coregen video

Wrapper files
(next zweite Seite)

Black box synthesis

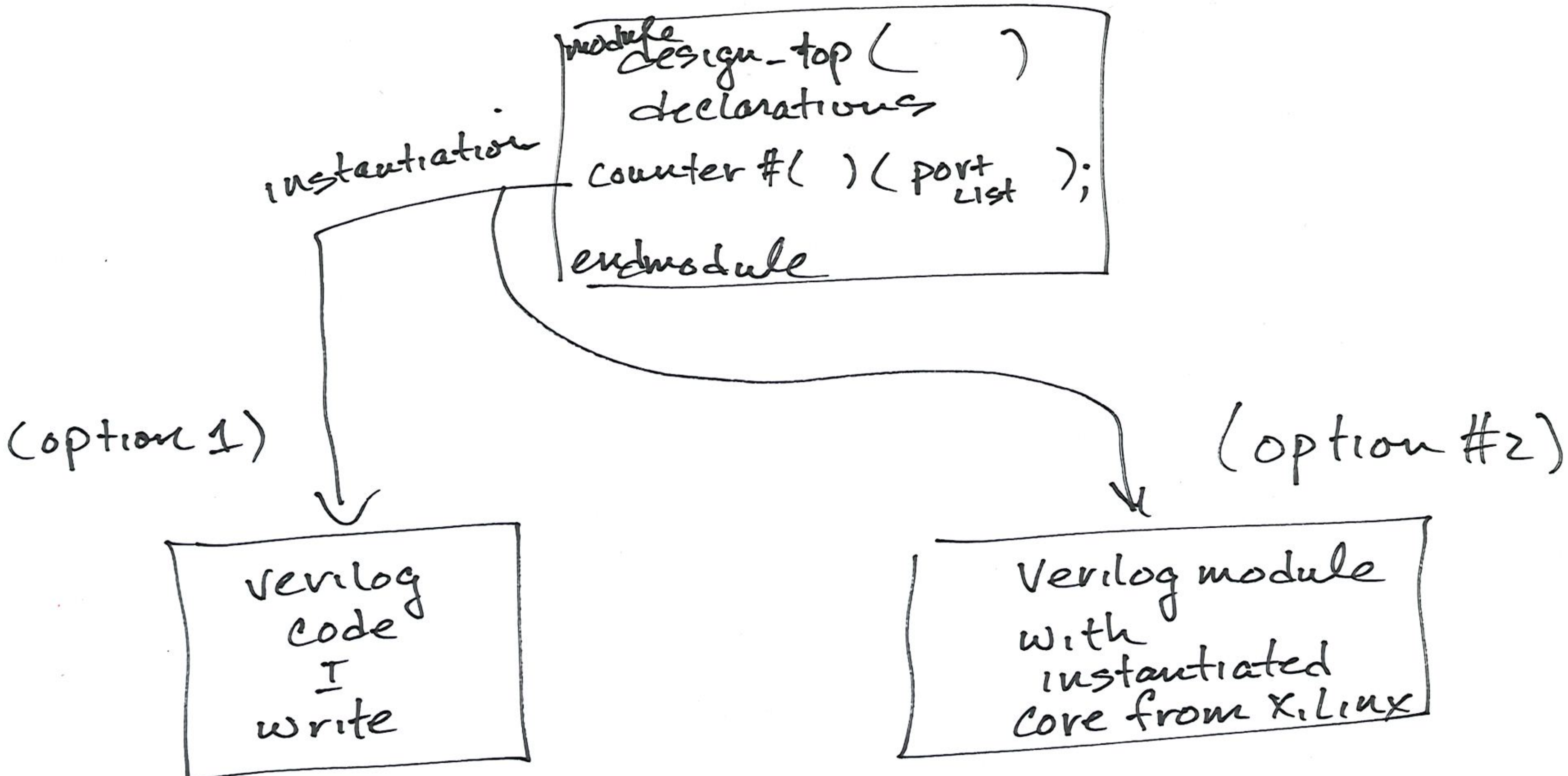
Virtex-5 HDL Libraries

Do BUFG as one ex.

---

now do JTAG (following pages)

Wrapper files



want to simulate using ModelSIM

Modulesim can handle option #1 but not option #2 (modelsim (doesnt have any idea of what an IP core generated by a Designautomation tool does) that's why the wrapper file is needed!

Introduction

The Xilinx® LogiCORE™ IP Binary Counter core provides LUT and single XtremeDSP™ slice counter implementations. The Binary Counter is used to create up counters, down counters, and up/down counters with outputs of up to 256-bits wide. Support is provided for one threshold signal that can be programmed to become active when the counter reaches a user defined count. The upper limit of the count is user programmable and the counter's increment value is user defined. When the counter reaches terminal count or the *count to value,* the next count is zero.

Features

- Drop-in module for Virtex®-7 and Kintex™-7, Virtex-6, Virtex-5, Virtex-4, Spartan®-6, Spartan-3/XA, Spartan-3E/XA, Spartan-3A/3AN/3A DSP/XA FPGAs
- Backwards compatible with version 9.1
- Generates up, down, and up/down counters
- Supports fabric implementation counters ranging from 1 to 256 bits wide
- Supports DSP48 implementation counters ranging from 1 to 36, or 48 bits wide (varies with device family)
- Pipelining added for maximal speed performance
- Predictive detection used for threshold and terminal count detection
- Optional synchronous set and synchronous init capability for legacy fabric implementations
- Optional user programmable threshold outputs
- Optional clock enable and synchronous clear
- Counter increment value is user defined
- User-programmable count limit
- For use with Xilinx CORE Generator™ and Xilinx System Generator for DSP 13.1

LogiCORE IP Facts Table					
Core Specifics					
Supported Device Family(1)	Virtex-7 and Kintex-7 Virtex-6, Virtex-5, Virtex-4, Spartan-6, Spartan-3/XA, Spartan-3E/XA, Spartan-3A/3AN/3A DSP/XA				
Supported User Interfaces	Not Applicable				
	Resources(2)				**Frequency**
Configuration	**LUTs**	**FFs**	**DSP Slices**	**Block RAMs**	**Max. Freq.**
Virtex-5, 18-bit, fabric up/down counter	19	18	0	0	450 MHz
Provided with Core					
Documentation	Product Specification				
Design Files	Netlist				
Example Design	Not Provided				
Test Bench	Not Provided				
Constraints File	Not Applicable				
Simulation Model	VHDL behavioral model in the xilinxcorelib library VHDL UniSim structural model Verilog UniSim structural model				
Tested Design Tools					
Design Entry Tools	CORE Generator tool 13.1 System Generator for DSP 13.1				
Simulation	Mentor Graphics ModelSim 6.6d Cadence Incisive Enterprise Simulator (IES) 10.2 Synopsys VCS and VCS MX 2010.06 ISIM 13.1				
Synthesis Tools	N/A				
Support					
Provided by Xilinx, Inc.					

1. For a complete listing of supported devices, see the release notes for this core.
2. For more complete device performance numbers, see "Performance and Resource Utilization," page 8.

© Copyright 2006, 2009-2011, Xilinx, Inc. XILINX, the Xilinx logo, ISE, Kintex, Spartan, Virtex, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. All other trademarks are the property of their respective owners.

Pinout

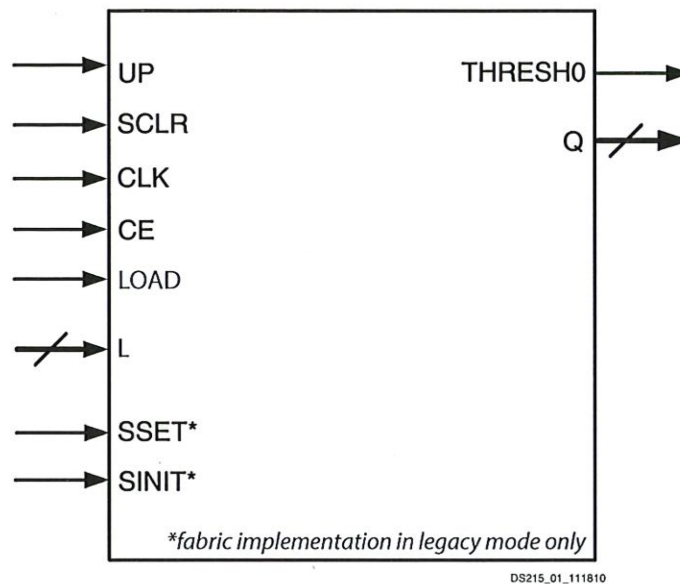


Figure 1: **Core Symbol**

Signal names for the core symbol are shown in Figure 1 and described in Table 1. Note that Figure 1 shows the SSET and SINIT pins which appear only on legacy fabric implementations. The fabric legacy mode has been provided to preserve backwards compatibility but allow performance to be improved. This is where **Implement using** = Fabric, **Latency** = 1 and **FB_Latency** = 0.

Table 1: **Core Signal Pinout**

Signal	Direction	Description
CLK	Input	Rising edge clock signal
UP	Input	Controls the count direction on an up/down counter. Counts up when high, down when low
CE	Input	Active high Clock Enable
SCLR	Input	Synchronous Clear: forces the output to a low state when driven high
THRESH0	Output	User-programmable active high threshold signal
Q[N:0]	Output	Output
L[N:0]	Input	Load data port
LOAD	Input	Load control signal
SSET(1)	Input	Synchronous Set: forces the output to a high state when driven high
SINIT(1)	Input	Synchronous Initialize: forces the outputs to a user defined state when driven high

1. Available only when in fabric legacy mode: **Implement using** = Fabric, **Latency** = 1, and **Feedback Latency** = 0

CORE Generator Graphical User Interface Parameters

The CORE generator GUI parameters for this module are described below:

- **Implement using**: Sets the implementation type to Fabric or DSP48.
- **Output Width**: Specifies the width of the counter.
- **Restrict Count**: When this parameter is true the counter only counts up (or down) to the value specified in the **Final Count Value** parameter. When it is false the counter counts up to the maximum value that can be represented using the specified output width. This option is mutually exclusive with the up/down counter option and with synchronous set controls.
- **Final Count Value**: When **Restrict Count** = true, this parameter specifies the hex representation of the upper limit of the counter.
- **Increment Value**: Specifies in hex the increment value of the counter. When **Restrict Count** is false, the valid range is 1 to $2^{\text{Output Width}} - 1$. When **Restrict Count** is true the valid settings for **Increment Value** are governed by the equation:

  Final Count Value / Increment Value = Integer

  for up counters, and:

  $2^{\text{Output Width}}$ - **Final Count Value / Increment Value** = Integer

  for down counters.
- **Count Mode**: This parameter specifies whether the counter counts up, down, or has its direction specified on the UP pin (up/down).
- **Sync Threshold Output**: When this parameter equals true, the THRESH0 combinatorial output is generated.
- **Threshold Value**: Specifies the value at which the THRESH0 value is activated as a hex value.
- **Loadable**: Activating the LOAD pin (**Loadable** = true) allows the value on the L[N:0] input port to pass through the logic and be loaded into the output register on the next active clock edge. See the section, "Use of LOAD" for more information.
- **Load Sense**: Specifies Active_High or Active_Low LOAD pin.
- **CE**: When set to true, the module is generated with a clock enable input.
- **Power on Reset Init Value**: Specifies, in hex, the value that the output initializes to during power-up reset.
- **Synchronous Clear (SCLR)**: Specifies if an SCLR pin is to be included.
- **SSET**: Specifies if an SSET pin is to be included. SSET pin is not valid in DSP48 implementations. See **Sync Set and Clear (Reset) Priority** for SCLR/SSET priorities.
- **SINIT**: Specifies if an SINIT pin is to be included which, when asserted, synchronously sets the output value to the value defined by **Init Value**. Note that if SINIT is present, then neither SSET nor SCLR may be present. SINIT pin is not valid in DSP48 implementations.
- **Init Value**: Specifies, in hex, the value that the output initializes to when SINIT is asserted. The Init Value is ignored if **SINIT** is false.
- **Synchronous Controls and Clock Enable (CE) Priority**: This parameter controls whether or not the SCLR, (and if fabric: SSET and SINIT) input is qualified by CE. When set to Sync_Overrides_CE, the synchronous control overrides the CE signal. When set to CE_Overrides_Sync, SCLR has an effect only when CE is high. Note that on the fabric primitives, the SCLR and SSET controls override CE, so choosing CE_Overrides_Sync generally results in extra logic.
- **Sync Set and Clear (Reset) Priority**: Controls the relative priority of SCLR and SSET. When set to Reset_Overrides_Set, SCLR overrides SSET. The default is Reset_Overrides_Set, as this is the way the primitives are arranged. Making SSET take priority requires extra logic.

- **Latency Configuration**: Automatic or Manual; Automatic sets optimal latency for maximum speed; Manual allows user to set Latency to one of the allowed values.
- **Latency**: Value used for latency when Latency Configuration is set to Manual. See the section, "Pipelined Operation" for more information.
- **Feedback Latency Configuration**: Automatic or Manual; Automatic sets optimal feedback latency for maximum speed; Manual allows user to set Feedback Latency to one of the allowed values.
- **Feedback Latency**: Value used for latency when **Feedback Latency Configuration** is set to Manual. See the section, "Pipelined Operation" for more information.

Table 2 is a cross-reference table from the GUI parameters listed above to the XCO parameter names in the XCO file

Table 2: **CORE Generator GUI and XCO Parameters**

GUI Name	Default Value	Valid Range	XCO Parameter
Component Name	c_counter_binary_v11_0		Component_Name
Implement using	Fabric		Implementation
Output Width	16	1 to 256 (fabric) 1 to 36 or 48 (DSP48)	Output_Width
Increment Value[1]	1	unrestricted: 1 to $2^{(\textbf{Output Width})}$- 1	Increment_Value
Latency Configuration	Manual		Latency_Configuration
Latency	1	1 to 32	Latency
Feedback Latency Configuration	Manual		Fb_Latency_Configuration
Feedback Latency	0	0 to 4	Fb_Latency
Clock Enable	false		CE
Synchronous Clear	false		SCLR
Synchronous Set	false		SSET
Synchronous Init	false		SINIT
Loadable	false		Load
Load Sense	Active_High		Load_Sense
Restrict Count	false		Restrict_Count
Final Count Value	1	1 to $2^{(\textbf{Output Width})}$- 2	Final_Count_Value
Sync Threshold Output	Active_Low		Sync_Threshold_Output
Threshold Value	1	restricted: 0 to (**Final Count Value**) unrestricted: 0 to $2^{(\textbf{Output Width})}$- 1	Threshold_Value
Count Mode	UP	UP, DOWN, UPDOWN	Count_Mode
Sync Set and Clear (Reset) Priority	Reset_Overrides_Set		Sync_Ctrl_Priority
Synchronous Controls and Clock Enable (CE) Priority	Sync_Overrides_CE		Sync_CE_Priority
Power on Reset Init Value	0	0 to $2^{(\textbf{Output Width})}$- 1	AINIT_Value
Sync INIT value	0	restricted: 0 to (**Final Count Value**) unrestricted: 0 to $2^{(\textbf{Output Width})}$- 1	SINIT_Value

1. See the list above for an explanation of **Increment Value** range when **Restrict Count** is true.

Core Use through CORE Generator

The CORE Generator GUI performs error-checking on all input parameters. Resource estimation and latency information is also available.

Several files are produced when a core is generated, and customized instantiation templates for Verilog and VHDL design flows are provided in the .veo and .vho files, respectively. For detailed instructions, see the CORE Generator software documentation.

Simulation Models

The core has a number of options for simulation models:

- VHDL behavioral model in the xilinxcorelib library
- VHDL UniSim structural model
- Verilog UniSim structural model

Xilinx recommends that simulations utilizing UniSim-based structural models are run using a resolution of 1 ps. Some Xilinx library components require a 1 ps resolution to work properly in either functional or timing simulation. The UniSim-based structural models might produce incorrect results if simulation with a resolution other than 1 ps. See the "Register Transfer Level (RTL) Simulation Using Xilinx Libraries" section in *Synthesis and Simulation Design Guide* for more information. This document is part of the ISE® Software Manuals set available at www.xilinx.com/support/software_manuals.htm.

Core Use through System Generator

The Binary Counter core is available through Xilinx System Generator for DSP, a design tool that enables the use of the model-based design environment Simulink® software for FPGA design. The Binary Counter core is one of the DSP building blocks provided in the Xilinx DSP blockset for Simulink. The Binary Counter core can be found in the Xilinx Blockset in the Math section. The block is called "Counter". See the System Generator User Manual for more information.

Migrating to Binary Counter v11.0 from Earlier Versions

Updating from Binary Counter v9.0 and later

The CORE Generator core update feature can be used to update an existing Binary Counter XCO file to version 11.0 of the core. The core can then be regenerated to create a new netlist. See the CORE Generator documentation for more information on this feature.

Updating from Versions Prior to Binary Counter v9.0

It is not currently possible to automatically update versions of the Binary Counter core prior to v9.0. Xilinx recommends that customers use the Binary Counter v11.0 GUI to customize a new core. Note that some features and configurations may be unavailable in Binary Counter v11.0. Also, some port names may differ between versions.

Priorities of Input Signals

- **SCLR/SSET**. The priority of SCLR versus SSET can be configured using the **Sync Set and Clear (Reset) Priority** parameter as described above.
- **LOAD**. The synchronous controls (SCLR, SSET, SINIT) take priority over LOAD.

Note that if SCLR, SSET, SINIT or LOAD are affected by CE (specified in the **Synchronous Controls and Clock Enable (CE) Priority** parameter), a low CE value causes these signals to be ignored. For example, if SCLR is affected by CE, then with a low CE, LOAD appears to override SCLR, contrary to what is presented above. However, because SCLR has no effect when CE is low, this is the correct effect.

Discussion of Restricted Counters

The restricted counter option is implemented using an equality test rather than a greater-than-or-equal-to test. This means that if the counter somehow manages to skip the **Final Count Value** value, it keeps going. Therefore, there are restrictions on allowable parameters for restricted counters:

- **Count Mode** cannot be UPDOWN.
- **SSET** must be false.

Additionally, there are restrictions added by pipelining as discussed in "Pipelined Operation" and there are further restrictions that differ for up counters and down counters.

Up Counters

Restricted up counters count up by **Increment Value** until Q = **Final Count Value**. The counter resets to 0 during the clock cycle after Q = **Final Count Value.**

There are two basic restrictions:

1. **Final Count Value** must be an integer multiple of **Increment Value**
2. **Increment Value** must be less than or equal to **Final Count Value**

In addition, the following formulae must be satisfied:

$$\langle value \rangle / (\text{IncrementValue}) = \text{Integer}$$

$$\langle value \rangle \leq \text{Final Count Value}$$

where **<value>** is any of the following:

- **Init Value**, if SINIT is used
- **AINIT Value** for power-on reset
- Any value loaded on the L data port.

Down Counters

Restricted down counters count down by **Increment Value** until Q = **Final Count Value**. The counter resets to 0 during the clock cycle after Q = **Final Count Value** and the counter continues counting down (wrapping around).

There are two basic restrictions:

1. $2^{\text{Output Width}} - \text{Final Count Value}$ must be an integer multiple of **Increment Value**
2. **Increment Value** must be less than or equal to $2^{\text{Output Width}} - \text{Final Count Value}$

In addition, the following formulae must be satisfied:

$$(2^{\textbf{Output Width}} - \textbf{<value>}) / \textbf{Increment Value} = \text{Integer}$$

$$\text{either } \textbf{<value>} \geq \textbf{Final Count Value} \text{ or } \textbf{<value>} = 0$$

where <**value**> is any of the following:

- **Init Value**, if SINIT is used
- **AINIT Value** for power-on reset
- Any value loaded on the L data port.

Use of LOAD

The Counter core can check on instantiation for sensible **Init Value** and power-on reset value, but it cannot check the data loaded on the L data port. Because of this, erroneous values loaded will cause unexpected behavior in the counter. For example, if a counter is given **Final Count Value** = 8 and **Increment Value** = 2, loading in 3 will cause it to count the odd numbers and completely miss the limit value.

Load Support

The load operation in an XtremeDSP slice requires opmode control. The opmode control affects some of the same bits that are controlled to allow the terminal count reset feature. For the bits affected, external gating is required. This either impacts performance or requires an additional layer of latency in the feedback path. An extra layer of latency in the load path requires external registers for the D port of DSP48A (Spartan-3A DSP/Spartan-6) and on the C register used for the **Increment Value**. Despite being a constant, the **Increment Value** is registered so that the recover from SCLR latency matches the LOAD latency. The latency of the counter for the XtremeDSP slice implementation is therefore width dependant and varies as a function of **Restrict Count, Loadable, Load Sense, Output Width** and the device family chosen.

The **Load Enable** in v9.0 and earlier has been deprecated. LOAD is always subject to CE (if present).

In Virtex-4, the A:B concatenated port width is 36 bits; hence, the output width is also limited to 36 bits when LOAD is used.

Moreover, the detection of the terminal count has been further improved in version 11.0, but to maintain backward-compatible behavior, the improved detection cannot be used when the counter is loadable.

Therefore it is strongly recommended that LOAD is not used with restricted counters; if such functionality is required, use external logic to create a greater-than-or-equal-to test rather than an equal-to test, or make sure the counter is simple (count by 1) and that a value is never loaded beyond **Final Count Value**.

Pipelined Operation

Pipelining the terminal count detection requires that the actual value detected is the terminal count value minus some multiple of the **Increment Value** value where the multiple is determined by the full cycle latency. This adds further restrictions to the valid combinations of **Increment Value** and **Final Count Value**. A run-time assertion (warning) flags if a value is loaded which would cause the actual terminal count value to be missed.

To allow for high performance, four new parameters have been added to the core: **Latency Configuration, Latency, Feedback Latency Configuration,** and **Feedback Latency.** The first two describe the number of cycles the core takes to recover from SCLR or from a LOAD value since it is the number of registers in the forward data path. When **Latency Configuration** is set to Manual, **Latency** can be set to a specific number for specific latency. By setting **Latency Configuration** to Automatic, the latency for maximal performance is calculated internally and used in

place of **Latency.** Maximal performance is defined as an operating frequency greater than or equal to the nominal operating speed of a fully pipelined XtremeDSP slice. For Virtex4-10 this is 400MHz. For Virtex5-1 this is 450MHz. For Spartan-3A DSP this is 250MHz. This is achieved by splitting the carry-chain of the main count operation into splices and pipelining. The number of splices required is a function of the counter bit width and the family in question. **Feedback Latency Configuration** and **Feedback Latency** refer to the latency in the terminal count feedback circuit and hence apply only to restricted counters. The total amount of latency gives the number of cycles by which the terminal count detection must predict the terminal count value. When **Feedback Latency Configuration** is set to Manual, **Feedback Latency** can be set to a specific number for specific feedback latency. By setting **Feedback Latency Configuration** to Automatic, the feedback latency for maximal performance will be calculated internally and used in place of **Feedback Latency.**

Performance and Resource Utilization

Tables 3 to 6 provide counter performance and resource usage for a number of different configurations.

The maximum clock frequency results were obtained by double-registering input and output ports to reduce dependence on I/O placement. The inner level of registers used a separate clock signal to measure the path from the input registers to the first output register through the core.

The resource usage results do not include the above "characterization wrapper" registers and represent the true logic used by the core. LUT counts include SRL16s or SRL32s (according to device family).

The map options used were: "`map -pr b -ol high`."

The par options used were: "`par -ol high`."

Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification.

The maximum achievable clock frequency and the resource counts may also be affected by other tool options, additional logic in the FPGA device, using a different version of Xilinx tools, and other factors. The Xilinx Xplorer™ script can be used to find the optimal settings.

All characterization was done using the following parameter settings unless otherwise noted:

- **Count Mode** = UPDOWN
- **Latency Configuration** = Automatic
- **Feedback Latency Configuration** = Automatic
- **Increment Value** = 1
- **Final Count Value** = 4B2A189 (fabric) and AF0C3 (DSP48)
- all else = default unless otherwise noted

Table 3: **Fabric Counter: Virtex-5 (Part = XC5VSX50T-1)**

Description				Load[1]			Restrict Count[2]		
	Sml	Med	Lrg	Sml	Med	Lrg	Sml	Med	Lrg
Output Width	18	47	200	18	47	200	18	47	200
Max Clock Frequency (MHz)	450	433	353	444	407	316	452	450	378
LUT6-FF pairs	19	92	437	19	137	770	25	129	462
LUTs	19	71	389	19	71	529	22	76	418
Flip-flops	18	86	419	18	132	751	25	128	454
DSP48Es	0	0	0	0	0	0	0	0	0

1.Load test cases are for a counter with the following parameter values: **Loadable** = true, **Load Sense** = Active_High.
2.Restrict Count test cases are for a counter with the following parameter values: **Restrict Count** = true, **Count Mode** = UP.

Table 4: **XtremeDSP Slice Counter: Virtex-5 (Part = XC5VSX50T-1)**

Description			Load High[1]		Load Low	
	Sml	Med	Sml	Med	Sml	Med
Output Width	35	48	35	48	35	48
Max Clock Frequency (MHz)	450	450	450	450	450	450
LUT6-FF pairs	7	43	45	94	45	94
LUTs	7	10	9	12	9	12
Flip-flops	7	43	45	94	45	94
DSP48Es	1	1	1	1	1	1

1.Load test cases are for a counter with the following parameter values: **Loadable** = true. ALL DSP48 tests have **Restrict Count** = true.

Table 5: **Fabric Counter: Spartan-3A DSP (Part = XC3SD3400A-4)**

Description				Load[1]			Restrict Count[2]		
	Sml	Med	Lrg	Sml	Med	Lrg	Sml	Med	Lrg
Output Width	18	47	200	18	47	200	18	47	200
Max Clock Frequency (MHz)	248	205	193	217	197	187	261	243	230
LUTs	49	94	512	88	165	937	56	104	554
Flip-flops	36	102	464	54	172	840	57	123	549
DSP48As	0	0	0	0	0	0	0	0	0

1.Load test cases are for a counter with the following parameter values: **Load** = true, **Load Sense** = Active_High.
2.Restrict Count test cases are for a counter with the following parameter values: **Restrict Count** = true, **Count Mode** = UP.

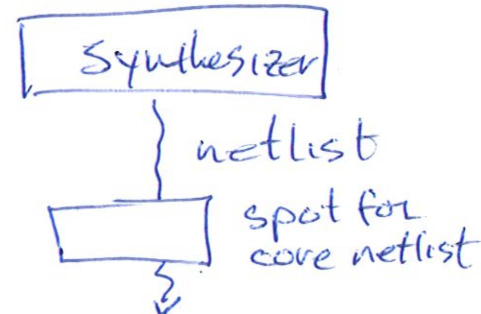
don't show this pg on projector

Suppose the myfifo IP was generated by a tool (e.g., the Xilinx coregen tool). Corgen uses a GUI to allow the designer to customize the IP. It then generates 3 files:

- a wrapper file for simulation (*not synthesizable*)
- a *.VEO file that shows how to instantiate the core (cut & paste)
- a *.NGC file which is a netlist file

So how is the synthesizer supposed to incorporate this core into the netlist it outputs????

REMEBER: synthesizers read *.v or *.vhd file—not netlist files. PAR tools *do* understand *.NGC file formats. Put another way, the synthesizer doesn't use *.NGC files but PAR tools do.)



Synthesizing black boxes

Black boxes are "placeholders" in the synthesis flow.

Black box modules are defined only in terms of their I/O—i.e., they contain only the port declarations and no functionality. *But you need to add a synthesis attribute to tell the synthesizer you are defining a black box module.* (more on this shortly)

Notes:

1) Since the synthesizer will only know the I/O and not the functionality of a black box module, there is no implementation of the module.

2) For the same reason, the synthesizer can't optimize it w.r.t. timing or area. (doesn't know the functionality)

3) But since it is a black box, with defined I/O, the design hierarchy is preserved.

IP generation tools create IP as a “hard macro” in vendor specific file formats.

FPGA	IP GenerationTool	File Format
Altera	MegaWizard	.edif, .edf, .adl
Actel	SmartGen	.vqm
Lattice	IPExpress	.edif, .ngo
Xilinx	CoreGen	.ngc, .edif

These file formats are understood by PAR tools—*not synthesizers!* (They only understand .v or .vhd file formats.)

So how does the IP core get into the design?

An observation

Coregen (Xilinx) or IPexpress (Lattice) provide a GUI to make customizing a core easy. They output then 3 files

- A wrapper file
- a VEO file showing a template for instantiation
- an NGC (or sometimes EDIF) netlist file

The wrapper file is needed so you can simulate a design than instantiates the core.

Synthesizing a black box tells the synthesizer a module exists that is not functionally described, but room needs to be made for it in the netlist generated. The I/O is described so the design hierarchy can be preserved.

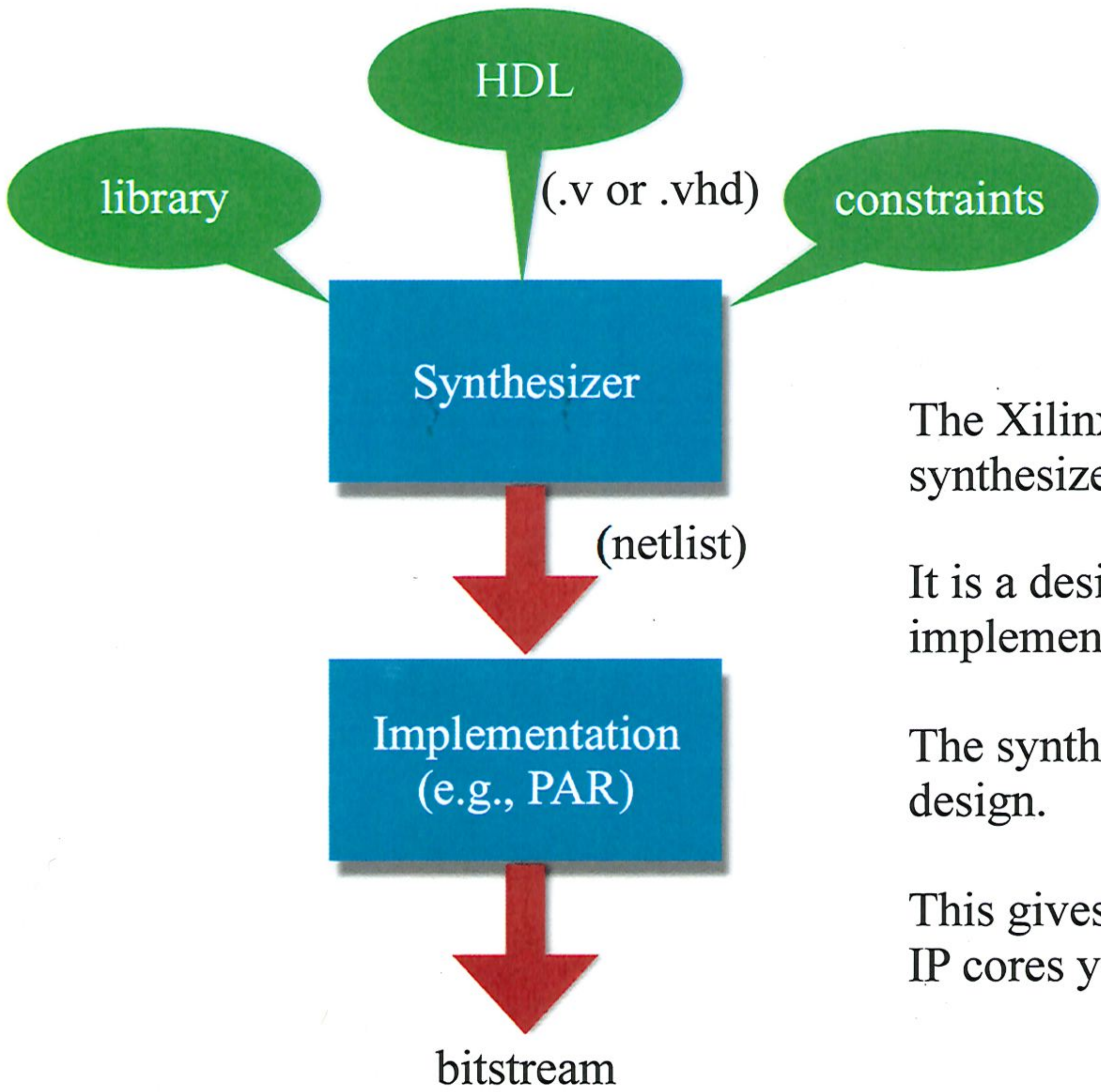
The NGC/EDIF file for the core is sent to the PAR tool to get placed inside the FPGA.

Important

You don't synthesize wrapper files!!!

use it only to do simulation
you synthesis the module that has the instantiated core

FPGA IMPLEMENTATION



The Xilinx Vilvado is more than just a synthesizer.

It is a design suite that does the entire design implementation.

The synthesizer only generates the netlist of the design.

This gives rise to an issue related to FPGA vendor IP cores you might use.

We create a “myfifo” as a black box module in the following way:

```
module myfifo (
    output  [15:0] dout,
    output         empty,
    output         full,
    input          clk,
    input   [15:0] din,
    input          rd_en,
    input          rst,
    input          wr_en)  /* synthesis syn_black_box */ ;
endmodule
```

Notes:

1) notice only the ports are declared; no functionality
2) synthesizer informed this is a black box by the synthesis attribute
3) If you do not create a black box module, this will usually generate a synthesizer warning.

Consider the following module where a counter output is pushed into a FIFO:

```
module fifotop(
  output [15:0] oDat,
  output        oEmpty,
  input         iClk, iReset,
  input         iRead,
  input         iPushReq);

  reg [15:0]     counter;
  wire          wr_en, full;

  assign wr_en  = iPushReq & !full;

  always @(posedge iClk)
    if (!iReset)  counter <= 0;
    else  counter <= counter + 1;

  myfifo  U22 (
      .clk(iClk),
      .din (counter),
      .rd_en(iRead),
      .rst(IReset),
      .wr_en(wr_en),
      .dout(oDat),
      .empty(oEmpty),
      .full(full));
endmodule
```

Vivado uses the black_box attribute. You can apply it directly to the module definition.

verilog

```verilog
(* black_box *) module my_hardware_core (
    input clk,
    input rst,
    input [7:0] data_in,
    output [7:0] data_out
);
    // Leave the body empty or include just the IOs
endmodule
```

```
## This is the constraints file for the Digital Scoreboard demo.  The
design is targeted
## to the Digilent Nexys4 DDR board.
## Modified by Roy Kravitz 06-Ict-2015
##
## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according
to the top level signal names in the project

## Clock signal
 set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 }
[get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports {clk}];


##Switches

# set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 }
[get_ports { sw[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
# set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 }
[get_ports { sw[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
# set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 }
[get_ports { sw[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
# set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 }
[get_ports { sw[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
# set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 }
[get_ports { sw[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
# set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 }
[get_ports { sw[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
# set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 }
[get_ports { sw[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
# set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 }
[get_ports { sw[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
# set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS33 }
[get_ports { sw[8] }]; #IO_L24N_T3_34 Sch=sw[8]
# set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 }
[get_ports { sw[9] }]; #IO_25_34 Sch=sw[9]
# set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 }
[get_ports { sw[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
# set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 }
[get_ports { sw[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
# set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 }
[get_ports { sw[12] }]; #IO_L24P_T3_35 Sch=sw[12]
# set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 }
[get_ports { sw[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
# set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 }
[get_ports { sw[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
```

```
# set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 }
[get_ports { sw[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]


## LEDs

# set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 }
[get_ports { led[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
# set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 }
[get_ports { led[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
# set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 }
[get_ports { led[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
# set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 }
[get_ports { led[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
# set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 }
[get_ports { led[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
# set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 }
[get_ports { led[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
# set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 }
[get_ports { led[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
# set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 }
[get_ports { led[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
# set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 }
[get_ports { led[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
# set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 }
[get_ports { led[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
# set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 }
[get_ports { led[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
# set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 }
[get_ports { led[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
# set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 }
[get_ports { led[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
# set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 }
[get_ports { led[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
# set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 }
[get_ports { led[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
# set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 }
[get_ports { led[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

##RGB LEDs

# set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 }
[get_ports { RGB1_Blue }]; #IO_L5P_T0_D06_14 Sch=led16_b
# set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 }
[get_ports { RGB1_Green }]; #IO_L10P_T1_D14_14 Sch=led16_g
# set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 }
[get_ports { RGB1_Red }]; #IO_L11P_T1_SRCC_14 Sch=led16_r
# set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 }
[get_ports { RGB2_Blue }]; #IO_L15N_T2_DQS_ADV_B_15 Sch=led17_b
# set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 }
```

Jack

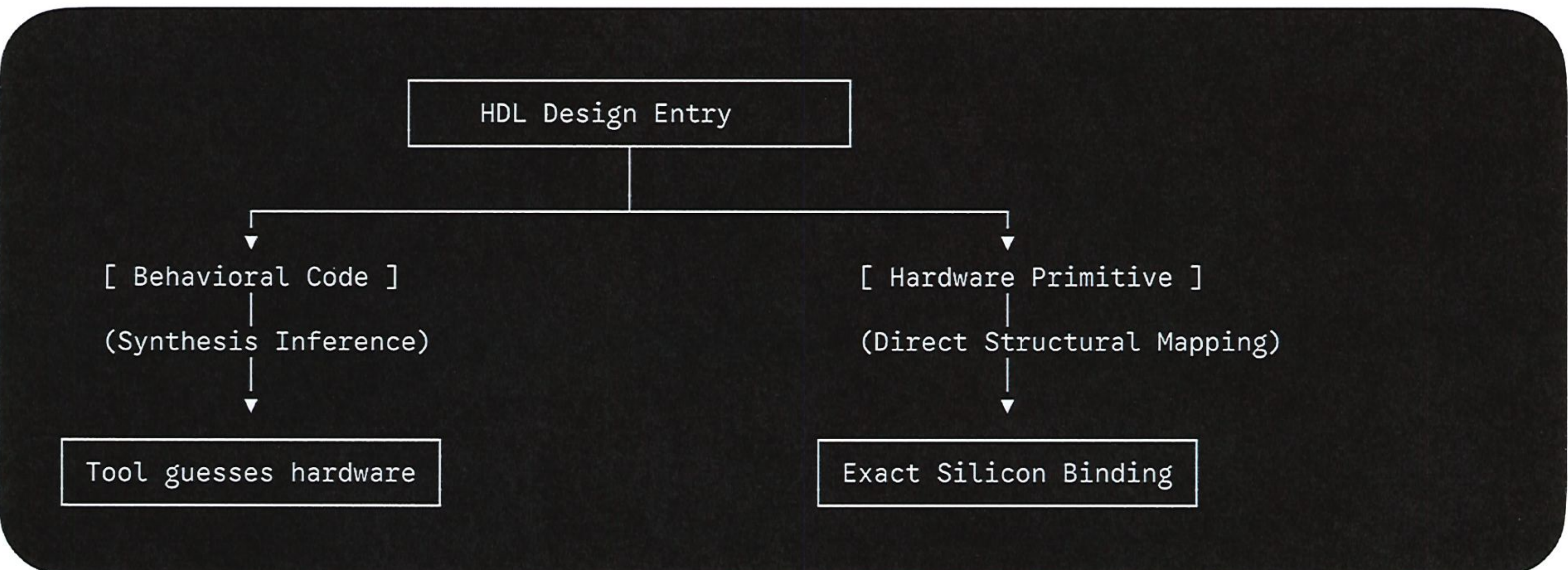
Power switch

VGA port

USB Port

POWER
ON
OFF
5V ONLY
USB HOST
ETHERNET
LINK/ACT
100/10
VGA
MONO AUDIO OUT
PROG UART
SD MICRO
BUSY
NEXYS 4 DDR
MODE
QSPI
JTAG
USB/SD
DONE
CPU RESET
PROG
DIGILENT
www.digilentinc.com
Copyright 2014
XILINX
ANALOG DEVICES
MIC
BTNU (M18)
BTNL (P17)
BTNC (N17)
BTNR (M17)
BTND (P18)
TEMP SENSOR

On board



Virtex-5 Libraries

when picking cores use

- BRAM
- FF
- BUFG

verilog

```verilog
module inferred_bram_write_first (
    input  clk,
    input  we,
    input  [9:0] addr,
    input  [15:0] data_in,
    output reg [15:0] data_out
);
    // 1024 words x 16 bits = 16,384 bits
    reg [15:0] ram [1023:0];

    always @(posedge clk) begin
        if (we) begin
            ram[addr] <= data_in;
        end

        // Synchronous read structured for WRITE_FIRST behavior.
        // By evaluating during the same clock cycle, the output register
        // latches the newly updated 'data_in' value when 'we' is active.
        data_out <= we ? data_in : ram[addr];
    end
endmodule
```

If you accidentally use an asynchronous read statement (such as assign data_out = ram[addr];), the synthesis tool will implement the memory as **Distributed RAM** rather than a dedicated Block RAM. This occurs because an asynchronous read does not reflect the underlying hardware component operation; a hard BRAM primitive physically lacks an unclocked read path, forcing Vivado to consume hundreds of general-purpose fabric LUTs (SLICEMs) to replicate the unclocked behavior.

Slide 1

Xilinx/AMD FPGA Initialization with GSR

- Xilinx (AMD) FPGAs use **Global Set/Reset (GSR)** to initialize all registers at configuration time
- Works for both:
  - Inline initialization (`reg [7:0] foo = 8'hAA;`)
  - `initial` blocks
- Both map to the **INIT** attribute on flip-flop primitives
- GSR deasserts after configuration → normal logic takes over

---

Slide 2

GSR Behavior Summary

- Asynchronously forces all flip-flops to their INIT values during configuration
- Fully synthesizable on AMD/Xilinx devices
- Matches hardware behavior in post-implementation simulation
- Provides reliable power-on state without user reset
- Common and recommended practice

Slide 3

Verilog Example

```verilog
reg [7:0] state = 8'h00;          // Declaration-time init

reg [7:0] counter;

initial begin
    counter = 8'hFF;              // Initial block
end
```

Both registers initialize via GSR on power-up

Slide 4

Periodic Global Reset Using GSR?

This is a Bad Idea

- GSR is intended **only** for post-configuration initialization
- Driving GSR at runtime (via STARTUPE2/STARTUPE3) is strongly discouraged by AMD

Slide 5

Why Runtime GSR is Problematic

1. Asynchronous release → risk of metastability and timing issues across clock domains
2. Resets **every** flip-flop in the entire FPGA (indiscriminate)
3. Does **not** reset Block RAM contents (incomplete reset)
4. Creates huge high-fanout event → power and timing problems
5. Difficult to properly constrain in Vivado

Slide 6

AMD Official Recommendation

> "Although you can access the GSR net after configuration, **avoid the use of the GSR circuitry in place of a manual reset.**"

- Use GSR only for initial configuration
- Not for periodic or runtime resets

Slide 7

Recommended Alternatives

- Generate a **synchronous reset** signal from a timer/counter (e.g., every 2 minutes)
- Distribute using synchronous resets or clock enables
- Apply reset only to subsystems that need it
- Use proper reset synchronizers for each clock domain
- Much cleaner, more reliable, and easier to time

Slide 8

Best Practice Summary

- Rely on GSR for initial power-on initialization
- Use clean **synchronous resets** for all runtime reset needs
- Avoid driving GSR after configuration
- Result: More reliable design and better timing closure

manual
external reset

Exactly. You have got it 100% straight.

By using a **BUFG**, you bypass the generic FPGA fabric routing altogether for that signal. The **BUFG** taps directly into a dedicated, heavy-duty global network line that is physically manufactured onto the silicon die.

The only routing the Place and Route (PAR) tool has to do is:

1. Connect your 2-bit synchronizer's output to the **BUFG** input.
2. Connect the pre-existing global network lines directly to the **CLR/PRE** pins of your target registers.

This completely eliminates the need for the tools to weave a massive, high-fanout reset net through the standard local fabric wiring, ensuring your manual reset has near-zero overhead and near-zero skew.

How the Synthesis Tool Translates This

When the compiler processes the code above, it executes the following logic mapping:

- **For clear_reg:** The tool sees if (global_reset) clear_reg <= 1'b0;. It hooks the global_reset net (the output of your BUFG) directly to the dedicated **CLR / SR** pin of that register's slice resource.
- **For preset_reg:** The tool sees if (global_reset) preset_reg <= 1'b1;. It routes that exact same global_reset net to the dedicated **PRE** pin of that register's slice resource.

```
module inferred_bufg_example (
   input  wire clk,
   input  wire manual_reset_in, // Raw asynchronous input from an external pin
   output reg  out_reg1,
   output reg  out_reg2
);

   // ----------------------------------------------------
   // 1. The 2-Bit Shift Register (Reset Synchronizer)
   // ----------------------------------------------------
   reg rst_sync_reg1 = 1'b1;
   reg rst_sync_reg2 = 1'b1;

   // This net will be inferred as a BUFG by the synthesis tool
   wire global_reset;

   always @(posedge clk or posedge manual_reset_in) begin
      if (manual_reset_in) begin
         rst_sync_reg1 <= 1'b1;
         rst_sync_reg2 <= 1'b1;
      end else begin
         rst_sync_reg1 <= 1'b0;
         rst_sync_reg2 <= rst_sync_reg1;
      end
   end

   // Assign the synchronized output to your global net
   assign global_reset = rst_sync_reg2;

   // ----------------------------------------------------
   // 2. Behavioral Register Logic (Triggers Inference)
   // ----------------------------------------------------

   // Example Register 1: Clear (Reset to 0)
   always @(posedge clk or posedge global_reset) begin
      if (global_reset) begin
         out_reg1 <= 1'b0; // Maps to physical CLR pin
      end else begin
         out_reg1 <= out_reg1 + 1'b1;
      end
   end

   // Example Register 2: Preset (Reset to 1)
   always @(posedge clk or posedge global_reset) begin
      if (global_reset) begin
         out_reg2 <= 1'b1; // Maps to physical PRE pin
      end else begin
         out_reg2 <= out_reg2 ^ out_reg1;
      end
   end

endmodule
```

```
module system_reset_example (
   input  wire clk,
   input  wire manual_reset_in, // Raw asynchronous input from pin
   output reg  clear_reg,       // To be cleared (0)
   output reg  preset_reg       // To be preset (1)
);

   // ----------------------------------------------------
   // 1. The 2-Bit Shift Register (Reset Synchronizer)
   // ----------------------------------------------------
   reg rst_sync_reg1 = 1'b1;
   reg rst_sync_reg2 = 1'b1;
   wire global_reset;

   always @(posedge clk or posedge manual_reset_in) begin
      if (manual_reset_in) begin
         rst_sync_reg1 <= 1'b1;
         rst_sync_reg2 <= 1'b1;
      end else begin
         rst_sync_reg1 <= 1'b0;
         rst_sync_reg2 <= rst_sync_reg1;
      end
   end

   // ----------------------------------------------------
   // 2. Explicit BUFG Instantiation
   // ----------------------------------------------------
   // This drives the low-skew global routing tree.
   BUFG bufg_reset_inst (
      .I(rst_sync_reg2),  // Input from the synchronizer
      .O(global_reset)    // Output to be used as the design's reset net
   );

   // ----------------------------------------------------
   // 3. Register Implementation (Clear vs. Preset)
   // ----------------------------------------------------

   // This register maps to the physical CLEAR input of the slice
   always @(posedge clk or posedge global_reset) begin
      if (global_reset) begin
         clear_reg <= 1'b0;  // Literal '0' tells synthesis to use the Clear pin
      end else begin
         clear_reg <= clear_reg + 1'b1; // Normal operation
      end
   end

   // This register maps to the physical PRESET input of the slice
   always @(posedge clk or posedge global_reset) begin
      if (global_reset) begin
         preset_reg <= 1'b1; // Literal '1' tells synthesis to use the Preset pin
      end else begin
         preset_reg <= preset_reg ~^ clear_reg; // Normal operation
      end
   end

endmodule
```

inferred automatically if reset fanout is > 100 or so

- **Synthesis Attributes (Optional Force):** If your design is small but you want to force the tool to infer a BUFG anyway, you can use a synthesis attribute directly above the wire declaration in your Verilog file:

```verilog
(* bufg = "yes" *) wire global_reset;
```

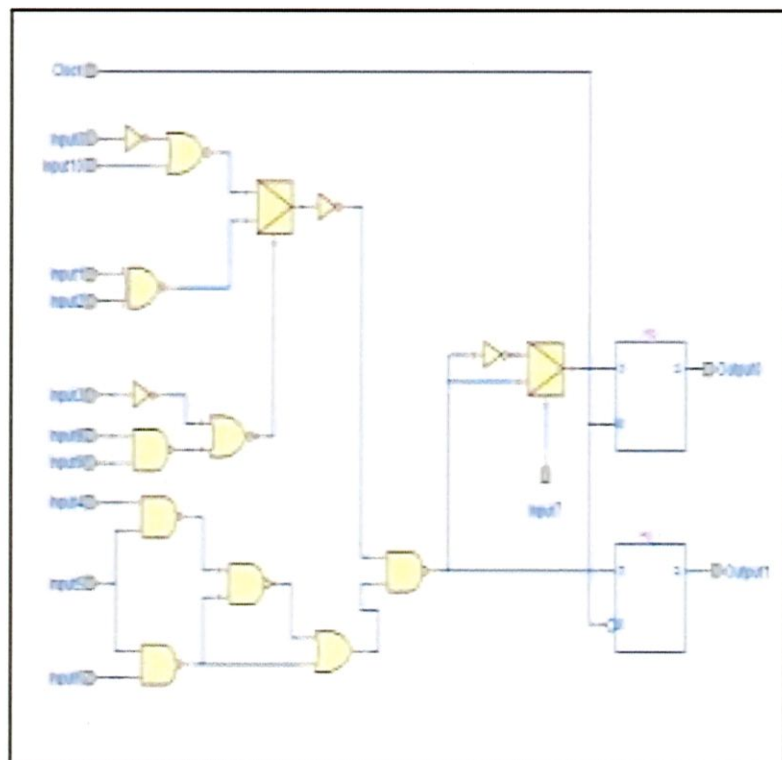
This metadata tag instructs tools like Vivado to bypass high-fanout evaluation thresholds and immediately assign a global routing buffer to the net.

Logic Synthesis

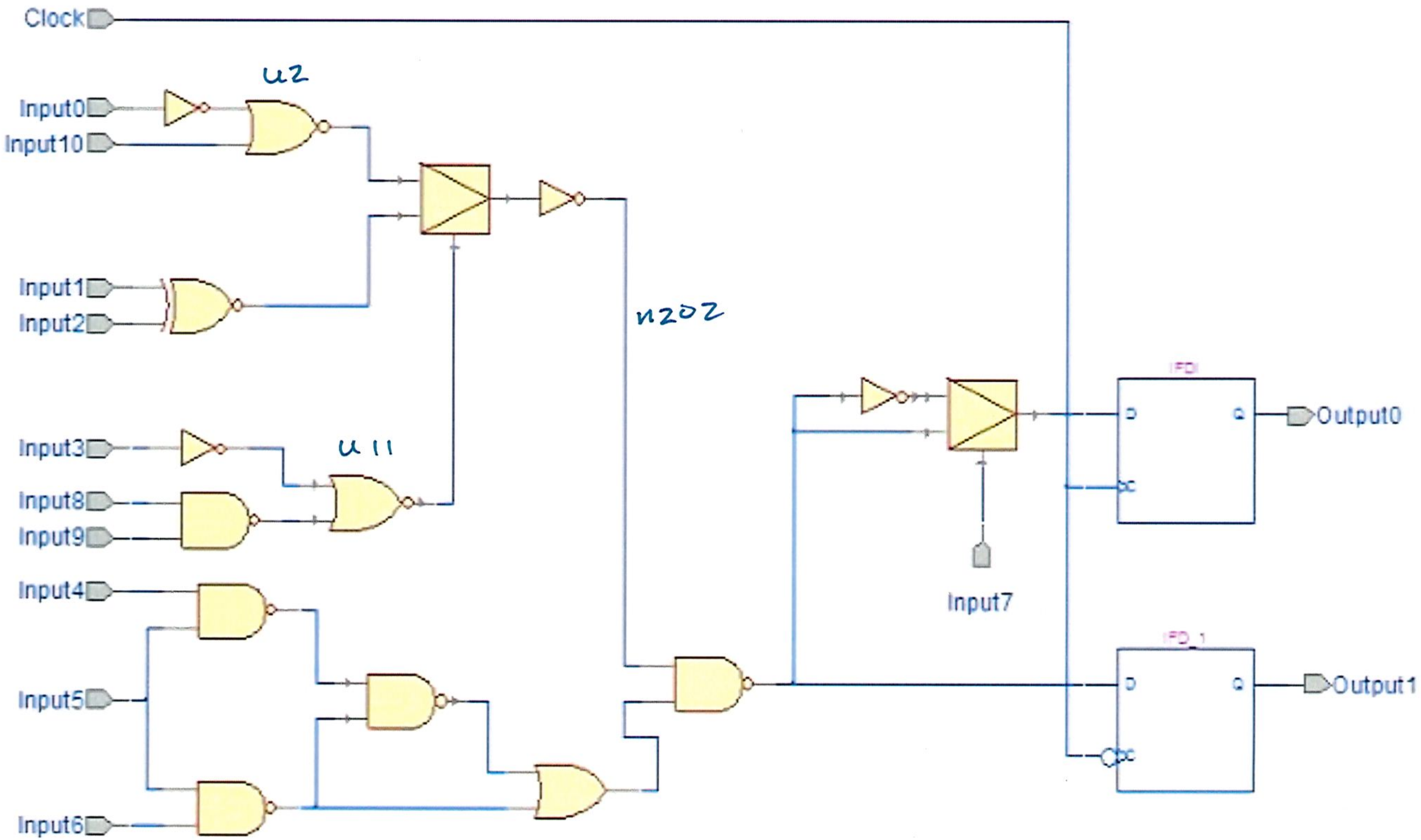
VHDL description

```
architecture MLU_DATAFLOW of MLU is

signal A1:STD_LOGIC;
signal B1:STD_LOGIC;
signal Y1:STD_LOGIC;
signal MUX_0,MUX_1,MUX_2,MUX_3: STD_LOGIC;

begin
            A1<=A when (NEG_A='0') else
                        not A;
            B1<=B when (NEG_B='0') else
                        not B;
            Y<=Y1 when (NEG_Y='0') else
                        not Y1;

            MUX_0<=A1 and B1;
            MUX_1<=A1 or B1;
            MUX_2<=A1 xor B1;
            MUX_3<=A1 xnor B1;

            with (L1 & L0) select
                        Y1<=MUX_0 when "00",
                                    MUX_1 when "01",
                                    MUX_2 when "10",
                                    MUX_3 when others;

end MLU_DATAFLOW;
```

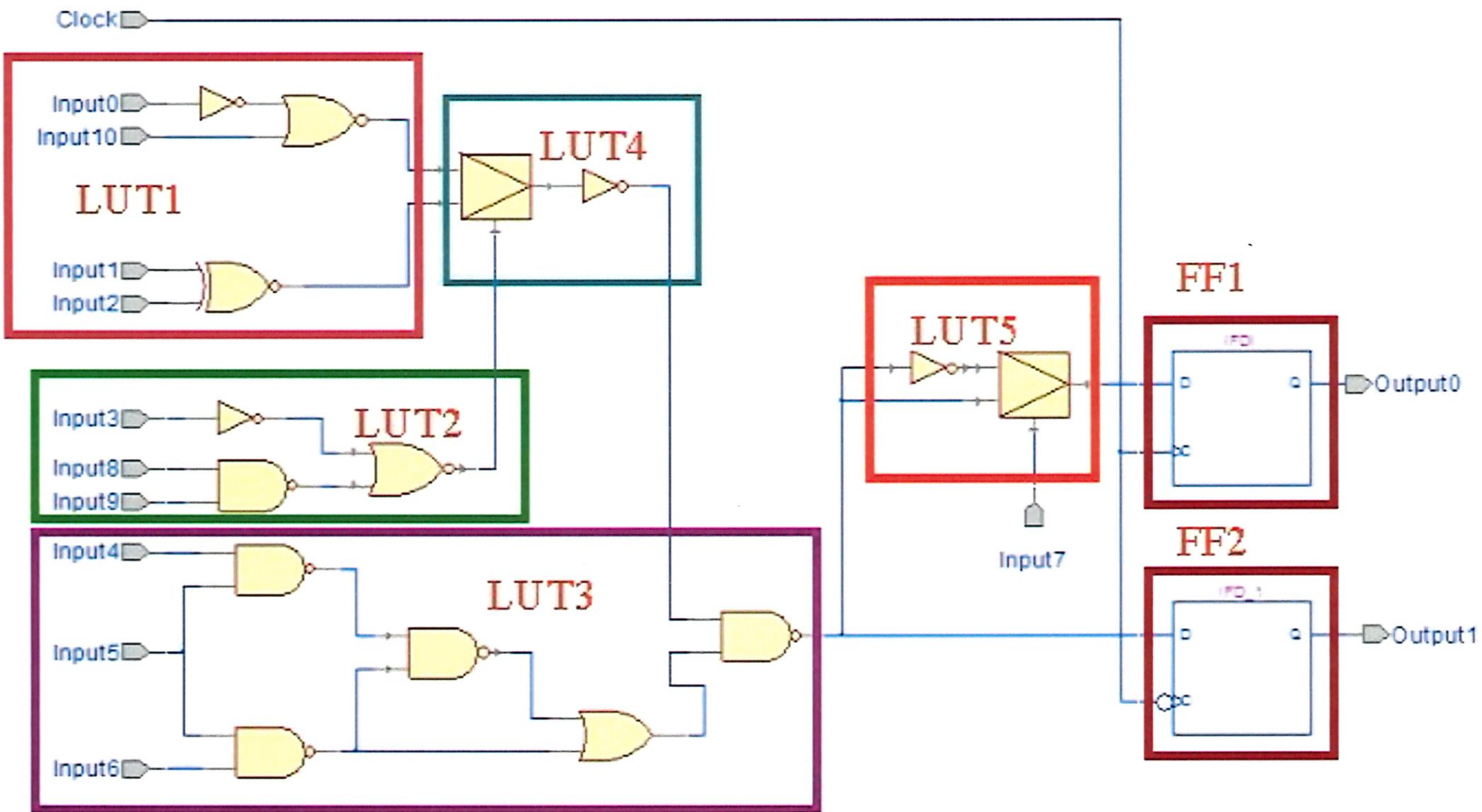
Circuit netlist



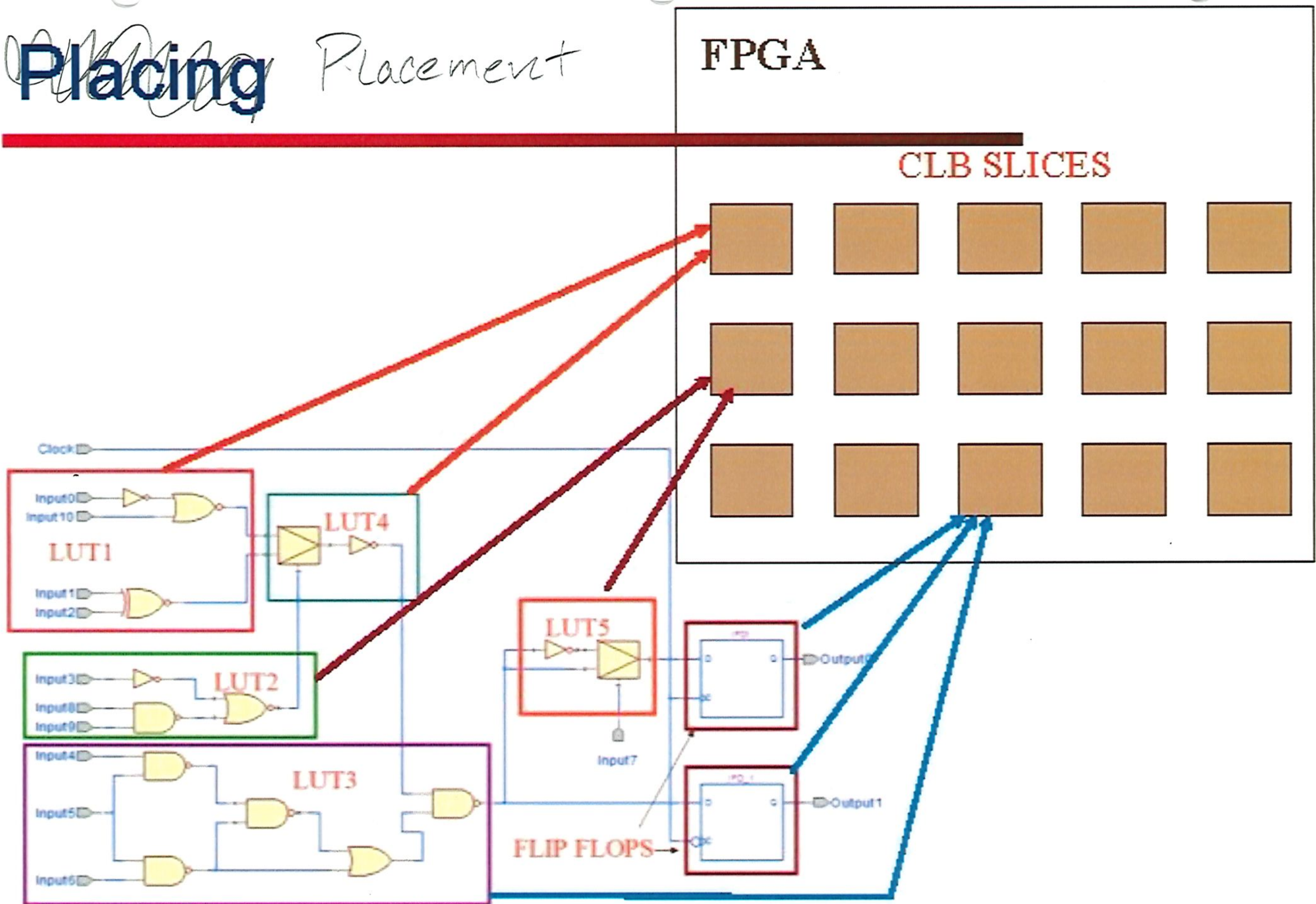
Circuit netlist



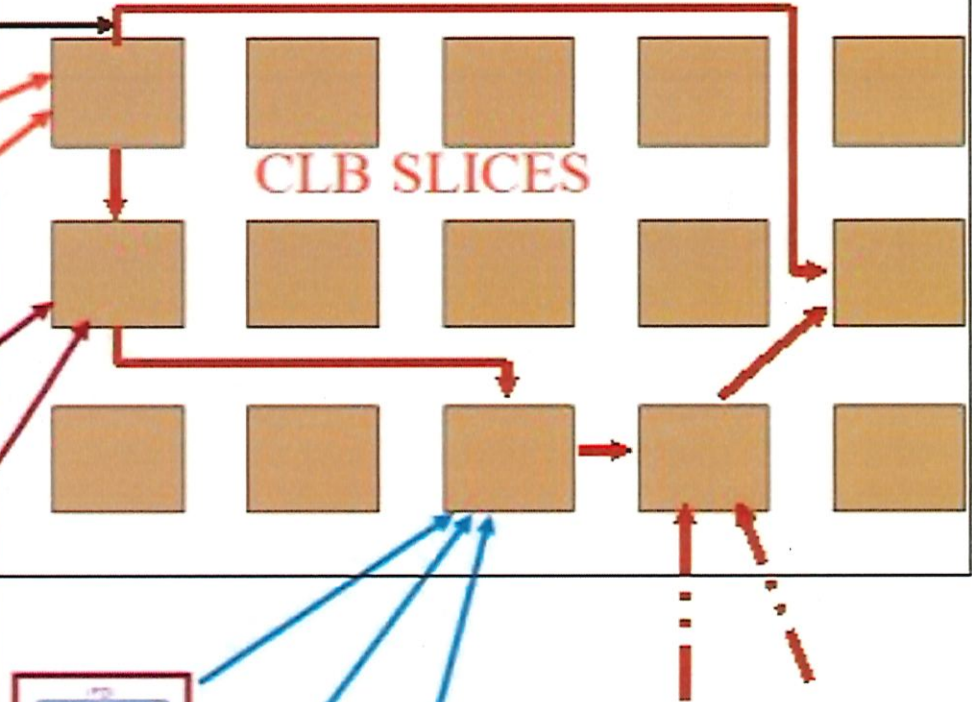
Mapping

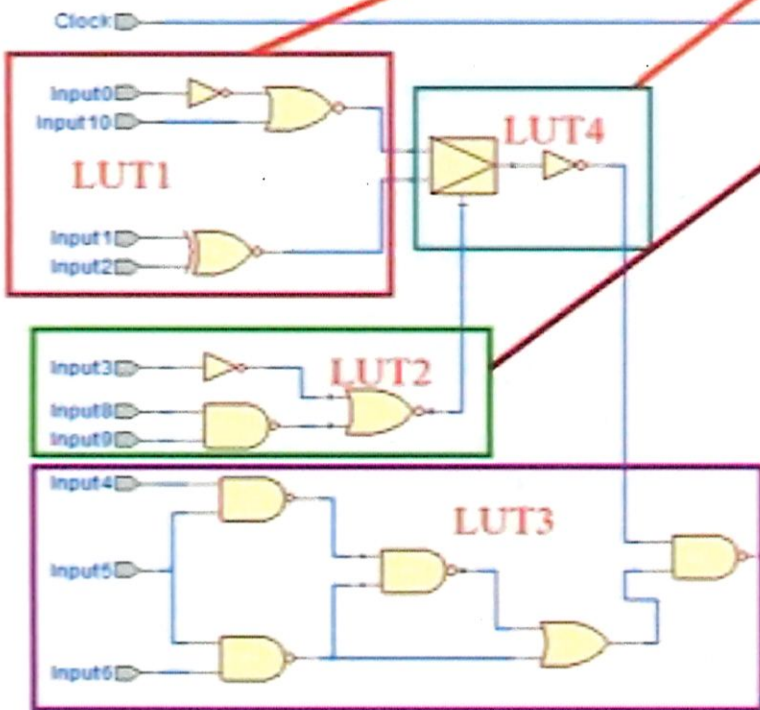


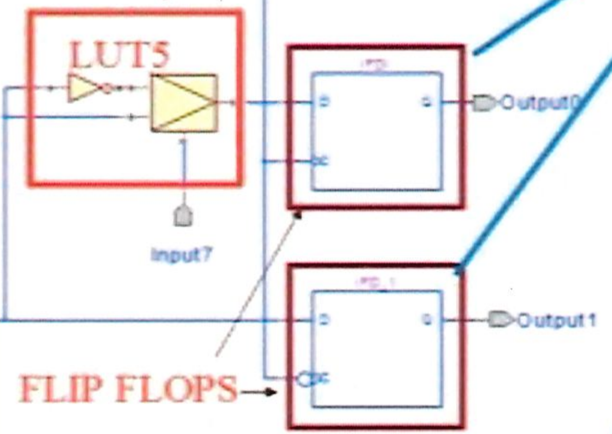
Placing



Routing







List of FPGA Synthesis Tools

The following list comprises or FPGA synthesis software tool available in the market:

- XST (delivered within ISE) by Xilinx
- Vivado by Xilinx
- Quartus II integrated Synthesis by Altera
- IspLever by Lattice Semiconductor
- Encounter RTL Compiler by Cadence Design Systems
- LeonardoSpectrum and Precision (RTL / Physical) by Mentor Graphics
- Synplify (PRO / Premier) by Synopsys
- BlastFPGA by Magma Design Automation