

Xilinx UltraScale and UltraScale+ 架构介绍及设计方法学

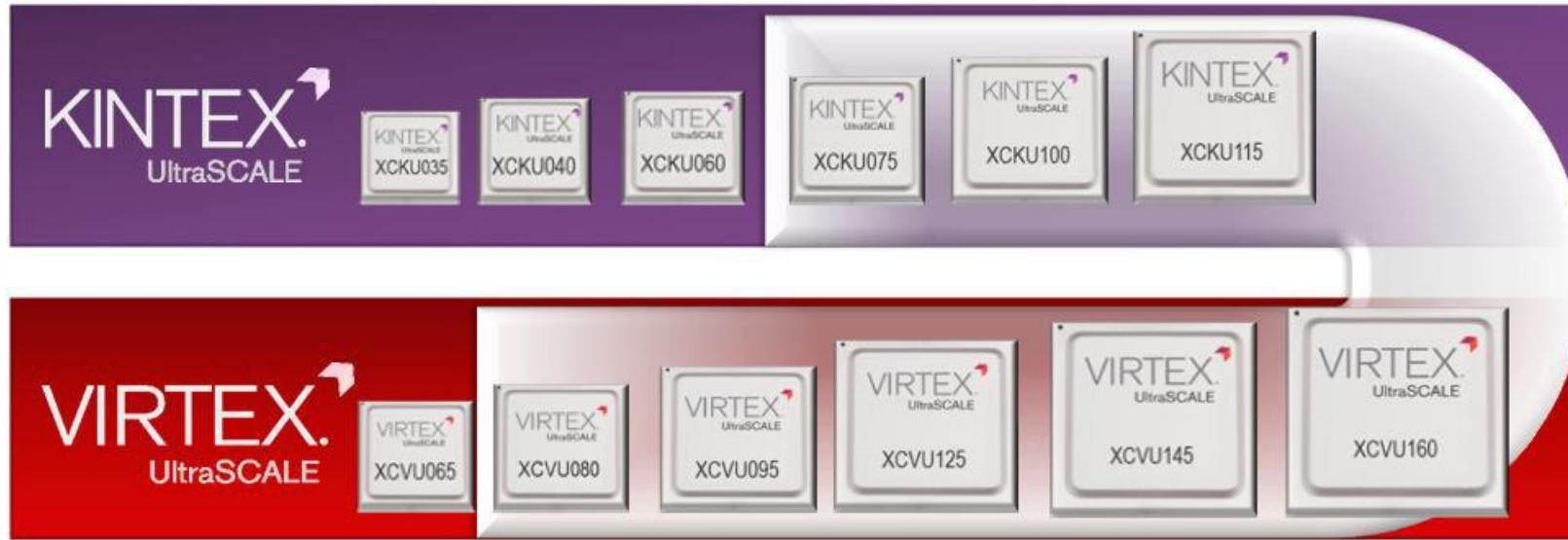
张剑森 Kenson.Zhang

依元素科技培训经理



- **Introduction to the UltraScale Architecture**
- **UltraFast Design Methodology**
- **HDL Coding Techniques**

Kintex and Virtex UltraScale Device Portfolio



Family Migration Path

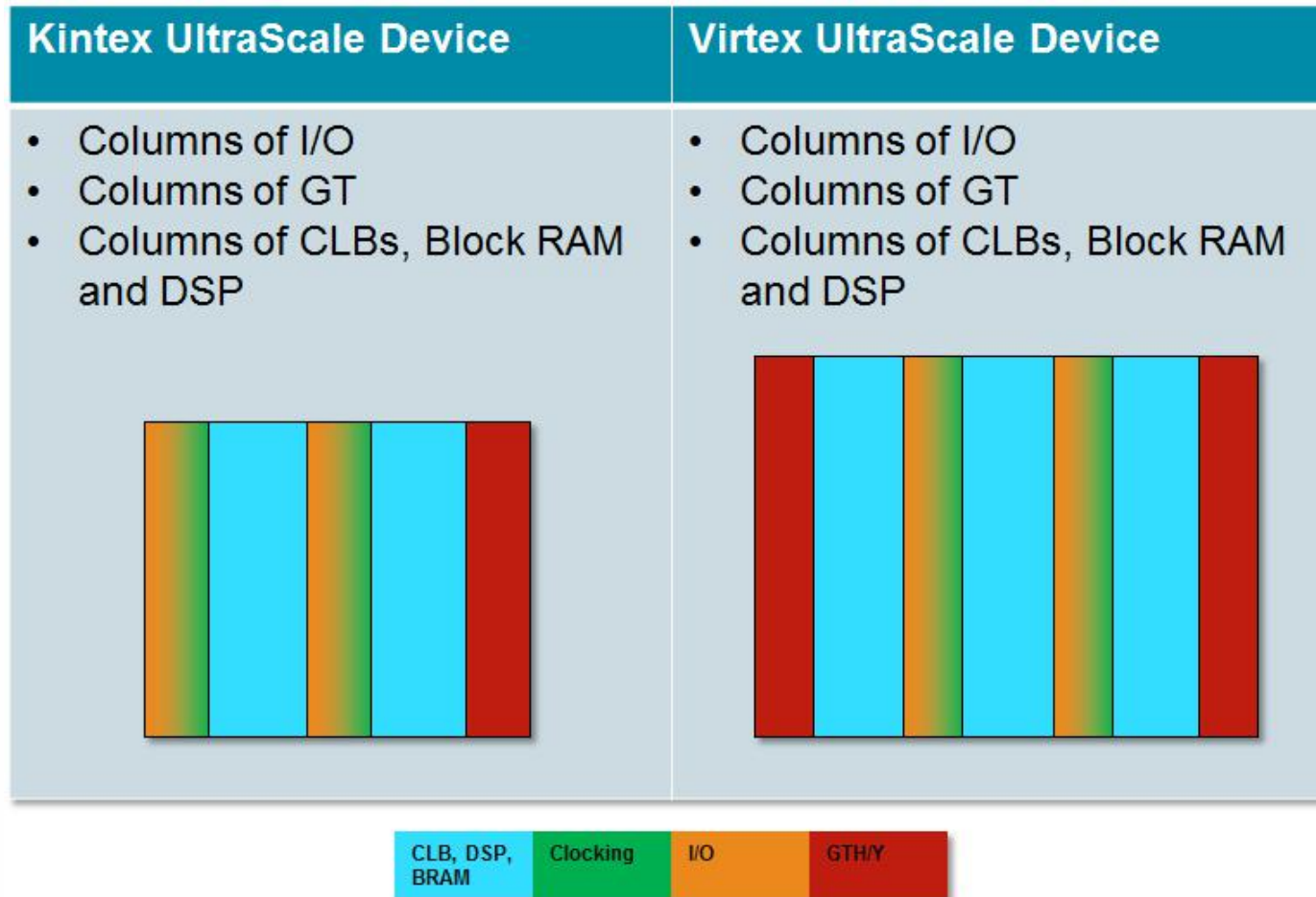
- ✓ Scalability for derivative applications
- ✓ Leverage PCB investment across platforms
- ✓ Future-proof with migration path to 16 nm

Kintex and Virtex UltraScale FPGA 20nm Capabilities

	KINTEX ⁷	KINTEX ⁷ UltraSCALE	VIRTEX ⁷	VIRTEX ⁷ UltraSCALE
Logic Cells (LC)	478	1,161	1,995	4,407
Block RAM (BRAM) (Mbits)	34	76	68	115
DSP-48	1,920	5,520	3,600	2,880
Peak DSP Performance (GMACs)	2,845	8,180	5,335	4,268
Transceiver Count	32	64	96	104
Peak Transceiver Line Rate (Gb/s)	12.5	16.3	28.05	32.75
Peak Transceiver Bandwidth (Gb/s)	800	2,086	2,784	5,101
PCI Express Blocks	1	4	4	6
Memory Interface Performance (Mb/s)	1,866	2,400	1,866	2,400
I/O Pins	500	832	1,200	1,456

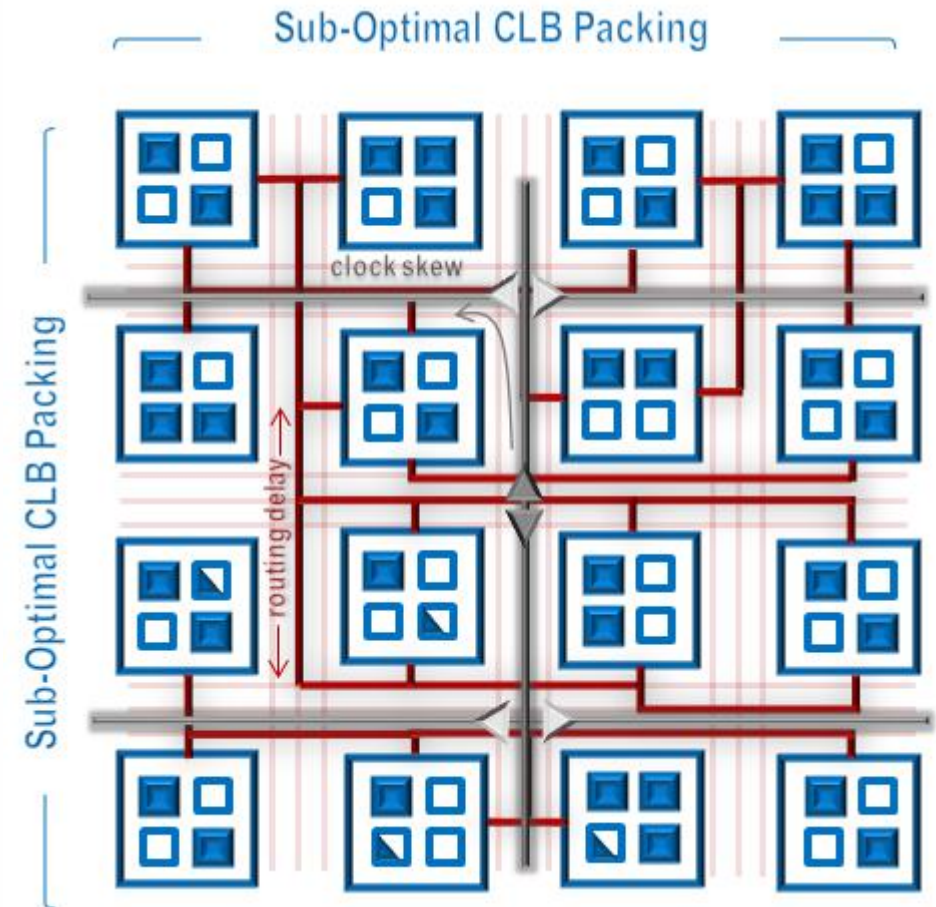
UltraScale Architecture Layout

> Side-by-side layout comparison (typical layout)



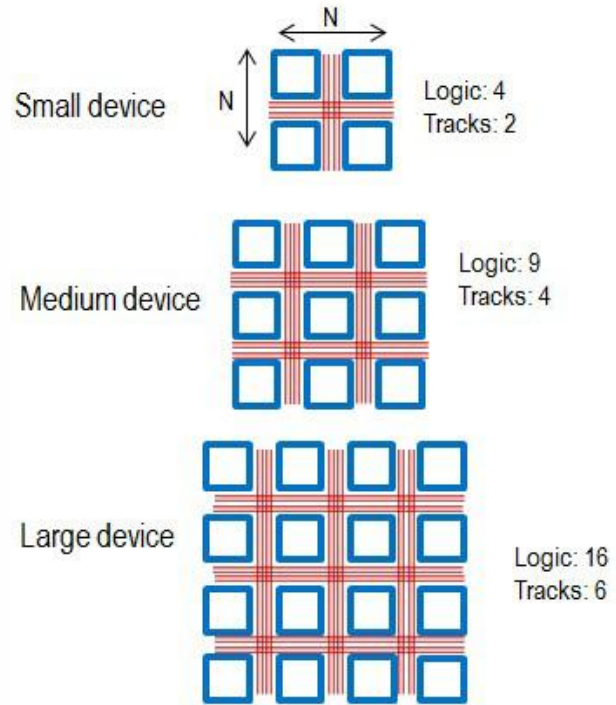
Silicon Architecture Address Interconnect Bottleneck for Next-Generation Designs

- > *Routing delay* dominates overall delay
- > *Clock skew* consumes more timing margin
- > Sub-optimal *CLB packing* reduces performance and utilization

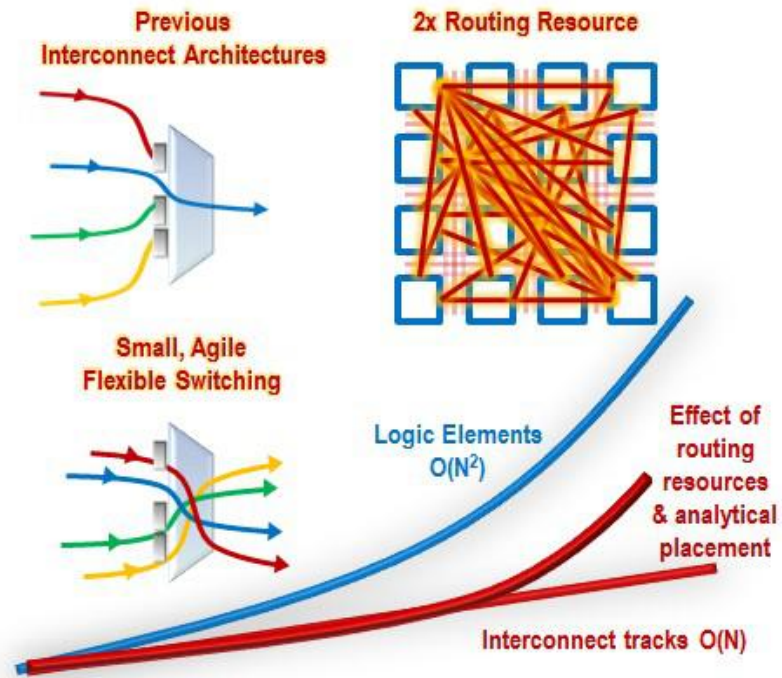


Next-Generation Routing for Utilization, Performance, and Run Time

- # Logic Elements grow in $O(N^2)$
- # Routing Tracks grow in $O(N)$

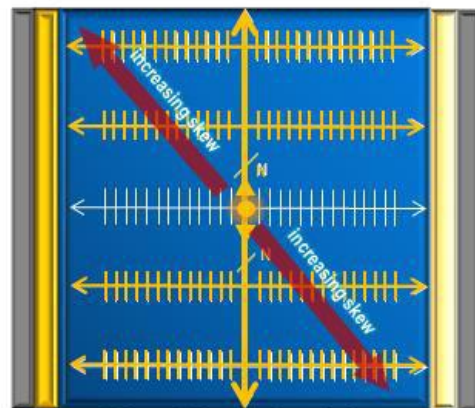


More Paths + Flexible Switching + Analytical Placement
Close the Gap and Deliver Full Routability

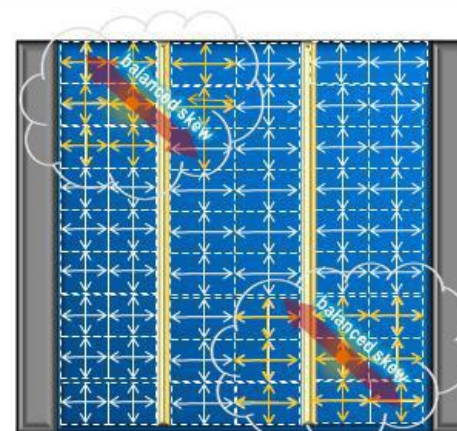


ASIC-like Clocking Maximizes Performance Margin and Reduces Dynamic Power

Feature	Benefit
Regional and segment-able clocking infrastructure	<ul style="list-style-type: none">• Clock resources scale with device density• Minimizes clock route wire length• Lowers dynamic power due to global clock net switching
Clock network centered on user logic	<ul style="list-style-type: none">• Balanced skew across clock distribution network delivers higher performance
Advanced clock management	<ul style="list-style-type: none">• 100s of global buffers with 1000s of placement options• Massive flexibility for global clock placement

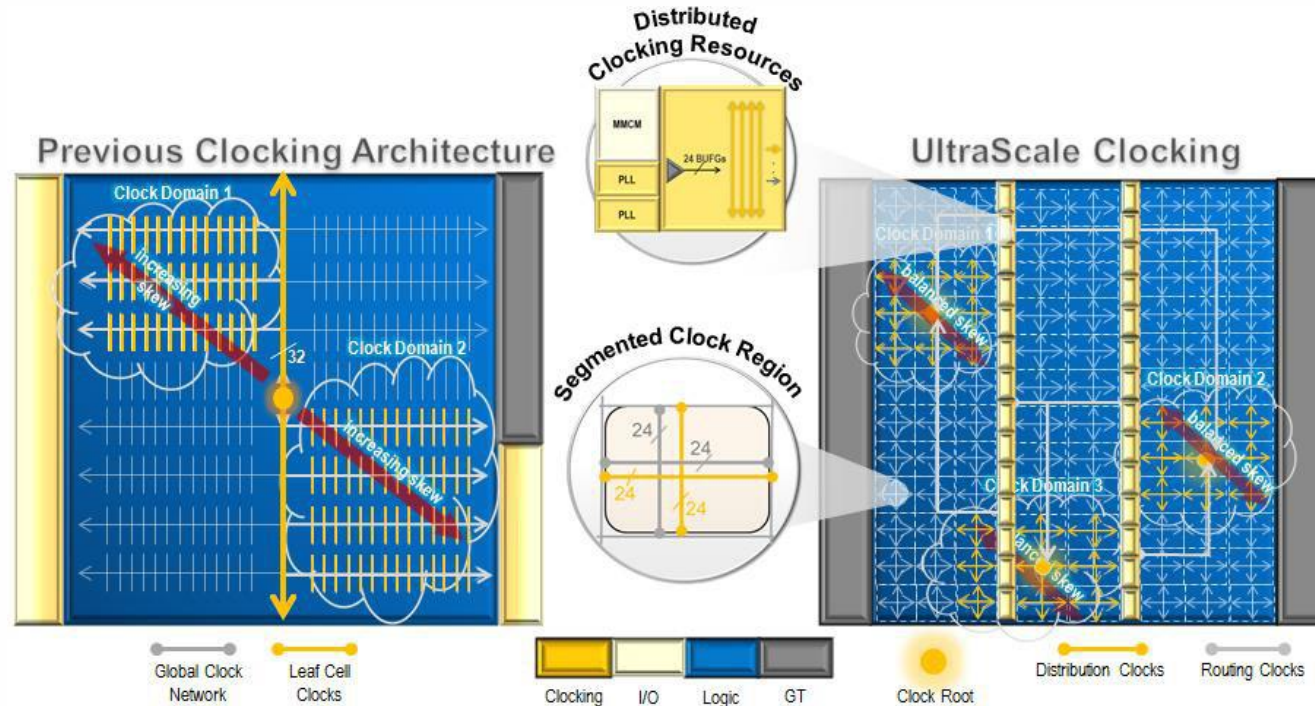


Previous Clocking Architectures
(Fixed Number of Clock Buffers)



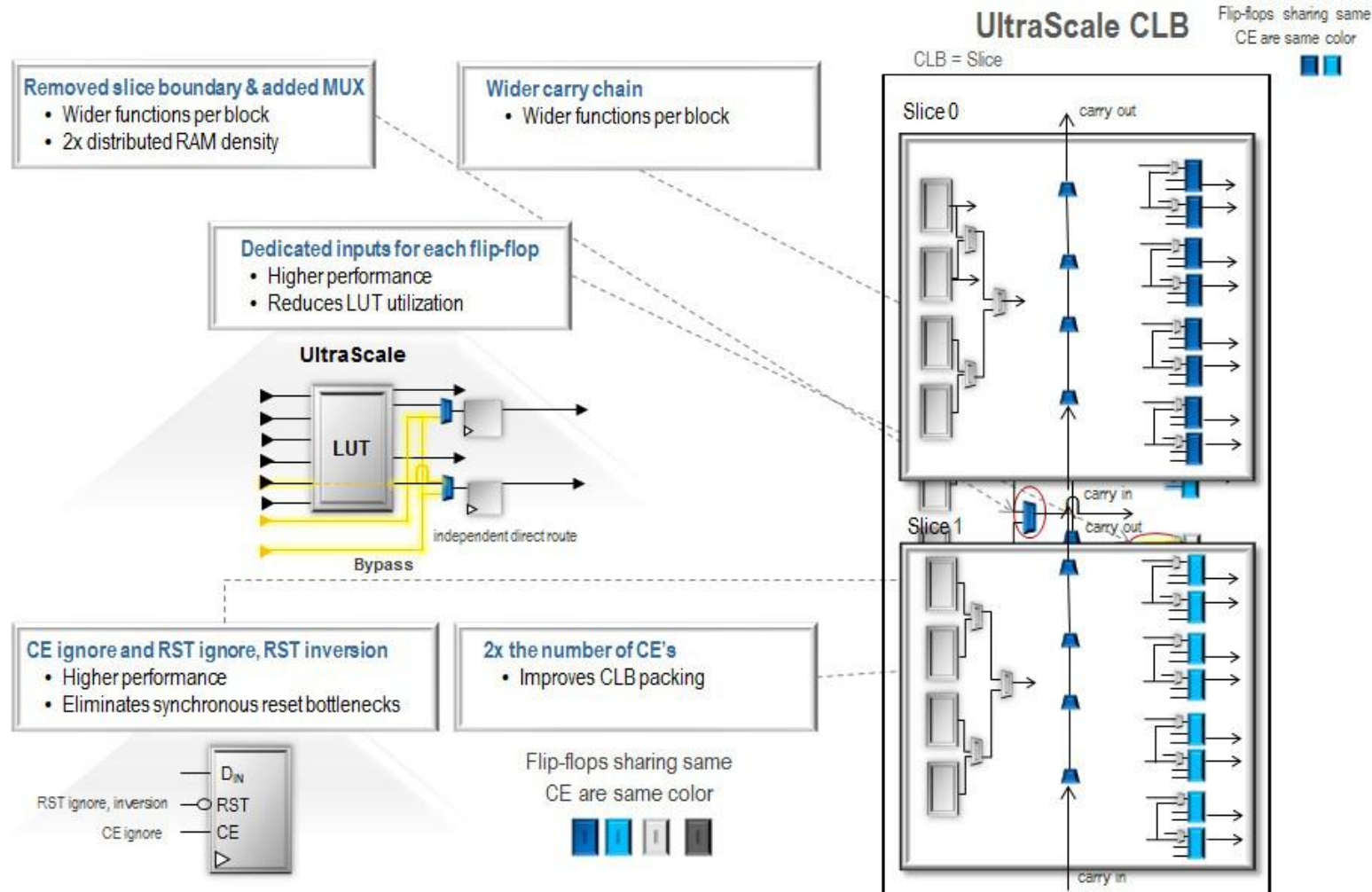
UltraScale FPGA Clock Routes
(Global Clock Buffers Scale)

Benefits of UltraScale FPGA Clock Architecture



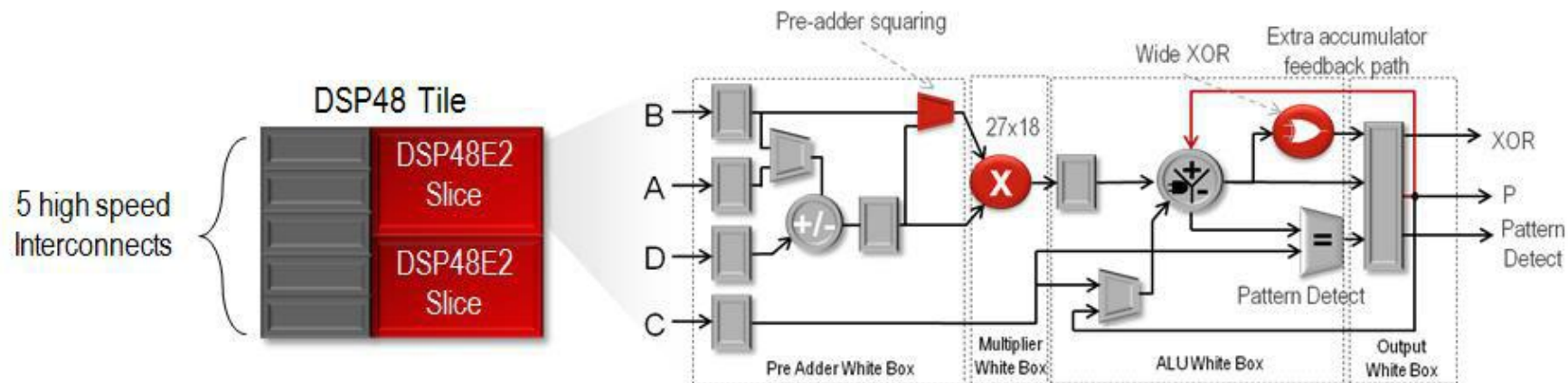
- > 32 centrally located global clock buffers
- > Root always in center of device
- > Skew accumulates from center to edge
- > 192 – 720 distributed global buffers
- > Root can be in any clock region
- > Balanced skew per clock network

CLB Enables Tighter Packing



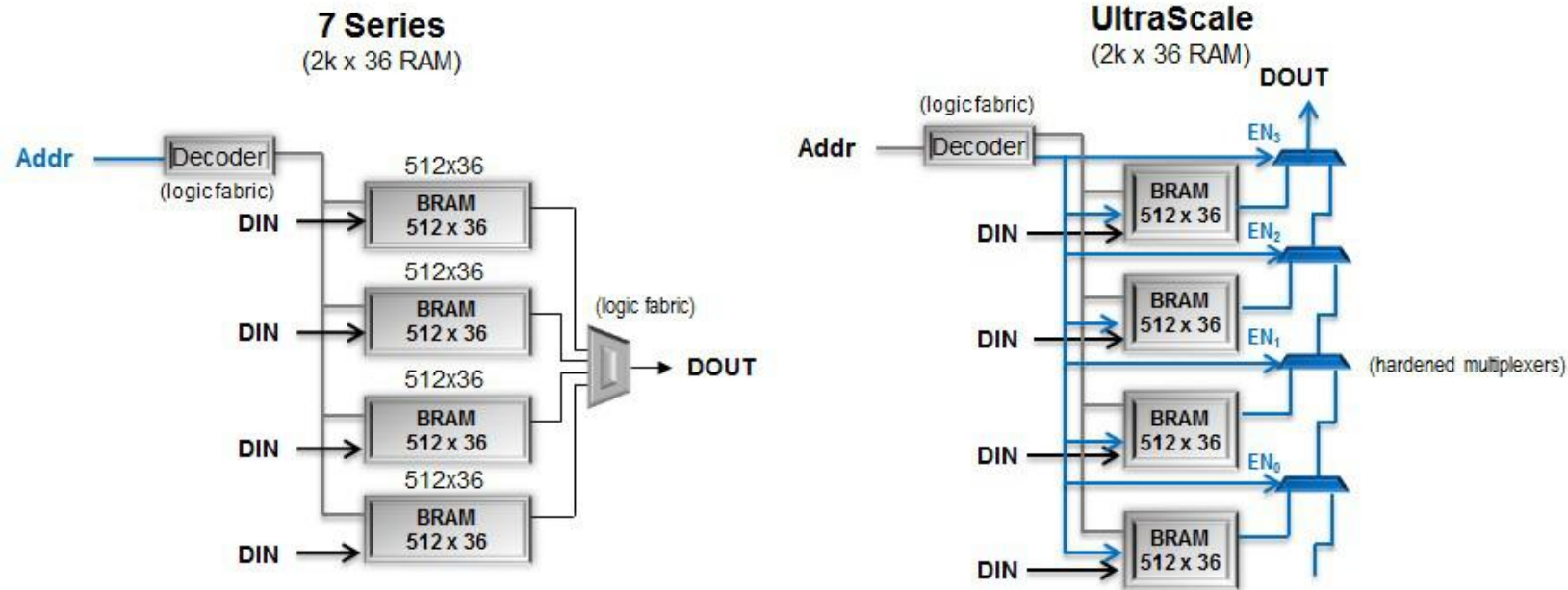
Enhanced DSP Sub-Systems for Performance and Efficiency

Feature	Benefit
27x18 multiplier in a DSP slice; 35x28 support in a DSP tile (2 slices)	<ul style="list-style-type: none"> Optimal performance per block Implement double-precision floating point in two-thirds the fabric
Pre-adder squaring	<ul style="list-style-type: none"> More efficient motion estimation in video applications Perform "sum-of-square-difference" calculations in 50% fewer resources
Extra accumulator feedback path	Implement complex multiply-accumulate in half the resources
Wide XOR	Implement EFEC, CRC, ECC functionality
White box modeling	Full visibility with accurate simulation and debug



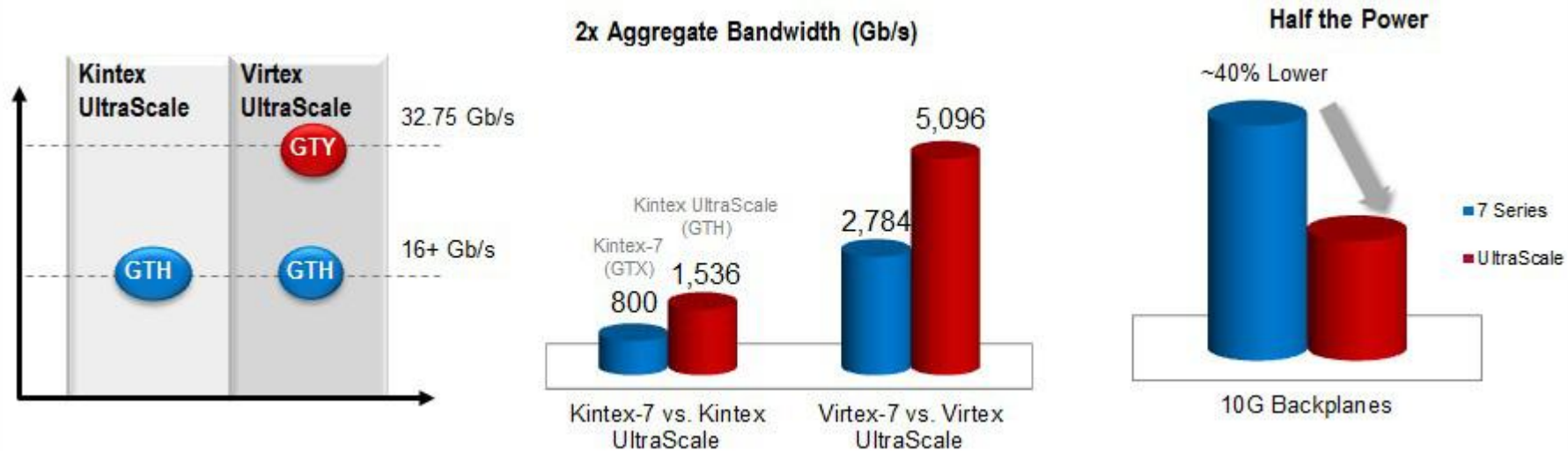
Optimized Block RAM Alleviate Bottlenecks for Many Applications

Feature	Benefit
Built-in, high-speed memory cascading	Eliminates CLB usage, reduces routing congestion and dynamic power consumption
Enhanced FIFO	<ul style="list-style-type: none"> • Lower power, greater performance than soft FIFO • Easy migration to soft core implementation for additional functionality • Asymmetric read and write port widths for clock domain crossings
User-accessible power gating of active BRAM	Reduces dynamic power when access to block RAM contents is temporarily not needed



Delivering Massive I/O Serial Bandwidth

Feature	Benefit
GTH (16 Gb/s) for price-performance	<ul style="list-style-type: none"> 12.5Gb/s performance in the lowest speed grade Enabled std: PCIe Gen4 (16G), JESD204B (12.5G), CPRI (16.3G), Serial Memory (HMC & MoSys)
GTY (32 Gb/s) for highest performance	<ul style="list-style-type: none"> 28Gb/s (CEI-25G-LR) backplane support for Nx100G to 400G systems Support for Interlaken, OTU4 over CFP4, 802.3bj (28G Ethernet backplane)
Major power reduction	~40% lower power for 10G backplanes
Continuous auto-adaptive equalization	Continuously optimizes link margin over PVT in increasingly challenging channel conditions



Integrated 100G Ethernet MAC, 150G Interlaken

Feature	Benefit
Large Scale Integration	<ul style="list-style-type: none"> • More headroom for power budget • Lower latency and higher performance • Frees up logic for additional functionality, e.g., packet processing • Simplified flow and easier routing for shorter run-times • No licensing requirements
Multiple configuration options	Flexibility to meet existing and future design requirements

Resource Savings

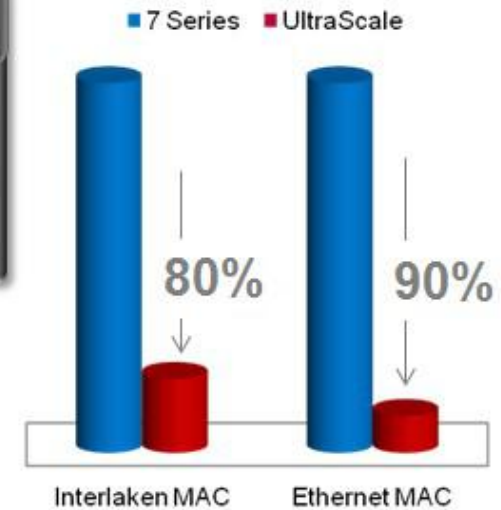
Interlaken (12 lane, 10G)	7-Series Soft IP	UltraScale Integrated IP
LUTs	32,700	0
Fabric Flip Flops	46,200	1,536
BRAM	16	0
Transceivers	12	12

Ethernet MAC + PCS (10x10G)	7-Series Soft IP	UltraScale Integrated IP
LUTs	70,000	0
Fabric Flip Flops	65,000	1,280
BRAM	41	0
Transceivers	10	10

Configuration Options

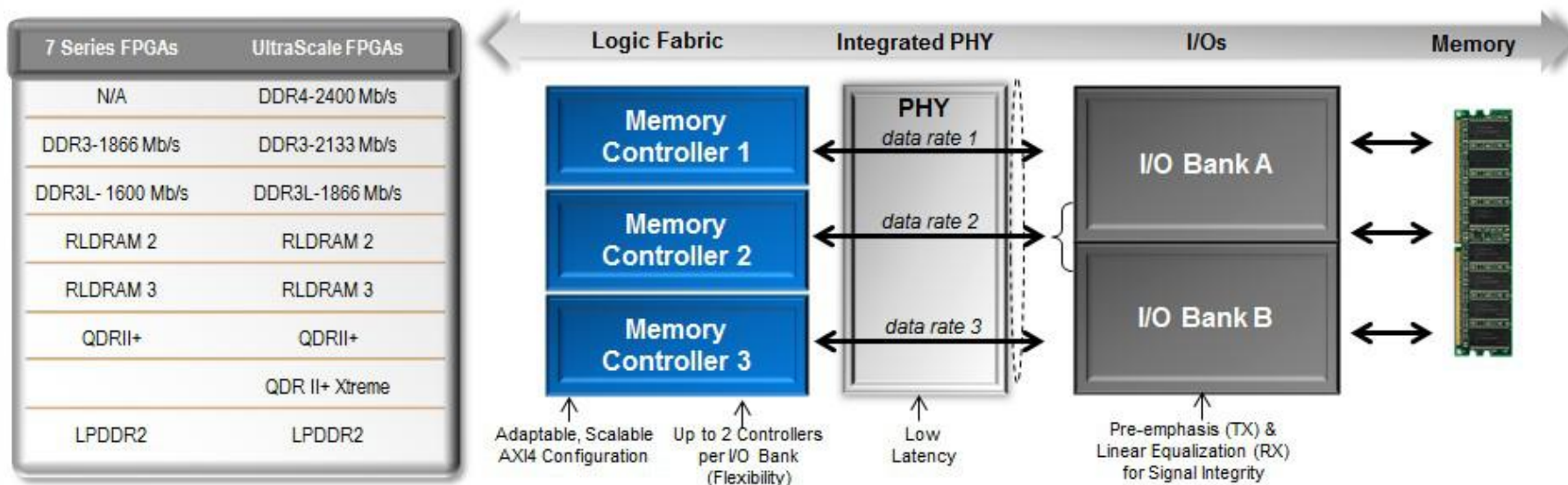
Hard IP	Lanes x Line Rate	
150G Interlaken	Up to 12 x 12.5Gb/s	Up to 6 x 25 Gb/s
100GE MAC	10 x 10 Gb/s	4 x 25Gb/s

Dynamic Power Savings



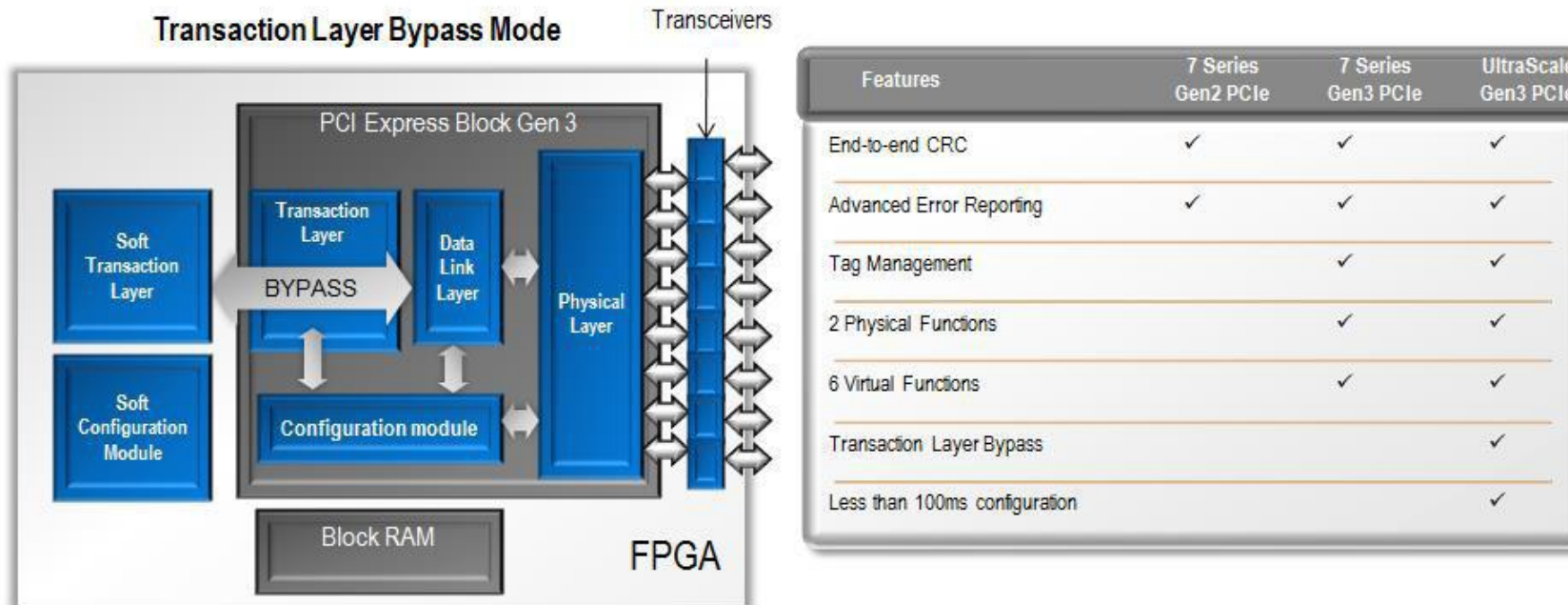
Enabling Massive External Memory Bandwidth

Feature	Benefit
DDR4 support (up to 2400 Mb/s)	40% higher data rates & 20% lower power than DDR3
Greater flexibility for different data rates	Up to two controllers per I/O bank
Next generation integrated PHY	Low latency along with high performance and low power
TX Pre-emphasis and RX linear equalization (CTLE)	<ul style="list-style-type: none"> Optimizes link margin for highest bandwidth Eases board design



Enhanced PCI Express Gen3 Integrated Core

Feature	Benefit
Built on existing Virtex-7 XT/HT core	Stability and production-proven foundation
Gen3 support in all devices & speed grades	Increase performance at lower price points
Transaction layer bypass mode	Ability to add additional physical & virtual functions for diverse topologies
Integrated circuitry for 100ms configuration	Meets PCI Express specification



Summary

- > The CLB architecture, routing architecture, and the Vivado Design Suite are designed to eliminate routing congestion
- > UltraScale devices have an ASIC-like clocking architecture that provides flexibility and performance for clock distribution
- > Logic enhancements reduce timing problems and design bottlenecks
- > I/O and transceiver bandwidth improvements are significant new features in the UltraScale architecture

- **Introduction to the UltraScale Architecture**
- **UltraFast Design Methodology**
- **HDL Coding Techniques**

Objectives

After completing this module, you will be able to:

- > Describe the RTL coding guidelines
- > Use control sets effectively in your design
- > Create an IP subsystem using IP integrator
- > Explain the Timing Reports generated by the Vivado® Design Suite
- > Apply Timing Exceptions to your design if necessary

Defining a Good Design Hierarchy

- > Add I/O components near the top level
- > Insert clocking elements near the top level
- > Register data paths at logical boundaries
- > Address floorplanning considerations
- > Optimize hierarchy for functional and timing debug
- > Apply attributes at the module level
- > Optimize hierarchy for advanced design techniques

RTL Coding Guidelines

- > Use Vivado Design Suite HDL templates
- > Control signals and control sets
- > Resets
- > Know what you infer
- > Coding styles to improve performance
- > Coding styles to improve power
- > Running RTL DRCs

HDL Coding Style Impact

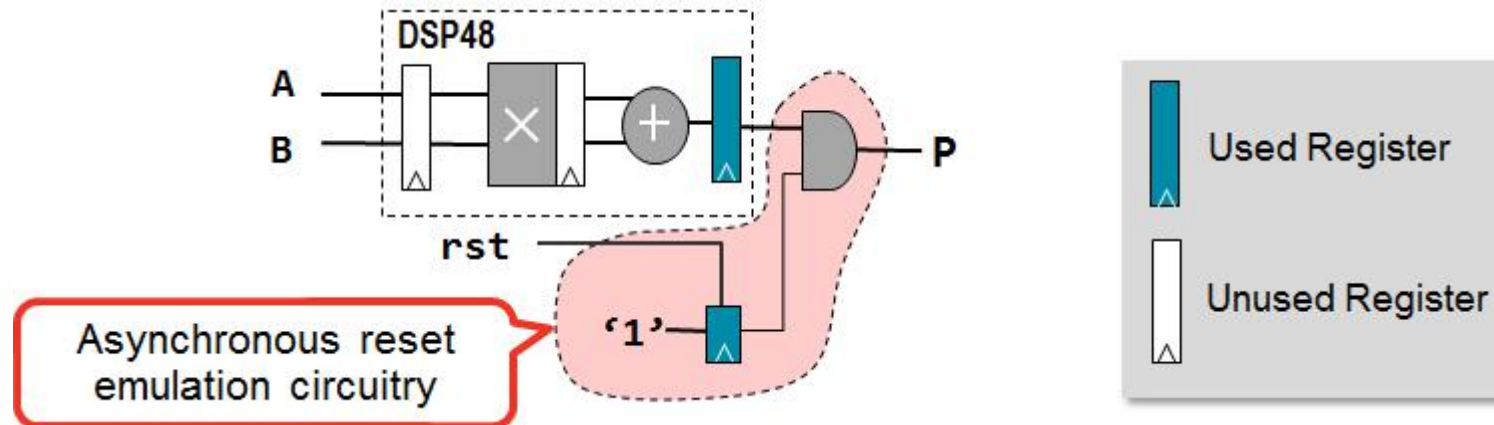
- > Follow recommended Vivado language templates for RAM and DSP inference
- > Use as many of the dedicated resources as possible (SRLs, DSP slices, block RAMs)
- > Pipeline your design to reduce levels of logic
- > Avoid reset
 - Resets can tax routing resources and are not always needed because Xilinx FPGAs always boot in a known state
- > Synchronous resets are preferred
 - Allow packing of registers into dedicated RAM and DSP blocks
- > Dedicated shifters (SRLs); do not use resets
- > RAM memory bits; do not use resets

Control Signals and Control Sets

- > A control set is the grouping of control signals (set/reset, clock enable and clock) that drives any given SRL, LUTRAM, or register
- > Designs with several unique control sets may have many wasted resources as well as fewer options for placement, resulting in higher power and lower performance
- > Designs with fewer control sets have more options and flexibility in terms of placement, generally resulting in improved results
- > In 7 series devices, slices all share common control signals and thus only registers with a common control set may be packed into the same slice
- > In UltraScale™ devices, there is more flexibility in control set mapping within a CLB
 - Resets that are undriven do not form part of the control set as the tie off is generated locally within the slice

Using Resets

- > Increase performance with the right reset choice
 - No reset at all (if possible) is best
 - Synchronous rather than asynchronous reset
 - Active HIGH rather than active low reset
 - Default register value can be controlled via the INIT property
- > Asynchronous reset interferes with DSP / RAM inference



Resets Recommendations

- > Remove resets where possible
 - Use INIT for initialization and only use explicit resets where needed
- > Use synchronous set/reset instead of asynchronous preset/clear when possible
 - Synchronous resets can be more efficient
- > When global resets are required, consider clock gating with BUFGCE to minimize timing impact of global resets
 - Allows more freedom to the placer and router

Tips for Control Signals

- > Check whether a global reset is really needed
- > Avoid asynchronous control signals
- > Keep clock, enable, and reset polarities consistent
- > Do not code a set and reset into the same register element
- > If an asynchronous reset is absolutely needed, remember to synchronize its deassertion

Know What You Infer

- > Anticipate hardware resources mapping when coding
- > Monitor actual result in elaborated design
- > RAM
 - Check for multi-fanout on the output of read data registers
 - Check for reset signals on the address/read data registers
 - Check for feedback structures in registers
 - Take retiming into account!
- > Optimal DSP and arithmetic inference
 - Fully pipeline code for DSP48
 - Avoid set or asynchronous reset around DSP48 (only synchronous reset)
 - Use signed values for most efficient, full-bit mapping
 - Beware of bus sizing
 - Can prevent synthesis from using the block fully while being logically correct
 - Account for bit growth for pre-adder / M to P path
 - Use pipelined (adder) chain instead of (adder) tree

Improving Performance

> High fanouts in critical paths

- Reduce loads to the portions of the design that do not require it
- Use register replication
 - Increases the speed of critical paths by making copies of registers to reduce the fanout of a given signal

> Pipelining

- Restructure the long datapaths with several levels of logic and distribute them over multiple clock cycles
- Allows for a faster clock cycle and increased data throughput at the expense of latency and pipeline overhead logic management
- Consider pipelining up front
- Balance latency
- Avoid unnecessary pipelining

Improving Power

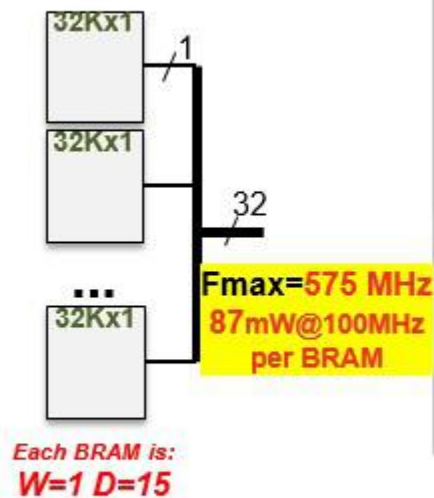
- > Gate clock or data paths
 - Common technique to prevent switching/glitches from propagation
 - The Vivado Design Suite gates logic for power, but some dependencies the tool does not know
- > Maximize gating elements
 - Gate entire clock domain
- > Use clock enable pins of dedicated clock buffers
 - Avoid LUTs or other methods to gate clock-signals
- > Keep an eye on control sets
 - Avoid fine-grained clock gating
- > Use case block when priority encoder not needed
 - Avoid large if-then-else constructs

Performance/Power Trade-off for Block RAMs

- > 32k x 32-bit RAM with cascade_height
 - Variations using cascade_height (all 32 BRAMs)

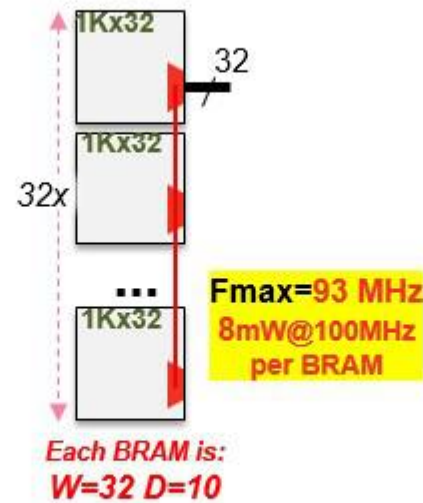
```
...  
(* cascade_height = 1 *)  
reg [31:0] mem [(2**15)-1:0];  
reg [14:0] addr_reg;  
...
```

cascade_height = 1



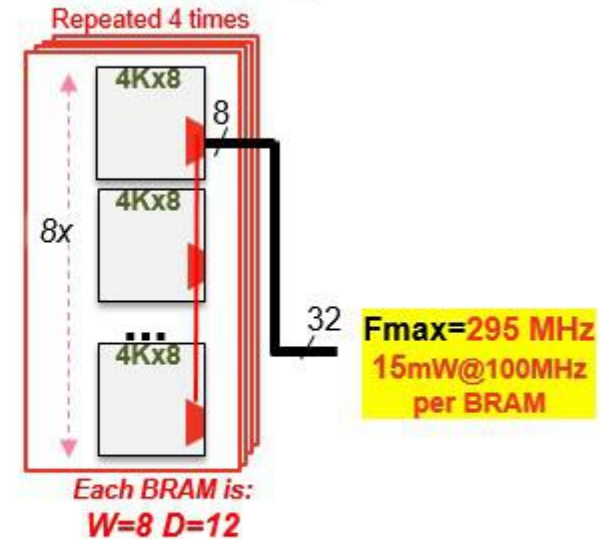
- 32 BRAMs enabled
- No extra logic

cascade_height = 32 (*)



- 1 BRAM enabled per access
- Lower power

cascade_height = 8



- 4 BRAMs enabled for each access
- Power / Performance tradeoff

Best Practices for Power Analysis

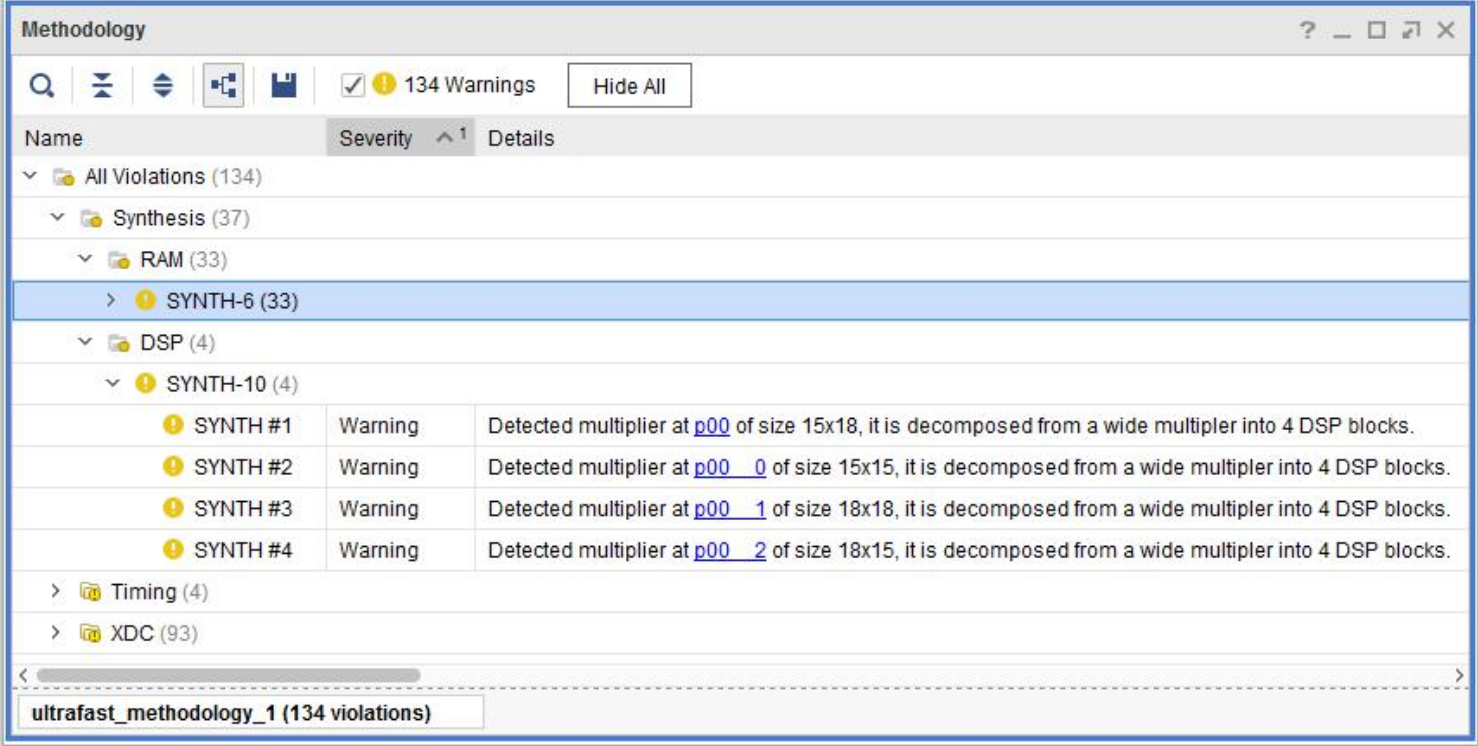
- > Use the Vivado Report Power to estimate the power through all stages of a design
 - The accuracy of the power estimates varies depending on the design stage when the power is estimated
- > Use Report Power in either of the two supported modes to estimate power depending on accuracy required
 - Vector-based power estimation is a more accurate estimate than vector-less mode
- > Use accurate clock constraints and I/O constraints in your design to obtain accurate power analysis

Using the UltraFast Design Methodology DRCs

- > The Vivado Design Suite contains a set of methodology-related DRCs you can run using the `report_methodology` Tcl command
- > This command has rules for each of the following design stages
 - Before synthesis in the elaborated RTL design to validate RTL constructs
 - After synthesis to validate the netlist and constraints
 - After implementation to validate constraints and timing-related concerns
- > For maximum effect, run the methodology DRCs at each design stage and address any issues prior to moving to the next stage

Automated UltraFast Design Methodology Report

- > The UltraFast Design Methodology report is automatically generated whenever a design that has violations is opened after synthesis or implementation



The screenshot shows a window titled "Methodology" with a toolbar containing search, zoom, and other icons. A status bar indicates "134 Warnings" and a "Hide All" button. The main area is a tree view of violations:

- All Violations (134)
 - Synthesis (37)
 - RAM (33)
 - SYNTH-6 (33)
 - DSP (4)
 - SYNTH-10 (4)

Name	Severity	Details
SYNTH #1	Warning	Detected multiplier at p00 of size 15x18, it is decomposed from a wide multiplier into 4 DSP blocks.
SYNTH #2	Warning	Detected multiplier at p00_0 of size 15x15, it is decomposed from a wide multiplier into 4 DSP blocks.
SYNTH #3	Warning	Detected multiplier at p00_1 of size 18x18, it is decomposed from a wide multiplier into 4 DSP blocks.
SYNTH #4	Warning	Detected multiplier at p00_2 of size 18x15, it is decomposed from a wide multiplier into 4 DSP blocks.
 - Timing (4)
 - XDC (93)

Working with IP: Packaging Custom IP

- > Pre-validated intellectual property (IP) cores significantly reduce design and validation efforts, and ensure a large advantage in time-to-market
- > Xilinx uses the industry standard IP-XACT format for delivery of IP, and provides tools (IP Packager) to package custom IP
- > The Vivado IP packager enables you to create custom IP for delivery in the Vivado IP catalog
- > Before packaging your IP HDL, ensure its correctness by simulating and synthesizing to validate the design
- > Ensure that the desired list of supported device families is defined properly while creating the custom IP definition
 - This is especially important if you want your IP to be used with multiple device families
- > The IP catalog is a single location for Xilinx-supplied IP
 - All Xilinx and third-party vendor IPs are categorized based on applications here

Creating IP Subsystems with IP Integrator

- > IP integrator is the interface for connecting IP cores to create domain specific subsystems and designs
- > IP subsystems are best configured using the IP integrator feature of the Vivado IDE
 - Interactive block design capabilities of the IP integrator make the job of configuring and assembling groups of IP easy

Revision Control

- > Manage sources in the Vivado Design Suite® with the revision control system
 - HDL, IP XCI, IP BD, XDC, Tcl scripts, etc.
 - Manage different source types in separate remote directories
- > Two main revision control strategies
 - Maximum flexibility
 - Shorter runtime to rebuild the project
 - Minimum number of files
 - Least number of files to manage at the expense of flexibility
- > Xilinx recommends following the maximum flexibility strategy
 - A large number of files in revision control maximizes flexibility

Using Timing Reports

> Create and validate clocks

- `check_timing`: for missing clocks and I/O constraints
- `report_clocks`: check frequency and phase
- `report_clock_networks`: possible clock root

> Validate clock groups

- `report_clock_interaction`

> Validate I/O delays

- `report_timing -from [input_port] -setup/-hold`
- `report_timing -to [output_port] -setup/-hold`

> Add exceptions if necessary

- Validate using `report_timing`

Understanding the Timing Reports

- > Timing Summary report provides high-level information on the timing characteristics of the design
- > Use the Timing Summary report for sign-off post-implementation
- > Use the Check Timing report to identify any missing timing constraints in the design

Working with Constraints

- > Design constraints define the requirements that must be met by the design in order for the design to be functional in hardware
- > Synthesis and implementation constraints
- > Timing constraints
 - Process of defining good timing constraints is broken into the four steps
 - Defining clock constraints
 - Constraining input and output ports
 - Defining clock groups and CDC constraints
 - Specifying timing exceptions

Timing Exceptions: Less is More!

> Goals

- Avoid higher implementation run times
- Adjust unrealistic timing requirements to help timing closure

> Start with fewer or no exceptions

- Meeting timing with fewer exceptions is OK
- Use clock group exceptions rather than point-to-point exceptions
- Avoid complex expressions with filters
- Avoid exceptions affecting too many paths

> Avoid these constraint types

- `set_false path -through...`
- `set_max_delay -from [all_fanout -from ck1] -to [all_fanout -from ck2]`

Summary

- > Use the given RTL coding guidelines
 - Use Vivado Design Suite HDL templates
 - Control signals and control sets
 - Resets
 - Know what you infer
 - Coding styles to improve performance
 - Coding styles to improve power
 - Running RTL DRCs
- > The Vivado IP packager enables you to create custom IP for delivery in the Vivado IP catalog
- > Timing reports help you determine why your design fails to meet its constraints

- **Introduction to the UltraScale Architecture**
- **UltraFast Design Methodology**
- **HDL Coding Techniques**

Objectives

After completing this module, you will be able to:

- > Identify how the use of control signals (sets, resets, and clock enables) can impact your device utilization
- > Describe the benefits of following Xilinx recommendations on resets
- > Describe the difference between the inference and instantiation
- > Code for your design so that you can infer the dedicated hardware resources
- > Describe the recommended coding techniques

Control Signals

- > Each flip-flop has three control signals
 - CK – clock
 - CE – clock enable (active High)
 - SR – asynchronous/synchronous set/reset (active High)
 - Either set or reset can be implemented (but not both)
- > A grouping of control signals is a control set
- > Designs with fewer control sets have more options and flexibility in terms of placement

Control Port Usage Rules

- > Clocks and asynchronous set/resets always gets connected to flip-flop control signals
 - They cannot be moved to the datapath (to build equivalent logic with a LUT)
- > Clock enables and synchronous set/resets
 - Connected to flip-flop control signals when most of the flip-flops in a slice share the same control sets (this is decided by the tools)
 - Can be moved to the datapath (to a LUT input)
- > Asynchronous sets/resets have priority access to the control signals over synchronous sets/resets
 - For example, if a global asynchronous reset and a local reset are inferred on a single register
 - The asynchronous reset gets the port on the register
 - The synchronous reset gets mapped to a LUT input

Resets

- > Two kinds of resets are still supported—global and local
 - Global: performed automatically after configuration has finished
 - Performed by default and does not need to be coded into the design
 - Access to this net is done with the global set/reset (GSR) port from the Startup component
 - Access is only necessary if you want to perform a global reset a second time
 - Note that if you are coding a global reset into your HDL, you are actually coding in a second reset
 - Local: Internally generated targeted reset
 - Used as a standard part of some components behavior (FSM, counters, etc.)

Synchronous Versus Asynchronous Resets

- > Asynchronous resets
 - Deassertion should be synchronous
 - Otherwise creates problems such as metastability
 - Use a reset bridge (use of two flip-flops back-to-back)
- > Synchronous sets/resets make FPGA designs more reliable
 - Do not need any special timing constraints
 - Are often the most critical net in a design
 - Are more predictable and stable
 - Less susceptible to accidentally missing timing, runt pulses, or other phenomenon from upsetting logical functionality
 - Less prone to race conditions
 - Note that the release of an asynchronous signal may not always have predictable timing results

Avoid the Use of Both a Set and Reset on a Flip-Flop

- > Using both a set and a reset will require additional logic
 - Flip-flops cannot implement a set and a reset without additional LUTs
 - This may or may not create an extra level of logic on the datapath
 - Use of an asynchronous set and reset can affect timing and resource utilization and should be avoided

> For example

```
always @(posedge reset, posedge set, posedge clk)
  if (reset)

    a_reg <= 1'b0;
  else if (set)
    a_reg <= 1'b1;
  else
    a_reg <= A;
```

- This would require extra logic to generate a single asynchronous set or reset signal from two signals
- Instead use synchronous control signals

Resource-Aware Coding

- > Coding mistakes and the random use of resources and control sets can lead to reduced utilization and speed/performance of a device
- > Xilinx recommends the following guidelines to ensure design efficiency
 - Take advantage of hard blocks to map large register arrays
 - Use as many of the dedicated resources as possible (SRLs, DSP slices, block RAMs)
 - Turn off the Logic Replication synthesis option to reduce your design size
 - Control the use of clock enables with HDL code

Dedicated Resources

- > Faster than LUTs/flip-flops
- > Consumes less power
- > Timing of the dedicated blocks is already taken care of
- > Offers as much as three times the performance
- > DSP48E, FIFO, block RAM, ISERDES, etc.
- > Xilinx recommends the use of dedicated resources

Inference Versus Instantiation

- > The key to Xilinx optimization is accessing and controlling device-level resources, as well as overall place & route results
 - From an HDL perspective, there are only two means to access any resource

> Inference

- Generic HDL code
 - Synthesis tool decides which vendor library to use
- Device optimization as per tool ability
- Maximum portability



> Instantiation

- Create "instance" of
- Designer references specific vendor macro
- Maximum device optimization
- May be required
- Limits portability

Resources Inference

- > Components can be inferred in the design by synthesis tool by providing their functionality
- > Can be inferred by all synthesis tools
 - Shift register LUT (SRLC32E)
 - F7, F8, and F9 multiplexers
 - Carry logic
 - Multipliers and counters using the DSP block
 - Global clock buffers (BUFG)
 - SelectIO (single-ended) standard
 - I/O registers (single data rate)
 - Input DDR registers
- > Can be inferred by some synthesis tools
 - Memories
 - Global clock buffers (BUFGCE, BUGFCTRL)
 - Some complex DSP functions
- > Cannot be inferred by any synthesis tools
 - SelectIO (differential) standard
 - Output DDR registers
 - MMCM/PLL
 - Local clock buffers (BUFIO, BUFR, BUFG_LEAF)

Instantiation

- > Components can be instantiated in the design by using adding an instance in the HDL code

- > Xilinx recommends that you instantiate the following elements
 - Memory resources
 - Block RAMs specifically (use the IP catalog)
 - SelectIO technology standard resources
 - Clocking resources
 - MMCM, PLL (use the IP catalog)
 - IBUFG, BUFGMUX, BUFGCE
 - BUFIO, BUFR

Register Initialization

- > Xilinx recommends that designers regularly initialize their registers on any inferred flip-flop, SRL, or RAM

```
signal reg: std_logic := '1';
```

```
...
```

```
process (clk) begin
```

```
  if rising_edge(clk) then if (rst='1') then
```

```
    reg <= '0'; else
```

```
    reg <= val;
```

```
  end if; end if;
```

```
end process;
```

> Benefits

- The initialization eliminates the need to specify a set condition for the sole purpose of simulation (creating a logic one)
- This saves resources and allows the RTL to more accurately behave as the FPGA

Coding Techniques

- > Avoid coding for active low control signals
- > Controlling the use of clock enables with HDL code will decrease the LUT use
- > Avoid unnecessary use of sets and resets; if required, use the synchronous set/reset
- > Avoid asynchronous resets on block RAMs, DSPs
- > Limit the use of low fanout control signals
- > Synthesis tools can move synchronous resets from control ports to the datapath
- > Xilinx recommends not using the synthesis option to convert asynchronous resets to synchronous

Summary

- > Minimize the use of control sets wherever possible
- > Use the local or global resets wisely
- > Use of both a set and reset control signal cannot be implemented on a flip-flop without the use of extra logic on the datapath
- > Xilinx recommends that designers regularly initialize their registers on any inferred flip-flop, SRL, or RAM
- > Use of the available dedicated resources offer three times performance
- > The Vivado Design Suite should be used to manage control signal replication, rather than designers manually replicating logic