paths for z0, z1, and z2 originating from inputs A, B, and C. As propagation from output to input occurs, each path can overlap to form a pipeline stage group. Because output wires z0, z1, and z2 share at least one common input, the MTNCLSyn script interprets them to be a part of the same MTNCL pipeline stage.

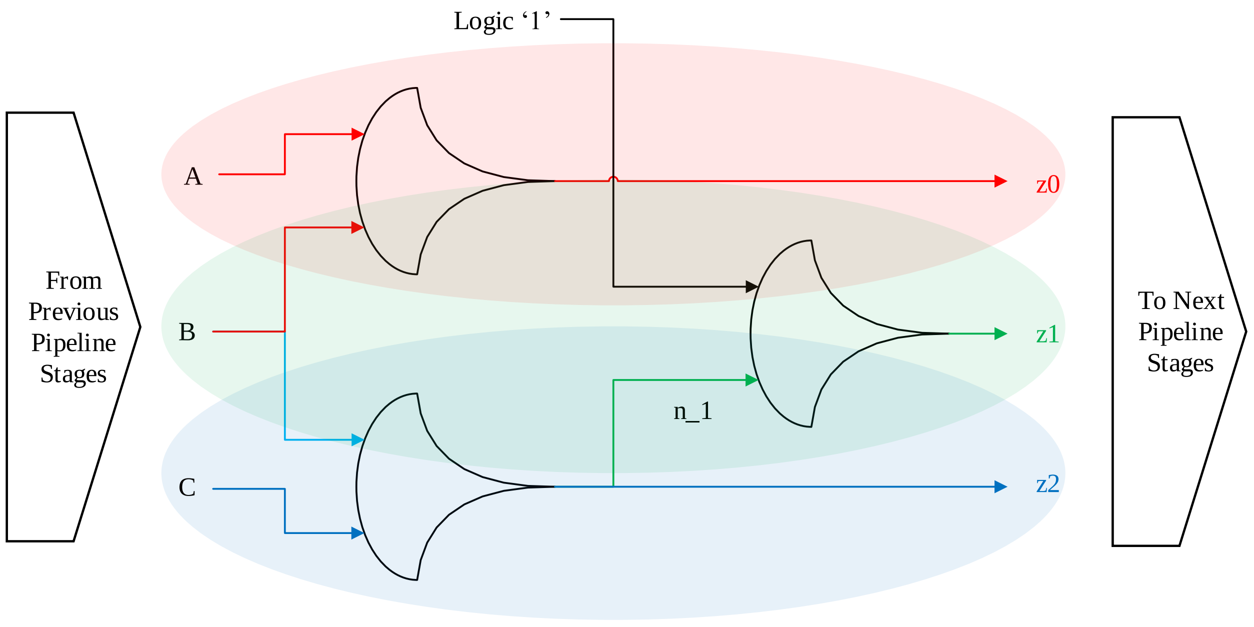


Figure 11: An example of wire paths as data flows in each pipeline stage

The names of pipeline stage outputs are known early in the MTNCL synthesis flow and can be matched to wire names found in each Boolean assignment. Primary input signals can be matched similarly. As shown by the Logic '1' wire input in Figure 11, dead-end wire constants can be found in the assign dictionary values. These wires do not have a key value associated with them in the assign dictionary and can be evaluated as UNCONNECTED. When a wire is evaluated as UNCONNECTED, it must be added to the top-level primary inputs and controlled manually to ensure MTNCL DATA and NULL wave-front propagation can continue. Otherwise, UNCONNECTED signals will output a constant DATA wavefront while the other signals in the pipeline stage are outputting a NULL wavefront, effectively stalling the pipeline stage. Figure 12 provides a flowchart of the steps taken by the MTNCLSyn script to evaluate each wire path from primary output to primary input.

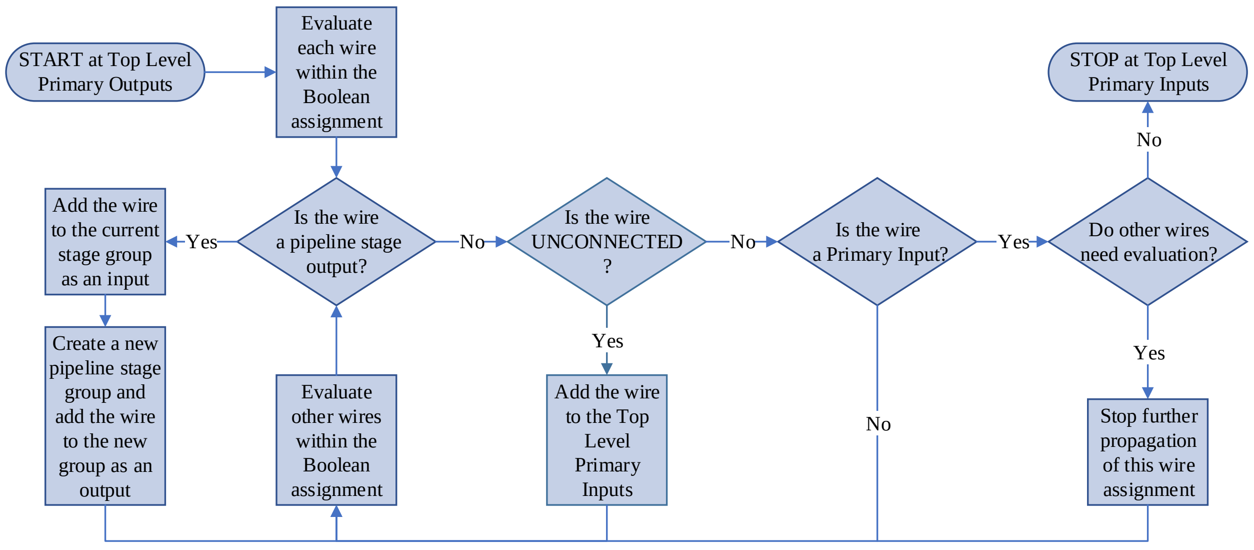
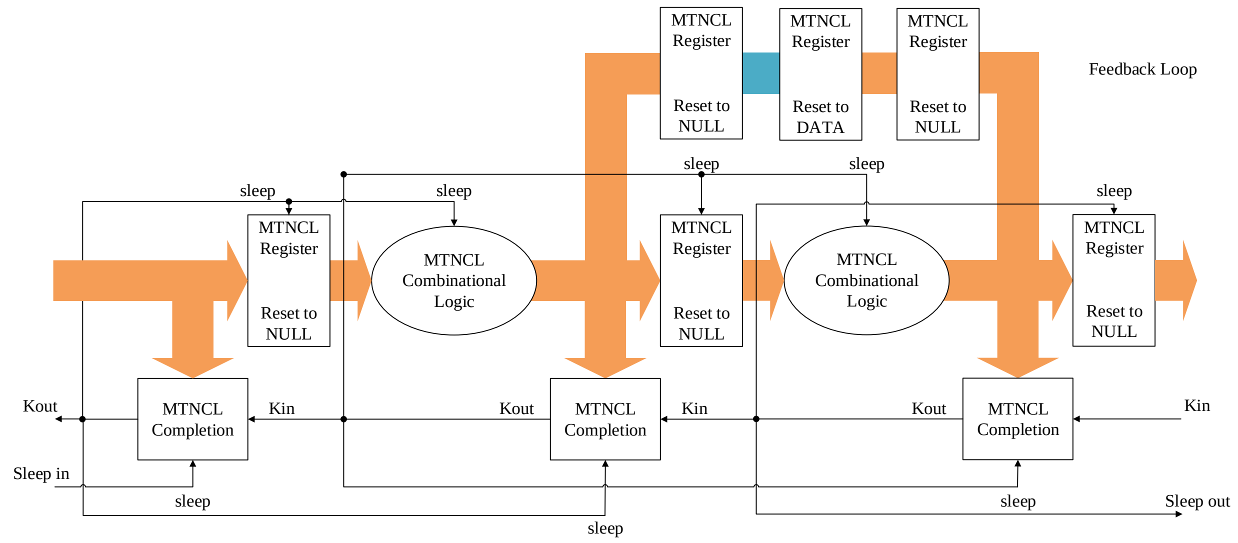

START at Top Level Primary Outputs
Evaluate each wire within the Boolean assignment
STOP at Top Level Primary Inputs
Add the wire to the current stage group as an input
Yes
Is the wire a pipeline stage output?
No
Is the wire UNCONNECTED ?
No
Is the wire a Primary Input?
Yes
Do other wires need evaluation?
No
Yes
Yes
No
Create a new pipeline stage group and add the wire to the new group as an output
Evaluate other wires within the Boolean assignment
Add the wire to the Top Level Primary Inputs
Stop further propagation of this wire assignment

Figure 12: The decision flow for evaluating wires in each path from primary output to primary input

It is possible to encounter pipeline stage outputs looping back into the inputs of other pipeline stages. To avoid the pipeline stall caused by these loops, the MTNCLSyn script needs to check if a feedback loop will be outputting a DATA or NULL wavefront to a pipeline stage expecting the opposite. This feedback looping predominantly occurs in pipeline stages with outputs that loop to inputs of the same pipeline stage. To fix this problem, additional MTNCL pipeline stages need to be placed between the feedback loop signal. In addition, the MTNCL register in the middle of the feedback loop is reset to a DATA wavefront instead of a NULL wavefront. Figure 13 shows the MTNCL feedback loop problem and the suggested fix to allow correct DATA and NULL wave-front propagation.



MTNCL Register Reset to NULL
MTNCL Register Reset to DATA
MTNCL Register Reset to NULL
Feedback Loop
sleep
sleep
sleep
sleep
sleep
MTNCL Register Reset to NULL
MTNCL Combinational Logic
MTNCL Register Reset to NULL
MTNCL Combinational Logic
MTNCL Register Reset to NULL
Kout
MTNCL Completion
Kin
Kout
MTNCL Completion
Kin
Kout
MTNCL Completion
Kin
Sleep in
sleep
sleep
sleep
Sleep out

Figure 13: MTNCL pipeline stage with an MTNCL register placed between the feedback loop

By following the MTNCLSyn decision flow, the generated RTL from single-rail synthesis is separated into multiple pipeline stages. Each pipeline stage is contained in a separate RTL file isolated from the other pipeline stages during dual-rail synthesis. In this way, the GENUS synthesis tool is constrained to individually evaluate the Boolean logic of each pipeline stage. This prevents the merging of Boolean logic between stage inputs and outputs and allows each stage to be converted into dual-rail logic in preparation for dual-rail synthesis and MTNCL library gate mapping.

3.4. Dual-Rail Synthesis

The second round of synthesis is executed on isolated pipeline stage from the generated RTL after conversion to dual-rail logic. As in the NCL-X synthesis tool, Dual-Rail expansion can be used to convert a single-rail logic design into a dual-rail logic design [12]. Dual-Rail expansion assigns every single-rail bit in the pipeline stage to Rail1 of the dual-rail logic design. Furthermore, the complement of Rail1 is computed and assigned to the Rail0 bit. Because the MTNCL handshaking protocol forces the NULL wavefront in each gate through the usage of a *sleep* signal input, the *sleep* signal is not included in the Boolean function of each fundamental MTNCL gate. Because, each pipeline stage is combinational, the MTNCLSyn script assigns output wires to a Boolean function using only inputs to the pipeline stage. Figure 14 provides an example of the assignment of *c1(0)* in terms of inputs from an 8-bit array multiplier after conversion to dual-rail logic.

```
●●●
 c1(0).RAIL0 <= as0(0).RAIL0 OR as0(1).RAIL0 OR bs0(0).RAIL0 OR bs0(1).RAIL0;
 c1(0).RAIL1 <= as0(0).RAIL1 AND as0(1).RAIL1 AND bs0(0).RAIL1 AND bs0(1).RAIL1;

●●●
```

Figure 14: The dual-rail assignment of output wire c1(0) in a pipeline stage

After Dual-Rail expansion, there will no longer be any intermediary wires in the pipeline stages, and the pipeline stage RTL can be mapped to fundamental MTNCL logic gates with the GENUS synthesis tool. However, the gate-level netlists created by GENUS in this step lack the necessary *sleep* signal ports. Therefore, the MTNCLSyn script will need to append an input *sleep* wire to the port map of every gate in the pipeline stage’s gate-level netlist. For pipeline stages with inputs and outputs originating from multiple branching pipelines, a Boolean logic AND tree is generated to combine the multiple *sleep* inputs for each stage. The MTNCL AND tree is comprised of TH22m, TH33m, and TH44m gates to combine vectors into a single-bit output. The AND tree can also be used to combine multiple *Kin* and MTNCL completion signals. The circuit design in Figure 15 details the structure of an AND tree combining seven *sleep* inputs into one.

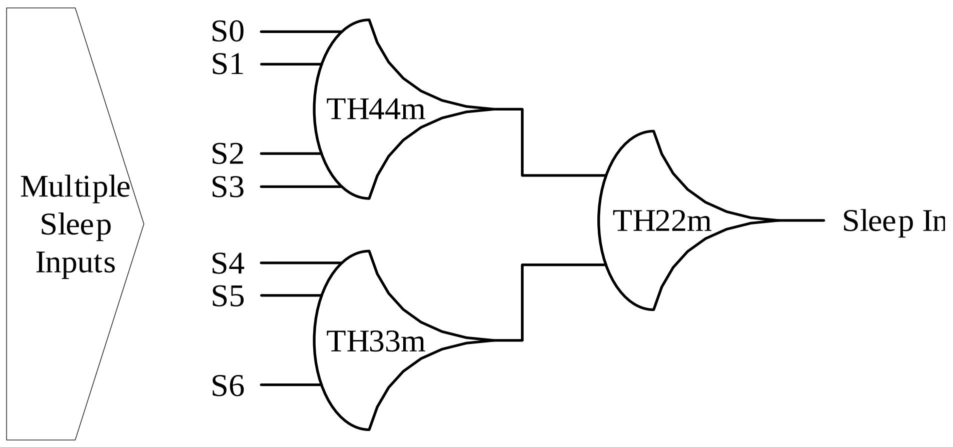


Figure 15: MTNCL AND tree with seven inputs

After synthesis is completed, MTNCL registers are placed at the inputs of individual pipeline stages to maintain the dual-rail values until the MTNCL combinational logic finishes computation. The MTNCL register design used in this thesis consists of a TH12nm threshold gate with two inputs and a threshold value of one. The functionality of a TH12nm threshold gate is identical to the functionality of a fundamental TH12m gate with the only difference being a reset input which can be used to set the output value to Logic ‘0’. The first input is a pipeline stage input wire while the second is a feedback loop of the output of the TH12nm gate. With this configuration, the TH12nm gate can hold the input value it receives at its output until it is forced to a Logic ‘0’ by the assertion of the *sleep* wire input. The outputs from the TH12nm gates used for MTNCL registers are also buffered by the MTNCLSyn script to eliminate a possible *sleep* race condition from occurring when a pipeline stage’s combinational logic input has a high capacitive load. Thus, the flow for each pipelines stage input is first to the input of an TH12nm threshold gate, then to a buffer gate, and finally to the input of the combinational logic cells in the pipeline stage. Figure 16 shows the MTNCL gate-level design of the MTNCL registers used in this thesis.

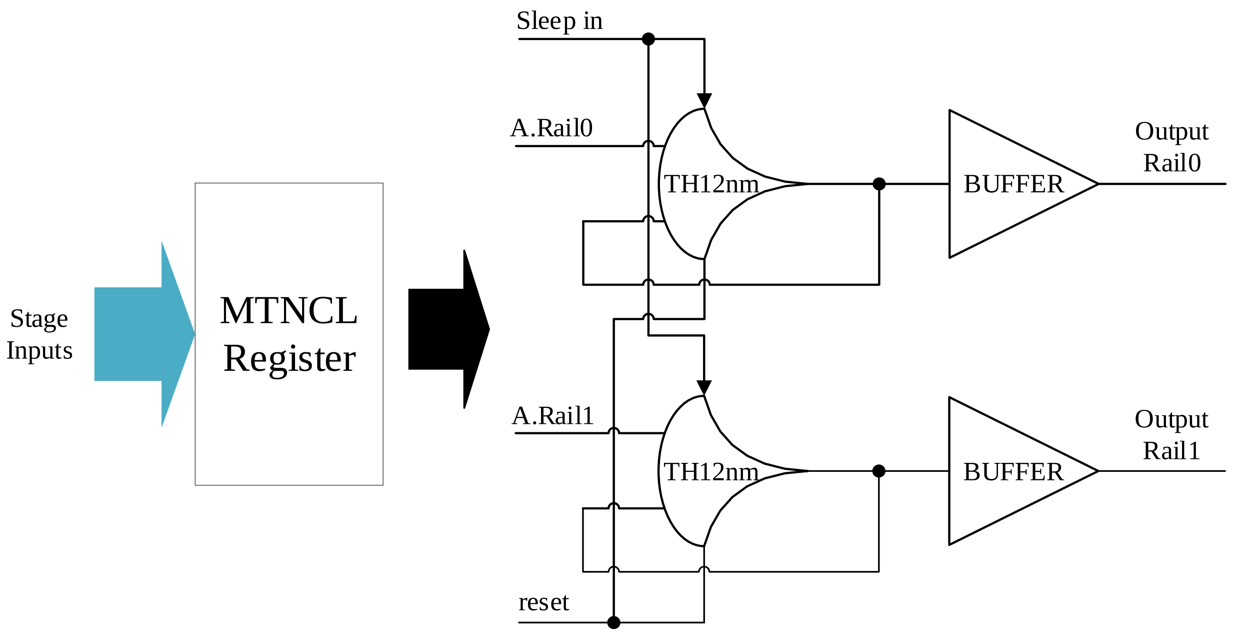


Figure 16: Gate-level MTNCL register with buffer

Uniquely named sleep domains are assigned to each pipeline stage to be combined after dual-rail synthesis. For each pipeline stage, the *sleep* input is the combined *sleep* output from all previous branching stages converging at the current stage. The *Kout* output is sent from the current stage to all previous branching stages as a *Kin* input. The handshaking of *sleep*, *Kout*, and *Kin* wires across each pipeline stage ensures that the previous stages know when the current stage expects a DATA or NULL wavefront.

MTNCL completion components are implemented to determine when a pipeline stage has finished its computation and is ready for the next DATA or NULL wavefront. To determine whether the current pipeline stage is ready for DATA or NULL, every output bit from the MTNCL combinational logic needs to evaluate to DATA or NULL. Until all output bits evaluate, the *Kout* wire sent to the previous stages does not change from Logic ‘0’ or Logic ‘1’. The MTNCL completion design used in this thesis utilizes the TH12m or TH24compm threshold gates with a Boolean AND tree and inverter gate to combine the outputs of *Kout* and *sleep out*. In this design, Rail0 and Rail1 of each dual-rail signal is checked by either a TH12m or TH24compm threshold gate, depending on the number of dual-rail signals used in the pipeline stage. The MTNCLSyn script generates the MTNCL completion design as efficiently as possible by pairing multiple dual-rail signals to the same TH24compm threshold gate when possible. For dual-rail signals unable to be paired, the TH12m threshold gate is used. Regardless of whether the wavefront is DATA0 or DATA1, one rail will be asserted to Logic ‘1’ for every dual-rail signal. In this way, the MTNCL TH12m and TH24compm threshold gates can be configured to assert a Logic ‘1’ on their outputs when all dual-rail inputs are either DATA0 or DATA1. As well as combining each TH12m and TH24compm output, the AND tree combines the *Kin* wire from the subsequent stage. The gate-level MTNCL completion design is presented in Figure 17.

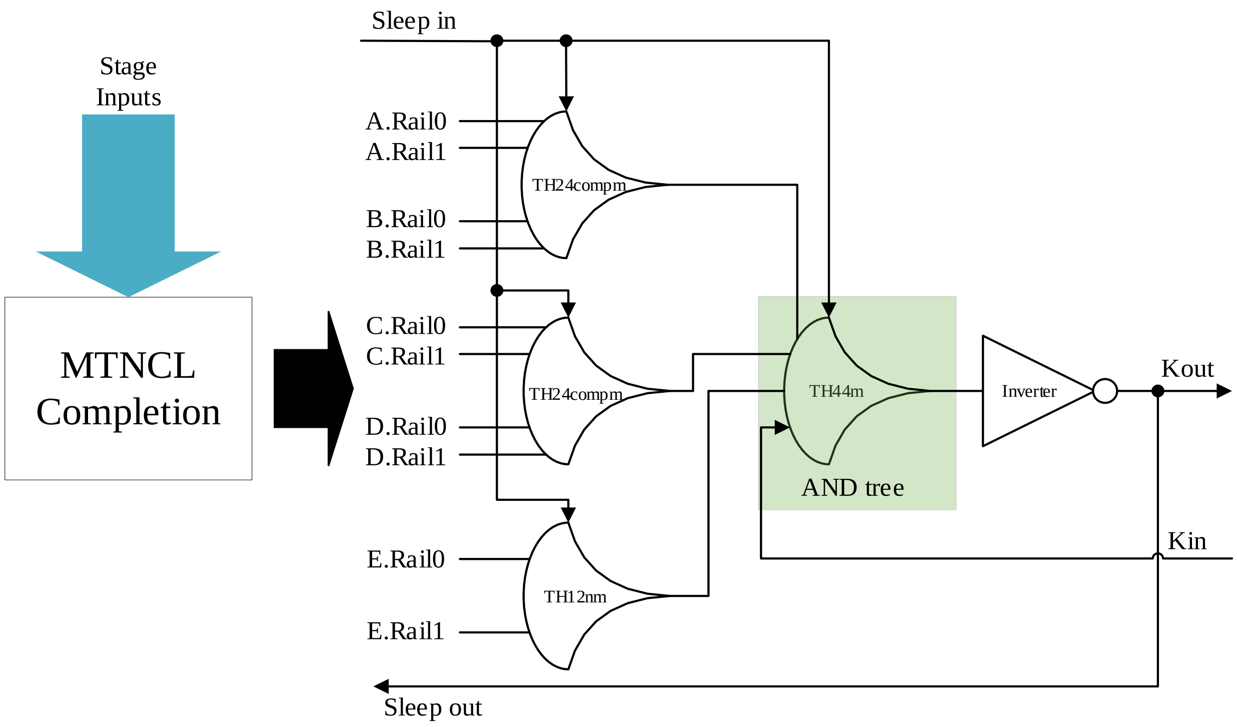


Figure 17: MTNCL Completion component with AND tree and inverter

According to Figure 17, two TH24compm threshold gates are generated to handle four of the dual-rail logic inputs. Because there is an odd number of dual-rail signals, an additional TH12m threshold gate is generated to check the completion of the *E.Rail0* and *E.Rail1* wires. Once the required MTNCL sleep, register, and completion components have been added to the gate-level netlist, the pipeline stages can be combined into a single gate-level netlist.

3.5. Combining of Pipeline Stages

When combining several gate-level design files together, various wires and MTNCL gate instantiation names will need to be uniquely named to prevent similarly named wires created by GENUS from overlapping with other pipeline stages. For example, a generated wire created by GENUS is renamed from “n_50” to “n_50_stage_0” to uniquely assign it to only the MTNCL logic of stage 0. The same naming technique is applied to MTNCL threshold gates mapped by

GENUS. Prior to combining, every pipeline stage will have a set of previous stages and next stages used by the MTNCLSyn script to establish how to connect the MTNCL sleep domain wires. Figure 18 illustrates an example arrangement of stages in a pipeline to be combined based on its previous and next stages.

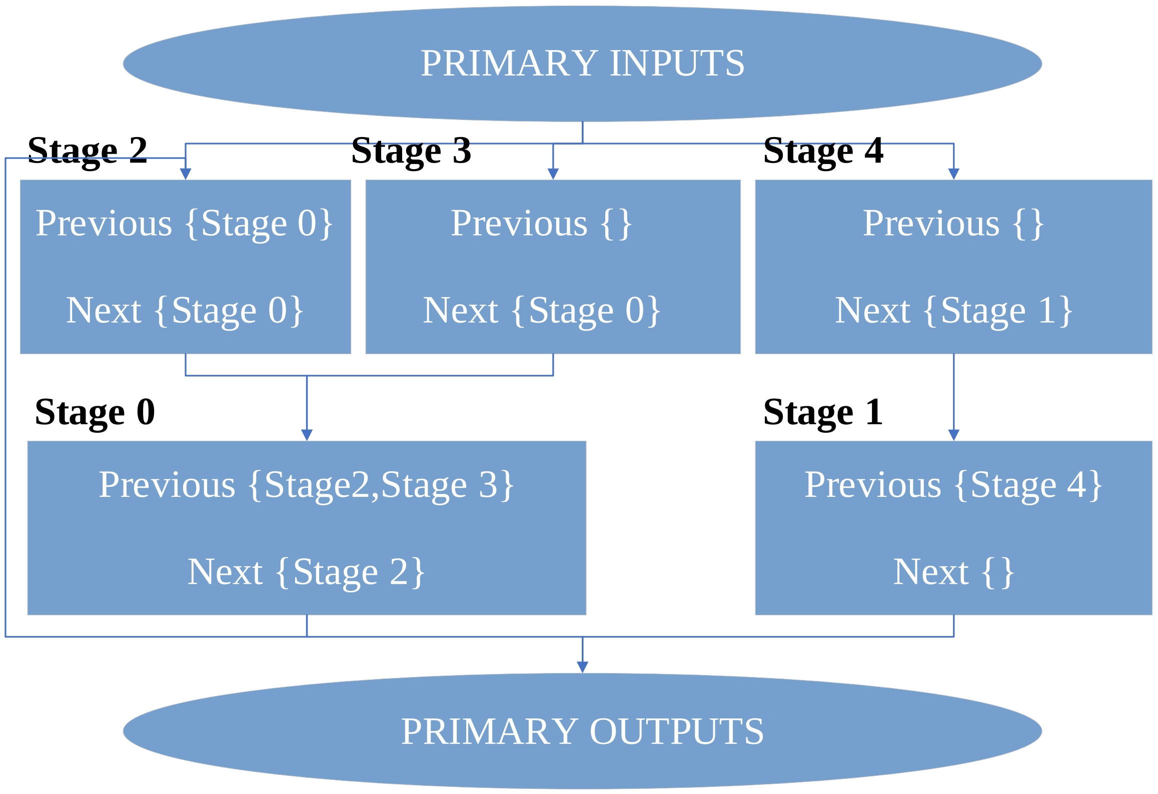


Figure 18: MTNCL pipeline stage relationships

The port maps of every MTNCL pipeline stage will also need to be checked to determine if their wires belong in the primary port map for the design. Additionally, pipeline stages with primary inputs must have dedicated *Kout* and *sleep* in wires in the primary port map. Pipeline stages with a primary output must have a dedicated *Kin* and *sleep out*. The MTNCLSyn script can handle such cases due to the unique name assigned to each stage prior to unification. However, pipeline stages in the middle of a circuit containing primary inputs or outputs will increase the complexity of a design at the top-level because additional *sleep in*, *sleep out*, *Kin*, and *Kout* wires will need to be properly controlled by top-level modules.

4. Results

Having completed the required pre-processing, single-rail synthesis, dual-rail synthesis, and combination of pipeline stages, the MTNCL gate-level netlist generated by the MTNCL synthesis flow can be functionally simulated with ModelSim. The MTNCL designs created by the MTNCL synthesis flow include a 4-bit ripple carry adder (RCA), pipelined oscillator, pipelined 4-bit arithmetic logic unit (ALU), and pipelined 8-bit array multiplier. In addition to testing the functionality, the gate utilization of a structural pipelined 8-bit array multiplier is compared against a generated gate-level netlist. Results of testing the MTNCL synthesis flow on a sample finite state machine from the ISCAS’99/ITC’99 benchmarks library is also provided [1].

4.1. 4-bit RCA

A 4-bit RCA was used to test the MTNCL synthesis flow against a design which has no pipeline stages and is purely combinational. For designs without any pipeline stages, the amount steps to be performed by the MTNCL synthesis flow is greatly reduced as the handshaking logic is no longer needed. Figure 19 shows the waveform results for the generated gate-level netlist from the MTNCL synthesis flow. The inputs and outputs have been converted to unsigned decimal in the waveform view.

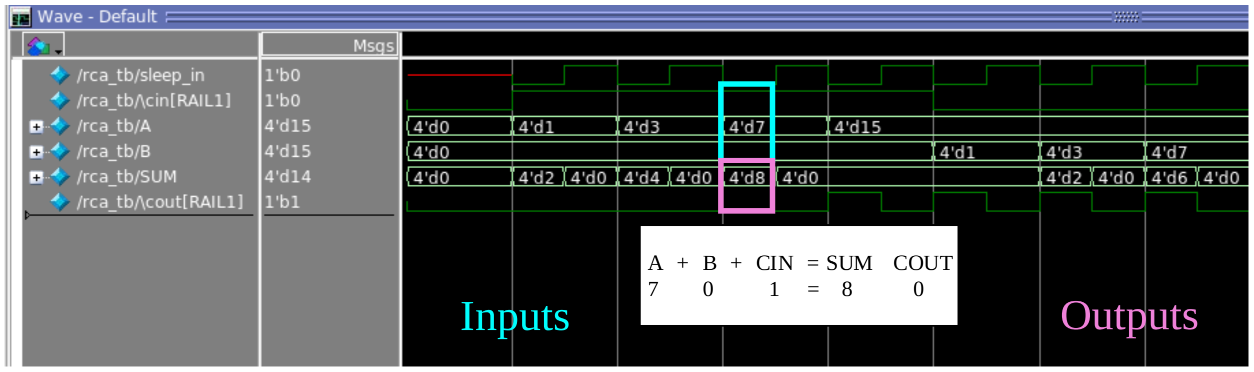


Figure 19: 4-bit RCA waveform

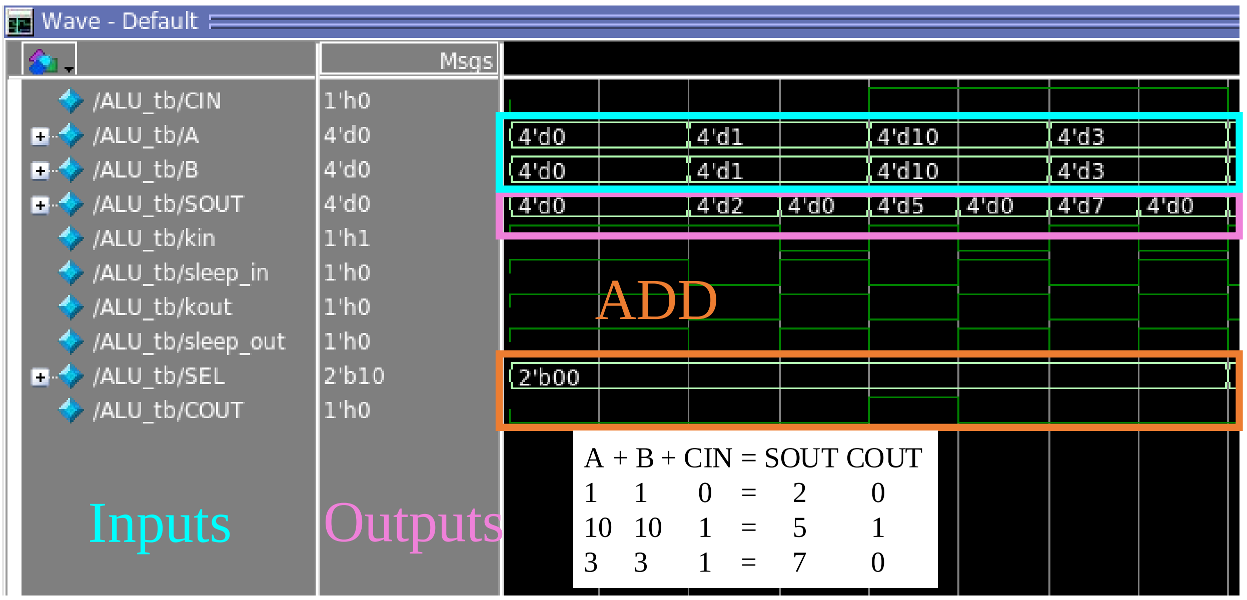
For testing, the 4-bit RCA was given a range of values to test the functionality of the *SUM* and *COUT* or carry out. When the *SUM* is less than 15 in unsigned decimal, the *COUT* value will remain Logic ‘0’. Since there are no pipeline stages, the entirety of the 4-bit RCA is described using MTNCL gates. NCL gates and Inverters are not needed since there is no MTNCL completion. Buffers are also not needed in this simple example since there are no MTNCL registers whose outputs need to be buffered. Table 3 provides the number of MTNCL gates used by the generated gate-level netlist.

Table 3: Gate utilization of the generated 4-bit RCA

	MTNCL gates	NCL gates	Inverters	Buffers	Total gates
Generated 4-bit RCA	29	0	0	0	29

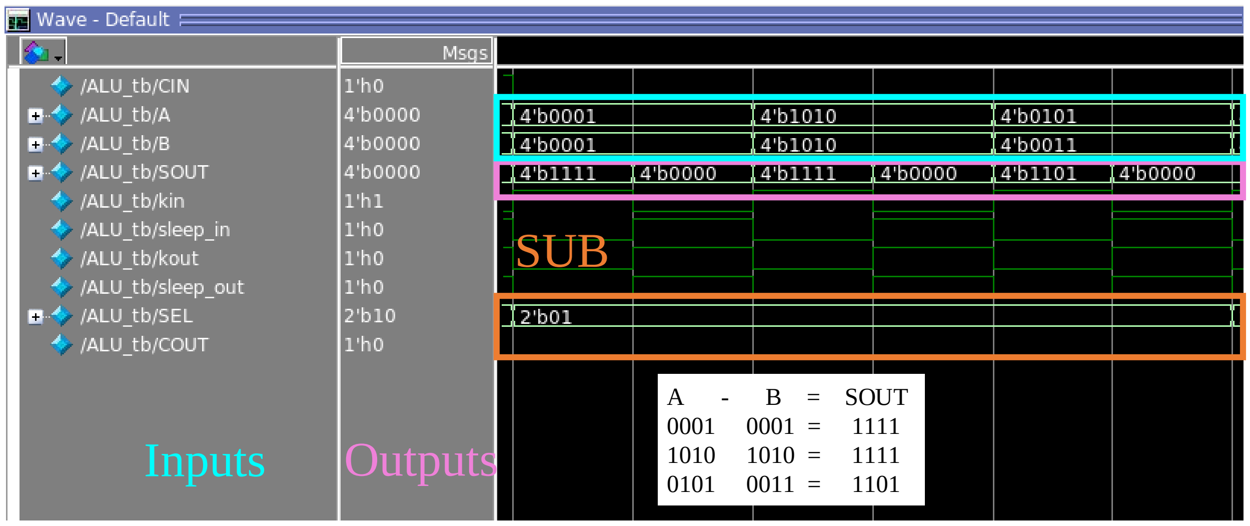
4.2. Pipelined 4-bit ALU

For designs which contain only one pipeline stage, a 4-bit ALU was synthesized. The 4-bit ALU contains a single pipeline stage to hold the input values for one DATA wavefront propagation. A 2-bit select signal *SEL* is also used to control what operation is to be computed for the input signals *a*, *b,* and *CIN*. For values of *SEL* “00”, “01”, “10”, and “11” the operation to be computed is addition, subtraction, bit-wise AND, or bit-wise OR respectively. Figure 20 provides the waveform for the 4-bit ALU performing an ADD, Figure 21 shows a waveform of the SUB, and Figure 22 shows the ADD and OR operations. The ADD waveform uses unsigned decimal, the SUB waveform is binary, and the ADD SUB operations are given in hexadecimal.



A	+ B	+ CIN	=	SOUT	COUT
1	1	0	=	2	0
10	10	1	=	5	1
3	3	1	=	7	0

Figure 20: 4-bit ALU addition waveform

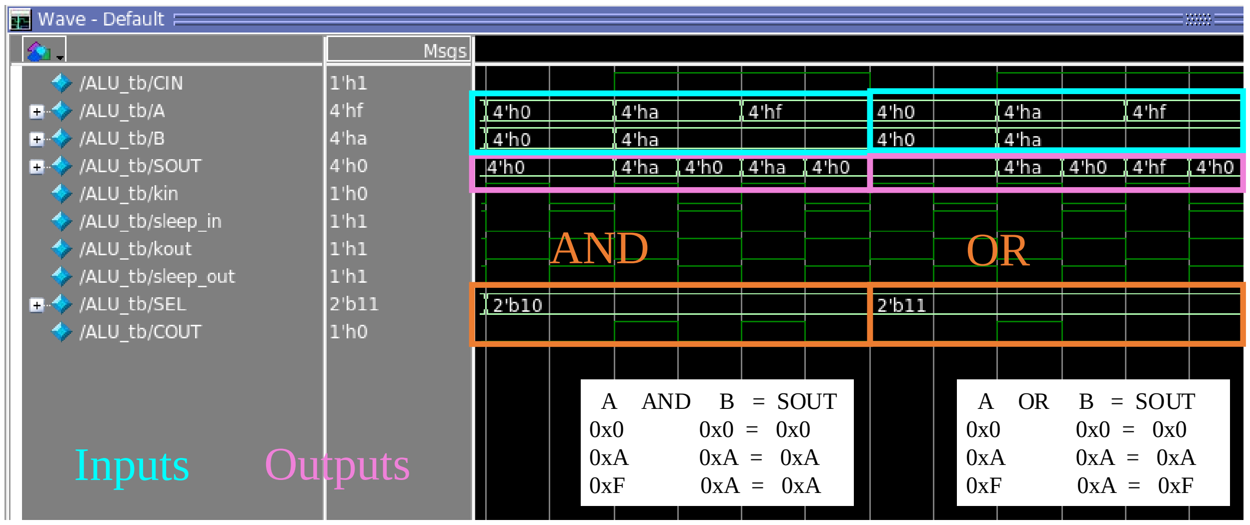


Signal	Msgs
/ALU_tb/CIN	1'h0
/ALU_tb/A	4'b0000
/ALU_tb/B	4'b0000
/ALU_tb/SOUT	4'b0000
/ALU_tb/kin	1'h1
/ALU_tb/sleep_in	1'h0
/ALU_tb/kout	1'h0
/ALU_tb/sleep_out	1'h0
/ALU_tb/SEL	2'b10
/ALU_tb/COUT	1'h0

```
4'b0001          4'b1010          4'b0101
4'b0001          4'b1010          4'b0011
4'b1111  4'b0000 4'b1111  4'b0000 4'b1101  4'b0000
2'b01
```

A	-	B	=	SOUT
0001		0001	=	1111
1010		1010	=	1111
0101		0011	=	1101

Figure 21: 4-bit ALU subtraction waveform



Signal	Msgs
/ALU_tb/CIN	1'h1
/ALU_tb/A	4'hf
/ALU_tb/B	4'ha
/ALU_tb/SOUT	4'h0
/ALU_tb/kin	1'h0
/ALU_tb/sleep_in	1'h1
/ALU_tb/kout	1'h1
/ALU_tb/sleep_out	1'h1
/ALU_tb/SEL	2'b11
/ALU_tb/COUT	1'h0

```
4'h0      4'ha      4'hf      4'h0      4'ha      4'hf
4'h0      4'ha                4'h0      4'ha
4'h0      4'ha 4'h0 4'ha 4'h0           4'ha 4'h0 4'hf 4'h0
2'b10                         2'b11
```

A	AND	B	=	SOUT
0x0		0x0	=	0x0
0xA		0xA	=	0xA
0xF		0xA	=	0xA

A	OR	B	=	SOUT
0x0		0x0	=	0x0
0xA		0xA	=	0xA
0xF		0xA	=	0xF

Figure 22: 4-bit ALU AND OR waveform

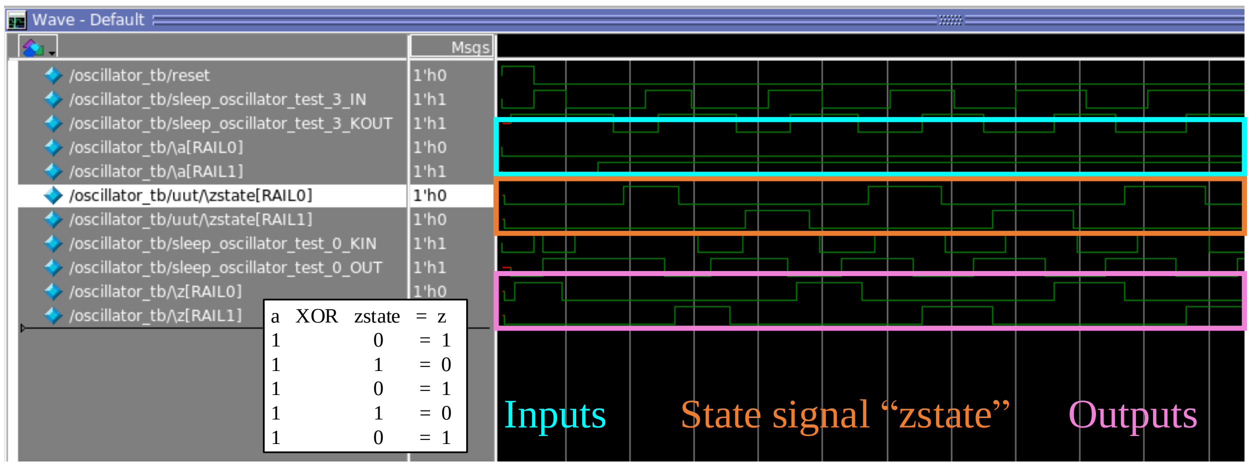
Unlike the RCA, the ALU produces its own *sleep* signal and has MTNCL registers and MTNCL completion components. These components all attribute to the need for NCL gates, inverters, and buffers. The MTNCL gate composition of the 4-bit ALU is provided by Table 4.

Table 4: Gate utilization of the generated 4-bit ALU

	MTNCL gates	NCL gates	Inverters	Buffers	Total gates
Generated 4-bit ALU	91	2	1	22	116

4.3. Pipelined Oscillator

A pipelined oscillator was used to test the functionality of a synthesized circuit containing a feedback loop from an output of one of its stages. In this design, three MTNCL registers were placed in series along the wires that feeds back into the previous stage. Figure 13 from Section 3.3 provides a circuit diagram of this implementation. Figure 23 provides the waveform given a constant input value of Logic ‘1’ for the dual-rail input *a*.



```
/oscillator_tb/reset                          1'h0
/oscillator_tb/sleep_oscillator_test_3_IN     1'h1
/oscillator_tb/sleep_oscillator_test_3_KOUT   1'h1
/oscillator_tb/\a[RAIL0]                      1'h0
/oscillator_tb/\a[RAIL1]                      1'h1
/oscillator_tb/uut/\zstate[RAIL0]             1'h0
/oscillator_tb/uut/\zstate[RAIL1]             1'h0
/oscillator_tb/sleep_oscillator_test_0_KIN    1'h1
/oscillator_tb/sleep_oscillator_test_0_OUT    1'h1
/oscillator_tb/\z[RAIL0]                      1'h0
/oscillator_tb/\z[RAIL1]
```

a	XOR zstate	= z
1	0	= 1
1	1	= 0
1	0	= 1
1	1	= 0
1	0	= 1

Figure 23: 1-bit oscillator waveform

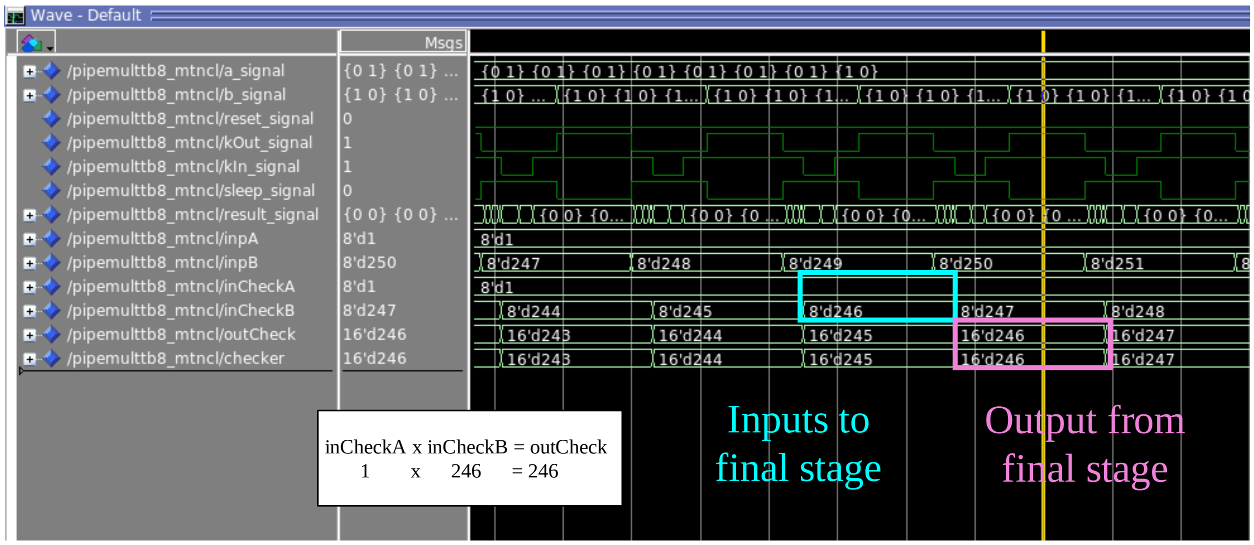
The oscillator implements five total pipeline stages. Two of the stages are used for holding the input and output values, while the other three stages are used in the feedback loop. This increase in pipeline stages leads to an increase in MTNCL registers and MTNCL completion components. The amount of NCL gates, inverters, and buffers used in the handshaking can be seen by the overall gate utilization shown in Table 5.

Table 5: Gate utilization of the generated oscillator

	MTNCL gates	NCL gates	Inverters	Buffers	Total gates
Generated Oscillator	21	7	5	12	45

4.4. 8-bit Pipelined Array Multiplier

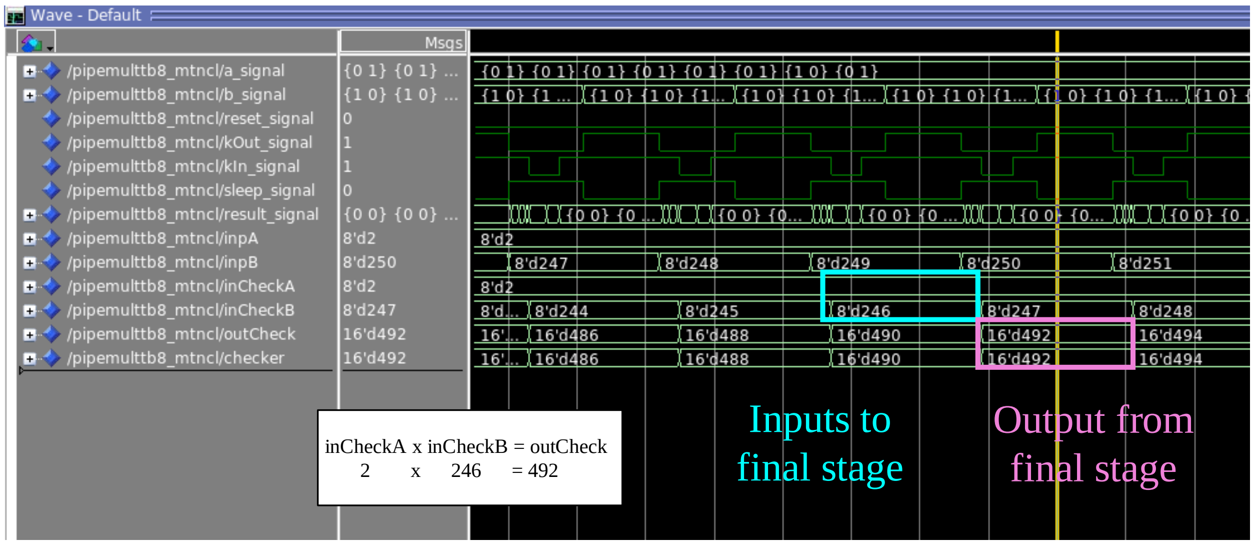
The same VHDL testbench was used to evaluate the MTNCL synthesis generated and MTNCL structural 8-bit multipliers. Both multipliers implement an eight-stage pipeline with multiple wires combined at the boundaries of each stage. This simulation was used to validate the MTNCL synthesis flow for large feed-forward pipelined designs. The results of both the structural and generated behavioral gate-level netlists are functionally identical and follow the waveforms shown by Figure 24 and Figure 25. Figure 24 evaluates the multiplier for a factor of 1 multiplied by 246 while Figure 25 evaluates for a factor of 2 multiplied by 246.



	Msgs
/pipemulttb8_mtncl/a_signal	{0 1} {0 1} ...
/pipemulttb8_mtncl/b_signal	{1 0} {1 0} ...
/pipemulttb8_mtncl/reset_signal	0
/pipemulttb8_mtncl/kOut_signal	1
/pipemulttb8_mtncl/kIn_signal	1
/pipemulttb8_mtncl/sleep_signal	0
/pipemulttb8_mtncl/result_signal	{0 0} {0 0} ...
/pipemulttb8_mtncl/inpA	8'd1
/pipemulttb8_mtncl/inpB	8'd250
/pipemulttb8_mtncl/inCheckA	8'd1
/pipemulttb8_mtncl/inCheckB	8'd247
/pipemulttb8_mtncl/outCheck	16'd246
/pipemulttb8_mtncl/checker	16'd246

inCheckA x inCheckB = outCheck
1 x 246 = 246

Figure 24: Generated 8-bit pipelined array multiplier simulated in ModelSim



	Msgs
/pipemulttb8_mtncl/a_signal	{0 1} {0 1} ...
/pipemulttb8_mtncl/b_signal	{1 0} {1 0} ...
/pipemulttb8_mtncl/reset_signal	0
/pipemulttb8_mtncl/kOut_signal	1
/pipemulttb8_mtncl/kIn_signal	1
/pipemulttb8_mtncl/sleep_signal	0
/pipemulttb8_mtncl/result_signal	{0 0} {0 0} ...
/pipemulttb8_mtncl/inpA	8'd2
/pipemulttb8_mtncl/inpB	8'd250
/pipemulttb8_mtncl/inCheckA	8'd2
/pipemulttb8_mtncl/inCheckB	8'd247
/pipemulttb8_mtncl/outCheck	16'd492
/pipemulttb8_mtncl/checker	16'd492

inCheckA x inCheckB = outCheck
2 x 246 = 492

Figure 25: Generated 8-bit pipelined array multiplier simulated in ModelSim

In terms of MTNCL gate utilization, the generated and structural multipliers vary greatly. Due to the Boolean logic optimizations that are performed by the GENUS synthesis tool, the MTNCL synthesis flow was able to create a functionally equivalent design to the structural

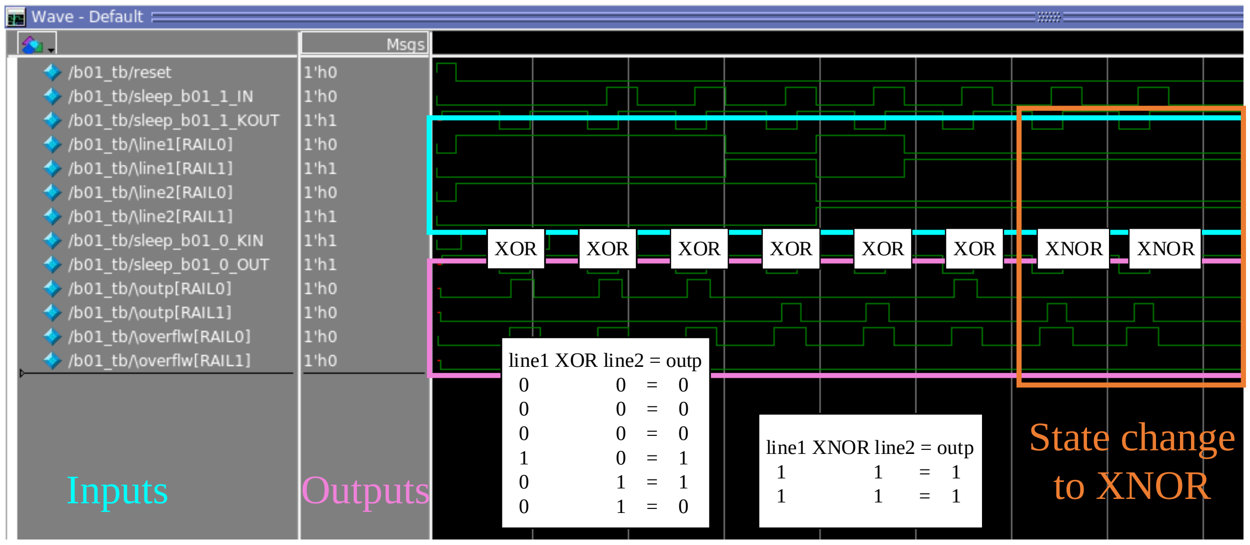
VHDL multiplier while using less MTNCL library gates. Table 6 compares the gate utilization of the generated and structural 8-bit multipliers.

Table 6: Gate utilization of structural vs generated 8-bit multiplier

	MTNCL gates	NCL gates	Inverters	Buffers	Total gates
Generated 8-bit Multiplier	1,232	8	8	424	1,672
Structural 8-bit Multiplier	1,825	16	16	1,122	2,979

4.5.ISCAS’99/ITC’99 Benchmarks

The ISCAS’99/ITC’99 benchmarks include various designs for benchmarking finite state machines of different sizes and complexity. One such finite state machine is the b01 design which performs a Boolean XOR operation on two inputs for serial comparison. The MTNCL synthesis flow was used to convert this state machine from its original behavioral description into an MTNCL gate-level netlist This operation is performed until the sixth MTNCL handshake where it changes to XNOR as seen in Figure 26.



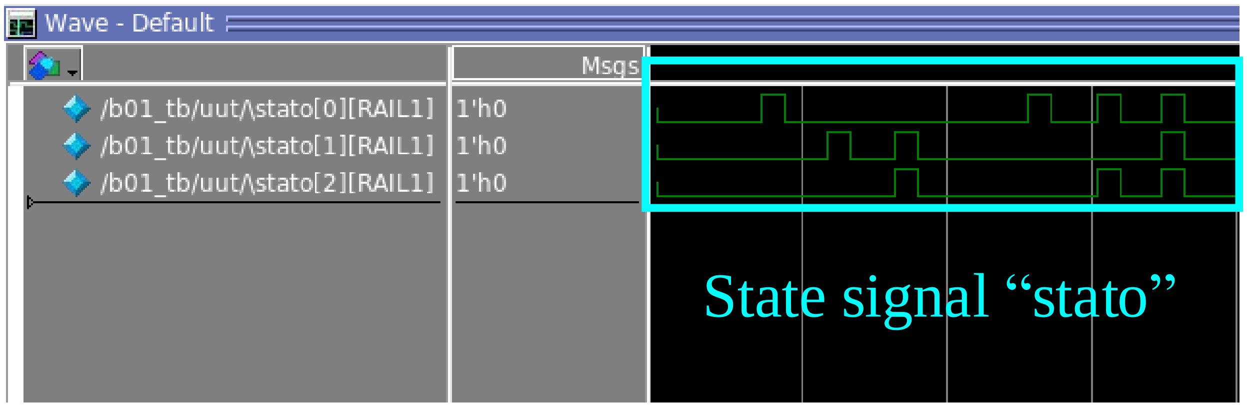
	Msgs
/b01_tb/reset	1'h0
/b01_tb/sleep_b01_1_IN	1'h0
/b01_tb/sleep_b01_1_KOUT	1'h1
/b01_tb/\line1[RAIL0]	1'h0
/b01_tb/\line1[RAIL1]	1'h1
/b01_tb/\line2[RAIL0]	1'h0
/b01_tb/\line2[RAIL1]	1'h1
/b01_tb/sleep_b01_0_KIN	1'h1
/b01_tb/sleep_b01_0_OUT	1'h1
/b01_tb/\outp[RAIL0]	1'h0
/b01_tb/\outp[RAIL1]	1'h0
/b01_tb/\overflw[RAIL0]	1'h0
/b01_tb/\overflw[RAIL1]	1'h0

line1	XOR line2	=	outp
0	0	=	0
0	0	=	0
0	0	=	0
1	0	=	1
0	1	=	1
0	1	=	0

line1	XNOR line2	=	outp
1	1	=	1
1	1	=	1

Figure 26: ISCAS'99/ITC'99 b01 waveform in MTNCL

The state signal changes with each handshake until a maximum value of "111" is reached. Internal state signal *stato* determines whether an XOR or XNOR operation is to be calculated for the two inputs *line1* and *line2*. The Rail1 values for the changing states are shown by the waveform in Figure 27. The overall gate utilization for the MTNCL generated b01 finite state machine is displayed in Table 7.



	Msgs
/b01_tb/uut/\stato[0][RAIL1]	1'h0
/b01_tb/uut/\stato[1][RAIL1]	1'h0
/b01_tb/uut/\stato[2][RAIL1]	1'h0

Figure 27: Waveform for b01 *stato* signal

Table 7: Gate utilization of the generated b01 FSM

	MTNCL gates	NCL gates	Inverters	Buffers	Total gates
Generated b01 FSM	77	10	4	24	115

Conclusion

The MTNCL synthesis flow was successfully implemented and tested on various designs which were originally described in VHDL behavioral code. By applying logic synthesis to the behavioral code, the GENUS synthesis tool was able to improve upon existing asynchronous designs which would traditionally need to be described in a structural VHDL model. Hardware designers can implement their designs without having to manually place the handshaking wires and control logic components that are needed for MTNCL to operate in a sequential circuit. This improves the accessibility of MTNCL to hardware designers with a limited understanding of how the architecture works.

By modifying the syntax of behavioral VHDL code and synthesized RTL, a synchronous design can be described in an asynchronous way. Industry standard tools for synthesis were used to implement as many steps in the MTNCL synthesis flow as possible to achieve an efficient gate-level design. The target MTNCL library used in this thesis is quasi delay-insensitive and requires no timing analysis to be performed due to the lack of a clock tree [3]. Data is no longer gated on the period of an oscillating clock signal and can instead flow from input to output depending on the delay of the combinational logic itself. This leads to an overall simpler design that can operate with less constraint.

Continuing work will be done on this MTNCL flow to further automate the design process and improve reliability. Currently as it is implemented, the MTNCLSyn script can completely automate the flow of pipelines in designs that are feed-forward or have no feedback signals in its pipeline stages.

References

[1] ITC and ISCAS, "ITC'99 benchmarks developed in the CAD Group at Politecnico di Torino (I99T) - squillero/itc99-poli," 1999. [Online]. Available: https://github.com/squillero/itc99-poli.

[2] "IEEE Standard for VHDL Register Transfer Level Synthesis," *IEEE Std 1076.6,* pp. 1-80, 10 March 2000.

[3] L. Zhou, R. Parameswaran, F. A. Parsan, S. C. Smith and J. Di, "MTNCL: An Ultra-Low Power Asynchronous Circuit Design Methodology," *Journal of Low Power Electronics and Applications,* vol. 5, no. 2, pp. 81-100, 2015.

[4] K. M. Fant and S. A. Brandt, "NULL Convention Logic: A Complete and Consistent Logic fair Asynchronous Digital Circuit Synthesis," in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96*, Chicago, IL, USA, , 1996.

[5] S. Smith and J. Di, "Designing Asynchronous Circuits using NULL Convention Logic (NCL)," in *Designing Asynchronous Circuits using NULL Convention Logic (NCL),* Morgan & Claypool, 2009.

[6] Cadence, "Genus Synthesis Solution," 2019. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html. [Accessed 9 April 2019].

[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, no. 9, pp. 1031-1044, September 1996.

[8] J. Cortadella, A. Kondratyev, L. Lavagno and C. P. Sotiriou, "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 10, pp. 1904-1921, October 2006.

[9] P. A. Beerel, G. D. Dimou and A. M. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design & Test of Computers,* vol. 28, no. 5, pp. 36-51, Sept.-Oct. 2011.

[10] A. Smirnov, A. Taubin, S. Ming and M. Karpovsky, "An automated fine-grain pipelining using domino style asynchronous library," in *Fifth International Conference on Application of Concurrency to System Design (ACSD'05),* Saint Malo, France, 2005.

[11] R. B. Reese, S. C. Smith and M. A. Thornton, "Uncle - An RTL Approach to Asynchronous Design," in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, Lyngby, Denmark, 2012.

[12] A. Kondratyev and K. Lwin, "Design of asynchronous circuits using synchronous CAD tools," *IEEE Design & Test of Computers,* vol. 19, no. 4, pp. 107-117, 7 August 2002.

[13] Synopsys, "Synopsys SiliconSmart," 2019. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/siliconsmart.html. [Accessed 11 April 2019].

[14] Cadence Design Systems, Inc, *GENUS Command Reference,* 2016.

[15] S. Davidson, "ITC'99 Benchmark Documentation," [Online]. Available: https://www.cerc.utexas.edu/itc99-benchmarks/bendoc1.html.