ADIUVO
ENGINEERING AND TRAINING, LTD.

AMD Vivado™ Design Suite Essentials: Key Techniques for Superior RTL Development

Adam@adiuvoengineering.com

Objective

Session Objectives:

1. Introduce the Ultrafast Design Methodology
2. Architecting your FPGA
3. Control Sets and Signals
4. RTL Design Templates
5. Design Review Checks

Architecture

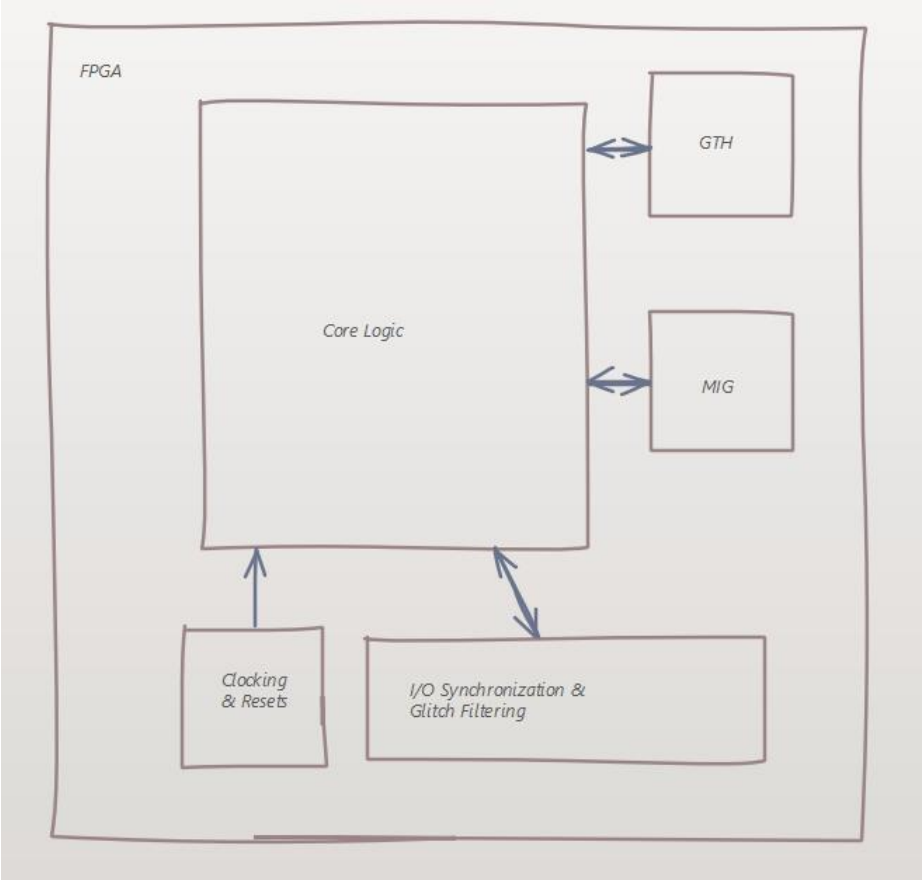
Challenge: Getting it wrong can have significant impacts

Plan the architecture from day one and try to leverage IP cores wherever possible to reduce the amount of development needed. When it comes to hierarchy, there are several considerations:

Architecture

Simple rules which can help significantly:

1. Keep clocking & IO functionality (e.g., gigabit serial links and memory interfaces) at the top level
2. Leverage hierarchy – design appropriately
3. Use existing IP blocks



Architecture

Leverage the register-rich environment of FPGAs.

1. Registering the inputs and outputs of all modules yield several benefits:
    - Contains critical paths to module
    - Prevents optimization across modules
    - Simplifies the debugging and analysis

2. Ensure the attributes are leveraged:
    - KEEP_HIERARCY
    - SHREG_EXTRACT

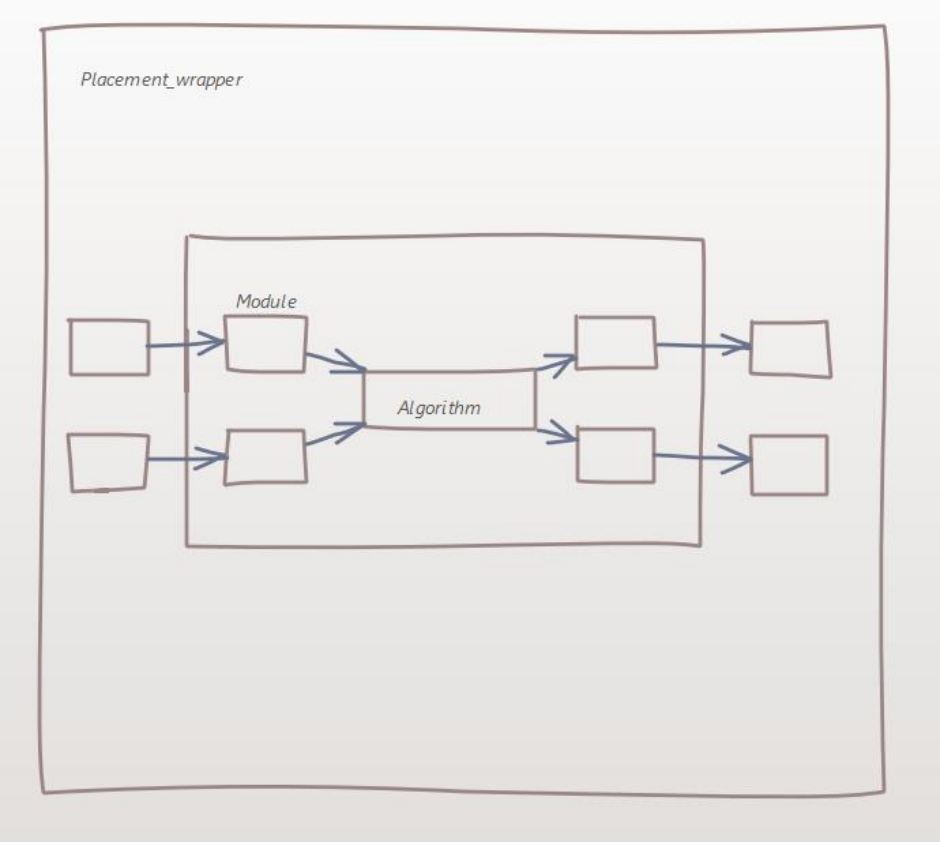
Architecture

Design modules are like an onion with several layers.

Placement wrapper – registers can help with implementations at higher speeds.

Registers in placement wrapper around module allows placement to be optimized for implementation.

Signals cannot cross die in one clock so additional registers enable more placement options as signals are registered.



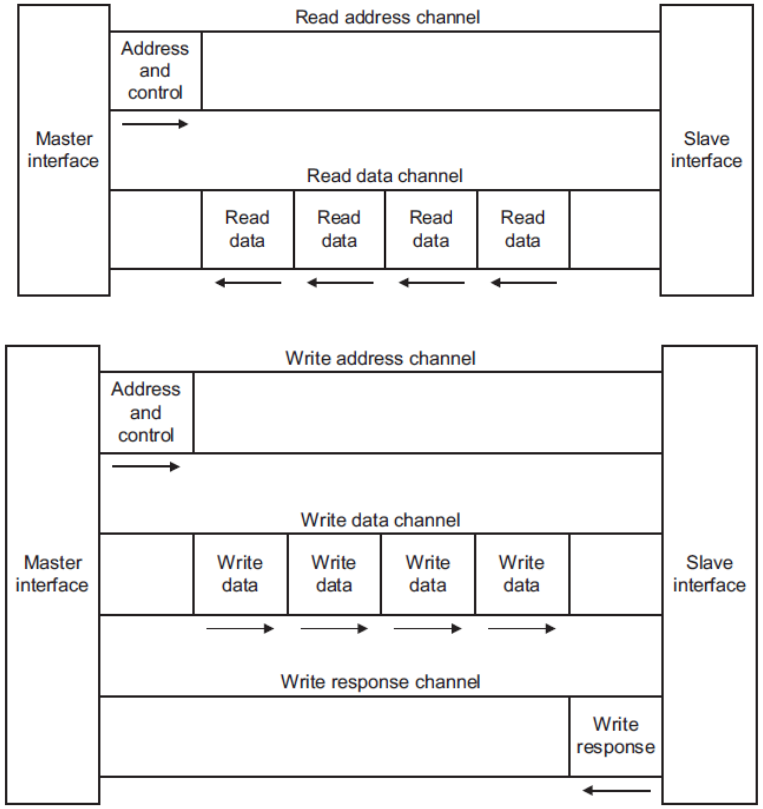
Architecture

Design reuse is a critical aspect and a high priority.

One key aspect to reuse is interfacing - when developing IP modules, use standard interfaces:

- AXI Memory Map – High bandwidth memory mapped transactions e.g. DMA
- AXI Lite – Low speed memory mapped configuration of registers in IP
- AXI Stream – Unidirectional data stream, unaddressed for point-to-point transfer

Architecture



Read Address

- ARVALID
- ARREADY
- ARID[m:0]
- ARADDR[a:0]
- ARLEN[7:0]
- ARSIZE[2:0]
- ARBURST[1:0]
- ARPROT[2:0]
- ARLOCK
- ARCACHE[3:0]
- ARREGION[3:0]
- ARQOS[3:0]

Handshake

Read Data

- RVALID
- RREADY
- RID[m:0]
- RDATA[n-1:0]
- RRESP[1:0]
- RLAST

Handshake

Write Address

- AWVALID
- AWREADY
- AWID[m:0]
- AWADDR[a:0]
- AWLEN[7:0]
- AWSIZE[2:0]
- AWPROT[2:0]
- AWBURST[1:0]
- AWLOCK
- AWCACHE[3:0]
- AWREGION[3:0]
- AWQOS[3:0]

Handshake

Write Data

- WVALID
- WREADY
- WDATA[n-1:0]
- WSTRB[n/8-1:0]
- WLAST

Handshake

Write Response

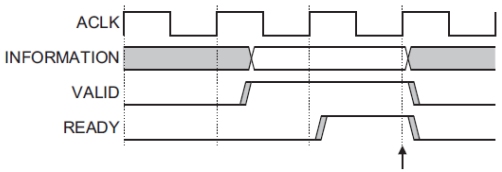
- BVALID
- BREADY
- BID[m:0]
- BRESP[1:0]

Handshake

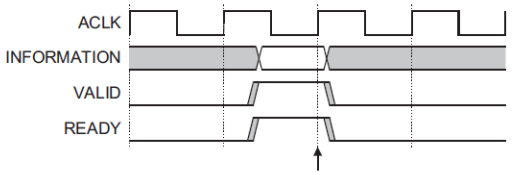
Architecture

All AXI channels follow READY/VALID handshake rules:

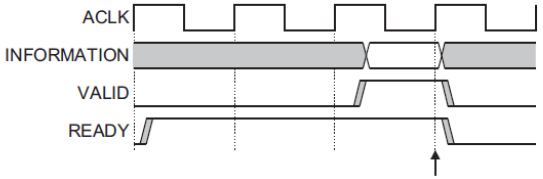
- Once ‘VALID’, it can’t be de-asserted until the corresponding ready is received
- Once ‘VALID’, other signals driven by initiator must not change
- ‘READY’ can be withheld
- ‘VALID’ must NEVER be withheld (system may deadlock)
- No combinatorial paths between input and output signals of the same interface



VALID before READY

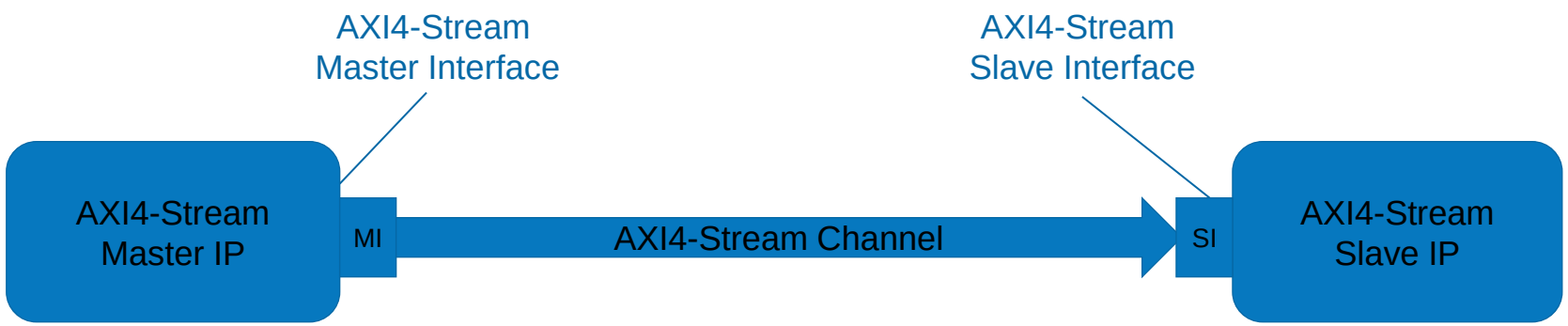


VALID co-incident with READY



READY before VALID
(“Pre-asserted READY”)
Master must **never** wait for READY

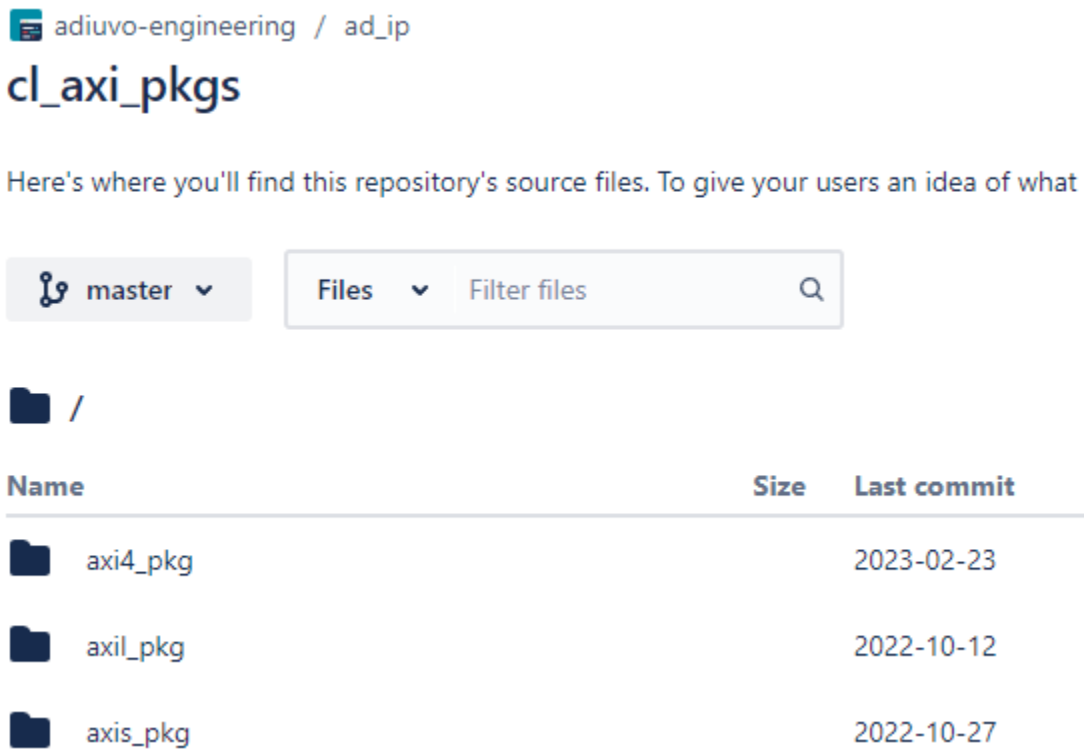
Architecture



- AXI4-Stream is a unidirectional, point-to-point interface standard for exchanging continuous streams of data
  - Like the write data channel of memory mapped AXI4
- Stream interfaces are flexible:
  - Most of the signals are optional so IPs can ‘opt-in’ to only those signaling features that are necessary for their application

Architecture

- AXI is key for development
- Develop a series of libraries / packages which define and implement AXI interfaces and create bus functional models

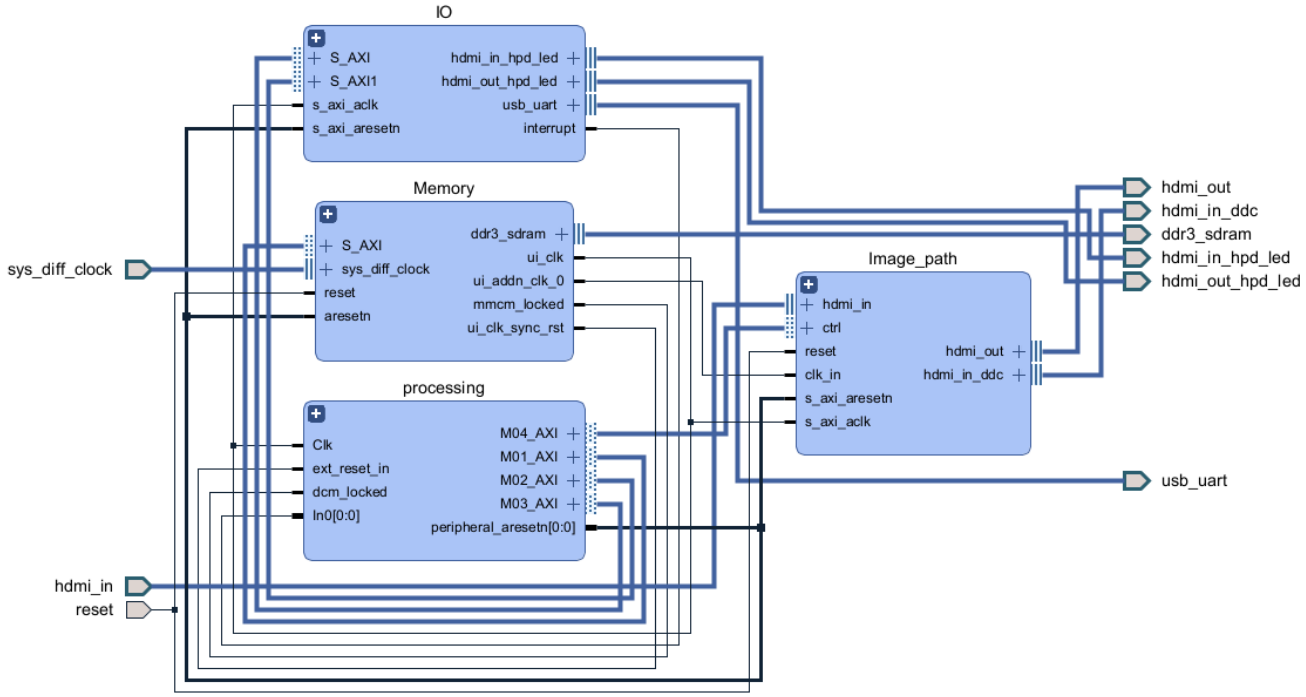


Architecture

Decompose hierarchy into functional blocks:

- Enables blocks (not just IP blocks) to be reused.
  - Write bd TCL

- Common blocks:
  - Processing
  - Image input
  - Image output
  - Algorithm



Architecture

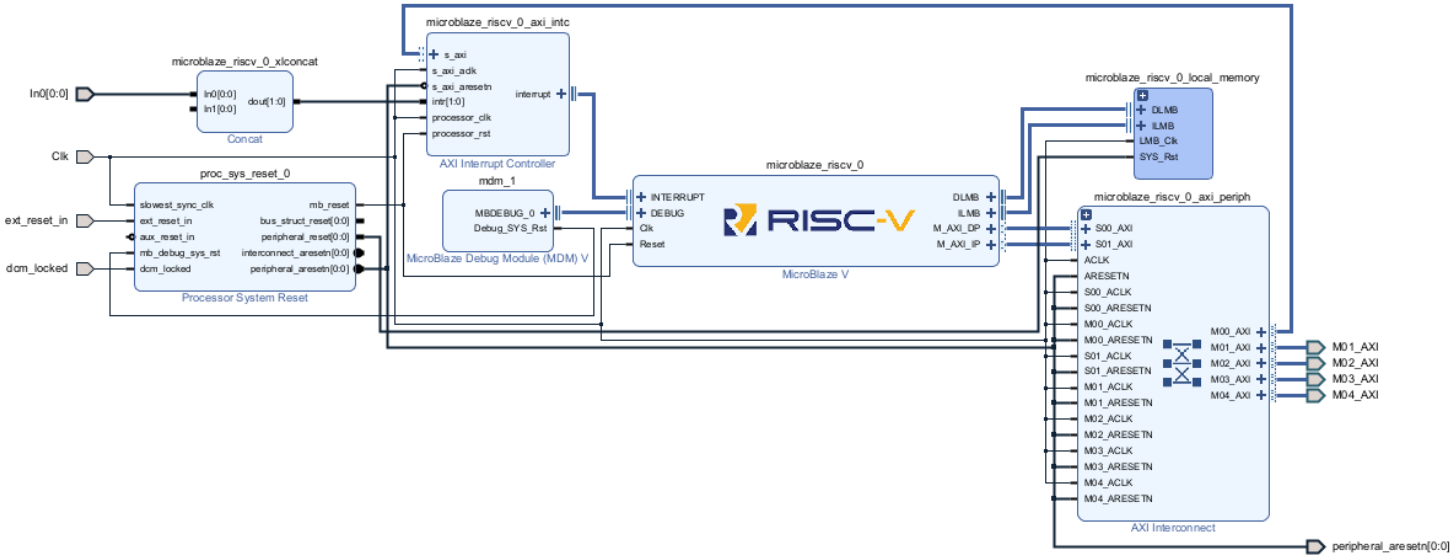
Processing:

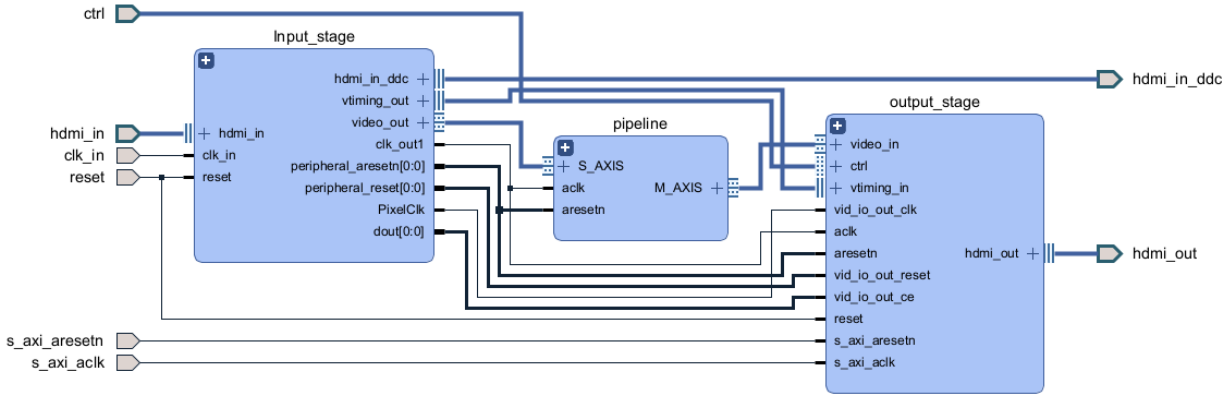
- AMD MicroBlaze™ V Processor and supporting IP modules

Image path:

- Further levels of hierarchy. Input stage, pipeline & output stage

- Makes additions to pipeline simple

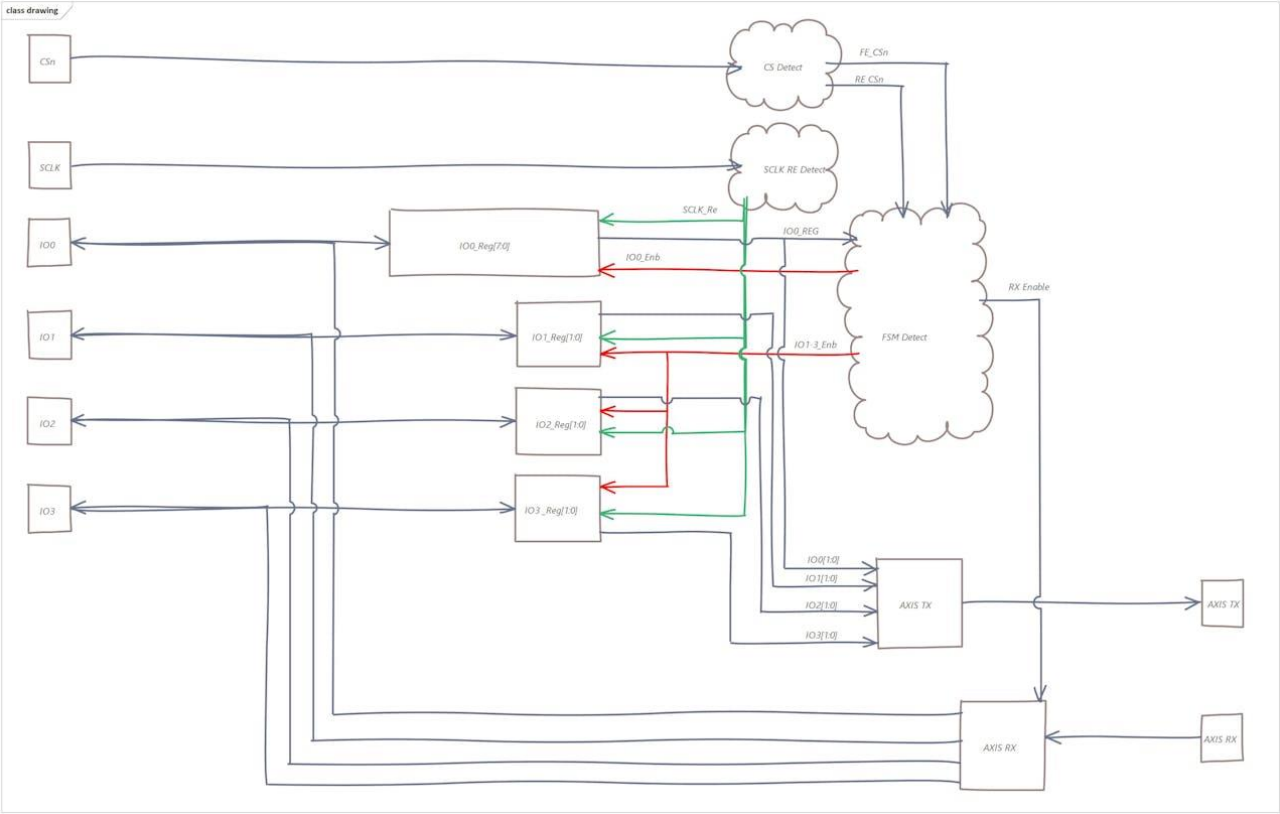




Architecture

When creating custom modules, follow a process:

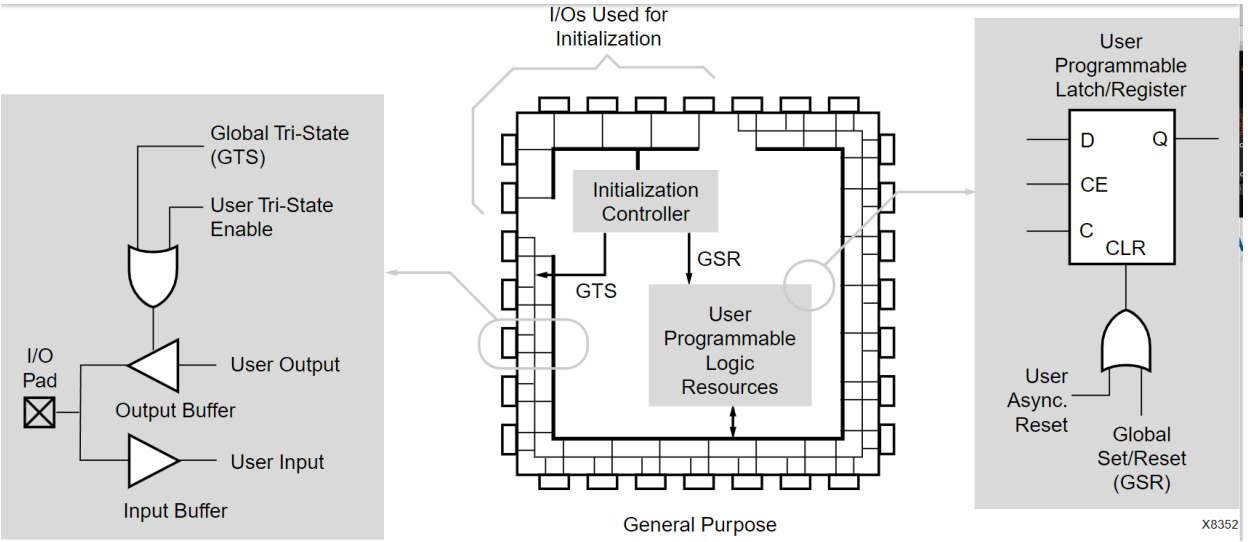
- Define requirements
- Define interfacing
- Create micro architecture
- Create test bench



Control Sets

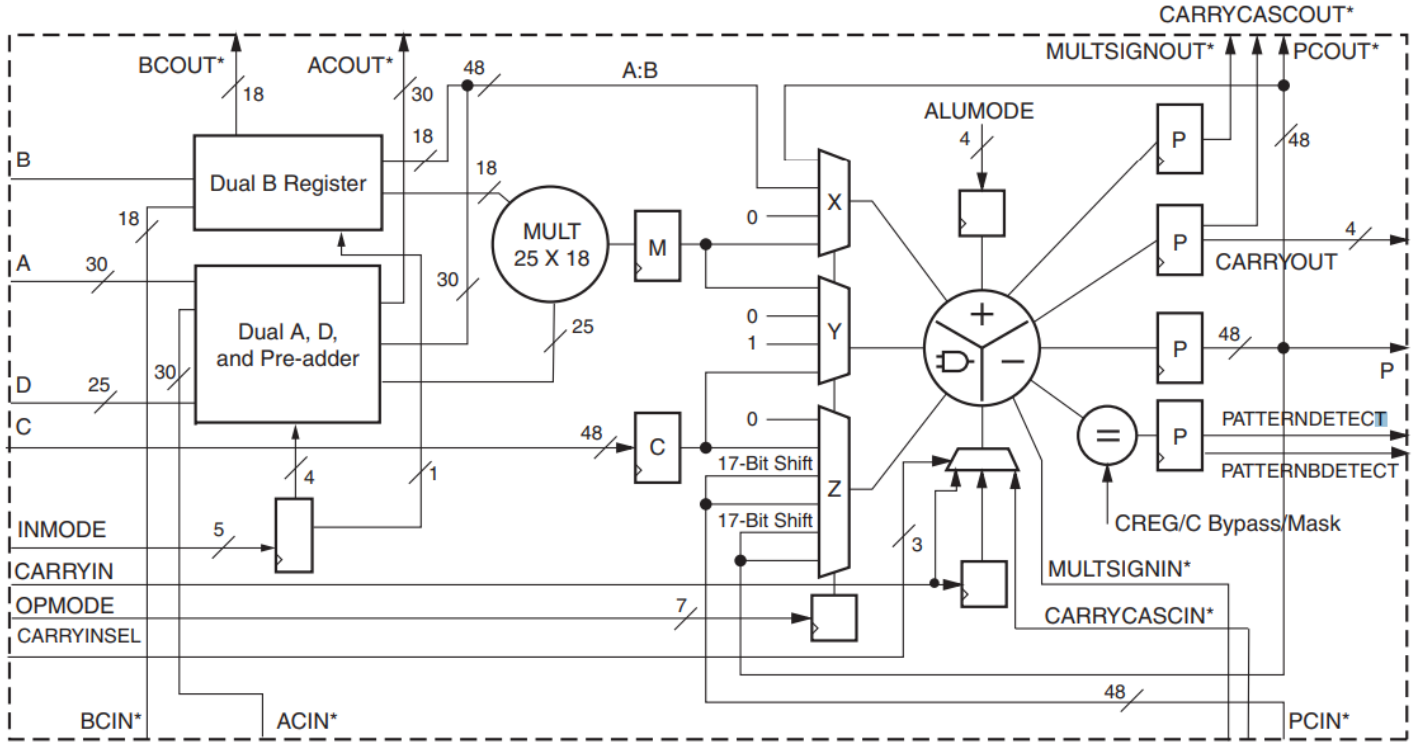
Resets: The question of using a reset or not is important in the AMD ecosystem.

Whenever possible, we should leverage the Global Set Reset, which is applied at the end of configuration.



Reset Example

To achieve maximum DSP48 pipeline, registers are needed to be used. These are dual input registers (B, A, D) along with middle (M), and op register (P).



*These signals are dedicated routing paths internal to the DSP48E1 column. They are not accessible via fabric routing resources.

UG369_c1_01_052109

Figure 2-1: **7 Series FPGA DSP48E1 Slice**

Reset Example

How do we write code that has a big impact on the options, synthesis, and placement the tool can make?

One key aspect is to reset the ‘use’ and ‘sync / async’ reset style.

Getting this wrong will have a big impact on how the synthesis maps to the primitives available.

Reset Example

Asynchronous Reset

```
sync: process(i_clk,i_resetn)
begin

    if i_resetn = '1' then
        s_a_delay_0 <= (others =>'0');
        s_b_delay_0 <= (others =>'0');
        s_a_delay_1 <= (others =>'0');
        s_b_delay_1 <= (others =>'0');
        o_res <= (others =>'0');
    elsif rising_edge(i_clk) then
        s_a_delay_0 <= i_a;
        s_b_delay_0 <= i_b;
        s_a_delay_1 <= s_a_delay_0;
        s_b_delay_1 <= s_b_delay_0;
        o_res <= std_logic_vector(unsigned(s_a_delay_1) *
                 unsigned(s_b_delay_1)) ;
    end if;
end process;
```

Implementation Resources

Utilization **Post-Synthesis** | Post-Implementation

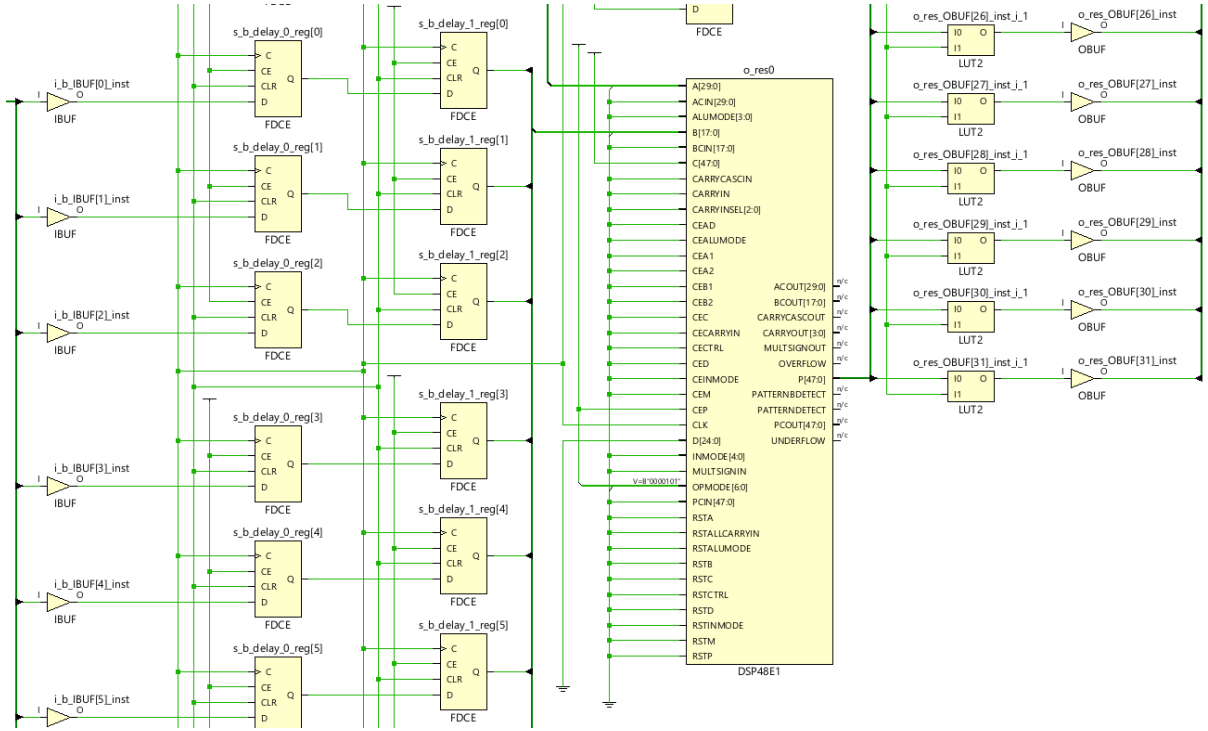
Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	16	3750	0.43
FF	65	7500	0.87
DSP	1	10	10.00
IO	66	100	66.00
BUFG	1	16	6.25

Note, the FF and LUT used for the register implementation

Reset Example

Clear registers are implemented within the fabric:



Reset Example

Synchronous Reset

```
sync: process(i_clk,i_resetn)
begin

    if i_resetn = '1' then
        s_a_delay_0 <= (others =>'0');
        s_b_delay_0 <= (others =>'0');
        s_a_delay_1 <= (others =>'0');
        s_b_delay_1 <= (others =>'0');
        o_res <= (others =>'0');
    elsif rising_edge(i_clk) then
        s_a_delay_0 <= i_a;
        s_b_delay_0 <= i_b;
        s_a_delay_1 <= s_a_delay_0;
        s_b_delay_1 <= s_b_delay_0;
        o_res <= std_logic_vector(unsigned(s_a_delay_1) *
                 unsigned(s_b_delay_1)) ;
    end if;
end process;
```

Implementation Resources

Utilization — **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
DSP	1	10	10.00
IO	66	100	66.00
BUFG	1	16	6.25

Note, no FF or LUT used as the registers are within the DSP

Reset Example

Control Sets

"A control set is the set / reset, clock enable, and clock which drives a SRL, LUTRAM, or register."

Large numbers of different control sets can impact the design performance.

Clock Enables – Clock enables can be very useful in our designs; we can use them to reduce power across the design or as part of our functionality. For example, generating an I^2C or SPI output which runs at a lower clock rate.

Control Sets

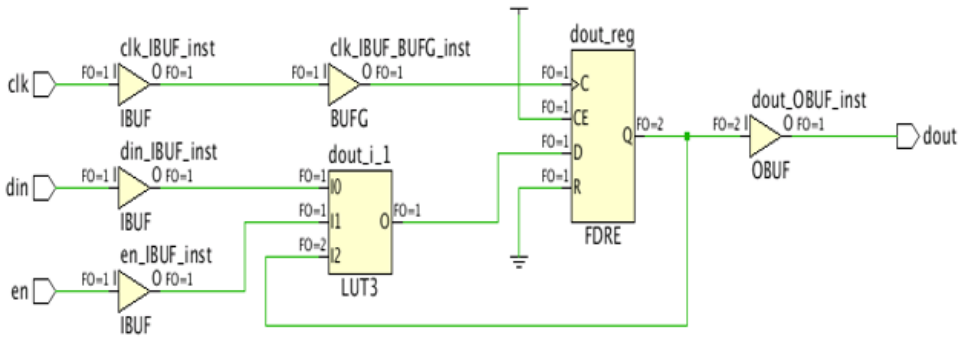
When the synthesis tool identifies a synchronous reset / set or enablement, it examines the load on the cone of identified logic.

If the logic is below the threshold identified in the ***control_set_opt_threshold***, the synthesis engine implements the reset / set using logic gates on the data path instead of using the register inputs.

Control Set

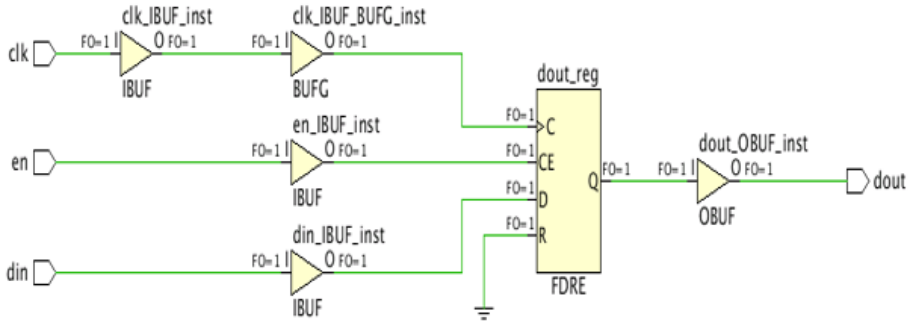
```
module test
(
input clk,
input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```

Data Path Implementation




```
module test
(
input clk,
(* direct_enable = "true" *) input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```

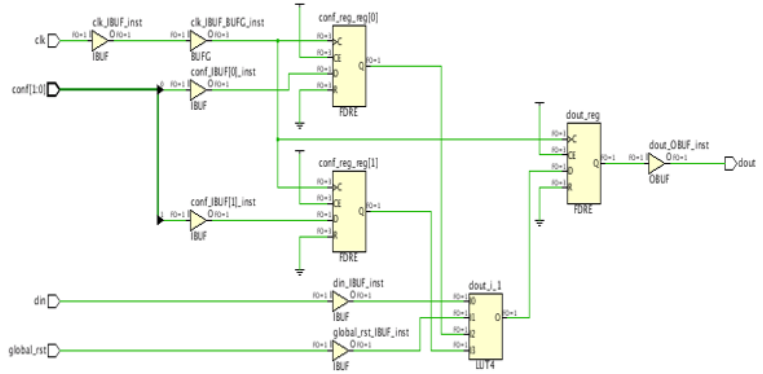
Direct Implementation



Control Sets

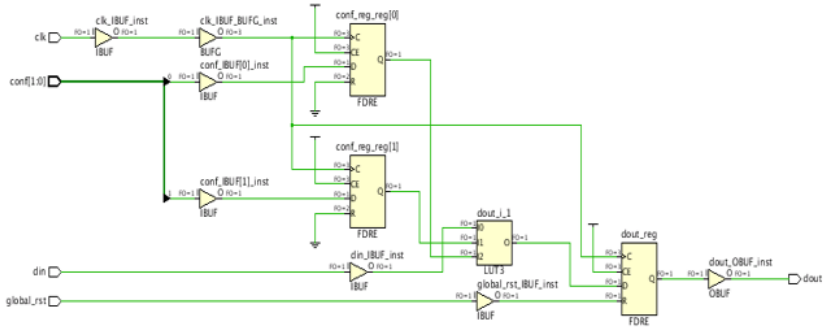
```
module test (
            input clk,
            input global_rst,
            input [1:0] conf,
            input din,
            output reg dout
           );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```

Data Path Implementation



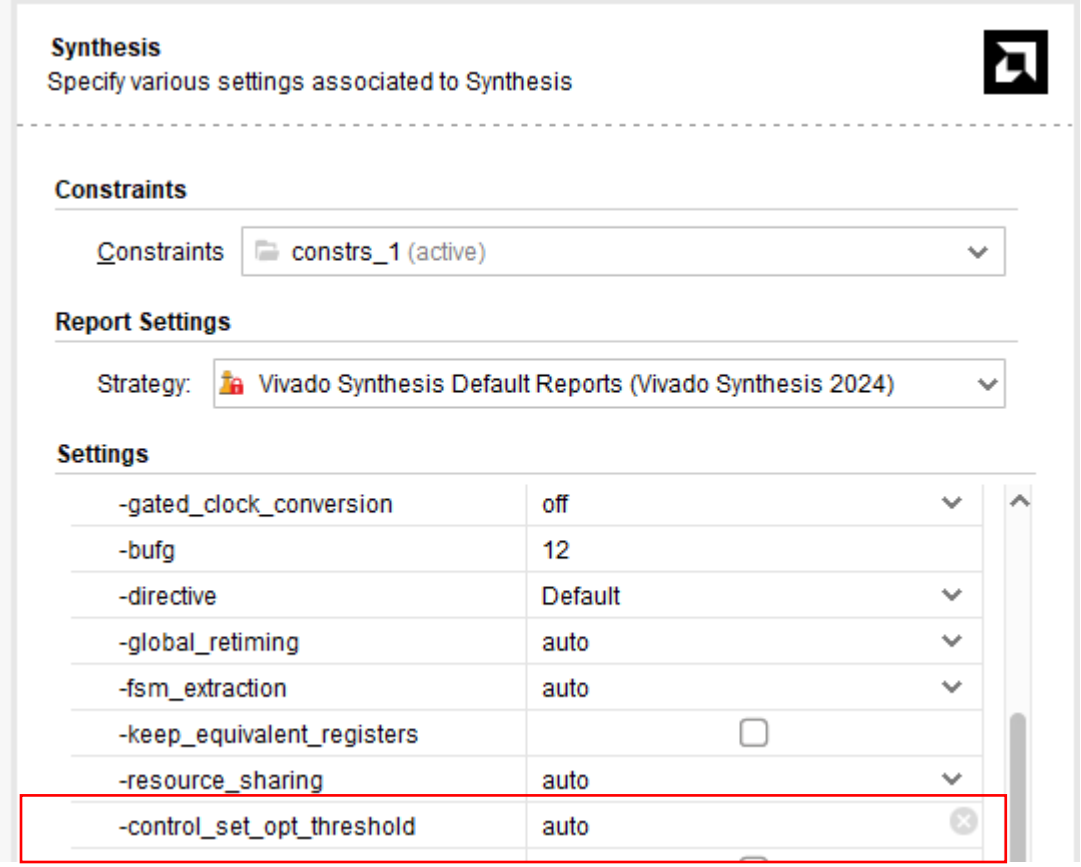

```
module test (
            input clk,
            (* direct_reset = "true" *) input global_rst,
            input [1:0] conf,
            input din,
            output reg dout
           );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```

Direct Path Implementation – We can define reset connection



Control Set

Control set extraction threshold is defined via the synthesis settings:



Performance

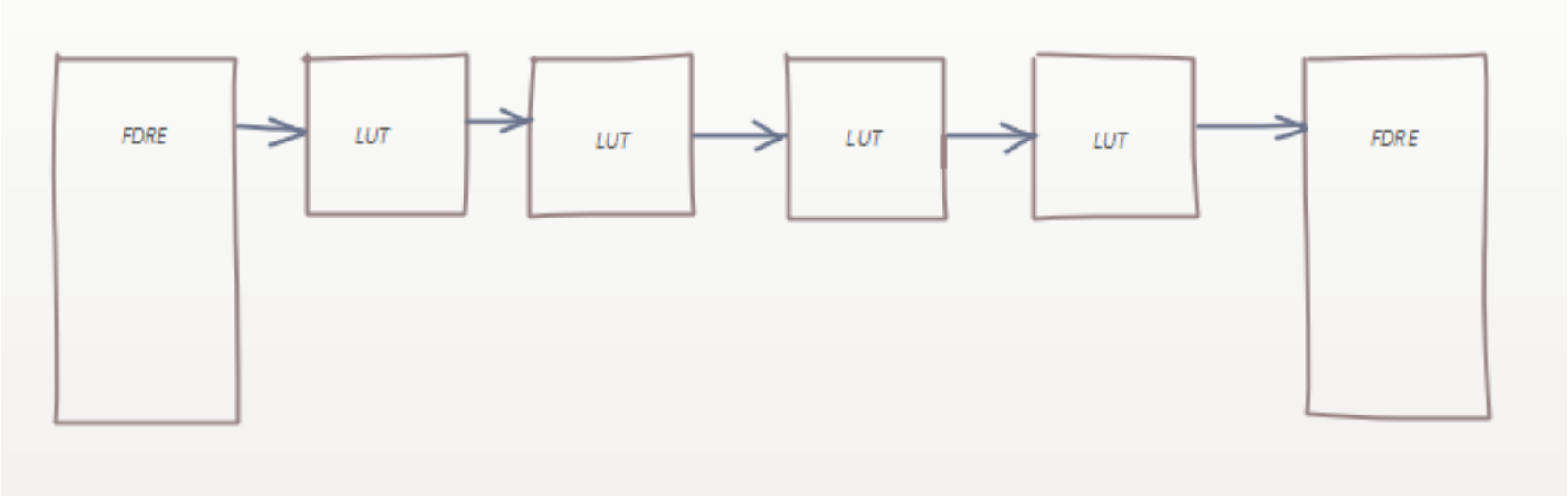
Pipelining – Pipelining allows us to restructure data paths which have several layers of logic.

One good method when writing code is to include several registers before or after your code to enable retiming of the pipeline.

Be sure to include these within the hierarchy as shown in the example below.

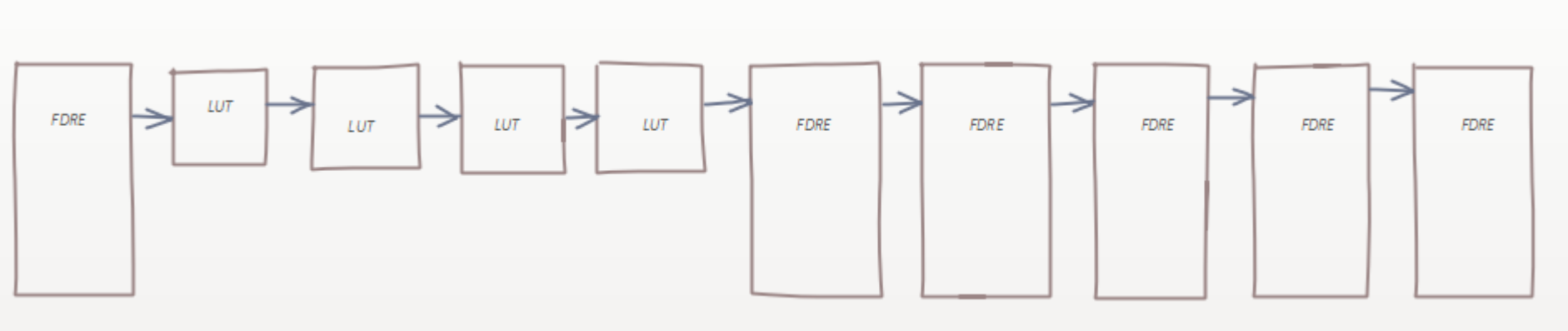
Performance

Original design, significant logic layers, between registers (reduces FMax).



Pipelining

1. Consider the pipelining from day one. Add extra registers before or after the path.
2. Global retiming or block synthesis retiming option, analyzes, and moves registers.
3. If you want more control, retiming_forward / retiming_backwards attributes can be used



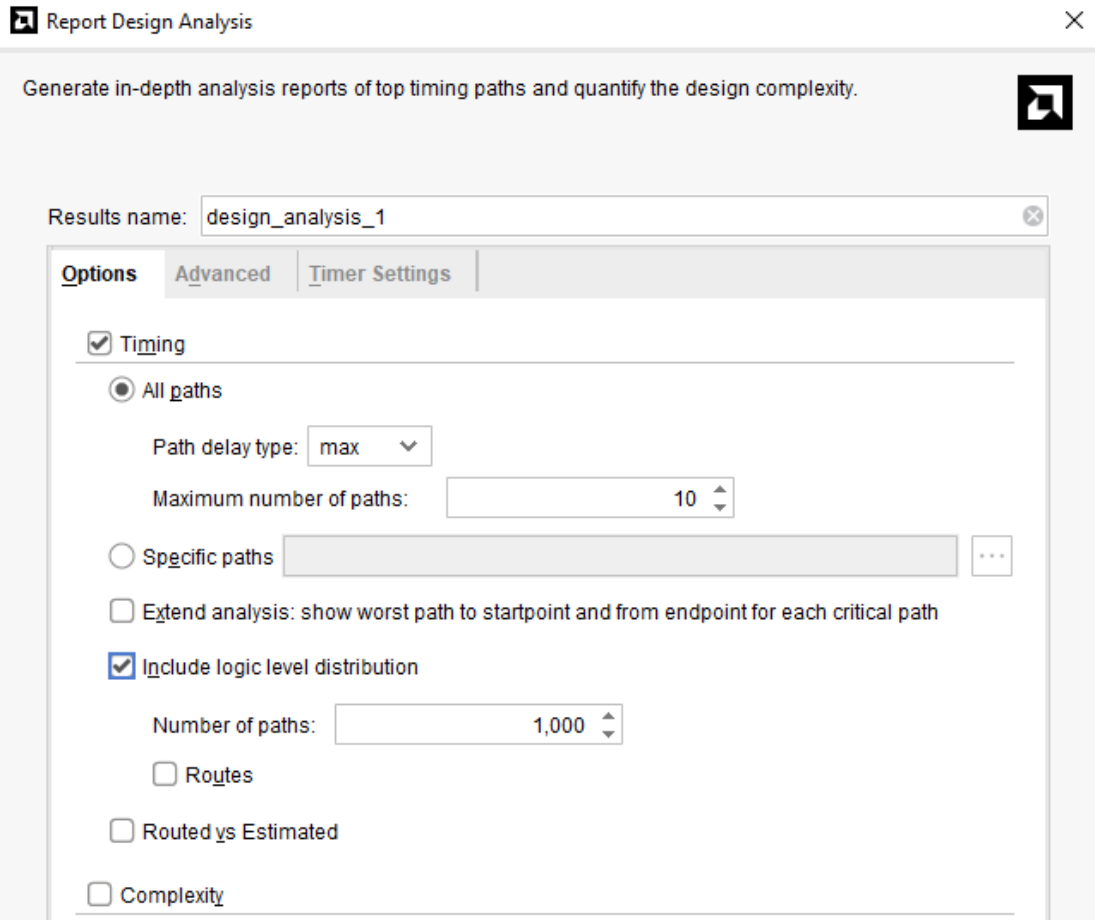
Pipelining Example

Pipelining

How do we tell if pipelining is needed?

We can run the design analysis report which will inform us about the logic level distribution.

Remember to consider the clock frequency.



Pipelining

Will report logic levels depth per clock. For example, below there are 5 paths in the clk_pll_i end point which have 1 logic level.

Knowing this, we can make adjustments. Note the high-level ones are in a processor instantiation.

Tcl Console | Messages | Log | Reports | Design Runs | DRC | **Design Analysis** | Methodology | Power | Timing

- General Information
- Setup Path Characteristics
- Logic Level Distribution
- Congestion
  - Placer Final

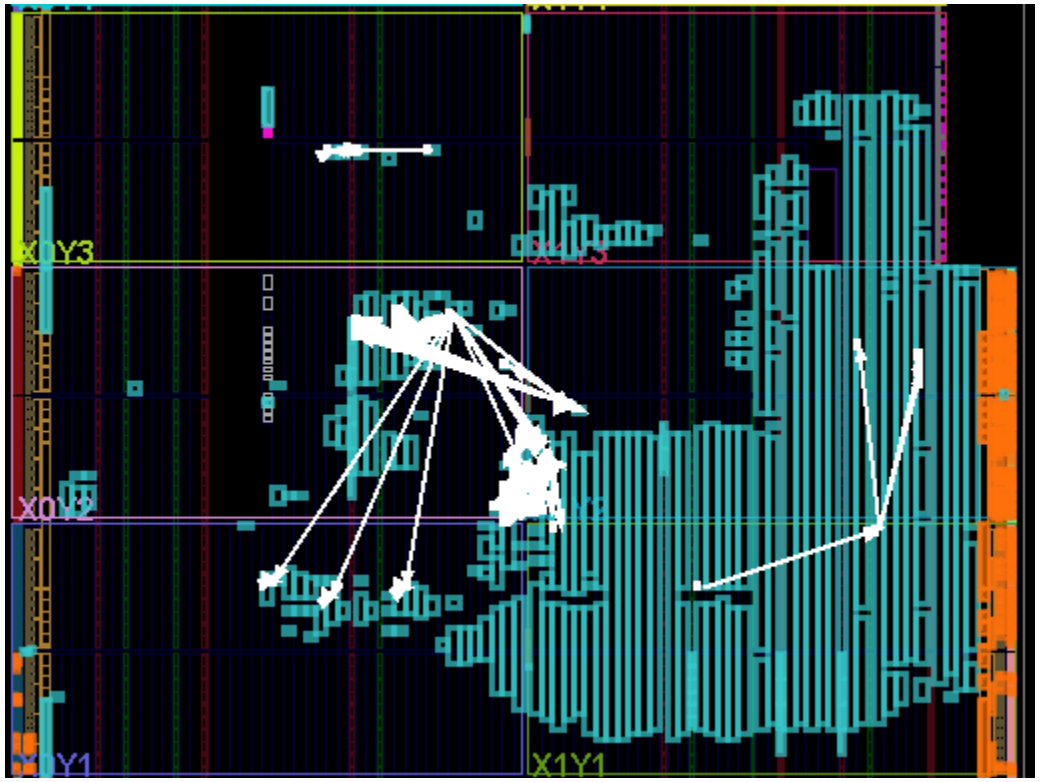
Logic Level Distribution

End Point Clock	Requirement	0	1	2	3	4	5	9	11	12	13	14	19
clk_pll_i	10.000ns	0	5	1	0	0	0	174	1	256	179	8	3
(none)	5.000ns	2	0	0	0	0	0	0	0	0	0	0	0
PixelClk_int	0.833ns	10	169	56	12	12	12	0	0	0	0	0	0
clk_out1_video_plat_clk_wiz_0_0	6.667ns	0	0	1	0	0	0	0	0	0	0	0	0

Pipelining

Selecting the logic levels will show the results in the floor plan.

This enables investigation, allowing us to look at paths with 0 logic levels. Also to consider adjustments (e.g. splitting the design).



Performance

Along with moving the pipelining forward and backward, there are some tricks we can also do.

- Balance latency:
  - Consider use of control signals to enable large data path pipelines if possible

- Code to use SRL in place of FF:
  - Use SRL Style attributes to control implementation style
  - Ensure reset is not used, (prevents mapping to SRL)

- `(* srl_style = "srl" *)`
- `(* srl_style = "reg_srl" *)`
- `(* srl_style = "srl_reg" *)`
- `(* srl_style = "reg_srl_reg" *)`

Performance

Auto Pipelining – Enables the tool to implement pipelining to meet requirements.

Identifies the number of required stage and locations.

Can be applied to AXI register slice or via HDL attributes / XDC constraints to custom RTL.

Performance

Several attributes used to implement auto pipelining:

- Autopipeline_group – Defines a group of signals which must have auto inserted balanced pipelining inserted
- Autopipeline_Include – Defines a signal to include
- Auto Pipline_Limit – Defines the number of stages allowable from 0 to 24
- Auto Pipline_Module – Enables modules with auto pipelining to be instantiated several times in the design

Auto Pipelining Example

Wrap Up

- Developing FPGA solutions that are scalable, reuseable, and meet timing requirements is not as challenging as one might think.

- Consider the approach outline:
- Architecture:
  - Reuse IP whenever possible
  - Leverage standard interfaces
- Control Sets:
  - Minimize and extract
- Pipelining:
  - Code for pipelining
  - Leverage auto insertion if possible

ADIUVO
ENGINEERING AND TRAINING, LTD.

AMD sponsored this webinar, including engineering hours. AMD, and the AMD Arrow logo, MicroBlaze, Vivado, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

ADIUVO

ENGINEERING AND TRAINING, LTD.

www.adiuvoengineering.com

info@adiuvoengineering.com