

Advance HDL Design Training On Xilinx FPGA

Yu-Tsang/Carven Chang
Associate Researcher, CIC/NSC

carven@cic.edu.tw

February, 2001

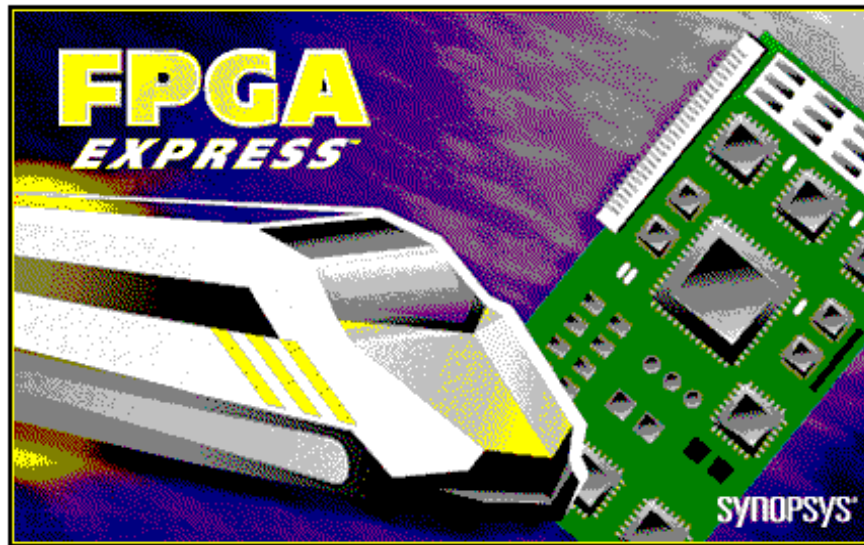
Outline I

- FPGA Express V3.x Overview
- Xilinx HDL Synthesis Flow
- Architecture
- Introduce to FPGA Design
- HDL Coding
- LAB I

Outline II

- Black Box Instantiation
- Introduce to Efficient Synthesis
- Timing Constraints with Foundation Express
- Simulation with Foundation Express Netlists
- Xilinx CORE Generator
- Synopsys FPGA Express & Xilinx Design Manager
- LAB II

FGPA Express V3.x



What is FPGA Express?

- A complete FPGA logic-synthesis and optimization tool.
 - Provide an integrated text editor
 - Analyze HDL(VHDL and Verilog HDL) design source files for correct syntax using Synopsys industry-standard HDL policy
 - accept any combination of VHDL, Verilog HDL, and FPGA netlist files as sources for a single design
 - Use architecture-specific algorithms to target Xilinx device(FPGA and CPLD)
 - Optimize logic for speed and area as directed by your design constraints.
 - Extract and display accurate post-synthesis delay information for timing analysis and debugging

FPGA Express Features

- FPGA Express v3.x Device Support
 - 3K A, 4K E / L / EX / XL / XV
 - Spartan, 5200, 9500
 - Virtex, Virtex-E, Virtex-II
- Xilinx Architecture-Specific Algorithms for Best Optimization
- Verilog (IEEE 1364) and VHDL (IEEE 1076-1987) Support
- FPGA Compiler II
 - Design Ware, Constraint-driven, retiming...

FPGA Express Features ...

- FPGA Express Technology
 - Constraint Entry GUI
 - Automatic I/O Pad and Global Signal Mapping
 - Built-in Timing Analyzer
 - Built-in Module Generation (Fast Carry, RLOC)
 - Resource Sharing
 - Hierarchy Control
 - Source Code Compatible With Synopsys Design Compiler
 - Schematic Viewer called Vista

Benefits of Using FPGA Express

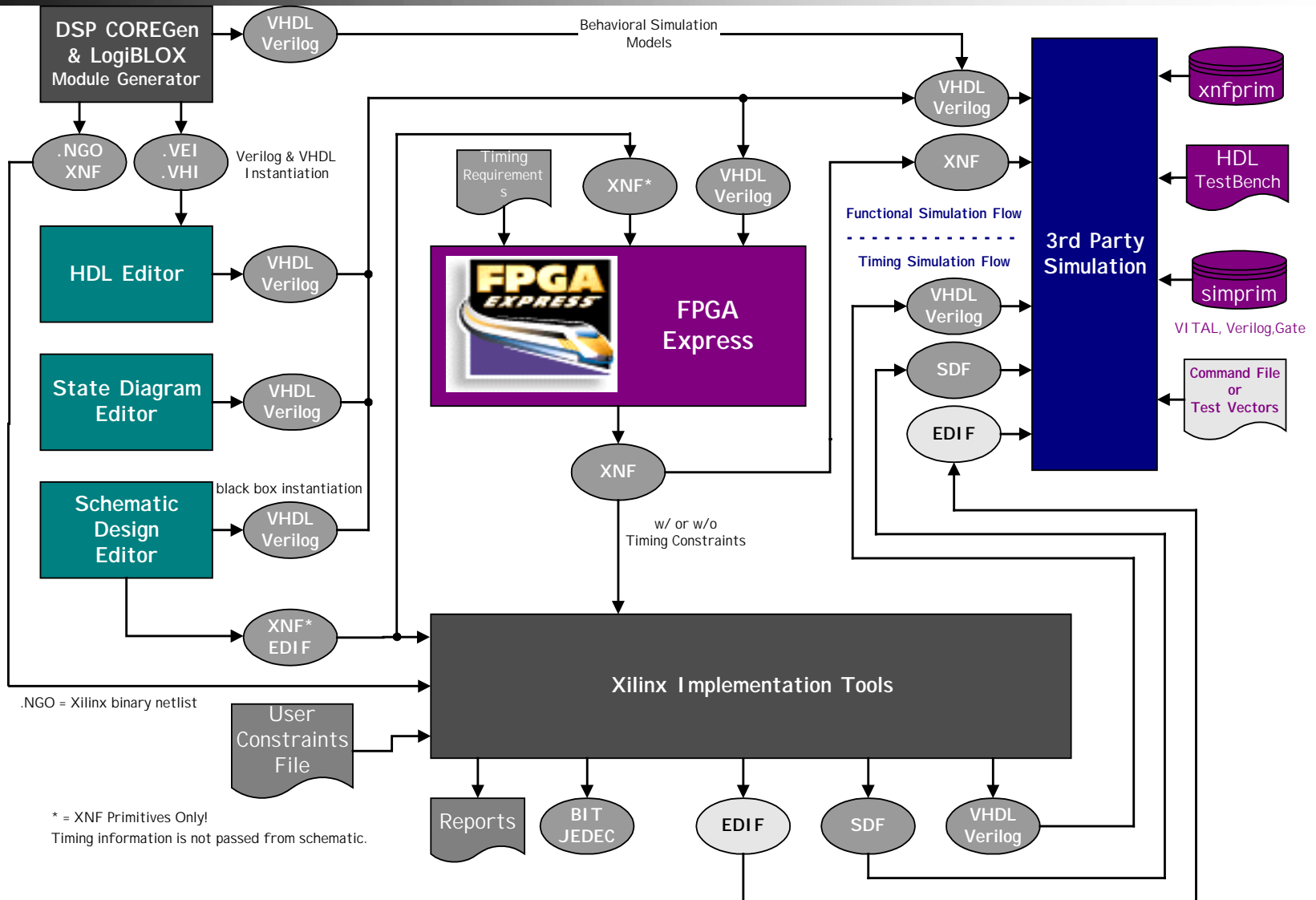


- Adding Control to the Design Process
 - The target performance is entered in advance with design-specific constraints, there is little or no time spent in iteration cycles
- Migrating to HDL Design Methodology
 - Using an HDL-based design methodology increases productivity because HDL code is vendor-independent, retargetable, and reusable
- Using Common, Familiar Systems
 - Using Windows 95/98/NT/2000 OS
 - Windows-compliant GUI uses standard commands and procedures
 - no command scripts are required for constraint entry

What's New in Express v3.x

- Don't_touch
 - Instructs Express to not re-optimize instantiated modules
 - Available via Constraint GUI, HDL attribute, script command in FE_SHELL
- Virtex architectural support additions
 - SRL 16 inference
 - ROM inference

FPGA Express Flow



FPGA Express Design Flow...



- **Toolbar**

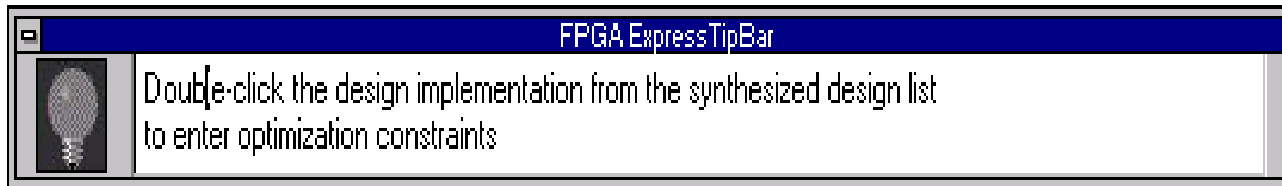
Contains the Design Steps from Left to Right



Only the available design steps are highlighted

- **Tipbar**

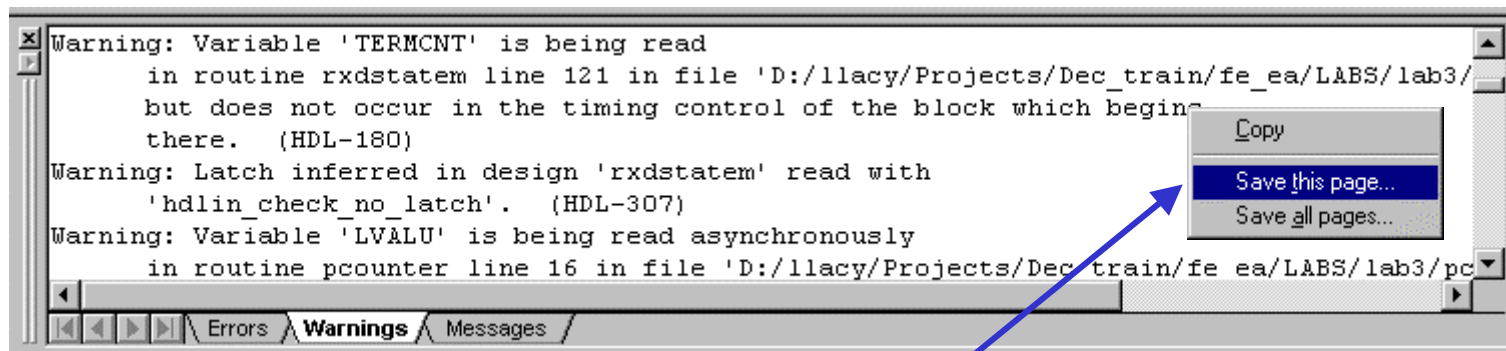
Provides Guidance at Each Step



FPGA Express Design Flow...



- Errors / Warnings / Messages Window
Provides feedback for selected design element



Right click inside window to save
information to a text file

- Errors / Warnings / Messages window visible at all times

Simple Four Step Design Process



The screenshot displays the Xilinx FPGA Express software interface for a project named 'lab3.exp'. The 'Design Sources' pane on the left shows a project tree with files like 'bernie_counter.v' and 'bernie_ff.v'. A red circle labeled '1 Analyze Files' is placed over this pane. The 'Chips' pane on the right shows a hierarchy of components, with 'rxld' selected. A red circle labeled '4 Optimize' is placed over this pane. The 'Create Implementation - bernie_counter' dialog box is open in the foreground. It shows the 'Implementation Name' as 'bernie_counter', the 'Target device' as 'Xilinx 4028EXHQ208', and the 'Speed grade' as 'ex-2'. A red circle labeled '2 Select Device' is placed over the 'Speed grade' dropdown. The 'lab3.exp - rxld [Constraints]' dialog box is also open, showing a table of constraints. A red circle labeled '3 Enter Constraints' is placed over the 'Output Delay (ns)' column of this table.

1 Analyze Files

2 Select Device

3 Enter Constraints

4 Optimize

Name	Direction	Input Delay (ns)	Output Delay (ns)
<default>			
DPLUS	input	20(RC,CLK)	
DMINS	input	20(RC,CLK)	
PIDBUS<7>	output		20(RC,CLK)
PIDBUS<6>	output		20(RC,CLK)

Black Box Instantiation

- What is Black Box Instantiation
 - Existing Design Modules or Elements (XNF, EDIF, .ngo)
 - LogiBLOX Components
 - Pre Optimized Netlists (PCI Cores)
- Procedure For Using Black Box
 - Create a place holder in the HDL
 - Use the Xilinx Implementation Tools (*ngdbuild*) to resolve (link in) all references

Black Box Instantiation ...

- Black Box using Verilog

- Instantiate the module in the RTL
- Create an empty module to represent the black box

- Black Box using VHDL

- Instantiate the entity in the RTL
- Create a component declaration

- Issues

- Will not automatically infer GSR nets
 - Under Constraints select “Ignore unlinked cells during GSR mapping”

How are the Constraints used?



- FPGA Express passes the constraints to the output .NCF file
- The exact value of the timing constraint is not directly used for synthesis optimization
 - Speed or Area selection under constraint editor “module” tab is used for selecting the type of optimization to perform on the sections of the design
- FPGA Express creates Xilinx recommended constraints:
 - Periods and Offsets for global constraints
 - FROM: TO for fast, slow or multi-cycle paths

Constraint Entry

- Constraints that Express can apply
 - FROM:TO timespecs which use FFS, LATCHES, and PADS
 - User Defined Sub-Paths
 - Control Use of IOB Register
 - Pin Location Constraints
 - Slew Rate
 - Pull-up / Pull-down
 - Input Register Delay

Constraint Entry ...



- Synthesis -> Edit Constraints...

lab3.exp - rxd [Constraints]

Define Clocks
PERIOD / RISE / FALL

Define Time Constraint
FROM : TO
Sub-Paths

Define Port Attributes
DELAY
PULLUP / PULLDOWN
SLEW / Global Buffers
Pin Locations

Define Hierarchy Preservation
Eliminate / Preserve
Operator Sharing
Optimize / Effort

Xilinx Constraints
Implementation Tool Target
GSR Usage

	Name	Direction	Input Delay (ns)	Output Delay (ns)	Global Buffer
1	<default>				AUTOMATIC
2	RST	input	20/(RC, CLK)		
3	CLK	input	20/(RC, CLK)		
4	DPLUS	input	20/(RC, CLK)		
5	DMINUS	input	20/(RC, CLK)		
6	PIDBUS<7>	output		20/(RC, CLK)	
7	PIDBUS<6>	output		20/(RC, CLK)	
8	PIDBUS<5>	output		20/(RC, CLK)	
9	PIDBUS<4>	output		20/(RC, CLK)	
10	PIDBUS<3>	output		20/(RC, CLK)	
11	PIDBUS<2>	output		20/(RC, CLK)	
12	PIDBUS<1>	output		20/(RC, CLK)	
13	PIDBUS<0>	output		20/(RC, CLK)	
14	DATABUS<7>	output		20/(RC, CLK)	
15	DATABUS<6>	output		20/(RC, CLK)	
16	DATABUS<5>	output		20/(RC, CLK)	

Constraint Entry ...



■ Creating Sub-Paths

Create / Edit Timing Sub Path

Primary Path: FF's clocked by rising /rxd/CLK -> All Output Ports

Sub-path Name: p0 Delay: 20

2 Select Sub-Path Elements

3 Set New Delay

1 Right Click on Path

lab3.exp - rxd [Constraints]

	From	To	Req. Delay
1	All Input Ports	RC - CLK	20
2	RC - CLK	All Output Ports	20
3	RC - CLK	RC - CLK	20
4	LH - /1/N31	RC - CLK	10

New Sub path
Edit Sub path
Delete Sub path

Clear All Select All Clear All

Select Select: OK Cancel Help

Timing Analysis

- Synthesis -> Edit Constraints...

	Clocks	Paths	Ports	Modules	Xilinx Options
		From	To	Req. Delay	Est. Delay
1		All Input Ports	RC - N128	16	13
2		RC - N128	All Output Ports	16	10
3		RC - N128	RC - N128	16	19

Timing Constraint

Estimated Timing

	RC - N128	RC - N128	Est. Delay
13	Q_reg	Q_reg	5.6
14	Q_reg	Q_reg	5.6
15	Q_reg	Q_reg	5.6
16	Q_reg	Q_reg	5.6
17	Q_reg	Q_reg	5.6
18	cntrout_reg<0>	PIDOUT_reg<7>	17.6
19	cntrout_reg<0>	PIDOUT_reg<6>	18.1
20	cntrout_reg<0>	PIDOUT_reg<5>	18.1
21	cntrout_reg<0>	PIDOUT_reg<4>	18.5
22	cntrout_reg<0>	PIDOUT_reg<3>	19.0
23	cntrout_reg<0>	PIDOUT_reg<2>	19.0
24	cntrout_reg<0>	PIDOUT_reg<1>	19.0

Instance Fanout

	Instance Path	Cell Type	Delay	Fanout
1	cntrout_reg<0>/C	DFF	0.0	66
2	cntrout_reg<0>/Q	DFF	2.8	11
3	C373/M0	EQN	8.3	11
4	C373/O	EQN	10.3	4
5	PIDOUT_reg<7>/D	OUTFF	13.0	4
6	PIDOUT_reg<7>/C	OUTFF	17.6	66

Path Delay

Constraint Entry ...

- Can set default of Exporting Timing Specifications (**Synthesis -> Options -> Project**)
- Can disable writing of TimeSpecs to XNF

Controls writing of
TimeSpecs

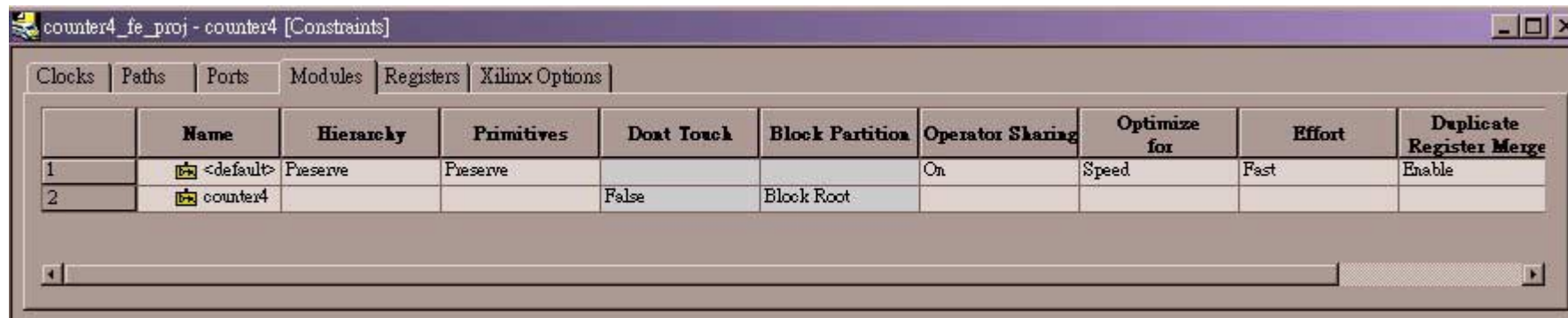


Timing Analysis ...

- Uses same timing numbers as Design Compiler / FPGA Compiler / Pre-Route TRCE
- High correlation between Express post-synthesis and Xilinx M1 pre-map
- Wire load models do not model fanout of greater than 10 loads
- Highlights constraints which are not met
- Shows individual instance fanout

Hierarchy Specification

- The module view lets the user SEE the design hierarchy
- Hierarchy Control is available you can preserve or eliminate level of hierarchy from optimization
- Operator Sharing, Speed/Area, Effort
- Primitive Control



GSR Insertion

- Automatically detects the presence of set / reset signals in the design
 - A net that sets / resets all sequential elements in the design
 - A net driven by a GSR cell (STARTUP)
- Merges / Resolves GSR nets if more than one exists
- (If needed) Inserts STARTUP symbol into netlist (if appropriate for the architecture)

Coding Styles (Tips & Tricks)



- ‘dont_touch’ and ‘keep’ equivalents
 - Preserve redundant logic by placing logic in its own entity or module, then apply ‘preserve’ to this level of hierarchy
- The following are inferred directly from HDL and implemented as optimized modules:
 - Multiplier
 - Adder, Subtractor, and Adder/Subtractor
 - Incrementer, decrementer, and incrementer/decrementer
 - Comparator
 - Mutiplexer (select operator)

Coding Styles ...

- Macros
 - Don't use 'define inside module definitions
 - Don't use nested 'define attributes
 - In Verilog, define local references as generics or parameters
- Modules
 - Ensure top-level modules only contain interconnects
 - Don't use logic expressions when passing values through ports
- Functions
 - Do not use global references within a function
 - Be aware that task and function local storage is static
- Expressions
 - Use parentheses to indicate precedence
 - Replace repetitive expressions with function calls or continuous assignments

Coding Styles ...

■ General

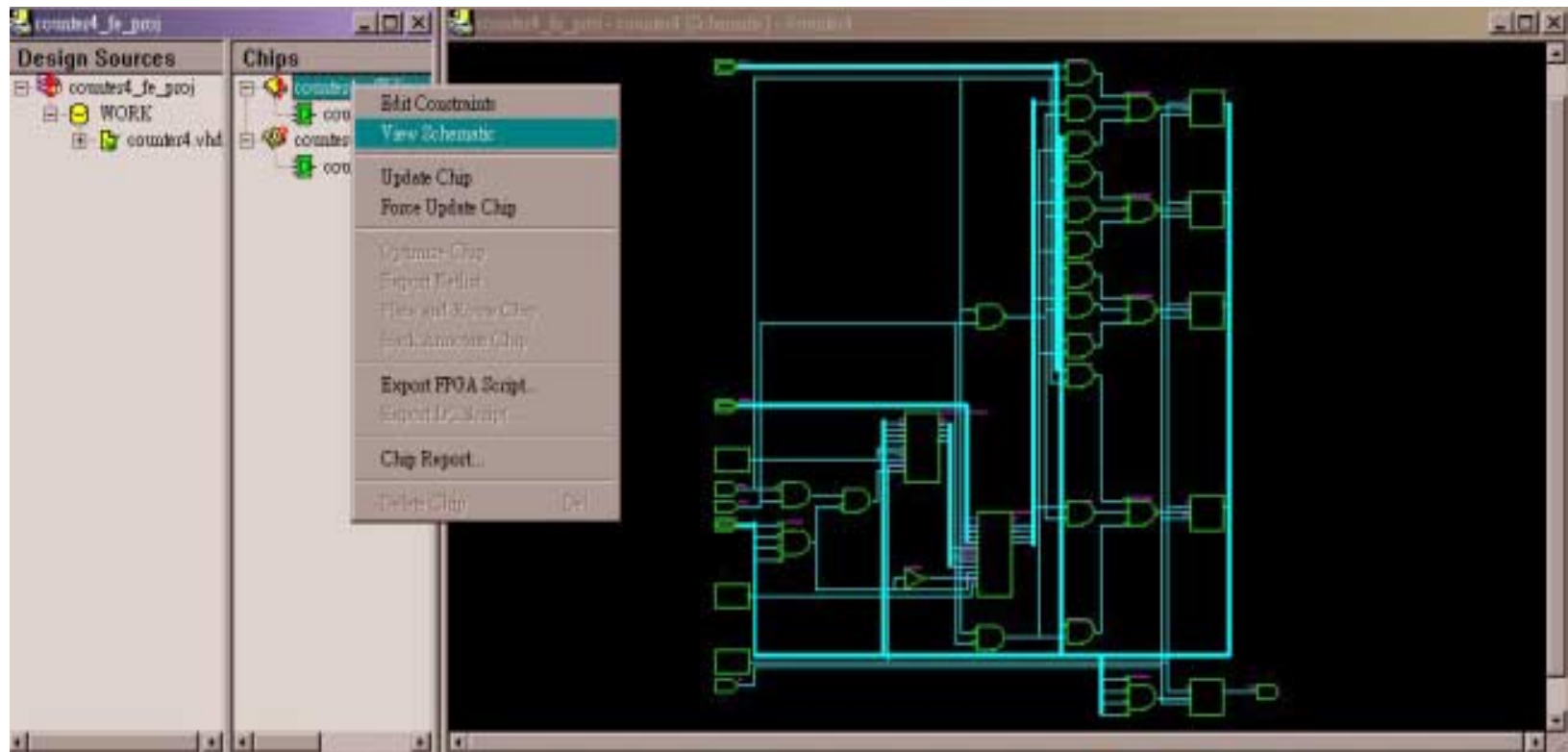
- An incomplete case statement results in the creation of a latch. Use “full_case” directive, or default clause
- Use Nonblocking statements when a sequence of statements refers to the same variable more than once
- Merge resources that can be shared into the same process (VHDL) or always (Verilog)
- Use an attribute (VHDL) or parameter (Verilog) statement to generate one-hot state machines
- VHDL generate statements can cause long compile times unfolding the logic - Use them wisely

Coding Styles ...

- Use VHDL arithmetic packages for comparators (<, >, <=, >=, =)
 - Uses architecture-specific resources
 - Better quality of results
 - Both signed and unsigned packages
 - VHDL libraries for STD_LOGIC
 - `library IEEE ;`
 - `use IEEE.STD_LOGIC_unsigned.all;`

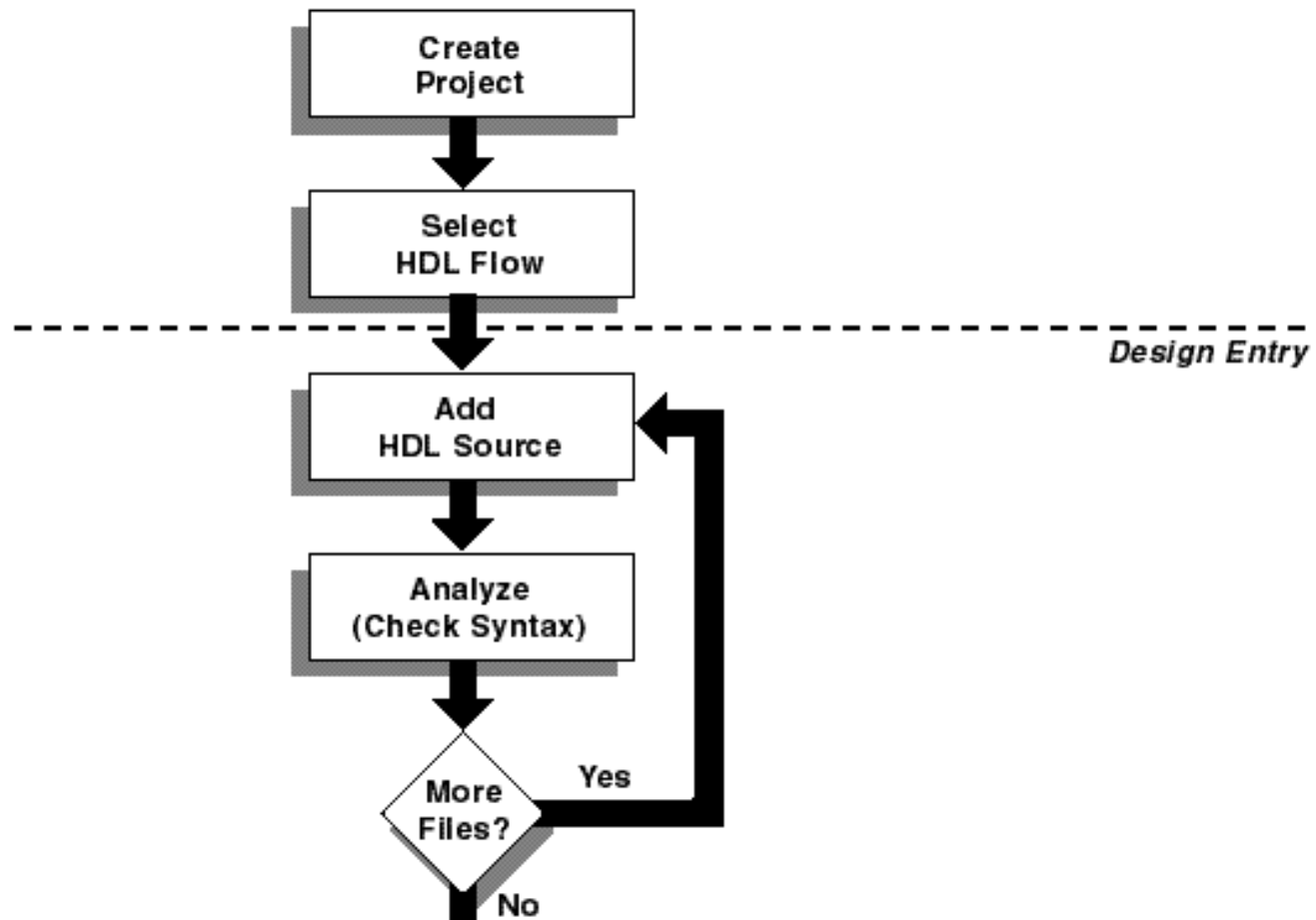
Schematic Viewer: Vista

- The “Vista” Schematic Viewer allows you to see what FPGA Express has built graphically

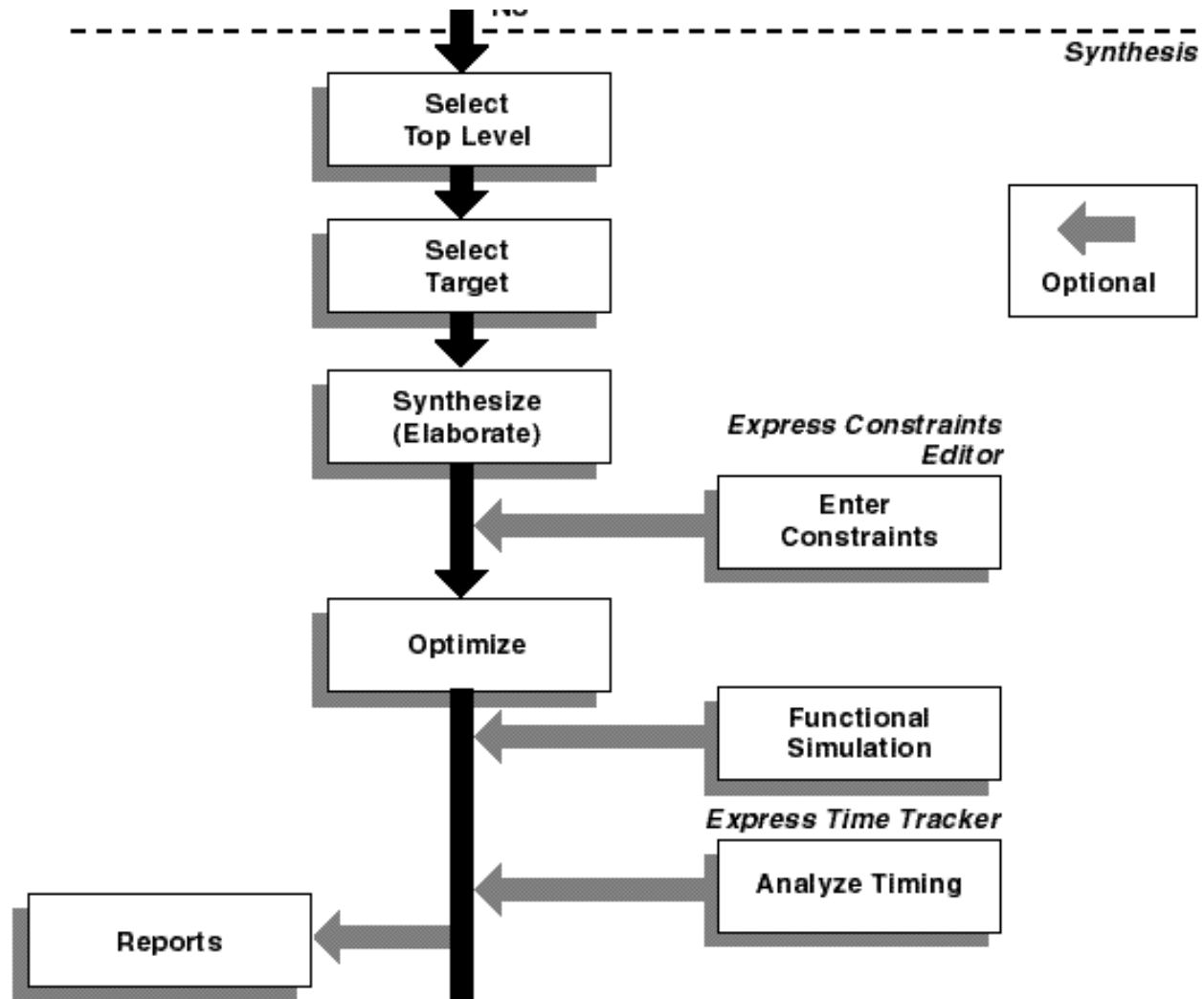


Xilinx HDL Synthesis Flow

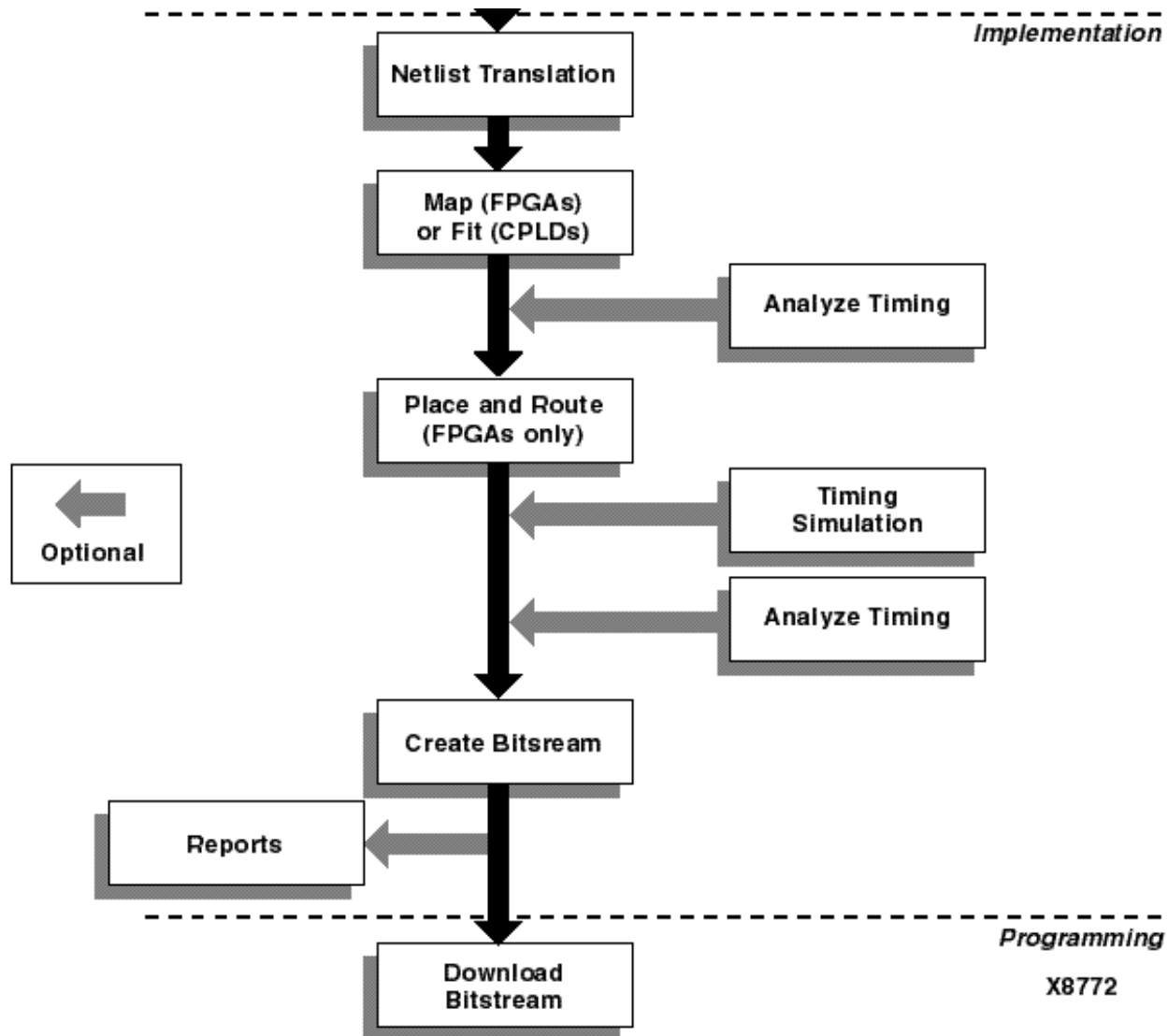
HDL Flow - Design Entry



HDL Flow - Synthesis



HDL Flow - Implementation & Programming



X8772

Top-level Design I

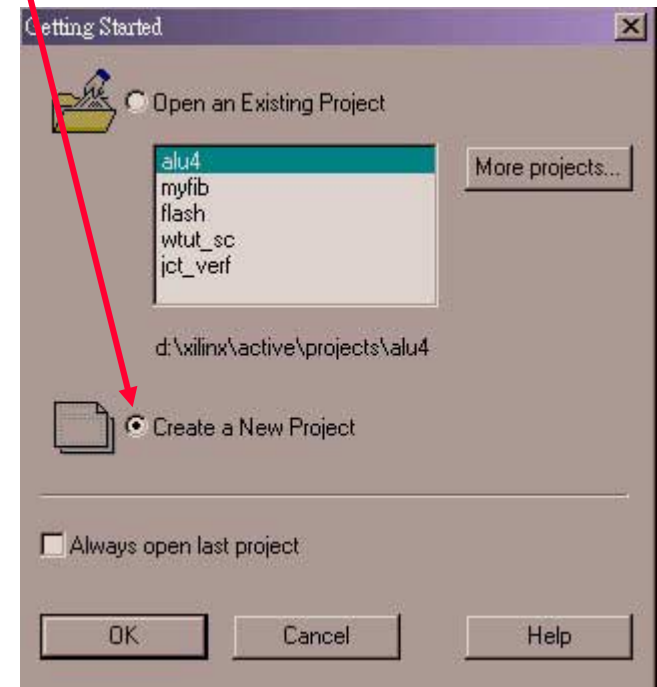
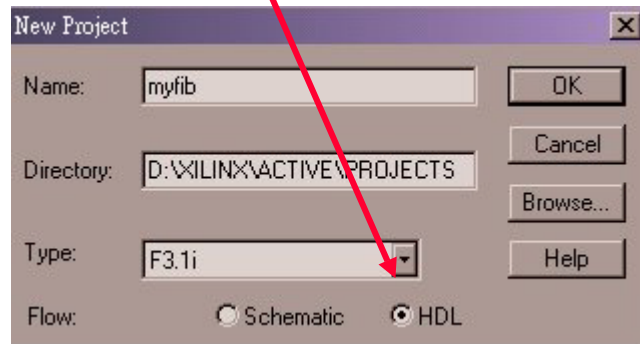
- HDL Flow projects do not require the designation of a top-level design until synthesis.
- Only VHDL or Verilog source files can be added to an HDL Flow project.
- VHDL and Verilog source files can be created by the HDL Editor, Finite State Machine Editor, or other text editors.
- When you initiate the synthesis phase, you designate one of the project's entities (VHDL) or modules (Verilog) as the top-level of the design.
- The list of entities and modules is automatically extracted from all the HDL source files added to the project.

Top-level Design II

- All HDL designs and modules below the top-level file are elaborated and optimized.
- HDL designs can contain underlying LogiBLOXs, schematics (EDIF files), ABEL modules, and XNF files that are instantiated in the VHDL and Verilog code as "black boxes."
- Black box modules are not elaborated and optimized during synthesis.
 - Refer to the "Black Box Instantiation" section for more information on Black Boxes.

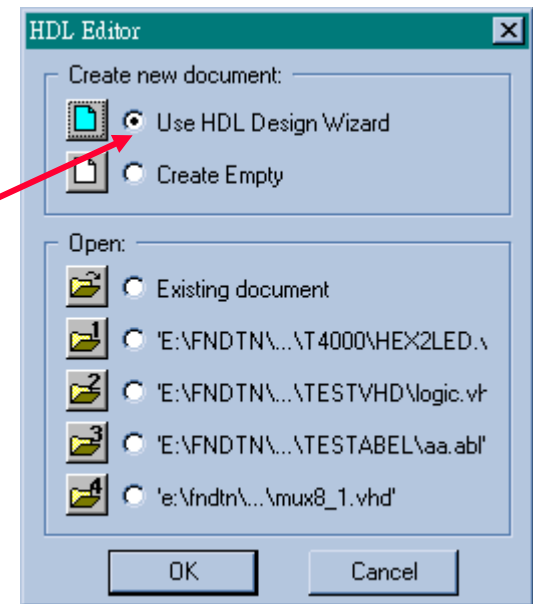
Create a HDL Project

- New a Project
 - Start Foundation Express and Select “Create a New Project”
 - In Foundation Project Manager, File > New Project
- Select HDL Flow



Creating the Design I

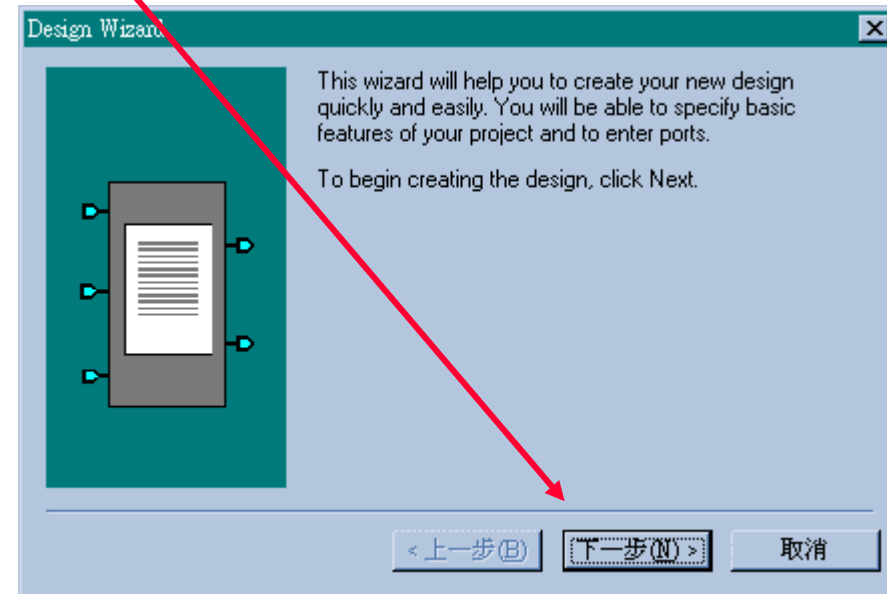
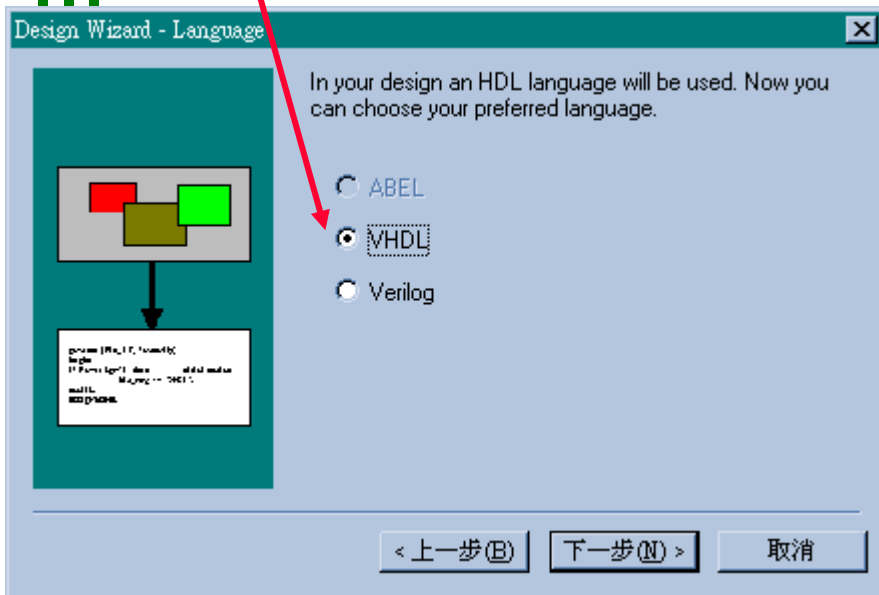
- 1. Open the HDL Editor
 - by clicking the HDL Editor icon in the Design Entry box on the Project Manager's Flow tab.



- 2. Use Design Wizard to create HDL
 - When the HDL Editor window appears, you may select an existing HDL file or create a new one. The following steps describe creating a new HDL file with the Design Wizard.
 - When the HDL Editor dialog box displays, select Use HDL Design Wizard. Click OK.

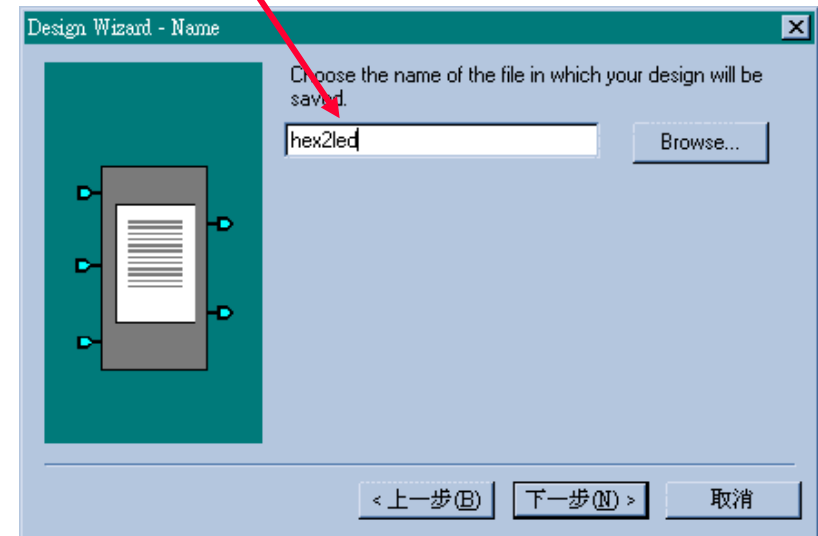
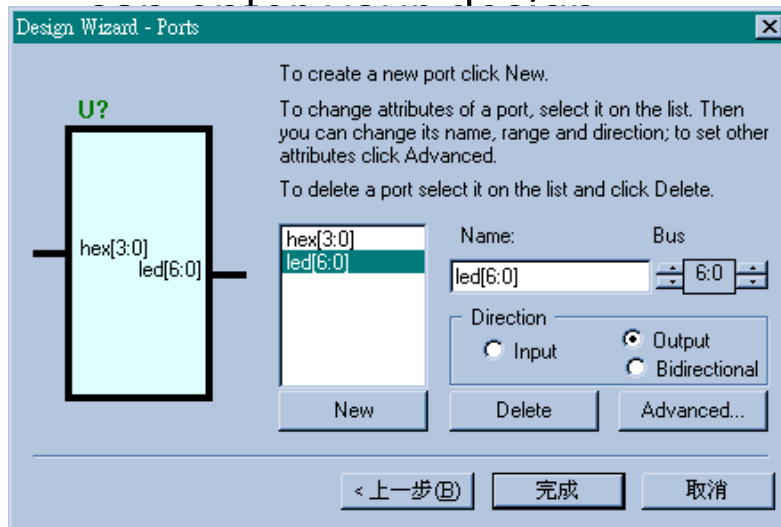
Creating the Design II

- Click Next in the Design Wizard window.
- From the Design Wizard - Language window, select VHDL or Verilog. Click Next.
- Note: For top-level ABEL designs, you must use the Schematic Flow.



Creating the Design III

- In the Design Wizard - Name window, enter the name of your design file. Click Next.
- Define your ports in the Design Wizard-Ports window by clicking NEW, entering the port name, and selecting its direction. Click Finish. The Wizard creates the ports and gives you a template (in VHDL or Verilog) in which you



-
- The screenshot shows the HDL Editor interface. The main window displays a VHDL code snippet for a 7-segment decoder. The code is as follows:
- ```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity hex2led is
5 port (
6 hex: in STD_LOGIC_VECTOR(3 downto 0);
7 led: out STD_LOGIC_VECTOR(6 downto 0);
8);
9 end hex2led;
10
11 architecture hex2led_arch of hex2led is
12 begin
13 -- <<Enter your state machine logic here>>
14
15 end hex2led_arch;

```
- The Language Assistant window is open, showing the "Synthesis templates" list on the left. The "HEX2LED Converter" template is selected. The right pane displays the corresponding VHDL code for the decoder:
- ```

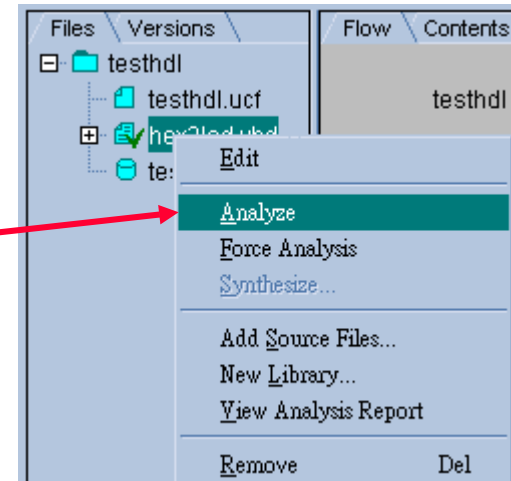
--HEX-to-seven-segment decoder
--  HEX:  in      STD_LOGIC_VECTOR (3 downto 0)
--  LED:   out     STD_LOGIC_VECTOR (6 downto 0)
--
-- segment encoding
--  0
--  ...
-- 5| 1|
--  ... <- 6
-- 4| 1|
--  ...
--  3
--
with HEX_SELECT
LED<= "1111001" when "0001", --1
      "0100100" when "0010", --2
      "0110000" when "0011", --3
      "0011001" when "0100", --4
      "0010010" when "0101", --5
      "0000010" when "0110", --6

```
- The status bar at the bottom indicates "For Help, press F1", "Ln 14, Col 3", "VHDL", and "NUM".

Analyzing Design File Syntax

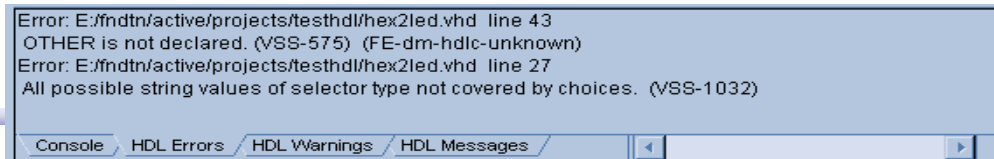
■ How to analyze syntax

- Syntax is checked automatically when the design is added to the project.
- In the HDL Editor, by selecting **Synthesis > Check Syntax**.
- In Project Manager, by selecting **Project > Analyze All HDL Source Files**.
- In Files tab, by selecting source file and right clicking, and then clicking "Analyze"



■ How to handle error and warning

- Use the HDL Error and HDL Warnings tabs in the messages area at the bottom of the Project Manager to view any syntax errors or messages output during analysis.



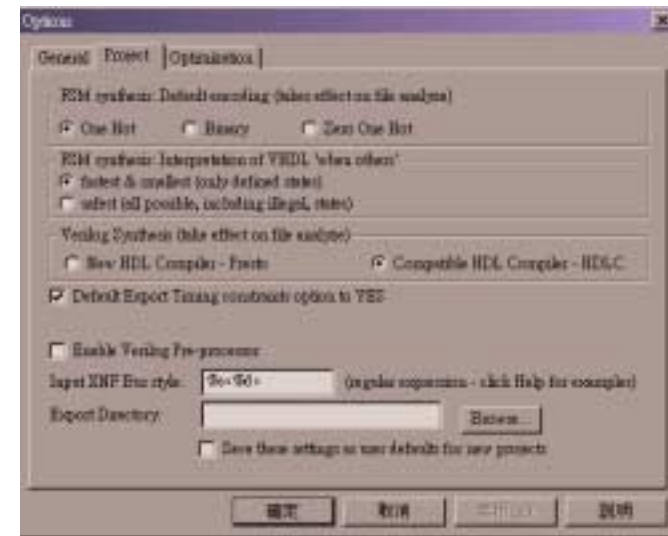


Performing HDL Behavioral Simulation

- If you installed an HDL simulation tool such as ACTIVE-VHDL or ModelSIM, you can perform a behavioral simulation of your HDL code.
- Please refer to the documentation provided with these tools for more information.

Synthesizing the Design I

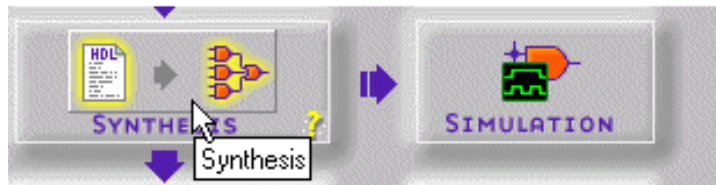
- Translate the design into gates and optimize it for a target architecture.
- 1. Set the global synthesis options
 - By selecting **Synthesis > Options** from the Project Manager.
 - In the Synthesis Options dialog, you can set the following defaults
 - FSM Encoding (One Hot or Binary)
 - FSM Synthesis Style
 - Export schematic
 - Default clock frequency
 - Export timing constraints to the place and route software
 - Input XNF bus style
- 2. Click **OK** to close the Synthesis Options dialog



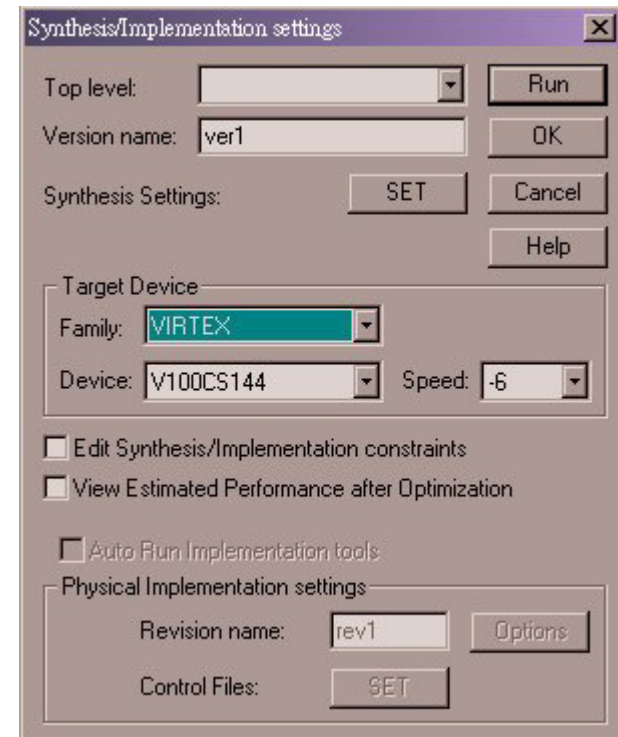
Synthesizing the Design II



- 3. Click the Synthesis icon on the Synthesis button on the Flow tab.



- 4. The Synthesis/Implementation dialog box is displayed.
- 5. Select the name of the top-level VHDL entity or Verilog module.
 - Processing will start from the file named here and proceed through all its underlying HDL modules.
- 6. Enter a version name.



Synthesizing the Design III

- 7. Select the target device.
- 8. Modify the synthesis processing settings as desired.
 - Modify the **target clock frequency**
 - Select to **optimize** the design for speed or area
 - Select the **effort** level as high or low
 - Select whether **I/O pads** should be inserted for the designated top-level module
 - Select **Preserve Hierarchy** to preserve the design hierarchy. This will result in producing output hierarchical netlists.
 - Select **Edit Synthesis/Implementation Constraints**.
 - Selecting this options pauses synthesis processing after the elaboration phase to allow you to specify constraints (controls) for the design using a constraints editor GUI.
 - This feature is also referred to as the "**Express Constraints Editor**."
 - For more information refer to the "Express Constraints Editor" section.

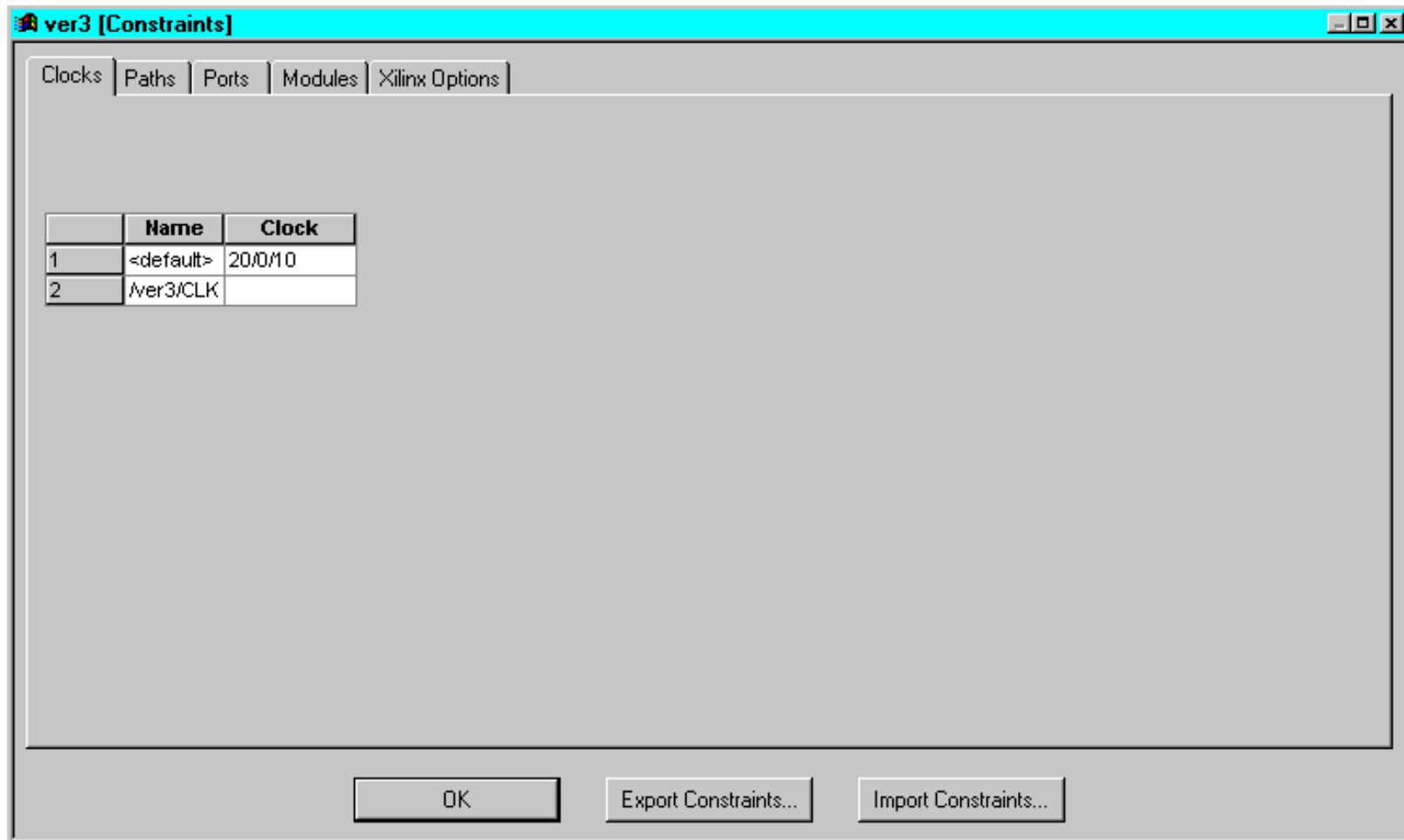
Synthesizing the Design IV

- Select View Estimated Performance after Optimization.
 - Select this option to view the estimated performance results after design optimization.
 - This feature is also referred as the "*Time Tracker*."
 - For more information refer to the "Express Time Tracker" section.
- 9. Click **Run** to synthesize the designated top-level HDL design and its underlying HDL modules.
 - The synthesis compiler automatically includes top-level input and output pads required for implementation (unless instructed not to do so in the Synthesis Settings on the Synthesis/Implementation dialog.)

Express Constraints Editor I

- Express Constraints Editor allows you to set performance constraints and attributes before optimization of FPGA designs.
 - 1. The Express Constraints Editor window automatically displays during Synthesis processing if you checked the Edit Synthesis/Implementation Constraints box on the Synthesis/Implementation dialog.
 - Alternatively, you can access the Express Constraints Editor via the Versions tab by right-clicking on a project version in the Hierarchy Browser and then selecting Edit Constraints.
 - 2. Design-specific information is extracted from the design and displayed in device-specific spreadsheets. Click the tabs to access the various spreadsheets.
 - If you un-checked Insert I/O pads on the Synthesis/Implementation dialog, only the Modules and Xilinx Options tabs are shown. The Clocks, Ports, and Paths tabs apply only to top-level HDL designs.

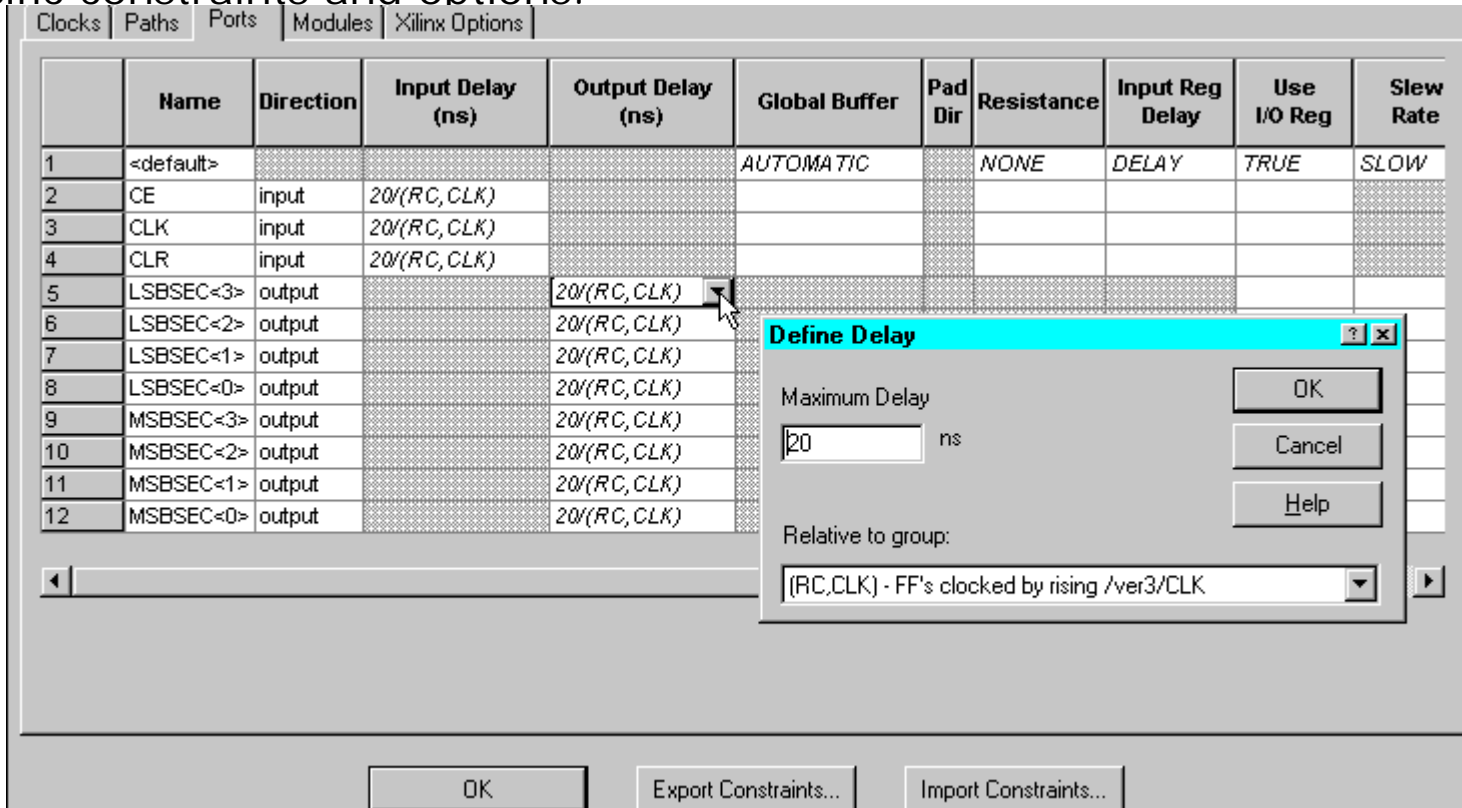
Express Constraints Editor II



Express Constraints Editor III



- 3. Right-click on an item in any of the spreadsheets to edit the value, access a dialog box to edit the value, or access a pulldown menu to select a value.
 - Use the online help in the dialog boxes to understand and enter specific constraints and options.



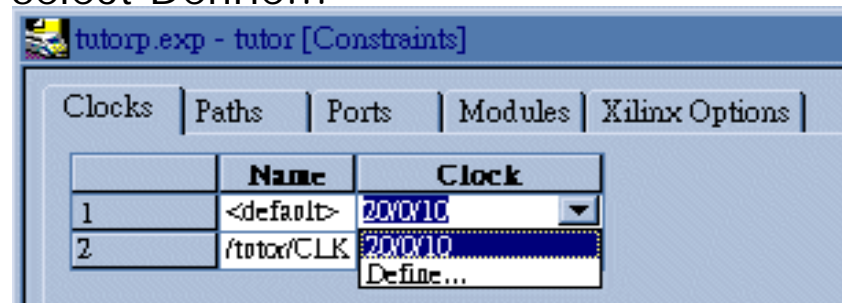
Express Constraints Editor IV



- 4. Import / Export Constraints
 - Optionally, you can import a constraints file (.exc) to use now (click Import Constraints) or you can export the entered constraints to a constraints file (.exc) for reuse (click Export Constraints).
- 5. After you finish editing the constraints, click OK to close the Constraints window and continue the synthesis using the specified constraints.

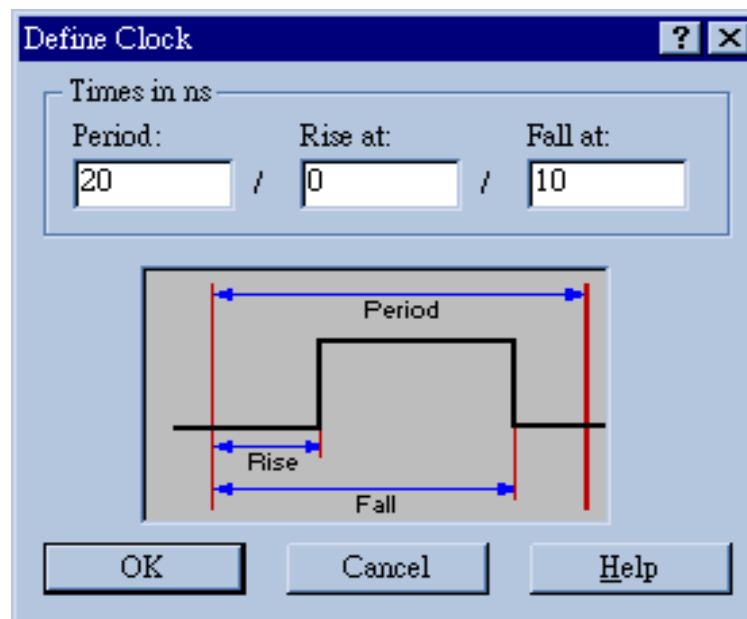
Clocks Constraint Entry I

- Use the Clocks constraint table to specify the waveforms of periodic signals in the design.
 - Name
 - This is the full name of the periodic signal in the design hierarchy (for example, top/module1/clock).
 - Clock
 - This is the waveform (period, rise time, fall time) of each periodic signal.
 - To specify a waveform, click the Clock cell, click the expand arrow that appears in the cell, and then select Define...



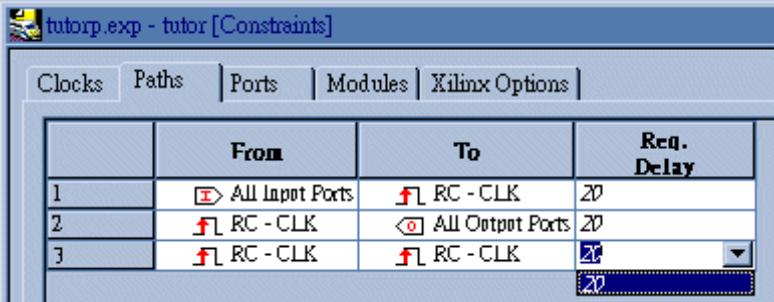
Clocks Constraint Entry II

- This displays the Define Clock dialog box where you can enter the period, rise time, and fall time of the signal.



Paths Constraint Entry I

- Use the Paths constraint table to specify timing constraints for timing groups.
 - From
 - all primary inputs of the design
 - all edge-sensitive sequential elements clocked by a specified periodic signal
 - all level-sensitive sequential elements clocked by a specified periodic signal
 - To
 - all primary outputs of the design
 - all edge-sensitive sequential elements clocked by a specified periodic signal
 - all level-sensitive sequential elements clocked by a specified periodic signal.









tutorp.exp - tutor [Constraints]

	From	To	Req. Delay
1	All Input Ports	RC - CLK	20
2	RC - CLK	All Output Ports	20
3	RC - CLK	RC - CLK	20

Paths Constraint Entry II

tutorp.exp - tutor [Constraints]

Clocks Paths Ports Modules Xilinx Options

	From	To	Req. Delay
1	 All Input Ports	 RC - CLK	20
2	 RC - CLK	 All Output Ports	20
3	 RC - CLK	 RC - CLK	20



All input ports



All output ports



Flip-flops clocked by rising edge



Flip-flops clocked by falling edge



Latches with active high enable



Latches with active low enable



Subpath start point or end point

Paths Constraint Entry III

- Required Delay
 - This is the maximum delay of the path, computed from the waveforms of the periodic signals.
 - This value is the difference between the active edge of the end group of the path and the active edge of the starting group of the path.
 - To enter a new value for a path group, click the Required Delay column to highlight the default value and type in the new value.
- Default Delay Values
 - FPGA Express computes default timing values using the default waveforms of the periodic signals.

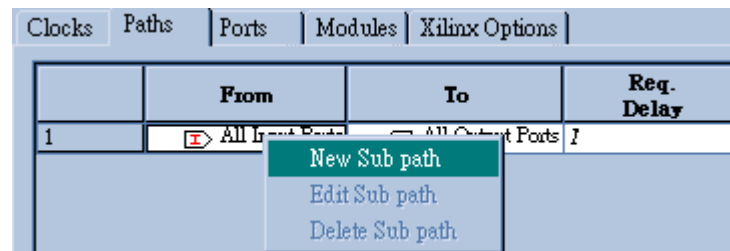
Sub Path I

■ Subpath

- A subset of the actual paths within a primary timing path need a tighter constraint to ensure correct behavior.
- Note that you cannot specify a false or multicycle path by making the subpath delay longer than that of the primary path.

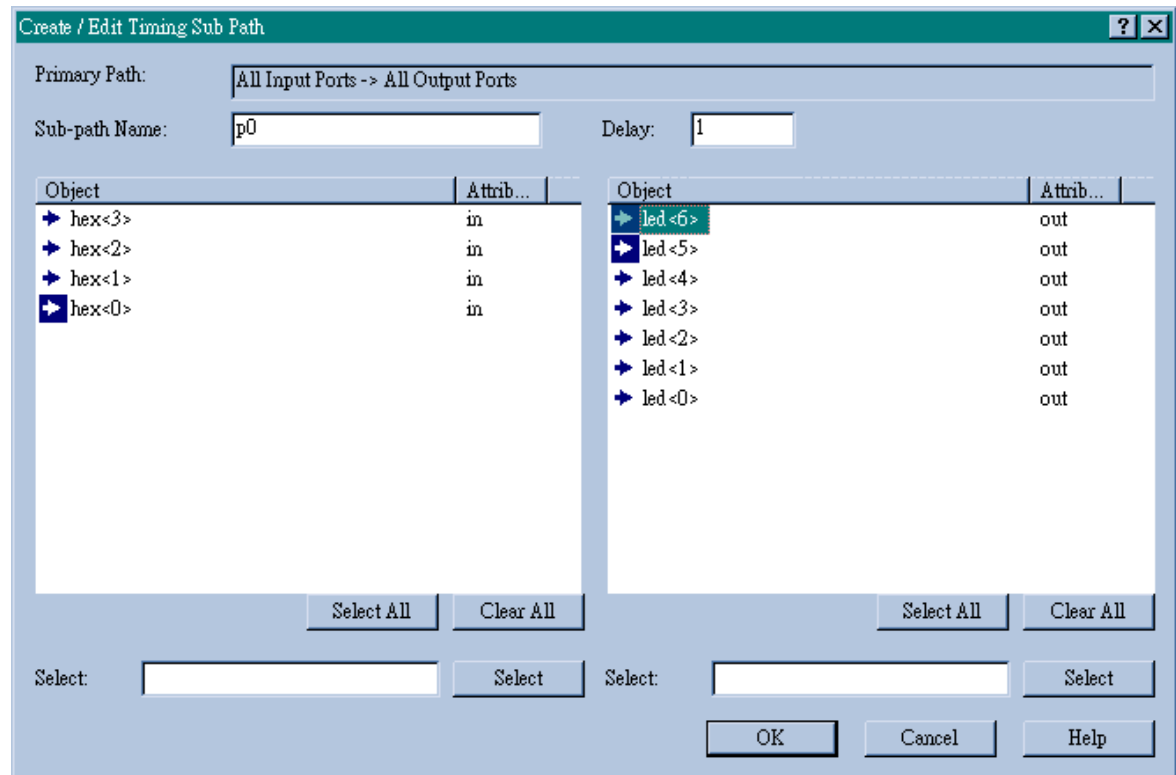
■ Create / Edit Timing Subpath

- Create a new subpath by clicking the right mouse button on either the appropriate primary path or any subpath of the primary path, and selecting New Sub path.
- To edit an existing subpath, click the right mouse button on it and select Edit Sub path



Sub Path II

- In the Create/Edit Timing Sub Path dialog box, enter the desired delay, and then double-click all of cells making up the subpath. The controls below each list give an alternate way of adding named cells to the list using global-style regular expressions (similar to filename wildcards).



Ports Constraint Entry I

- Enter port-specific constraints in the Ports constraint table.

The screenshot shows the 'ver1 [Constraints]' window with the 'Ports' tab selected. The table below represents the data shown in the window.

	Name	Direction	Input Delay (ns)	Output Delay (ns)	Global Buffer	Pad Dir	Resistance	Input Reg Delay	Use I/O Reg	Slew Rate	Pad Loc
1	<default>				AUTOMATIC		NONE	DELAY	TRUE	SLOW	
2	hex<3>	input	0(NC)								
3	hex<2>	input	0(NC)								
4	hex<1>	input	0(NC)								
5	hex<0>	input	0(NC)								
6	led<6>	output		0(NC)							
7	led<5>	output		0(NC)							
8	led<4>	output		0(NC)							
9	led<3>	output		0(NC)							
10	led<2>	output		0(NC)							
11	led<1>	output		0(NC)							
12	led<0>	output		0(NC)							

Buttons at the bottom: OK, Export Constraints..., Import Constraints...

Ports Constraint Entry II

■ Input Delay

- To define an input delay other than the default, click the Input Delay cell for a port and select Define.



■ Output Delay

- To define an output delay other than the default, click the Output Delay cell for a port and select Define.

Ports Constraint Entry III

- Global Buffer
 - To specify insertion of a global buffer for a port or to select automatic global buffer insertion, click the Global Buffer cell for a port and make the appropriate selection.
 - *AUTOMATIC* (default), *DONT USE*.
 - You cannot define global buffers for a design that has the Do Not Insert I/O Pads option selected.
- Pad Direction
 - Use the pull-down list in this cell to specify this port as three-state (bidirectional).
- Resistance
 - Resistance options for a port or pad are specific to the implementation's target device.
 - *NONE* (default), *PULLUP*, *PULLDOWN*

Ports Constraint Entry IV



- Input Register Delay
 - Control whether optimization uses an input delay inside an input register. The use of an input delay reduces the hold time requirement for an input transition.
 - *DELAY* (default), *NODELAY*
- Use I/O Register
 - You can specify that I/O register pads be used on a port where applicable.
 - *TRUE* (default), *FALSE*
- Slew Rate
 - You can select a fast or slow slew rate for an output or bidirectional port.
 - *SLOW* (default), *FAST*

Ports Constraint Entry V

- Pad Location
 - To specify the location of pads for a port, enter it in this cell.
 - Pad locations are vendor-specific.
 - You cannot specify pad locations for a design that has the Do Not Insert I/O Pads option selected.

Modules Constraint Entry I

- Use the Modules constraint table to control optimizing the design hierarchy.
 - Hierarchy
 - Use this setting to control the preservation or elimination of the module's boundary during optimization.
 - Set to *Preserve*, FPGA Express retains the boundary of this module during optimization.
 - The logic of this module is optimized independently of the rest of the design's logic, and the module remains a module in the optimized implementation.

Clocks Paths Ports Modules Xilinx Options							
	Name	Hierarchy	Primitives	Operator Sharing	Optimize for	Effort	Duplicate Register Merge
1	 <default>	Preserve	Preserve	On	Speed	High	Disable
2	 hex2led						

Modules Constraint Entry II

- An *Eliminate* setting allows elimination of the boundary between the module and its container during optimization.
 - FPGA Express optimizes the logic of this module with the logic of the rest of the module's container.
- Default : Eliminate
- Primitives
 - Use this setting to control the preservation or optimization of the primitives instantiated in the module.
 - When set to *Preserve*, FPGA Express retains the boundaries of all the primitives instantiated in the module and uses the implementation of the primitives provided by the architecture vendor.
 - An *Optimize* setting eliminates the boundaries of all the primitives instantiated in the module and FPGA Express optimizes the primitive logic with the logic of the module.
 - default : Preserve.

Modules Constraint Entry III



- Operator Sharing
 - Use this setting to control sharing the operators in the module.
 - *Off*: FPGA Express implements each operator using a separate hardware resource and selects the best implementation of each of these resources.
 - *On*: FPGA Express determines which operators can be implemented sharing the same hardware resource in order to improve the area and speed of your design. It also selects the best implementation of every hardware resource it uses.
 - default : On
- Optimize for Speed or Area
 - Use this setting to control whether you want to optimize for faster speed or smaller area.
 - default : speed

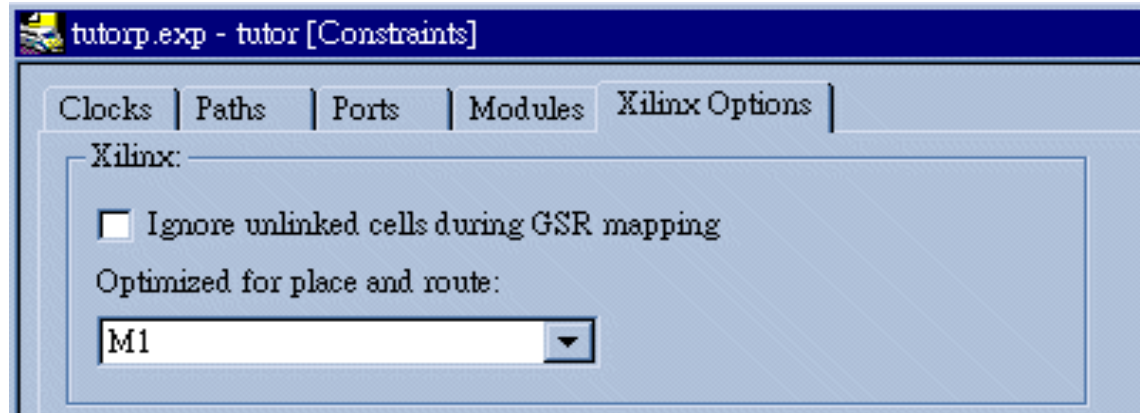
Modules Constraint Entry IV



- Effort
 - Use this setting to control whether the mapping effort for this design is high or low.
 - This option affects timing optimization most with little influence on area.
 - Low effort takes the least time to compile. Use low if you are running a test to check the logic. Low is not recommended if the design must meet area or timing goals.
 - High effort takes longer to compile but should produce better designs. The mapping process proceeds until it has tried all strategies.
 - default : High
- Duplicate Register Merge
 - Removes duplicate cells. Values include Enable, Disable.

Xilinx Options

- Ignore unlinked cells during GSR mapping
 - A check in this check box enables GSR mapping for black boxes.
- Optimized for place and route
 - Select a place and route tool from the drop-down list.

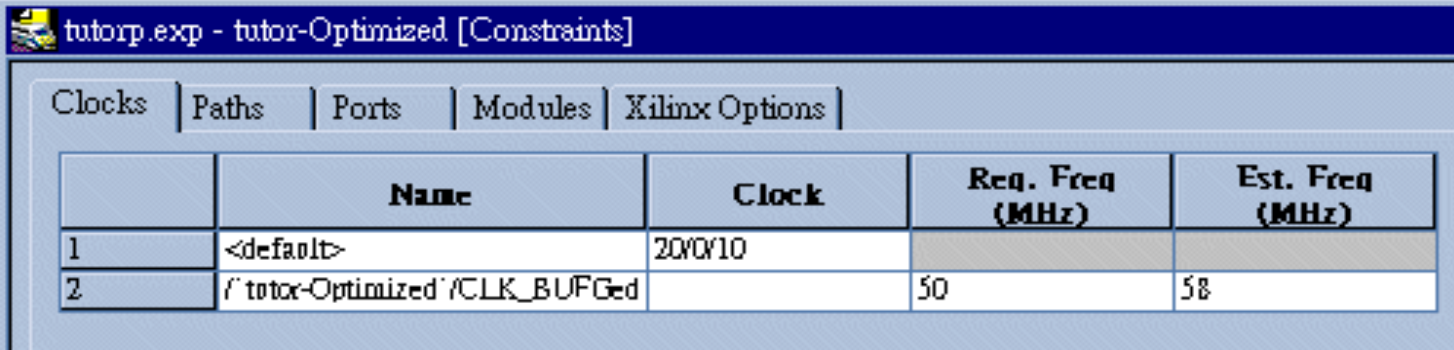


Express Time Tracker

- The Express Time Tracker allows you view estimated performance results after optimization of your designs.
- 1. The Optimized (Constraints) window automatically displays after Synthesis processing if you checked the View Estimated Performance after Optimization box in the Synthesis/Implementation dialog window.
 - Alternatively, you can access the Optimized (Constraints) window via the Versions tab by right-clicking on an optimized structure in the Hierarchy Browser and then selecting View Synthesis Results.
- 2. Click the tabs to access the performance results in the various spreadsheets.
 - If you un-checked Insert I/O pads on the Synthesis/Implementation dialog, only the Models and Xilinx Options tabs are shown. The Clocks, Ports, and Paths tabs apply only to top-level HDL designs.

Analyzing Timing - Clocks

- Check the *Clocks* constraint table to see the maximum clock frequencies FPGA Express calculated for each of the clocks in the design. Clock frequency violations appear in red.



	Name	Clock	Req. Freq (MHz)	Est. Freq (MHz)
1	<default>	200/10		
2	/ 'tutor-Optimized' /CLK_BUFGcd		50	58

Analyzing Timing - Paths

- Check the Paths constraint table for more detail on timing violations.
 - Select a path group to see a list of paths in that group.
 - Select a path from the list to see the details of path composition, cumulative delays, and fanout.

tutorp.exp - tutor-Optimized [Constraints]

Clocks | **Paths** | Ports | Modules | Xilinx Options

	From	To	Req. Delay	Est. Delay
1	All Input Ports	RC - CLK_BUFGed	20	17
2	RC - CLK_BUFGed	All Output Ports	20	9
3	RC - CLK_BUFGed	RC - CLK_BUFGed	20	15

	All Input Ports	RC - CLK_BUFGed	Est. Delay
1	NOTLD	QOUT_reg<0>	8.5
2	NOTLD	QOUT_reg<1>	13.8
3	UPCNT	QOUT_reg<2>	14.1
4	UPCNT	QOUT_reg<3>	14.9
5	UPCNT	QOUT_reg<4>	15.2
6	UPCNT	QOUT_reg<5>	16.1
7	UPCNT	QOUT_reg<6>	16.4
8	UPCNT	QOUT_reg<7>	17.2

	Instance Path	Cell Type	Delay	Fanout
1	NOTLD/NOTLD	IN	0.0	1
2	C146/I	IBUF	0.0	1
3	C146/O	IBUF	1.1	16
4	C011	BQN	7.0	16
5	C010	BQN	8.3	1
6	QOUT_reg<0>:D	DFF	8.5	1
7	QOUT_reg<0>:C	DFF	8.5	8

Analyzing Timing - Ports

Check the Ports constraint table for information about input and output delays.

tutorp.exp - tutor-Optimized [Constraints]

Clocks | Paths | Ports | Modules | Xilinx Options

	Name	Direction	Input Delay (ns)	Input Slack	Output Delay (ns)	Output Slack	Global Buffer	Pad Dir	Resistance	Input Reg Delay	Use I/O Reg	Slew Rate	Pad Loc
1	<default>						AUTOMATIC		NONE	DELAY	TRUE	SLOW	
2	CLK	input	20(IRC, CLK_BUFGcd)	14.4			BUFG						
3	NOTCLR	input	20(IRC, CLK_BUFGcd)	N/A									
4	CLKEN	input	20(IRC, CLK_BUFGcd)	N/A									
5	NOTLD	input	20(IRC, CLK_BUFGcd)	6.2									
6	UPCNT	input	20(IRC, CLK_BUFGcd)	2.8									
7	DI<7>	input	20(IRC, CLK_BUFGcd)	13.4									
8	DI<6>	input	20(IRC, CLK_BUFGcd)	13.4									
9	DI<5>	input	20(IRC, CLK_BUFGcd)	13.4									
10	DI<4>	input	20(IRC, CLK_BUFGcd)	13.4									
11	DI<3>	input	20(IRC, CLK_BUFGcd)	13.4									
12	DI<2>	input	20(IRC, CLK_BUFGcd)	13.4									
13	DI<1>	input	20(IRC, CLK_BUFGcd)	13.4									
14	DI<0>	input	20(IRC, CLK_BUFGcd)	15.3									
15	QO_LO<7>	output			20(IRC, CLK_BUFGcd)	11.2							
16	QO_LO<6>	output			20(IRC, CLK_BUFGcd)	10.8							
17	QO_LO<5>	output			20(IRC, CLK_BUFGcd)	10.8							

Analyzing Resources - Modules



- Check the Modules constraint table for information about the device resources used.
 - Double-click the rows in the Area column for details about cell count.

The screenshot shows the Xilinx ISE software interface. The main window is titled "tutorp.exp - tutor-Optimized [Constraints]". It has tabs for "Clocks", "Paths", "Ports", "Modules", and "Xilinx Options". The "Modules" tab is selected, displaying a table with columns: "Name", "Hierarchy", "Primitives", "Operator Sharing", "Optimize for", "Effort", and "Area".

	Name	Hierarchy	Primitives	Operator Sharing	Optimize for	Effort	Area
1	<default>	Eliminate	Preserve	On	Speed	High	
2	tutor						17

A red arrow points from the "Area" column of the "tutor" row to the "Cell Counts" dialog box. The dialog box is titled "Cell Counts" and contains a table with "Cell Type" and "Count".

Cell Type	Count
BUFG	1
CY4	5
CY4_32	3
CY4_34	1
CY4_42	1
DFF	8
FMAP	17
IBUF	12
OBUF	8
STARTUP	1

At the bottom of the dialog box, there is a checkbox labeled "Counts cumulative for all sub-hierarchy" which is checked. There are "Close" and "Help" buttons at the bottom.

Performing Functional Simulation I



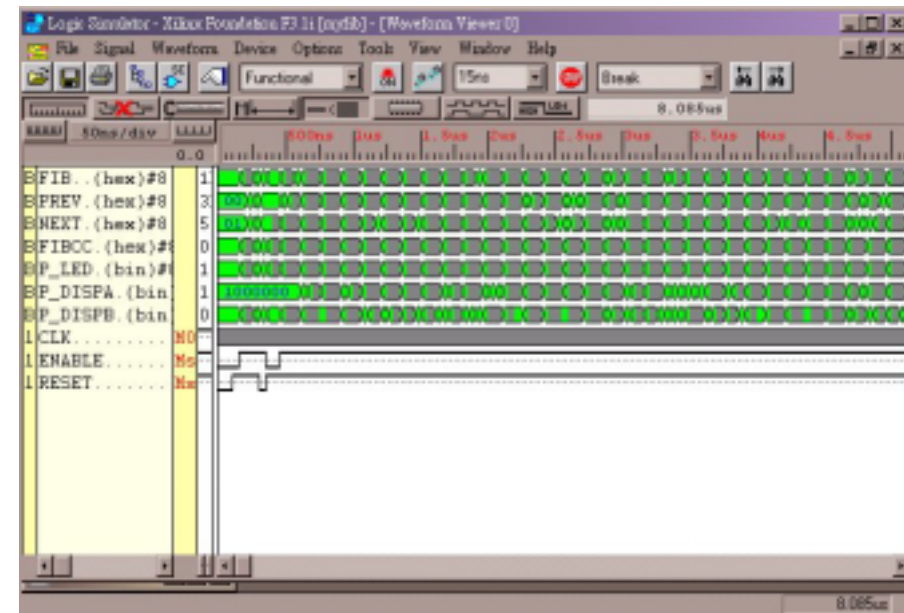
Functional Simulation may be performed to verify that the logic you created is correct.

1. Open the Logic Simulator

- By clicking the Functional Simulation icon in the Simulation box on the Project Manager's Flow tab.



2. The design is automatically loaded into the simulator.



Performing Functional Simulation II

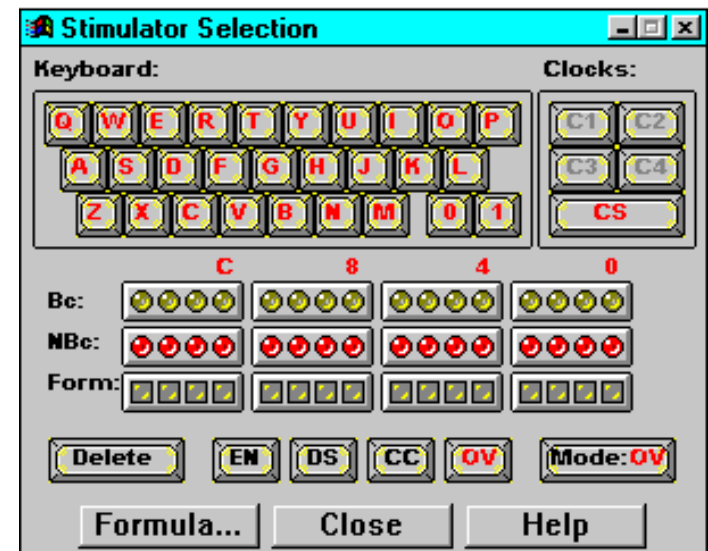


3. Add signals

- By selecting *Signal > Add Signals*.
- From the Signals Selection portion of the Components Selection for Waveform Viewer window, select the signals that you want to see in the simulator.
- Use **CTRL-click** to select multiple signals. Make sure you add output signals as well as input signals.
- Click Add and then Close. The signals are added to the Waveform Viewer in the Logic Simulator screen.

4. Attach Stimulus

- Invoke Stimulator Selection window
 - Select *Signal > Add Stimulators* from the Logic Simulator menu.
- Create the waveform stimulus by attaching stimulus to the inputs.
 - For more details on how to use the Stimulus Selection window, click Help.



Performing Functional Simulation III



■ 5. Run Simulation

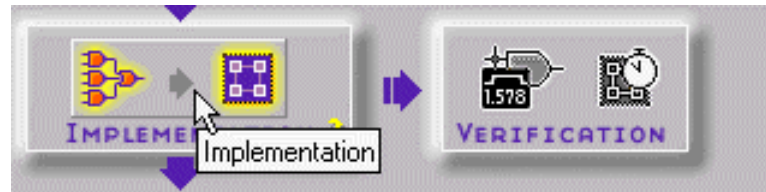
- After the stimulus has been applied to all inputs, click the Simulator Step icon on the Logic Simulator toolbar to perform a simulation step.
- The length of the step can be changed in the Simulation Step Value box to the right of the Simulation Step box. (If the Simulator window is not open, select View > Main Toolbar.)



- 6. Verify that the output waveform is correct.
 - Click the step button repeatedly to continue simulating.
- 7. Save the stimulus for future viewing or reuse

Implementing the Design I

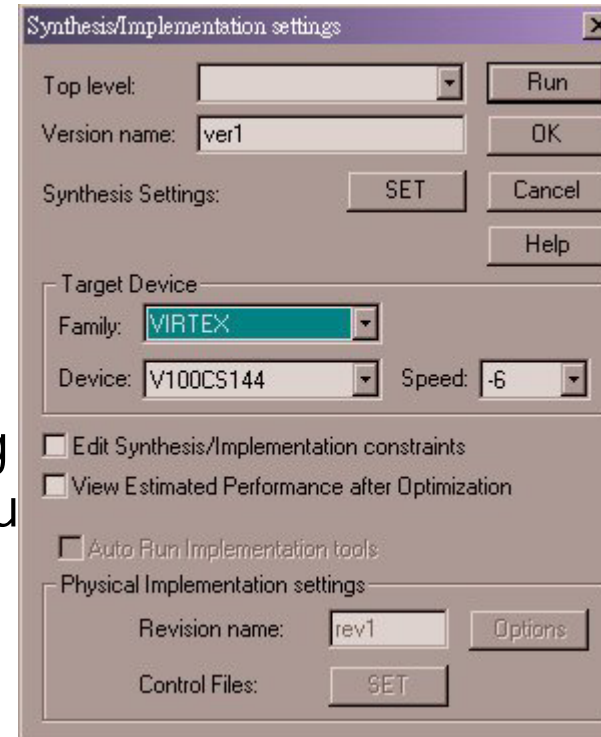
- Design Implementation is the process of translating, mapping, placing, routing, and generating a Bit file for your design.
 - Optionally, it can also generate post-implementation timing data.
- 1. Click the Implementation icon on the Implementation phase button on the Project Manager's Flow tab.



- 2. The Synthesis/Implementation dialog box appears if the design version has been synthesized but no revisions exist.
 - A revision represents an implementation on the version selected for the Synthesis phase.
 - Modify the name in the Revision Name box, if desired. The other settings are the same ones set for the synthesis phase.

Implementing the Design II

- 3. Click the Options button under the Physical Implementation Settings area. The Options dialog box displays.
- 4. Choose any desired implementation option.
 - If you are planning on conducting a timing simulation, select the Produce Timing Simulation Data option.
- 5. Click OK to return to the Synthesis/Implementation dialog box.
- 6. Click Run to implement your design. The Flow Engine displays the progress of the implementation.



Implementing the Design III

- The Project Manager displays a status message when Implementation is complete.
- View the Console tab on the Project Manager window for the results of all stages of the implementation.
- The Versions tab also indicates the status of the implemented revision.
- 7. Review Reports
 - Click the Versions tab in the Hierarchy Browser area of the Project Manager.
 - Select the current revision.
 - Then select the Reports tab on the Project Manager window to review the reports and logs for that revision of the design.
 - Click on the Implementation Report Files icon to view the implementation reports.
 - Click on the Implementation Log File icon to view the Flow Engine's processing log.
- For more information on the Flow Engine, select Help > Foundation Help Contents > Flow Engine.

Verifying the Design I

- The Timing Analyzer or the Timing Simulator can be used to perform a timing analysis on your design.
 - The *Timing Analyzer* performs a static timing analysis that does not include insertion of stimulus vectors.
 - The *Timing Simulator* uses input stimulus to run the simulation.
- Performing a Static Timing Analysis
 - 1. Click the Timing Analyzer icon in the Verification box on the Project Manager's Flow tab.



- 2. Perform a static timing analysis on mapped or place and routed designs.
 - For **FPGAs**, you can perform a post-MAP or post-place timing analysis to obtain rough timing information before routing delays are added.

Verifying the Design II

- For **CPLDs**, you can perform a post-implementation timing analysis after a design has been implemented.
- For details on how to use the Timing Analyzer, select *Help > Foundation Help Contents > Timing Analyzer*.

■ Performing a Timing Simulation

- 1. Open the Timing Simulator by clicking the Timing Simulation icon in the Verification box on the Project Manager's Flow tab. The implementation timing netlist will be loaded into the simulator.



- 2. Refer to the "Performing Functional Simulation (Optional)" section earlier in this chapter for instructions on simulating the design. (The operation of the simulator is the same for functional and timing simulation.)
- 3. If you have already saved test vectors (for instance, in the functional simulation), you may load these vectors into the timing simulator by selecting *File > Load Waveform*.

Programming the Device

- 1. Invoke Program Box
 - Click the Device Programming icon in the Programming box on the Project Manager's Flow tab.



- 2. From the Select Program box, choose the Hardware Debugger, the PROM File Formatter, or the JTAG Programmer.
 - For CPLD designs, use the JTAG Programmer.
 - For instructions, select *Help > Foundation Help Contents > Advanced Tools > JTAG Programmer*.
 - For FPGA designs, use the JTAG Programmer, Hardware Debugger, or PROM File Formatter.
 - For instructions, select *Help > Foundation Help Contents > Advanced Tools and then select the desired tool*.

Spartan II Architecture

- Describe the capabilities of the Spartan-II device resources
- Improve your design performance and device utilization using the Spartan-II CLB, memory, and routing resources

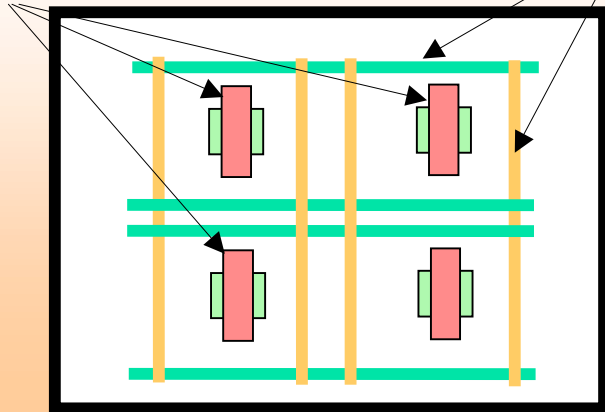
Overview

- Spartan (5.0 Volt) family introduced in Jan '98
 - Fabricated on 0.5 μ process technology
- Spartan-XL (3.3 Volt) family introduced in Nov '98
 - Fabricated on advanced 0.35 μ process technology
 - Power management features
 - Higher performance over Spartan (5.0 Volt)
- Spartan-II (2.5 Volt) family introduced in Jan 2000
 - Fabricated on new advanced 0.22/0.18 μ process technology
 - Power management features
 - Higher performance over Spartan
- Similar to the Virtex device family
 - Lower density range than Virtex
 - Minimal overlap between Spartan-II and Virtex
 - Higher density than Spartan (15K-150K gates)
 - 4 DLLs and same Block RAM capacity as Virtex
 - No differential I/O standards as in Virtex-E
 - Power down mode (in place of thermal diode)

Spartan-II Technology

- Combined 0.22/0.18m process
 - 2.5V technology
- Die size about 1/3 smaller than equivalent Virtex device

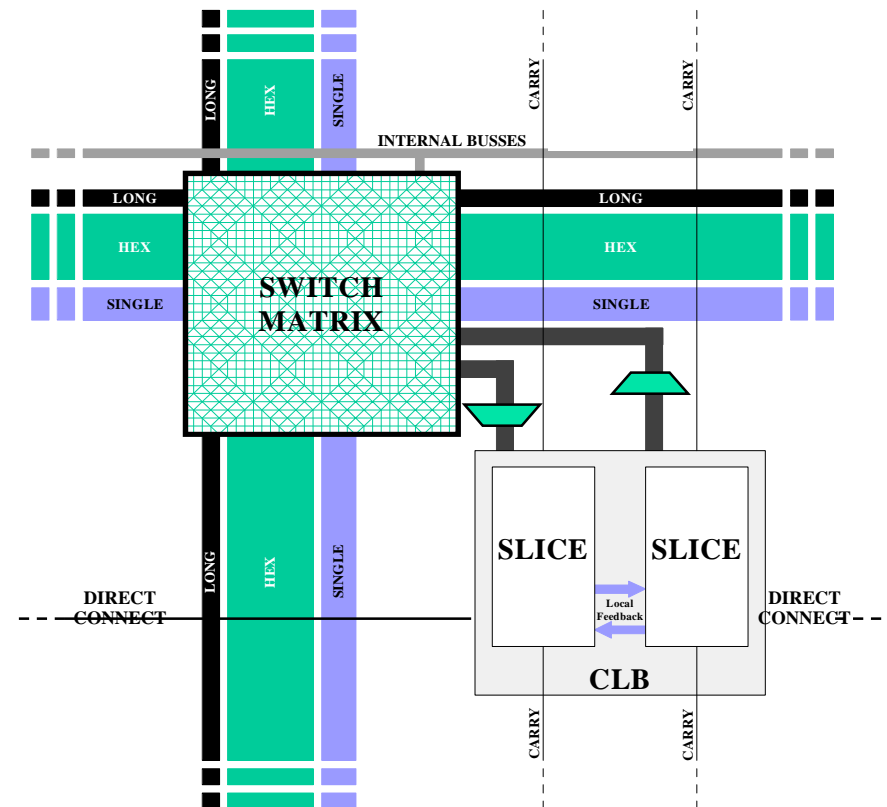
Transistor gates 0.22 μ
- allows 2.5 V supply



All other features 0.18 μ
- small size
- low capacitance
- performance
- low power

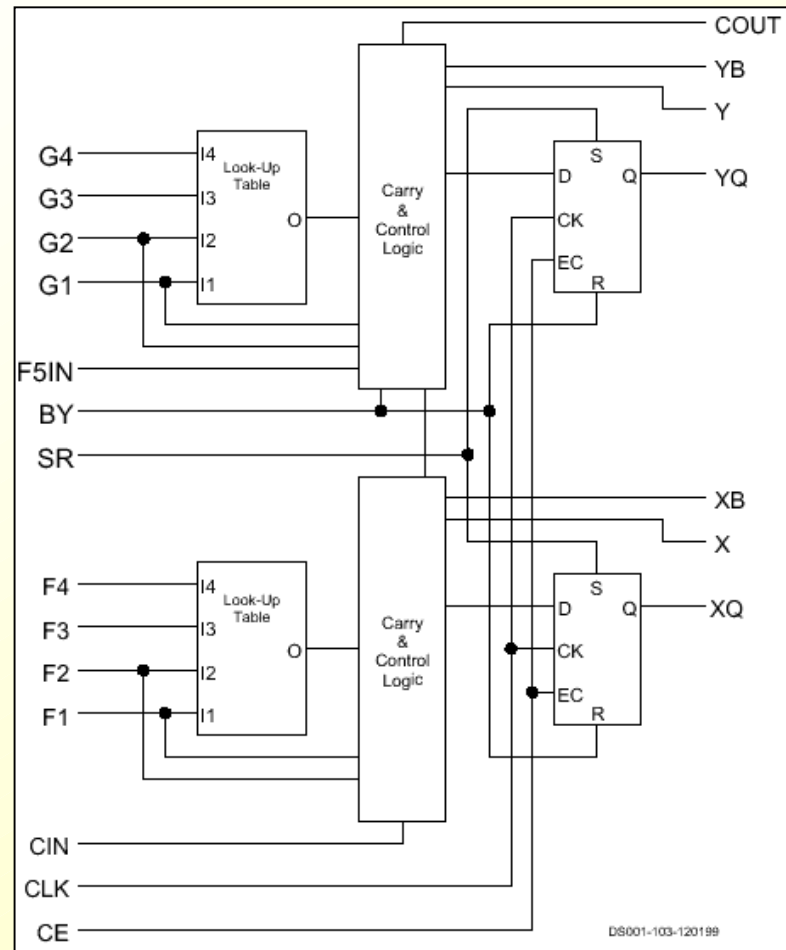
The CLB Tile

- CLB Tile is composed of:
 - Switch matrix
 - Configurable Logic Block and associated general routing resources
 - IMUX and OMUX
- All CLB inputs have access to interconnect on all 4 sides
- Wide single CLB functions
- Fast local feedback within the CLB & direct connects to east and west CLBs
 - Allows for the creation of 13 input functions in a single CLB



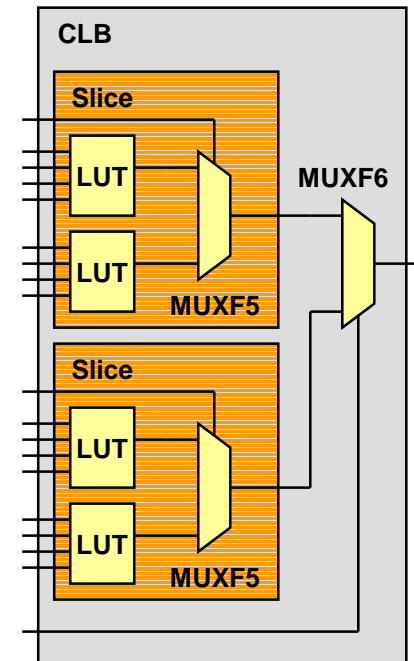
Simplified CLB Structure

- Two Slices in Each CLB
 - Each Spartan-II slice is similar to the XC4000 CLB
- Two BUFTs associated with each CLB, accessible by all 8 CLB outputs
- Carry Logic runs vertically, up only



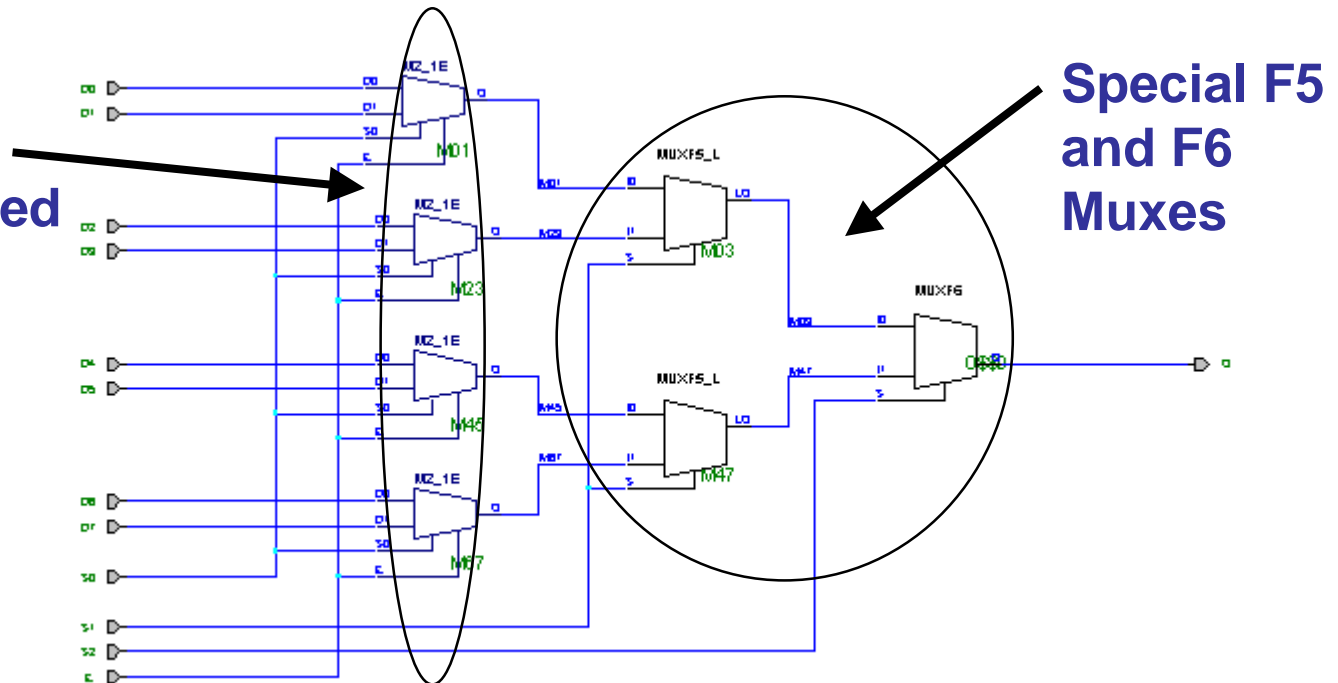
Connecting Function Generators

- Some functions need several function generators
 - F5 MUXs connect pairs of function generators
 - Any function of 5 inputs or some up to 9 inputs
 - F6 MUXs connect all 4 function generators
 - Any function of 6 inputs or some up to 17 inputs



F5 and F6 Mux Application

Muxes
Implemented
in LUTs

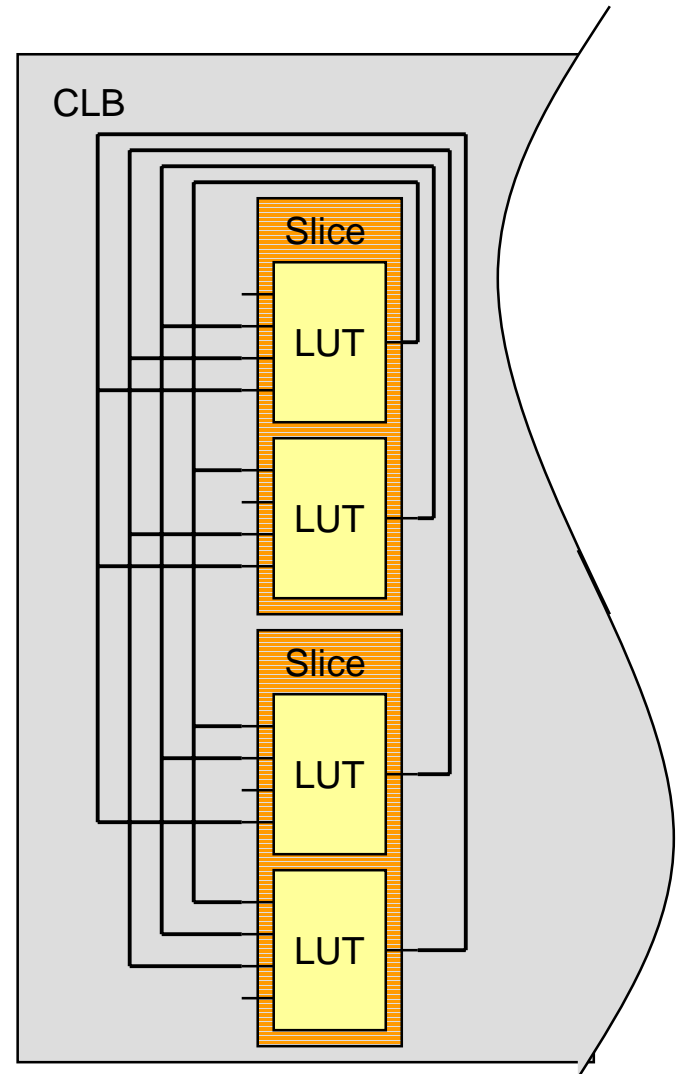


Implement an 8x1 Mux in a single CLB

- Dense structure utilizes resources efficiently
- 2 Logic Levels and 1 Local Interconnect Yield a 2.5ns Max Delay

Fast Local Feedback Routing

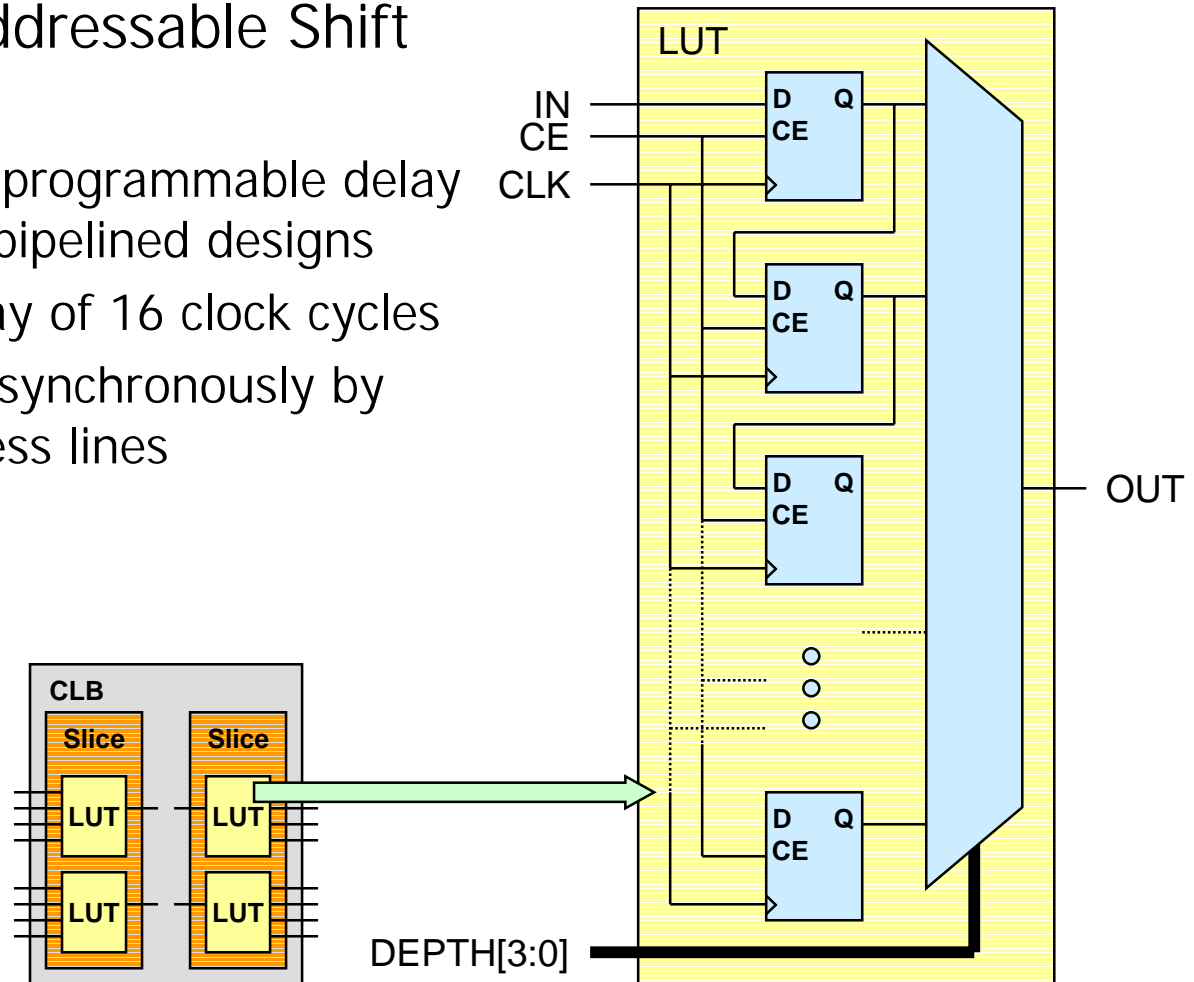
- Each LUT output can connect to three other LUTs in the same CLB
 - 100ps to 300ps maximum delay
 - Create 13-input functions within the same CLB - 2.5ns total delay
 - Synthesis tools use fast connects on critical paths



Look-Up Table Shift Registers

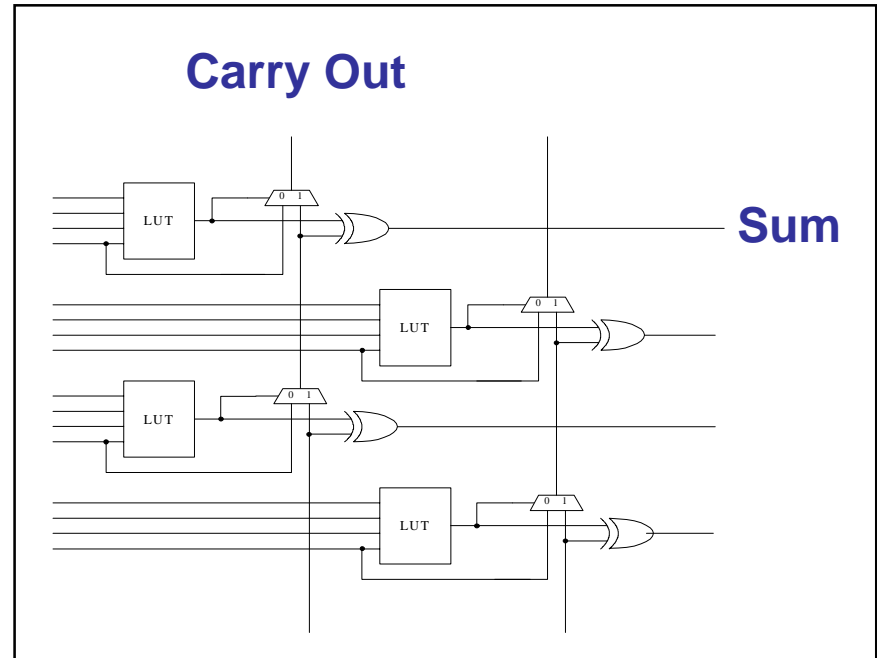


- Dynamically Addressable Shift Register (SRL)
 - Ultra-efficient programmable delay for balancing pipelined designs
 - Maximum delay of 16 clock cycles
 - Can be read asynchronously by toggling address lines

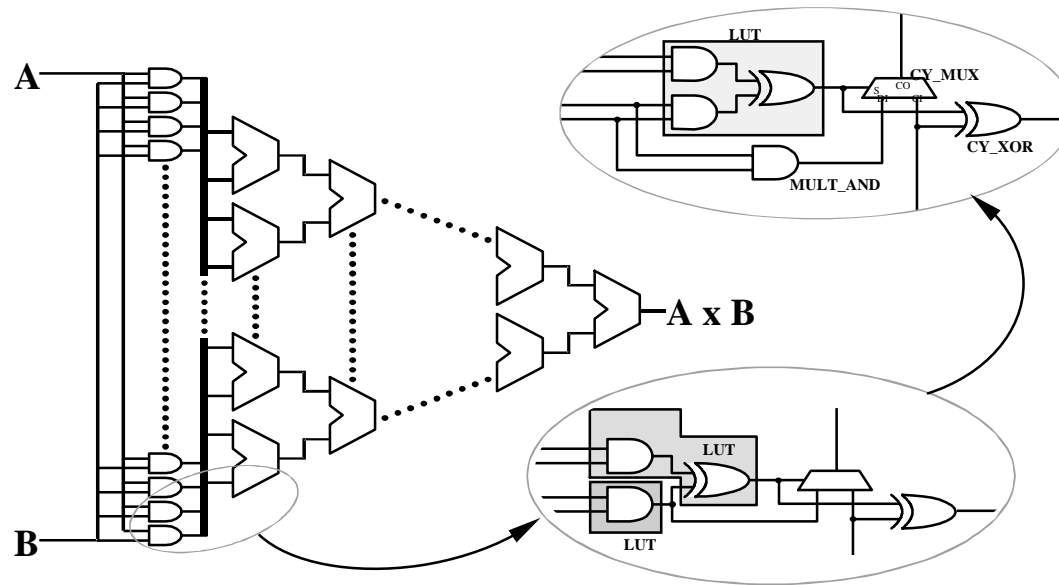


Fast Carry Logic

- Simple, fast & complete Arithmetic Logic
 - Dedicated XOR gate for single level sum completion
 - All synthesis tools can infer carry logic



Dedicated Multiplier Fabric



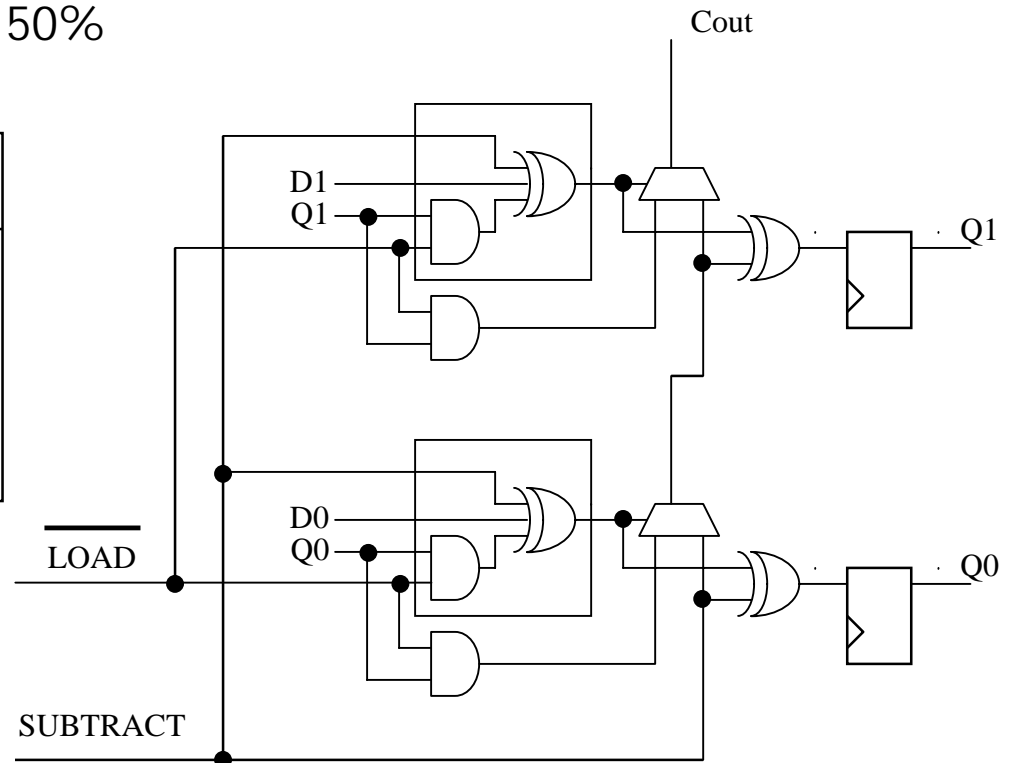
- Highly efficient *Shift & Add* implementation
 - Tree style multipliers require that separate function generator create the multiply and add
 - Spartan-II enables a 30% reduction in area for a 16x16 multiply & 1 less logic level

Spartan-II Loadable Functions

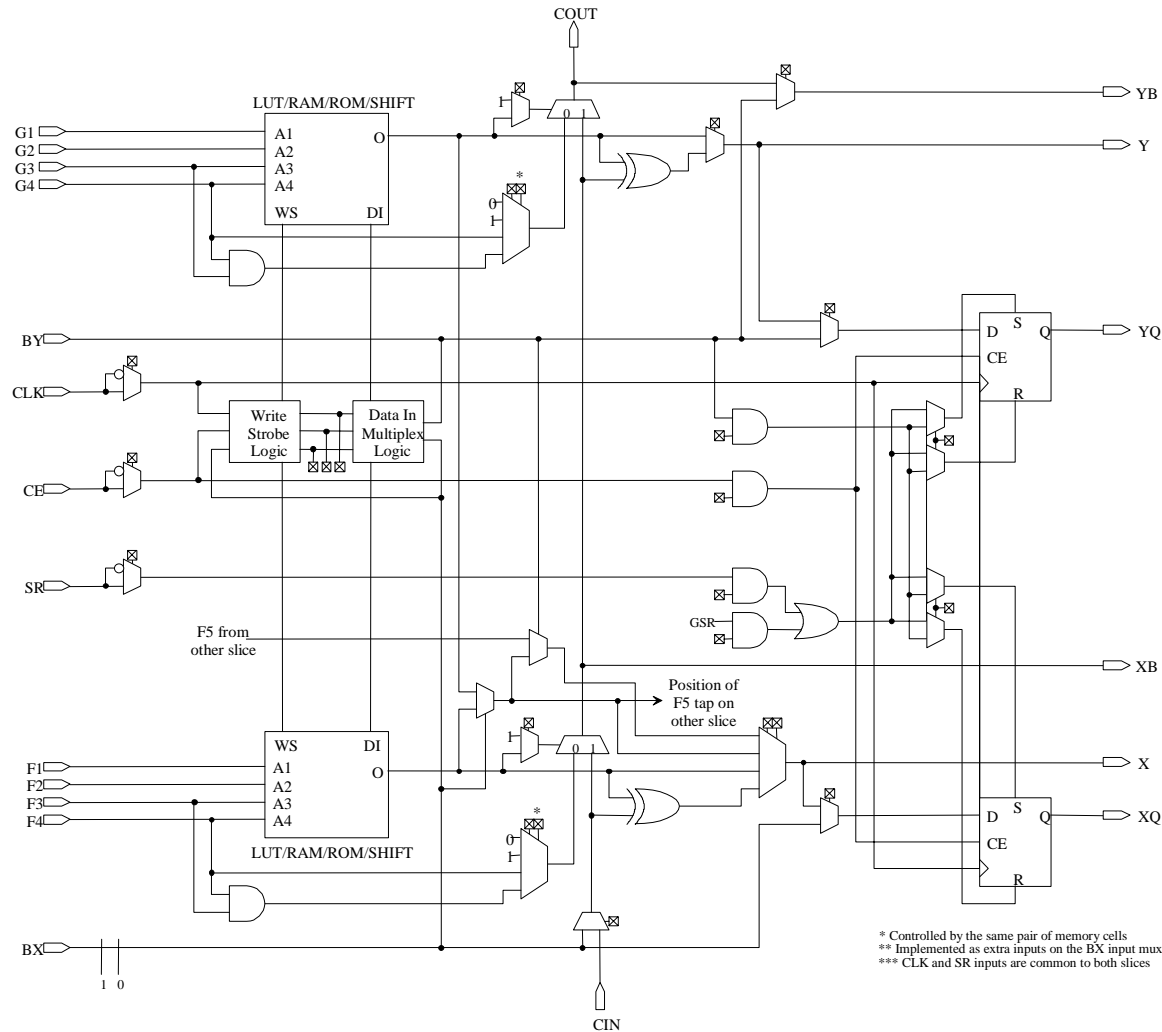


- The MULT_AND can be used for creating compact and fast loadable up/down counters
- The MULT_AND disables the carry bit propagation
- When compared to an implementation in an XC4000, the required resources are reduced by 50%

<u>LOAD</u>	SUBTRACT	Function
0	0	$Q \leq D$
0	1	$Q \leq D$
1	0	$Q \leq Q+D$
1	1	$Q \leq Q-D$

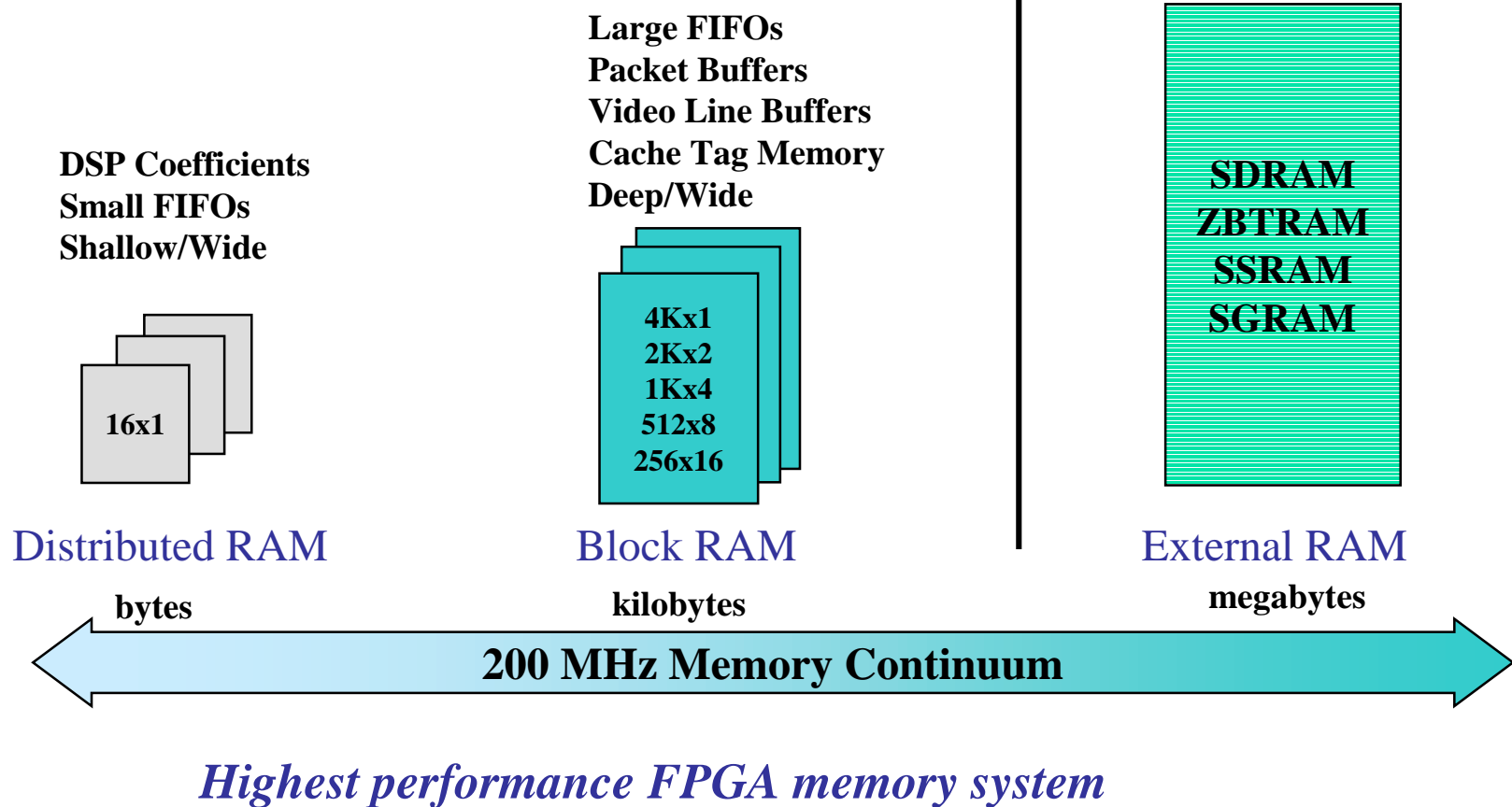


Detailed Slice Structure



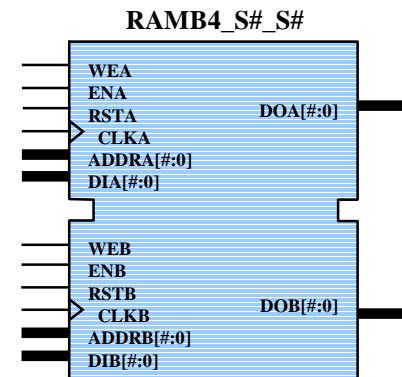
Data Storage Hierarchy

Spartan-II On-Chip SelectRAM+™ Memory



Block RAM

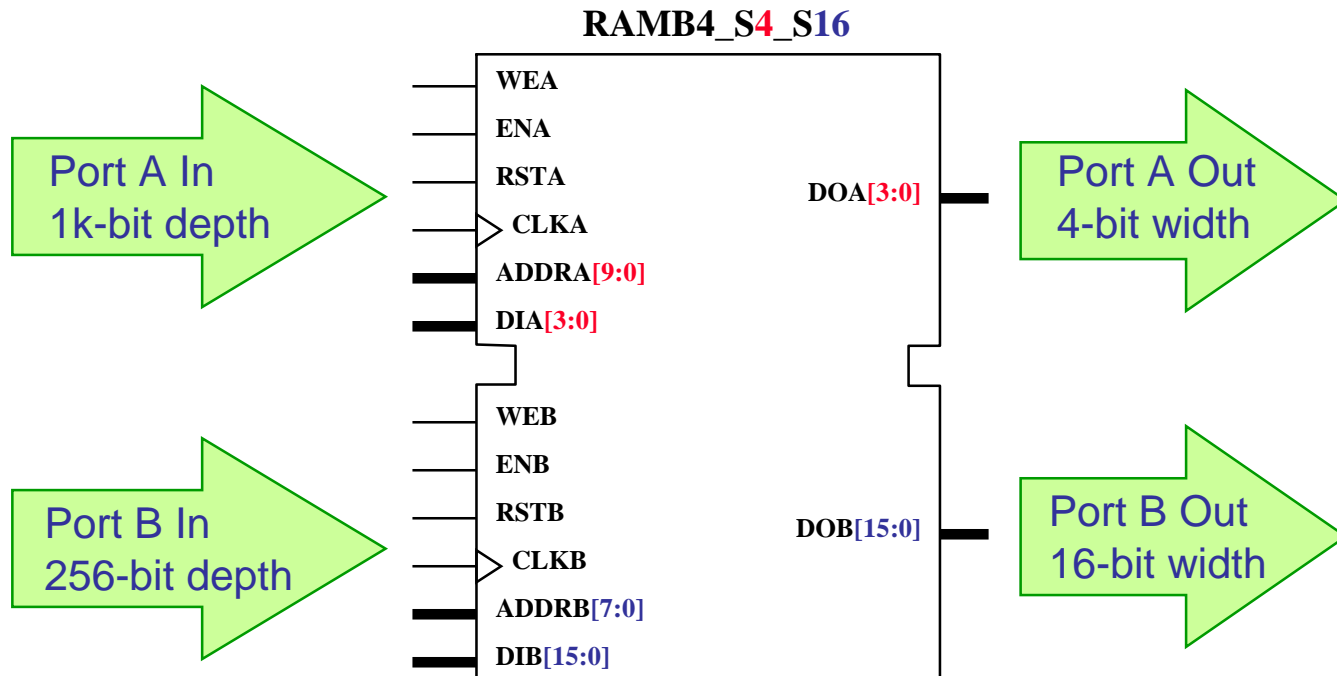
- Up to 12 dual-ported 4096-bit RAM Blocks
 - Synchronous read and write
 - Located on left & right sides with 1 block every 4 rows
- True dual-port memory
 - Each port has synchronous read and write capability
 - Different clocks for each port
- Synchronous Reset & INIT Values
 - State machines, decodes, serial to parallel conversion, etc.
- CORE Generator currently supports creation of single port and dual port Block RAMs



Allowed Widths

ADDR	DATA	#/Width	Depth
(11:0)	(0:0)	1	4096
(10:0)	(1:0)	2	2048
(9:0)	(3:0)	4	1024
(8:0)	(7:0)	8	512
(7:0)	(15:0)	16	256

Block RAM



- Each Dual Port can be configured with a different width (configurable aspect ratio)
- Library name specifies the size and port configuration

Review Questions

- How can the dedicated muxes improve design performance?
- Can RAMs be made that are larger than 4096 bits? (T/F)
- What dedicated resource allows for the fast creation of any function of up to 13 inputs?

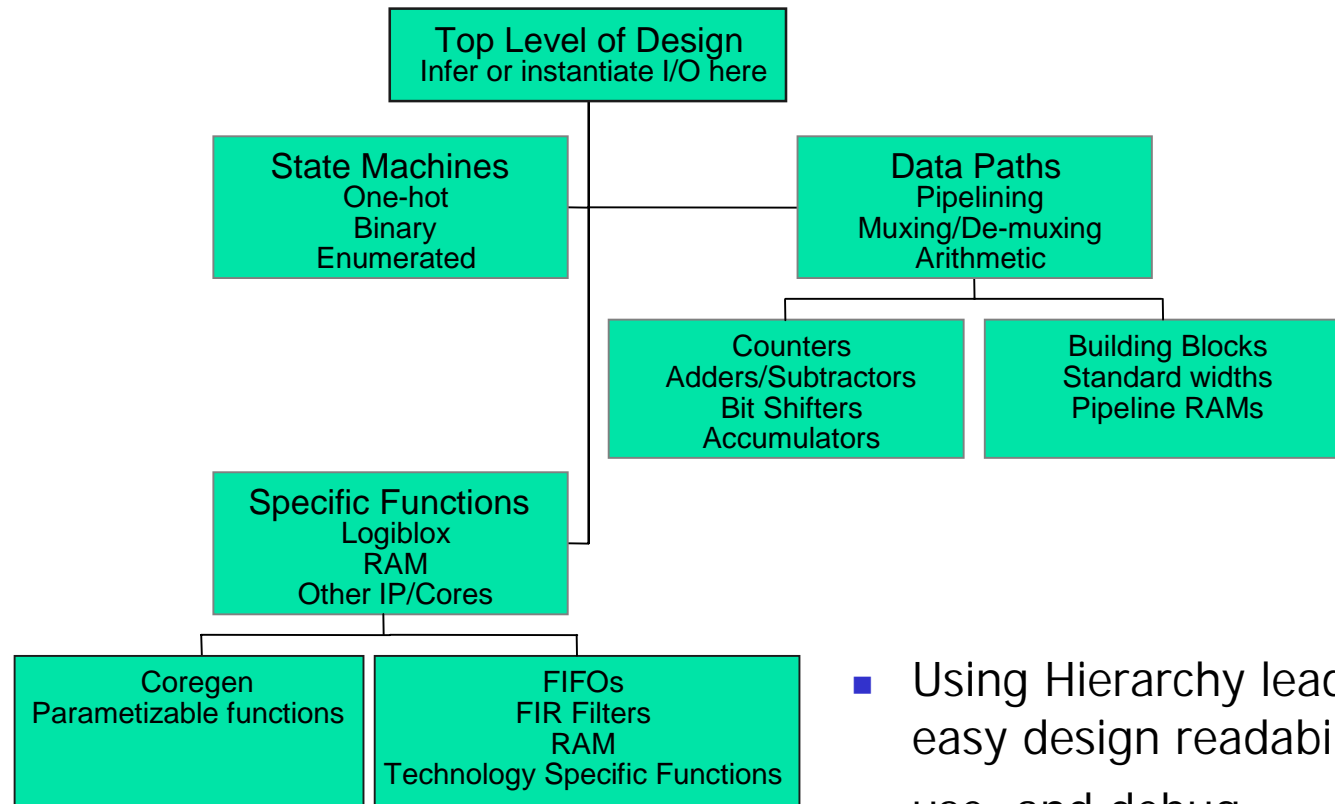
Summary

- The Spartan-II CLB and its local feedback routing resources, enables the creation of high fan-in functions (up to 13 inputs) in a single clb
- The Carry Logic resources creates very fast and efficient arithmetic function
- The dedicated muxes enable the creation of large muxes with a very short delay
- The Dedicated Multiplier Fabric enables the creation of fast and efficient DSP functions
- Spartan II contains distributed and block RAM for a variety of application
- The Block RAM is 4kBits of true-dual port memory
- The DLL eliminates internal clock and board level clock distribution delay

Introduction to FPGA Design

- Effectively use hierarchy
- Increase circuit reliability and performance by applying synchronous design techniques

Hierarchical Design



- Using Hierarchy leads to easy design readability, re-use, and debug

Benefits of Using Hierarchy

- Use best design entry method for each type of logic
- Design readability
 - Easier to understand design functionality and data flow
 - Easier to debug
- Easy to reuse parts of a design
- Synthesis tool benefits
 - Will be covered in a later section

Design Entry Methods

- Use HDL for:
 - State machines
 - Control logic
 - Bussed functions
- Use schematics for:
 - Top level design
 - Data path logic
 - Manually optimized logic
- “*Mixed mode*” designs utilize the best of both worlds

Design Readability

- Choose hierarchical blocks that have:
 - Minimum of routing between blocks
 - Logical data flow between blocks
- Keep clock domains separated
 - Makes the interaction between clocks very clear
- Choose descriptive names for blocks and signals

Design Reuse

- Build a set of blocks that are available to all designers
 - Register banks
 - FIFOs
 - Other standard functions
 - Custom functions commonly used in your applications
- Name blocks by function and target Xilinx family
 - Easy to locate the block you want
 - Example: REG_4X8_SP (bank of four 8-bit registers, targeting Spartan)
- Store in a separate directory from the Xilinx tools
 - Prevents accidental deletion when updating tools

Why Synchronous Design?

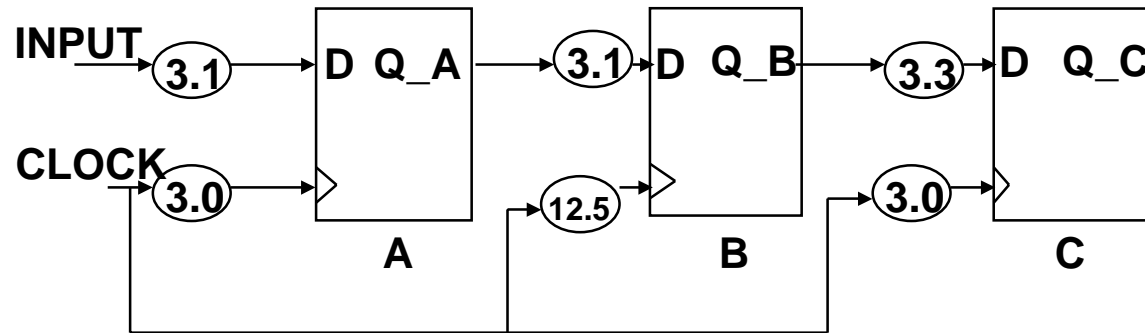
- Synchronous circuits are more reliable
 - Events are triggered by clock edges which occur at well-defined intervals
 - Outputs from one logic stage have a full clock cycle to propagate to the next stage
 - Skew between data arrival times is tolerated within the same clock period
- Asynchronous circuits are less reliable
 - A delay may need to be a specific amount (e.g. 12ns)
 - Multiple delays may need to hold a specific relationship (e.g. DATA arrives 5ns before SELECT)

Design Tips

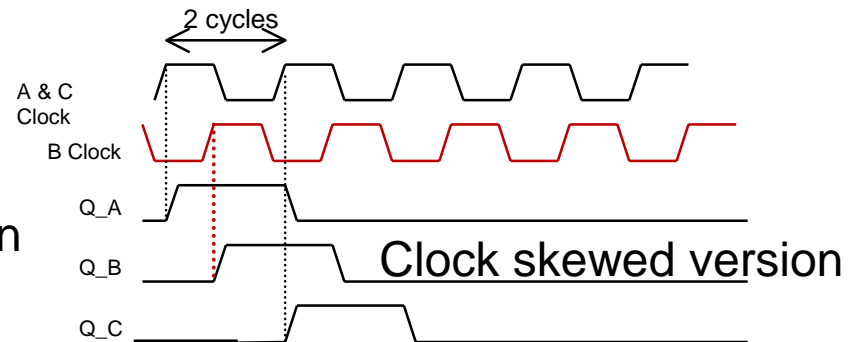
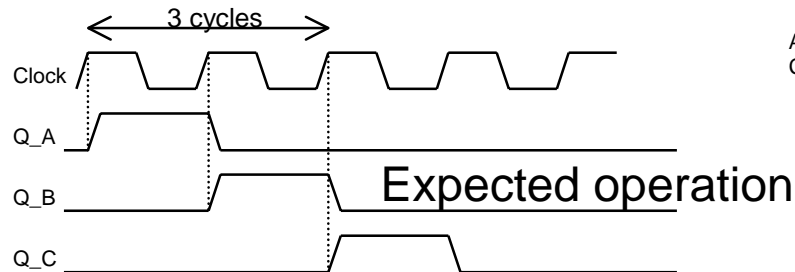
Xilinx FPGA Design

- Reduce clock skew
- Clock dividers
- Avoid glitches on clocks and asynchronous set/reset signals
- The Global Set/Reset network
- Select a state machine encoding scheme
- Access carry logic
- Build efficient counters

Clock Skew



- This shift register will not work because of clock skew!



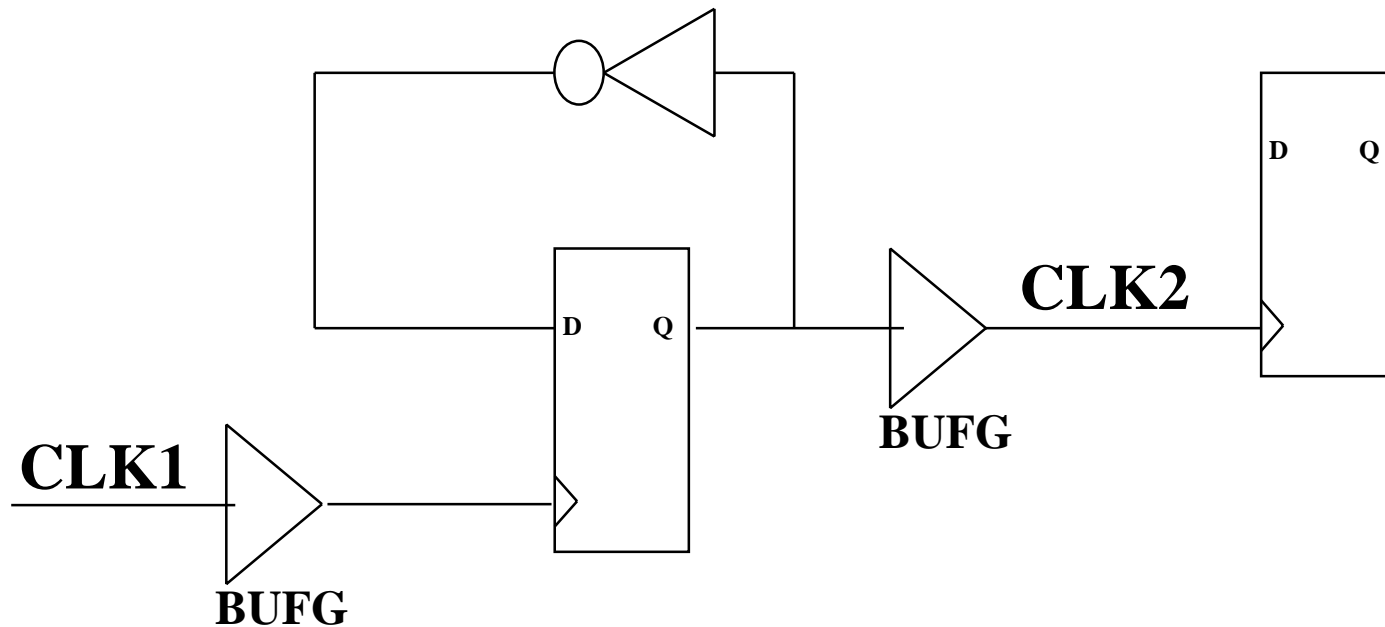
Global Buffers to Reduce Clock Skew



- Global buffers are connected to dedicated routing
 - This routing network is balanced to minimize skew
- All Xilinx FPGAs have global buffers
- Different types of global buffers
 - XC4000E/L and Spartan have 4 BUFGPs and 4 BUFGSs
 - XC4000EX/XL/XV have 8 BUFGLSs
 - Virtex has 4 BUFGs or BUFGDLLs
- You can always use a BUFG symbol and the software will choose an appropriate buffer type
 - Most synthesis tools can infer global buffers onto clock signals

Traditional Clock Divider

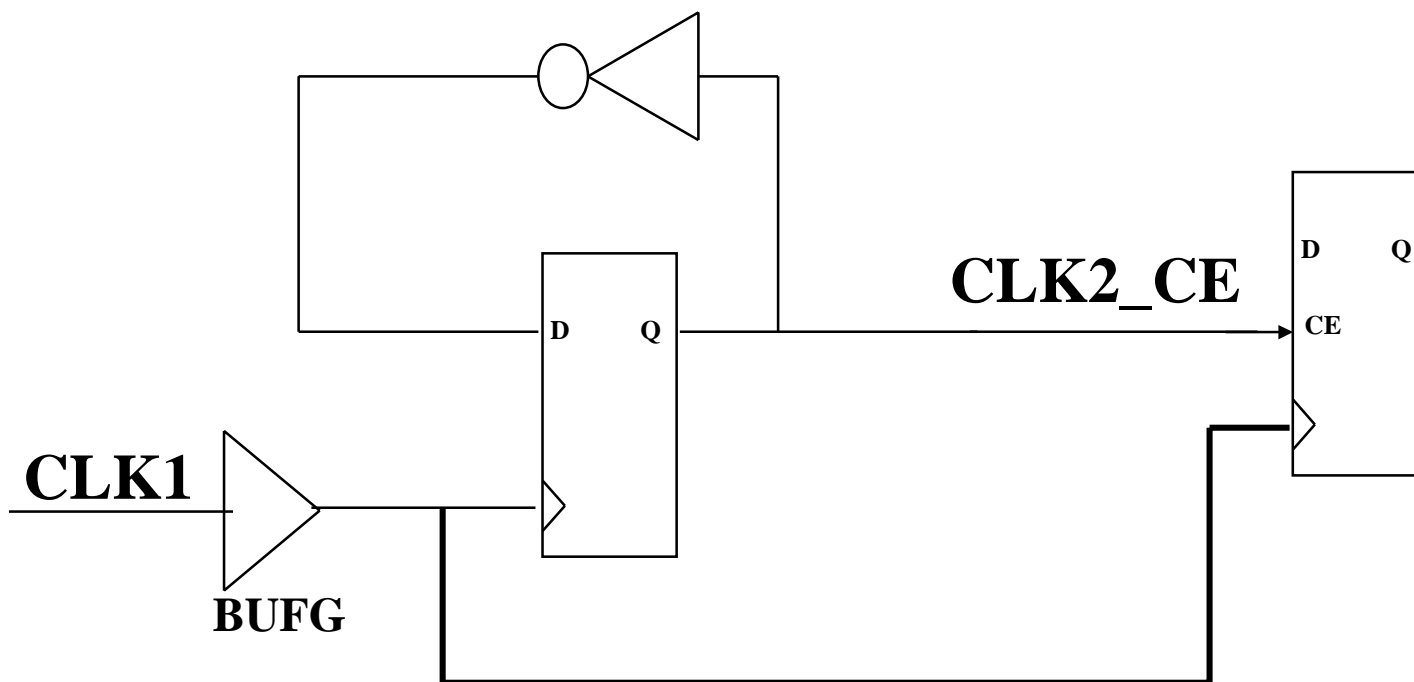
- Introduces clock skew between CLK1 and CLK2
- Uses an extra BUFG to reduce skew on CLK2



Recommended Clock Divider



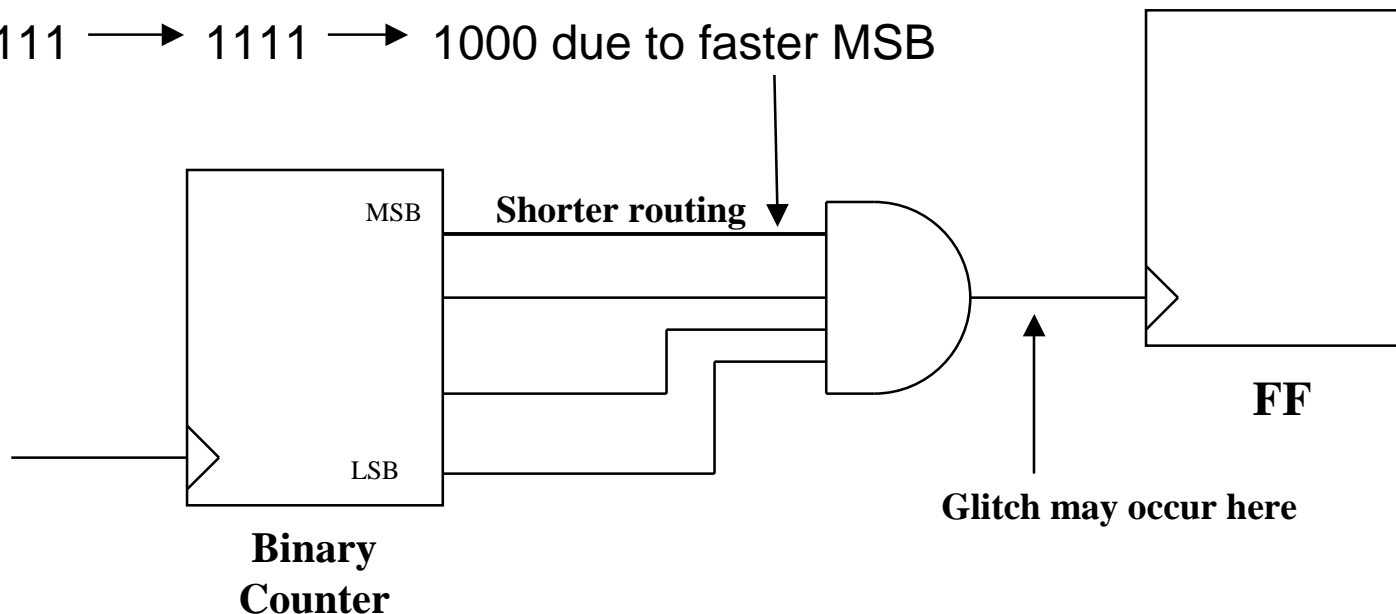
- No clock skew between flip-flops



Avoid Clock Glitches

- Because flip-flops in today's FPGAs are very fast, they can respond to very narrow clock pulses
- Never source a clock signal from combinatorial logic
 - Also known as “gating the clock”

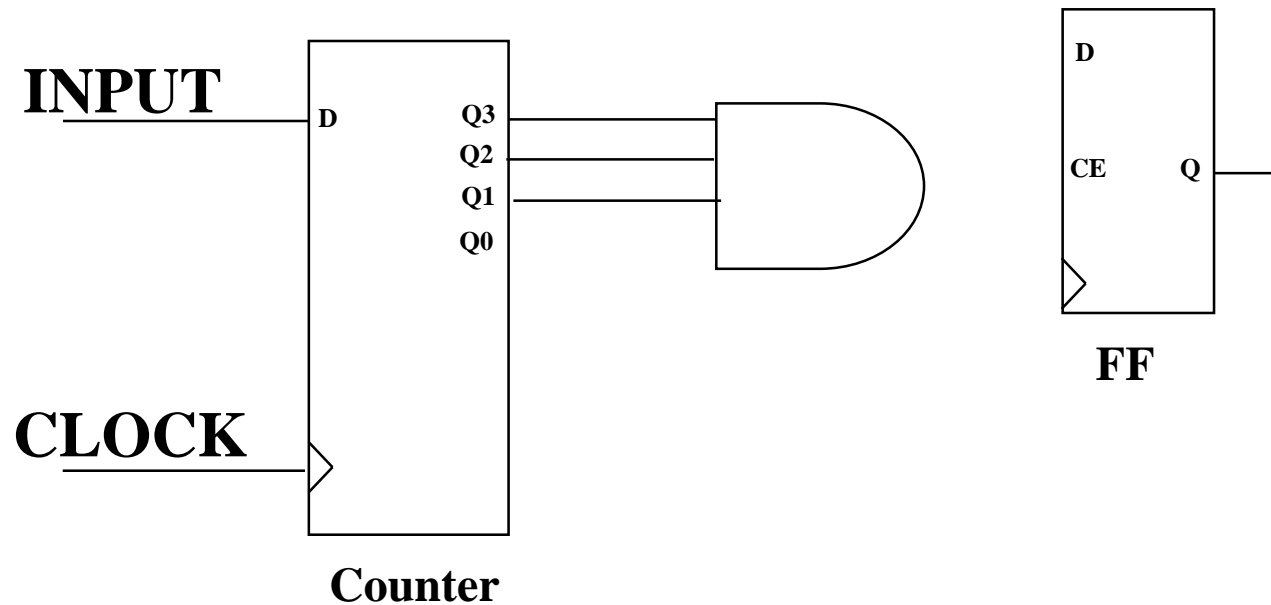
MSB
0111 → 1000 transition can become
0111 → 1111 → 1000 due to faster MSB



Avoid Clock Glitches: Exercise

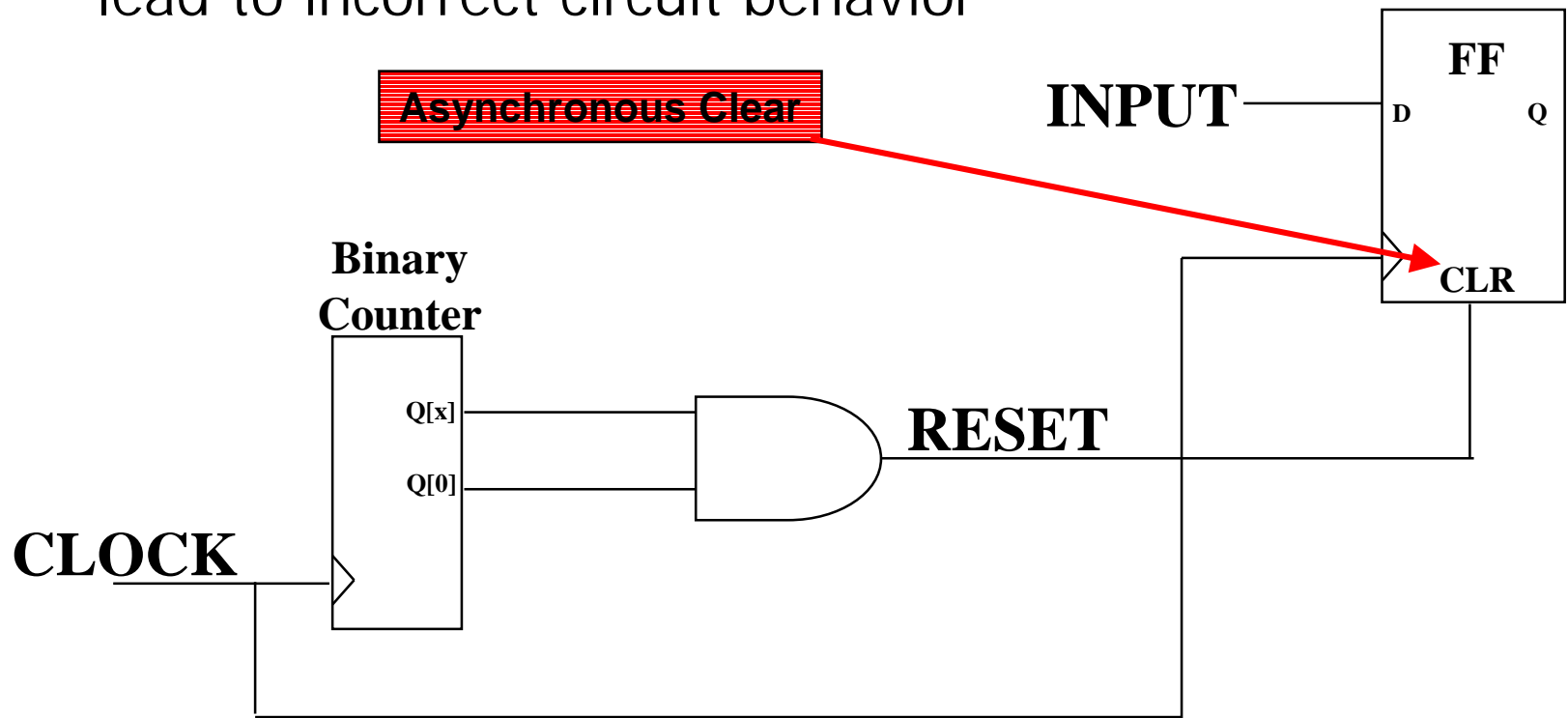


- Complete the circuit to create the same function as the previous slide, but without glitches on the clock



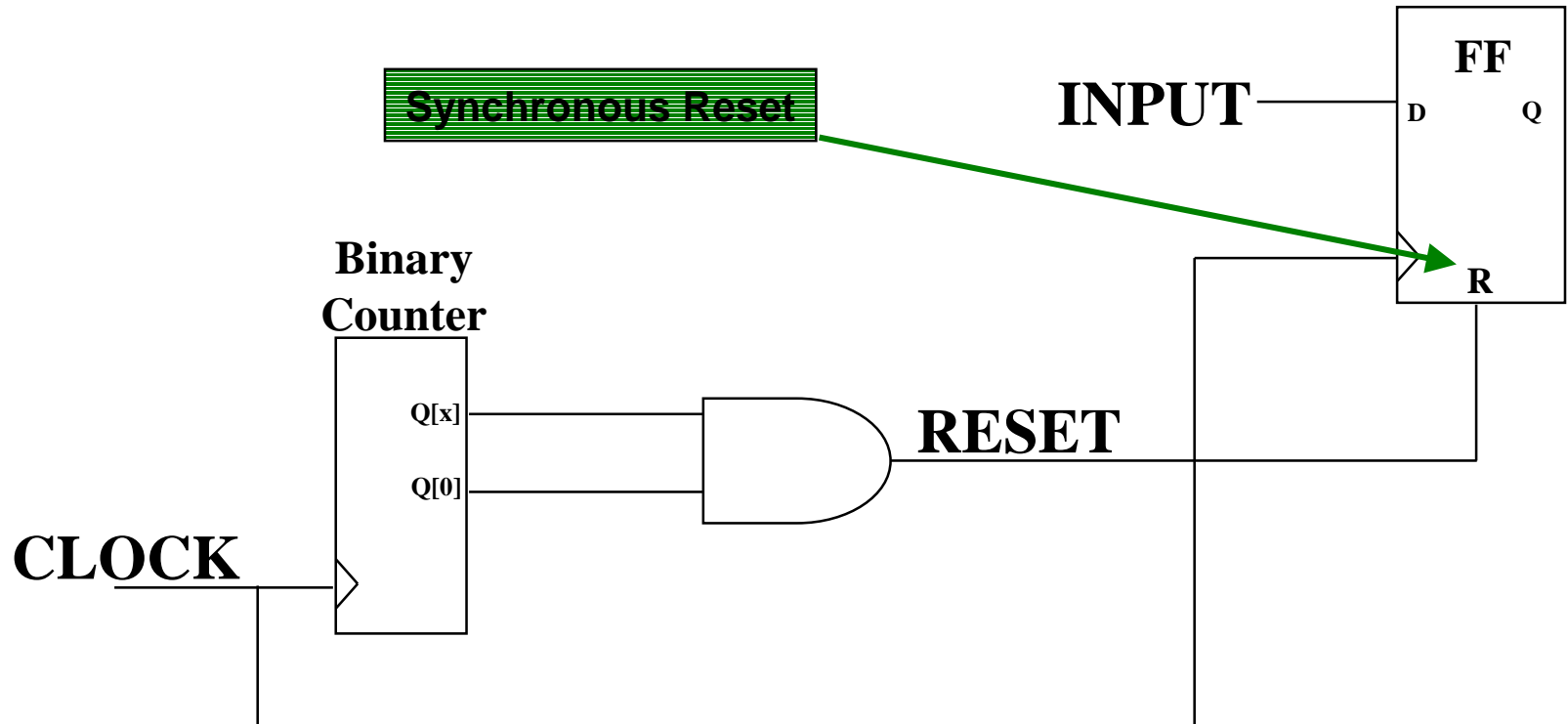
Avoid Set/Reset Glitches

- Glitches on asynchronous clear or preset inputs can lead to incorrect circuit behavior



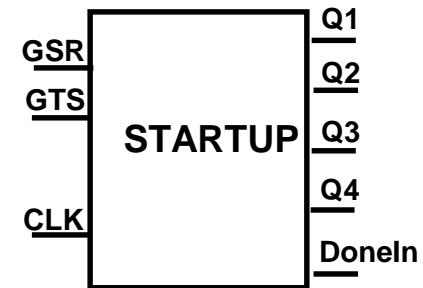
Avoid Set/Reset Glitches

- Convert to synchronous set or reset when possible



Global Set/Reset (GSR)

- Automatically connects to all CLB and IOB flip-flops via a dedicated routing network
 - Saves general routing resources for your design
- Access GSR by instantiating the STARTUP primitive
 - Some synthesis tools can infer GSR
- Use GSR when you have a chip-wide asynchronous reset signal
- Do not use GSR if:
 - You are targeting Virtex
 - Plenty of general routing is available, and is faster than GSR
 - You don't have a chip-wide reset signal
 - You want some flip-flops to have synchronous set/reset only
 - You don't want to reset IOB flip-flops



State Machine Encoding

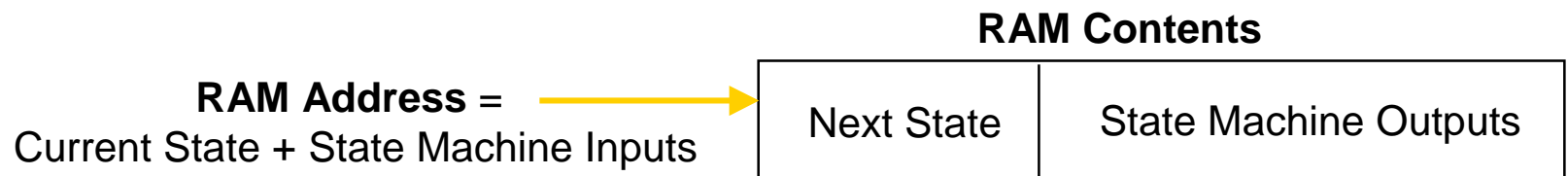
- Two main FSM encoding schemes
- Full Encoding: minimum number of flip-flops possible
 - Example: $S1 = 00$, $S2 = 01$, $S3 = 10$, $S4 = 11$
 - State assignments may follow a simple binary count sequence (Binary Encoding), or a more complex sequence
- One-Hot Encoding: one flip-flop for each state
 - $S1 = 0001$, $S2 = 0010$, $S3 = 0100$, $S4 = 1000$

Encoding Considerations

- Full encoding
 - Fewer flip-flops
 - Uses logic to decode the present state
 - Can create multiple logic levels, decreasing performance
 - Careful state assignment may simplify decoding
- One-Hot encoding
 - More flip-flops
 - No logic required to decode current state
 - Can simplify next-state logic, increasing performance
- Other encoding schemes fall in between the extremes
 - Can be used to balance speed and area considerations

FPGAs and State Machines

- For higher performance, use One-Hot Encoding
 - FPGAs have lots of flip-flops
- Virtex: Use Block SelectRAM to implement fast, complex, fully encoded state machines
 - Initialize the RAM with a “microcode” program



Carry Logic

- Use carry logic to increase arithmetic speed
 - XC4003E-3 Examples
 - 16-bit adder without carry 10 CLBs & 38Mhz
 - 16-bit adder using carry 8 CLBs & 63Mhz
 - 32-bit adder without carry 21 CLBs & 27Mhz
 - 32-bit adder using carry 17 CLBs & 44Mhz
- Accessing carry logic
 - Carry-based library macros
 - ACCx accumulators
 - ADDx adders
 - ADSUx adder/subtractors
 - CCx counters
 - COMPMCx magnitude comparators
- Many synthesis tools can infer carry logic for arithmetic functions
 - Addition ($SUM \leq A + B$)
 - Subtraction ($DIFF \leq A - B$)
 - Comparators (if $A < B$ then...)

Efficient Counters



- Prescaler for ultra-fast, non-loadable counters
 - LSBs toggle quickly
 - Remaining bits have more time to settle
- Gray Code or Johnson counter if decoding outputs
 - One bit changes per transition to eliminate glitches
- Linear feedback shift register for speed
 - Useful when terminal count is all that is needed
 - Also when any regular sequence is acceptable (e.g. FIFO address counter)

Review Questions

- List three benefits of hierarchical design
- Why should you use global buffers for your clock signals?

Summary

- Proper use of hierarchy aids design readability and debug
- Synchronous designs are more reliable than asynchronous designs
- FPGA design tips
 - Global clock buffers and DLLs eliminate skew
 - Avoid glitches on clocks and asynchronous set/resets
 - FSM encoding scheme can affect design performance
 - Increase performance of arithmetic functions by using carry logic
 - Consider different counter styles to meet your design needs

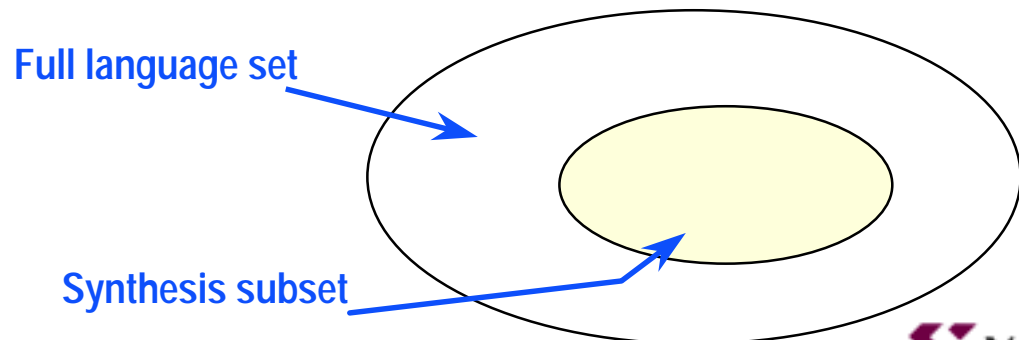
HDL Coding

Generic Coding Techniques & Considerations

HDL Synthesis Subset
HDL Coding Techniques & Examples
Considerations for Synthesis

HDL Synthesis Subset

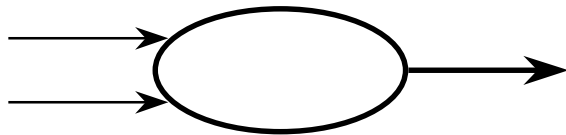
- Verilog & VHDL (Very High Speed Integrated Circuit Hardware Description Language) are hardware description and simulation language and was not originally intended as an input to synthesis
 - Many hardware description and simulation constructs are not supported by synthesis tools
 - Various synthesis tools use different subset of HDL -- *synthesis subset*
 - Each synthesis tool has its own synthesis subset
 - Up to now, most synthesis tools can read, translate & optimize *RTL* (Register-Transfer Level) HDL well
 - Behavioral synthesis is under development(Synopsys Behavior Compiler)



RTL Coding Style

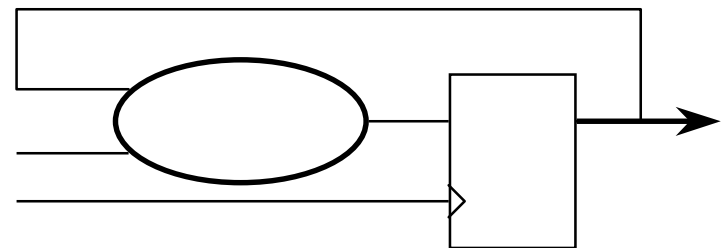
Combinational process

```
process(      )  
begin  
-----  
-----  
-----  
end process;
```



Clocked process

```
process(      )  
begin  
-----  
-----  
-----  
end process;
```



Synopsys Verilog/VHDL Synthesis Subset



- We are going to learn Synopsys Verilog/VHDL synthesis subset
 - Synopsys RTL coding style is widely accepted as a guideline to write a synthesizable Verilog or VHDL design file
 - Learning Synopsys Verilog or VHDL coding style can get the best synthesis results
 - It also can help avoid synthesis mismatch with simulation
- Optional
 - Synplicity: Synplify Pro
 - Mentor Graphic: LeoNardoSpectrum

HDL Coding Techniques

- Why we need to learn HDL coding techniques?
 - The designer has maximum influence on the final performance & density of the target device in the original coding style of the HDL
 - The effect of coding style on final circuit performance can be very dramatic
 - In some cases, circuit speed can be improved by a factor of three and utilization by a factor of two by making simple changes to source code
 - Ideally, you should code the design as abstractly as possible to preserve technology independence
 - However, the design can become faster and smaller as you design closer to the specific technology or device architecture

Naming Conventions for Identifiers



- User-defined names can not violate HDL naming rules
 - Using HDL keywords as identifiers for objects is not allowed
- FPGA tools may have other limitations on the character set
 - You should ordinarily use identifiers consisting of letters and digits, with underscores (“_”) included where necessary to separate parts of a name
 - Do not use “/”, “-”, “\$”, “<”, “>” as name separator
 - FPGA resource names are reserved and should not be used
- Case Considerations
 - FPGA tools may use formats in which case does not matter
 - Although Verilog considers upper and lower case letters to be different, you should not have two identifiers in the same Verilog module that differ only in case
 - For example, don't name one module “adder” and another one “Adder”

Omitting Delay Statements/ Initial Values



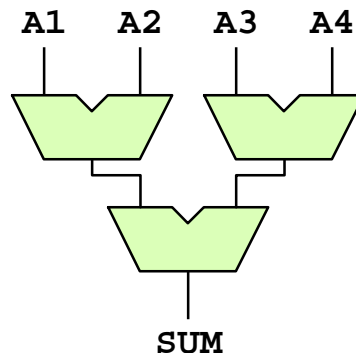
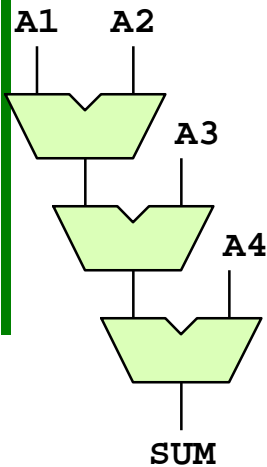
- Avoid using any delay statements to create code that simulates the same way before and after synthesis
 - Synthesis tools always ignore delay statements
 - Do not use delay statement in the design files
 - Do not use VHDL `wait for xx ns` or `after xx ns` statement
 - Do not use Verilog `#` delay token
- Do not assign signals and variables initial values to avoid synthesis mismatch with simulation
 - Most synthesis tools will ignore signal and variable initial values
 - For flip-flops, use clear or reset signal to initialize their values

Creating Readable Code

- Create code that is easy to read, debug and maintain
 - Indent blocks of code to align related statements
 - Use empty lines to separate top-level constructs, designs, architecture, process, ...
 - Use space to make your code easier to read
 - Break long lines of code at an appropriate point or a colon
 - Add comments to your code
- **Use labels** to group logic
 - You can use optional labels on flow control constructs to make the code structure more obvious
 - However, the labels are not exactly translated to gate or register names in your implemented design
 - It's suggested to label all VHDL processes, functions, and procedures and Verilog blocks

Ordering & Grouping Arithmetic Functions

- The ordering & grouping of arithmetic functions may influence design performance, depending on the bit width of input signals
 - Example1 (VHDL): $SUM \leq A1 + A2 + A3 + A4;$
Example2 (VHDL): $SUM \leq (A1 + A2) + (A3 + A4);$
 - Example1 cascades 3 adders in series
 - Example2 creates two adders in parallel: $(A1 + A2)$ and $(A3 + A4)$, and the results are combined with a third adder



VHDL IEEE Library

- Available packages for synthesis in the Synopsys IEEE library
 - `std_logic_1164`
 - `std_logic_arith`
 - `std_logic_unsigned`
 - `std_logic_signed`

VHDL Data Type

- Std_logic (IEEE 1164) type is recommended for synthesis
 - This type is effective for hardware descriptions because it has nine different values
 - Std_logic type is automatically initialized to an unknown value
 - It forces you to initialize your design to a known state
 - Do not override this feature by initializing signals and variables to a known value when they are declared because the result may be a gate-level circuit that cannot be initialized to a known value
 - Use Std_logic package for all entity port declarations to make it easier to integrate the synthesized netlist back into the design hierarchy without requiring conversion functions for the ports

VHDL Buffer Port Declaration



- Minimize the use of ports declared as buffers
 - Declare a buffer when a signal is used internally and as an output port
 - Every level of hierarchy in your design that connects to a buffer port must be declared as a buffer
 - To reduce the amount of coding in hierarchical designs, you may insert a dummy signal and declare the port as an output
 - It's often the best method because it does not artificially create a bidirectional port

```
P1: process begin
  wait until CLK'event and CLK = '1';
  C <= A + B + C;
end process P1;
```

```
C <= C_INT;
P1: process begin
  wait until CLK'event and CLK = '1';
  C_INT <= A + B + C_INT;
end process P1;
```

VHDL Signals & Variables

- Compare signals and variables
 - Signals are similar to hardware and are not updated until the end of a process
 - Variables are immediately updated
 - They can mask glitches that may impact how your design functions
 - It's better to use signals for hardware descriptions
 - However, variables allow quick simulation

VHDL Variable Assignments

- VHDL variable assignments execute in zero simulation time
 - The value of a variable assignment takes effect immediately after execution of the assignment statement

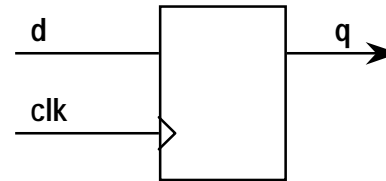
```
architecture BEHAV of SHIFTER is
begin

  VAR_ASSIGNMENT: process(CLK)

    variable VAR_D: STD_LOGIC;

  begin
    if (CLK'EVENT and CLK = '1') then
      VAR_D := D;
      Q <= VAR_D;
    end if;
  end process VAR_ASSIGNMENT;

end BEHAV;
```

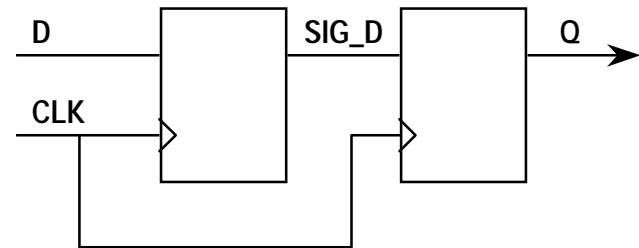


The variable VAR_D is optimized.

VHDL Signal Assignments

- VHDL signal assignments evaluate the right side expression and cause the simulator to schedule an update of the target
 - The actual assignment takes effect only after an actual or implied, (such as in a process with a sensitivity list), WAIT statement

```
architecture BEHAV of SHIFTER is  
  
    signal SIG_D: STD_LOGIC;  
  
begin  
  
    SIG_ASSIGNMENT: process(CLK)  
    begin  
        if (CLK'EVENT and CLK = '1') then  
            SIG_D <= D;  
            Q <= SIG_D;  
        end if;  
    end process SIG_ASSIGNMENT;  
  
end BEHAV;
```



Verilog Procedural Assignments

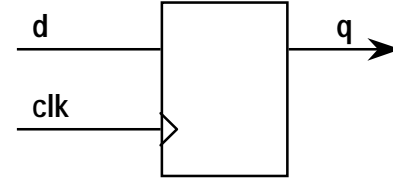


- Blocking procedural assignment (=)
 - The assignment is executed before the statement following the assignment is executed
 - This is a very essential form of assignment that should be used in testbenches
- Non-blocking procedural assignment (<=)
 - Non-blocking assignments allow the simulator to schedule assignments without blocking the procedural flow
 - The assignments are performed within the same time step without any order dependence
 - This is a very essential form of assignment that should be used in all design modules

Verilog Blocking Assignments

- Verilog immediately assign the values in blocking assignments

```
module nbk1(q, d, clk);  
  
  input d, clk;  
  output q;  
  
  reg q, bk_d;  
  
  always @(posedge clk) begin  
    bk_d = d;  
    q = bk_d;  
  end  
  
endmodule
```



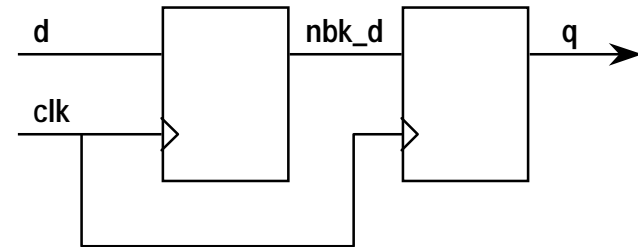
The reg bk_d is optimized.

Verilog Non-Blocking Assignments



- Verilog schedules the update of the target in non-blocking assignments

```
module nbk2(q, d, clk);  
  
    input d, clk;  
    output q;  
  
    reg q, nbk_d;  
  
    always @(posedge clk) begin  
        nbk_d <= d;  
        q <= nbk_d;  
    end  
  
endmodule
```



Component Instantiation

- Named association vs. positional association
 - Use positional association in function and procedures calls and in port lists only when you assign all items in the list
 - Use named association when you assign only some of the items in the list
 - Using named association is suggested

Positional Association

- The signals are connected up in the order in which the ports were declared

-- VHDL example

architecture STRUCT of INC is

 signal X,Y,S,C : bit;

 component FADD

 port(A,B : in bit;

 SUM, CARRY : out bit);

 end component;

begin

U1: FADD port map (X,Y,S,C);

-- other statements

end STRUCT;

// Verilog example

module fadd(a, b, sum, carry);

 :

 :

 :

endmodule

module inc(...)

 :

 :

 wire x, y, s, c;

 :

 :

 fadd u1(x,y,s,c);

 :

 // other statements

endmodule

Named Association

- The signals are connected up where the ports are explicitly referenced and order is not important

-- VHDL example

```
architecture STRUCT of INC is
  signal X,Y,S,C : bit;

  component FADD
    port(A,B : in bit;
         SUM, CARRY : out bit);
  end component;

begin

  ADDER1: FADD port map
    (B => Y, A => X, SUM => S, CARRY => C);

  -- other statements

end STRUCT;
```

// Verilog example

```
module fadd(a, b, sum, carry);
  :
  :
  :
endmodule

module inc(...)
  :
  :
  wire x, y, s, c;
  :
  :
  fadd adder1(.a(x), .b(y), .sum(s), .carry(c));
  :
  // other statements

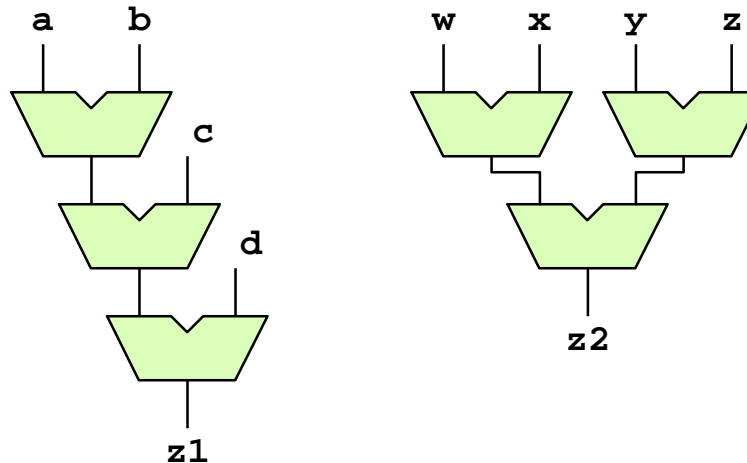
endmodule
```


Using Parentheses

- Use parentheses efficiently
 - You may use parentheses to control the structure of a complex design

```
// Verilog  
assign z1 = a + b + c + d;  
assign z2 = (w + x) + (y + z);
```

```
-- VHDL  
Z1 <= A + B + C + D;  
Z2 <= (W + X) + (Y + Z);
```



Specifying Bit-Width

- Use operator bit-width efficiently
 - You may directly specify the operand bits to reduce area

```
// Verilog
input [7:0] a, b;
output [7:0] z1, z2;

assign z1 = a + b;
assign z2 = a + b[4:0];
```

```
-- VHDL, must use std_logic_unsigned package
A, B: in STD_LOGIC_VECTOR(7 downto 0);
Z1, Z2: out STD_LOGIC_VECTOR(8 downto 0);
...
Z1 <= '0'&A + B;
Z2 <= '0'&A + B(4 downto 0);
```

Level-Sensitive Latches

- Level-sensitive latch
 - A variable assigned within an always block that is not fully specified in all branches will infer a latch
 - In some FPGA devices, latches are not available in the logic function block
 - Latches described in RTL HDL are implemented with gates, and will result in combination loops and hold-time requirements

Latch Inference - Verilog

```
// Ex1. simple D-latch
always @(datain or enable)
begin
    if (enable) dataout = datain;
end

// Ex2. simple D-latch with active-low RESET
always @(datain or enable or reset)
begin
    if (!reset) dataout = 0;
    else if (enable) dataout = datain;
end
```

Latch Inference - VHDL

```
-- Ex1. simple D-latch
DLAT: process(DATAIN, ENABLE)
begin
    if (ENABLE = '1') then DATAOUT <= DATAIN;
    end if;
end process DLAT;

-- Ex2. simple D-latch with active-low RESET
DLAT_R: process(DATAIN, ENABLE, RESET)
begin
    if (RESET = '0') then DATAOUT <= '0';
    elsif (ENABLE = '1') then DATAOUT <= DATAIN;
    end if;
end process DLAT_R;
```

Edge-Sensitive D Flip-Flops

- Edge-sensitive D flip-flops
 - In contrast to latches, flip-flops(registers) are inferred by the detection of the clock edge
 - Verilog: `@(posedge clk)` or `@(negedge clk)` in a always block
 - VHDL: `(clk'event and clk = '1')` or `(clk'event and clk = '0')`



D Flip-Flop Inference - Verilog

```
// Ex1. simple DFF
always @(posedge clk)
begin
    dataout = datain;
end

// Ex2. DFF with asynchronous active-low RESET
always @(posedge clk or negedge reset)
begin
    if (!reset) dataout = 0;
    else dataout = datain;
end

// Ex3. DFF with synchronous active-low RESET
always @(posedge clk)
begin
    if (!reset) dataout = 0;
    else dataout = datain;
end

// Ex4. DFF with active-high clock ENABLE
always @(posedge clk)
begin
    if (enable) dataout = datain;
end
```

D Flip-Flop Inference - VHDL

```
-- Ex1. simple DFF
DFF1: process(CLK)
begin
    if (CLK'EVENT and CLK = '1') then DATAOUT <= DATAIN;
    end if;
end process DFF1;

-- Ex2. DFF with asynchronous active-low RESET
DFF2_R: process(RESET, CLK)
begin
    if (RESET = '0') then DATAOUT <= '0';
    elsif (CLK'EVENT and CLK = '1') then DATAOUT <= DATAIN;
    end if;
end process DFF2_R;

-- Ex3. DFF with synchronous active-high ENABLE
DFF3_EN: process(CLK)
begin
    if (CLK'EVENT and CLK = '1') then
        if (ENABLE = '1') then DATAOUT <= DATAIN;
        end if;
    end if;
end process DFF3_EN;

-- Another style
-- if (ENABLE = '0') then null;
-- elsif (CLK'EVENT and CLK = '1') then
--     DATAOUT <= DATAIN;
-- end if;
```


Output Pad Tri-State Buffers

- Output pad tri-state buffers
 - Output tri-state buffers are available in I/O block of FPGA devices
 - To drive an output port
 - To drive internal logic connected to an inout port
 - Can implement tri-state registered output in some FPGA architectures
 - The flip-flop must directly drive the 3-state signal

Tri-State Buffer Inference - Verilog



```
// Ex1. Output pad tri-state buffers (for top-level output port)
assign out1 = (oe1) ? data1 : 1'bz;

// Ex2. Bidirectional pad
assign io1 = (oe1) ? data1 : 1'bz;
assign data2 = io1 & in1 & in2;

// Ex3. 3-state registered output
always @(posedge clk) bus_q = data3;
always @(oe2 or bus_q) begin
    if (!oe2) bus_out = bus_q;
    else bus_out = 1'bz;
end
```

Tri-State Buffer Inference - VHDL



```
-- Ex1. Output pad tri-state buffers (for top-level output port)
TRI1: process(DATA1, OE1)
begin
    if (OE1 = '1') then OUT1 <= DATA1;
        else OUT1 <= 'Z';
    end if;
end process TRI1;

-- Ex2. Bidirectional pad
TRI2: process(DATA1, OE1)
begin
    if (OE1 = '1') then IO1 <= DATA1;  -- IO1 is declared as an inout port
        else IO1 <= 'Z';
    end if;
end process TRI2;
DATA2 <= IO1 & IN1 & IN2;

-- Ex3. 3-state registered output
TRI_REG: process(CLK)
begin
    if (CLK'EVENT and CLK = '1') then BUS_Q <= DATA3;
    end if;
end process TRI_REG;
BUS_OUT <= BUS_Q when (OE2 = '0') else 'Z';
```

Internal Tri-State Buses

- Internal tri-state buses
 - Xilinx devices support many internal tri-state buffers
 - Easy to implement bus functions
 - Can be used to implement some wide functions
 - Xilinx internal tri-state buffers (TBUFs) are active-low enable
 - Altera tri-state emulation
 - Altera devices do not support internal tri-state buffers
 - Altera software will convert internal tri-state buffers into multiplexers

Tri-State Bus Inference

- Verilog

```
// Ex1. Internal tri-state buffers
assign busa = (!oe_a) ? a : 8'bz;
assign busa = (!oe_b) ? b : 8'bz;
```

- VHDL

```
-- Ex1. Internal tri-state buffers
BUS_IN1: process(A, OE_A)
begin
    if (OE_A = '0') then BUSA <= A;
    else BUSA <= "ZZZZZZZZ";
    end if;
end process BUS_IN1;

BUS_IN2: process(B, OE_B)
begin
    if (OE_B = '0') then BUSA <= B;
    else BUSA <= "ZZZZZZZZ";
    end if;
end process BUS_IN2;
```

Using If Statements

- Use `if` statements
 - `if` statement provides more complex conditional actions
 - An `if` statement generally builds a priority encoder that gives the shortest path delay to the first branch in the `if` statement
 - You must specify all the branches in a `if` statement, otherwise Synopsys may infer latches
 - Do not use deeply nested `if` statement
 - Improper use of the nested-if statement can result in an increase in area and longer delays in your designs

If Statement Examples - Verilog

```
// Ex1. Use if statements
always @(a or b or c or d or e or f or g or sel)
begin
    if (sel == 3'b000) mux_out = a;
    else if (sel == 3'b001) mux_out = b;
    else if (sel == 3'b010) mux_out = c;
    else if (sel == 3'b011) mux_out = d;
    else if (sel == 3'b100) mux_out = e;
    else if (sel == 3'b101) mux_out = f;
    else if (sel == 3'b110) mux_out = g;
    else mux_out = 1'b0;
end
```

If Statement Examples - VHDL

```
-- Ex1. use if statement
IF_1: process (A,B,C,D,E,F,G,SEL)
begin
    if (SEL = "000") then MUX_OUT <= A;
    elsif (SEL = "001") then MUX_OUT <= B;
    elsif (SEL = "010") then MUX_OUT <= C;
    elsif (SEL = "011") then MUX_OUT <= D;
    elsif (SEL = "100") then MUX_OUT <= E;
    elsif (SEL = "101") then MUX_OUT <= F;
    elsif (SEL = "110") then MUX_OUT <= G;
    else mux_out <= '0';
    end if;
end process IF_1;
```


Using Case Statements

- Use `case` statements
 - A `case` statement generally builds a multiplexer, where all the paths have similar delays
 - If a `case` statement is not a “full case”, it will infer latches
 - Use `default` statements or use “don't care” inference
 - If Synopsys can't determine that case branches are parallel, it may build a priority decoder

Case Statement Examples - Verilog



```
// Ex1. Use case and default statements
always @(a or b or c or d or e or f or g or sel)
begin
    case (sel)
        3'b000: mux_out = a;
        3'b001: mux_out = b;
        3'b010: mux_out = c;
        3'b011: mux_out = d;
        3'b100: mux_out = e;
        3'b101: mux_out = f;
        3'b110: mux_out = g;
        default: mux_out = 1'b0; // You may assign to 1'bx for specifying a don't care
    endcase
end

// Ex2. Use don't cares in case statements
always @(bcd) begin
    case (bcd)
        4'd0: bout = 3'b001;
        4'd1: bout = 3'b010;
        4'd2: bout = 3'b100;
        default: bout = 3'bxxx;
    endcase
end
```

Case Statement Examples - VHDL



```
-- Ex1. Use case & default statements
CASE_1P: process (A,B,C,D,E,F,G,SEL)
begin
    case SEL is
        when "000" => MUX_OUT <= A;
        when "001" => MUX_OUT <= B;
        when "010" => MUX_OUT <= C;
        when "011" => MUX_OUT <= D;
        when "100" => MUX_OUT <= E;
        when "101" => MUX_OUT <= F;
        when "110" => MUX_OUT <= G;
        when others => MUX_OUT <= '0'; -- You may assign a '-' for specifying a don't
care
    end case;
end process CASE_1P;

-- Ex2. Use don't cares in case statements
CASE_2P: process (BCD)
begin
    case BCD is
        when "0000" => BOUT <= "001";
        when "0001" => BOUT <= "010";
        when "0010" => BOUT <= "100";
        when others => BOUT <= "---";
    end case;
end process CASE_2P;
```

The full_case Directive

- You can attach the `full_case` directive to tell the synthesis tool the Verilog `case` item expressions cover all the possible cases
 - Synopsys does not do an exhaustive data flow analysis for the case statements, and so cannot tell that all the possible case items have been enumerated
 - Consequently, Synopsys implements extra logic to keep the value when it has some values other than those listed in the case statements
 - If you are sure the unspecified branches will never occur, you may use Synopsys `full_case` directive to specify a full case in Verilog and eliminate the extra logic
 - The `full_case` directive should appear immediately following the test expression of the `case` statement
 - It is recommended that you still use `full_case` if the case items do not include a `default` item
 - This may improve the efficiency of the synthesis process

The parallel_case Directive

- Attach the `parallel_case` directive to improve the efficiency if no two case item expressions ever match the test expression at the same time
 - You may use Synopsys `parallel_case` to specify a parallel case in Verilog, which force to generate multiplexer logic instead of priority decoder
 - The `full_case` directive should appear immediately following the test expression of the case statement

Using full_case & parallel_case



```
module johnson (clock, reset, count);

input clock, reset;
output [2:0] count;

wire clock, reset;
reg [2:0] count;

always @(negedge clock or posedge reset)
    if (reset == 1'b1)
        count <= 3'b000;
    else
        case (count) // synopsys full_case parallel_case
            3'b000: count <= 3'b001;
            3'b001: count <= 3'b011;
            3'b011: count <= 3'b111;
            3'b111: count <= 3'b110;
            3'b110: count <= 3'b100;
            3'b100: count <= 3'b000;
        endcase
endmodule
```

Using For Loop Statements

- Use `For` loop statements
 - Use `for` loop to duplicate statement
 - Loop index variable must be integer type
 - Step, start & end value must be constant

For Loop Statement Examples



- Verilog

```
-- Ex1. parity tree
always @(word)
begin
    is_odd = 0;
    for (i=0; i<=7; i=i+1) begin
        is_odd = is_odd xor word[i];
    end
end

assign parity = is_odd;
```

- VHDL

```
-- Ex1. parity tree
FOR_1: process (WORD)
    variable IS_ODD: STD_LOGIC;
begin
    IS_ODD := '0';
    LOOP1: for I in 0 to 7 loop
        IS_ODD := IS_ODD xor WORD(I);
    end loop LOOP1;
    PARITY <= IS_ODD;
end process FOR_1;
```


Designing Counters

- Counters can be built by inference or instantiation
 - Infer a simple counter
 - Synopsys will build it out of gates
 - Instantiate the optimized counter macrofunctions provided by FPGA vendors

Counter Examples - Verilog

```
// Ex1. Infer a loadable up-down counter with asynchronous reset
always @(negedge reset or posedge clk)
begin : upcnt_1
  if (!reset) count = 8'b00000000;
  else if (load) count = datain;
  else if (en)
    if (updn) count = count + 1;
    else count = count - 1;
end
assign dataout = count;
```

Counter Examples - VHDL

```
-- Ex1. Infer a loadable up-down counter with asynchronous reset
UPCNT_1: process(RESET, CLK)
begin
  if (RESET = '0') then COUNT <= "00000000";
  elsif (CLK'EVENT and CLK = '1') then
    if (LOAD = '1') then COUNT <= DATAIN;
    elsif (EN = '1') then
      if (UPDN = '1') then COUNT <= COUNT + 1;
      else COUNT <= COUNT - 1;
      end if;
    end if;
  end if;
  DATAOUT <= COUNT;
end process UPCNT_1;
```

Designing Multipliers

- Design multipliers
 - Infer a simple, non-DesignWare multiplier using the HDL operator “*”, which will be built out of gates and optimized
 - Instantiate the optimized multiplier macrofunctions provided by FPGA vendors

Multiplier Examples

- Verilog

```
// Ex1. Infer a simple non-pipelined multiplier  
assign multout = a * b;
```

- VHDL

```
-- Ex1. Infer a simple non-pipelined multiplier  
multout <= a * b;
```

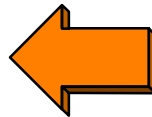
Designing FSMs

- Design FSMs (Finite State Machines)
 - A FSM is a design that has a clearly defined set of internal states and a defined method of moving between them
 - Most synthesis tools recognize any clocked process as an implicit state machine, that is, as a state machine for which the state variable is implicit
 - You can explicitly define a Verilog register or VHDL signal to be used as the state variable for state machine
 - FSMs designed for synthesis are usually represented as two processes/blocks, one calculates the next state (combinational logic), and one assigns the next state to the state vector register(s) on the clock edge (storage element)

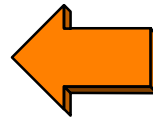
FSM Example - Verilog

```
parameter [4:0]
    sun = 5'b10000, mon = 5'b01000, tue = 5'b00100, wed = 5'b00010,
    thu = 5'b00001, fri = 5'b10001, sat = 5'b01110;
reg [4:0] day, next_day;

always @(posedge clk or negedge clear)
begin : SYNC
    if (!clear) day = sun;
    else day = next_day;
end
always @(day)
begin : COMB
    bell_ring = 0;
    case (day) // synopsys full_case
        sun: next_day = mon;
        mon: next_day = tue;
        tue: next_day = wed;
        wed: next_day = thu;
        thu: next_day = fri;
        fri: next_day = sat;
        sat: begin
            next_day = sun; bell_ring = 1;
        end
    endcase
end
```



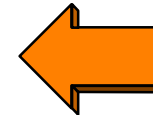
update state vector



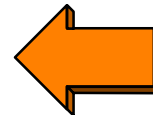
combinational logic

FSM Example - VHDL

```
architecture BEHAVIOR of FSM2 is
  type DAY_TYPE is (SUN, MON, TUE, WED, THU, FRI, SAT);
  signal DAY, NEXT_DAY: DAY_TYPE;
begin
  SYNC: process(CLK, CLEAR)
  begin
    if (CLEAR = '0') then DAY <= SUN;
    elsif (CLK'EVENT and CLK = '1') then DAY <= NEXT_DAY;
    end if;
  end process SYNC;
  COMB: process (DAY)
  begin
    BELL_RING <= '0';
    case DAY is
      when SUN => NEXT_DAY <= MON;
      when MON => NEXT_DAY <= TUE;
      when TUE => NEXT_DAY <= WED;
      when WED => NEXT_DAY <= THU;
      when THU => NEXT_DAY <= FRI;
      when FRI => NEXT_DAY <= SAT;
      when SAT => NEXT_DAY <= SUN; BELL_RING <= '1';
    end case;
  end process COMB;
```



update state vector



combinational logic

Designing Memories

- Do not behaviorally describe RAMS in your code because the synthesis tool will create combinational loops and a lot of registers
 - None of synthesis tools can judge a RAM block or process
 - Instead, you should instantiate a RAM module or primitive provided by the vendor
- Xilinx provide LogiBLOX to create the following items for simulation and synthesis:
 - Behavioral Verilog/VHDL simulation model files
 - They can not be used as the input to the synthesis tool
 - Example files to tell you how to instantiate the RAM/ROM module in your design file
 - Timing model files for the synthesis tool
 - The netlist files required by the FPGA implementation tools

Synchronous & Asynchronous RAM



- Xilinx provides synchronous and asynchronous RAM blocks
 - Use synchronous RAM instead of asynchronous RAM if possible
 - FPGA RAM block usually has a self-timed write pulse
 - Address can be change after it is clocked into RAM
 - Besides, you may register RAM outputs for pipelining

Considerations for Synthesis



- The main consideration for synthesis: overall cost
 - Time is money!
 - Area is money!
 - Quality is money!

Considering the Design

- The main considerations in the design stage are:
 - Level of abstraction in HDL modeling
 - Levels of hierarchy
 - Technology-dependent or independent
 - Reliability
 - Easy to maintain and debug your code
 - Development time
 - Simulation time
 - Easy to upgrade
 - Migration to ASIC

Considering Synthesis Process



- The main considerations in the synthesis process are:
 - Optimization constraints
 - Hierarchy structure of your design
 - CPU time
 - Memory requirements

Considering Synthesis Results



- The main considerations for the synthesis results are:
 - Performance
 - Critical timing paths
 - Area
 - FPGA resource utilization

Instantiation vs. Inference

- Instantiation vs. inference
 - Instantiation: explicitly select a technology-specific element in the target library
 - Inference: direct the HDL Compiler to infer latches or flip-flops or other resource blocks from your Verilog or VHDL description
- Technology-independent vs. technology-dependent
 - Keeping the pre-synthesis design source code technology-independent allows you to re-target to other technologies at a later time, with a minimum of re-design effort
 - Technology-dependent design uses dedicated functions or design techniques optimized for speed or area
 - The designer may require detailed understanding on device architectures

Synchronous vs. Asynchronous



- Synchronous designs run smoothly through synthesis, simulation, test and implementation
 - Synchronous designs uses more registers, which are rich in FPGAs
 - **Watch for unintentional latch inference**
 - Remember completely specify all branches for every case or if statement
 - If asynchronous logic blocks are required, put them into separate blocks

Design Partitioning

- Design partitioning
 - Manage your design hierarchy to reduce CPU time during synthesis, and also achieve higher performance
 - A very big design without hierarchy may require 24 hours or above of run time
 - A design with many levels of hierarchy may reduce efficiency during synthesis process

Considering Design Partitioning



- Strategies for design partitioning
 - Partition each design block to target < 5,000 gates size
 - Do not partition a basic logic element, for example, a flip-flop or one-bit adder, into a basic block in a large design
 - The number of the hierarchy levels will be too large and the compile time will increase very much
 - Separate datapath with control logic
 - Separate structured circuit with random logic
 - Structured circuit: adder, multiplier, comparator, MUXs, ALU, ...
 - Unstructured circuit: random control logic, PLA, ...
 - Merge blocks that can share resources
 - Isolate state machines
 - Keep related combinational logic together in a single block
 - Keep critical paths in a single block and remove circuit that is not speed critical from the critical path block

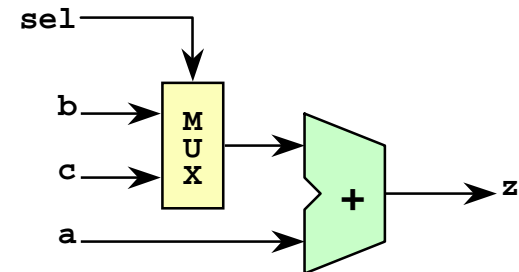
Registered Outputs

- Registered outputs
 - Register the outputs of lower-level files in the design hierarchy whenever possible
 - This process generates code that is easier to maintain and it can be more successfully synthesized
 - Registered outputs also make it easier to provide accurate timing constraints
 - Use a single clock for each design clock
 - Separate combinational and sequential assignments

Resource Sharing

- Resource sharing
 - Operations can be shared if they lie in the same block
 - May improve resource utilization & design performance
- Strategies
 - Keep resources that might be shared between different functions or that might require a similar configuration in the same block of the design hierarchy
 - Group similar designs into the same hierarchy to allow Synopsys to share resources when practical

```
always @(a or b or c or sel)
  if (sel) z = a + b;
  else z = a + c;
```



Pipelining & Pipeline Re-Timing



- Pipelining & pipeline re-timing
 - To pipeline high-speed sequential designs whenever possible
 - FPGA devices are register-rich
 - FPGA devices provide a high ratio of flip-flops to combinational logic
 - The architecture makes pipelining & pipeline re-timing especially advantageous techniques for FPGAs
 - There is often little or no area penalty associated with using extra registers
 - You can make a pipelined design to achieve higher performance & higher device utilization simultaneously in the FPGA architecture

Designing FSMs

- FSM (Finite State Machine) encoding is the key issue
 - The number of available registers affects the design of FSMs
 - Binary encoding of FSMs is designed to minimize the usage of flip-flops
 - Gray encoding is designed to avoid glitches
 - “One-hot” encoding is typically the fastest state machine implementation available in FPGA architecture
 - All state flip-flops must be driven by the same clock
 - In cases where “fail-safe” behavior is required, specify default clause in every `case` statement
 - Synopsys will generate the required logic to reset the state machine to a known state when it goes to an unknown state

State Machine Partitioning

- State machine partitioning
 - A complex state machine can be partitioned into smaller state machines
 - For example, you may be able to reduce a 100-state state machine into 3 state machines of 6 states each
 - With good designs, there can be a positive effect on both performance & area
 - Each sub-machine needs an *IDLE* state during which the control is passed to another sub-machine
 - Partition your design into control logic & datapath elements and use a state machine for the control blocks

Appendix A

Instantiation Examples

Clock Buffer (VHDL)

```
entity CLOCKBUF is
  port(CS,STROBE,DATAIN: in std_logic; DATAOUT: out std_logic);
end CLOCKBUF;
```

```
architecture XILINX of CLOCKBUF is
  component BUFG
    port(I : in std_logic; O : out std_logic );
  end component;
  signal CLOCKIN, CLOCKOUT : std_logic;
begin
  CLOCKIN <= CS and STROBE;
  MYBUFG : BUFG port map( I=> CLOCKIN, O => CLOCKOUT );
  P1 : process( CLOCKOUT )begin
    if( CLOCKOUT'event and CLOCKOUT='1' ) then
      DATAOUT <= DATAIN;
    end if;
  end process P1;
end XILINX;
```

Clock Buffer (Verilog)

```
module CLOCKBUF ( CS, STROBE, DATAIN, DATAOUT );
input  CS, STROBE, DATAIN;
output DATAOUT;
reg    DATAOUT;
wire   CLOCKIN;
wire   CLOCKOUT;

assign CLOCKIN = CS & STROBE;
BUFG MYBUFG ( .I(CLOCKIN), .O(CLOCKOUT) );

always @ (posedge clockout)
begin
    dataout = datain;
end
endmodule
```

Registered Bi-directional I/O (VHDL)

```

entity bidi_reg is
    port (SIGA: inout BIT;
          LOADA, CLOCK1, CLOCK2, RST: in BIT);
end bidi_reg;
architecture BEHAV of bidi_reg is
    component reg1
        port (INX, LOAD, CLOCK, RESET: in BIT;
              OUTX: out BIT);
    end component;
    component OFDT_F
        port (D, C, T: in BIT;
              O : out BIT);
    end component;
    component IBUF
        port (I: in BIT;
              O: out BIT);
    end component;

    signal INA, OUTA: BIT;
begin
    U0: reg1 port map (INA, LOADA, CLOCK1, RST, OUTA);
    U1: OFDT_F port map (OUTA, CLOCK2, LOADA, SIGA);
    U2: IBUF port map (SIGA, INA);
end BEHAV;

```



Registered Bidirectional I/O (Verilog)

```
module bidi_reg (SIGA, LOADA, CLOCK1, CLOCK2, RST) ;

  inout SIGA;
  input LOADA, CLOCK1, CLOCK2, RST;

  wire INA, OUTA;

  REG1      U0 (INA, LOADA, CLOCK1, RST, OUTA);
  OFDT_F    U1 (.D(OUTA), .C(CLOCK2), .T(LOADA), .O(SIGA)) ;
  IBUF      U2 (.I(SIGA), .O(INA)) ;

endmodule
```

Boundary Scan (VHDL)

entity EXAMPLE is

port(A,B,C,D : in std_logic; O : out std_logic);

end EXAMPLE;

architecture XILINX of EXAMPLE is

component TCK port(I : out std_logic); end component;

component TDI port(I : out std_logic); end component;

component TMS port(I : out std_logic); end component;

component TDO port(O : in std_logic); end component;

component BSCAN port(TDI,TMS,TCK : in std_logic;
TDO : out std_logic); end component;

signal TCK_NET, TDI_NET, TMS_NET, TDO_NET : std_logic;

begin

U1:BSCAN port map(TDI=>TDI_NET,TMS=>TMS_NET,TCK=>TCK_NET,TDO=>TDO_NET);

U2:TCK port map (I=>TCK_NET);

U3:TDI port map (I=>TDI_NET);

U4:TMS port map (I=>TMS_NET);

U5:TDO port map (O=>TDO_NET);

O <= A and b and C and D; -- A token bit of logic

end XILINX;

Boundary Scan (Verilog)

```
module EXAMPLE ( A, B, C, D, O );  
input  A, B, C, D;  
output O;  
wire TDI_NET, TDI_NET, TMS_NET, TDO_NET;  
  
BSCAN U1 (.TDI(TDI_NET),.TMS(TMS_NET),.TCK(TCK_NET),.TDO(TDO_NET));  
TDI  U2 ( TDI_NET );  
TDO  U3 ( TDO_NET );  
TMS  U4 ( TMS_NET );  
TCK  U5 ( TCK_NET );  
  
assign O = A & B & C & D;  
  
endmodule
```

Wide Edge Decoders (Verilog)



```
module EXAMPLE ( ADDRESS, STROBE, SELECT );

input  [15:0] ADDRESS;
input      STROBE;
output     SELECT;

parameter MASK [15:0] 16'b1110000000000101;

PULLUP U1 ( .O(SELECT) );

DECODE1_IO D0 ( .I(MASK[0] ^ ADDRESS[0]), .O(SELECT) );
DECODE1_IO D1 ( .I(MASK[1] ^ ADDRESS[0]), .O(SELECT) );
DECODE1_IO D2 ( .I(MASK[2] ^ ADDRESS[0]), .O(SELECT) );

DECODE1_IO D13 ( .I(MASK[13] ^ ADDRESS[0]), .O(SELECT) );
DECODE1_IO D14 ( .I(MASK[14] ^ ADDRESS[0]), .O(SELECT) );
DECODE1_IO D15 ( .I(MASK[15] ^ ADDRESS[0]), .O(SELECT) );
DECODE1_IO D16 ( .I(STROBE), .O(SELECT) );

endmodule
```

Behavioral 16x4 ROM (Verilog)



```
module rom16x4_4k(ADDR, DATA) ;  
input [3:0];  
output [3:0];  
  
reg [3:0] DATA;  
  
always @(ADDR)  
begin  
    case (ADDR)  
        4'b0000 : DATA = 4'b0000 ;  
        4'b0001 : DATA = 4'b0000 ;  
        // Continue rest of case statement  
        4'b1110 : DATA = 4'b0000 ;  
        4'b1111 : DATA = 4'b0000 ;  
    endcase;  
end  
  
endmodule
```


LogiBLOX 16x4 ROM (Verilog)



```
module rom_LogiBLOX (ADDR,  
                    DATA);  
  
    input [3:0] ADDR ;  
    output [3:0] DATA ;  
  
    promdata u1 (.A3(ADDR[3]),.A2(ADDR[2]),.A1(ADDR[1]),.A0(ADDR[0]),  
                .O3(DATA[3]),.O2(DATA[2]),.O1(DATA[1]),.O0(DATA[0]));  
endmodule  
  
module promdata(A3, A2, A1, A0, O3, O2, O1, O0);  
  
    input A3, A2, A1, A0;  
    output O3, O2, O1, O0;  
  
endmodule
```

RAM (Verilog)

```
module EXAMPLE ( ADDR, DATAIN, WRITE, DATAOUT );
input  [3:0] ADDR, DATAIN;
input      WRITE;
output [3:0] DATAOUT;

RAM16X1 RAMBANK_0 ( .D(DATAIN[0], .A0(ADDR[0]), .A1(ADDR[1]),
                    .A2(ADDR[2]), ..A3(ADDR[3]), O(DATAOUT[0]), .WE(WRITE) );
RAM16X1 RAMBANK_1 ( .D(DATAIN[1], .A0(ADDR[0]), .A1(ADDR[1]),
                    .A2(ADDR[2]), .A3(ADDR[3], .O(DATAOUT[1]), .WE(WRITE) );
RAM16X1 RAMBANK_2 ( .D(DATAIN[2], .A0(ADDR[0]), .A1(ADDR[1]),
                    .A2(ADDR[2]), .A3(ADDR[3]),.O(DATAOUT[2]), .WE(WRITE) );
RAM16X1 RAMBANK_3 ( .D(DATAIN[3], .A0(ADDR[0]), .A1(ADDR[1]),
                    .A2(ADDR[2]), .A3(ADDR[3]), .O(DATAOUT[3]), .WE(WRITE) );

endmodule;
```

STARTUP Block (Verilog)

```
module EXAMPLE ( DATAIN, CLOCK, CLEAR, DATAOUT );
```

```
input [63:0] DATAIN;
```

```
input      CLOCK, CLEAR;
```

```
output [63:0] DATAOUT;
```

```
reg [63:0] DATAOUT;
```

```
STARTUP GSRBLK ( .GSR(CLEAR) );
```

← Connection to GSR pin
on STARTUP block

```
always @ ( posedge CLOCK or posedge CLEAR )
```

```
begin
```

```
  if ( CLEAR == 1'b1 )
```

```
    DATAOUT = 16'hFFFFFFFFFFFFFFFF;
```

```
  else
```

```
    DATAOUT = DATAIN;
```

```
end
```

```
endmodule
```

Appendix B

State Machines Examples

Binary Encoding (Verilog)

```
reg [3:0] current_state, next_state;

parameter state1 = 2'b00, state2 = 2'b01,
           state3 = 2'b10, state4 = 2'b11;

always @ (current_state)
  case (current_state)
    state1 : next_state = state2;
    state2 : next_state = state3;
    state3 : next_state = state4;
    state4 : next_state = state1;
  endcase

always @ (posedge clock)
  current_state = next_state;
```

One-Hot Encoding (Verilog)

```
module onehot(signala,clock,reset)
input                signala,clock,reset;
reg [4:0]            current_state,next_state;
parameter            s0 = 0,s1 = 1,s2 = 2,s3 = 3,s4 = 4;
// combinatorial process
always @(current_state or signala)
begin
    next_state = 5'b0;
    case (1'b1) // synopsys parallel_case full_case
        current_state[s0]:
            if (signala)
                next_state[s0] = 1'b1;
            else
                next_state[s1] = 1'b1;
        current_state[s1]:
            next_state[s2] = 1'b1;
        ....
    endcase
end
// process for synchronous elements
always @(posedge clock or posedge reset)
begin
    if (reset)
        current_state <= 'b00001;
    else
        current_state <= next_state;
    end
endmodule
```

Appendix C

TBUF Examples

Efficient Muxes (Verilog)

```
module EXAMPLE ( DATAIN, SELECTOR, SELECTION );
input [15:0] DATAIN; input [3:0] SELECTOR; output SELECTION;
```

```
tri      SELECTION;
reg  [15:0] TBUFNET;
integer  I;
```

```
always @ ( SELECTOR or DATAIN ) begin
  for( I=0; I<=15; I=I+1 )
    TBUFNET[I] = 1'bz;
```

← Implies a TBUF

```
  case ( SELECTOR )
    0 : TBUFNET[0] = DATAIN[0];
    . . . . .
    15 : TBUFNET[15] = DATAIN[15];
  endcase
end
```

```
assign SELECTION = TBUFNET[0];
. . . . .
assign SELECTION = TBUFNET[15];
```

```
endmodule
```

Coding-85

Wide Wire-AND (Verilog)

```
module EXAMPLE ( DATAIN, ALLHIGH );  
input [15:0] DATAIN; output ALLHIGH;
```

```
reg [15:0] TBUFNET;  
integer I;
```

```
always @ ( DATAIN ) begin
```

```
  for( I=0; I<=15; I=I+1 )  
    if (DATAIN[I] == 0)  
      TBUFNET[I] = 1'b0;  
    else  
      TBUFNET[I] = 1'bz;
```

← Implies a TBUF

```
end
```

```
assign ALLHIGH = TBUFNET[0];
```

```
  . . . . .  
assign ALLHIGH = TBUFNET[15];
```

```
PULLUP U1 ( .O(ALLHIGH) );  
endmodule
```

State Editor & Black Box Instantiation

HDL Designs with State Machines

- How to create a state machine and add it into a HDL Flow project?
- Creating a State Machine Macro
 - 1. Open the State Editor by clicking the FSM icon in the Design Entry box on the Project Manager's Flow tab.
 - 2. Select Use the HDL Design Wizard. Click OK.
 - 3. From the Design Wizard window, select Next.
 - 4. From the Design Wizard - Language window, choose VHDL or Verilog and select Next.
 - 5. In the Design Wizard - Name window, enter a name for your macro. Select Next.
 - 6. Define your ports in the Design Wizard-Ports window. Select Next.
 - 7. In the Design Wizards - Machines window, select the number of state machines that you want. Click Finish. The Wizard creates the ports and gives you a template in which you can enter your macro design.
 - 8. Complete the design for your macro in the State Editor.
 - 9. Add the macro to the project by selecting **Project > Add** to Project from the Project Manager.

HDL Designs with Instantiated Xilinx Unified Library Components



- It is possible to instantiate certain Xilinx Unified Library components directly into your VHDL or Verilog code.
 - In general, you will find this most useful for components that the Express compiler is unable to *infer*, such as BSCAN, RAM, and certain types of special Xilinx components.
- When instantiating Unified Library components, the component must first be declared before the begin keyword in VHDL the architecture and then may be instantiated multiple times in the body of the architecture.

Instantiated Components I

- **STARTUP Component**
 - The STARTUP component is typically used to access the global set/reset and global 3-state signals.
 - STARTUP_VIRTEX for Virtex
- **BSCAN Component**
 - To use the boundary-scan (BSCAN) circuitry in a Xilinx FPGA, the BSCAN component must be present in the input design.
 - The TDI, TDO, TMS, and TCK components are typically used to access the reserved boundary scan device pads for use with the BSCAN component but can be connected to user logic as well.
- **READBACK Component**
 - To use the dedicated readback logic in a Xilinx FPGA, the READBACK component must be inserted in the input design.
 - The MD0, MD1, and MD2 components are typically used to access the mode pins for use with the readback logic but can be connected to user logic as well.

Instantiated Components II

- RAM and ROM
 - Some of the most frequently instantiated library components are the RAM and ROM primitives.
 - Because most synthesis tools are unable to infer RAM or ROM components from the source HDL, the primitives must be used to build up more complex structures.
 - The following list of RAM and ROM components is a complete list of the primitives available in the Xilinx library.
 - RAM16X1, RAM16X1D, RAM16X1S, RAM32X1, RAM32X1S, ROM16X1, ROM32X1
- Global Buffers
 - BUFG, BUFGP, BUFGS, BUFGLS, BUFGE, BUFFCLK, BUFGSR, BUFGTS
 - For most designs, the BUFG, BUFGS, and BUFGP components can be inferred or instantiated, thus allowing the design implementation tools to make an appropriate physical buffer allocation.

An example of an instantiated Xilinx Unified Library component



- The following sample written in VHDL shows an example of an instantiated Xilinx Unified Library component, STARTUP.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity gsr_test is
  port (
    CLK: in STD_LOGIC;
    D_IN: in STD_LOGIC;
    RESET: in STD_LOGIC;
    Q_OUT: out STD_LOGIC
  );
end gsr_test;
```

```
architecture gsr_test_arch of gsr_test is
  component STARTUP
  port (GSR: in std_logic);
  end component;

begin
```

```
U1: STARTUP port map (GSR=>RESET);
```

```
process (CLK)
begin
  if (CLK event and CLK='1') then
    Q_OUT <= D_IN;
  end if;
end process;

end gsr_test_arch;
```

HDL Designs with Black Box Instantiation

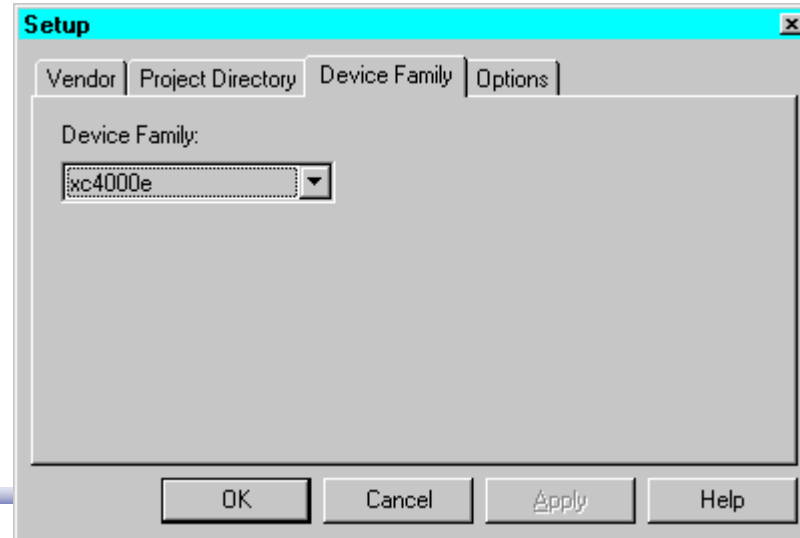


- LogiBLOXs, schematics (EDIF file), ABEL modules, and XNF files can be instantiated in the VHDL and Verilog code using the "black box instantiation" method.
 - The Files tab in the Hierarchy Browser does not display the black box module name under the HDL file(s) in which it is instantiated.

LogiBLOX RAM Modular in a VHDL Design I



- Using LogiBLOX, create a RAM module.
 - 1. To start LogiBLOX,
 - From the Project Manager, select Tools > Design Entry > LogiBLOX module generator
 - From the HDL Editor, select Synthesis > LogiBLOX
 - From Schematic Capture, select Options > LogiBLOX
 - 2. Click Setup on the LogiBLOX Module Selector screen. (The first time LogiBLOX is invoked, the Setup screen appears automatically.)
 - 3. In the Setup window, enter the following items.
 - Under the Device Family tab, use the pulldown list to select the target device family (XC4000E, for example).

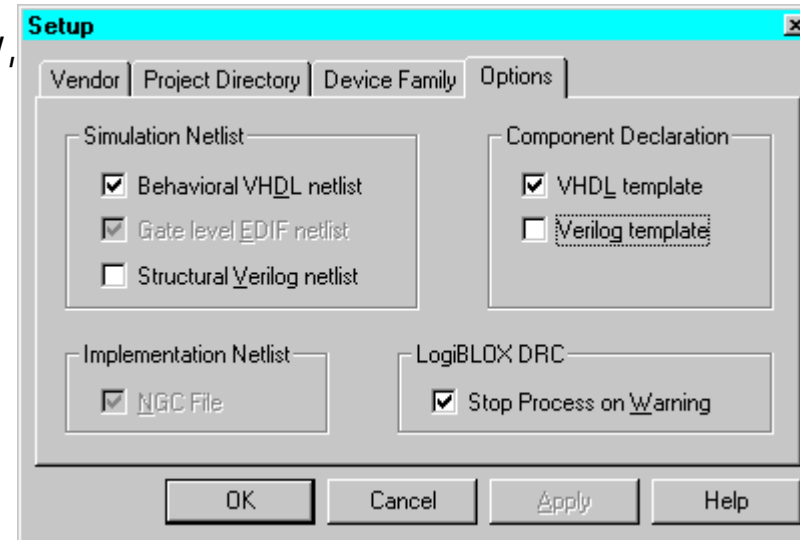


LogiBLOX RAM Modular in a VHDL Design II



- Under the Options tab, select the Simulation Netlist and Component Declaration template.
- To instantiate the LogiBLOX module in VHDL code, select VHDL template in the Component Declaration area and Behavioral VHDL netlist in the Simulation Netlist area, as shown below. Click OK

- 4. In the LogiBLOX Module Selector window, define the type of LogiBLOX module and its attributes. The Module Name specified here is used as the name of the instantiation in the VHDL code.
- 5. When you click OK, the LogiBLOX module is created automatically and added to the project library.



LogiBLOX RAM Modular in a VHDL Design III



- The LogiBLOX module is a collection of several files including those listed below.
 - The files are located in your Xilinx project directory for the current project.
 - component_name.ngc Netlist used during the Translate phase of Implementation
 - component_name.vhi Instantiation template used to add a LogiBLOX module into your VHDL source code
 - component_name.vhd VHDL file used for functional simulation
 - component_name.mod Configuration information for the module
 - logiblox.ini LogiBLOX configuration for the project
 - The component name is the name given to the LogiBLOX module in the GUI. The port names are the names provided in the .vhi file.
- Using the *.vhi* file as a reference, write your VHDL code to instantiate the LogiBLOX RAM module

.VHI File Created by LogiBLOX

```

-----
-- LogiBLOX SYNC_RAM Module "memory"
-- Created by LogiBLOX version M1.4.12
--   on Fri May 22 17:30:23 1998
-- Attributes
--   MODTYPE = SYNC_RAM
--   BUS_WIDTH = 4
--   DEPTH = 48
-----

-- Component Declaration
-----

component memory
  PORT(
    A: IN std_logic_vector(5 DOWNT0 0);
    DI: IN std_logic_vector(3 DOWNT0 0);
    WR_EN: IN std_logic;
    WR_CLK: IN std_logic;
    DO: OUT std_logic_vector(3 DOWNT0 0));
end component;

-----

-- Component Instantiation
-----

instance_name : memory port map
(A => ,
 DI => ,
 WR_EN => ,
 WR_CLK => ,
 DO => );

```

VHDL File With LogiBLOX Instantiation

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity top is
port (
    D: in STD_LOGIC;  CE: in STD_LOGIC;
    CLK: in STD_LOGIC;  Q: out STD_LOGIC;
    Atop: in STD_LOGIC_VECTOR (5 downto 0);
    D0top: out STD_LOGIC_VECTOR (3 downto 0);
    D1top: in STD_LOGIC_VECTOR (3 downto 0);
    WR_ENTop: in STD_LOGIC;
    WR_CLKtop: in STD_LOGIC);
end top;

architecture inside of top is

component userff
port (
    D: in STD_LOGIC;  CE: in STD_LOGIC;
    CLK: in STD_LOGIC;  Q: out STD_LOGIC);
end component;

component memory
PORT(
    A: IN std_logic_vector(5 DOWNTO 0);
    DI: IN std_logic_vector(3 DOWNTO 0);
    WR_EN: IN std_logic;
    WR_CLK: IN std_logic;
    DO: OUT std_logic_vector(3 DOWNTO 0));
end component;

begin

U0: userff port map (D=>D, CE=>CE, CLK=>CLK, Q=>Q);
U1: memory port map (A=>Atop, DI=>D1top, WR_EN=>WR_ENTop,
    WR_CLK=>WR_CLKtop, DO=>D0top);

end inside;

```

LogiBLOX RAM Modular in a VHDL Design IV



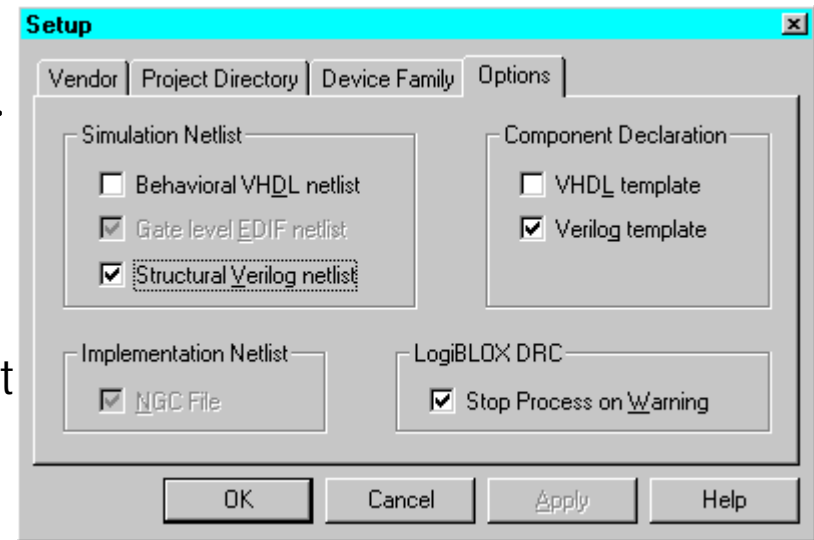
- When the design is synthesized, a warning is generated that the LogiBLOX module is unexpanded.
- Modules instantiated as black boxes are not elaborated and optimized.
- The warning message is just reflecting the black box instantiation.

LogiBLOX RAM Modular in a Verilog Design I



Using LogiBLOX, create a RAM module.

- The same with VHDL
- Under the Options tab, select Verilog template in the Component Declaration area and select Structural Verilog netlist in the Simulation Netlist area.



The LogiBLOX module is a collection of several files including those listed below.

- The files are located in your Xilinx Project directory for the current project.
- component_name.ngc Netlist used during the Translate phase of Implementation
- component_name.vei Instantiation template used to be added into your Verilog code
- component_name.v Verilog file used for functional simulation
- component_name.mod Configuration information for the module

LogiBLOX RAM Modular in a Verilog Design II



- logiblox.ini LogiBLOX configuration for the project
- The component name is the name given to the LogiBLOX module in the GUI. The port names are the names provided in the .vei file.
- Using the *.vei* file as a reference, write your Verilog code to instantiate the LogiBLOX RAM module

**.VEI File Created
by LogiBLOX**

```
//-----
// LogiBLOX SYNC_RAM Module "memeory"
// Created by LogiBLOX version M1.4.12
//   on Mon May 25 14:58:11 1998
// Attributes
//   MODTYPE = SYNC_RAM
//   BUS_WIDTH = 4
//   DEPTH = 48
//-----
memeory instance_name
( .A(),
  .DO(),
  .DI(),
  .WR_EN(),
  .WR_CLK());

module memeory(A, DO, DI, WR_EN, WR_CLK);
input [5:0] A;
output [3:0] DO;
input [3:0] DI;
input WR_EN;
input WR_CLK;
endmodule
```

```
module top (D,CE,CLK,Q,  
            Atop,Dotop,Ditop,WR_Entop,WR_CLKtop);  
    input D;  
    input CE;  
    input CLK;  
    output Q;  
  
    input [5:0] Atop;  
    output [3:0] D0top;  
    input [3:0] DItop;  
    input WR_Entop;  
    input WR_CLKtop;  
  
    userff U0 (.D(D),.CE(CE),.CLK(CLK),.Q(Q));  
    memory U1 (.A(Atop),.DO(D0top),.DI(DItop),  
              .WR_EN(WR_Entop),  
              .WR_CLK(WR_CLKtop));  
endmodule  
  
module mememory(A, DO, DI, WR_EN, WR_CLK);  
    input [5:0] A;  
    output [3:0] DO;  
    input [3:0] DI;  
    input WR_EN;  
    input WR_CLK;  
endmodule
```

Verilog File With LogiBLOX Instiation

LogiBLOX RAM Modular in a Verilog Design III



- An alternate method is to place the module declaration from the .vei file into a new, empty Verilog file (MRMEMORY.V) and add the new file (shown below) to the project.

```
//-----  
// LogiBLOX SYNC_RAM Module "memory"  
// Created by LogiBLOX version M1.5.18  
//   on Wed Jun 24 10:40:25 1998  
// Attributes  
//   MODTYPE = SYNC_RAM  
//   BUS_WIDTH = 4  
//   DEPTH = 48  
//   STYLE = MAX_SPEED  
//   USE_RPM = FALSE  
//-----  
module MRMEMORY (A, DO, DI, WR_EN, WR_CLK);  
input [5:0] A;  
output [3:0] DO;  
input [3:0] DI;  
input WR_EN;  
input WR_CLK;  
endmodule
```

Schematic (EDIF file) in an VHDL or Verilog Design I



- Schematics in Foundation can be instantiated as a black box in a VHDL or Verilog file.
- Procedure
 - 1. Open Schematic Capture
 - By clicking the Schematic Capture icon in the Design Entry box on the Project Manager's Flow tab.



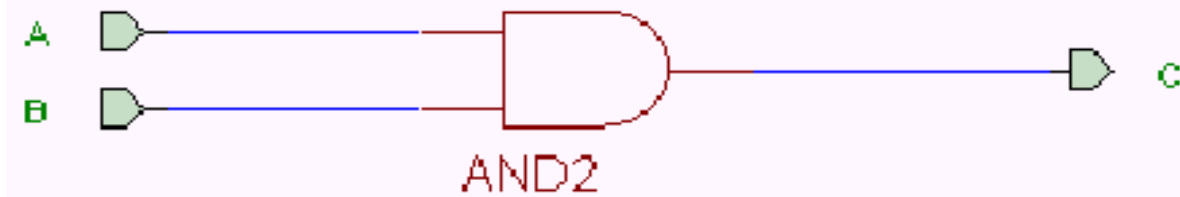
- 2. Create the schematic in Schematic Capture.
 - **The schematic must not have I/O library components like IBUF, OBUF, OBUFT.**
 - **Use hierarchy connectors in place of I/O components.**
 - **The name of the hierarchy connectors will be the name of the pins for the instantiation in the HDL code.**
 - The schematic can contain hierarchy.

Schematic (EDIF file) in an VHDL or Verilog Design II



- If you don't assign any unified library (such as xc4000e, xc4000x...) in the project, you will see nothing in Symbol Toolbox. You must use Project Libraries to add the unified library first.

test1 Design with I/O Terminals



- 3. Save the schematic
 - By selecting File > Save As and choose a name for the module.
 - **Save the file to your Foundation project directory, but do not add the schematic to the project.**
- 4. Create a netlist
 - Select Options > Create Netlist from Current Sheet.
- 5. Create an EDIF file
 - Select Options > Export Netlist.
 - Select the newly created netlist and make sure that the file is exported to the **.edn** format.

Schematic (EDIF file) in an VHDL or Verilog Design III



- 6. Instantiate the Foundation schematic.
 - For **Verilog**, instantiate the Foundation schematic, using the schematic module name as the instantiated module name and the name of the hierarchy connectors as the name of the pins. In the following example, the EDIF file was created in a project called test1.

Verilog Instantiation Example

```
module test (FIRST,SECOND,THIRD);  
  
  input FIRST;  
  input SECOND;  
  output THIRD;  
  
  test1 U1 (.A(FIRST),.B(SECOND),.C(THIRD));  
  
endmodule
```

- For **VHDL**, instantiate the Foundation schematic, using the schematic module name as the instantiated module name and the name of the hierarchy connectors as the names of the pins. In this example, the EDIF file was created in a project called test1.

VHDL Instantiation Example

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity test is
port( FIRST: in STD_LOGIC;
      SECOND: in STD_LOGIC;
      THIRD: out STD_LOGIC);
end test;

architecture inside of test is

component test1
port (A: in STD_LOGIC; B: in STD_LOGIC; C: out STD_LOGIC);
end component;

begin

U1: test1 port map(A=>FIRST,B=>SECOND,C=>THIRD);

end inside;
```

Schematic (EDIF file) in an VHDL or Verilog Design IV



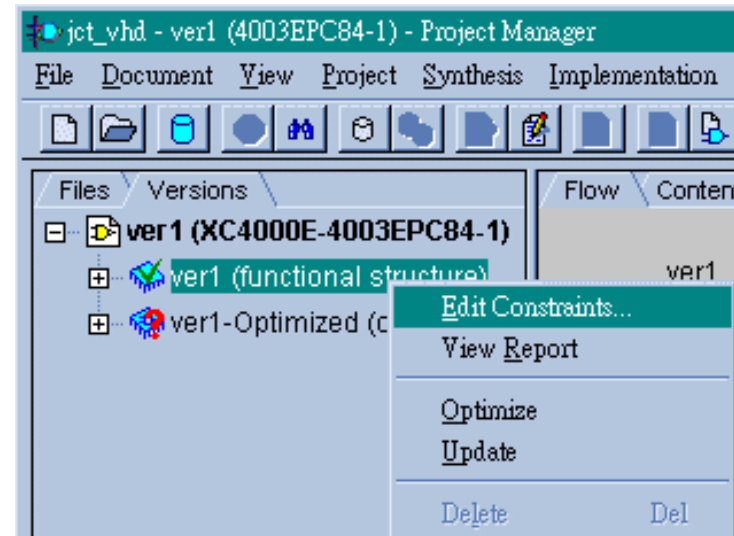
- 7. The HDL design with the instantiated EDIF file can then be synthesized (click the Synthesis button on the Flow tab).
- **Note:**
 - When the design is synthesized, a warning is generated that the EDIF schematic module is unexpanded.
 - Modules instantiated as black boxes are not elaborated and optimized.
 - The warning message is just reflecting the black box instantiation.
 - Expansion of the EDIF module takes place during the Translation stage of the Implementation phase.
 - **The instantiated file must be in the same directory as the HDL code in which it is instantiated.**

Timing Constraints with Foundation Express

Applying Constraints with the Express GUI



- Use Express Constraint GUI



- The Implementation window has three different tabs where M1 constraints can be applied: Clock, Paths, and Ports.
 - The Clocks tab allows you to specify overall speeds for the clocks in a design.
 - The Paths tab allows you precise control of point-to-point timing in a design.
 - The Ports tab allows NODELAY, pullups/pulldowns, and pin locations to be specified in a design.

Constraints that Express can apply



- Constraints that Express can apply:
 - FROM:TO timespecs which use FFS, LATCHES, and PADS
 - Pin location constraints
 - Slew rate
 - Pullup / Pulldown

Constraints that Express cannot apply

- Constraints that Express cannot apply:
 - TPSYNC
 - TPTHURU
 - TIG
 - OFFSET:IN:BEFORE
 - OFFSET:OUT:AFTER
 - user-RLOCs, RLOC_ORIGIN, RLOC_RANGE
 - non-I/O LOCs
 - KEEP
 - U_SET,H_SET,HU_SET
 - user-BLKNM and user-HBLKNM
 - PROHIBIT

Xilinx Logical Constraints

- In the M1 design flow, constraints or attributes that can be applied within a schematic, netlist, or UCF file are known as logical constraints.
- In order to use a logical constraint correctly, the "instance" name of the logic in a design must be used. Instance names are
 - XNF **SYM** record names
 - XNF **SIG** record names
 - XNF net names
 - **EXT** record names

XNF Example

- The SYM record name can be referenced by a logical constraint by using the instance name, `current_state_reg<4>`.
- A net called N10 or `current_state<4>` can also be used in a logical constraint.
- EXT records correspond to pins used on a package.
- The EXT records named CLK, DATA, and SYNCFLG can be referenced in a pin locking constraint.

```
SYM, current_state_reg<4>, DFF, LIBVER=2.0.0
PIN, D, I, next_state<4>, ,
PIN, C, I, N10, ,
PIN, Q, O, current_state<4>, ,
END
SIG, current_state<4>
EXT, CLK, I, ,
EXT, DATA, I, ,
EXT, SYNCFLG, O, ,
```

Reading Instance Names from an XNF file for UCF Constraints



- In M1, UCF constraints are applied by referencing instance names that are found in the XNF file.
 - TNM, LOC/RLOC use instance names
INST “current_state_reg<4>” TNM=group1;
INST “current_state_reg<4>” LOC=CLB_R5C5;
 - A pin on a device may be locked to a package-specific position by referencing the EXT record name and adding the .PAD string:
INST “DATA.PAD” LOC=P124;
 - KEEP or TNM, can be referenced by referencing the netname on the PIN record or SIG record:
NET “current_state<4>” KEEP;
NET “current_state<4>” TNM=group2;

Instance Names for LogiBLOX RAM/ROM



- Calculating Primitives for a LogiBLOX RAM/ROM Module
 - If the RAM/ROM depth is divisible by 32, then 32x1 primitives are used.
 - If the RAM/ROM depth is not divisible by 32, then 16x1 primitives are used instead.
 - In the case of dual-port RAMs, 16x1 primitives are always used.
- Naming Primitives in LogiBLOX RAM/ROM Modules
 - Example: RAM48X4

MEM0_0 MEM1_0 MEM2_0 MEM3_0
MEM0_1 MEM1_1 MEM2_1 MEM3_1
MEM0_2 MEM1_2 MEM2_2 MEM3_2

test.v:

```
module test(DATA,DATAOUT,ADDR,C,ENB);
input [3:0] DATA;
output [3:0] DATAOUT;
input [5:0] ADDR;
input C;
input ENB;
wire [3:0] dataoutreg;
reg [3:0] datareg;
reg [3:0] DATAOUT;
reg [5:0] addrreg;

inside U0 (.MDATA(datareg),.MDATAOUT(dataoutreg),.MADDR(addrreg),.C(C),.WE(ENB));

always @(posedge C)
    datareg = DATA;
always @(posedge C)
    DATAOUT = dataoutreg;
always @(posedge C)
    addrreg = ADDR;
endmodule
```

```
inside.v:

module inside(MDATA,MDATAOUT,MADDR,C,WE);

input [3:0] MDATA;
output [3:0] MDATAOUT;
input [5:0] MADDR;
input C;
input WE;

memory U1
(.A(MADDR),
 .DO(MDATAOUT),
 .DI(MDATA),
 .WR_EN(WE),
 .CLK(C));

endmodule
```

```
test.ucf
```

```
INST U0_U1 TNM = usermem;
TIMESPEC TS_6=FROM:FFS:TO:usermem:50;
INST U0_U1/mem0_0 LOC=CLB_R7C2;
```

Introduction to Efficient Synthesis

Xilinx HDL Design Methodology

To ensure the best design implementation, you need to understand two things:

- 1 Ramifications of coding styles
 - 2 How to take advantage of the Xilinx architecture
- This module explains ramifications of coding style. The Coding Tips for Virtex module explains how to take advantage of the Xilinx architecture

Synthesis Design Flow

- The synthesis process consists of two steps:
 - Translation to generic boolean logic, independent of ASIC or FPGA (chip) vendor and constraints
 - Optimization to a specific vendor (like Xilinx). Driven by constraints, and dependent on silicon features
- After synthesis, the design is ready for implementation

What Happens to the Design?



- Consider this simple circuit
- The design is synthesized into a netlist, then implemented

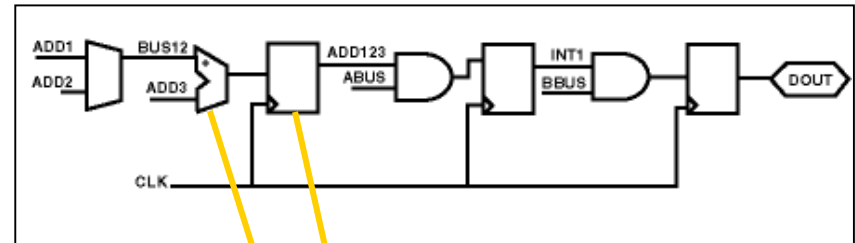
```
assign BUS12 = (SEL == 0)
               ? ADD1 : ADD2;

always @(posedge CLK)
  ADD123 <= BUS12 + ADD3;

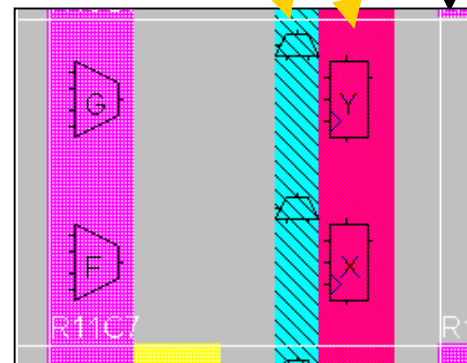
always @ (posedge CLK)
  INT1 <= ADD123 & ABUS;

always @ (posedge CLK)
  DOUT <= INT1 & BBUS;
endmodule
```

Synthesis



Implementation



Synthesis Tips

- Decisions made at the synthesis stage can affect implementation
 - This is where your HDL coding style has an impact
- What kinds of decisions are important?
 - Instantiation vs. Inference
 - Choosing good design hierarchy
 - Using parentheses to explicitly control design structure
 - Knowing the best coding style for your synthesis tool

Instantiation

- Components created with the Core Generator System and other “black boxes” must be instantiated
- Architectural features that may need to be instantiated

RAM/ROM

Readback

OSC

Bscan

WOR

WAND

SRL16

MUXF6

DLL

Instantiation vs. Inference

- Instantiate a component when you must dictate exactly which resource is needed
- Xilinx recommends inference whenever possible
 - Inference makes your code more portable
 - Synthesis tools cannot estimate timing delays through instantiated components
- Xilinx recommends use of the Core Generator System to create ALUs, fast multipliers, FIR Filters, etc. for instantiation

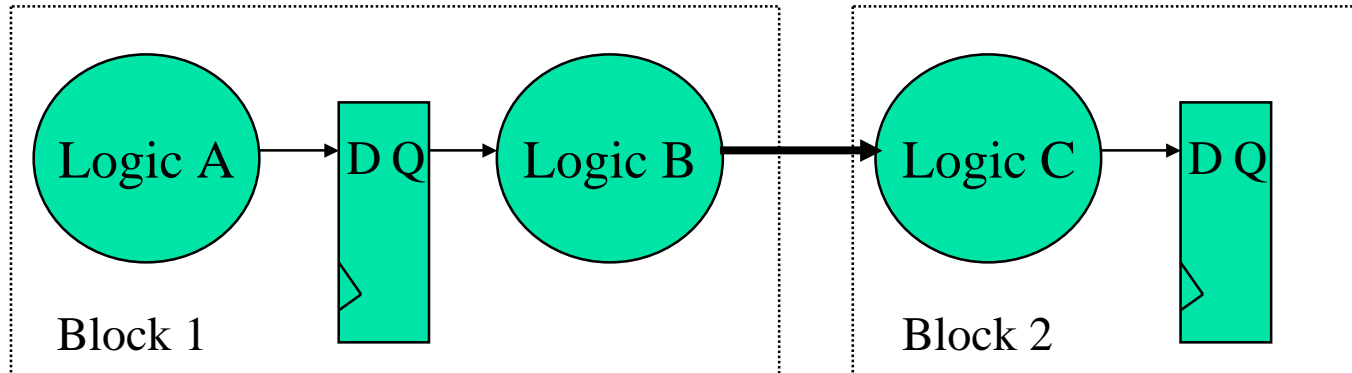
Hierarchy Guidelines

- Guidelines for good hierarchy
 - Use hierarchy to isolate technology
 - Place technology-specific cores in hierarchical blocks for design re-use
 - All arithmetic operators should be in the same process or always block to allow resource sharing
 - Isolate distinct logic types (state machines, random logic, data paths, etc)
 - Appropriate synthesis optimizations may be used for each logic type
 - Keep the number of lines of code below 400
 - Modules are easier to read, synthesize, and debug

Hierarchy Guidelines

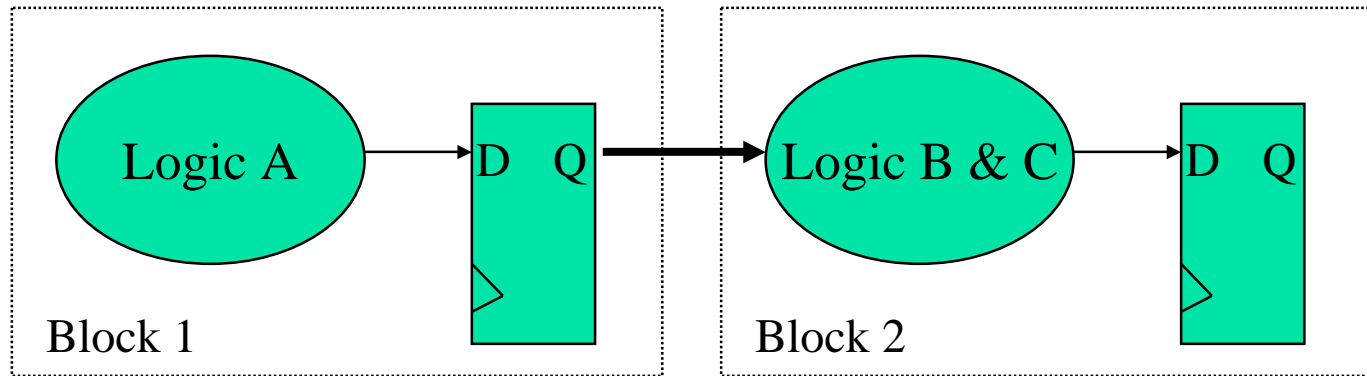
- Guidelines for good hierarchy
 - Create Hierarchical boundaries to minimize routing between modules and to maintain logical data flow between modules
 - Makes designs easier to read and debug
 - Repeatable results
 - Keep clock domains separated by hierarchy
 - Clock interaction is clearly defined
 - Easy to constrain
 - Allows different design sections to be synthesized individually, and tested before they are integrated into the whole design

Bad Hierarchical Example



- Synthesis tool must decide how to optimize Logic B and Logic C together
 - Can result in additional logic levels
- When simulating Block 2, you don't know the delays on the inputs coming from Block 1

Good Hierarchical Example



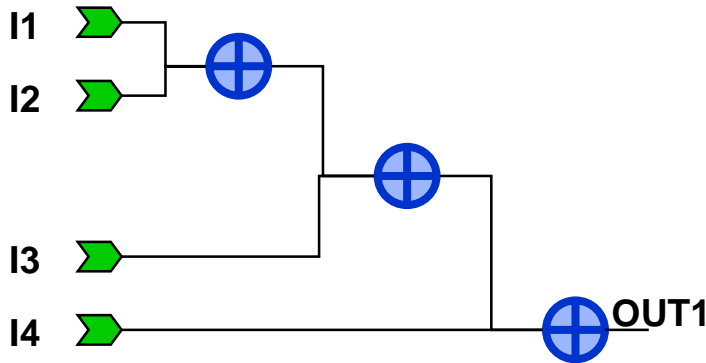
- Synthesis tool has no decisions to make
- Logic B & C will be optimized the same way every time
- When simulating Block 2, you know the delay on inputs from Block 1 is just a clock-to-Q delay

Use Parentheses to Control Structure

- What are the advantages and disadvantages of the first and second examples?

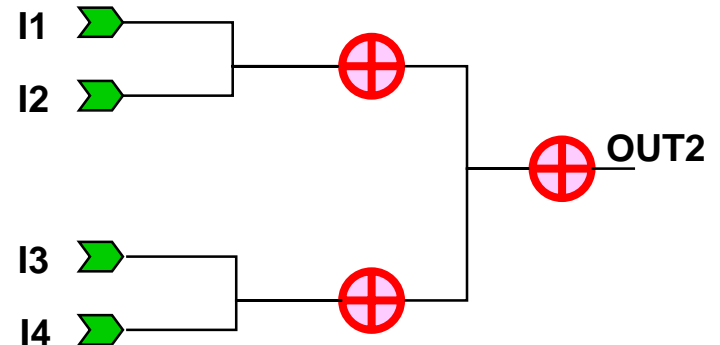
Example 1: No parentheses

```
OUT1 <= I1 + I2 + I3 + I4
```



Example 2: With parentheses

```
OUT2 <= (I1 + I2) + (I3 + I4)
```



Experiment with Your Synthesizer

- Learn how your coding style fits with your tools before you try a large design
- Pick a simple to mildly complex function
 - Code it a few different ways
 - Try instantiating components
 - Try different synthesis options
- Run this example through the implementation tools and see what you get
 - Which is smaller? Which is faster?
 - Which method provides the best overall results?

Experiment with Your Synthesizer

- In general, don't let the synthesizer make decisions for you
 - Decide how you want a function to fit in the device before coding it
 - Be familiar with the speed, area, and flexibility trade-offs you can make to reach your goals
- The faster your clock speed, the more you need to know about your synthesis tool and the target architecture

Synthesize From the Bottom Up



- Build a set of standard functions you can re-use throughout your design
 - MUXes, register banks, FIFOs, adders, counters and other standard functions
- Compile from the bottom up
 - Synthesize each low level module separately the first time
 - Merge these lower levels in the implementation tools to get a resource estimate for each module and design sub-section
 - Be sure each sub-block can meet your requirements before adding it to the main design. Is it the right size? Is it fast enough?

Review Questions

- List and define the two steps that make up synthesis
- Compare the applications of inference and instantiation

Simulation With Foundation Express Netlists



Simulation Flows in Foundation Express 3.x

- Pre-synthesis RTL simulation (functional simulation)
- Post-synthesis pre-map simulation (post-translate simulation)
- Post-map pre-route simulation (post-map simulation)
- Post-par simulation (timing simulation).

Pre-Synthesis RTL Simulation

- Functional simulation
- Determine if the behavior of the HDL is what is expected.
- For RTL simulation, the HDL code can not have instantiated FFS, OBUF, and OBUFTs.
- Need Verilog/VHDL simulator
 - Verilog: Verilog XL(Cadence), VCS(Viewlogic), VeriWell, ModelSIM
 - VHDL: VSS(Synopsys), Speedwave(Viewlogic), ACTIVE-VHDL, ModelSIM

Simulation of Modules/Black Box Designs



- Black box modules instantiated in Express
 - LogiBLOX modules, LogiCORE, or unified XNF files.
- These instantiated modules cannot be simulated in the pre-synthesis RTL simulation flow, except for LogiBLOX instantiations.
 - The LogiBLOX tool can create behavioral HDL models which can be used for pre-synthesis RTL simulation.
 - **Instantiated black boxes without behavior, like an XNF file, can only be functionally simulated after NGDBuild has been run.**

Post-Synthesis Pre-Map Simulation

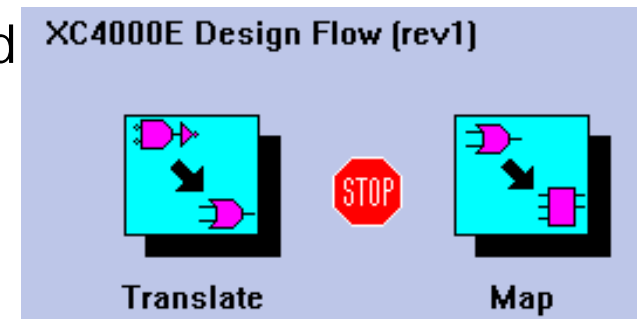
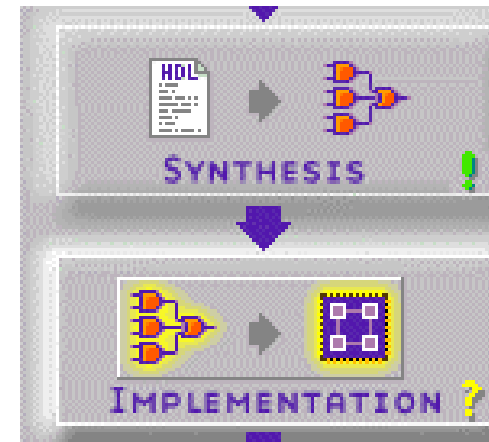


- Functional simulation
- The behavior in this type of simulation is post-synthesis
- NGDBuild (Translate) converts all logic in the design to simulation primitives (SIMPRIMS)
- Logic converted to SIMPRIMS can be simulated by the Xilinx Verilog or VITAL simulation libraries.
- The logic simulated in this flow has not been trimmed.
 - It is always possible that trimming of a design could change behavior.

Post-Synthesis Pre-Map Simulation Flow



- 1. Synthesize HDL codes
 - Click **SYNTHESIS** button
- 2. Run Translate to create .ngd.
 - Click **IMPLEMENTATION** button to run Flow Engine
 - Stop after Translate
 - Setup > Stop After... > Translate
- 3. Open a DOS window.
- 4. Run NGD2VHDL or NGD2VER on the .ngd file produced by Translate.
 - **ngd2ver** input_file.ngd output_file.v
 - **ngd2vhd** input_file.ngd output_file.vhd
- 5. Combine the behavioral .v file or .vhd file with the testbench file and then simulate them.

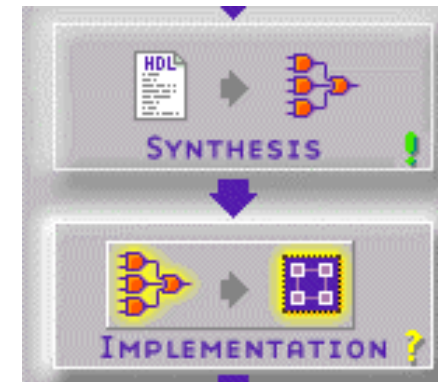


Post-Map Pre-Route Simulation

- FPGA Only
- The M1 MAP tool trims redundant logic from a design and maps logic to the appropriate technology.
- All the logic in the design has already been transformed into SIMPRIMS.

Post-Map Pre-Route Simulation Flow I

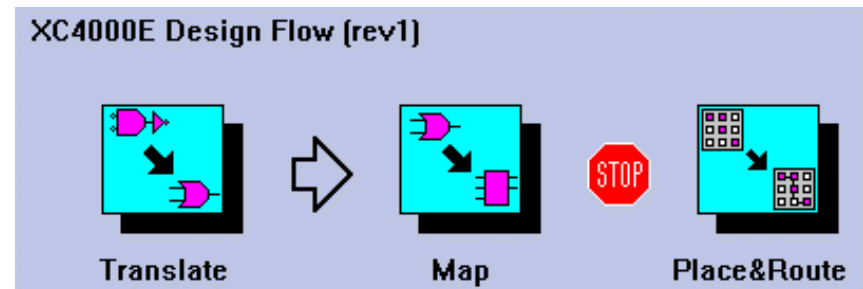
1. Synthesize HDL code in Foundation Express
2. Run Translate and Map to create .ncd.
 - Click **IMPLEMENTATION** button to run Flow Engine
 - Stop after Map
 - Setup > Stop After... > Map
 - Map produces an .ncd and an .ngm file from the .ngd file created by Translate.
 - The .ncd file that is created is named **map.ncd**. The file is located in the revision directory.
3. Open a DOS window.



4. Run NGDANNO command to create an .nga file.

ngdanno -o output_file.nga map.ncd map.ngm

- 5. Run NGD2VHDL or NGD2VER on the .nga file produced by NGDANNO.



Post-Map Pre-Route Simulation Flow II



`ngd2ver` input_file.nga output_file.v

`ngd2vhdl` input_file.nga output_file.vhd

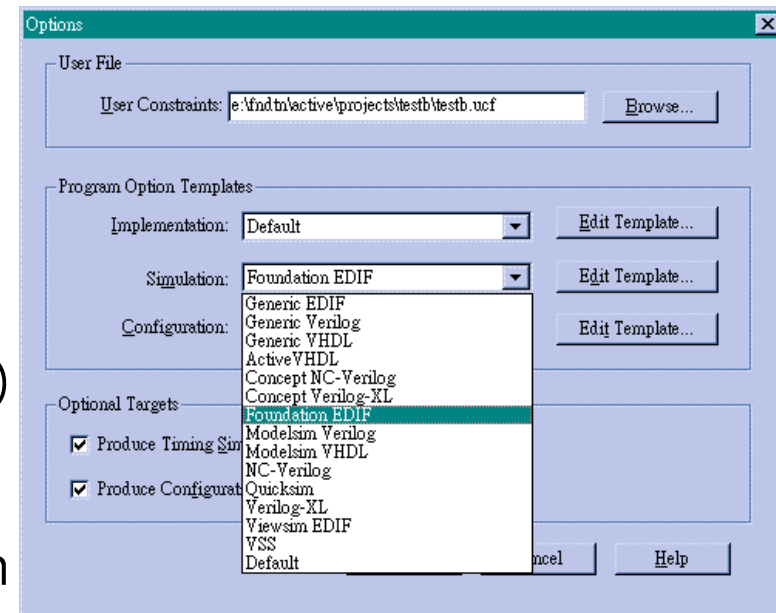
- 6. Combine the behavioral .v file or .vhd file with the testbench file and then simulate them

Post-PAR Simulation

- The traditional HDL timing simulation
- Timing information for a design is back-annotated into an SDF file.
- Separation of the timing information into an SDF file has several advantages.
 - Timing information separated into a SDF file allows the Xilinx design to be simulated in many third party HDL simulators of choice.
 - Separation of timing into an SDF file allows for the increased speed of timing simulation.

Post-PAR Simulation Flow I

1. Synthesize HDL codes
2. Run Flow Engine to create timing simulation netlists.
 - A. Click **IMPLEMENTATION** button to run Flow Engine
 - B. Click the **Options** button. This opens the Options dialog box.
 - C. Verify that the Simulation Template is you want. (Change it to Foundation EDIF, if necessary.)
 - D. In the Options dialog box under Optional Targets, select **Produce Timing Simulation Data** and then click OK.
 - E. Implement the design. Click OK in the Synthesis/Implementation dialog box.



Post-PAR Simulation Flow II

- 3. Combine the behavioral .v file or .vhd file with the testbench file and simulate.
 - Depending on whether you created VHDL or Verilog simulation data, a *time_sim.v* or *time_sim.vhd* file is created in your project directory.
 - In addition, a *time_sim.sdf* file with complete timing data is also produced.

Simulating a Design With the Foundation Logic Simulator



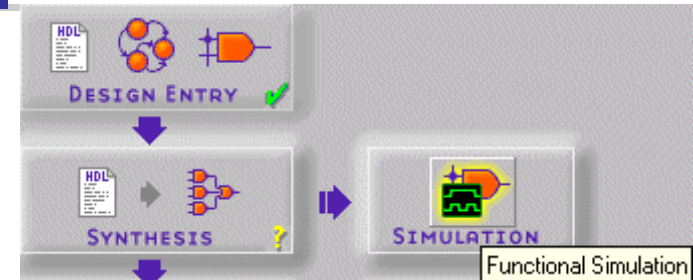
- The Foundation 3.x gate-level simulator can be used for functional and timing simulation.
- Simulation using the gate simulator requires that a flat netlist be created before the simulator can be used.

Functional Simulation using the Logic Simulator I



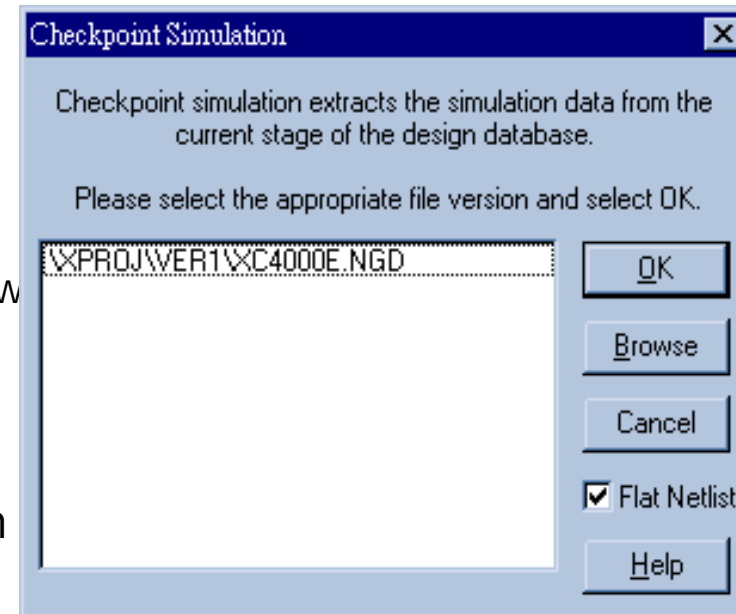
Pure HDL Designs

- 1. Synthesize HDL codes first.
- 2. Click **SIMULATION** button in Project Manager to invoke Gate-level functional simulator



Top-Level HDL Designs with Black Box

- 1. Synthesize HDL codes first.
- 2. Run Translate
 - Click **IMPLEMENTATION** button to run Flow Engine
 - Stop after Translate
 - Setup > Stop After... > Translate
- 3.. When the Flow Engine is finished, return Foundation Project Manager and select *Tools -> Simulation/Verification -> Checkpoint Gate Simulation Control...*
- 4. Select the design listed in the Checkpoint Simulation window and click OK.



Functional Simulation using the Logic Simulator II

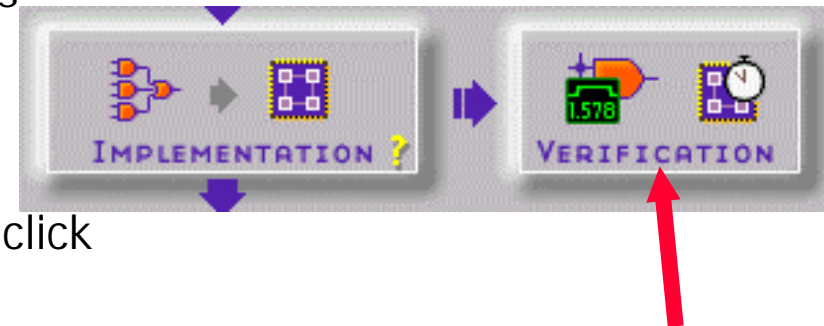


- 5. After selecting OK in the Checkpoint Simulation window, the Project Manager indicates that NGD2EDIF is running.
 - When NGD2EDIF is finished, the Foundation simulator automatically **starts**.
- 6. Proceed with simulation by selecting the signals to stimulate for the *Signals -> Add Signals*.
 - The signals listed correspond to the top-level entity ports in VHDL or top-level module ports in Verilog.
- For more information on using the Foundation Simulator, please consult the Foundation Online Help.

Timing Simulation using the Logic Simulator



- 1. Synthesize HDL codes.
- 2. Run Flow Engine.
 - Click **IMPLEMENTATION** button to run Flow Engine
 - Verify that the Simulation Template is **Foundation EDIF**.
 - In the Options dialog box under Optional Targets, select **Produce Timing Simulation Data** and then click OK.



- 3. After implementing the design in the Design Manager, return to the Foundation Project Manager and click the *Timing Simulation* button in Verification to start timing simulation.
- Foundation automatically translates this back-annotated timing netlist to an EDIF file and loads the simulator.

Xilinx CORE Generator



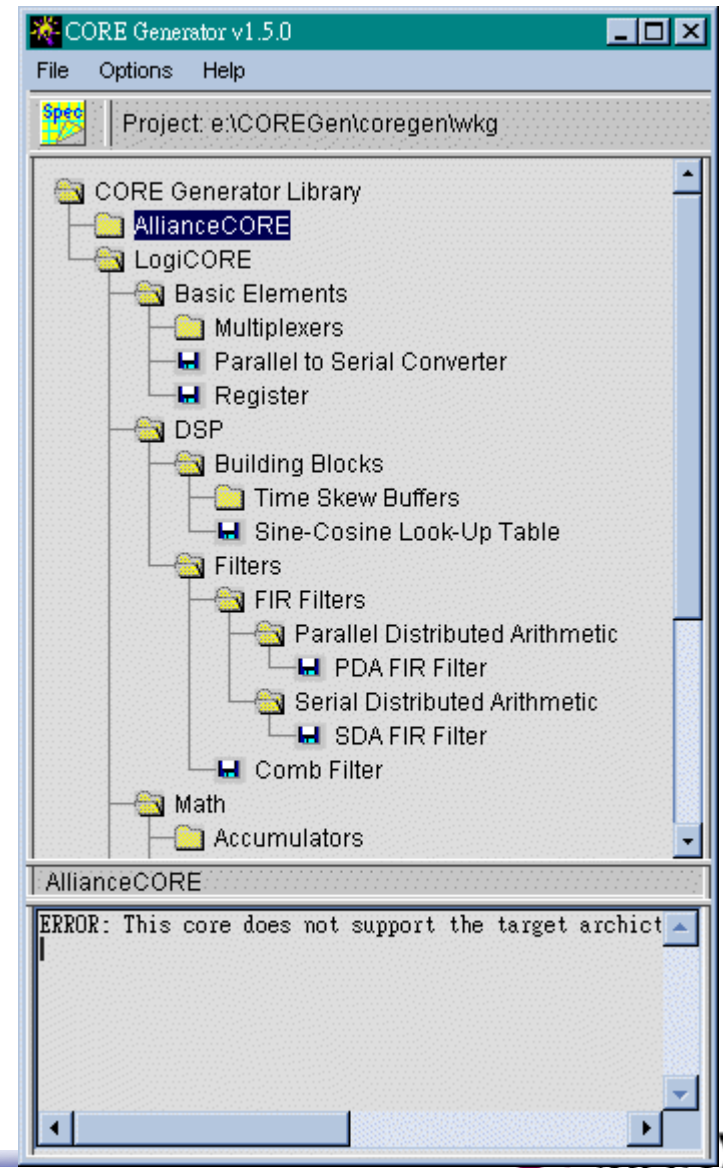
Overview I

- Parameterizable cores optimized for FPGAs.
- Tree-like presentation
- Deliver behavioral simulation models, schematic symbols and HDL instantiation templates for fitting design environment.
- Many functions can be used
 - **Basic Elements**: Multiplexers, Parallel to Serial Converter, Register
 - **DSP**: Time Skew Buffers, Sine-Cosine Lookup Table, PDA FIR Filter, SDA FIR Filter Single-Channel, Comb Filter

Overview II



- **Math:** Accumulator, Adders/Subtractors, Complements, Constant Coefficient Multipliers, Parallel Multipliers, Integrator, square Root
- **Memories:** Delay Element, Registered DualPort RAM, Registered ROM, Registered SinglePort RAM, Synchronous FIFO



How to Obtain New Cores and Updates



- New cores can be downloaded from the Xilinx web site and easily added to the CORE Generator.
- Bookmark

<http://www.xilinx.com/products/logicore/coregen>
and keep in touch regularly for updates.

Project Management

- Setup the Coregen.ini File
- Set the CORE Generator System Options
- Set the CORE Generator Output Options

Setup the Coregen.ini File

- coregen.ini File
 - “coregen.ini” is created during the install of the CORE Generator and can be found in the following directory:
<CORE_Generator_Install_Path>/coregen/wkg
 - Many parameters are set in this file
- The parameters expressed in the coregen.ini file:
 - The following parameters are determined during installation, and usually do not need to be changed:
 - **CoreGenPath, FoundationPath, AcrobatPath, and AcrobatName.**
 - Other settings will usually need to be changed by the user, such as
 - **SelectedProducts, ProjectPath, BusFormat, and XilinxFamily.**
 - Changes to your output options and system options are best done through the CORE Generator **Output format** and **System Options forms**.
 - **Options -> Output format** for the Output format form
 - **Options -> System Options...** for the System Options form.

Coregen.ini Syntax Summary



```
SET CoreGenPath = <some platform-specific path to the
                  $COREGEN\coregen directory>
SET FoundationPath = <some user-specific path to the Foundation tools>
SET ProjectPath = <some user-specific path to the current project>
SET TargetSymbolLibrary = <Viewlogic library name>
SET AcrobatPath = <some user-specific path to Acrobat install>
SET AcrobatName = <Acrobat executable name>
SET SelectedProducts = [ImpNetlist | FoundationSym| VHDLSym |
                        VHDLSim | ViewSym | VerilogSim | VerilogSym]
SET XilinxFamily = [All_XC4000_Families | All_Spartan_Families
                    | All_Virtex_Families]
SET BusFormat = [BusFormatAngleBracket | BusFormatSquareBracket
                 | BusFormatParen | BusFormatNoBracket]
SET OverwriteFiles =[true | false]
```

Sample coregen.ini file

```
SET FoundationPath = C:\FNDTN\ACTIVE

SET ProjectPath = C:\cg95_cur\coregen\wkg

SET TargetSymbolLibrary = primary

SET XilinxFamily = All_XC4000_Families

SET AcrobatPath = C:\Acrobat3\Reader\

SET AcrobatName = AcroRd32.exe

SET SelectedProducts = ImpNetlist FoundationSym
```

Setup SelectedProducts I

- **SelectedProducts** - This setting defines the type of output files to be created each time a module is built.
 - SET SelectedProducts = ImpNetlist FoundationSym**
- **ImpNetlist**: Implementation Netlist.
 - This is the gate level netlist that will be used to implement the logic of the particular core that the COREGenerator System has created.
 - In an HDL synthesis design flow, an HDL instantiation of the core references this netlist as a “black box” in a schematic design flow, a schematic symbol references this netlist.
- **ViewSym**: ViewLogic Schematic Symbol.
 - When specified this option will create a ViewLogic schematic symbol that can be used in your ViewLogic schematic capture tools to instantiate the module netlist.
- **FoundationSym**: Foundation Schematic Symbol.
 - When specified this option will create a Foundation schematic symbol that can be used in your Foundation schematic capture tools to instantiate the module netlist.

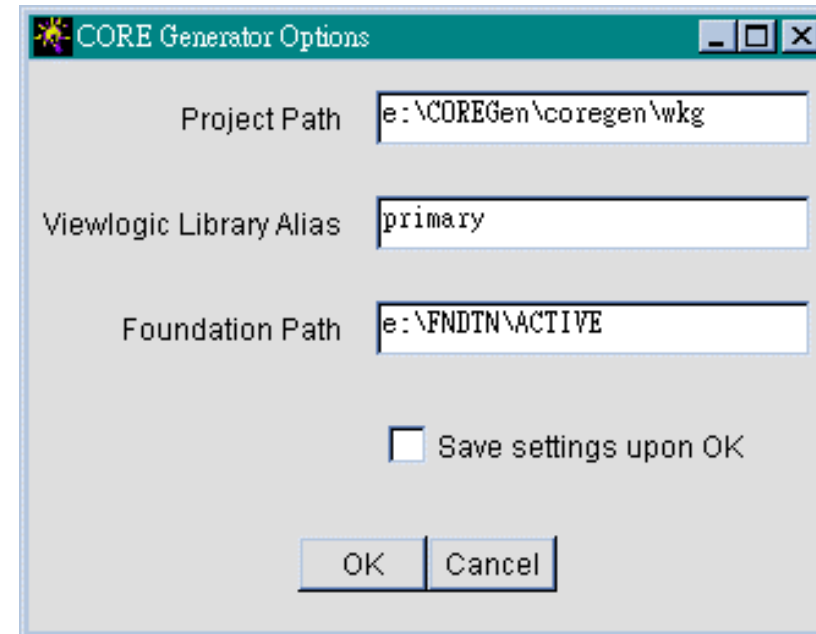
Setup SelectedProducts II

- **VHDLSym**: VHDL Instantiation Template.
 - When specified this option will create a VHDL instantiation template that can be used in your HDL design capture tools to instantiate the module netlist.
- **VHDLSim**: VHDL Behavioral Simulation Model.
 - When specified this option will create a VHDL simulation model, which can be used to verify the functional simulation of the module netlist.
- **VerilogSym**: Verilog Instantiation Template.
 - When specified this option will create a Verilog instantiation template that can be used in your HDL design capture tools to instantiate the module netlist.
- **VerilogSim**: Verilog Behavioral Simulation Model.
 - When specified this option will create a Verilog simulation model, which can be used to verify the functional simulation of the module netlist.

Set the CORE Generator System Options



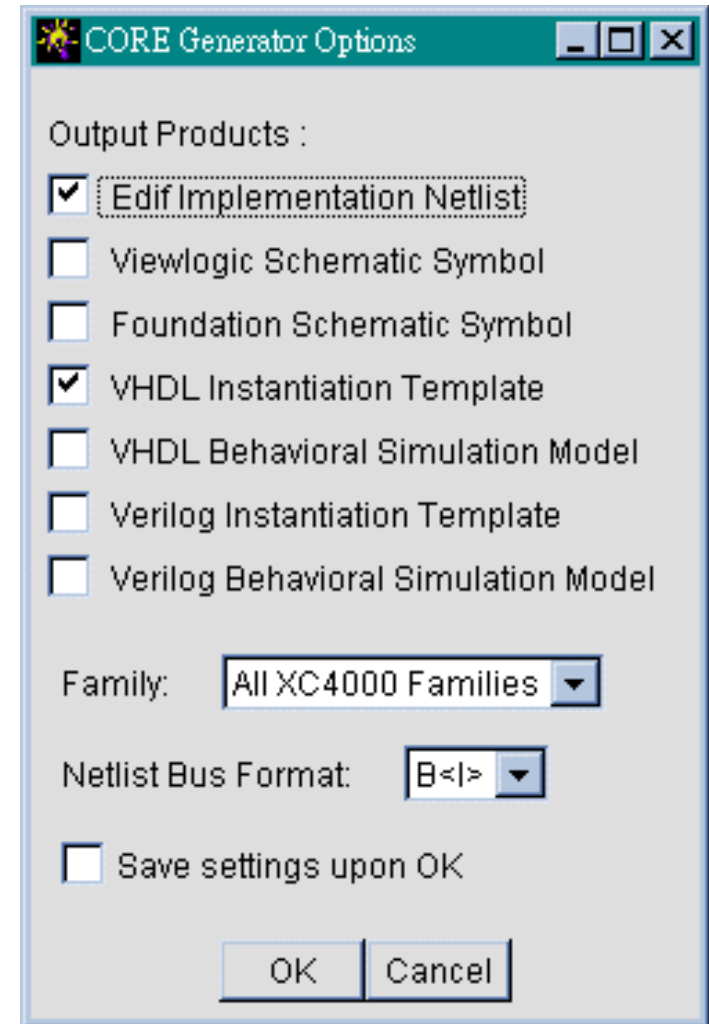
- Options -> System Options ...
 - All default settings are set in “coregen.ini”
- Project Path
 - This setting defines the project working directory
- Viewlogic Library Alias
 - This setting defines the name of the ViewLogic library alias.
- Foundation Path
 - This setting defines the path location of the Foundation CAE tools
- Save settings upon OK
 - Make these options permanent



Set the CORE Generator Output Options I



- Options -> Output Format...
- Edif Implementation Netlist
 - A gate level netlist
 - Output: `<CoreName>.edn`
- Viewlogic Schematic Symbol
 - A Viewlogic schematic symbol and a simulation wire file
 - Output: `wir\<CoreName>.1`
`sym\<CoreName>.1`
- Foundation Schematic Symbol
 - A Foundation schematic symbol and a simulation file
 - Output: `<CoreName>.alr`
`lib\project_name.sym`



Set the CORE Generator Output Options II



- VHDL Instantiation Template
 - A VHDL instantiation template that can be used in your HDL design capture tools to instantiate the module netlist.
 - Output: `<CoreName>.vhi`
- VHDL Behavioral Simulation Model
 - A VHDL simulation model which can be used to verify the functional simulation of the module netlist.
 - This file is not intended to be synthesized.
 - It is only provided for behavioral simulation
 - Output: `<CoreName>.vhd`
- Verilog Instantiation Template
 - A Verilog instantiation template that can be used in your HDL design capture tools to instantiate the module netlist.
 - Output: `<CoreName>.vei`

Set the CORE Generator Output Options III



- Verilog Behavioral Simulation Model
 - A Verilog simulation model which can be used to verify the functional simulation of the module netlist.
 - This file is not intended to be synthesized.
 - It is only provided for behavioral simulation
 - Output: `<CoreName>.v`
- The Family drop-down box
 - Restrict the CORE Generator module browser to show only those modules that may be targeted to the selected family of devices.
 - At this time the supported families of devices are
 - All XC4000 Families
 - All Spartan Families
 - All Virtex Families

Using the CORE Generator

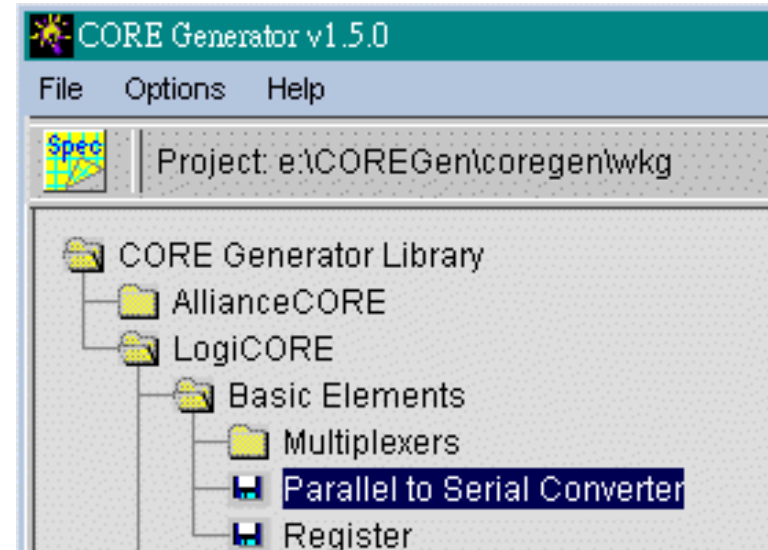
- Core Browser Tree
- Getting Module Data Sheets
- Parameterizing a Module
- COE Files

Core Browser Tree

- The most common view of the CORE Generator is the **Core Browser** window.
- This window allows you to browse the many cores that are available from the CORE Generator installation.
- Cores that fall into particular application categories are grouped into folders to assist you in locating the module appropriate for your needs.
- To expand a folder, double click on the folder icon to the left of the folder name.

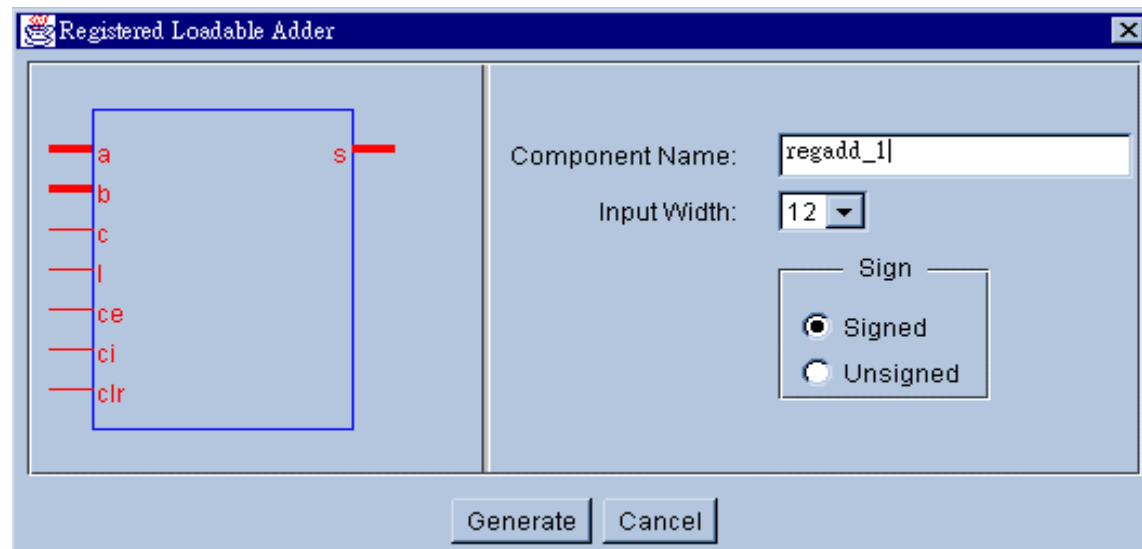
Accessing Core Data Sheet

- A data sheet for the selected core can be requested at any time
 - 1. select the core in the core browser
 - 2. click on the **Spec** button on the CORE Generator toolbar.
 - This action will launch the Acrobat Reader application and display the core data sheet.



Parameterizing a Core I

- Most cores have a parameterization window.
- Double-clicking on a core's icon or descriptive text will reveal the parameterization window for that module.
- For example : Registered Loadable Adder



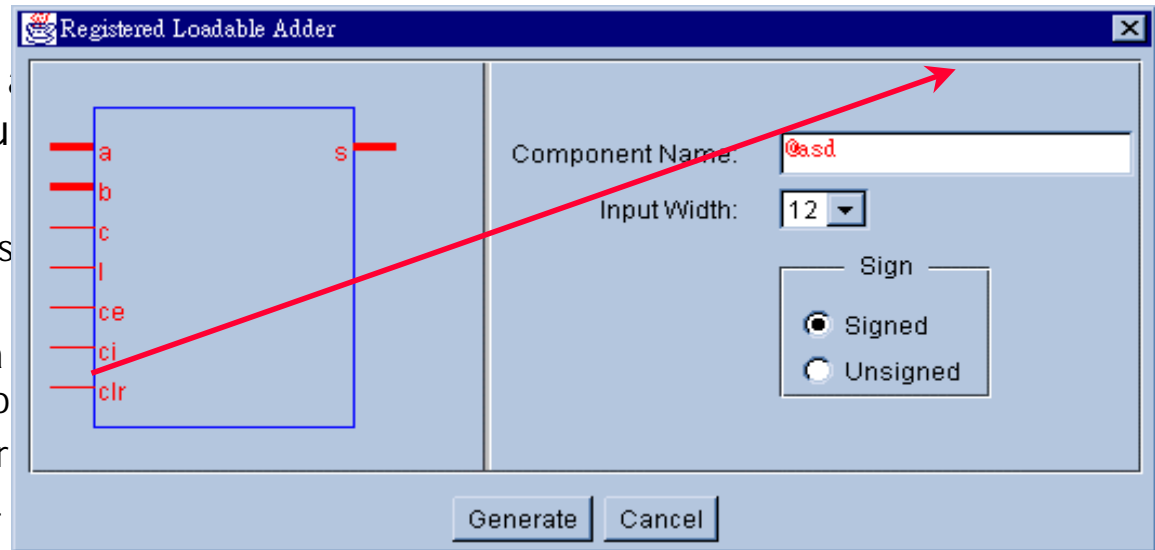
Parameterizing a Core II

■ Component Name

- allows you to assign a name to a module that you create

■ Restrictions:

- Up to 8 characters
- No extensions
- Must begin with a letter character: a-z (No 0-9, _)
- May include (after first character): 0-9, _



- The illegal or invalid field will be highlighted in red until the problem is corrected.

- Assuming there are no problems with any of the parameters that have been specified, pressing **Generate** will cause the CORE Generator to create files of the requested types.
- Pressing **Cancel** will return you to the module browser window without generating any files.

COE Files

- Some cores can require a significant amount of information to completely specify their behavior.
- Cores of this type will have a button on their parameterization windows with which you can load their parameterization information from a file.
- Additional information about a particular Core's COE file can be found in that Core's datasheet.
 - For examples of PDA FIR, RAM, and ROM COE files, please look in the [coregen/wkg](#) directory.

Example - PDA FIR Filter

PDA FIR Filter

Component Name:

Input Width:

Output Width:

For full precision filter output, select 20 bits.

Sign

☒ Signed Input Data

☐ Unsigned Input Data

Impulse Response

☒ Symmetry

☐ No Symmetry

☐ Anti-Symmetry

☐ Generate Cascadable Section

Number Of Taps:

Coefficient Width:

☐ Trim Empty ROMs

Load Coefficients... Show Coefficients...

.coe file:

Generate Cancel

.COE Format

- The parameterization window for a FIR filter, which has a Load Coefficients..., button.
- Files containing this type of information should be ASCII text files and take the extension **.COE**.
- The format for the .COE file is illustrated below:

Keyword = Value ; Optional Comment

Keyword = Value ; Optional Comment

...

CoefData = Data_Value, Data_Value, ...;

- **CoefData** or **MemData** keywords must appear at the end of the file as any further keywords will be ignored.
- Any text after the semicolon is treated as a comment and will be ignored.

.COE Examples - PDA FIR

```
***** EXAMPLE: PDA FIR *****  
component_name=fltr16;  
Number_of_taps=16;  
Input_Width = 8;  
Signed_Input_Data = true;  
Output_Width = 15;  
Coef_Width = 8;  
Symmetry = true;  
Radix = 10;  
Signed_Coefficient = true  
coefdata=1,-3,7,9,78,80,127,-128;
```

.COE Examples - SDA FIR

```
***** EXAMPLE: SDA FIR *****  
component_name = sdafir;  
number_of_taps = 6;  
radix = 10;  
input_Width = 10;  
output_Width = 24;  
coef_Width = 11;  
symmetry = false;  
coefdata = -1,18,122,418,-40,3;
```

.COE Examples - RAM

```
***** EXAMPLE: RAM *****  
component_name=ram16x12;  
Data_Width = 12;  
Address_Width = 4;  
Depth = 16;  
Radix = 16;  
memdata=346,EDA,0D6,F91,079,FC8,053,FE2,03C,FF2,02D,  
FFB,022,002,01A,005;
```

.COE Examples - ROM

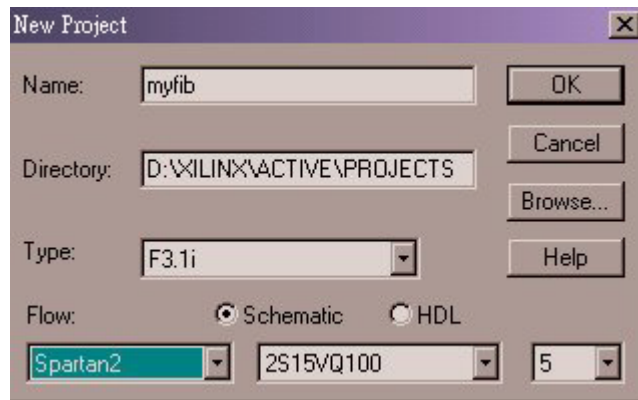
```
***** EXAMPLE: ROM *****  
component_name=rom32x8;  
Data_Width = 8;  
Address_Width = 5;  
Depth = 32;  
Radix = 10;  
memdata=127,127,127,127,127,126,126,126,125,125,  
125,4,3,2,0,-1,-2,-4,-5,-6,-8,-9,-11,-12,-13,-38,  
-39,-41,-42,-44,-45,-128;
```

CORE Generator Design Flow

Foundation Schematic Flow
and
Foundation Express Flow

Foundation Schematic Flow I

- 1. Set a Foundation Project
 - Create a new project or Select an existing Schematic project from the Foundation Project Manager.

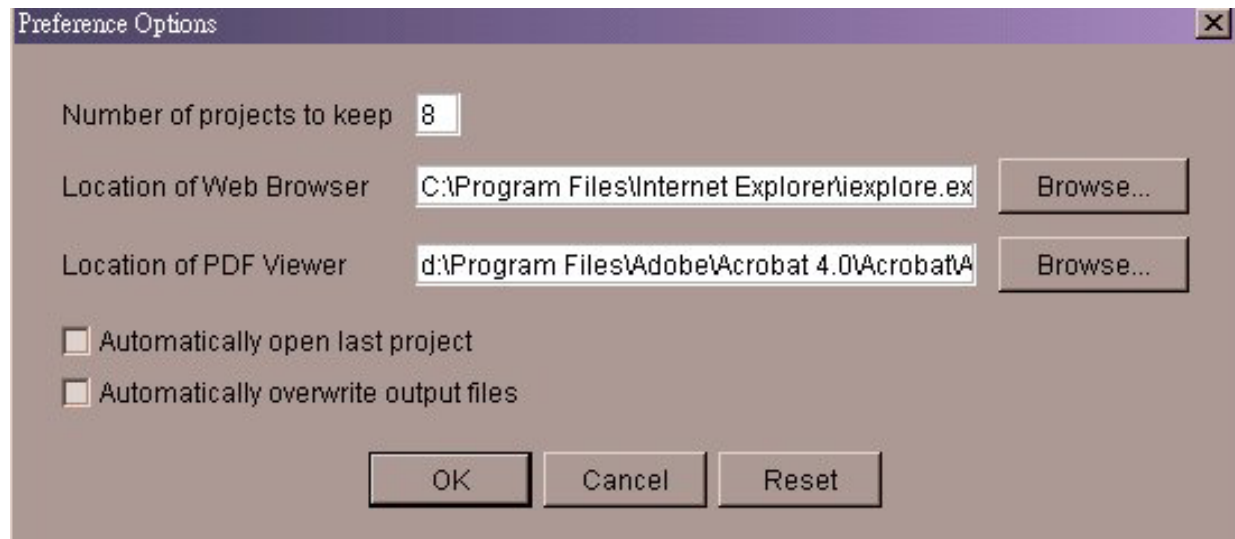


- 2. Set the CORE Generator Output Format
 - From the CORE Generator **Options** menu, select **Output Format**, and check the following options:

Foundation Schematic Flow II



- 3. Set the System Preference
 - From the CORE Generator Project menu, select **Preference**.
 - Set the **Path** to point to your related application directory.
 - Web Viewer
 - PDF Viewer
 - The selected Project Path should be a valid



Foundation Schematic Flow III



- 4. Select the module you want to generate by navigating to the desired module and clicking on it.
 - Click on the SPEC button on the CORE Generator toolbar to review the module's datasheet
 - Double-click on the selected module to call up its parameterization window.
 - When you have entered all the parameterization details required by the module click the **Generate** button.
- 5. Output Files
 - A Foundation symbol, a Netlist File (**.EDN**) and a simulation file (**.ALR**) are created.
 - The symbol is automatically copied to the Foundation Project directory.
- 6. Load the Symbol in the Schematic Editor
 - Open the Foundation schematic editor, the new symbols will be found in the symbol list for the selected project.
 - The simulation and compilation flow is the same as the flow for a design containing only Unified Library components.

Foundation HDL Flow I

- 1. Set a Foundation Project
 - Create a new project or Select an existing HDL Flow project from the Foundation Project Manager.
 - The files generated by the CORE Generator System will automatically be copied into the selected project directory.
- 2. Set the CORE Generator Output Format
 - From the CORE Generator Options menu select **Output Format**, and check the following options:

Select either VHDL or Verilog Instantiation template



Foundation HDL Flow II

- 4. Select the module you want to generate by navigating to the desired module and clicking on it.
 - Click on the SPEC button on the CORE Generator toolbar to review the module's datasheet
 - Double-click on the selected module to call up its parameterization window.
 - When you have entered all the parameterization details required by the module click the **Generate** button.
 - **Note: Do not name your Module with the same name as a Unified Library component as this will cause the Synthesizer to use the Unified Library XNF file instead of the EDIF file generated by the CORE Generator System.**

Foundation HDL Flow III

■ 5. Output Files

- A VHDL or Verilog instantiation template (module_name.**VHI** or module_name.**VEI**) and a Netlist File (**.EDN**) will be created and copied into the CORE Generator project directory.
- The instantiation template contains the component declaration as well as the Port Map/Module declaration for the module that has been selected.
- This instantiation template can be copied and pasted into the top-level HDL file.

VHDL Example - Instantiation template

**** 8 Bit Adder VHDL Instantiation template: adder8.vhi****

```
component adder8 port (  
    a: IN std_logic_VECTOR(7 downto 0);  
    b: IN std_logic_VECTOR(7 downto 0);  
    s: OUT std_logic_VECTOR(8 downto 0);  
    c: IN std_logic;  
    ce: IN std_logic;  
    ci: IN std_logic;  
    clr: IN std_logic);  
end component;
```

```
yourInstance : adder8 port map (  
    a => a,  
    b => b,  
    s => s,  
    c => c,  
    ce => ce,  
    ci => ci,  
    clr => clr);
```

VHDL Example - Top Level VHDL

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity adder8_top is
    port (ina, inb: in STD_LOGIC_VECTOR (7 downto 0);
          clk, enable, carry, clear: in STD_LOGIC;
          qout: out STD_LOGIC_VECTOR (8 downto 0));
end adder8_top;
architecture BEHAV of adder8_top is
-- Instantiate the adder8.edn file.
component adder8 port (
    a: IN std_logic_VECTOR(7 downto 0);
    b: IN std_logic_VECTOR(7 downto 0);
    s: OUT std_logic_VECTOR(8 downto 0);
    c: IN std_logic;
    ce: IN std_logic;
    ci: IN std_logic;
    clr: IN std_logic);
end component;
begin
u1 : adder8 port map (
    a => ina,
    b => inb,
    s => qout,
    c => clk,
    ce => enable,
    ci => carry,
    clr => clear);
end BEHAV;
```

Verilog Example - Instantiation template

```
** 8 Bit Adder Verilog Instantiation template: adder8.vei **

module adder8 ( a, b, s, c, ce, ci, clr);
input [7:0] a;
input [7:0] b;
output [8:0] s;
input c;
input ce;
input ci;
input clr;
endmodule

// The following is an example of an instantiation :
adder8 YourInstanceName (
    .a(a),
    .b(b),
    .s(s),
    .c(c),
    .ce(ce),
    .ci(ci),
    .clr(clr));

*****
```

Verilog Example -

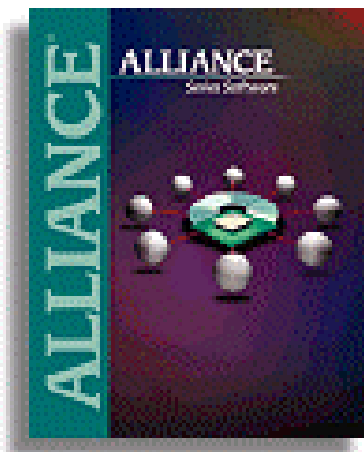
Top Level Verilog

Instantiation Module Declaration

```
***** Top Level Verilog file: adder8_top.v *****
module adder8_top(ina, inb, clk, enable, carry, clear, qout);
input [7:0] ina;
input [7:0] inb;
input clk;
input enable;
input carry;
input clear;
output [8:0] qout;
//instantiate the adder8.xnf file
adder8 U1 (
    .a(ina),
    .b(inb),
    .s(qout),
    .c(clk),
    .ce(enable),
    .ci(carry),
    .clr(clear));
endmodule
*****
***** Instantiation Module Declaration: adder8.v *****
module adder8 ( a, b, s, c, ce, ci, clr);
input [7:0] a;
input [7:0] b;
output [8:0] s;
input c;
input ce;
input ci;
input clr;
endmodule
*****
```

Foundation HDL Flow IV

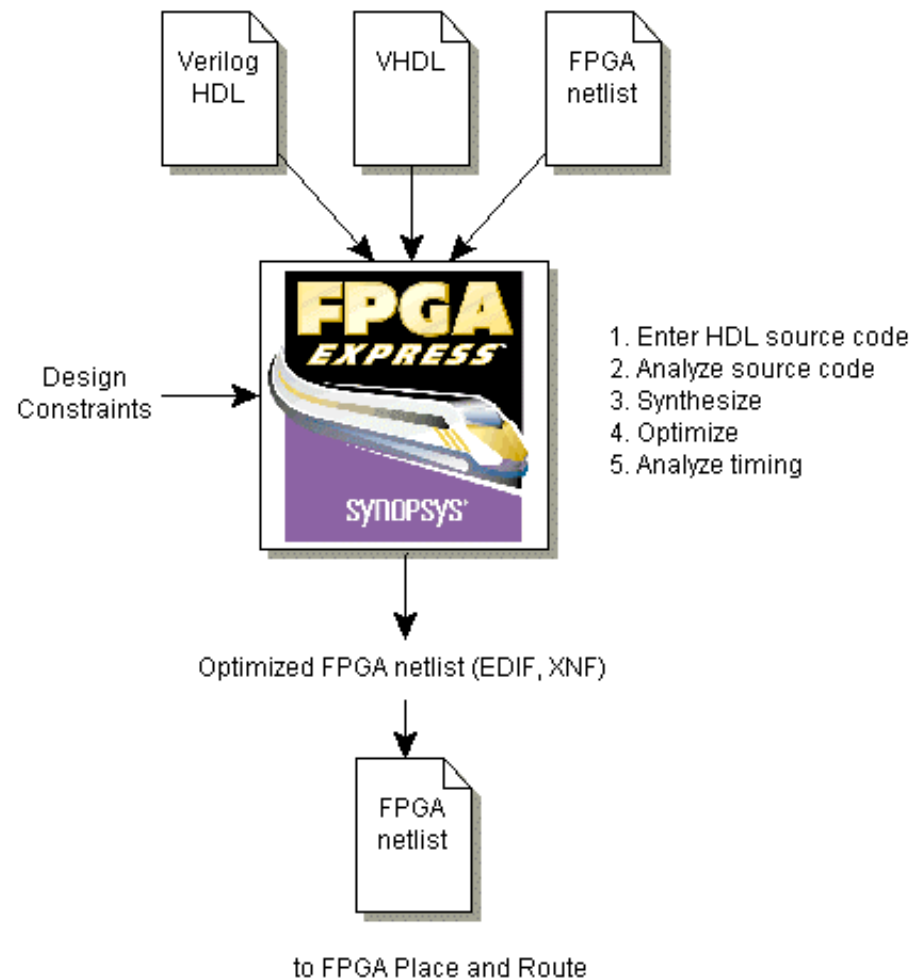
- 6. Compiling the Design in Foundation Express
 - 1. Create a new project or open an existing one in Foundation Express.
 - 2. Add all HDL files to be synthesized for the project.
Note: Do NOT add the EDIF files created by the CORE Generator System to the Express project. Also, do NOT add any HDL simulation files.
 - 3. **Verilog Only: Add a .v module declaration file for each instantiated block.**
 - 4. Select the top level entity and select Create Implementation to generate a new implementation.
 - 5. Optimize the implementation.
 - 6. Write out the EDIF file for this implementation.
 - 7. The EDIF file written by Express and the EDIF file(s) created by the CORE Generator System are required as inputs to the XACTstep M1 Implementation Tools, and should all be located in the same directory when 1 the design is input to M1.



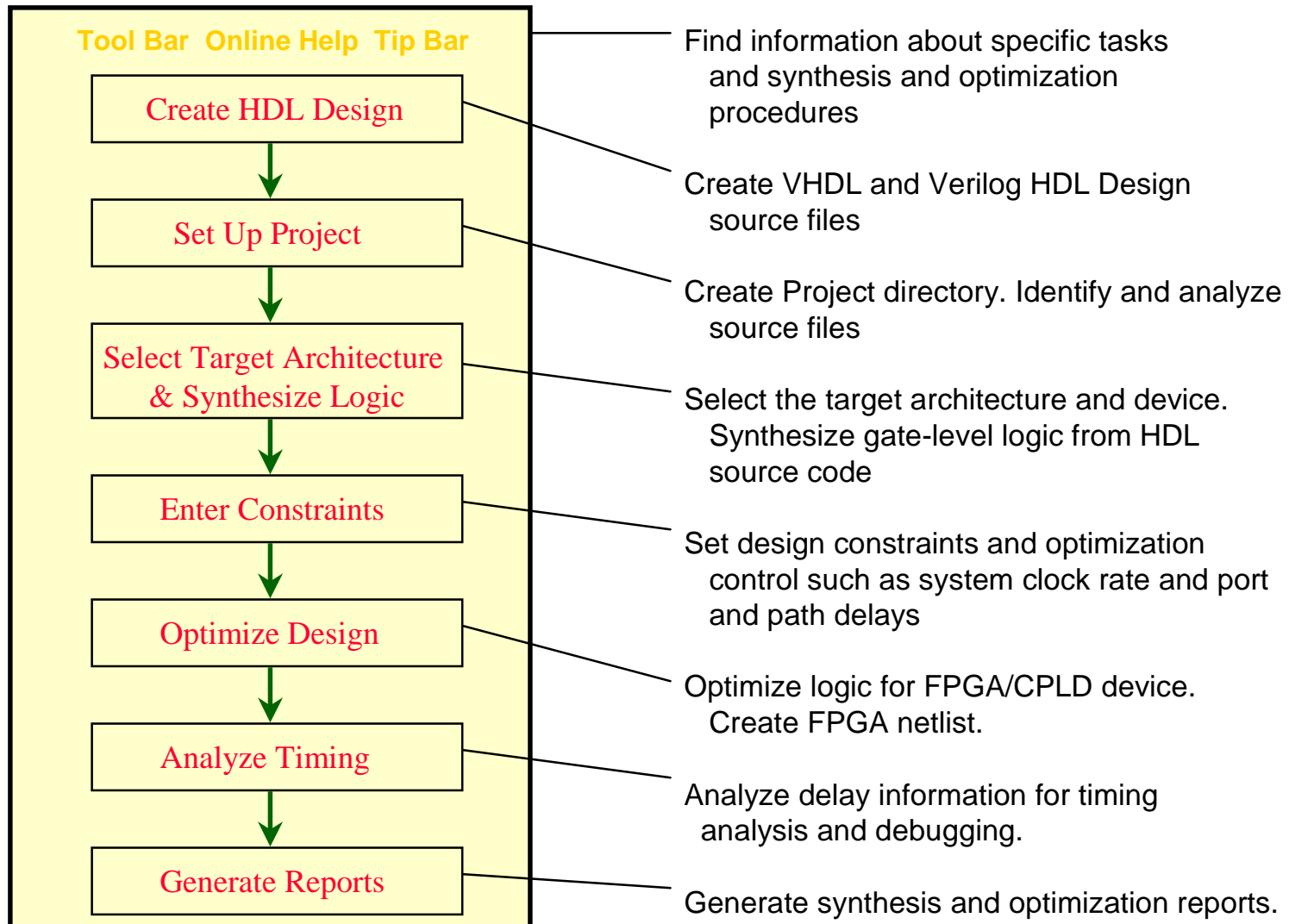
Alliance:

FPGA Express
&
Design Manager

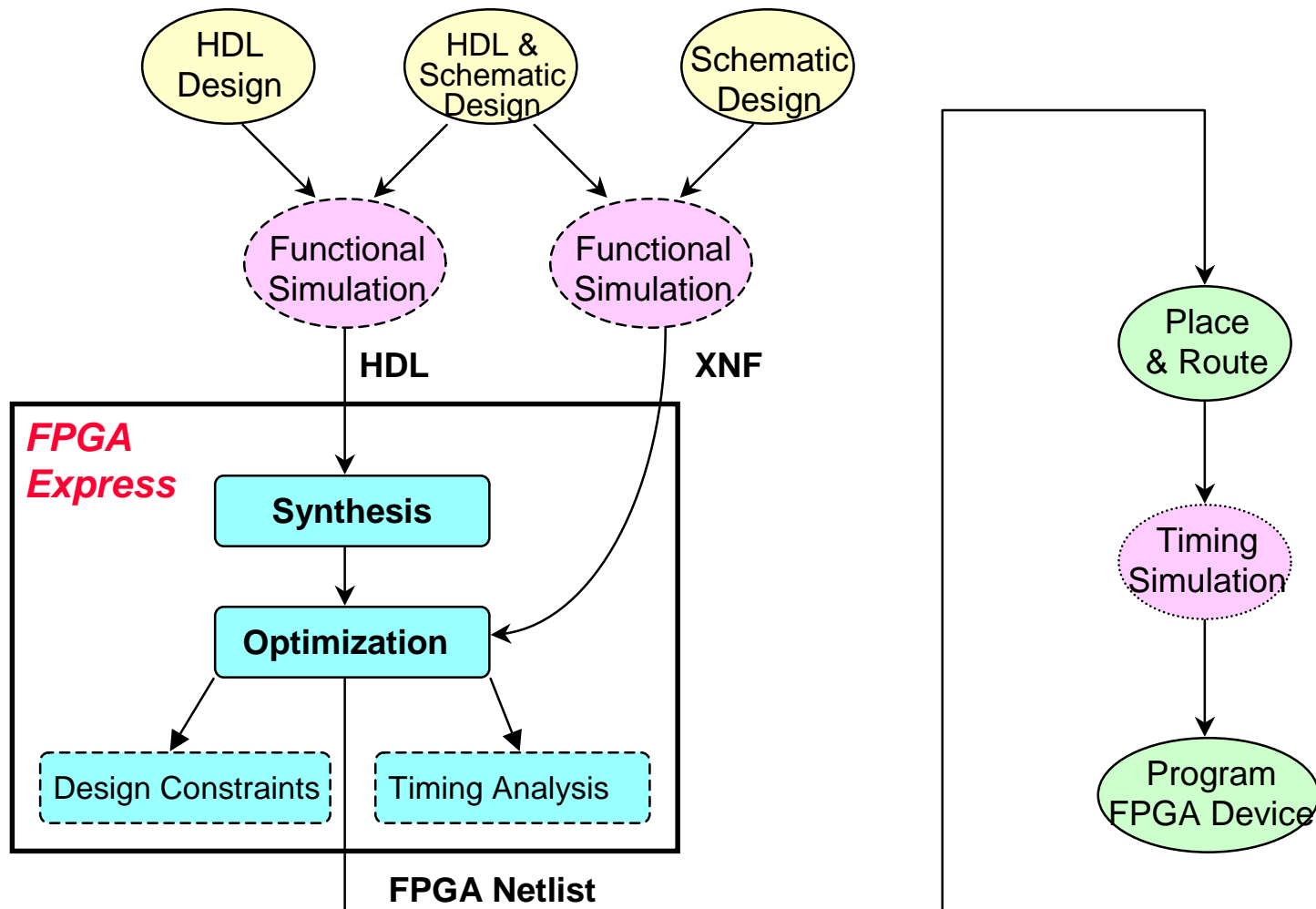
How FPGA Express Fits in the Design Flow



FPGA Express Design Flow

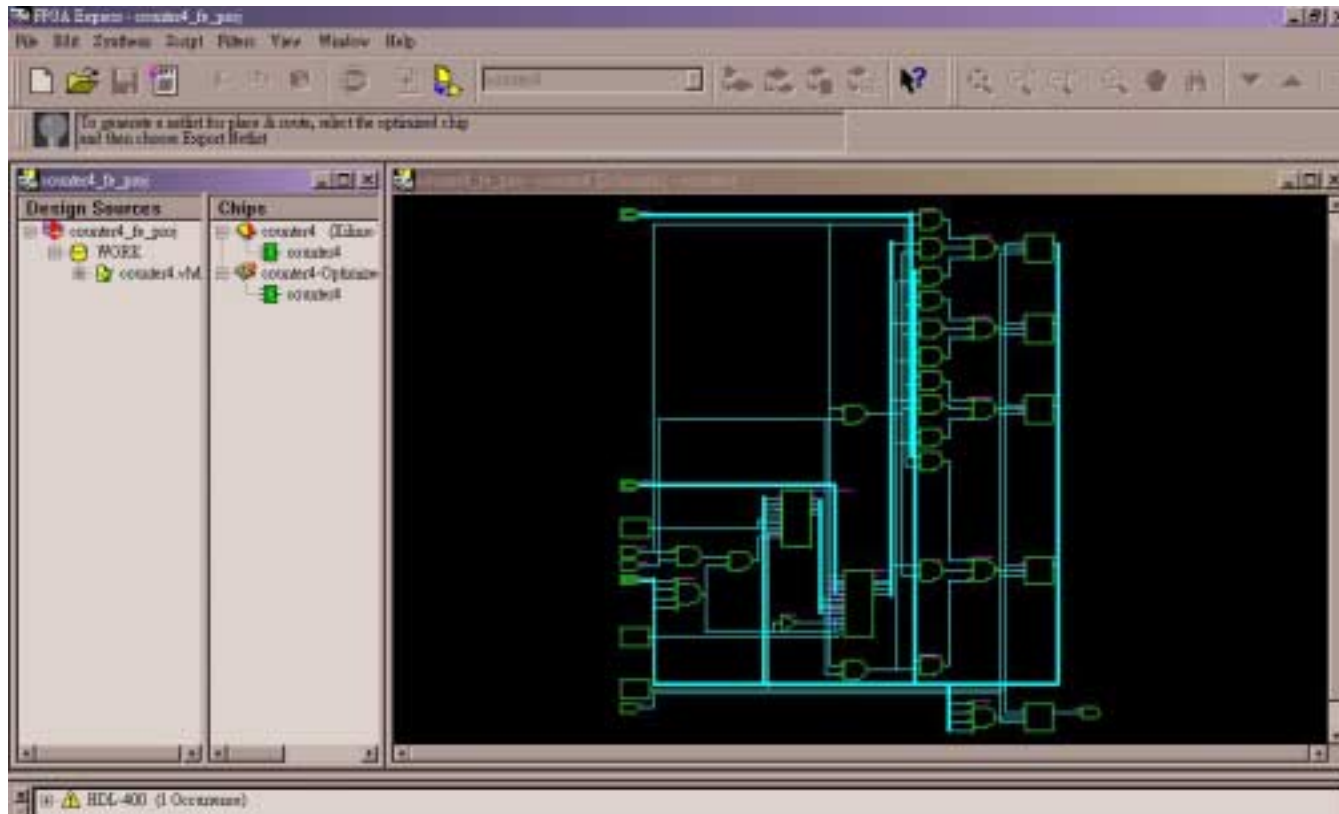


Different Design Environments



FPGA Express

- How to Start FPGA Express
 - 開始 > 程式集 > Xilinx Foundation Series > Accessories > Foundation Express

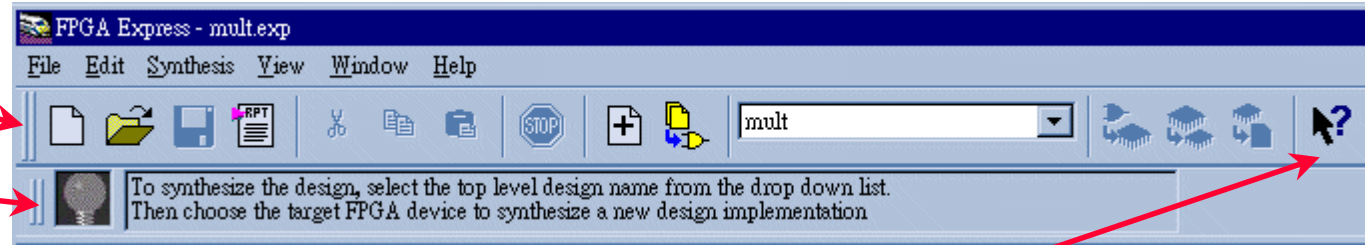


Online Help System

- Tools Bar and Tip Bar

Tool Bar

Tip Bar



- Context-Sensitive Help

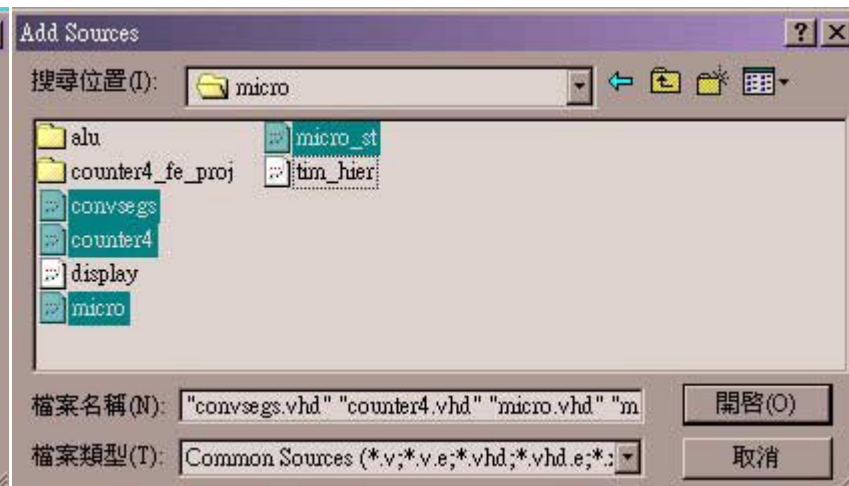
- Shift - F1
- question mark button

- Detailed Online Help

Process for designing an FPGA with FPGA Express I



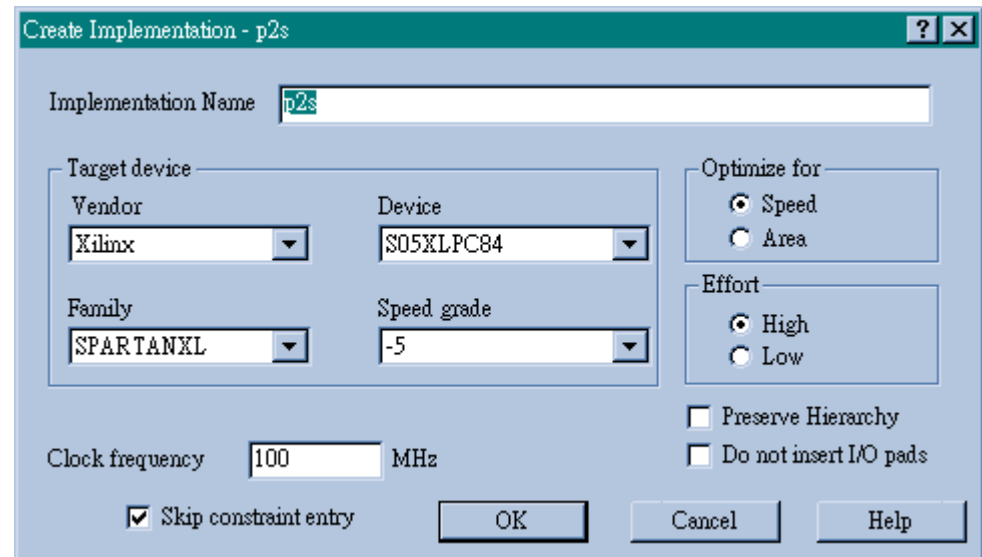
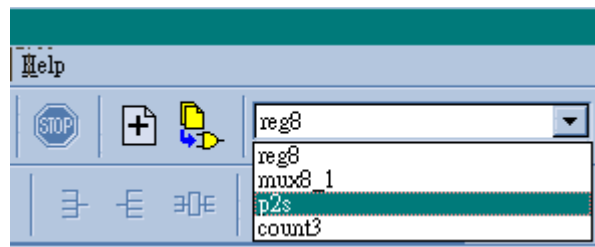
- 1. Create the design
 - Write the HDL source code for the design. FPGA Express process the HDL code to produce an optimized netlist that is ready for FPGA place and route tools.
- 2. (Optional) Verify the design functionally.
 - Use an HDL simulator.
- 3. Set up the design and analyze the source files.
 - File > New Project...



Process for designing an FPGA with FPGA Express II



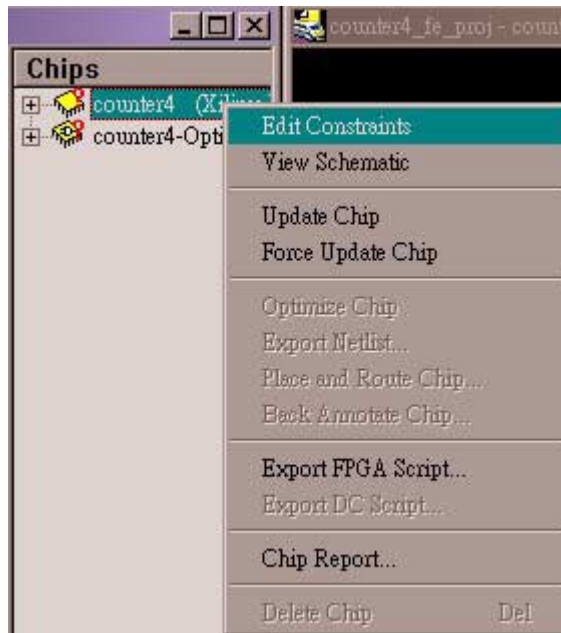
- 4. Synthesize a design
 - select the top level design name from the drop down list.
 - Then choose the target FPGA device to synthesize a new design implementation
 - If you want to enter design constraints, uncheck “Skip constraint entry”



Process for designing an FPGA with FPGA Express III



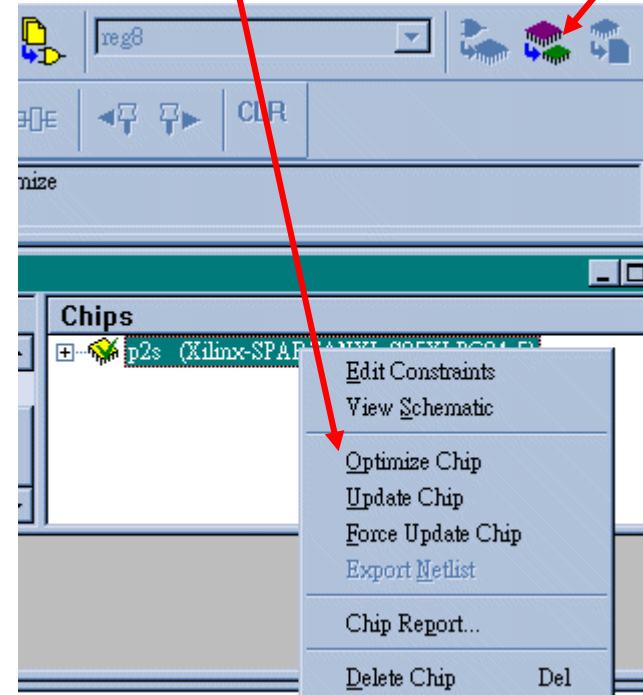
- 5. Enter design constraints
 - Select the design implementation, click the right mouse button, and select Edit Constraints to open the design constraint and optimization-control tables.



Process for designing an FPGA with FPGA Express IV



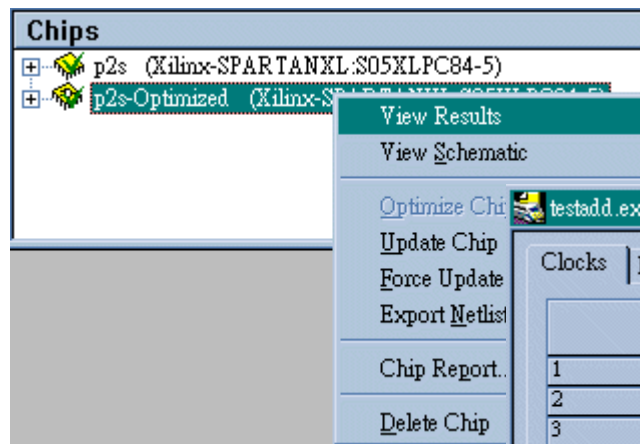
- 6. Optimize the design logic
 - 1. Click the design implementation in the Chips window to select it. Its name is displayed in the top-level design field of the tool bar.
 - 2. Click the right mouse button and select Optimize Chip, or click Optimize Button in the tool bar.



Process for designing an FPGA with FPGA Express V



- 7. Analyze timing
 - To view the results of optimization:
 - 1. Open an optimized implementation by clicking the right mouse button and selecting View Results.
 - 2. Check the Clocks constraint table to see the maximum clock frequencies FPGA *Express* calculated for each of the clocks in the design. Clock frequency violations appear in red.

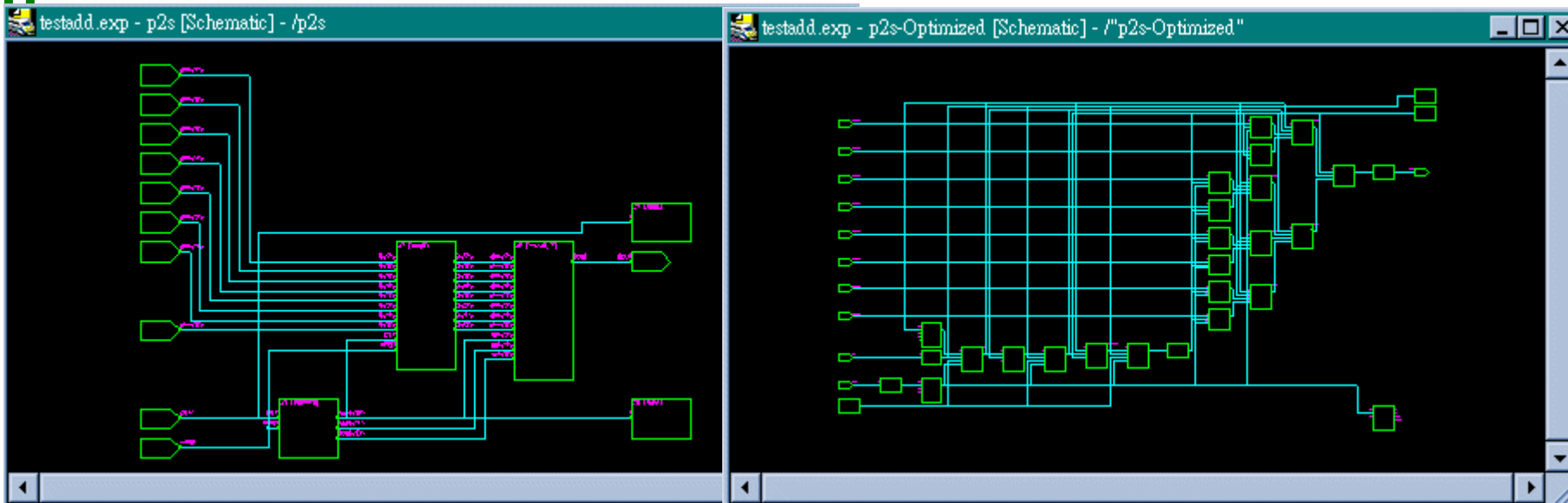


	Name	Clock	Req. Freq (MHz)	Est. Freq (MHz)
1	<default>	100/5		
2	/p2s-Optimized/N75_BUFGeD		100	60
3	/p2s-Optimized/CLK_BUFGeD		100	58

Process for designing an FPGA with FPGA Express VI



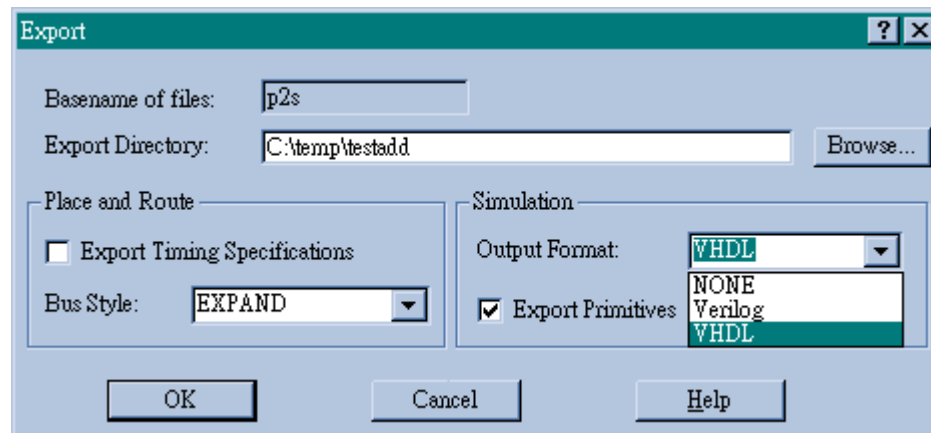
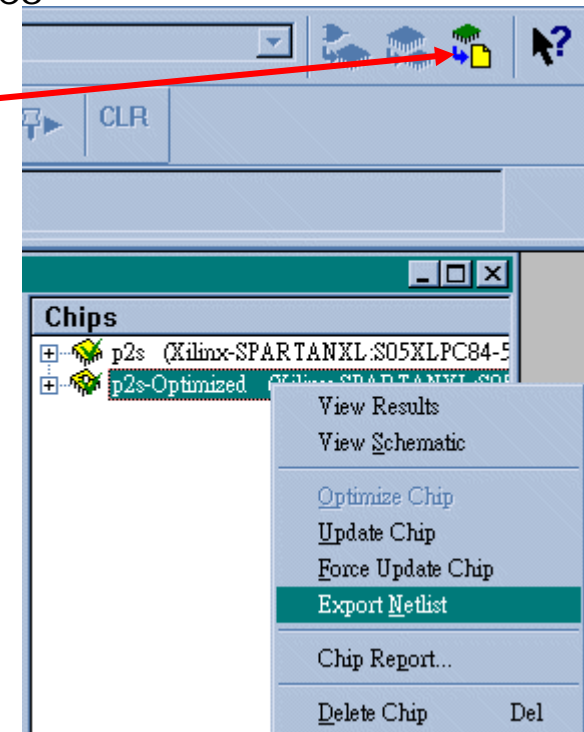
- 8. View schematics
 - View an RTL or Generic Design
 - 1. In the Chips window, right-click on the un-optimized implementation.
 - 2. From the popup menu, select View Schematic.
 - View a Gate-level (or Mapped / Optimized) Design
 - 1. In the Chips window, right-click on the optimized implementation.
 - 2. From the popup menu, select View Schematic.



Process for designing an FPGA with FPGA Express VII



- 9. Generate optimized FPGA netlists and reports.
 - You can generate XNF or EDIF files formatted for immediate place-and-route by the FPGA vendor systems, or
 - You can generate VHDL or Verilog files for functional simulation.
 - Generating Netlist Files(*.xnf) & Simulation Files
 - 1. Select the optimized design implementation.
 - 2. Click Export Netlist button in the tool bar, or click the right mouse button and select Export Netlist.



Process for designing an FPGA with FPGA Express VIII



- Generating a Report
 - 1. Select the project, library, design, or chip icon in the project window.
 - 2. Click the right mouse button and select the report menu, or click on the tool bar. Specify the name and choose the location for the report.
- 10. (Optional) In FPGA Express, analyze timing information to verify acceptable circuit performance.

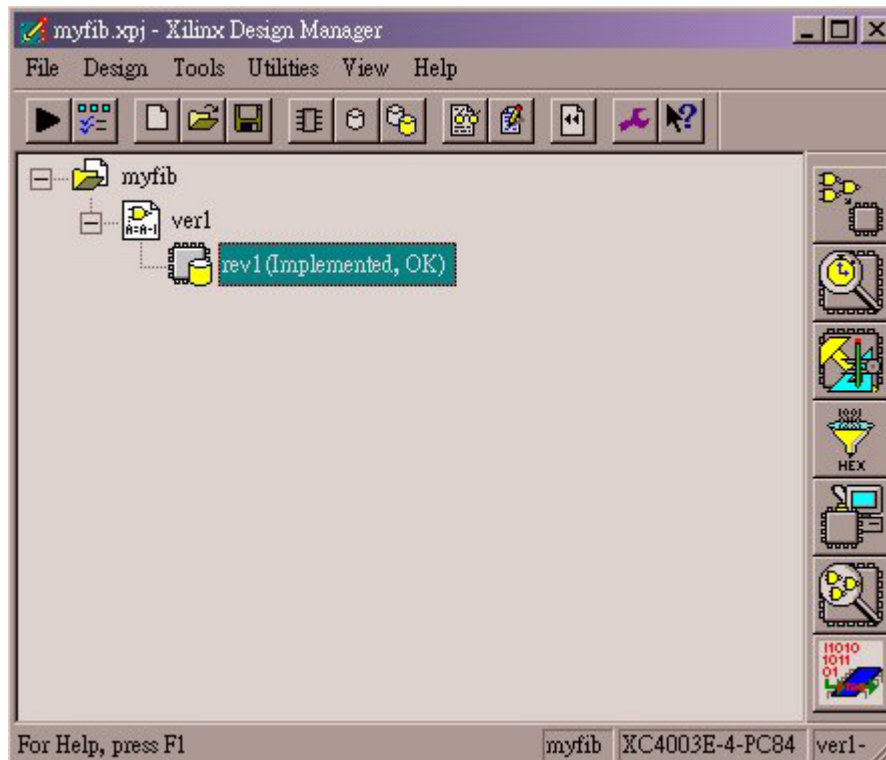
Design Manager

What is Design Manager

- Suite for implementing a design into a Xilinx FPGA or CPLD.
- Provides access to all the tools you need to read a design file from a design entry tool and implement it in a Xilinx device.
- Design Manager Capabilities
 - Organize and manage your design implementation data, including projects, design versions, and implementation revisions
 - Target different devices
 - Manage data for and access to the following tools in the Xilinx Design Manager system. The tools differ for FPGA and CPLD families.
 - Flow Engine (FPGA and CPLD)
 - Timing Analyzer (FPGA and CPLD)
 - Floorplanner (Spartan and XC4000 families only)
 - PROM File Formatter (FPGA)
 - Hardware Debugger (FPGA)
 - EPIC Design Editor (FPGA)
 - JTAG Programmer (CPLD)
 - Generate timing simulation data for external simulation tools
 - Generate configuration data & Program a device

Xilinx Design Manager

- How to Start FPGA Express
 - 開始 > 程式集 > Xilinx Foundation Series > Accessories > Design Manager

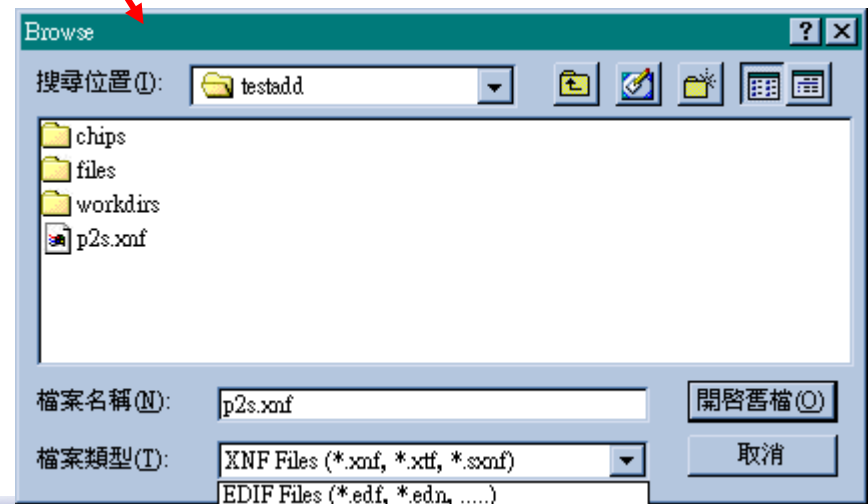
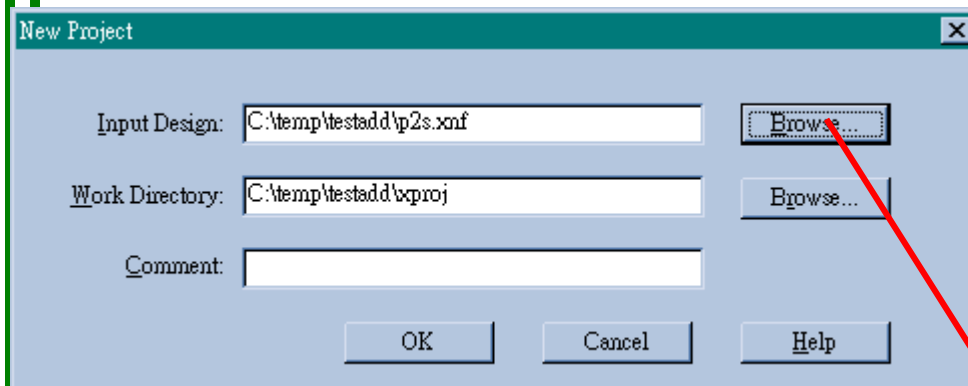


Flow Engine
Timing Analyzer
Floorplaner
PROM File Formatter
Hardware Debugger
FPGA Editor
JTAG Programmer

Process for designing an FPGA with Design Manager I



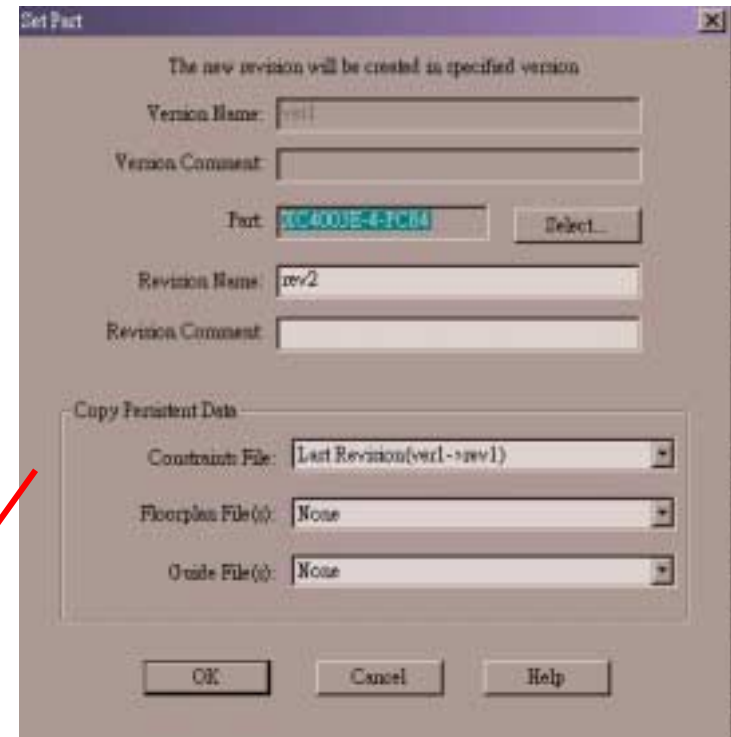
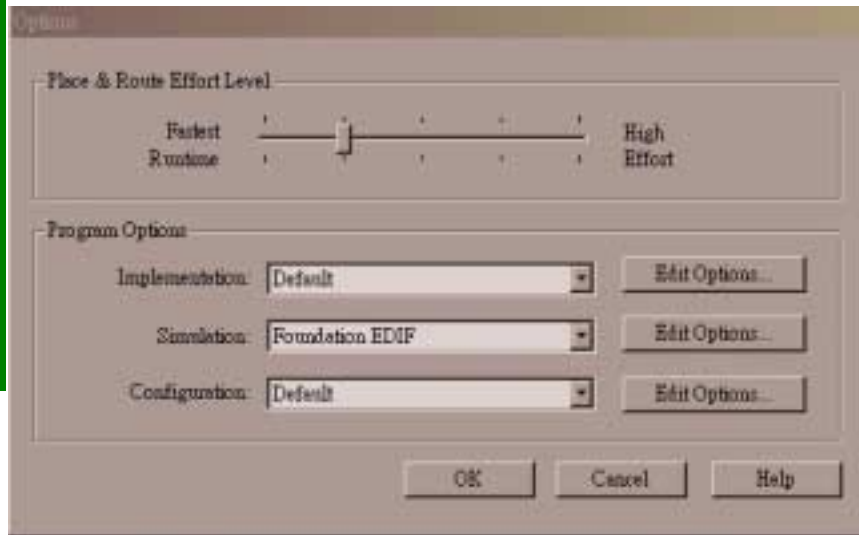
- 1. Set up the Project
 - File > New Project...



Process for designing an FPGA with Design Manager II



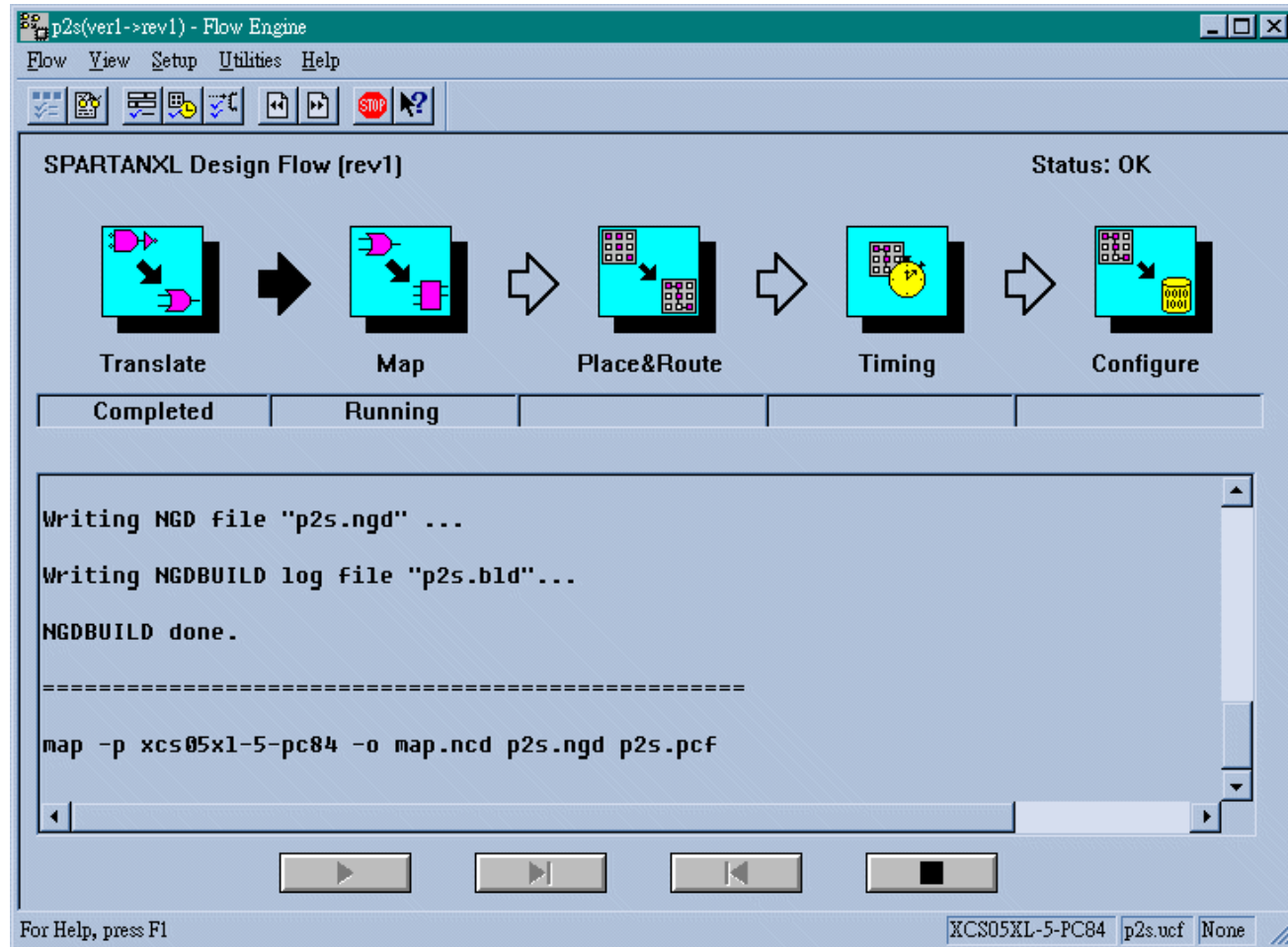
- 2. Implementation
 - Design > Implement...
 - Set up Options...



Process for designing an FPGA with Design Manager III



- Run Flow Engine



Process for designing an FPGA with Design Manager IV



- 3. (Optional) Simulate the design with back-annotated timing delays.
- 4. Program the FPGA device.
 - Hardware Debugger