DOCUMENTATION OF ASIC SYNTHESIS WITH ADVANCED OPTIMIZATION TECHNIQUES

Date: 6-Feb-25

Prepared By: Masum Saimon

Table of Contents

ASIC Synthesis

Introduction to Synthesis:

Definition: Synthesis in VLSI design is the process of converting high-level design descriptions into gate-level representations. In other words, Synthesis is the process of turning a high-level description of a circuit (like a blueprint) into a detailed design that can be manufactured. Think of it as turning a rough sketch into a detailed construction plan.

- The output of synthesis is called a netlist. A netlist is a textual representation of all the nets and cells in the designs and their connections.
- It's crucial for transforming abstract designs into physical implementations that can be manufactured.

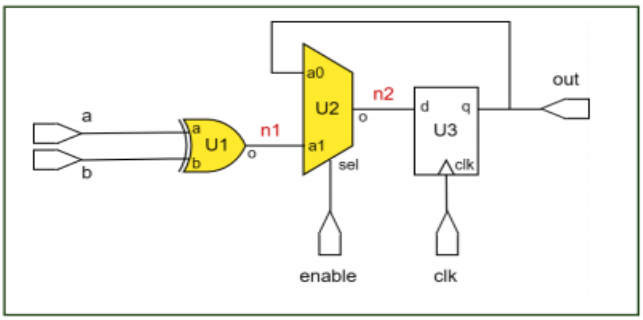
```
input clk, enable
input a,b;
output reg out;
always@(posedge clk)
  if(enable)
    out <= a^b;
  else
    out <= out;
```

Behavioral Verilog Code


```
input clk, enable;
input a, b;
output out;
wire n1, n2;

XOR U1 (.a(a), .b(b), .o(n1));
MUX U2 (.a0(out), .a1(n1), .sel(enable), .o(n2));
FFP U3 (.clk(clk), .d(n2), .q(out));
```

Synthesized Logic Netlist



Synthesized Logic Schematic

Why Synthesis is Needed:

Synthesis is a crucial step in the VLSI design process for several reasons:

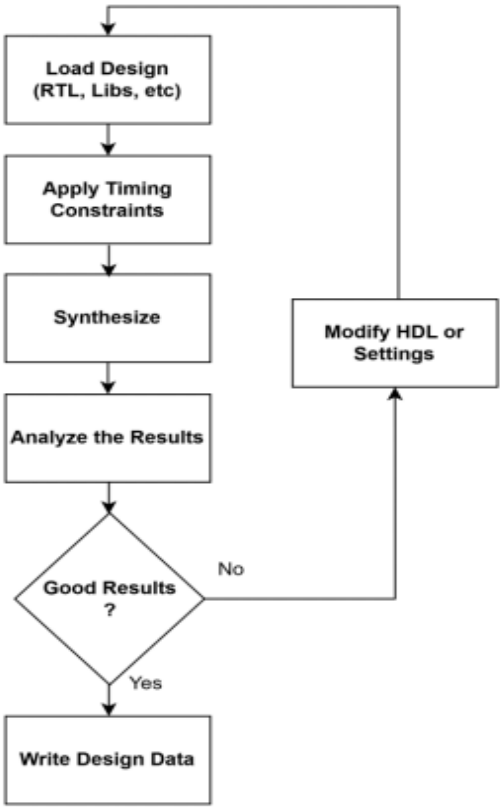
Translation of High-Level Design: It converts high-level design descriptions (like RTL code) into a detailed gate-level representation that can be manufactured.
Optimization: Synthesis optimizes the design for area, speed, and power, ensuring the final product is efficient and meets performance requirements.
Automation: It automates the process of creating a gate-level netlist, reducing the time and effort required compared to manual design.
Consistency: Ensures that the design follows to the specified constraints and standards, reducing the risk of errors.

Consequences of Skipping Synthesis:

If the synthesis stage were skipped in processing a design, several issues could arise:

Manual Errors: Without synthesis, the design would need to be manually converted to a gate-level representation, increasing the likelihood of human errors.
Inefficiency: The design might not be optimized for area, speed, or power, leading to a less efficient and potentially more expensive product.
Inconsistency: The design might not meet the required constraints and standards, resulting in functional or performance issues.
Increased Time and Cost: Manual design processes are time-consuming and costly, delaying the product development cycle.

ASIC Synthesis Processes:



Basic Steps from RTL to Gate-Level:

RTL Design: Writing the design in a hardware description language like Verilog or VHDL.

Elaboration: Converting the RTL code into an intermediate representation.

Optimization: Improving the design for area, speed, and power.

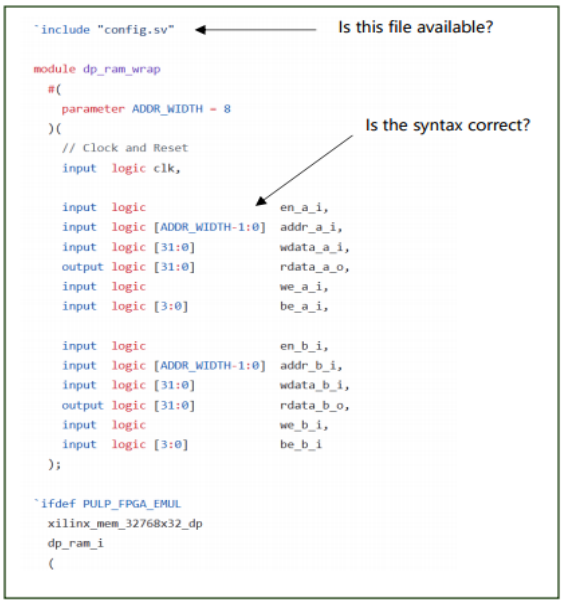
Mapping: Converting the optimized design into a gate-level netlist using a technology library.

Processing Steps:

Analyze:

The analyze step will check that:

- HDL codes contain no syntax issues.
- All modules/include files/functions/etc referenced inside the codes are available and nothing is missing.



Elaborate:

The second step is called elaborate and sometimes called translate. It's the process of converting the HDL codes into actual logic gates.

The elaboration step outputs are:

- A technology-independent (generic) netlist without any optimizations
  done. The cells referenced in the netlist only have the functional
  information but no timing, power, or physical information exist.
- Reports about linting and design issues in the codes such as missing
  modules, multi-driven nets, width mismatch, etc.
- Reports about the cells, their types, and counts

```
Attributes:
    b - black box (unknown)
    c - control logic
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell                    Reference       Library         Area  Attributes
-------------------------------------------------------------------------
C4                      GTECH_OR3       gtech           0.000000  u
C10                     GTECH_OR3       gtech           0.000000  u
C13                     GTECH_OR3       gtech           0.000000  u
C18                     GTECH_OR2       gtech           0.000000  u
C32                     GTECH_OR2       gtech           0.000000  c, u
C33                     GTECH_OR2       gtech           0.000000  c, u
I_0                     GTECH_NOT       gtech           0.000000  u
I_1                     GTECH_NOT       gtech           0.000000  u
I_2                     GTECH_NOT       gtech           0.000000  u
I_3                     GTECH_NOT       gtech           0.000000  u
I_4                     GTECH_NOT       gtech           0.000000  u
I_5                     GTECH_NOT       gtech           0.000000  u
I_6                     GTECH_NOT       gtech           0.000000  c, u
current_state_reg[0]    **SEQGEN**                      0.000000  n, u
current_state_reg[1]    **SEQGEN**                      0.000000  n, u
current_state_reg[2]    **SEQGEN**                      0.000000  n, u
-------------------------------------------------------------------------
Total 16 cells                                          0.000000
```

Example: Cell Report From DC

Compile:

Compile is the most interesting and important step. Here we map the generic netlist into a technology-dependent netlist and then do several optimizations to meet the given constraints.

The constraints are supplied by the user and fall into 3 categories:

- Timing constraints: The clock frequency, the input/output delays, false paths, multi-cycle paths, max transitions, etc.
- Power constraints: Max dynamic power consumption, max leakage power consumption, etc.
- Area constraints

COMPILE STEPS:

 # Mapping:

In the case of ASIC, the generic cells are mapped to the cells defined inside the standard cell libraries.

#Optimization:

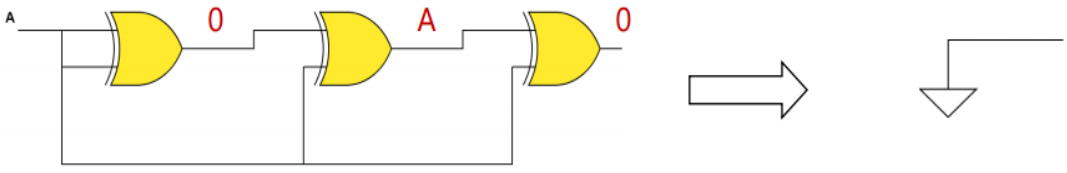
After mapping, the synthesis tool will do many optimizations and tradeoffs to fulfill the design constraints.

CONSTANT PROPAGATION

This optimization propagates constant values to remove redundant logic. For example, a 2-input AND where one of its inputs is a constant "0" will always produce "0"
regardless of the other input value.

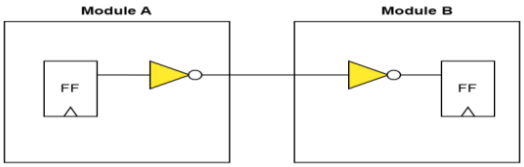
Consider the example below:

- The 1st XOR gate with both inputs having the same logic value will always produce "0". Therefore, we can remove the first XOR and propagate zero to the next XOR.
- The 2nd XOR has A in one input and "0" in the other. An XOR where one of its inputs is a "0" will pass the value of the other input as it is. Therefore, we can remove the 2ndXOR and propagate A to the next XOR.
- The 3rd XOR has A in both its inputs and so, will always produce 0. The entire circuit can be optimized away and a logic "0" be propagated to its output.



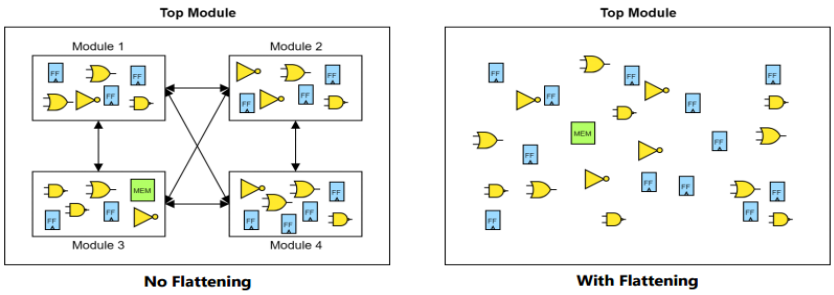
CROSS-BOUNDARY OPTIMIZATION

- The RTL code consists of several modules that are connected to each other.
- By default, the synthesis tools will synthesize each module separately and then connect them at the top module.
- In the example below, the inverter pair can be removed. But since the inverters exist in different modules the synthesis tool won't remove them.
- One solution to this is to enable cross-boundary optimization. This allows the tool to perform the optimizations and also propagate constants across the modules. The
- drawback is an increase in runtime.



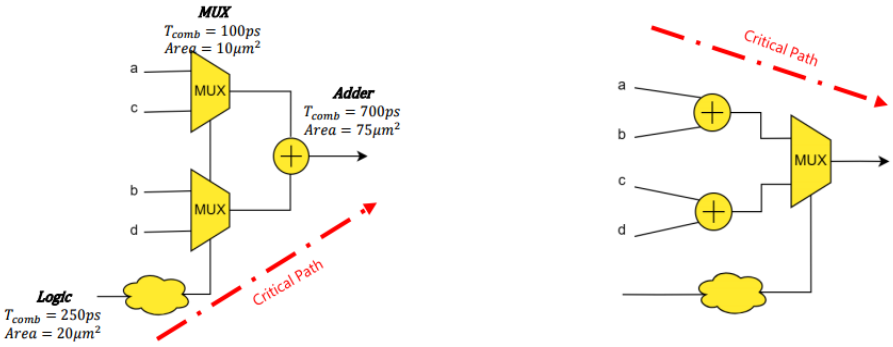
HIERARCHY FLATTENING

The other solution is to remove the module boundaries altogether. This is called flattening and generally produces better results but takes higher runtime and makes post-synthesis simulations and the PNR flow more difficult.



AREA VS TIMING

- Synthesis tools have switches to control the tradeoff between area vs timing.
- Consider the example below, both circuits perform the same function but:
  - The circuit on the left has less area 115 $\mu m2$ a but longer critical path 1050 ps (better for area)
  - The circuit on the right has more area 180 $\mu m2$ a but shorter critical path 800 ps (better for timing)
  - The synthesis tool will choose which circuit to use based on the user settings.



Types of ASIC Synthesis:

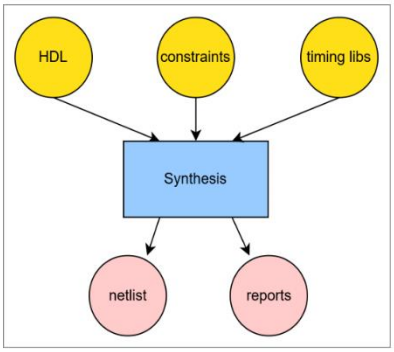
Logical Synthesis: Converts RTL (Register Transfer Level) code into a gate-level netlist. Key Concept is to Map logic level to gates like AND, OR, and NOT.

- Converts Boolean equations or truth tables into an optimized gate-level netlist.
- Targets area, power, and delay minimization.
- Common in standalone small-scale circuits or components

The inputs to ASIC synthesis are:

- **HDL:** The Verilog or VHDL design files
- **Constraints:** The timing (SDC), power, and area constraints
- **Timing Libraries:** The standard cell libraries.
- **Synthesis Commands:**
  Dc Command:

```
set target_library <STDCELL_LIBRARY>
set link_library1 "* $target_library io.db rams.db"
read_verilog <RTL FILES LIST>
current_design <TOP_MODULE_NAME>
link
source <TIMING_CONSTRAINTS>
compile #Synthesize the design
```



Genus Command:

```
set_db library <technology_library>
read_hdl <hdl_file_names>
elaborate <top_level_design_name>
source <TIMING_CONSTRAINTS>
syn_generic
syn_map
syn_opt
```

The outputs are:

- Design netlist (Gate level Netlist (.v file))
- Various reports about the design such as timing, power or area reports
- Synthesis Commands:

```
write -f verilog -o ./netlist.v
report_timing
report_power
report_area
```

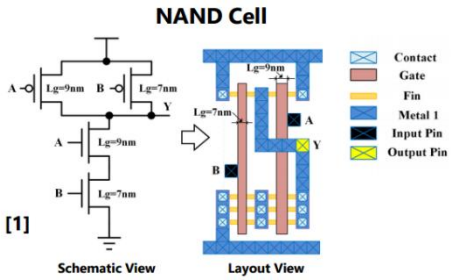
Genus Command:

```
report_timing >  <specify_timing_report_file_name>
report_area   > <specify_area_report_file_name>
write_hdl > <specify_netlist_name>
write_script > <script_file_name>
```

Important Parameters:

*TARGET LIBRARIES-

The target_library variable specifies the library that Design Compiler uses to select cells for optimization and mapping. The target library in ASIC is called a standard cell library:



- It's a collection of pre-designed and pre-characterized logic gates and other
  digital functions used for VLSI design.
- The information can be timing, power consumption, physical layout, logic
  functionality, etc
- This information is scattered into multiple files. For example, the timing and
  power information exist in timing lib/db file while the physical layout exists in a LEF/GDS files.
- These files are sometimes called "Views" (e.g. timing view) as they represent the cell info from a certain point of view.

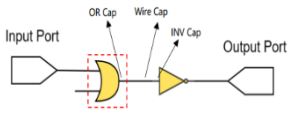
Input Transition Time t

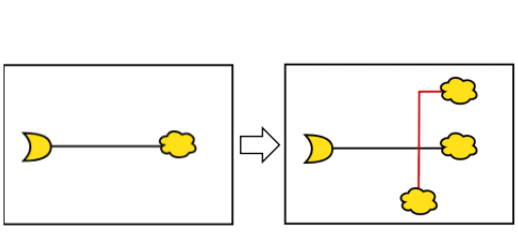
Load Capacitance C_L	1.1	1.2	1.3	1.4
10	2.10	2.20	2.27	3.00
20	2.50	3.00	3.45	3.96
30	2.90	3.40	3.80	4.15

Example Timing View[2]

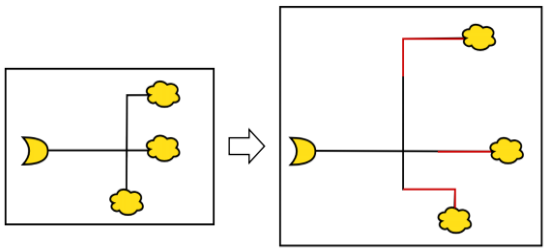
* WIRE LOAD MODEL (WLM)-

- For the synthesis to know the cell delay and power, it needs to know the input transition and capacitive load.
- Both values depend on the cell type and also the wires connecting the cells.
- The cell information is known from the standard cell library. So, the missing info is the wire values (resistance and capacitance).
- In older technologies, the wire values were estimated using a wire load model (WLM).
- This model estimates the length of a wire (and therefore the resistance and capacitance) based
  on the number of fanouts and the block size as shown in the diagrams
- These estimations are based on results from previous designs





More Fanouts => More Wire Length



Larger Block Size => More Wire Length

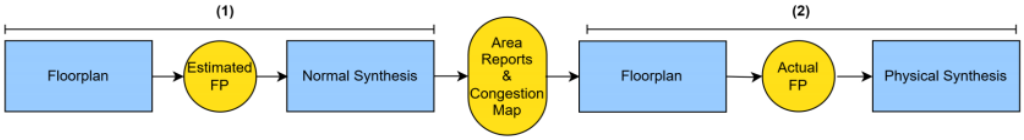
Physical Synthesis: In newer tech nodes the WLM produced bad estimations so tool vendors tried another approach called Physical Synthesis. In this approach the floorplan and physical info (techfile, cell layout, parasitics, etc) are passed to the synthesis.

- This allows the synthesis to do cell placement along with logic synthesis and optimization.

- Since, it knows the distance between the cells, the synthesis can more accurately estimate the expected wire length.
- Physical synthesis produces much better results compared to the WLM approach but has a longer design time
- Two-pass Synthesis: Tool vendors recommend doing physical synthesis in two steps:

1. Synthesize the design with an initial floorplan. The resulting netlist gives info about the cell counts total area, and congestion which enables us to create a better floorplan.
2. Create a new floorplan then redo the synthesis with the physical info.

One-Pass Synthesis (Not Recommended)



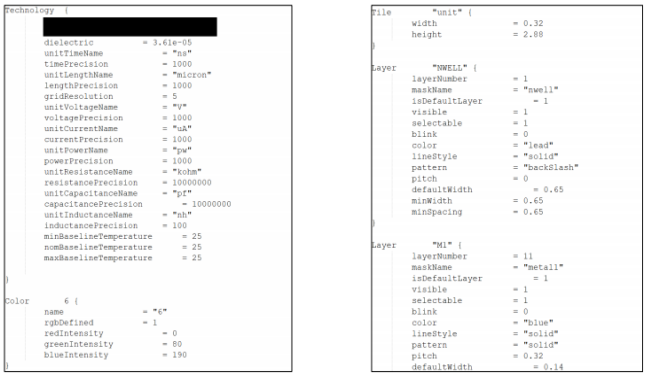
Two-Pass Synthesis (Recommended)

Physical Synthesis Inputs:

Tech File:

The tech file contains various info about the technology like:

- The units and precision.
- The coloring of the metals in the GUI.
- The minimum standard cell height and width.
- The design rules such as the layers' default width
- and spacing, etc.
- Via definitions



```
Technology (
        [redacted]
        dielectric              = 3.61e-05
        unitTimeName            = "ns"
        timePrecision           = 1000
        unitLengthName          = "micron"
        lengthPrecision         = 1000
        gridResolution          = 5
        unitVoltageName         = "v"
        voltagePrecision        = 1000
        unitCurrentName         = "uA"
        currentPrecision        = 1000
        unitPowerName           = "pw"
        powerPrecision          = 1000
        unitResistanceName      = "kohm"
        resistancePrecision     = 10000000
        unitCapacitanceName     = "pf"
        capacitancePrecision        = 10000000
        unitInductanceName      = "nh"
        inductancePrecision     = 100
        minBaselineTemperature      = 25
        nomBaselineTemperature      = 25
        maxBaselineTemperature      = 25
)

Color       6 (
        name                = "6"
        rgbDefined          = 1
        redIntensity            = 0
        greenIntensity          = 80
        blueIntensity           = 190
)
```

```
File        "unit" (
        width                   = 0.32
        height                  = 2.88
)

Layer       "NWELL" (
        layerNumber             = 1
        maskName                = "nwell"
        isDefaultLayer              = 1
        visible                 = 1
        selectable              = 1
        blink                   = 0
        color                   = "lead"
        lineStyle               = "solid"
        pattern                 = "backSlash"
        pitch                   = 0
        defaultWidth                = 0.65
        minWidth                = 0.65
        minSpacing              = 0.65
)

Layer       "M1" (
        layerNumber             = 11
        maskName                = "metal1"
        isDefaultLayer              = 1
        visible                 = 1
        selectable              = 1
        blink                   = 0
        color                   = "blue"
        lineStyle               = "solid"
        pattern                 = "solid"
        pitch                   = 0.32
        defaultWidth                = 0.14
```

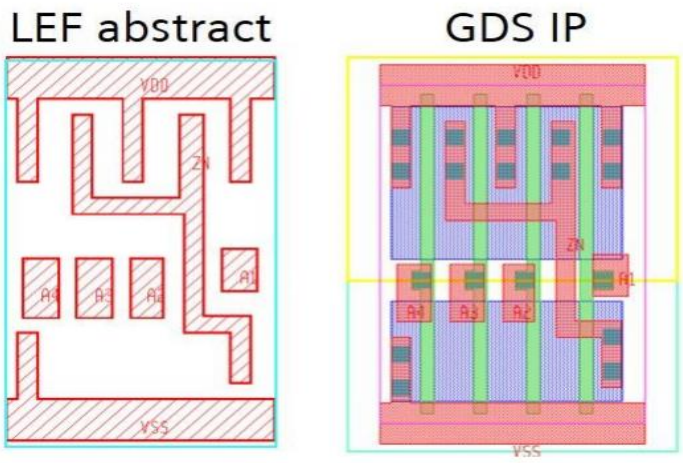
Example Tech File

ITF & TLUPLUS OR QRC TECH:

- The ITF (Interconnect Technology File) is a text-based file that contains raw information about each technology layer such as the thicknesses, resistivity, and dielectric constants
- These values are further processed to generate the TLU+ file which contains tables of R, and C values as functions of metal layer widths, and spacing. This is done while taking into account all adjacent layers' effects.
- The TLU+ contents are binary and only contain a text header showing the ITF that was used to generate the TLU+ file
- Along with TLU+, we use a layer mapping file that maps the layer names between the tech file and the TLU+

LEF (Library Exchange Format):

- The GDS file contains full data about the design layout and masks and is sent to the fabrication plant to fabricate the chip.
- From a runtime and memory usage point of view, we don't need all the info of the GDS when doing placement. We only care about the cell boundary, pin shapes and locations.
- The LEF file contains only the necessary info needed to perform placement and is used during physical synthesis and across the PnR stages.
- Once PnR is finished, the LEF views are replaced by the GDS views to produce the final GDS file that contains all the info needed by the fabrication plant



Synthesis Advance Options:

Register Duplication:

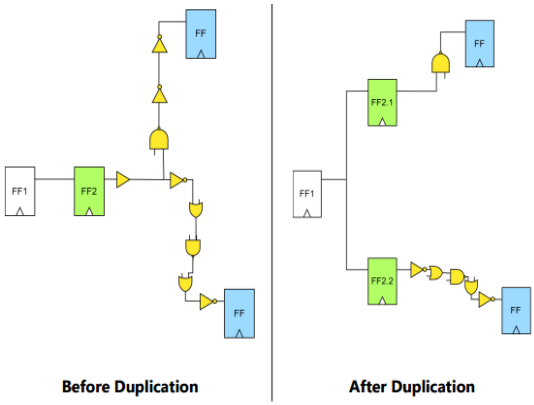
• By duplicating registers, the timing paths can be shortened, reducing the wire and cell propagation delays.

• This also reduces the fanout on the register which may enhance the output delay of the register

• **Consider the example on the right :**

- By duplicating the green registers we managed to move each copy near one of the blue register
- This first, reduces the wire length between the green and blue registers and second, allows us to remove the buffers and inverter pairs on the nets and both reduce the total combinational delay
- This shows that this method becomes more useful when the capture registers (the blue ones) are placed far away from each other in the chip.
- However, FF1 now drives double the fanout so the delay of the timing path between FF1 and FF2 is increased. We need to make sure this increase doesn't cause the path to violate setup timing.

• Duplication can be enabled globally or on a cell-by-cell basis



• **Synthesis Commands:**

DC Commnd:

```
set compile_register_replication true
```

#When this variable is set to true, compile tries to identify registers in the current design that can be split to balance the loads for better QoR.

```
set_register_replication -num_copies 3 <REGISTER>
```

#Duplicate a certain register 3 times.

Genus Command:

```
set_attr iopt_sequential_duplication true /     (In Legacy_UI)
set_db iopt_sequential_duplication true       (In Common_UI)
```

#With this attribute enabled, the tool will duplicate the flops on the critical timing path to improve the timing of this path during incremental synthesis (Command: syn_opt).

Legacy_ui

```
duplicate_register instance_list [ [-num_copies integer] [-fanout_list {{pin|port}...}]... | -timing integer] [-verbose] [-local] [-no_cross] [-score]

Common_ui

duplicate_register inst|hinst... [-num_copies integer] [-fanout_list {{pin|hpin|port}...}]... | -timing integer] [-verbose] [-local] [-no_cross] [-score]
```

#This command duplicates the specified register(s) to reduce the load on the register outputs and thus improving the timing.
The duplicated registers will be part of the same hierarchy as the original register. Duplication is controlled by the specified options. You can run this command any time after the design has been elaborated. If the design has been placed, the placement of the leaf instances in the fanout will be taken into account.

 # When to Use:

- High fanout nets causing timing violations.
- Critical paths with large capacitive loads.

Multi-Bit Banking:

ASIC standard cell libraries contain special flip-flops that can store more than one bit. These FFs are called multi-bit banking registers.

- The area of a multi-bit register is less than the total area of the registers if implemented individually.
- Also, the clock tree have less buffers (less area and power) when multi-bit banking is enabled.
- The disadvantage is the limited placement and since all the bits are forced to be placed at the same location.
- The other disadvantage is the limited CTS flexibility since all bits are forced to have the same clock latency which limits fixing timing violations using local skew
- optimizations.

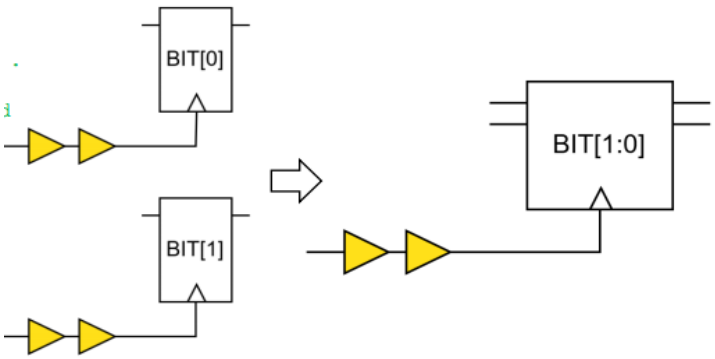
- **Synthesis Commands:**

```
set hdlin_infer_multibit [never | default_all | default_none]
```

#The **never** setting prevents inference of multibit components from HDL regardless of directives (Verilog) or attributes (VHDL).

#The **default_all** setting infers multibit components on all bused registers except where directives or attributes indicate otherwise.

#The **default_none** setting specifies that only attributes or directives are used to infer multibit components. This is the default for the hdlin_infer_multibit variable.



Genus Multibit Banking control:
How to control the default multibit merging behavior of Genus.

Mapping to multibit cells is enabled by setting the use_multibit_cells root-level attribute as true. Once enabled, mapping to multibit cells is performed by the syn_map and syn_opt steps of synthesis.
The ways to select or exclude the candidates for (from) Multibit Cell Mapping are as follows:

1. Add the following pragmas in the RTL:

```
// cadence merge_multibit
// cadence dont_merge_multibit
```

For example:

```
always // cadence merge_multibit "q"
@( posedge clk)
  q <= d;
```

Note: Pragmas can also be used for combinational cells.

2. Set the following attributes on instances in the netlist:

```
set_db <instance> .merge_multibit {true | false}
set_db <instance> .dont_merge_multibit {true | false}
```

Note: The pragmas in the RTL are also translated to these attributes during elaboration.

The following table shows when the tool considers multibit merging for the instance:

dont_merge_multibit	merge_multibit	Multibit merging is
false	false	attempted if use_multibit_cells is set to true.
false	true	attempted.
true	false	not attempted.
true	true	attempted. The dont_merge_multibit attribute setting is ignored.

1. For an automatic selection of candidates for multibit cell mapping, set the use_multibit_cells root attribute (default: false) prior to synthesis:

```
set_db use_multibit_cells true.
```

All single-bit cell candidates are replaced with the multibit cells during the incremental optimization.
2. To enable or disable the multibit mapping of all combinational cells, use the use_multibit_combo_cells root attribute (default: false).
3. To enable or disable the multibit mapping of all sequential and tristate cells, use the use_multibit_seq_and_tristate_cells root attribute (default: false).

When you set use_multibit_cells as true, both use_multibit_combo_cells and use_multibit_seq_and_tristate_cells are set to true.
For more information on multibit cell mapping, refer to ***Mapping to Multibit Cells*** in the *Genus Synthesis Flows Guide*.

How to instruct Genus to merge some specific flops to multibit flop.

There are two ways to do that:

1. By using the below flow:

```
set_db use_multibit_cells true                          // In Common UI
OR
set_attr use_multibit_cells true /          // In Legacy UI
```

- Set all sequential instances with the dont_merge_multibit attribute as true before syn_map as shown below:

In Common UI:

```
@genus:root: 23> get_db insts -if {.is_sequential==true} -foreach { puts "[set_db
$object .dont_merge_multibit true]"}
```

In Legacy UI:

```
foreach obj [filter sequential true [find / -instance *]] {
set_attr dont_merge_multibit true $obj
   }
```

- For those flops that you want to be merged to multibit, set them using the dont_merge_multibit attribute as false before syn_map as shown below:

```
set_db <instance> .dont_merge_multibit false                          // In Common
UI
set_attr dont_merge_multibit false <instance> /                    // In Legacy UI
```

Note: If you do not want some specific flops to be merged to multibit, follow the above two steps and in the third step, set the rest of the flops that you want to be merged to multibit using the merge_multibit attribute to 'true'.
OR
Follow the first step and for those flops that you do not want to be merged to multibit, set them using the dont_merge_multibit attribute to 'true'.

2. By setting those specific flops that you want to be merged to multibit using the merge_multibit attribute to true as shown below:

```
set_db <instance> .merge_multibit true                    // In Common UI
set_attr merge_multibit true <instance> /                 // In Legacy UI
```

#When to use:

- When multiple single-bit flip-flops are used in parallel.
- To optimize area and power in large designs.

Preferred MUX Implementation:

Standard cell libraries have the basic cells needed to build a MUX (2 AND ,1 OR ,1 Inverter) but also have integrated MUX cells.

- It's better to use the basic cells to build a MUX because each cell can be placed and optimized individually
  allowing for greater flexibility for placement and optimizations which produces better timing and area
  results.
- The problem is this approach increases the number of pins. For example, a 2:1 MUX will have 11 pins (6 pins
  for the 2 ANDs, 2 for Inverter, 3 for OR) compared to 4 pins for the integrated MUX (2 inputs, 1 output, 1
  selection).
- This might create pin congestion and make routing difficult. In such cases, it's better to use the MUX cells
- ASIC tools allow you to instruct the synthesis about which implementation it should prefer over the other.
- **Synthesis Commands:**

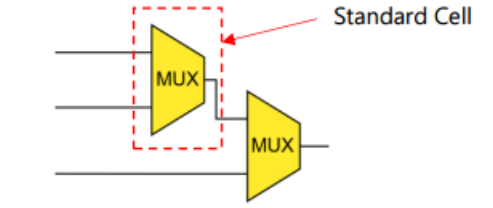
  DC Settings control:

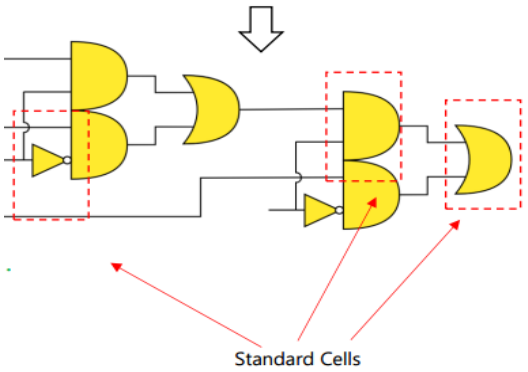
```
set compile_prefer_mux true
```

#The default flow typically maps most multiplexers to and-or-invert (AOI) logic in order to minimize area, but in some cases this can result in congestion hotspots. With compile_prefer_mux enabled, multiplexing logic that is likely to cause congestion is converted to MUX trees where possible.

```
set hdlin_infer_mux all
set_size_only [get_cells -hier * -filter "@ref_name =~ *MUX_OP*"]
```

#These commands forces the compiler to use MUX cells instead of the basic gates. However, this restricts the tool and might degrade QoR.





Genus preferred **mux control:**

There are two ways to force MUX mapping:

1. Using the map_to_mux pragma in the RTL code on every MUX logic.

```
module map2mux2(a,b,s,z);
   input a,b,s;
   output z;
   reg z;
   always @(a or b or s) begin
      if (s) // cadence map_to_mux
         z = a;
      else
         z = b;
      end
endmodule

module map2mux1 (a,sel,z);
   input [3:0] a;
   input [1:0] sel;
   output z;
   reg z;
   always @ (a or sel) begin
     case (sel) // cadence map_to_mux
        2'b00 : z <= a[0];
        2'b01 : z <= a[1];
        2'b10 : z <= a[2];
        2'b11 : z <= a[3];
     endcase
   end
endmodule

module map2mux3(a,b,s,z);
   input a,b,s;
   output z;
   assign z = s ? // cadence map_to_mux
    a : b;
endmodule
```

Note: User can use the input_map_to_mux_pragma attribute to define a Tcl list of one or more names, each being treated as map_to_mux pragma when reading in Verilog or VHDL.

In Legacy_UI

```
set_attr input_map_to_mux_pragma "map_to_mux infer_mux" / ; #Default:
map_to_mux infer_mux
In Common_UI
set_db / .input_map_to_mux_pragma "map_to_mux infer_mux"
```

2. Using the map_to_mux attribute on every binary mux instance in the design after elaboration:

```
In Legacy_UI
set_attr map_to_mux true [filter -regexp subdesign bmux [find /des* -
inst *]]
In Common_UI
set_db [ get_db insts * -if {.primitive_function == bmux}]
.map_to_mux true
```

#When to use:

- When specific mux implementations are more efficient for the target technology.
- To meet timing or power constraints.

Resource Sharing:

Resource sharing is a technique where **common logic operations** are shared among multiple operations to reduce **area** and **power**.

Example:

If two operations require an **adder**, a single adder can be shared between them instead of using two separate adders.

Genus Command:

```
dp_sharing {inherited | none | advanced}
```

#Controls resource sharing in datapath on the design during syn_generic with high effort. If the value is set to inherited, the effort level of the optimization on the design will be identical to ("inherited from") the effort specified at the root.

inherited	Inherits the value from the dp_sharing root attribute. **Note**: Only applies to design, hinst, and module.
advanced	Applies advanced level optimization.
none	Turns off optimization.

#When to Use:

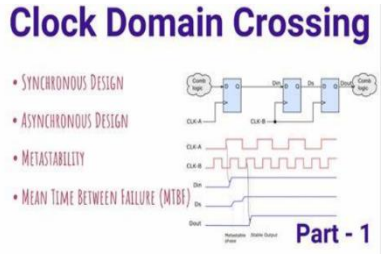
- When multiple operations use the same logic

Pre-Synthesis Sanity Checks:

These checks are performed **before running synthesis** to ensure the RTL design is clean, synthesizable, and ready for synthesis.

1.1. RTL Linting

- **Purpose**: Check for coding issues that could cause synthesis problems.
- **Tools**: SpyGlass, Ascent Lint, or Synopsys VC LP.
- **Checks**:
    - **Synthesizability**: Ensure the RTL is synthesizable (e.g., no combinational loops, latches, or undefined states).
    - **Clock Domain Crossing (CDC)**: Verify proper handling of signals crossing clock domains.
    - **Reset Issues**: Ensure proper reset handling (e.g., synchronous vs. asynchronous resets).
    - **Unused Signals**: Identify and remove unused signals or logic.
    - **Multi-Driven Nets**: Check for nets driven by multiple sources.



1.2. Clock and Reset Checks

- **Purpose**: Ensure clocks and resets are properly defined and handled.
- **Checks**:
    - **Clock Definitions**: Verify all clocks are defined with correct frequencies and waveforms.
    - **Reset Synchronization**: Ensure resets are properly synchronized to avoid metastability.
    - **Gated Clocks**: Check for proper clock gating implementation.

1.3. Timing Constraints Review

- **Purpose**: Ensure timing constraints are complete and accurate.
- **Checks**:
    - **Clock Definitions**: Verify all clocks are defined with correct periods and waveforms.
    - **Input/Output Delays**: Ensure input and output delays are specified for all ports.
    - **False Paths and Multicycle Paths**: Verify false paths and multicycle paths are correctly defined.
    - **Max/Min Delay Constraints**: Check for any additional delay constraints.

1.4. Design Readiness

- **Purpose**: Ensure the design is ready for synthesis.
- **Checks**:
    - **Hierarchy**: Verify the design hierarchy is clean and modular.
    - **Black Boxes**: Ensure all black boxes (e.g., IP blocks) are properly instantiated.
    - **Library Files**: Verify all required library files (e.g., standard cell libraries) are available.

Post-Synthesis Sanity Checks:

These checks are performed **after synthesis** to ensure the gate-level netlist is valid, meets timing, and is ready for physical design.

2.1. Netlist Quality Checks

- **Purpose**: Ensure the synthesized netlist is clean and free of issues.
- **Checks**:
  - **Unconnected Ports**: Check for unconnected input/output ports.
  - **Floating Nets**: Identify and fix floating nets.
  - **Multi-Driven Nets**: Ensure no nets are driven by multiple sources.
  - **Unused Cells**: Remove unused cells or logic.

Example:

```
            // Bad: Unconnected port
module my_module (
   input clk,
   input data_in,
   output data_out
);
   assign data_out = data_in;
endmodule

// Good: All ports connected
module my_module (
   input clk,
   input data_in,
   output data_out
);
   reg q;
   always @(posedge clk) begin
      q <= data_in;
   end
   assign data_out = q;
endmodule
```

2.2. Timing Checks

- **Purpose**: Verify the design meets timing constraints.
- **Checks**:
  - **Setup/Hold Violations**: Check for setup and hold violations.
  - **Clock Skew**: Verify clock skew is within acceptable limits.
  - **Critical Paths**: Analyze critical paths and ensure they meet timing.

  - **Timing Exceptions**: Verify false paths and multicycle paths are correctly applied.
- **Example**:

```
# Check setup violations
report_timing -setup

# Check hold violations
report_timing -hold
```

2.3. Power Checks

- **Purpose**: Ensure the design meets power constraints.
- **Checks**:
  - **Dynamic Power**: Verify dynamic power is within the budget.
  - **Leakage Power**: Check for excessive leakage power.
  - **Clock Gating**: Ensure clock gating is properly implemented.
- **Example**:

```
# Report dynamic power
report_power -dynamic

# Report leakage power
report_power -leakage
```

2.4. Functional Equivalence Checks

- **Purpose**: Verify the gate-level netlist matches the RTL functionally.
- **Checks**:
  - **Equivalence Checking**: Perform formal verification to ensure the netlist matches the RTL.
  - **Debugging**: Debug and fix any mismatches.
- **Example**:

```
#Run Formal Verification using Formality or Conformal flow
```

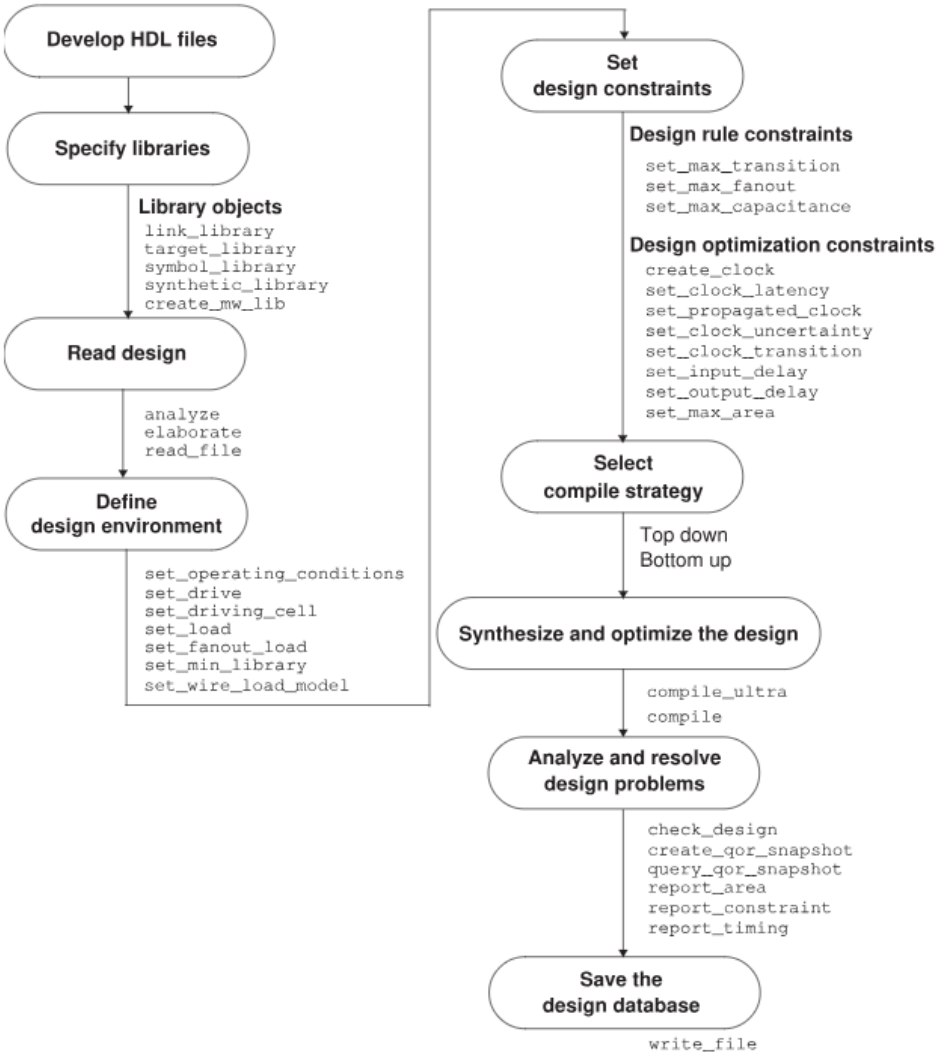
2.5. Testability Checks

- **Purpose**: Ensure the design is testable after manufacturing.
- **Checks**:
  - **Scan Chain Insertion**: Verify scan chains are properly inserted.
  - **Test Coverage**: Ensure test coverage meets the target (e.g., 95%+).
  - **ATPG**: Generate and verify ATPG (Automatic Test Pattern Generation) patterns.
- **Example**:

```
#insert_scan
#generate_atpg
```

Synthesis Commands:

Synopsys DC Commands:



Cadence Genus Synthesis Main Commands template:

The following script is a simple script which delineates the very basic Genus flow.

```
# ******************************************************

# *

# * A very simple script that shows the basic Genus flow

# *

# ******************************************************

set_db init_lib_search_path <full_path_of_technology_library_directory>

set_db init_hdl_search_path <full_path_of_hdl_files_directory>

set_db library <technology_library>

read_hdl <hdl_file_names>

elaborate <top_level_design_name>

set clock [create_clock -period <periodicity> -name <clock_name> [clock_ports]]

external_delay -input <specify_input_external_delay_on_clock>

external_delay -output <specify_output_external_delay_on_clock>

syn_generic

syn_map

report_timing >  <specify_timing_report_file_name>

report_area   > <specify_area_report_file_name>

write_hdl > <specify_netlist_name>

write_script > <script_file_name>

quit
```

#Synopsys Useful Variables and attributes can be found in this below link:
Synthesis Variables and Attributes

Synthesis Issues & Solve in N3E Flow:

1. Some of the Sequential flops were removed getting constant logic from RTL.

Information: The register'cluster/neuron_pool/alu/gen_alu_lanes(0)/dnn_alu_readout/s3_mux_state_quant_reg[6]' is a constant and will be removed. (OPT-1206)

In the `n3 wrapper` Verilog, the output pin `ROP` was forcefully tied to a constant low (`0`). This means the pin was hardwired to always output `0`, regardless of the design's operation.

- Additionally, the `.lib` file for the `N3 SRAM` (which contains timing and functionality information for the SRAM) did not include any details about the `ROP` pin. This means the synthesis tool couldn't determine how `ROP` should behave or connect in the design.

How the Problem Was Identified:

1. **Synthesis Logs:**

- The synthesis tool generated a warning/note (`OPT-1206`) indicating that the register was constant and removed. This helped identify that the register was unnecessary.

- The issue with the `ROP` pin was likely flagged during synthesis or subsequent verification steps, as the tool couldn't find corresponding information in the `.lib` file.

```
        );


// ------------------------------------------------------------------------------
// Output assignments for stub mode
// ------------------------------------------------------------------------------

        assign                                          Q = q;
        assign                                          ROP = 1'b0;
           // FIXME: no output from TSMC

endmodule
`endcelldefine
```

2. **Design Analysis:**

- By reviewing the RTL (Register Transfer Level) code, it was confirmed that the register `s3_mux_state_quant_reg[6]` was indeed getting a constant logic and not contributing to the design's functionality.

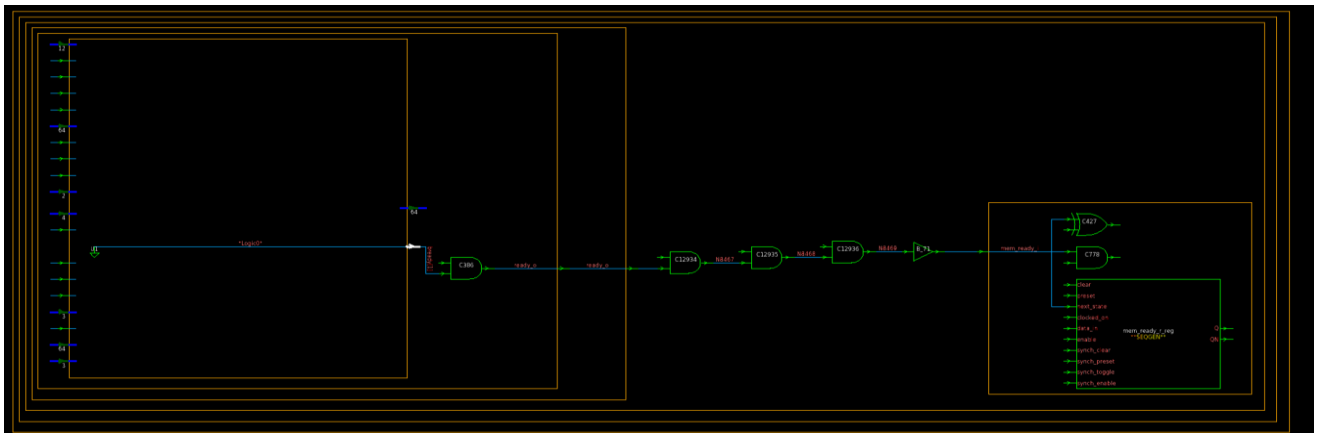


Figure: ROP output pin drives data to the sequential cells pin through combo logic

- The `ROP` pin issue was traced back to the `n3 wrapper` Verilog, where it was tied low, and the missing information in the `.lib` file was identified.

Key Takeaways for Beginners:

1. Synthesis Optimization:

- Synthesis tools remove unnecessary logic (like constant registers) to optimize the design. This is normal but should be verified to ensure it aligns with the design intent.

2. Pin and Library Issues:

- If a pin is tied to a constant or lacks information in the `.lib` file, it can cause issues during synthesis or verification. Always cross-check the RTL, wrapper, and library files for consistency.

3. Debugging Steps:

- Use synthesis logs and warnings to identify issues.
- Review RTL and library files to understand the root cause.
- Verify if the issue is intentional or needs correction.

References:

1. Genus Synthesis Flows Guide -- Table of Contents
2. Preparing for Synthesis
3. Amr Adel on LinkedIn: Logic Synthesis All Parts
4. https://youtu.be/WeRkpkBDFGs?si=uYVJythInrcfmxF1