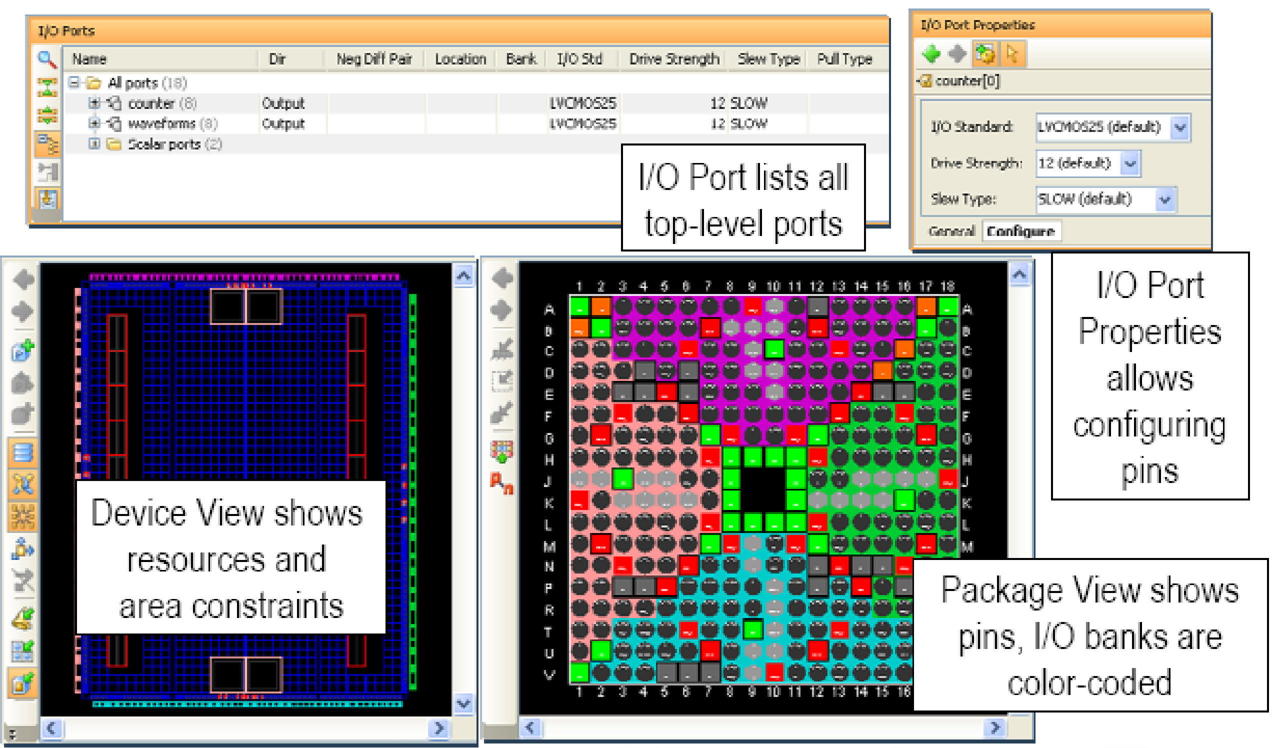
FPGAs 2

Some of the Slides picked from Xilinx Educational Resources

Pin Assignment

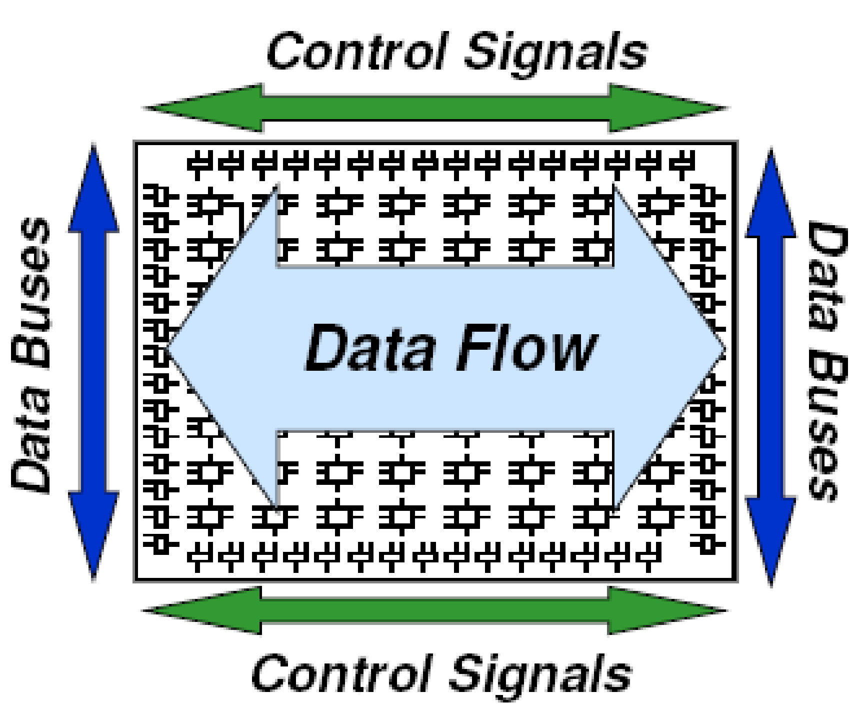
- The process of assigning design ports to FPGA IO pins, requires:
- Configuring direction (input/output/inout)
- Defining signaling standard for each of the pins

Pin Assignment



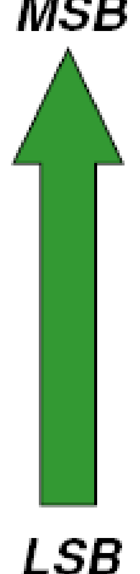
I/O Layout Guidelines

- I/O for control signals on the top or the bottom
  - Signals are routed vertically
- I/O for data buses on the left or the right
  - FPGA architecture favors horizontal data flow



Data Bus Layout

- Arithmetic functions with more than five bits use carry logic
- Carry chains require specific vertical orientation



Reading Reports

- After you have implemented your design, how can you tell whether the implementation was successful?
- First and foremost, how do you define a successful design?
- Answer: A successful design:
  - Fits into the device
  - Achieves performance goals

Device Utilization Summary

Get quick access to used and available resources through the FPGA Design Summary

- Design Overview
  - Summary
  - IOB Properties
  - Control Set Information
  - Module Level Utilization
  - Timing Constraints
  - Pinout Report
  - Clock Report
  - Static Timing
- Errors and Warnings
  - Synthesis Messages
  - Translation Messages
  - Map Messages
  - Place and Route Messages
  - Timing Messages
  - Bitgen Messages
  - All Current Messages
- Detailed Reports
  - Synthesis Report
  - Translation Report
  - Map Report
  - Place and Route Report
  - Post-PAR Static Timing Rep...
  - Power Report

Design Properties
- [x] Enable Enhanced Design Summary
- [ ] Display Incremental Messages
- [ ] Enable Message Filtering

Optional Design Summary Contents
- [ ] Show Clock Report
- [ ] Show Failing Constraints
- [ ] Show Warnings

time_const Project Status (02/10/2009 - 14:43:39)			
Project File:	time_const.ise	Current State:	Placed and Routed
Module Name:	loopback	• Errors:	No Errors
Target Device:	3s500efg320-4	• Warnings:	151 Warnings
Product Version:	ISE 11.1 - Full	• Routing Results:	All Signals Completely Rout
Design Goal:			All Constraints Met
Design Strategy:			0 (Setup: 0, Hold: 0, Comp

Design Utilization Summary

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	141	9,312	1%
Number of 4 input LUTs	281	9,312	3%
Number of occupied Slices	162	4,656	3%
Number of Slices containing only related logic	162	162	100%
Number of Slices containing unrelated logic	0	162	0%
Total Number of 4 input LUTs	291	9,312	3%
Number used as logic	177		
Number used as a route-thru	10		
Number used for Dual Port RAMs	16		
Number used for 32x1 RAMs	52		
Number used as Shift registers	36		
Number of bonded IOBs	21	232	9%

Device Utilization Summary

Get quick access to used and available resources from the Map report

Design Overview
- Summary
- IOB Properties
- Control Set Information
- Module Level Utilization
- Timing Constraints
- Pinout Report
- Clock Report
- Static Timing

Errors and Warnings
- Synthesis Messages
- Translation Messages
- Map Messages
- Place and Route Messages
- Timing Messages
- Bitgen Messages
- All Current Messages

Detailed Reports
- Synthesis Report
- Translation Report
- Map Report
- Place and Route Report
- Post-PAR Static Timing Rep...
- Power Report

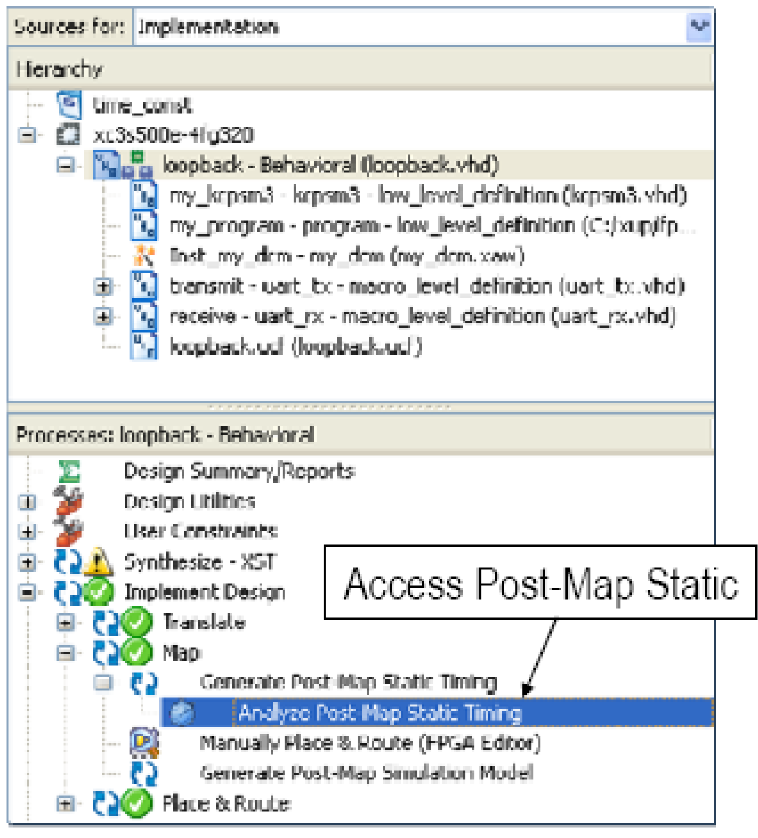
- Section 6: IOB Properties
- Section 7: RPMs
- Section 8: Guide Report
- Section 9: Area Group and Partition ...
- Section 10: Modular Design Summary
- Section 11: Timing Report

```
Release 11.1 Map L.30 (nt)
Xilinx Mapping Report File for Design 'loopback'

Design Information
------------------
Command Line   : map -ise time_const.ise -intstyle ise -p xc3s500e-fg320-4 -cm
area -ir off -pr off -c 100 -o loopback_map.ncd loopback.ngd loopback.pcf
Target Device  : xc3s500e
Target Package : fg320
Target Speed   : -4
Mapper Version : spartan3e -- $Revision: 1.51 $
Mapped Date    : Tue Feb 10 14:42:47 2009

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Logic Utilization:
  Number of Slice Flip Flops:            141 out of   9,312    1%
  Number of 4 input LUTs:                281 out of   9,312    3%
Logic Distribution:
  Number of occupied Slices:             162 out of   4,656    3%
    Number of Slices containing only related logic:     162 out of     162 100%
    Number of Slices containing unrelated logic:          0 out of     162   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:          291 out of   9,312    3%
    Number used as logic:                177
    Number used as a route-thru:          10
    Number used for Dual Port RAMs:       16
      (Two LUTs used per Dual Port RAM)
    Number used for 32x1 RAMs:            52
      (Two LUTs used per 32x1 RAM)
    Number used as Shift registers:       36
```

Access the Map report through the Detailed Reports

Post-Map Static Timing Report

Evaluate logic delays to see if you should proceed to place & route



Analyze Logic Timing

Look at critical paths to determine if timing is reasonable

Select critical paths

Source and destination registers of path illustrated

Detailed listing of logic and estimated routing delays

Total delay

Post-Place & Route Static Timing Report

Determine if the constraints were met

Sources for: Implementation

Hierarchy

- time_const
- xc3s500e-4fg320
  - loopback - Behavioral (loopback.vhd)
    - my_kcpsm3 - kcpsm3 - low_level_definition (kcpsm3.vhd)
    - my_program - program - low_level_definition (C:/xup/fp...
    - Inst_my_dcm - my_dcm (my_dcm.xaw)
    - transmit - uart_tx - macro_level_definition (uart_tx.vhd)
    - receive - uart_rx - macro_level_definition (uart_rx.vhd)
    - loopback.ucf (loopback.ucf)

Processes: loopback - Behavioral

- Design Summary/Reports
- Design Utilities
- User Constraints
- Synthesize - XST
- Implement Design
  - Translate
  - Map
  - Place & Route
    - Generate Post-Place & Route Static Timing
      - Analyze Post-Place & Route Static Timing
    - Generate Primetime Netlist
  - Analyze Timing / Floorplan Design (PlanAhead)

Access Post-Place & Route Static Timing report

Timing Summary

Provides statistics on average routing delays and performance versus constraints

Access Place & Route report through Detailed Reports

FPGA Design Summary
- Design Overview
- Errors and Warnings
- Detailed Reports
  - Synthesis Report
  - Translation Report
  - Map Report
  - Place and Route Report
  - Static Timing Report
  - Bitgen Report
- Show or Hide Reports...

```
+-----------------------------------+--------------+-----------------------------+-
|                                   |    Period    |        Actual Period        |
|            Constraint             | Requirement  |-------------+---------------|-
|                                   |              |   Direct    |  Derivative   |
+-----------------------------------+--------------+-------------+---------------+-
|TS_clk                             |     10.000ns|          N/A|         2.025ns|
| TS_Inst_clockgen_CLK0_BUF         |     10.000ns|      2.025ns|             N/A|
+-----------------------------------+--------------+-------------+---------------+-

All constraints were met.
INFO:Timing:2761 - N/A entries in the Constraints list may indicate that the
   constraint does not cover any paths or that it has no requested value.
```

Period Requirement for global clock is 10 ns

Actual Period of 2.025 ns

Timing Constraints

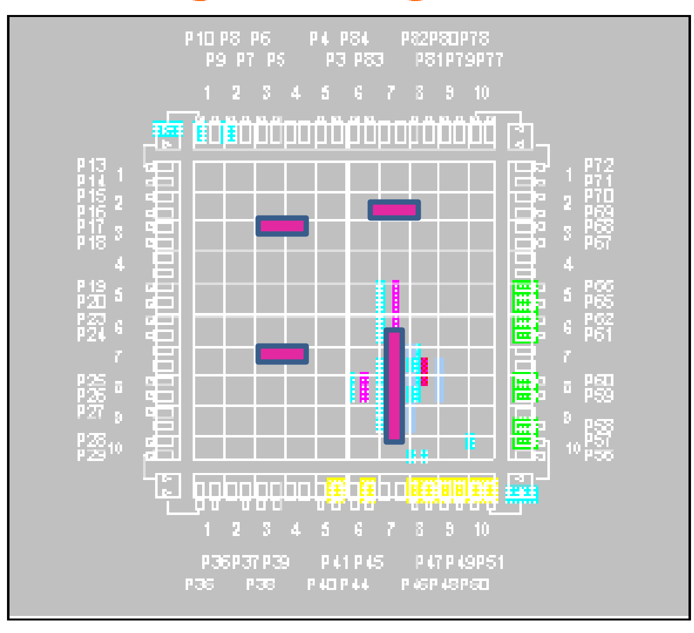
What effects do timing constraints have on your project?

- The implementation tools do not attempt to find the Place & Route that will obtain the best speed
  - Instead, the implementation tools try to meet your performance expectations
- Performance expectations are communicated with timing constraints
  - Timing constraints improve the design performance by placing logic closer together so that shorter routing resources can be used
  - Note: The Constraints Editor refers to the Xilinx Constraints Editor

Design Without and With Constraints

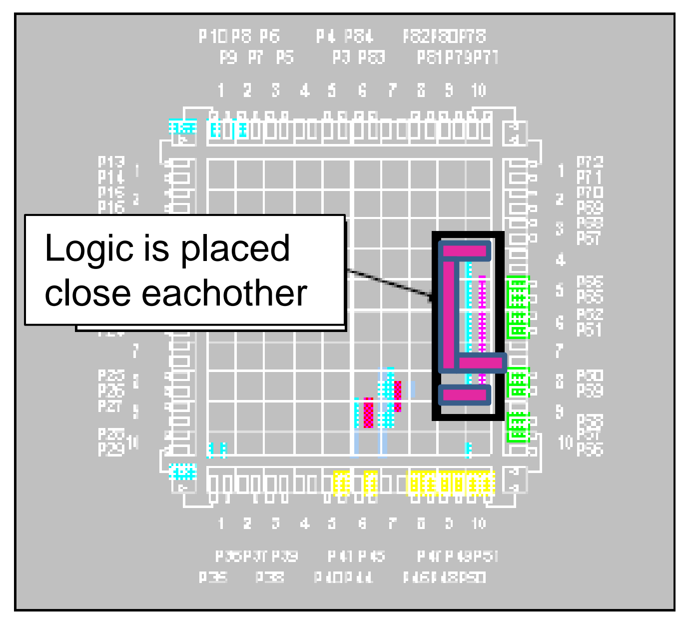
Logic Placement Can Be Very Different

Without global timing constraints



Logic is placed randomly

With global timing constraints

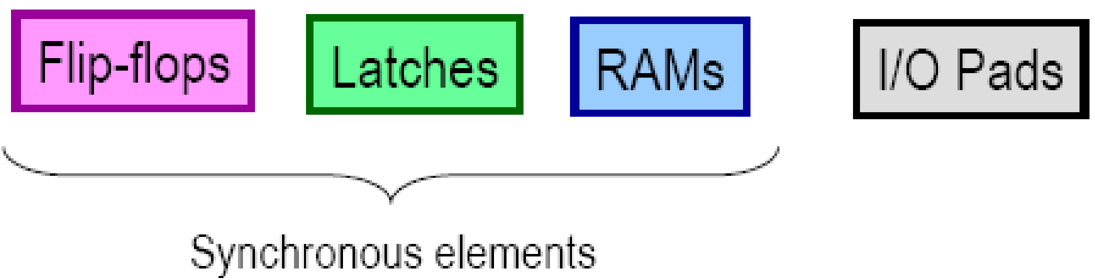


Logic is placed to result in a faster design

Create a Timing Constraint

Create groups of path end points and specify a timing requirement between groups

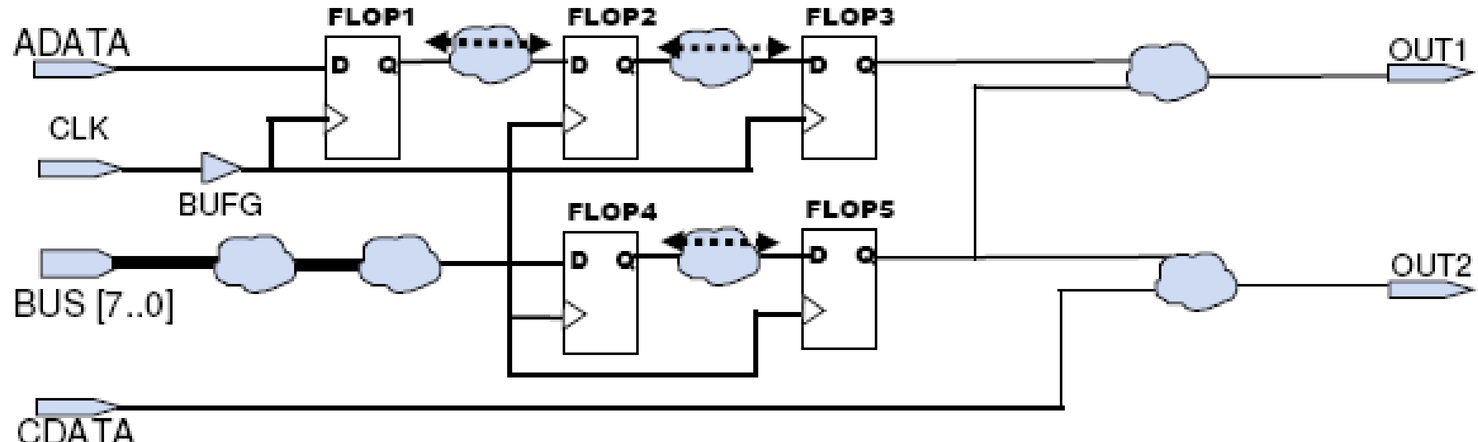
Step 1: Create groups of path end points



Step 2: Specify a timing requirement between the groups

Period Constraints

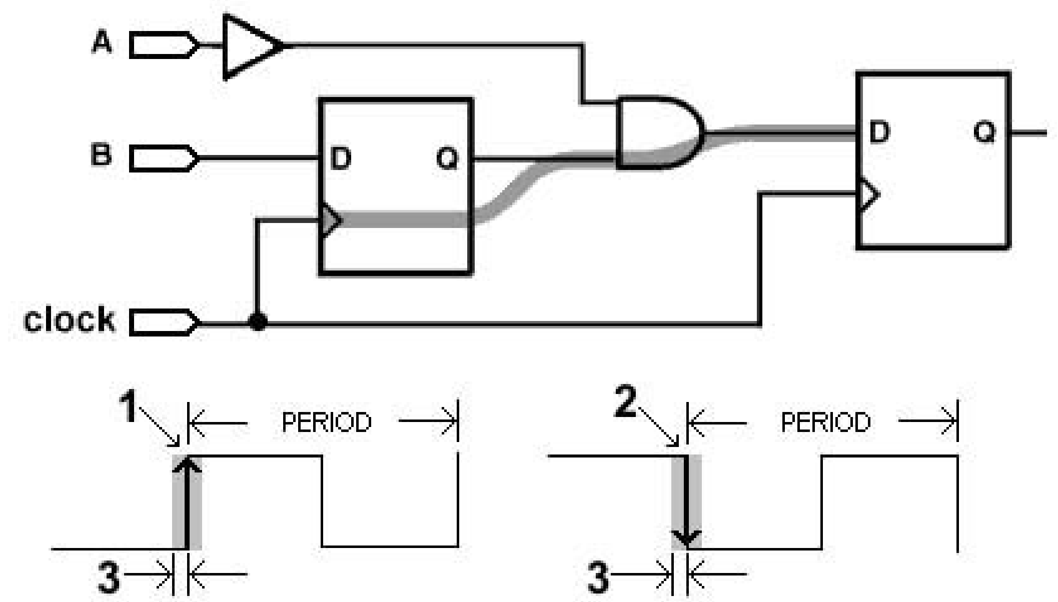
Cover Paths Between Synchronous Elements



PERIOD Constraint

Use the Most Accurate Timing Information

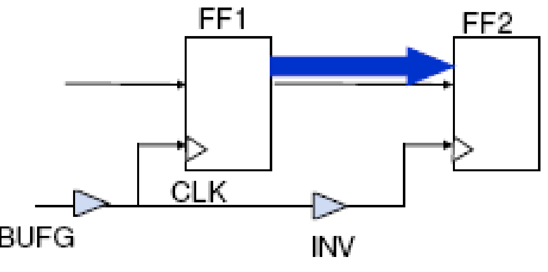
- ✓ Clock skew between the source and destination flip-flops
- ✓ Synchronous elements clocked on the negative edge
- ✓ Unequal clock duty cycles
- ✓ Clock input jitter



PERIOD Constraint

Calculation takes into account inverted clock edges

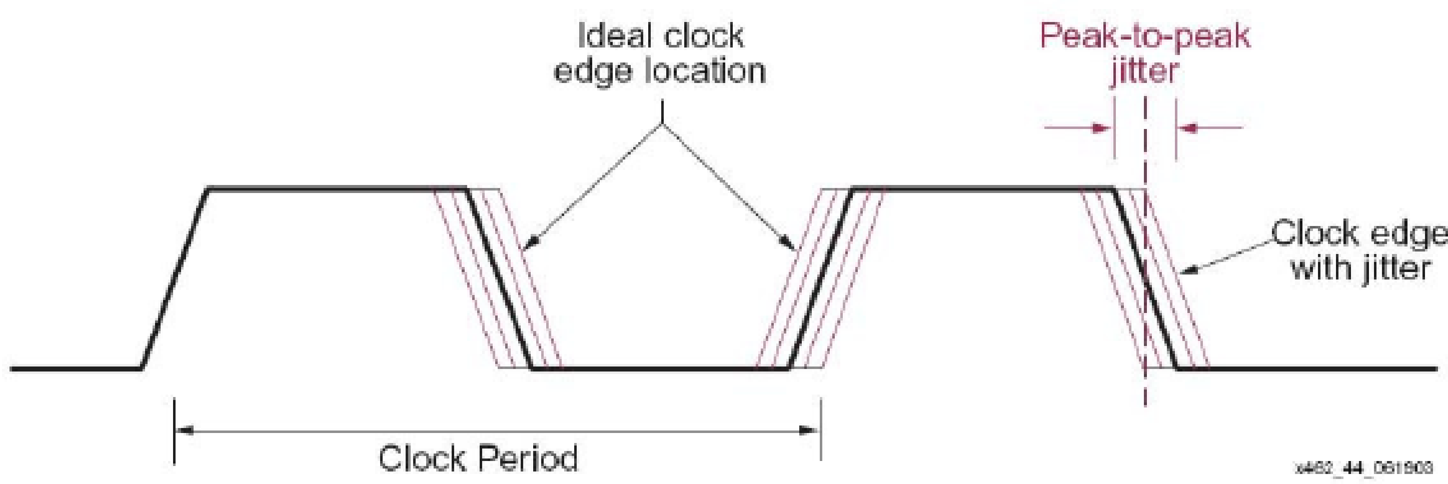
- Assume:
  - 50 percent duty cycle on CLK
  - PERIOD constraint of 10 ns
  - Because FF2 will be clocked on the falling edge of CLK, the path between the two flip-flops will be constrained to 50 percent of 10 ns = 5 ns



Period Constraint

Clock uncertainty is automatically accounted for in global constraint calculations

Clock jitter is a form of clock uncertainty



Period Constraint

Timing Analyzer calculation accounts for most accurate timing information

Timing constraint: TS_Inst_clockgen_CLK0_BUF_0 = PERIOD TIMEGRP "Inst_clockgen_CLK0_BUF_0" TS_clk HIGH 50% INPUT_JITTER 0.001 ns;

10 paths analyzed, 4 endpoints analyzed, 0 failing endpoints

0 timing errors detected. (0 setup errors, 0 hold errors)

Minimum period is 2.016ns.

Slack: 7.984ns (requirement - (data path - clock path skew + uncertainty))

Equation takes into account data path delay, clock skew and clock uncertainty

Source: bincount/BU2/U0/q_i_0 (FF)	clk: ckgen_out rising at 0.000ns
Destination: bincount/BU2/U0/q_i_3 (FF)	clk: ckgen_out rising at 10.000ns

Requirement	Data Path Delay	Clock Path Skew:	Clock Uncertainty
10.000ns	2.015ns (Levels of Logic = 2)	0.000ns	0.001ns

Clock Uncertainty: 0.001ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE	
Total System Jitter (TSJ):	0.000ns
Total Input Jitter (TIJ):	0.001ns
Discrete Jitter (DJ):	0.000ns
Phase Error (PE):	0.000ns

Maximum Data Path: bincount/BU2/U0/q_i_0 to bincount/BU2/U0/q_i_3

Delay type	Delay(ns)	Logical Resource
Tcko	0.370	bincount/BU2/U0/q_i_0
net (fanout=2)	0.246	dout_0_OBUF

Offset Constraints

Constrains I/O Pads To/From Synchronous Elements relative to associated clock signal

OFFSET IN

OFFSET OUT

ADATA

CLK

BUFG

BUS [7..0]

CDATA

FLOP

D Q

OUT1

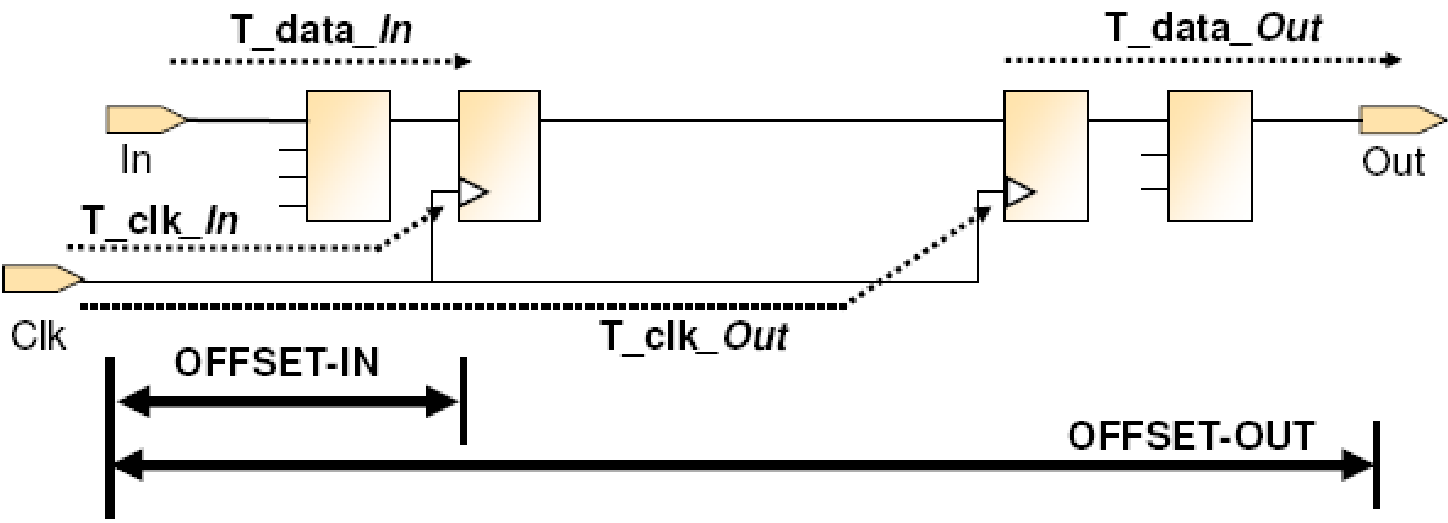
OUT2

Combinatorial Logic

Offset Constraints

Accounts for Clock Delay

- OFFSET IN = T_data_In - T_clk_In
- OFFSET OUT = T_data_Out + T_clk_Out



Offset Constraints

Timing Analyzer calculation accounts for most accurate timing information

Timing constraint: OFFSET = IN 5 ns BEFORE COMP "clk";

30 paths analyzed, 8 endpoints analyzed, 0 failing endpoints

0 timing errors detected. (0 setup errors, 0 hold errors)

Minimum allowable offset is 3.693ns.

Slack: 1.307ns **(requirement - (data path - clock path - clock arrival + uncertainty))**

Source: load (PAD)	
Destination: bincount/BU2/U0/q_i_3 (FF)	**clk:** clkgen_out rising at 0.000ns

Requirement	Data Path Delay	Clock Path Del
5.000ns	2.877ns (Levels of Logic = 2)	-0.815ns (Levels

Clock Uncertainty: 0.001ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE	
Total System Jitter (TSJ):	0.000ns
Total Input Jitter (TIJ):	0.001ns
Discrete Jitter (DJ):	0.000ns
Phase Error (PE):	0.000ns

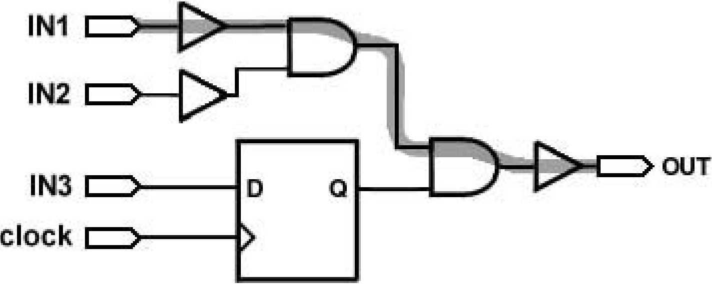
Maximum Data Path: load to bincount/BU2/U0/q_i_3

Delay type	Delay(ns)	Logical Resource
Tiopi	0.736	load load_IBUF
net (fanout=5)	1.147	load_IBUF

Equation takes into account clock path, clock arrival, and clock uncertainty

Pad to Pad Constraints

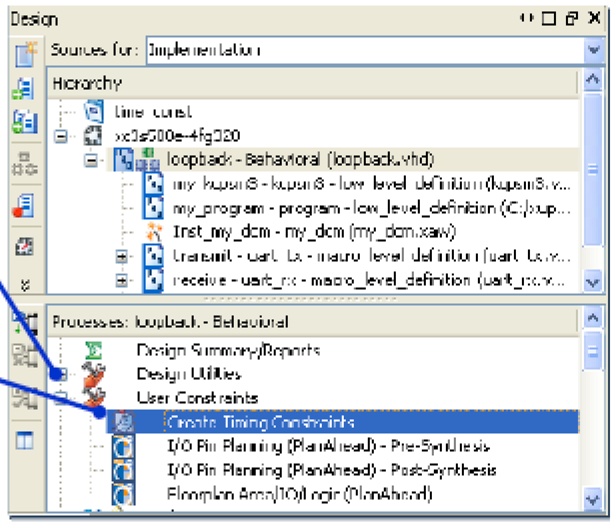
Covers Purely Combinatorial Paths that start and end at I/O pads



Access the Constraints Editor

Enter constraints in the Constraints Editor GUI

- Expand **User Constraints** in the Processes for Source window
- Double-click **Create Timing Constraints**



FPGA Design Techniques

Simple Coding Steps Yield 3x Performance

- Use pipeline stages—more bandwidth
- Use synchronous reset—better system control
- Use Finite State Machine optimizations
- Use inferable resources
  - Multiplexer
  - Shift Register LUT (SRL)
  - Block RAM, LUT RAM
  - Cascade DSP
- Avoid high-level constructs (loops, for example) in code
  - Many synthesis tools produce slow implementations

FPGA Design Techniques

Synthesis Guidelines

- Use timing constraints
  - Define tight but realistic individual clock constraints
  - Put unrelated clocks into different clock groups
- Use proper options and attributes
  - Turn off resource sharing
  - Move flip-flops from IOBs closer to logic
  - Turn on FSM optimization
  - Use the *retiming* option

FPGA Design Techniques

Basic Performance Tips

- Avoid high-level loop constructs
  - Synthesis tools may not produce optimal results
- Avoid nested if-then-else statements
  - Most tools implement these in parallel; however, multiple nested if-then-else statements can result in priority encoded logic
- Use case statements for large decoding
  - Rather than if-then-else
- Order and group arithmetic and logical functions and operators
  - A <= B + C + D + E; should be: A <= (B + C) + (D + E)
- Avoid inadvertent latch inference
  - Cover all possible outputs in every possible branch
    - Easily done by making default assignments before if-then-else and case

FPGA Design Techniques
Instantiation versus Inference

- Xilinx recommends inferring FPGA resources whenever possible
  - Inference makes your code more portable
- In some cases, the synthesis tool is unable to infer or fails to infer resources
  - You can instantiate resources when you must dictate exactly which resource is needed

- Can be inferred by all synthesis tools:
  - Shift register LUT (SRL16/ SRLC16)
  - F5, F6, F7, and F8 multiplexers
  - Carry logic
  - MULT_AND
  - MULT18x18/MULT18x18S
  - 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 (ROM/RAM)
  - Global clock buffers (BUFGCE, BUFGMUX, BUFGDLL)
- Cannot be inferred by any synthesis tools:
  - SelectIO (differential) standard
  - Output DDR registers
  - DCM
  - Gigabit transceivers

FPGA Design Techniques

Synthesis Options

- There are many synthesis options that can help you obtain your performance and area objectives:
  - Timing-Driven Synthesis
  - FSM Extraction
  - Retiming
  - Register Duplication
  - Hierarchy Management
  - Resource Sharing
  - Physical Optimization
- XST Constraints are entered in a .xcf file

Synthesis Options

Timing-Driven Synthesis

- Supported in various synthesis tools: Synplify, Precision, and XST
- Timing-driven synthesis uses performance objectives to drive the optimization of the design
  - Based on your performance objectives, the tools will try several algorithms to attempt to meet performance while keeping the amount of resources in mind
  - Performance objectives are provided to the synthesis tool via timing constraints

Synthesis Options

FSM Extraction

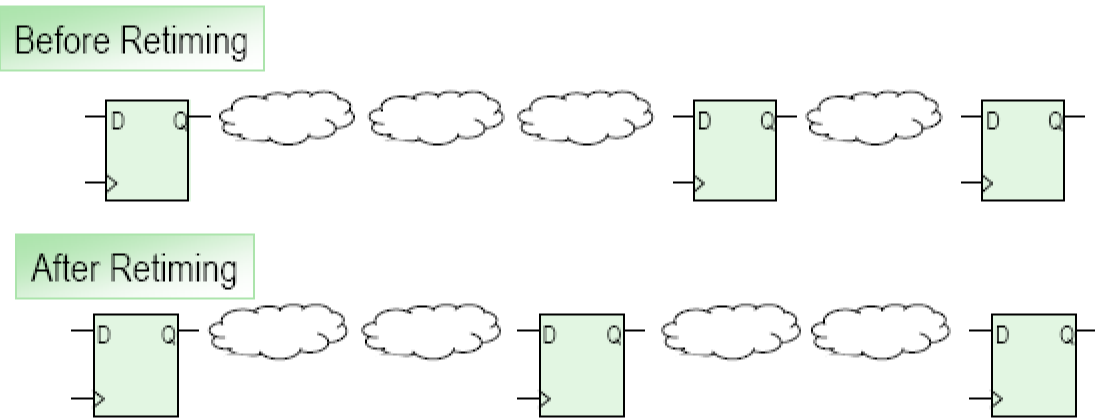
- Finite State Machine (FSM) extraction optimizes your state machine by re-encoding and optimizing your design based on the number of states and inputs
  - By default, the tools will use FSM extraction

- Safe state machines
  - By default, the synthesis tools will remove all decoding for illegal states
    - Even if you include VHDL “when others” or Verilog “default” cases
  - The “safe” FSM implementation option must be turned on

Synthesis Options

Retiming

- Synplify, Precision, and XST software
- Retiming: The synthesis tool automatically tries to move register stages to balance combinatorial delay on each side of the registers



Synthesis Options

Register Duplication

- Synplify, Precision, and XST software
- Register duplication is used to reduce fanout on registers (to improve delays)
- Xilinx recommends manual register duplication
  - Most synthesis vendors create signals <signal_name>_rep0, _rep1, etc.
    - Implementation tools pack these signals into the same slice
    - This can prohibit a register from being moved closer to its destination
  - When manually duplicating registers, do *not* use a number at the end
    - Example: <signal_name>_0dup, <signal_name>_1dup

Synthesis Options

Hierarchy Management

- The basic settings are:
  - Flatten the design: Allows total combinatorial optimization across all boundaries
  - Maintain hierarchy: Preserves hierarchy without allowing optimization of combinatorial logic across boundaries

Hierarchy Preservation Benefits

- Easily locate problems in the code based on the hierarchical instance names contained within static timing analysis reports
- Enables floorplanning and incremental design flow
- The primary advantage of flattening is to optimize combinatorial logic across hierarchical boundaries
  - If the outputs of leaf-level blocks are registered, there is no need to flatten

Shift Register LUT (SRL16)

Synplify, Precision, and XST

- To infer the Shift Register LUT (SRL), the code must have the following primary characteristics:
  - No set or reset signal
  - Serial-in, serial-out
- SRLs can be initialized on power-up via an INIT attribute in the Xilinx User Constraints File (UCF)

VHDL:

```
process(clk)
begin
    if rising_edge(clk) then
      if ce = '1' then
        sr <= din & sr(0 to 14);
      end if;
    end if;
end process;
dout <= sr(15);
```

Verilog:

```
always @ (posedge clk)
begin
    if (ce)
      sr = {din, sr[0:14]};
end
assign dout = sr[15];
```

Flip-Flop Example

VHDL:

```
process(clk, reset, set)
begin
	if (reset = '1') then q <= '0';
	elsif (set = '1') then q <= '1';
	elsif rising_edge(clk) then
	   if (sync_set = '1') then
	         q <= '1';
	   elsif (sync_reset = '1') then
	         q <= '0';
	   elsif (ce = '1') then
	         q <= d;
	   end if;
	end if;
end process;
```

Verilog:

```
always @ (posedge clk or posedge
reset or posedge set)
	if (reset)
		q <= 0;
	else if (set)
		q <= 1;
	else if (sync_set)
		q <= 1;
	else if (sync_reset)
		q <= 0;
	else if (ce)
		q <= d;
end
```

Carry Logic

Synplify, Precision, and XST

- Synthesis maps directly to the dedicated carry logic
- Access carry logic through adders, subtractors, counters, comparators (>15 bits), and other arithmetic operations
  - Adders and subtractors (SUM <= A + B)
  - Comparators (if A < B then)
  - Counters (COUNT <= COUNT + 1)
- Note: Carry logic will not be inferred if arithmetic components are built with gates
  - For example: XOR gates for addition and AND gates for carry logic will not infer carry logic

Carry Logic Examples

VHDL:

```
count <= count + 1 when
        (addsub = '1') else count - 1;

if (a >= b) then
        a_greater_b <= '1';

product <= constant * multiplicand;
```

Verilog:

```
assign count = addsub ? count + 1:
count - 1;

if (a >= b)
        a_greater_b = 1;

assign product = constant *
multiplicand;
```

Block SelectRAM

Synplify, Precision, and XST

- Synplicity: Set the attribute *syn_ramstyle* to "block_ram"
- Other requirements:
  - Place the attribute on the output signal that is driven by the inferred RAM
  - Requires synchronous write
  - Requires registered read address
  - Dual-port RAM is inferred if read or write address index is different
- XST: Based on the size and characteristics of the code, XST can automatically select the best style
  - Available settings: Auto, Block, and Distributed

Block RAM Inference Notes

Synplify, Precision, and XST

- Synthesis tools cannot infer:
  - Dual-port block RAMs with configurable aspect ratios
    - Ports with different widths
  - Block RAMs with enable or reset functionality
    - Always enabled
    - Output register cannot be reset
    - **Exception:** Synplify Pro software 7.6 can infer reset
  - Dual-port block RAMs with read and write capability on both ports
    - Block RAMs with read capability on one port and write on the other port can be inferred
  - Dual-port functionality with different clocks on each port
- These limitations on inferring block RAMs can be overcome by creating the RAM with the CORE Generator™ system or instantiating primitives

Block RAM Example

VHDL:

```
signal mem: mem_array;
attribute syn_ramstyle of mem: signal is
"block_ram";
...
process (clk)
begin
        if rising_edge(clk) then
            addr_reg <= addr;
            if (we = '1') then
                        mem(addr) <= din;
            end if;
        end if;
end process;
dout <= mem(addr);
```

Verilog:

```
reg [31:0] mem[511:0] /*synthesis
syn_ramstyle = "block_ram"*/;

always @ ( posedge clk)
begin
        addr_reg <= addr;
        if (we)
                        mem[addr] <= din;
end

assign dout = mem[addr_reg];
```

Inferring Memory

ROM

Synplify, Precision, and XST

- Synplicity: Infer ROM primitives with an attribute
  - Set *syn_romstyle* to select_rom
  - Otherwise, Synplify infers a LUT primitive with equations
    - Same implementation, except it is not a ROM primitive
- XST automatically maps to ROM primitives

Distributed ROM Example

VHDL:

```
type rom_type is array(7 downto 0) of
std_logic_vector(1 downto 0);
constant rom_table: rom_type := (“10”,
“00”, “11”, “01”, “11”, “10”, “01”, “00”);
attribute syn_romstyle: string;
attribute syn_romstyle of rom_table:
signal is “select_rom”;
...
rom_dout <= rom_table(addr);
```

Verilog:

```
reg [1:0] rom_dout /*synthesis
syn_romstyle = “select_rom”*/;

always @ ( addr)
case (addr)
        3'b000: rom_dout <= 2'b00;
        3'b001: rom_dout <= 2'b01;
        3'b010: rom_dout <= 2'b10;
        3'b011: rom_dout <= 2'b11;
        3'b100: rom_dout <= 2'b01;
        3'b101: rom_dout <= 2'b11;
        3'b110: rom_dout <= 2'b00;
        3'b111: rom_dout <= 2'b10;
endcase
```

SelectIO Standard

Synplify, Precision, and XST

- Instantiate in HDL code (required for differential I/O)
  - For a complete list of buffers, see the following elements in the *Libraries Guide*:
    - IBUF_selectIO, IBUFDS
    - IBUFG_selectIO, IBUFGDS
    - IOBUF_selectIO
    - OBUF_selectIO, OBUFT_selectIO, OBUFDS, OBUFTDS
- Synplicity: Use attribute in HDL code
- Specify in the UCF or XST Constraints File (XCF)
- Use the Xilinx Constraints Editor
  - In the Ports tab, check the I/O Configuration Options box

SelectIO Standard Example

VHDL:

```
ibuf_data_in_inst: IBUF
    generic map (IOSTANDARD =
    "HSTL_III")
    port map (I => data_in, O =>
    data_in_i);
```

Verilog:

```
/* For primitive instantiations in Verilog
you must use UPPERCASE for the
primitive name and port names */

IBUF #(
        .IOSTANDARD("HSTL_III")
        ) ibuf_data_in_inst
        (.I(data_in), .O(data_in_i));
```

Global Buffers

- BUFG
  - All synthesis tools will infer on input signals that drive the clock pin of any synchronous element
- BUFGDLL
  - Synplicity: Can be inferred through synthesis by setting attribute *xc_clockbuftype* = BUFGDLL
  - XST: Must instantiate