Power Matters.™

RTG4 Timing Optimization and Closure Space Forum 2017

Hichem Belhadj
VP, Field Applications and Systems Engineering

Overview

- Timing challenges are a concern for all FPGA designers
- Space FPGA designers have the additional consideration of needing to make sure their design is robust against radiation effects while trying to solve these timing challenges
- Space FPGA designers are also conscious of power dissipation
- This presentation focuses mainly on timing closure, but radiation effects are considered when applicable
  - Timing and power-aware techniques are mentioned when appropriate

Timing Closure Methodology in Five Questions

- The first question to ask is, “Is my performance goal reasonable or not?”
- If the answer to the previous question is yes, the second question is, “How far am I from achieving my goal?”
- If the answer is, “It is far but not impossible,” the third question is, “What’s causing the timing challenge?”
- Once the root cause or causes of the challenge have been identified, the next question is, “What can I do about it?”
- The final question is, “Is what I did enough to increase functionality?”

Outline

- How does Microsemi address timing challenges?
- How tough are the challenges?
- Root causes, identification, and scope of these challenges
- Dealing with timing challenges for pure logic designs
- Dealing with timing challenges for DSP-intensive designs
- Dealing with timing challenges around instantiate IPs and their interfaces

Outline (continued)

- Timing challenges
  - Continuous tools' QoR improvements
  - Dealing with logic depth or a large number of logic levels
  - Dealing with IO timing challenges
    - Explicit reduction of driver fanout
  - Dealing with high-fanout signals and nets
  - Dealing with DSP-intensive IPs or designs
  - Dealing with interface IPs

How Does Microsemi Address Timing Challenges?

RTG4 Libero SoC Support and Continuous QoR Improvement

Libero SoC 11.7 SP2—RTG4 QoR

- Production timing—QoR improvement over the releases of Libero SoC

Device	Capture	Date	# Designs	Delta QoR
RT4G150*	11.5.7.11	Feb. 2015	344	**+7.2%**
RT4G150*	11.6.0.34	Sep. 2015	344	**+4.2%**
RT4G150	11.7.0.119	Feb. 2016	345	**+12.0%**
RT4G150	11.7.1.9	May. 2016	346	**+7.0%**
RT4G150	11.7.1.14	Aug. 2016	346	**+7.2%**

Libero v11.8

- Synplify Pro Enhancements
  - Libero SoC v11.8 includes a new version of Synplify Pro ME (L2016.09M-2) with enhancements for RTG4:
    - Infer wide-Mux hard macros
    - Bus-aware replication of registers on select lines of Mux
    - Pack enable signal with higher priority than sync-reset into SLE
    - Logic reduction with tied inputs
    - Infer Enable on address register of uSRAM
    - RTG4 LSRAM: do not infer Feed-Through Write mode
    - RTG4 Math: infer only one asynchronous reset
- RTG4 SgCore Changes

The following SgCores have been updated to be compatible with Libero v11.8. If an existing project contains any of the following cores, update the core to the latest version and regenerate the component before continuing with the design flow in Libero SoC v11.8.

Core	Libero 11.8 Compatible Version	Changes
RTG4FCCC	1.1.217	Hold violation on Dynamic CCC on APB_S_PSEL and APB_S_PADDR paths
RTG4CCCAPB_IF	1.1.106	

v11.8—Release Notes Excerpts

- 1.2.11 RTG4 Global Net Report—Clocks and Resets not Radiation-Protected
  - The Global Net Report for RTG4 designs in Libero SoC v11.8 appends a new warning section that lists all clock nets and asynchronous reset/preset nets whose implementation may not be protected from radiation upsets.

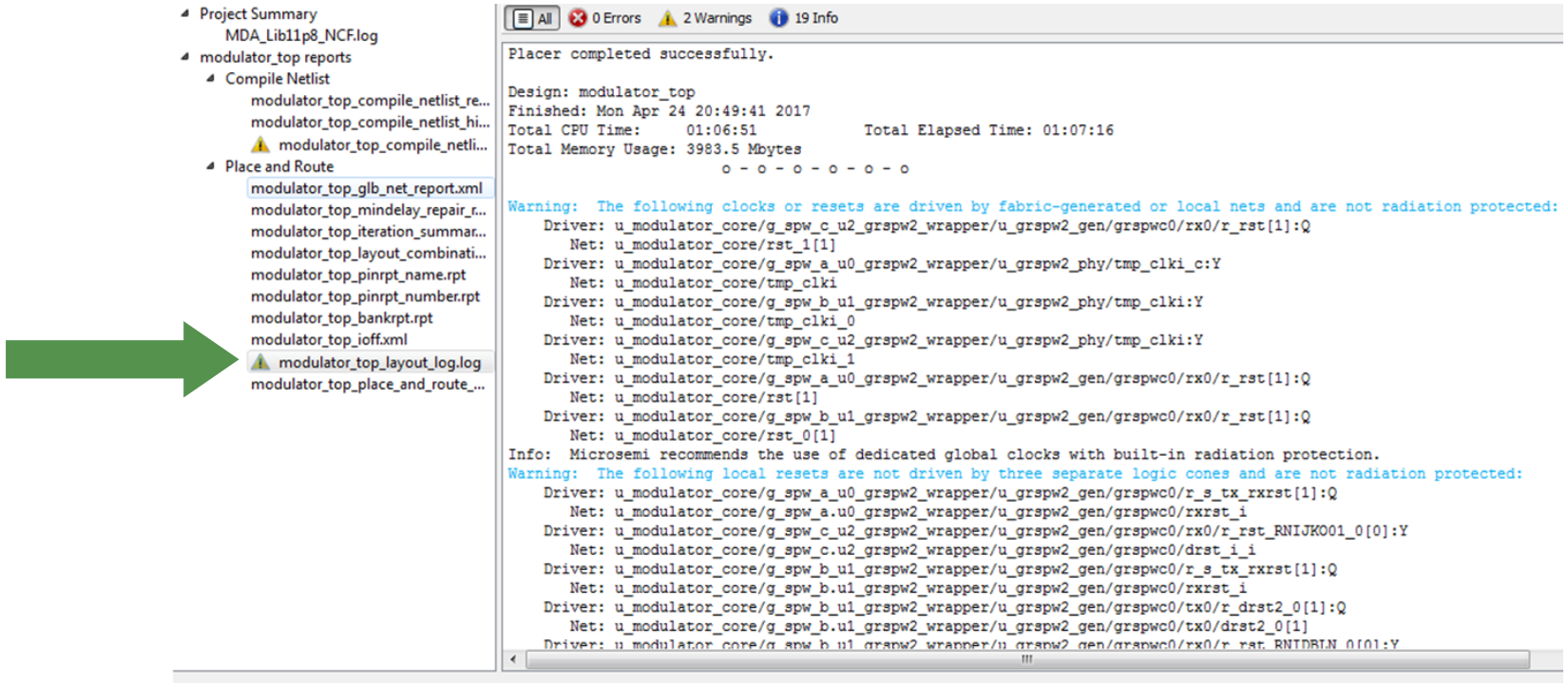
- 1.2.12 RTG4 CCC—Simulation Runtime Improvement
  - The CCC simulation model for RTG4 designs in Libero SoC v11.8 has been optimized to run up to three times faster.

- 1.2.13 RTG4 Single Event Transient (SET) Mitigation—Option Location Change
  - The SET Mitigation option for RTG4 designs has been relocated to the Device Settings window of the New Project Wizard (**Project > New Projects**) and the Project Settings dialog box (**Project > Project Settings**).

- To save time, please read the software release notes thoroughly.

Libero 11.8 Rad-Hard Global Reports

- Libero SoC 11.8—Layout Log Report lists all global clocks and resets that are NOT implemented as radiation-tolerant
  - This means the clocks and resets need TMR logic or glitch filtering.
  - Resets on RGREST need to be driven by three separate logic cones.
  - Clocks need to pass through global buffer.



How Tough Are the Timing Challenges?

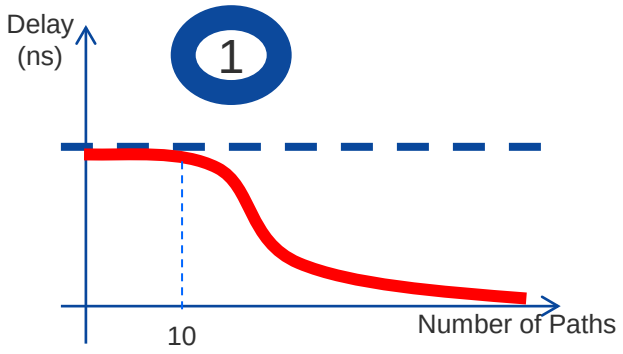
Assessment Techniques

- Assuming that your timing constraints are reasonable (~15% higher than the required specifications) and that you explicitly identified all multi-cycle and false path constraints, then do the following for each clock domain:
  - Draw the slack distribution chart that explicitly shows the slack (negative or positive) of the 100+ most critical paths
  - Identify if the scope of the timing challenges. For example, are the top critical paths are confined within one module, or are they crossing multiple functional or hierarchical blocks?
- For the entire design, review all the clock domains and don't focus on the most critical one
  - The clock domain meeting the spec may pop up as critical if you do not take care of them
  - The clock domains that meet the spec with a nice margin may help you

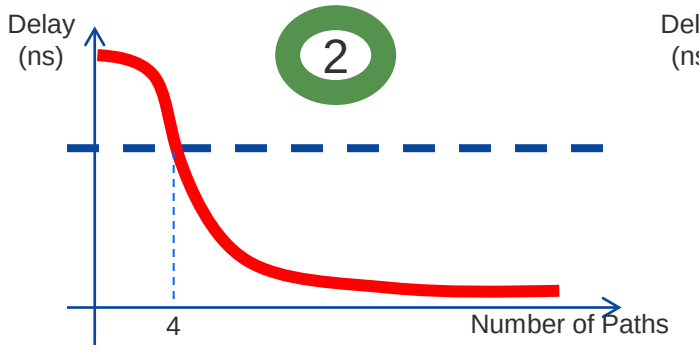
Slack Distribution Charts

- The slack distribution chart depicts the slack (negative or positive) of the most critical paths with respect to the timing constraint
- Identifying the paths with positive slacks is as important as identifying the paths with negative slack (or timing violations) to determine the complexity of the timing optimizations
- Interpretation of the slack distribution profile will help users check how much leeway they have with each clock domain (see next slide for more information)

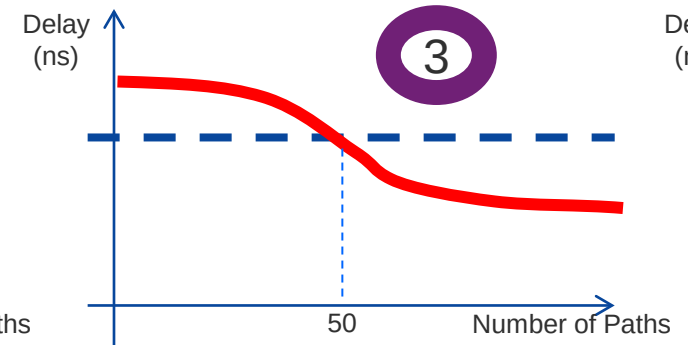
Slack Distribution Charts—Profiles and Interpretations



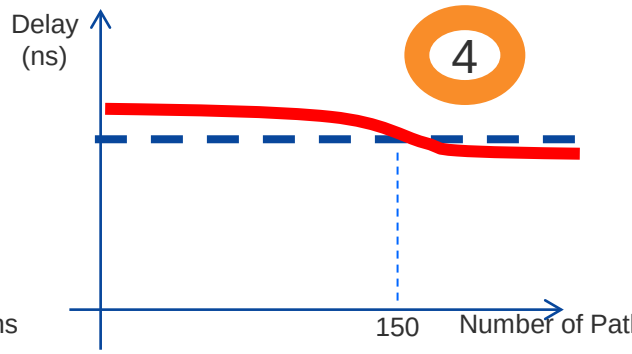
- All paths are meeting spec
- Handful of paths (10) meet marginally
- Other paths have very large positive slack
- Slight improvement of the top ten paths **will not disturb** the profile of the slack distribution for this clock domain
- User may relax the frequency constraints and use set_max_delay on the top ten paths.



- Handful of paths are not meeting spec
- Other paths have large positive slack
- Improvement of the top paths **will not disturb** the profile of the slack distribution for this clock domain
- User may relax the frequency constraints and use set_max_delay on the top ten paths.



- Several paths are not meeting spec
- Other paths have decent positive slack
- Improvement of the top paths **may or may not disturb** the profile of the slack distribution for this clock domain
- In addition to clock frequency, user may further tighten the constraint on the top paths only



- Many paths are not meeting spec
- Many paths have very slim positive slack
- Improvement of the top paths **will disturb** the profile of the slack distribution for this clock domain
- In addition to clock frequency, user may further tighten the constraint on the top paths only

Scope of the Timing Challenge?

- If the critical paths are confined within one block or module, the challenge is easier to tackle than when these critical paths cross several blocks/modules
- To identify the scope, check the source and sink of the top critical paths
  - If they belong to the same block/module, then expand two or three of these paths to make sure all the nets and cells belong to the identified module
- If the critical path crosses two or more blocks/modules, add registers at the boundaries of these blocks if the overall latency allows
  - If not, remove one hierarchy level and allow optimization across hierarchical boundaries

Timing Challenges:
Root Causes, Identification, and Scope

Outline

- What causes timing challenges?
- How to identify the root cause(s)?

What Causes Timing Challenges?

- Timing challenges could be caused by one or more of the following root causes:
  - Deep logic cones from/to IOs or from register to register
  - Existence of high-fanout control or data nets
  - DSP- or RAM-intensive blocks that interact with logic and IOs
  - Congested designs due to cross-bar, barrel-shifters, or multiplexors/de-multiplexor structures
  - Too many asynchronous and synchronous reset signals
    - A reset signal per clock domain is not good design practice unless power-saving is a MUST (Suspend the activity of some blocks using reset)

- Timing challenges could also be caused by tools or user options and constraints:
  - Poor synthesis or logic mapping results
  - Lack of or very stringent physical and timing constraints
  - Locally inefficient placement or routing
  - Floor plan constraints create artificial congestion

How to Identify the Root Cause(s)?

- Take the following steps to identify deep logic cones from/to IOs or from register to register
  - Expand the top critical paths and check the number of cells involved
  - If the number of cells in the path is above five, your design is considered to have deep logic and it will be difficult to meet the timing spec
    - Use the 50–50 or 45–55 rules of thumb (45% of the path delay is logic and 55% is routing related) unless one or more nets have high fanout
- To identify high-fanout control or data nets
  - Check the Compile report for the highest fanout nets
  - Expand the top critical paths and check the delay penalty associated with each net. Nets with high delay penalty usually have a high fanout (check the FO column)
  - If the net's fanout is less than 24 and the delay penalty is 4 ns or larger, then you have a placement issue and not a high-fanout net situation
- To identify DSP- /RAM-intensive blocks that interact with logic and IOs
  - Check the resource utilization reports from Synthesis and Compile

How to Identify the Root Cause(s)? (continued)

- Perform the following tests to identify if the issue is the inherent congestion of a design
  - Expand the top 20 critical paths—if the number of logic levels is low (less than four) and the fanout of the nets is low (below 20), yet the timing associated with the nets is high, then the design is potentially congested
  - RTL code includes a lot of busses and the logic utilization for the blocks including these busses is relatively low (less than 1000 LUTs)
  - RTL code includes many case statements that are mapped into Mux structures (not for FSMs)
  - See the Multiplexor Structures section for more information
    - Hidden Mux structures
    - RTL explicit Mux structures

How to Identify the Root Cause(s)?

- How to identify if you have too many Asynchronous and Synchronous Reset Signals
  - Check the Compile report for the high-fanout signals/nets and identify the asynchronous and synchronous reset signals
  - If utilization of global routing resources is high (80% and over) and several reset signals (Fanout Compile Report) are not mapped to global routing, then your design will have challenges meeting timing, as these high-fanout reset signals will be using regular routes
    - These reset signals will fight with other nets for routing resources
    - These reset signals may now cause removal time issues
  - RTL code for instantiation of IPs or blocks developed by other teams could lead to an unintended high number of reset signals in the design
    - User must reduce the number of asynchronous reset signals to a strict minimum. This will reduce design congestion, timing challenges, and radiation effects.

Identifying Root Cause(s) From Tools and User Settings/Constraints

- How to identify if the issue is poor synthesis or logic mapping results
  - Check if the timing report shows several cells made of two or three input LUTs
  - Check if deep logic paths are built without using embedded carry chain routing
  - Check the efficiency of the synthesis inference of RAMs and DSP blocks
    - RTL code intended to be mapped efficiently based on configurations of embedded SRAM may have been mapped into less efficient cascaded SRAMs
  - RTL code may force the hand of the Synthesis tool
  - For example, priority encoding (nested if statements) vs. balanced Mux structures (case statement)
    - IP netlists and inhibited cross-hierarchy optimization
- Synthesis options (particularly resource sharing, along with FSM encoding, optimization across hierarchy) may also lead to poor implementations

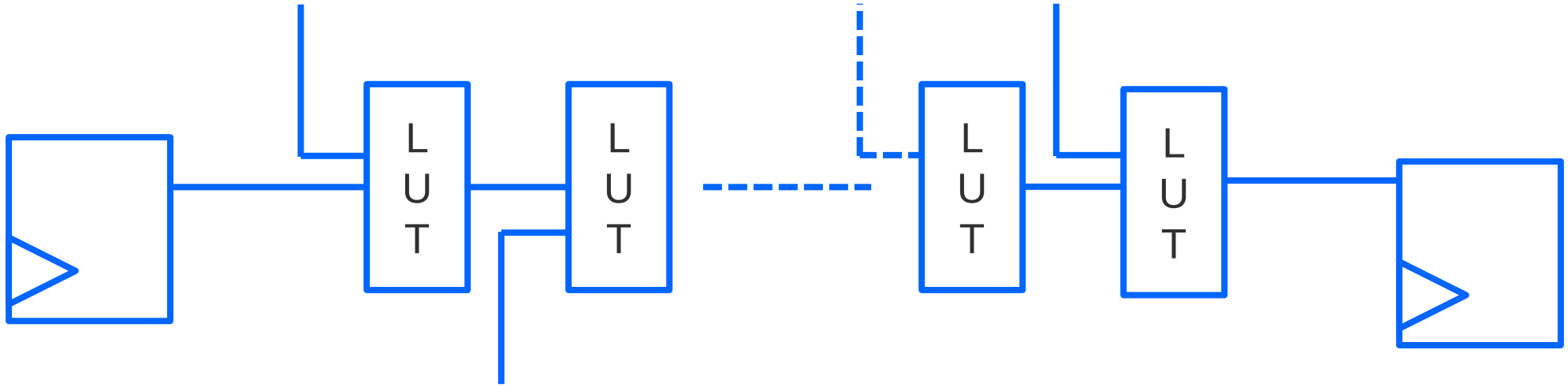
Dealing with Timing Challenges for Pure Logic Designs or Blocks

Outline

- Handling deep logic paths
- Fixing high-congestion designs
  - Focus on Mux structures
- State machines coding style and state encoding options
- Dealing with high and medium fanout control and data nets
- Resets and clocks

Symptoms of Too Many Logic Levels/Depth of Logic

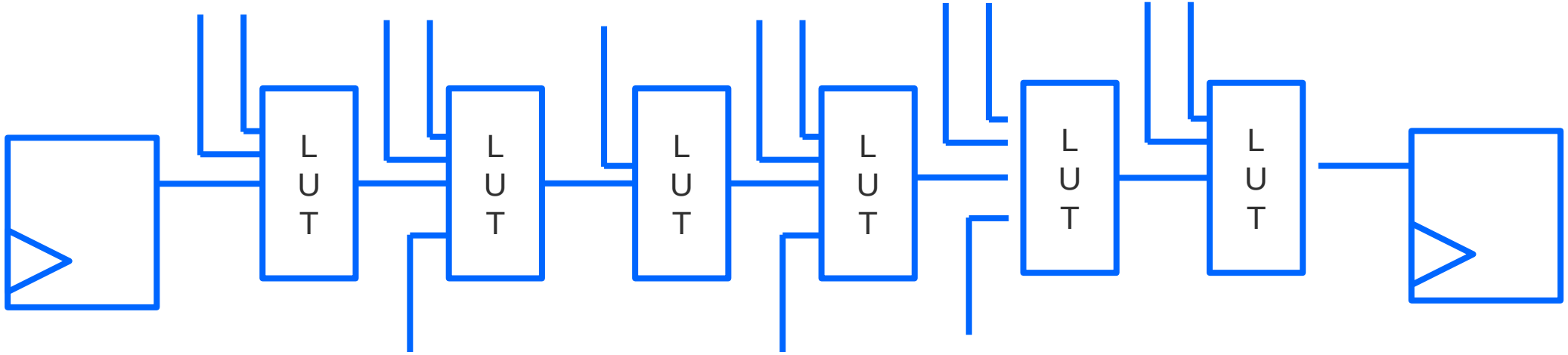
- If a critical path includes a too many logic levels and each LUT in the path has only two or three inputs, then the following problems may occur
  - Because LUT4 are able to map any logic function of four inputs, there is a waste of capacity when LUT4 are mapped as two- or three-input LUTs
  - Either synthesis is doing a poor mapping, timing constraints are very loose, or the RTL logic expressions need to be rewritten
  - A simple review of the RTL code will reveal these cases



Large Number of Logic Levels in Boolean Equations

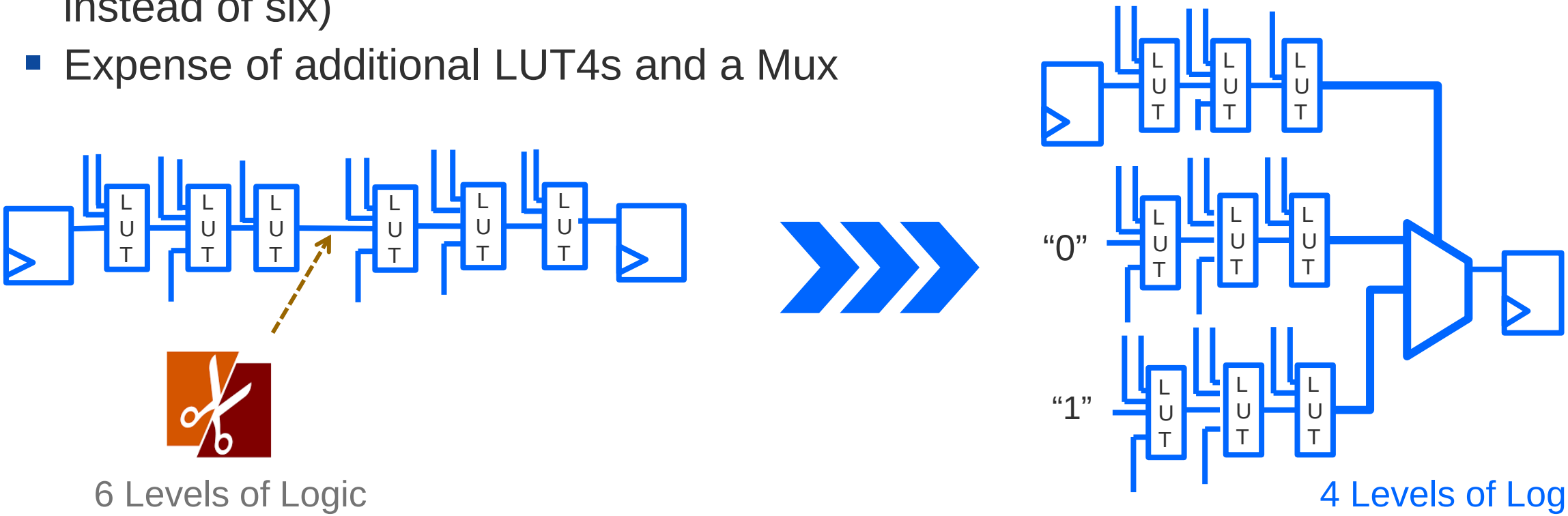
- Investigate if you can balance the number of logic levels for all the primary inputs by using parenthesis in your RTL code

- If not, rewrite your RTL code using the “delay the decision” technique
  - For example, consider the following illustration of a critical path with six logic levels



Reducing the Logic Depth—ALAP Decision

- The “delay the decision” technique balances the number of logic levels (now four instead of six)
- Expense of additional LUT4s and a Mux



- This technique is also used to push the toggling down the depth of the logic and helps reduce power dissipation

Reducing the Number of Logic Levels of Boolean Equations: Beware of Counters!

- In many designs, several values of counters are used to decide the execution of several branches of if statements
  - Example:

```
If (Count = x) then  … … …
   elseif ((Count = y or Count = z) and <expression1>) then
   elseif ((Count /= t) or <expression2>) then
   … … … …
   else ….  endif;
```

- If the binary counter is four or five bits long, consider using a 16- or 32-bit 1-Hot Shift register to implement the counter instead.
  - Each value of the counter is represented by the HOT bit of the shift register instead of four or five variables
  - This leads to
    - Simplification of large expressions such as (Count = y or Count = z) and <Boolean Expression1>
    - Less LUTs and fewer logic levels
    - More registers

Reducing the Number of Logic Levels of Boolean Equations: State Machines Encoding

- In many designs, users use state machines (FSMs) to sequence processing of data and control signals changes.
- If several registers or control signals are manipulated in many states of the FSM, use One Hot to encode the states and reduce the combinatorial logic associated with the product terms associated with the codes of these states.
- For example, consider a 12-state FSM assigning the same value “V” to register “R” in four states. The logic driving the selection of V in the Mux in front of R depends on the product terms associated with these states.

Reducing the Number of Logic Levels of Boolean Equations: State Machines Encoding

- If these four states are encoded as “0001,” “0011,” “0101,” and “1100,” the select line will be an OR of the product terms associated with these codes

  OR (!S0&!S1&!S2&!S3, S0&S1&!S2&!S3, S0&!S1&S2&!S3, !S0&!S1&S2&S3) or at least two logic levels

- If these 4 states are encoded as “00000000001,” “00000000100,” “00000010000,” and “10000000000,” the select line will be driven by

  OR (S0, S2, S4, S11) mapped with 1 LUT or 1 Logic Level

- The next section has more recommendations for state machine coding styles

Dealing with Large Number of Logic Levels: Anticipate (ASAP) or Delay (ALAP)

- Investigate if a processing (logic or arithmetic) can be done either in a previous clock cycle (as soon as possible) or in a future cycle (as late as possible)
  - If not possible, investigate if a partial pre-computation can be performed in a previous clock cycle or if a partial post-computation can be performed in the next clock cycle, even if this will add logic

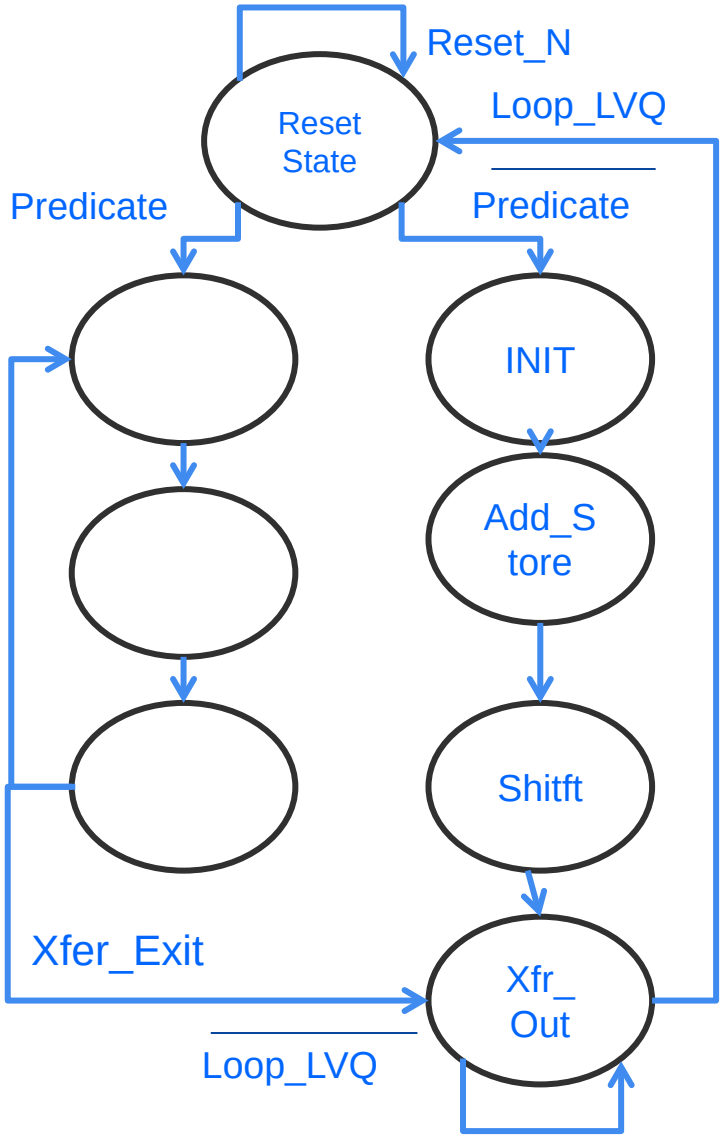
```
@(posedge clock) //down counter
if (condition_1==true)
   count <= 8'hff;
 else
 if (condition_2==true)
     count = count-1;


@(posedge clock)
 wide_bus[63:0]=
(count == (8'b0) ? Data1 : data2;
```

```
@(posedge clock)
if (condition_1 == true)
  count <= 8'hff;
  count_is_0 = '0';
else if (Condition_2 == true) begin
    count <= count - 1; //down counter
    if (count == 8'h01) count_is_0 = 1;
        else  count_is_0 = 0;
end


@(posedge clock)
wide_bus[63:0] =
   (count_is_0==1) ? Data1: Data2;
```

Dealing with Large Number of Logic Levels: Anticipate (ASAP) or Delay (ALAP)



- In this example, the computation of R1 + R2 – Rot is split into a pre-computation R1 + R2 in the Add_Store state, and the final computation when actually needed in state Xfer_out
- As a result
  - No cascaded arithmetic (fewer logic levels)
  - Data dependencies respected

```
… … …
If (State = INIT)
    R1 <= Value_init
    R2 <= Default_init
    Rot <= In1

If (State = Add_Store)
 Intermediate <= R1 + R2 – Rot;

 if (State = Shift)
     Rot <= Rot << 2;

 if (State = Xfer_Out)
    OutRes <= Intermediate;

… … ….
```

```
… … …
If (State = INIT)
    R1 <= Value_init
    R2 <= Default_init
    Rot <= In1

If (State = Add_Store)
  //  Intermediate <= R1 + R2 - Rot;
      Intermediate <= R1 + R2;

 if (State = Shift)
    begin
    Intermediate <= Intermediate – Rot;
     Rot <= Rot << 2;
    end;

 if (State = Xfer_Out)
    OutRes <= Intermediate;
… … ….
```

Dealing with Large Number of Logic Levels: Last Resort

- If the previous actions do not reduce the number of logic levels to meet the timing, then the Synthesis Retiming option should be set to ON

- While Retiming may help reduce the number of logic levels, it comes with a cost
  - Additional registers inserted to break the depth of the logic cones
  - Additional one or two clock cycles will increase the latency of the data transaction

- Beware
  - If the critical and deep logic paths are within a particular module or block of a design, use retiming locally to that module (for example, associate retiming attribute to a module when the critical and long logic path is located)
  - If the critical path is crossing two or more blocks, revisit your RTL code and insert explicit registers at the boundaries of these modules.
  - Do not apply retiming to the entire design.

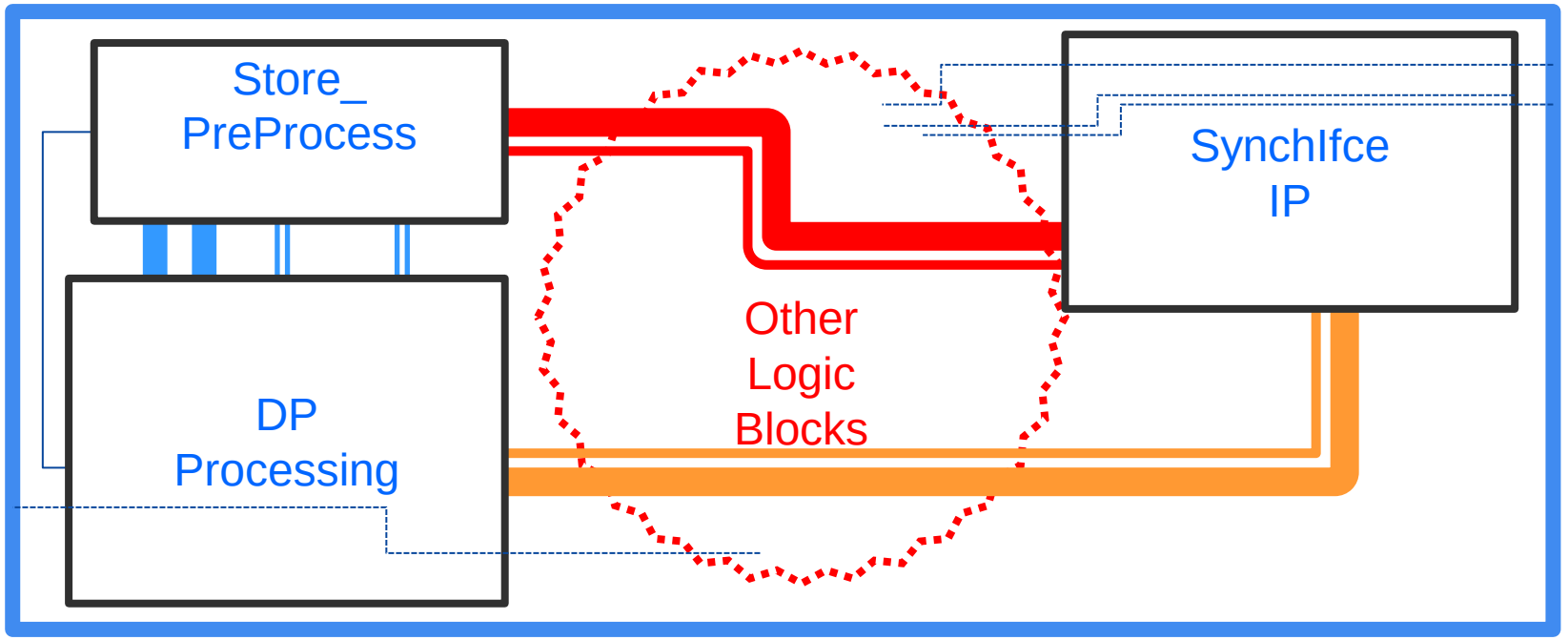
Fixing High-Congestion Designs

What Causes Routing Congestion?

- Blocks of design with wide buses on their interfaces
- Large number of high-fanout nets
- User physical constraints
- High utilization of resources
  - 85% or more utilization of logic cells, IOs, RAMs, DSPs, etc.
- Inherent congested design or blocks/modules of the design

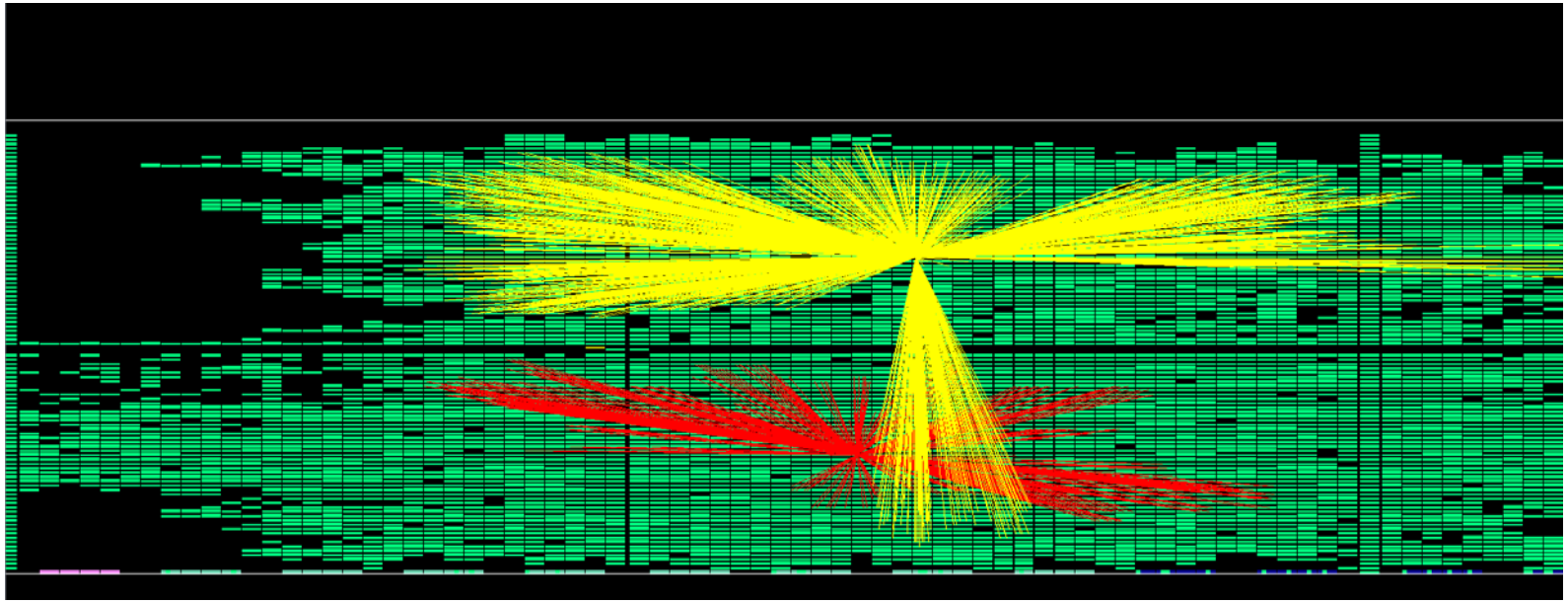
Congestion Type 1—Multiple Blocks with Wide Ports/Pins

- Consider the hierarchical design with three major blocks sharing wide bus interfaces
  - The placement of these blocks due to either IO placement/board layout or user region constraints leads to two large busses crossing the die
  - If the other logic blocks in between do not have enough porosity or are themselves congested, then the red and orange busses will fight for routing resources and may end up following a snake path, causing routing congestion



Congestion Type 2—Large Number of High-Fanout Nets

- The screenshot below shows an example of two high-fanout nets and the spread of routing resources they require
- If several similar nets exist in the design, the routing demand will cause congestion, and the router will have difficult time satisfying the timing associated with paths involving the these nets

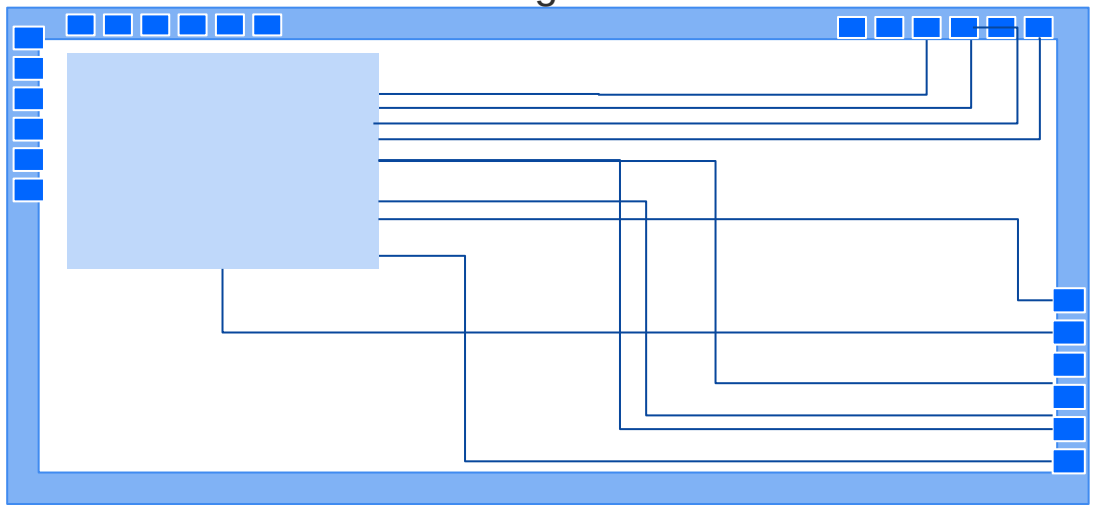


Congestion Type 3—User Constraint-Induced Congestion

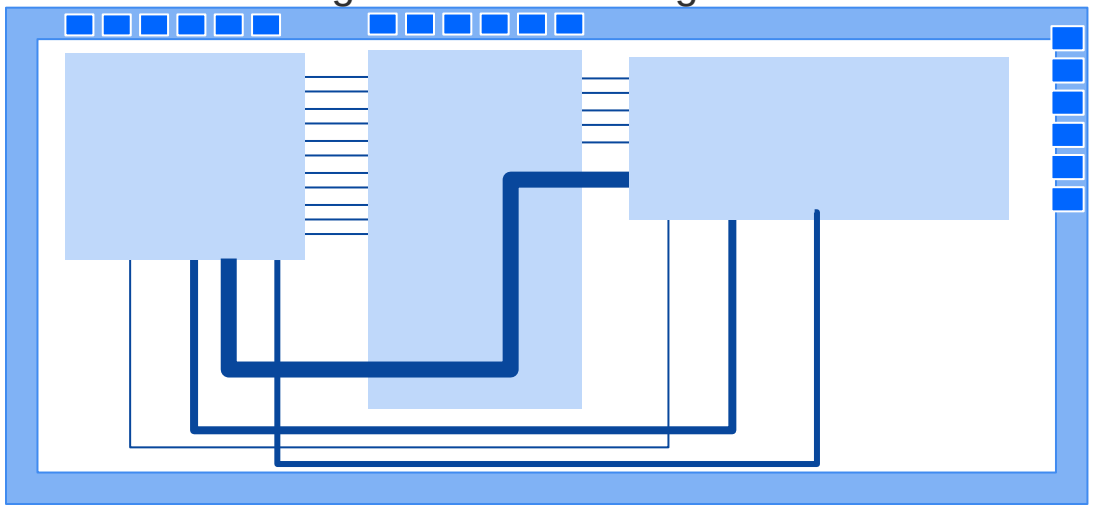
- In several designs, users create constraints for the following reasons
  - Floor-planning their design to interface with other devices or to use particular resources (SerDes, DSP, RAMs, and so on)
  - Assign IOs to certain banks and locations to satisfy some board layout considerations
  - Assign external or internal clock sources to a particular clock conditioning circuitry, PLL, or a specific global network
- If these constraints do not consider the architecture of the FPGA, its limitations, and the design timing constraints, these constraints risk creating artificial congestion
- The following slide shows examples of artificial congestion created by user constraints

Examples of User Constraint-Induced Congestion

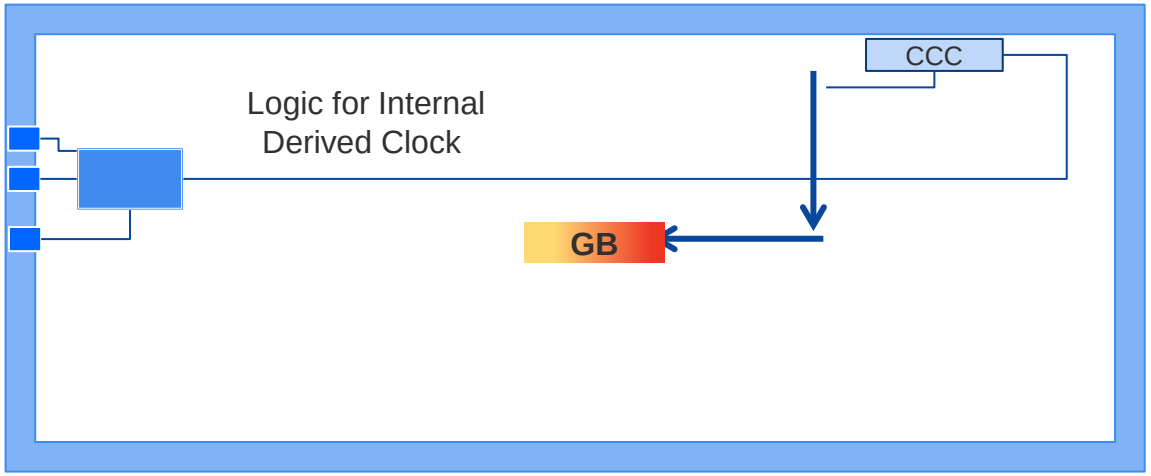
IO Assignment Forces Unnecessary
Long Routes



Blocks Floor Plan Forces Unnecessary
Long Routes and Congestion

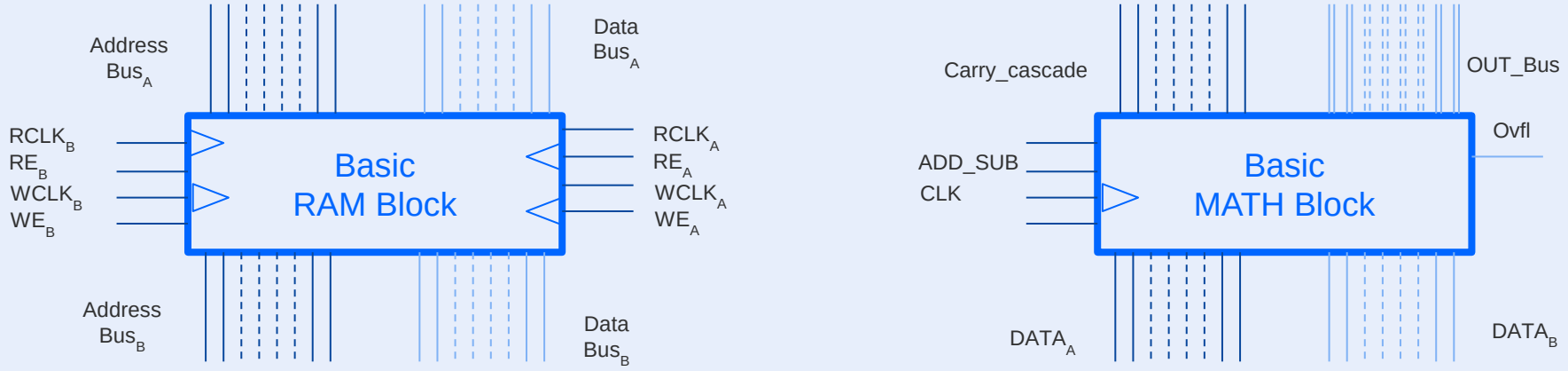


IO Assignment Forces Unnecessary
Long Routes and Large Clock Insertion Delay



Congestion Type 4—High Resource Utilization

- RTG4 architecture and the Place and Route tool have been designed and implemented to handle up to 95% resource utilization.
- Congestion associated with a design increases more than linearly with the utilization of the device resources.
  - RAM and DSP blocks demand lots of routing resources, as they have large input/output busses and are placed on particular rows of the die. If the fanout of the busses and control signals is high, the congestion around these rows will also be extremely high.



  - Logic cells are finer grain, more abundant, and spread across the die. However, If the utilization of these logic cells is high, there is a high potential for two issues to occur.
    - Congestion, particularly when several LC outputs have a high fanout.
    - The placer mishandles a small percentage of logic cells, causing additional routing congestion.

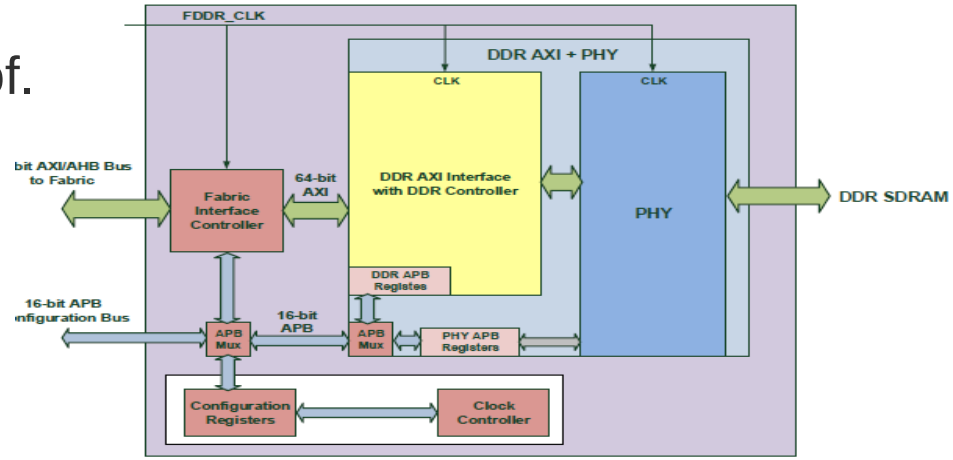
Congestion Type 4—High Resource Utilization

- High utilization of IOs
  - Regular IOs are also fine grain. However, because of the bank organization, there is a potential that accessing them may require long routes, particularly if the board designer and the FPGA designer do not communicate.
    - The board designer may impose severe IO placement to satisfy requirements such as lowering the traces on the board, signal integrity, thermal management, and solid power and ground planes.
    - FPGA designer needs to close timing and have the freedom to decide the pin-out of their chip design
  - IOs assignment has to be a co-design involving iterative activity between FPGA and board designers

Congestion Type 4—High Resource Utilization

- Use of high-speed IOs for FDDR interface
  - FDDR controller is embedded in the die at a fixed location—a constraint for the placer
  - FDDR interface must be handled carefully to avoid congestion and long routes in and out of this block.
  - High fanout or fan-in exasperates the congestion problem
  - AXI/AHB bus implementations can have large fanout if used in a complex bus structure

- Minimize fanout or separate the connections to/from FDDR
  - Good up-front logic design
  - Synplify “syn_maxfanout” attribute helps but not bullet proof.
  - Region constraints/high effort placement help greatly



Congestion Type 5—Inherent Design Congestion

- In many cases, designs are inherently congested because of the high demand of routing for logic blocks
- Typical and obvious examples of small logic with heavy demand for routing (IOs) are Mux structures, rotators, barrel shifters, and cross-bar interconnect blocks
- The following slides will focus on Mux structures

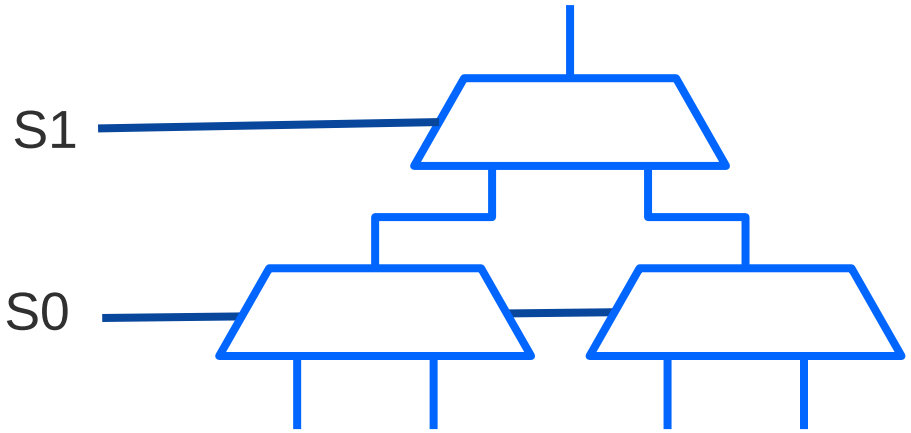
Multiplexors in Your RTL Code

- The basic form that generates Mux in your design are case statements.
  - The following illustration shows a 16-bit 12-to-1 Mux

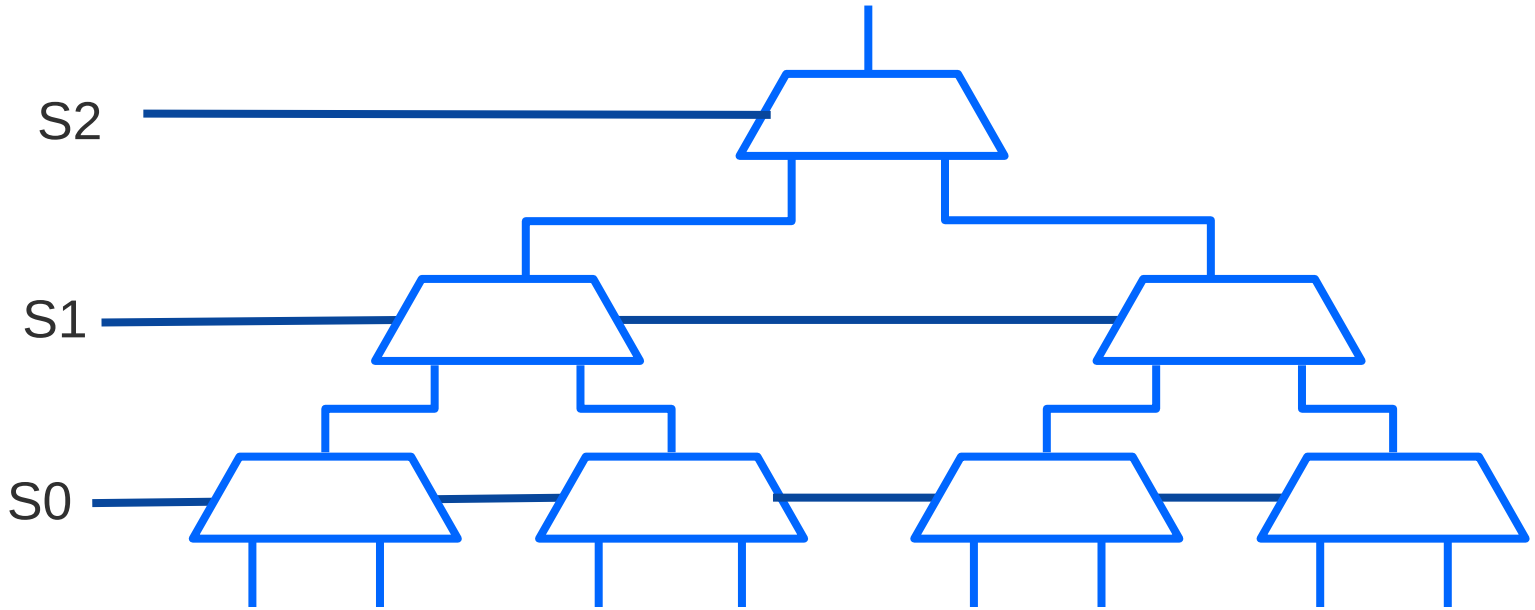
```
case Reg4 is
when "0000" => mux_out<= A;
when "0001" => mux_out <= B;
when "0010" => mux_out <= C;
when "0011” => mux_out <= D;
when "0100" => mux_out <= E;
when "0101" => mux_out <= F;
when "0110" => mux_out <= G;
when "0111" => mux_out <= H;
when "1000" => mux_out <= I;
when "1001" => mux_out <= J;
when "1010" => mux_out <= K;
when "1011" => mux_out <= L;
when others => mux_out<= “0000000000000000”;
end case;
```

Watch Out for the Mux Structures

- Mux structures cause delay penalties, as it combines both the number of logic levels and hidden high-fanout select lines.
- These structures are a typical cause of routing congestion

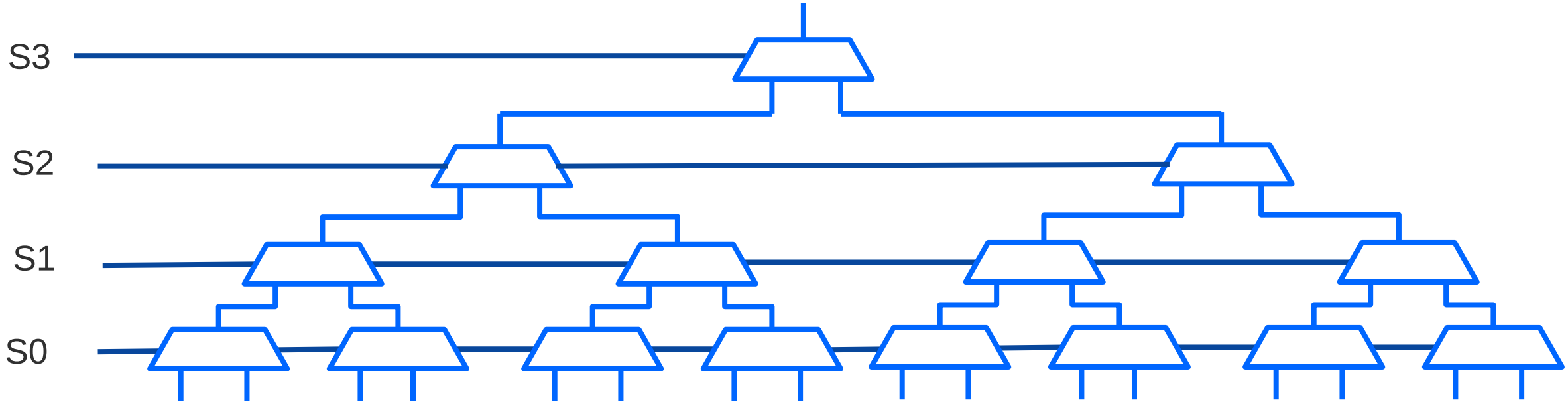


4-to-1 Mux
2 Logic Levels
S0 Fanout = 2



8-to-1 Mux
3 Logic Levels
S0 Fanout = 4

Mux Structure and Select Lines Fanout (continued)

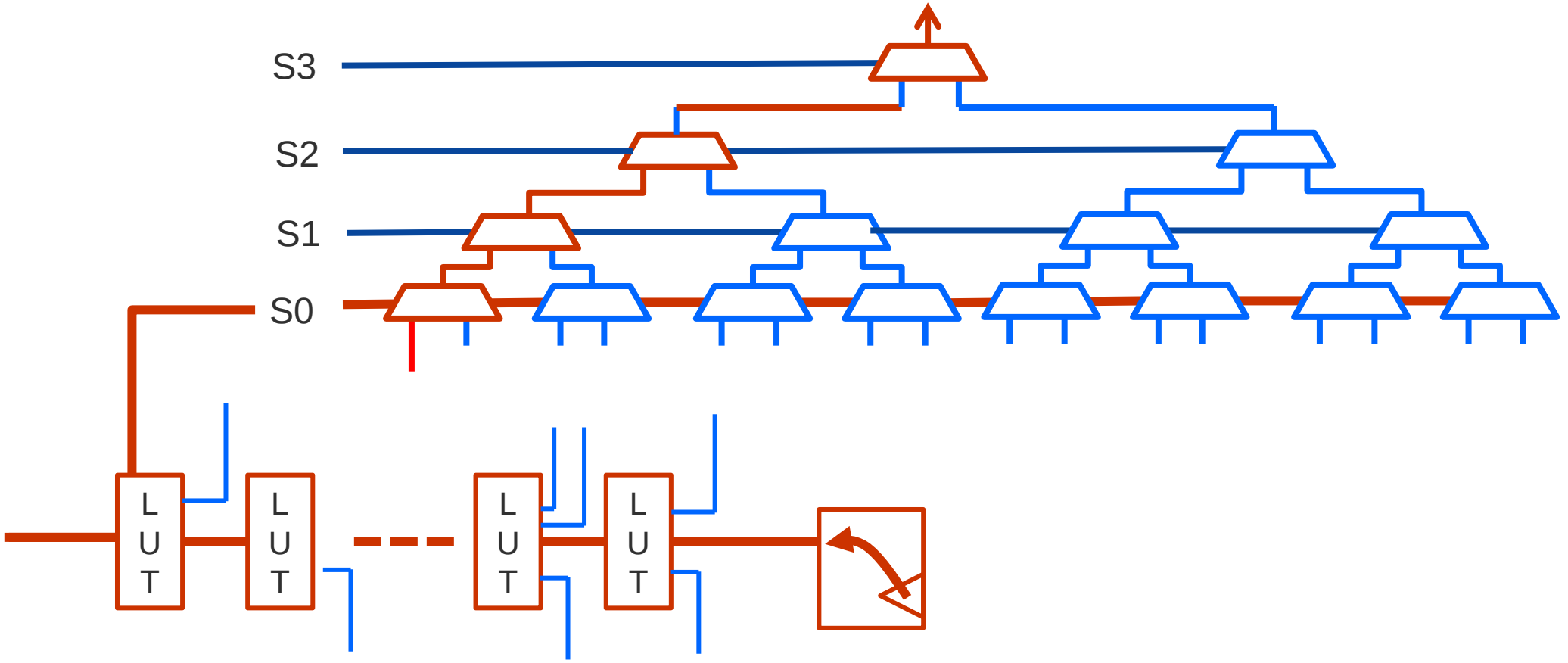


16-to-1 Mux
4 Logic Levels
S0 Fanout = 8

Mux Structure and Select Lines Fanout

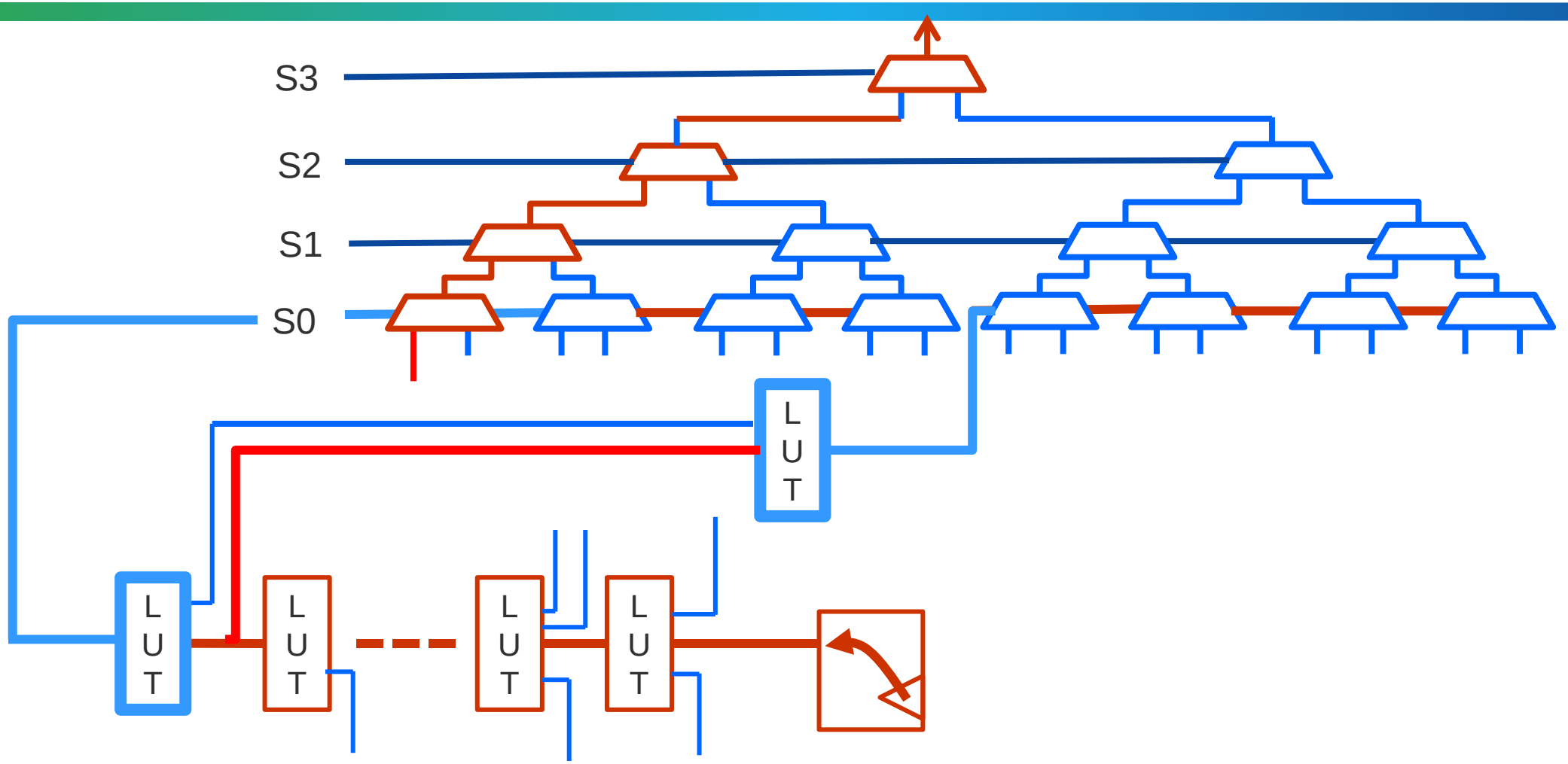
Number of Inputs	Input Bit Width	Select0 Fanout	Select1 Fanout	Select 2 Fanout	Number of Logic Levels
2	1	1	-	-	1
8	1	4	2	1	3
16	1	8	4	2	4
2^N	1	2^{N-1}	2^{N-2}	2^{N-3}	N
8	M	M*4	M*2	M*1	3
16	M	M*8	M*4	M*2	4
2^N	M	$M*2^{N-1}$	$M*2^{N-2}$	$M*2^{N-2}$	N

Things Can Get Nasty with Mux Structures
Case 1: Select Line Deep Logic



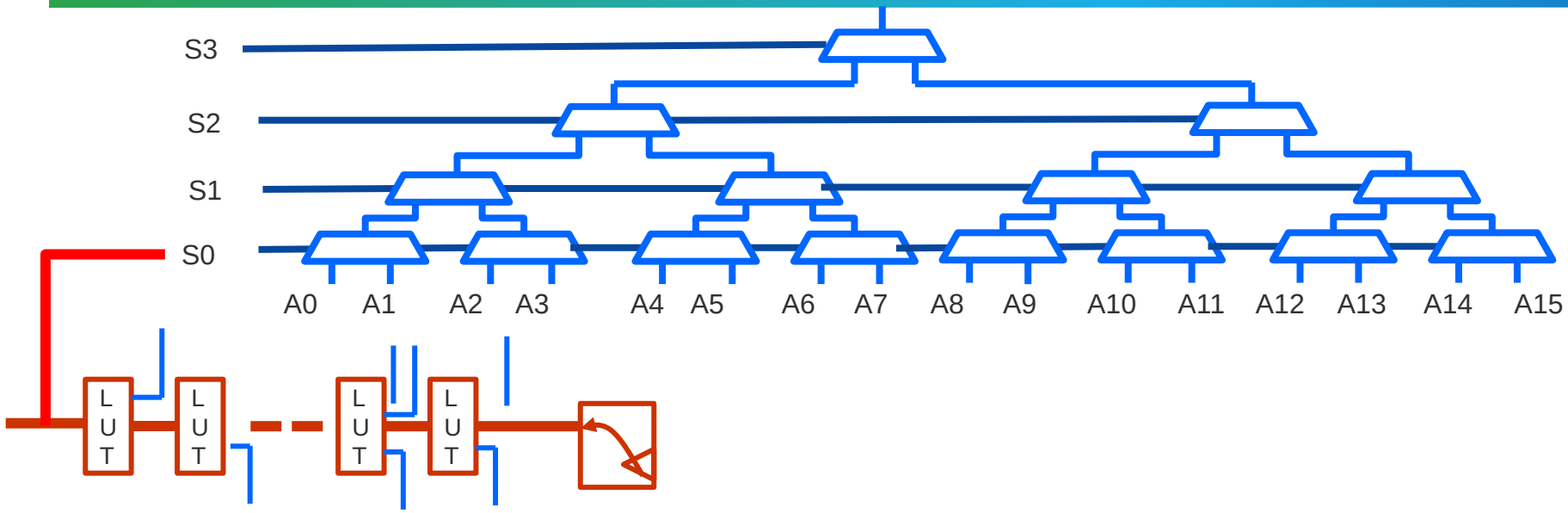
- This design is suffering from too many logic levels and the high-fanout select S0
- S0 toggle ripples through the Mux structure

Incorrect Path Fix: Logic Replication of the Select0 Driver



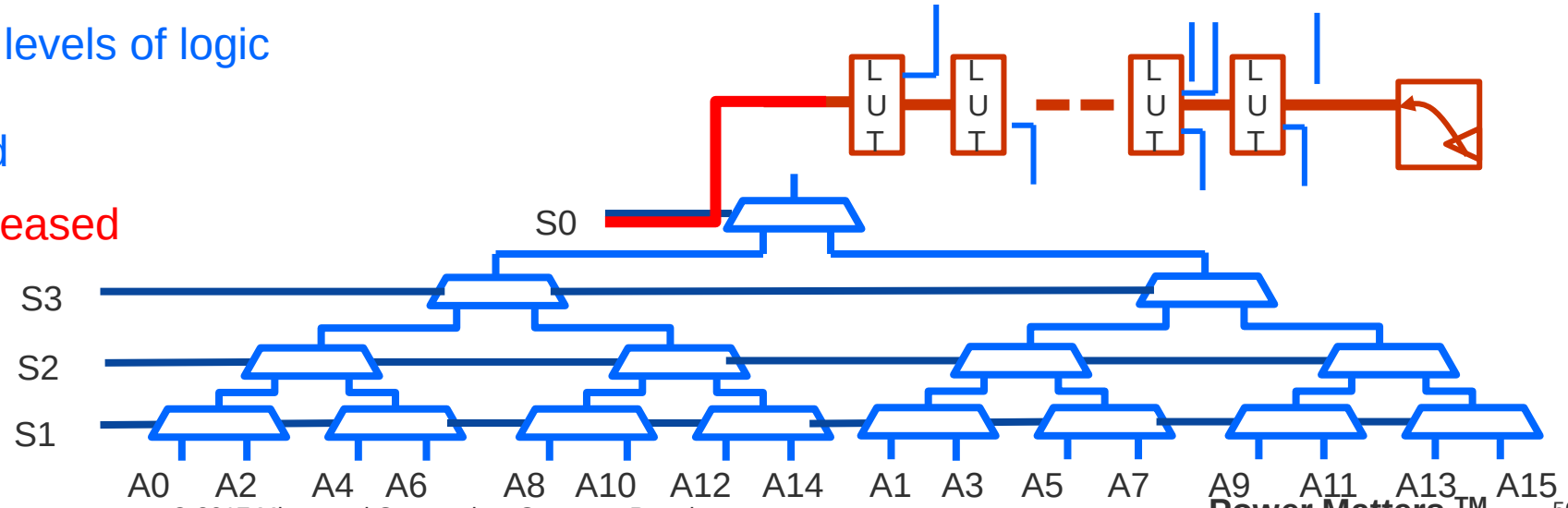
- S0 fanout reduced to half
- S0 toggle still ripples through the Mux structure

Better Solution: Reordering Select Lines, Swapping Inputs

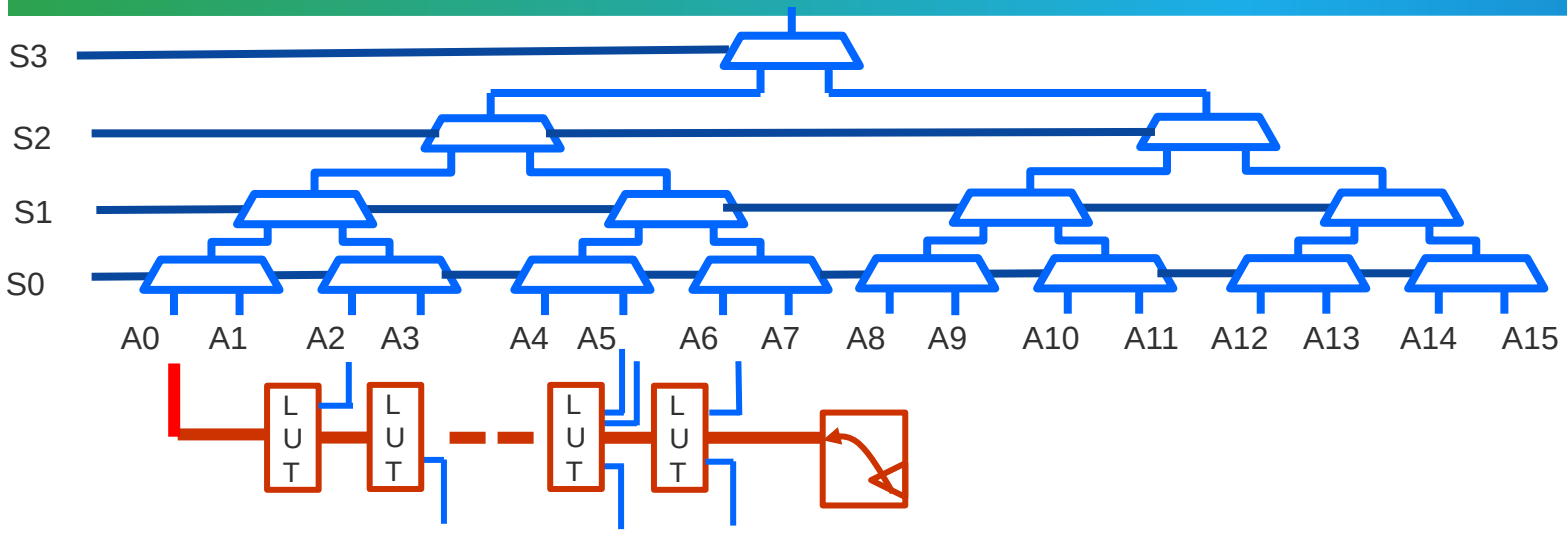


\+ S0 path reduced by three levels of logic
S0 fanout reduced to one
S0 toggling ripple reduced

– S1, S2, and S3 fanout increased

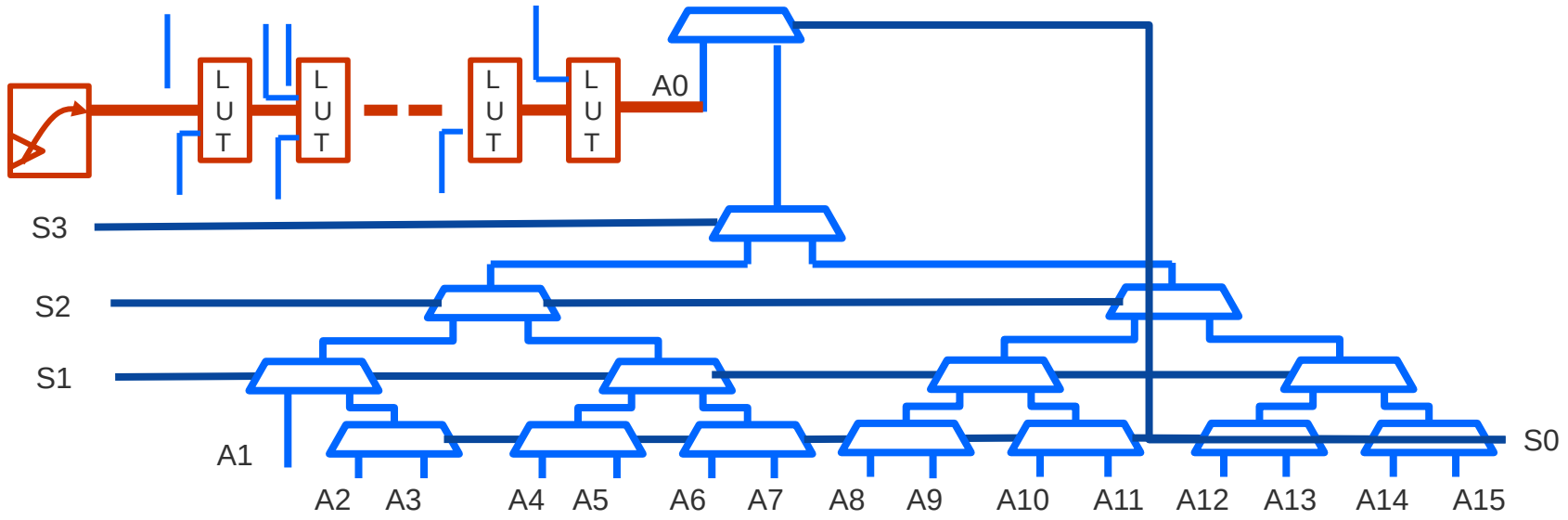


Things Can Get Nasty with Mux Structures
Case 2: Mux Input Deep Logic, Say A0

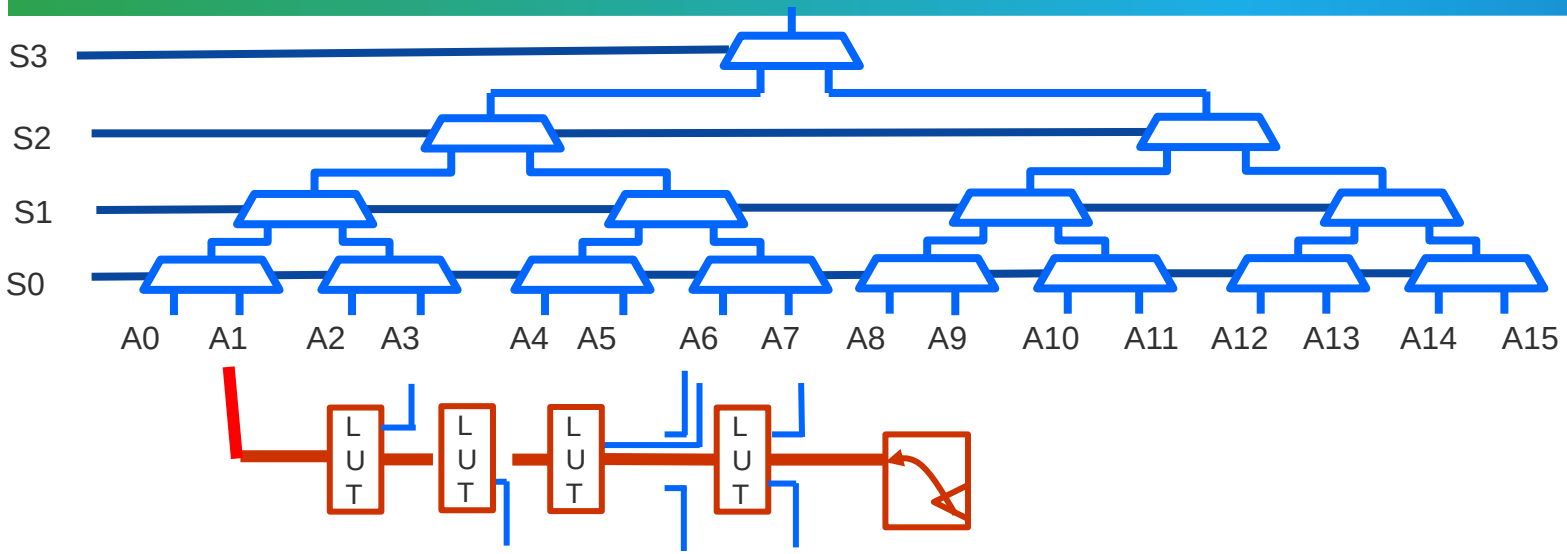


A0 path reduced by three logic levels
No Mux added
A0 toggling ripple reduced

Ai (i = 2, 15) will see one additional level of logic

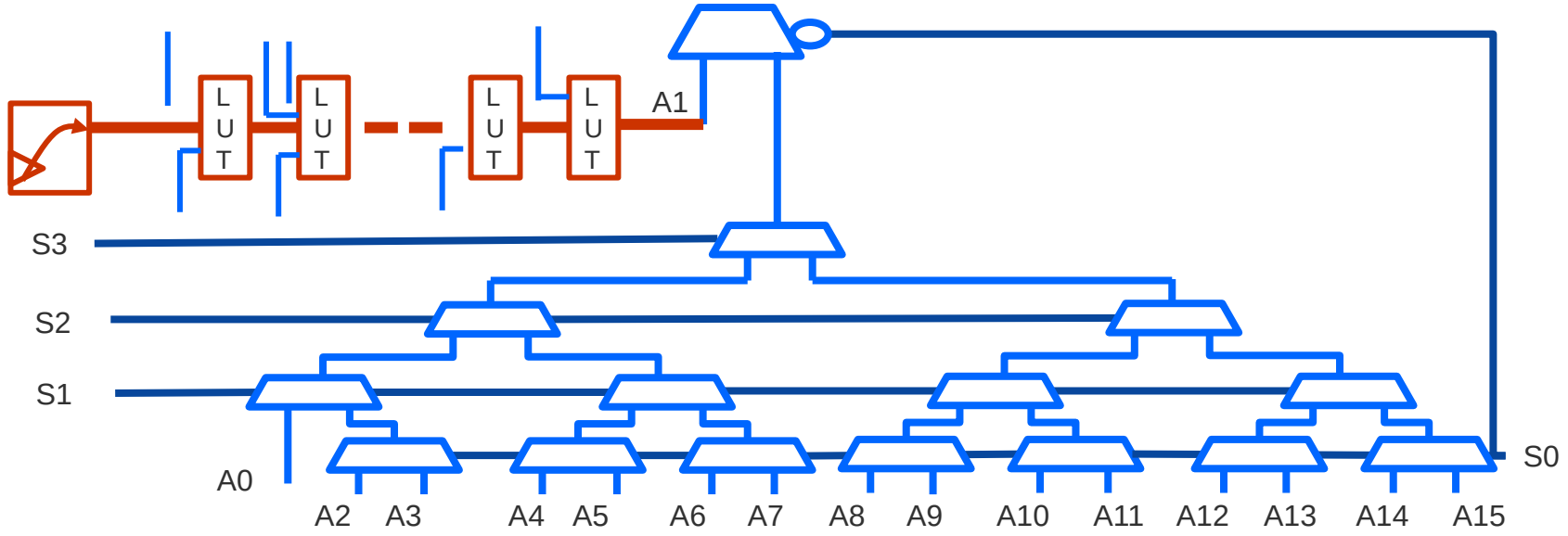


Things Can Get Nasty with Mux Structures
Case 3: Mux Input Deep Logic, What If It is A1?



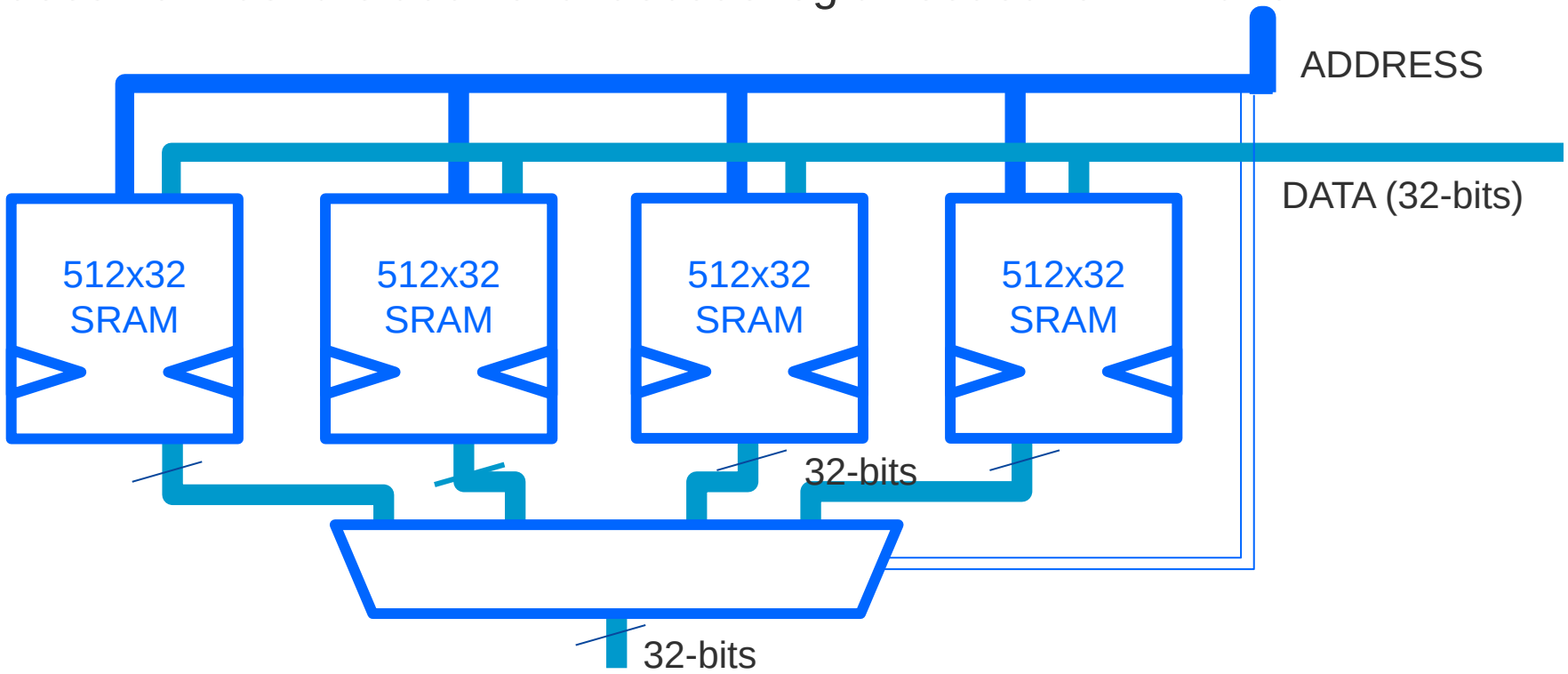
\+ A1 path reduced by three logic levels
No Mux added
A1 toggling ripple reduced

– Ai (i = 2 .. 15) will see one additional level of logic



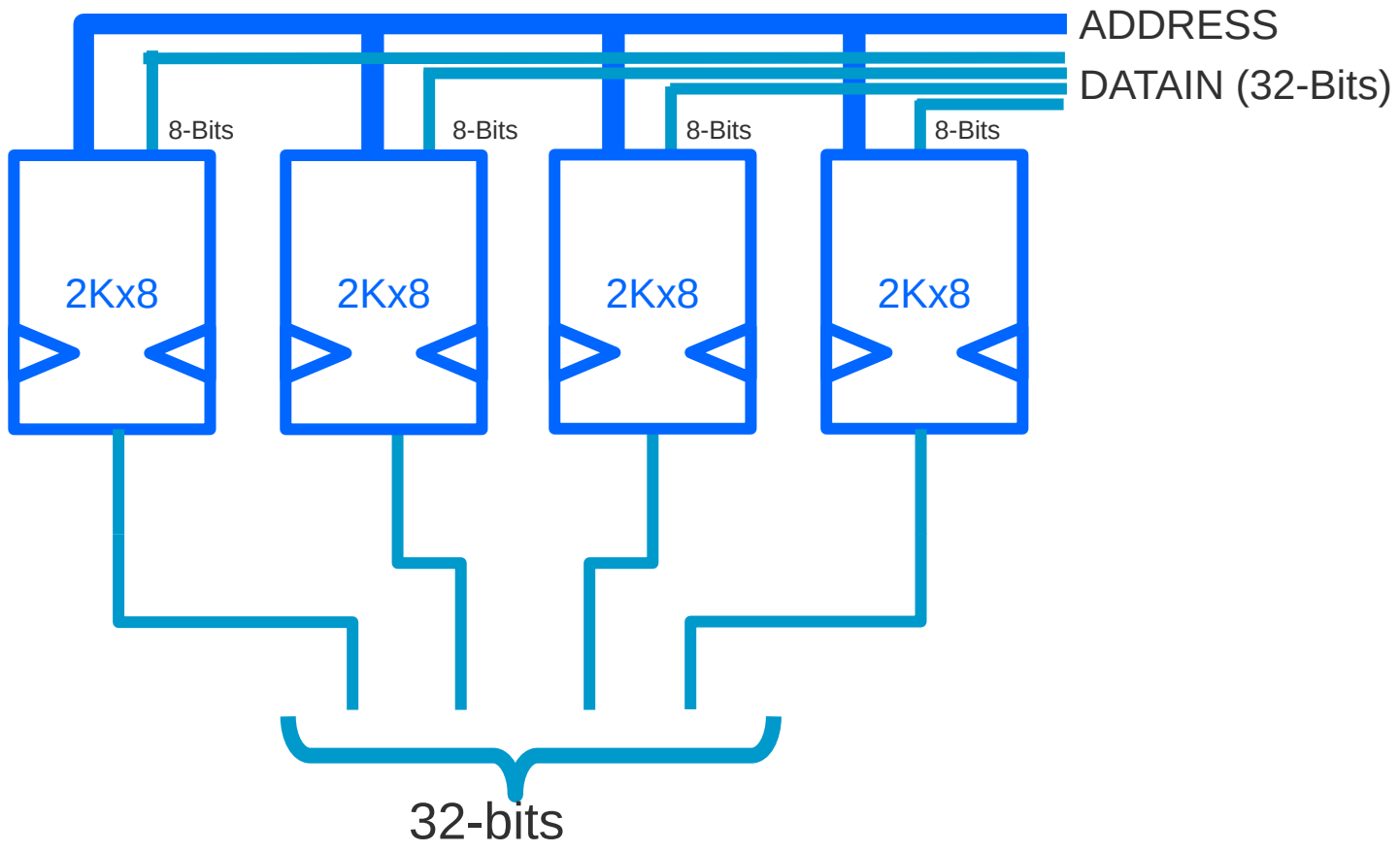
Hidden Multiplexors in Your RTL Code!

- RTL code intends to create a 2Kx32 RAM block
- Synthesis may infer an implementation that uses 512x32 basic SRAM blocks, as shown in the following illustration
  - Notice 4-to-1 Mux for 32-bit busses—remember the S0 fanout
  - Figure does not illustrate additional decode logic needed for WE and RE



How to Fix Hidden Multiplexors in Your RTL Code

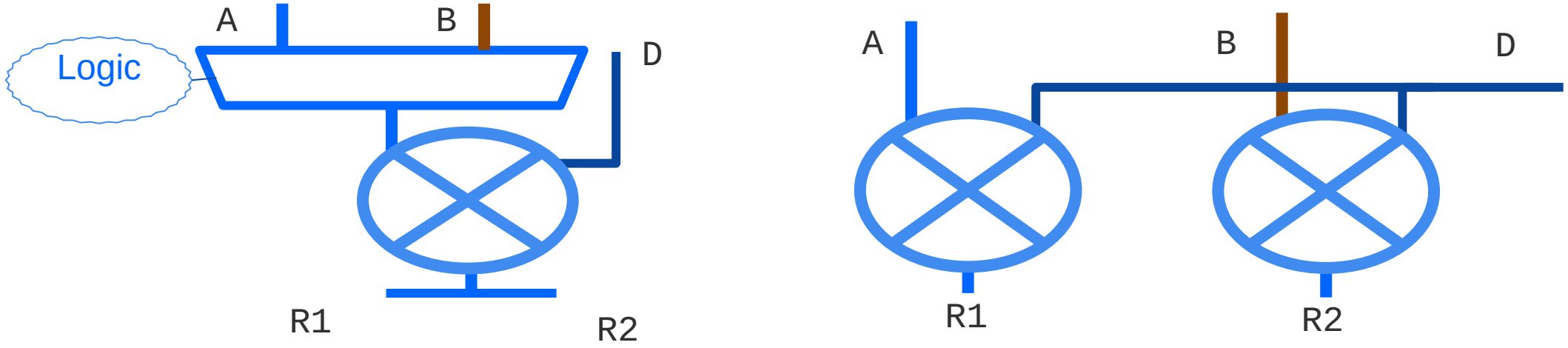
- Choosing the right aspect ratio removes unnecessary multiplexing and decode logic
- The outputs of all 2Kx8 RAMs are concatenated to form the 32-bit



Hidden Multiplexors in Your Synthesis Options

- If Resource Sharing is ON, synthesis reduces the number of DSP blocks to the minimum. While this reduces the utilization of these resources, it introduces large multiplexors in the ingress path of these embedded DSP/arithmetic blocks.

```
Example of RTL Code:
If (BooleanExpression) then R1 <= A + D; else R2 <= D + B;
```



- Unless you are running out of DSP resources, set Resource Sharing to OFF

State Machine Coding Style and State Encoding Options

- Recommendations
  - Enumerate all the possible states and use one-hot encoding
  - If you can't enumerate all the possible states, use HDL clauses
    - When Others => Next_state <= "a particular state of your choice" – – VHDL
    - "default" : Next_state = ""a particular state of your choice"; /* Verilog */
  - Do not use "safe" encoding
    - All registers in RTG4 are TMRs and SEU immune
    - Synplify creates an overhead logic for recovery that is not necessary as it will exacerbate the timing
    - Synplify may create additional reset signals that will be mapped to regular routing (not radiation-hardened)

Dealing With High and Medium Fanout Data and Control Nets

Patent Application in Progress

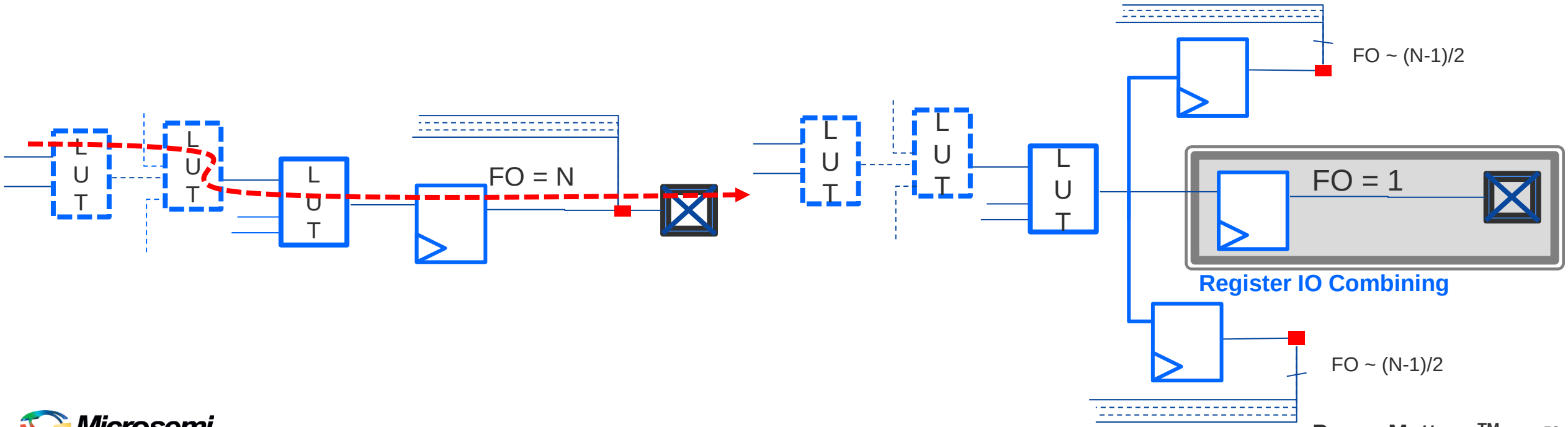
Outline

- IO timing-aware manual unbalanced logic replication
- IO Placement-Aware manual unbalanced logic replication
- Register-2-Register-aware replication

Clock-2-Out or IO Timing-Aware Adaptive Logic Replication: Simple Case

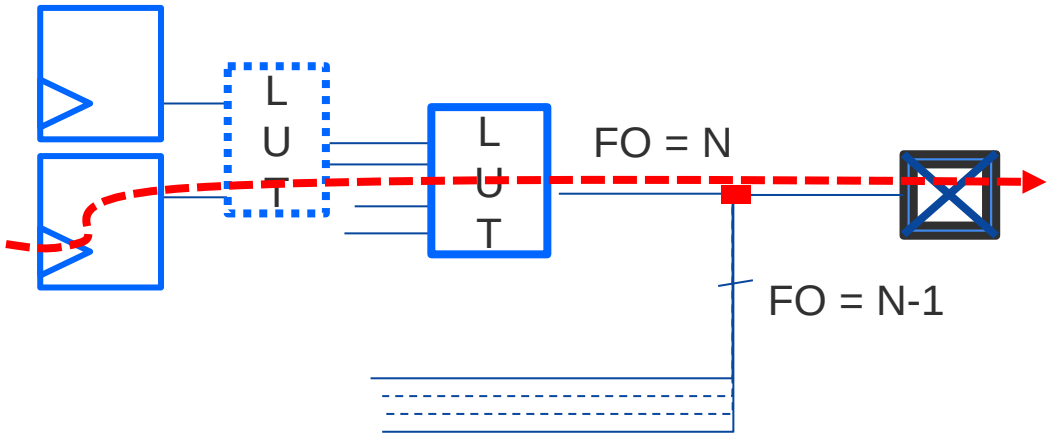
- High load on driver because of high fanout
- Large Clock-2-Out delay due to higher capacitance on register output

- Register or IO driver now has a fanout of 1
- Better Clock-2-Out delay with potential register combining
- Replicate register carries the N-1 load

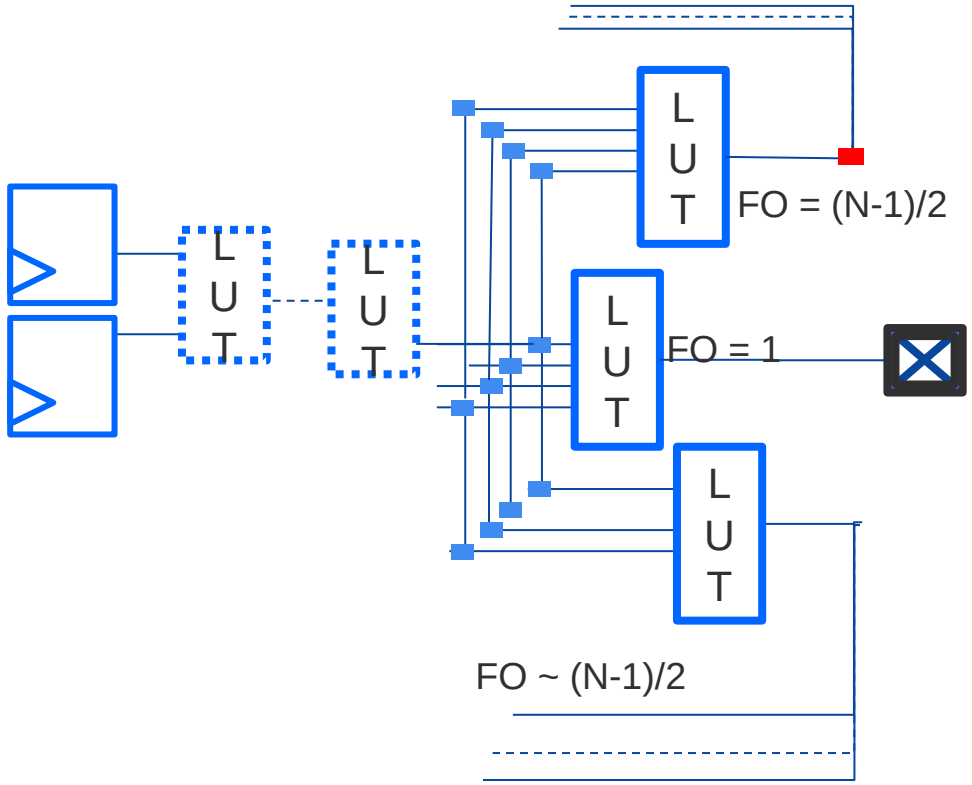


IO Timing-Aware Manual RTL Fanout Control

- High load on driver LUT because of high fanout
- Large Clock-2-Out delay or IO due to higher capacitance on LUT output

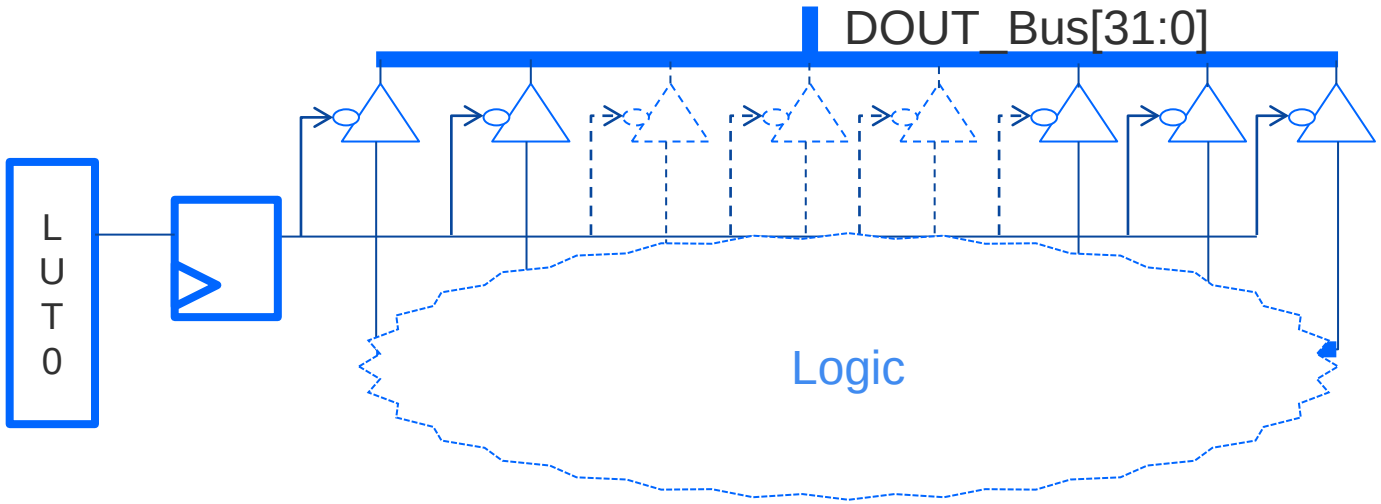


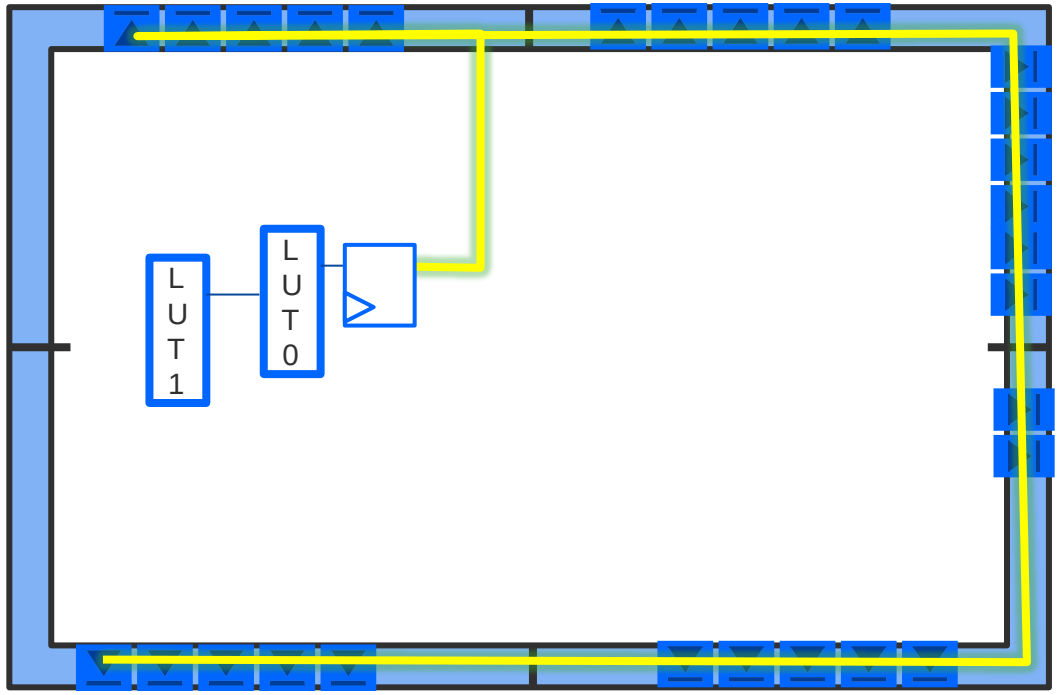
- LUT driving the IO now has a fanout of 1
- Better Clock2Out or IO delay
- Replicate LUT carries a load of (N – 1)/2



Clock-2-Out or IO Timing-Aware Adaptive Logic Replication: Complex Case

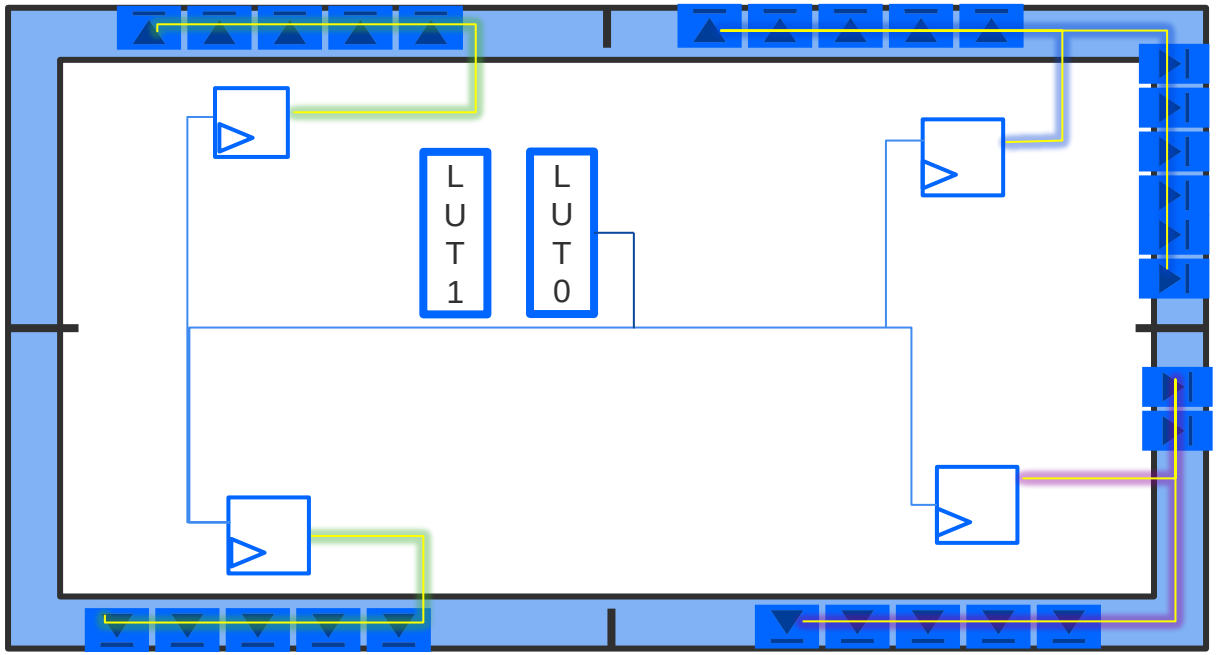
- Consider the RTL that generates the OE for a 32-bit output bus DOUT_Bus[31:0]
  - The RTL appearance does not reveal a situation of delay penalty associated with the OE signal, as the designer has registered it and the fanout is only 32
  - However, if the 32-bit pads are placed on different IO banks, the routing of the output of the register (OE) will have high delay penalty no matter how the placer optimizes the placement of the driving OE register





IO Placement-Aware Logic Replication

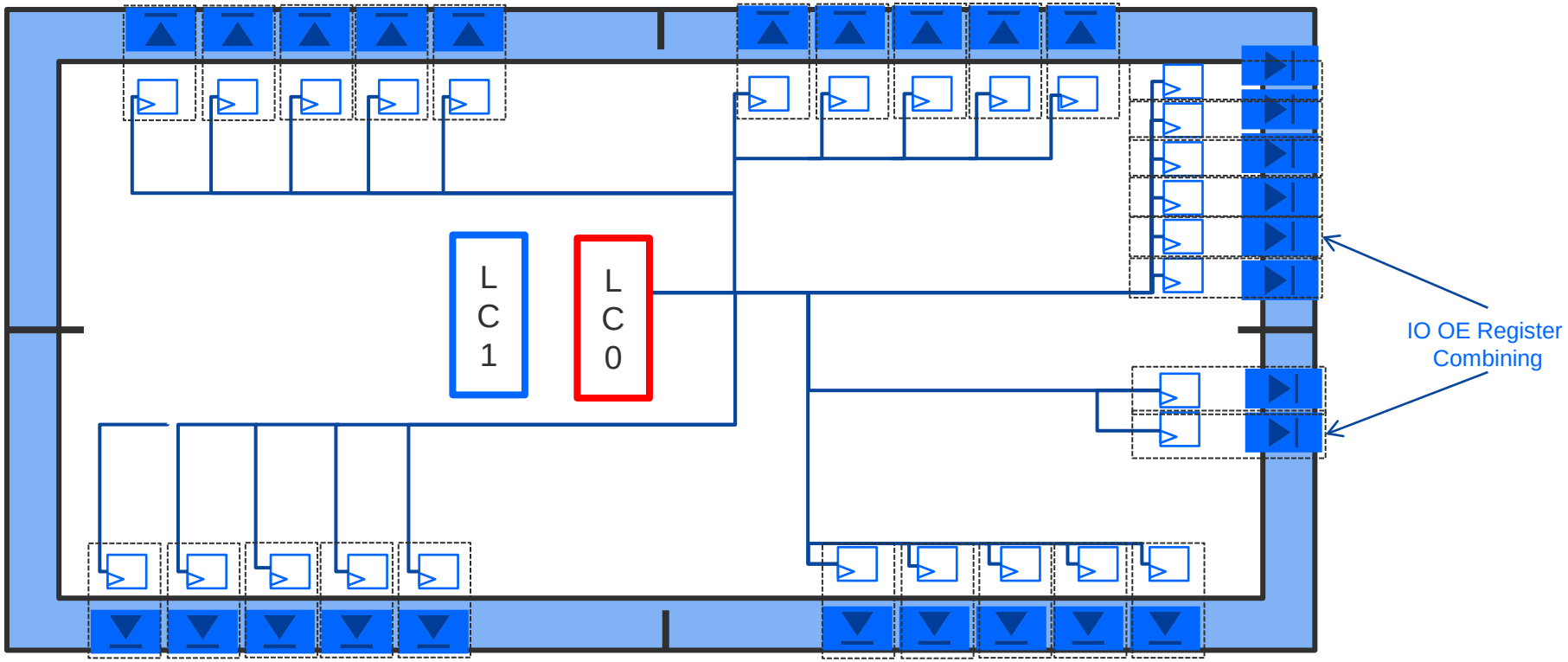
- Designers need to identify these situations and modify the RTL code(*) to map the OE to drive the slices of the DOUT_Bus as the load of each replicated OE register
- The following solution helps, but is tedious for the user because every minor change in the pin-out will cause a complete review of the RTL code



(*) For example, setting the max_fanout to 4 does not guarantee that synthesis will be selective for the sink load and distinguish what to drive with each replicated register

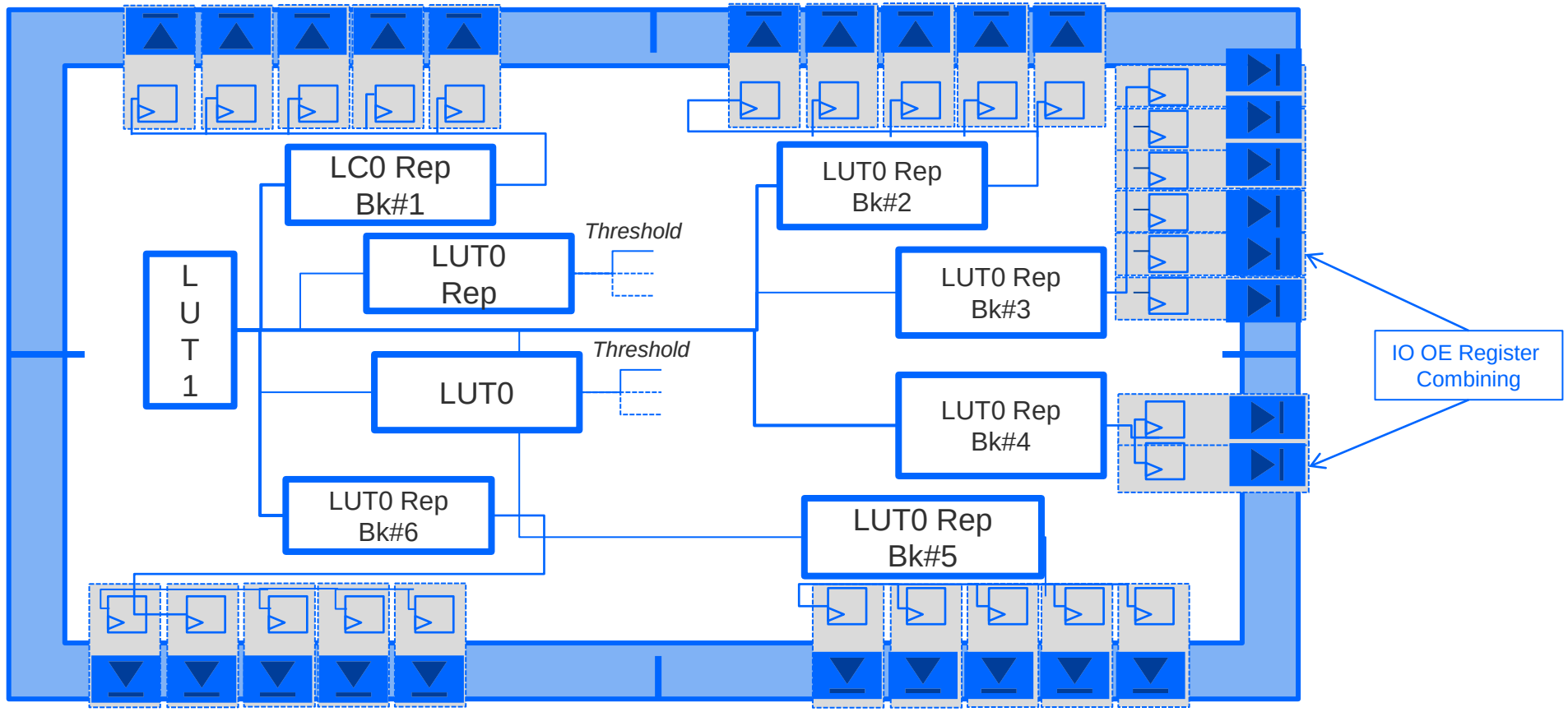
IO Placement-Aware Logic Replication

- User should set the max_fanout to 1 on the OE and allow synthesis to infer 32 registers (one for each bit of DOUT_Bus). Additionally, the user could set the IO Register Combining to ON, as the IO tiles include a free register for the OE
- The upside is that the Clock-2-Out timing is now optimal
- The drawback of this solution is that the high fanout is transferred to the LUT, LUT0, driving the OE register D inputs and has a similar challenge (solution in next slide)



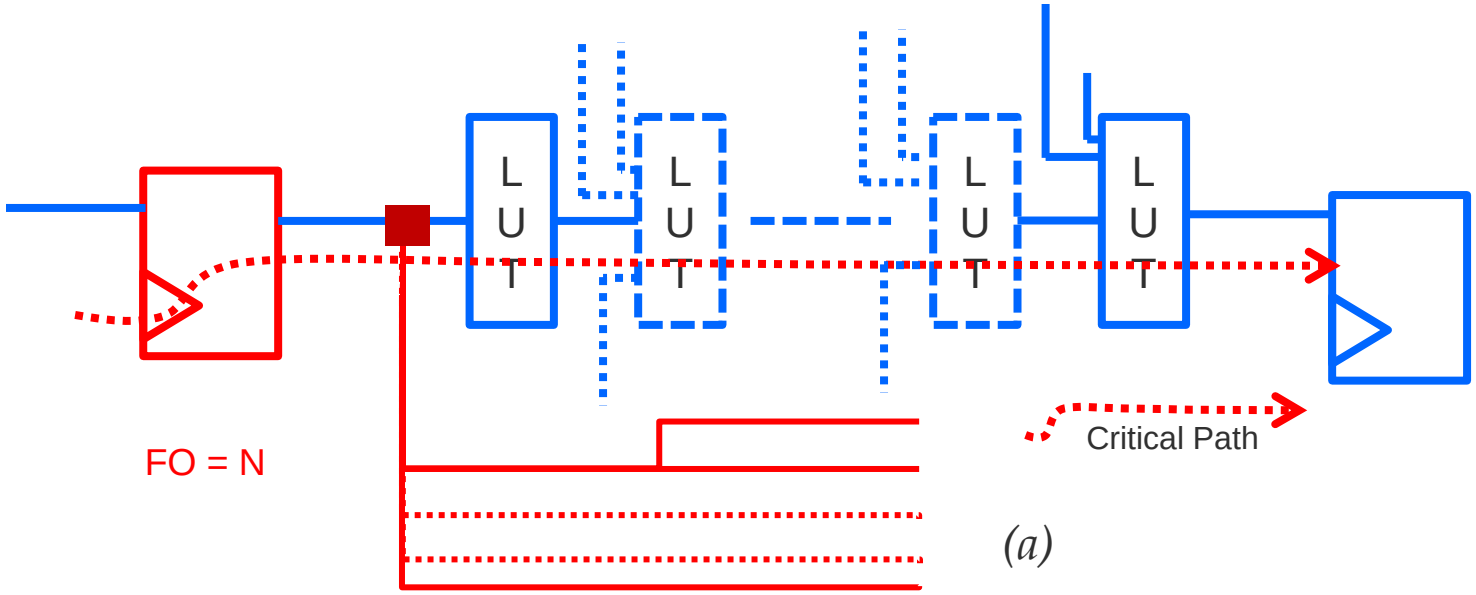
Multi-Layer Adaptive Logic Replication

- Second layer of logic replication—LUT0
- Replicating LUT0 and dedicate a replicate to a quadrant will allow the P&R to
- Replicating the LUT 4 times will relieve its fanout output, but will increase its input fanout by 3



Register-to-Register Critical Timing and Fanout Control

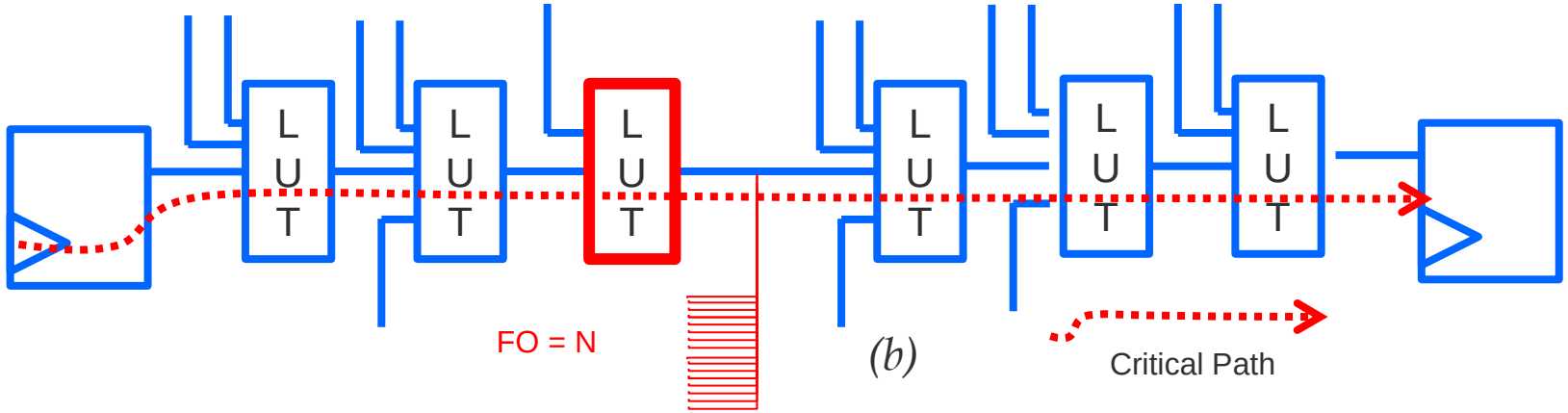
- Case 1: Register driving high-fanout net



- Criticality at the start of the path
  - Delay penalty carried by all the paths in the outgoing logic cone
  - Has to be addressed to relieve all these paths
- Register replication is tedious but feasible at the RTL code

Register-to-Register Critical Timing and Fanout Control

- Case 2: Combinatorial high-fanout net inside deep logic

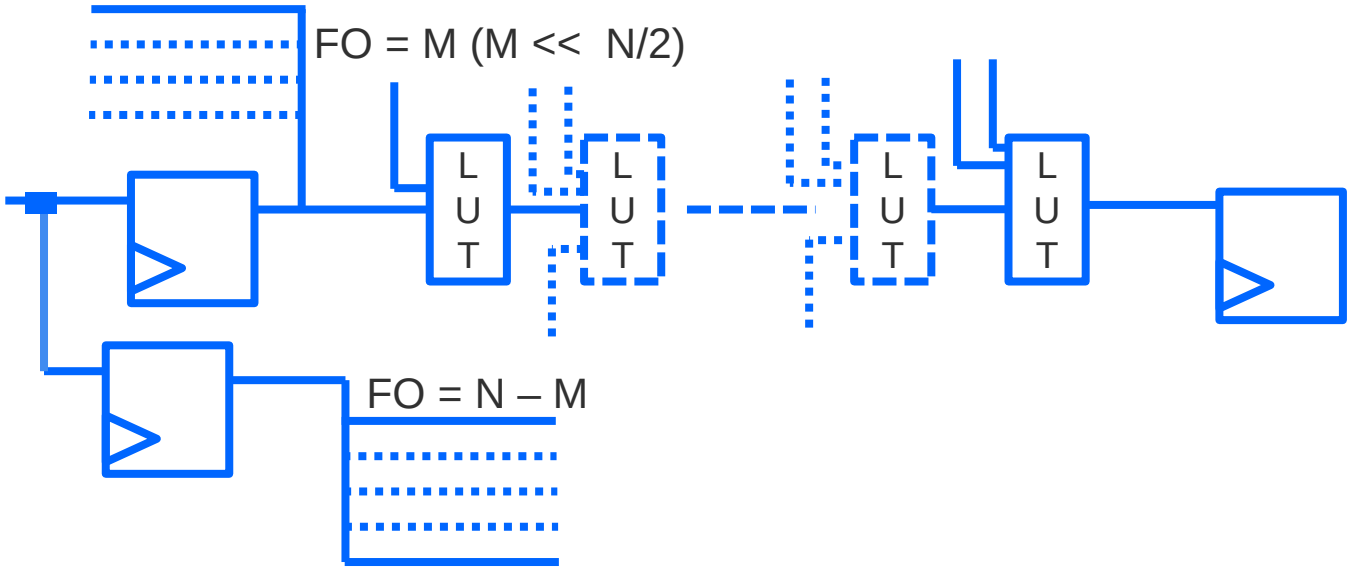


- All paths starting with high-fanout cell will suffer
  - Penalty due to high fanout and capacitance associated with routing
  - Number of logic levels up to the cell driving the high-fanout net

- Difficult to predict at the RTL code, but once identified (post-compile), user can proceed with explicit replication
  - See possible solutions in next slide

Case 1: Explicit Driver Replication to Relieve Critical Paths

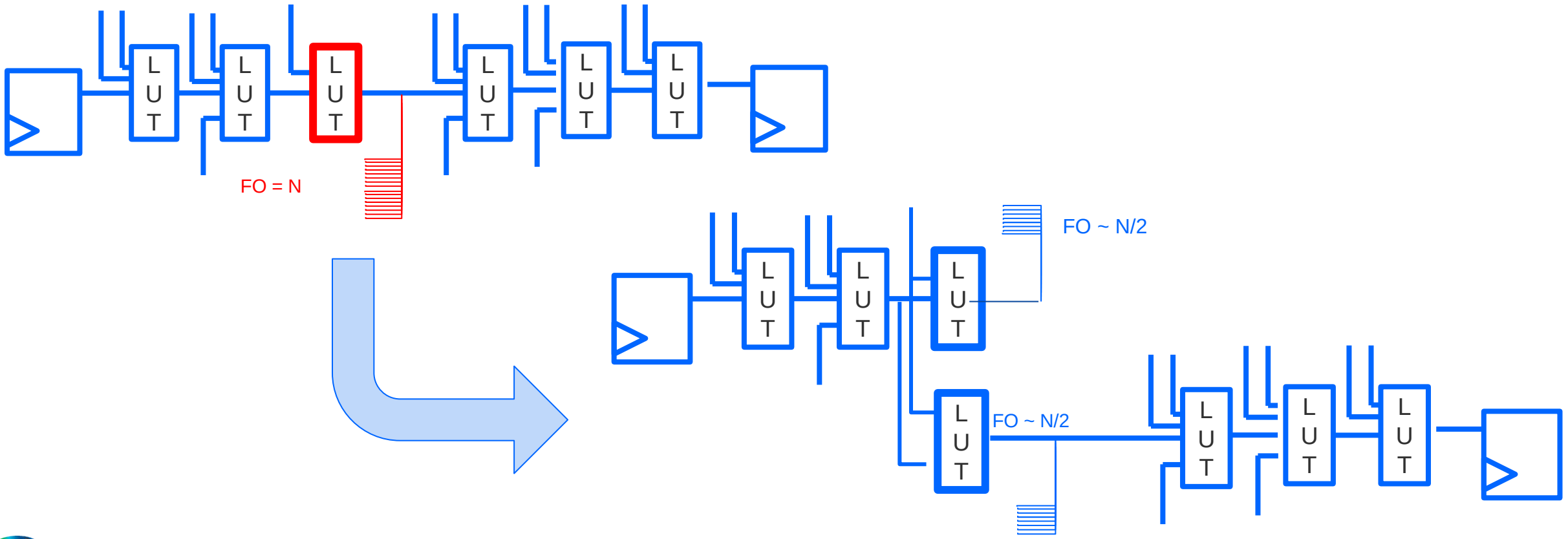
- When manually replicating the register, make sure you have an understanding of how to allocate the load to the copies/replications
  - Paths with a high number of logic levels or high-fanout nets should be allocated to the replication with lower fanout



  - May consider additional replications if floor plan constraints require logic to be pulled to different sides of the die

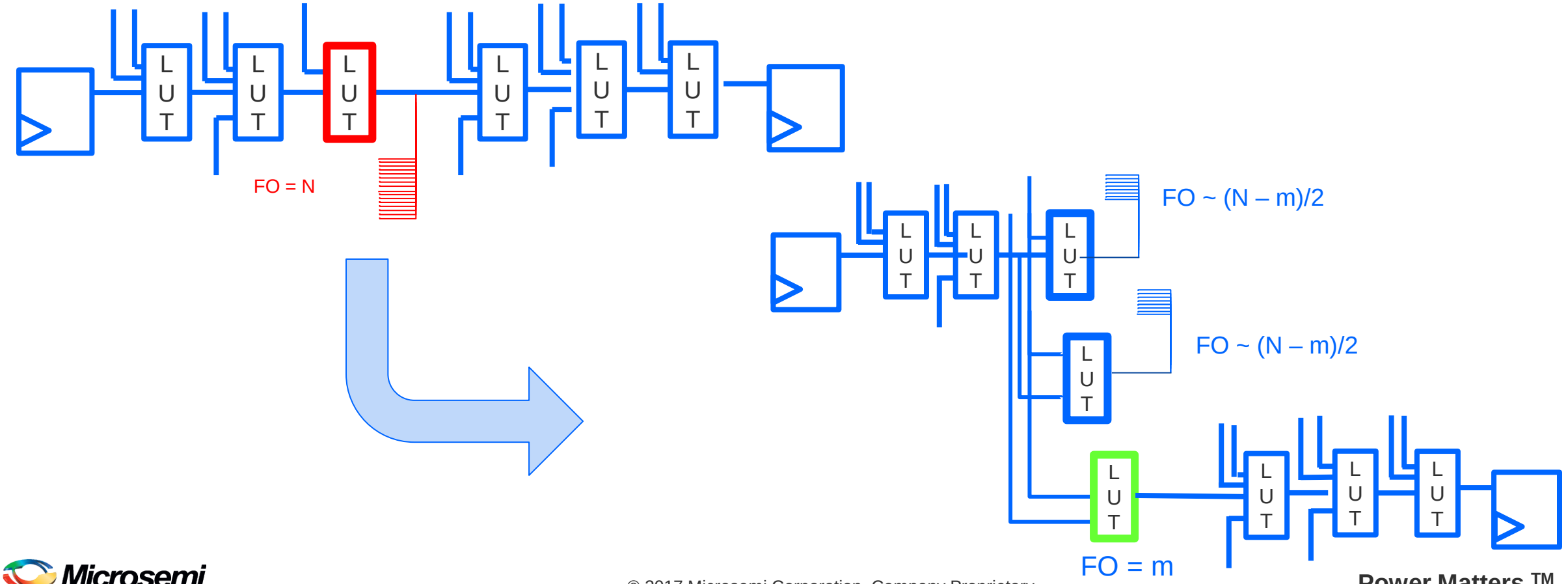
Case 2: High-Fanout Net Inside Deep Logic

- Balanced replication—automated approach
  - Use syn_max_fanout attribute in RTL code
    - Example syn_max_fanout = N/2 will lead to a load, as shown in the following illustration



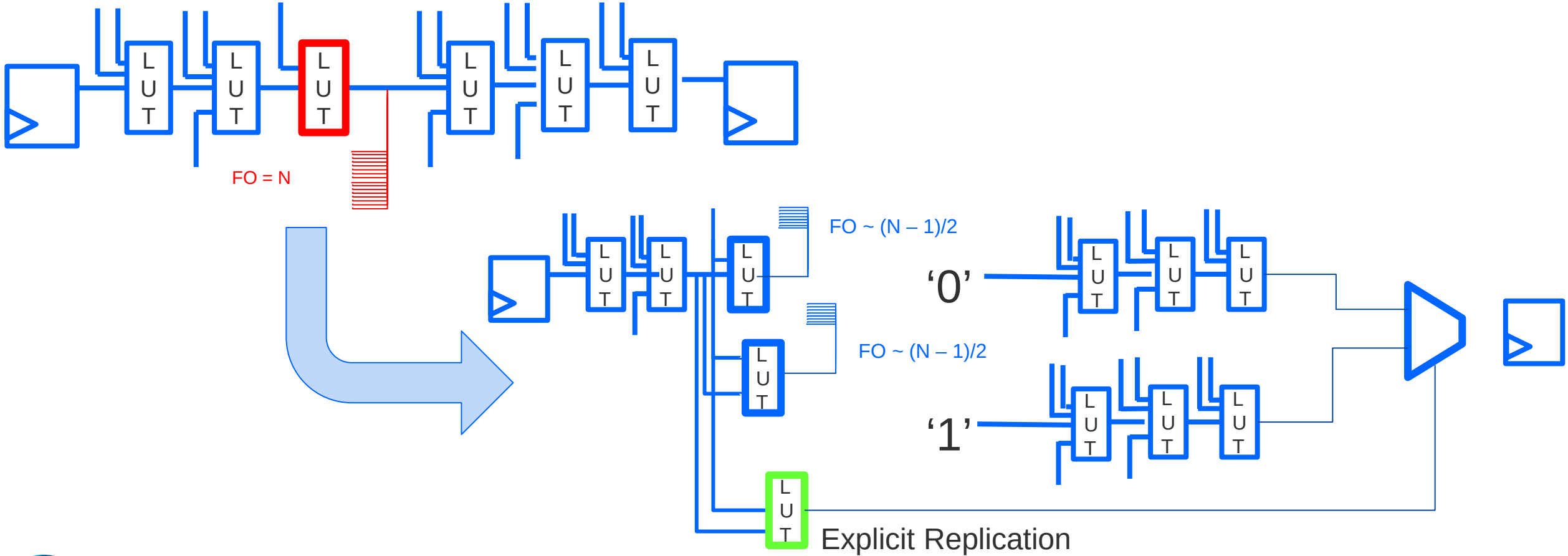
Case 2: High-Fanout Net Inside Deep Logic

- Unbalanced replication—automated approach
  - Use of syn_max_fanout attribute in RTL code, but the set value (N – m) is such that one branch will have much less load than other branches



Case 2: High-Fanout Net Inside Deep Logic

- Unbalanced and explicit replication—manual approach
  - Use of syn_max_fanout attribute
  - Modify RTL code to implement “decide as late as possible” technique



Improving DSP Design Performance

Identify Potential Performance Hurdles

- When design utilizes hard MACCs, pay attention to the following connections that can impact the performance
  - Connections between RAM blocks and MACCs
  - Connections from fabric to multiple MACCs such as, resets, enable data buses, and so on
  - Datapath connections between the MACC rows

- The longer the chain of MACCs, the more attention is required
  - Shorter chains (<10 MACCs) are less likely to cause performance degradation when it comes to fabric connections. The place and route tools perform a decent job (however, it never hurts to double check).

Pipeline the Design

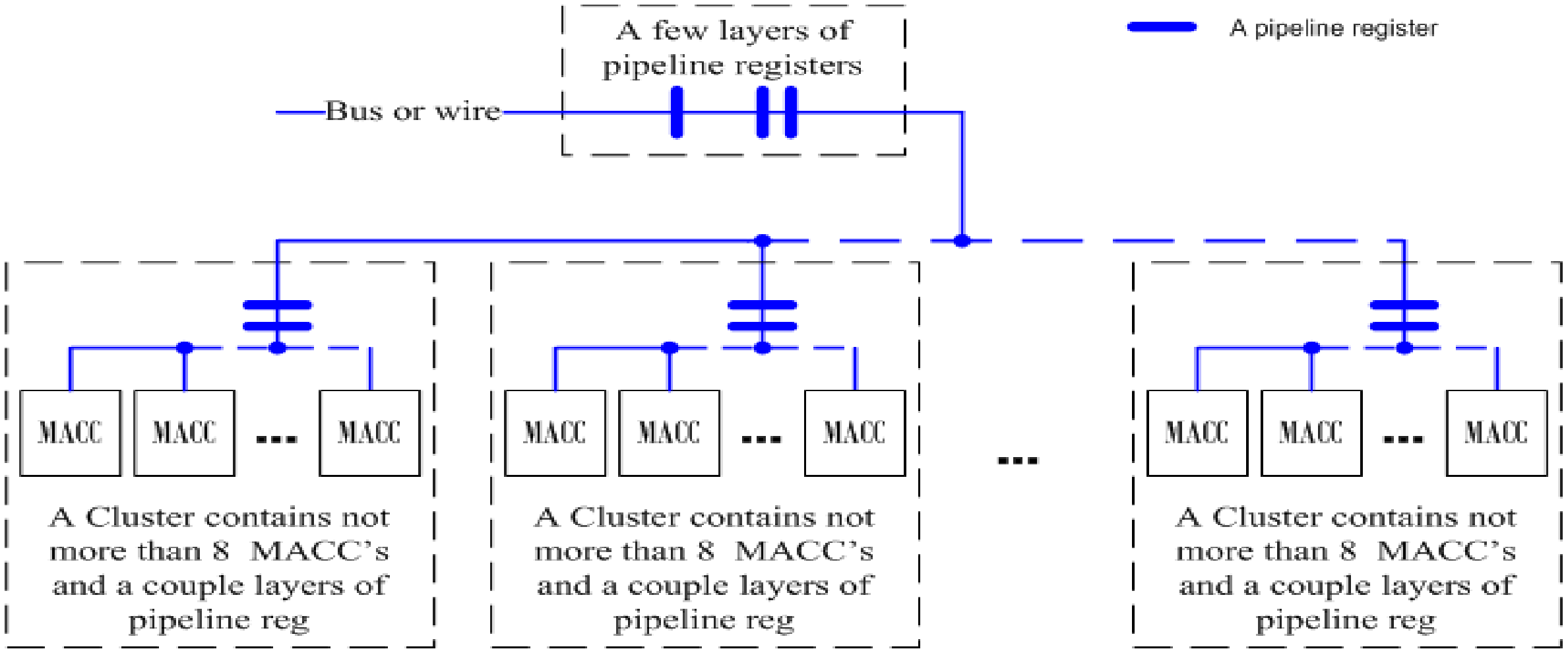
- Extensive pipelining improves the performance
  - Make sure all the internal MACC registers that receive or transmit data are enabled (this is possible in majority of cases, no matter the application)
  - Pipeline all the connections between MACCs and other components (RAM and fabric)
- In some cases, too many pipes can be counter-productive
  - Having room to play with pipeline layers and their locations is the best

Long Input Lines and Buses

- A chain of MACCs often requires one or more signals or buses to come to every MACC
  - Such signal fanout can be moderate, but the routing can be really long, as the MACCs are distributed edge-to-edge on a die
  - Check the routing of such high-fanout signals/nets to find out if the MACC placement is appropriate or needs modification
  - In any case, setting a strict “max_delay” timing constraint helps avoid these nets to have high delay penalty
- Separate the MACC chain into clusters with pipelines assigned to every cluster
  - The cluster structure is shown on the next slide
  - The cluster contains eight MACCs or less. The number eight has no scientific significance, it was just what worked for CoreFIR design
  - Protect the following pipeline registers associated with the clusters from removing by SynplifyPro:

```
attribute syn_preserve : boolean;
attribute syn_preserve of pipe_reg : signal is true;
```

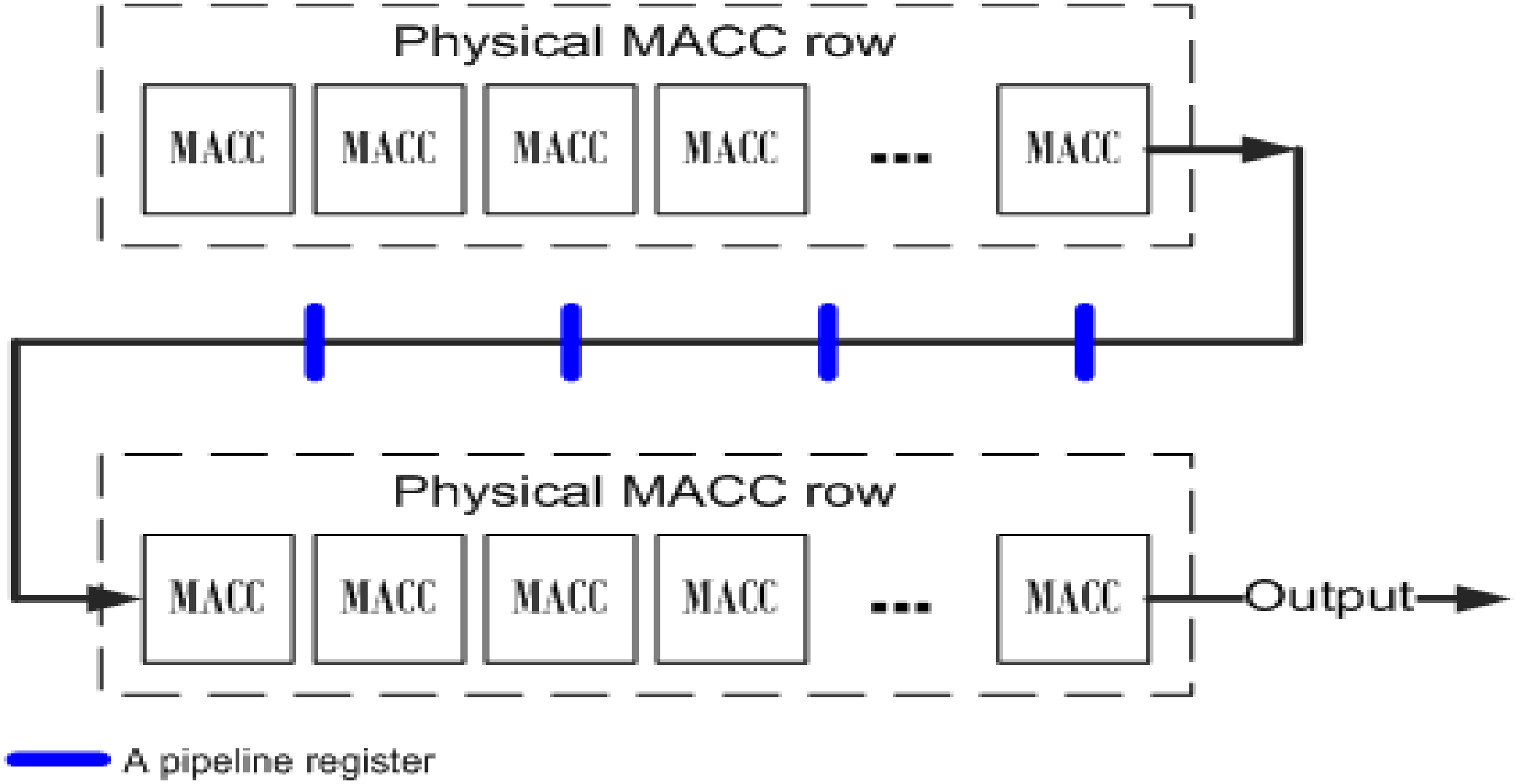
Long Input Lines and Buses (continued)



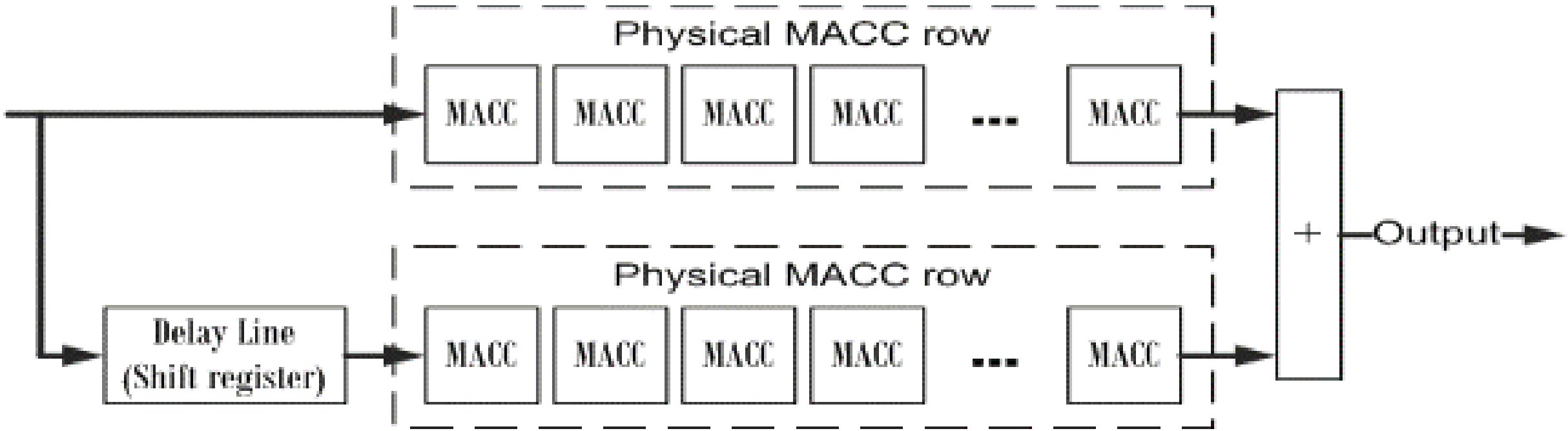
Cross-Row Connections

- There is no dedicated connection (cascade chain) between MACC physical rows
- Two techniques are used to establish the cross-row connections on FIR filter-like structures
  - Standard technique: connect the last MACC of a row to the first MACC of the next row (see the next slide for more information)
  - Advanced technique: use additional delay line (see the next slide for more information)
  - When using the standard technique, pipeline registers are not always placed optimally, which can cause unnecessary performance degradation
  - With long MACC chains, the second technique often provides better performance

Standard Cross-Row Connection



Advanced Cross-Row Connection



Conclusion

- Coping with timing challenges starts with understanding the root cause(s)
  - Analysis of slack distribution
  - Thorough review of compile reports and timing bottlenecks helps identify the root causes

- Methodology-based timing closure proposed
  - Provided hints and recommendations to cope with potential root causes
  - Main recommendations were as follows:
    - Consider all clock domains, not just the critical one
    - Reduce the number of asynchronous/synchronous reset signals
    - Clear and concise timing constraints and timing exception (multi-cycle and false paths)
    - Avoid floor plan that creates artificial congestion (IO placement, blocks floor plan, and so on)
    - Use hints when appropriate

- More to come soon
  - Block flow with incremental compile points
  - Power-aware design on RTG4

Thank You

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo, CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996
email: sales.support@microsemi.com
www.microsemi.com

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif., and has approximately 4,800 employees globally. Learn more at www.microsemi.com