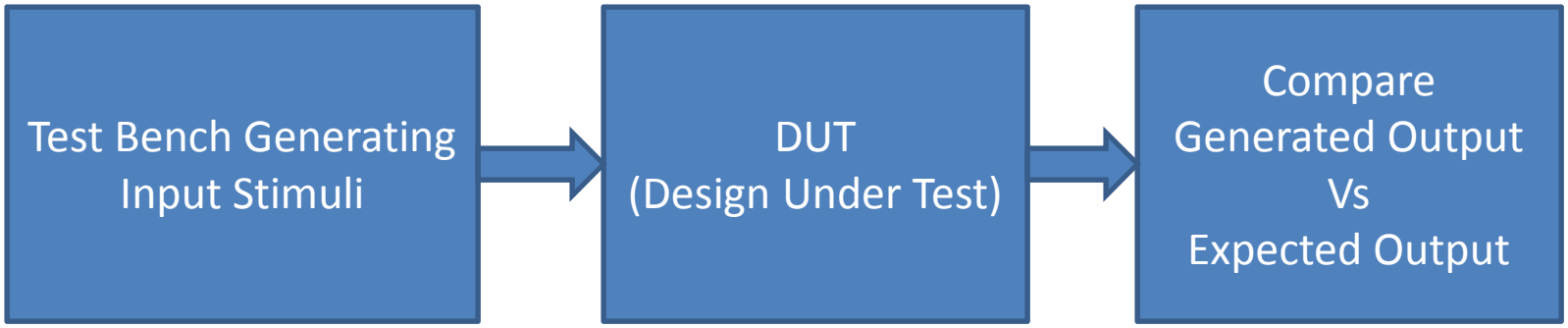
Writing Test Bench in HDL

What is a test bench?

- Test Bench is a HDL program that verifies the functional correctness of the Hardware design.
- The test bench HDL program checks whether the designed hardware model does what it is supposed to do and is not doing what it is not supposed to do.

Functions of a Test Bench

- Generate stimulus for testing the hardware design(DUT: Design Under Test).
- Apply the stimulus vectors to DUT.
- Compare the generated outputs against the expected outputs.



Linear Verilog Test Bench Example

Linear test bench is the simplest way of writing a test bench. Typically, linear test benches perform the following tasks:

- Instantiate the design under test (DUT)
- Stimulate the DUT by applying Input test vectors.
- Results output to an waveform window or to a terminal for display.

```
//DUT adder Module
module adder(a,b,c);

input [15:0] a;
input [15:0] b;
output [16:0] c;
assign c = a + b;
endmodule
```

```
module top();
reg [15:0] a;
reg [15:0] b;
wire [16:0] c;
adder DUT(a,b,c); //DUT Instantiation
initial  //apply the stimulus
   begin  a = 16'h45; b = 16'h12;
          #10 $display("a=%0d,b=%0d,c=%0d",a,b,c);
   end
endmodule
```

Self Checking Test Bench Example

A self-checking test bench checks expected results against actual results obtained from the simulation. Design verification and Debugging time is significantly shortened by useful error-tracking information that can be built into the test bench to show where a design fails.

```
//DUT adder Module
module adder(a,b,c);

input [15:0] a;
input [15:0] b;
output [16:0] c;
assign c = a + b;
endmodule
```

```
module top();
reg [15:0] a;
reg [15:0] b;
wire [16:0] c;
adder DUT(a,b,c); //DUT Instantiation
initial  //apply random stimulus using $random
repeat(100) begin  a = $random; b = $random;
        #10  $display(" a=%0d,b=%0d,c=%0d",a,b,c);
        if( a + b != c) // monitor logic.
            $display(" *ERROR* "); end
endmodule
```

File I/O Test Bench

Stimulus vectors can be supplied from an external file. Verilog HDL contains the $readmemb or $readmemh system tasks to do the file read if the file data is formatted in a specific way using either binary or hexadecimal format. Sometimes outputs are also written to external files. Example below illustrates how to initialize a memory array from data stored in a file in hex format.

Note: The data file must reside in the same directory as the .v file for the module in this example.

```
module readmemh_demo;
reg [31:0] Memory [0:7];  //memory with 8 location and 32 bit wide
initial $readmemh("data.txt",Memory); //read data from file “data.txt “
integer i;
initial
  begin
      #10;  $display("Contents of Memory after reading data file:");
      for (i = 0; i < 8; i = i + 1) $display("%d : %h ", i,Memory[i]);
end
endmodule
```

File I/O Test Bench Continue….

```
//data.txt file in hex format
//Example: data.txt file
234ac
23bc5
23c34
23d4a
258ca
b5234
abcd5
2345c
```

```
//each location is 32 bit
//Simulation Output Result:
0 : 000234ac
1 : 00023bc5
2 : 00023c34
3 : 00023d4a
4 : 000258ca
5 : 000b5234
6 : 000abcd5
7 : 0002345c
```

File I/O Test Bench with `include

Reading or writing to files during simulation is costly to performance, because the simulator must halt and wait while the OS completes each transaction with the file system. One way to improve performance is to replace ASCII vector files with a constant table in HDL itself.

```
module readmemh_demo;
reg [31:0] Memory [0:6];
`include "data.v"  //ASCII vector file with a constant  table in HDL
integer i;
initial
    begin
            #20;
            $display("Contents of Memory after reading data file:");
            for (i = 0; i < 6; i = i+1) $display("%d:%h", i, Memory[i]);
    end
endmodule
```

File I/O Test Bench Continue….

```
//Example: data.v file
EXAMPLE: data.v file
initial
begin
Mem[0] = 32'h234ac;
Mem[1] = 32'h23ca5;
Mem[2] = 32'hb3c34;
Mem[3] = 32'h23a4a;
Mem[4] = 32'h234ca;
Mem[5] = 32'hb3234;
end
```

```
//each location is 32 bit
//Simulation Output Result:

0:000234ac
1:00023ca5
2:000b3c34
3:00023a4a
4:000234ca
5:000b3234
```

Task and Function Based Test Bench

Task and function based test bench is more flexible for verification The task based BFM (Bus Functional Model) is extremely efficient if the device under test performs many complex tasks. Each task or function focuses on one single functionality. Verification of DUT using the task based test bench is faster.

```
//Example:
task write(input integer data, input integer addr); //write task
begin @(posedge clock);
    read_write = 1;
    addr = $random;
    data = $random; end
endtask
task read(input integer addr, output integer data);  //read task
begin  @(posedge clock);
    read_write = 0;
   addr = $random;  end
endtask
```

Task and Function Based Test Bench Continue ….

Lets see how we can use the read and write task to execute only write, only read, alternate read and write and simultaneous read and write operation.

```
initial  //10 write operations.
        repeat(10)  write($random, $random);

initial  / /10 read operations
        repeat(10) read($random, data);

initial          repeat(10) // Alternate read and write operations.
        begin
                write($random, $random);
                read($random, data);
        end
initial  // Do the write and read same location
        begin
                write(10,20);
                read (10,data);
        end
```

System Function $random

Verilog system function $random can be used to generate random input vectors. With $random, we can generate values which we can not generate manually. Sometimes Random vectors can catch some of the critical silicon bugs.

```
//Example: Use of $random system function
module mem_tb();
reg clock;  reg read_write;  reg [31:0] data;  reg [31:0] addr;
initial
       begin   clock = 0;  forever  #10 clock = ~clock; end
initial
  begin   repeat(5)@(negedge clock)
          begin  read_write = $random ; data = $random; addr= $random; $finish; end
  end
initial
  $monitor($time,"read_write = %d ;data =%d ;addr= %d;",read_write, data, addr);
endmodule
```

System Function $random Continue ….

$random() system function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative.

Simulation Result:

20 read_write = 0 ; data = 32302280 ; address = 22232980;
40 read_write = 1 ; data = 11281898; address = 11890589;
60 read_write = 1 ; data = 23021040; address = 15983361;
80 read_write = 1 ; data = 99221131; address = 51260959;

$random(seed); The value of seed is optional and is used to ensure the same random number sequence each time the test is run.

```
module test; integer r-seed; reg [31:01 addr;//input to ROM
wire [31:01 data; //output from ROM ... ...
ROM rom1 (data, addr) ; initial r-seed = 2;//arbitrarily define seed as 2
always @(posedge clock) addr = $random(r-seed);
endmodul e
```

File Handling Test Bench using System task

System task $fopen, $fclose: **$fopen** opens the file specified as the filename argument and returns either a 32 bit multi channel descriptor, or a 32 bit file descriptor, determined by the absence or presence of the type argument. **$fclose** closes the file.

```
//Example: $fopen and $fclose
module fopen_close();
integer mcd, number; //mcd is multi channel descriptor
initial  //Display mcd value before and after the opening the file.
    begin $display("value of mcd before file opening %b " , mcd);
          mcd = $fopen("myfile.txt"); // opening the file myfile.txt
          $display("value of mcd after opening the file %b " , mcd);
repeat(7)  begin
          number = $random ;
          $fdisplay(mcd, " Number is ", number); end
          $fclose(mcd); // closing the file
        end endmodule
```

RESULT : **value of mcd before opening the file** **xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**
value of mcd after opening the file **00000000000000000000000000000010**

File Handling Test Bench using System task

System task $fwrite: $fwrite, $fwriteb, $fwriteo, $fwriteh
Like $display; $write also have counterparts.

```
// file open close example with $fwrite
module write_task();
integer mcd1,mcd2,number,pointer;
initial begin
$display("value of mcd1 before opening the file %b " , mcd1);
$display("value of mcd2 before opening the file %b " , mcd2);
mcd1 = $fopen(“abc.txt");  mcd2 = $fopen("ptr.txt");
$display("value of mcd1 after opening the file %b " , mcd1);
$display("value of mcd2 after opening the file %b " , mcd2);
repeat(7) begin
pointer = $random; number = $random % 10;
$fwriteo(mcd1, " Number is ", number);
$fwriteh(mcd2, " Pointer is ", pointer); end
$fclose(mcd1);
$fclose(mcd2);  end
endmodule
```

File Handling Test Bench Output

One of the reasons behind writing the example in previous slide is to show how the integers are getting different value as per the number of files are opened.

RESULT :

value of mcd1 before opening the file

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

value of mcd2 before opening the file

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

value of mcd1 after opening the file

00000000000000000000000000000010

value of mcd2 after opening the file

00000000000000000000000000000100

//In file ptr.txt

Pointer is 12173524 Pointer is 8484d609 Pointer is 06b97b0d Pointer is b2c28565 Pointer is 00f3e301 Pointer is 3b21f176 Pointer is 76db57ed

//In file abc.txt

Number is 37777777767 Number is 37777757767 Number is 00000000007 Number is 37777777774 Number is 00000000011 Number is 00000010007 Number is 00000000002

Simultaneously Writing to 2 files using mcd

This example shows how to set up multi channel descriptors and Simultaneously writing same data to two different file. In this example, two different channels are opened using the $fopen function. The two multi channel descriptors that are returned by the function are then combined in a bit-wise or operation and assigned to the integer variable "broadcast". The "broadcast" variable can then be used as the first parameter in a file output task to direct output to all two channels at once.

```
module writetask();
integer mcd1,mcd2,broadcast,number;
initial begin
    mcd1 = $fopen(“file1.txt"); mcd2 = $fopen(“file2.txt");
    broadcast = mcd1 |mcd2 ;
repeat(7) begin number = $random;
            $fdisplayh(broadcast," Number is ", number); end
    $fclose(mcd1);
    $fclose(mcd2);  end
endmodule
```

Simultaneously Writing to 2 files Result

```
//In file1.txt
Number is 12234524
Number is c0ab5e81
Number is 8484e409
Number is b1f03363
Number is 06b97b0d
Number is 46df998d
Number is b2c28465

//In file2.txt
Number is 12167524
Number is c08afe81
Number is 8484d318
Number is b1f00363
Number is 06b97b0d
Number is 46df888d
Number is b2ace465
```

System Function $dumpvars,$dumpfile

System tasks are provided for selecting module instances or module instance signals to dump ($dumpvars), name of VCD(Value change Dump) file ($dumpfile), starting and stopping the dump process ($dumpon, $dumpoff), and generating checkpoints ($dumpall). Example below shows all usage case.

```
initial
  $dumpfile("design.dmp"); //Simulation info dumped to design.dmp
Initial $dumpvars; //no arguments, dump all signals in the design
initial $dumpvars(1, top); //dump variables in module instance top.
initial $dumpvars (2, top .m1 ) ; //dump up to 2 levels of hierarchy below top .m1
initial $dumpvars(0, top.m1); //Number 0 means dump the entire hierarchy below top.m1
initial
    begin  $dumpon ; //start the dump process.
              #10000 $dumpoff; //stop the dump process after 10,000 time units
    end
//Create a checkpoint, Dump current value of all VCD variables
initial $dumpall;
```

Simple Counter Test Bench Example

```
// counter Verilog source file
‘timescale 1 ns / 100 ps
module counter (count, count_tri, clk, rst, load, enable, cnt_in, oe);
output [3:0] count;
output [3:0] count_tri;
input clk, rst, load, enable, oe;
input [3:0] cnt_in;
reg [3:0] count;
// tri-state buffers
assign count_tri = (!oe) ? count : 4’hZ;
always @ (posedge clk or negedge rst)
begin
  if (!rst)
            count <= #1 4’b0000;
  else if (!load)
            count <= #1 cnt_in;
  else if (!enable)
            count <= #1 count + 1;
end
endmodule
```

Simple Counter Test Bench Example Cont..

```
// counter test bench Verilog file
‘timescale 1 ns / 100 ps
module counter_tb ();
// inputs to the DUT are reg type
reg clk_50;
reg rst, load, enable;
reg [3:0] count_in;
reg oe;
// outputs from the DUT are wire type
wire [3:0] cnt_out;
wire [3:0] count_tri;
// instantiate the Device Under Test (DUT)
// using named instantiation
count16 U1 ( .count(cnt_out),
.count_tri(count_tri),.clk(clk_50),.rst_l(rst),.load(load),
.cnt_in(count_in),.enable(enable),.oe(oe));
```

Simple Counter Test Bench Example Cont..

```
// create a 50Mhz clock
always #10 clk_50 = ~clk_50; // every ten nanoseconds invert
// initial blocks are sequential and start at time 0
initial
begin
        $display($time, " << Starting the Simulation >>");
        clk_50 = 1'b0;
        rst = 0;  //reset is active
        enable = 1'b1; //disabled
        load = 1'b1; // disabled
        count_in = 4'h0;
        oe = 4'b0; // enabled
        #20 rst = 1'b1; // reset released at time 20
        $display($time, " << Coming out of reset >>");
        @(negedge clk_50); // wait till the negedge of clk_50
        load_count(4'hA); // call the load_count task and pass 4'hA
```

Simple Counter Test Bench Example Cont..

```
@(negedge clk_50);
$display($time, " << Turning ON the count enable >>");
enable = 1'b0; //turn ON enable
// let the simulation run,
wait (cnt_out == 4'b0001); // wait until the count equals 1 and continue
$display($time, " << count = %d - Turning OFF the count enable >>", cnt_out);
enable = 1'b1;
#40; // let the simulation run for 40ns
// the counter shouldn't count
$display($time, " << Turning OFF the OE >>");
oe = 1'b1; //disable oe and count_tri goes high Z
#20;
$display($time, " << Simulation Complete >>");
$stop;
// stop the simulation
end
```

Simple Counter Test Bench Example Cont..

```
// This initial block runs concurrently with the other
initial begin
// $monitor will print whenever a signal changes
$monitor($time, " clk_50=%b, rst=%b, enable=%b, load=%b,
count_in=%h, cnt_out=%h, oe=%b, count_tri=%h", clk_50, rst,
enable, load, count_in, cnt_out, oe, count_tri);
end
task load_count; // The load_count task loads the counter with value passed
input [3:0] load_value;
begin
        @(negedge clk_50);
        $display($time, " << Loading the counter with %h >>", load_value);
        load = 1'b0; // load enabled
        count_in = load_value;
        @(negedge clk_50);
        load = 1'b1;//load disabled
end
endtask
endmodule
```

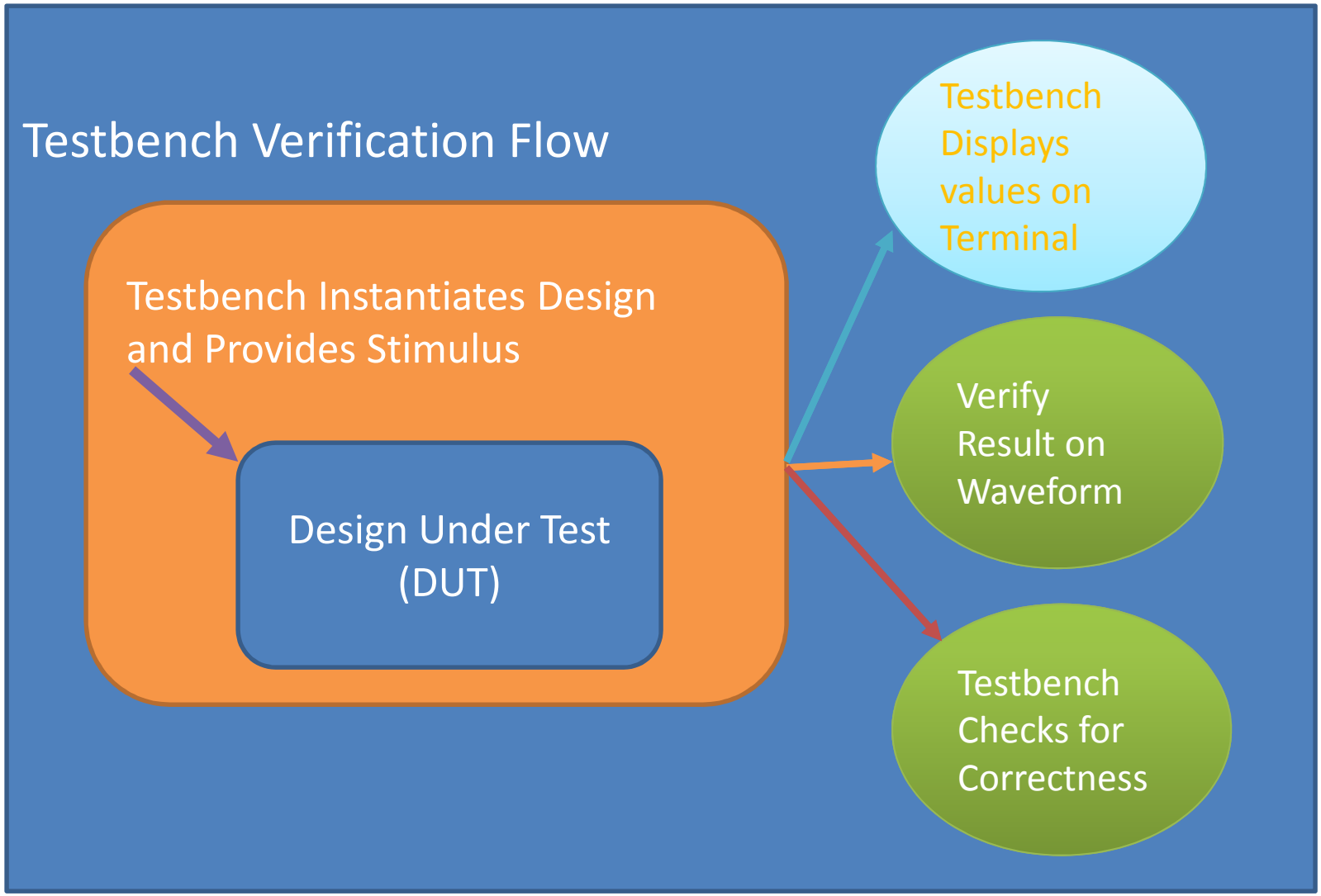
Writing Effective Test Benches

Due to increases in design size and complexity, digital design verification has become an increasingly difficult and laborious task. To meet this challenge, verification engineers rely on several verification tools and methods. For large, multi-million gate designs, engineers typically use a suite of formal verification tools. However, for smaller designs, design engineers usually find that HDL simulators with testbenches work best.

Testbenches have become the standard method to verify HLL (High-Level Language) designs. Typically, testbenches perform the following tasks:

- Instantiate the design under test (DUT)
- Stimulate the DUT by applying test vectors
- Output results to a terminal or waveform window for visual inspection
- Optionally compare actual results to expected results

Testbench Verification Flow



Writing Effective Test Benches

Step1: Generating Clock Signals

```
// Declare a clock period constant using Parameter.
parameter ClockPeriod = 10;
// Clock Generation method 1:
initial
   begin
         forever Clock = #(ClockPeriod / 2) ~ Clock;
   end
// Clock Generation method 2:
initial
    begin
         always #(ClockPeriod / 2) Clock = ~Clock;
    end
```

Writing Effective Test Benches

Step2: Providing Stimulus

To obtain testbench verification results, stimulus must be provided to the DUT. Concurrent stimulus blocks are used in testbenches to provide the necessary stimuli. Two methods are employed:

- Absolute-time stimulus
- Relative-time stimulus.

In the first method, simulation values are specified relative to simulation time zero. By comparison, relative-time stimulus supplies initial values, then waits for an event before retriggering the stimulus. Both methods can be combined in a testbench, according to the designer's needs.

Writing Effective Test Benches

Verilog-ABSOLUTE TIME

```
initial
  begin
        Reset = 1;
        Load = 0;
        Count_UpDn = 0;
        #100 Reset = 0;
        #20 Load = 1;
        #20 Count_UpDn = 1;
end
```

Verilog-RELATIVE TIME

```
always @ (posedge clock)
  TB_Count <= TB_Count + 1;
initial
  begin
    if (TB_Count <= 5)  begin
          Reset = 1;  Load = 0;
          Count _UpDn = 0;  end
    else begin
          Reset = 0;  Load = 1;
          Count_UpDn = 1; end
  end
initial
    begin
          if (Count == 1100) begin
          Count_UpDn <= 0;
          $display("Terminal Count
          Reached, now counting down.");
          end
    end
```

Writing Effective Test Benches

Step3: Displaying Results

```
// pipes the ASCII results to the terminal or text editor
initial
  begin
      $display(" Time Clk Rst Ld SftRg Data Sel");
      $monitor("%t %b %b %b %b %b %b", $realtime,
                    clock, reset, load, shiftreg, data, sel);
  end
```

```
# Time      Clk   Rst   Ld    SftRg     Data     Sel
  #0         0     1     0     xxxxx     00000    00
  #50        1     1     0     00000     00000    00
 #100        0     1     0     00000     00000    00
 #150        1     1     0     00000     00000    00
```

Automatic Verifications

Automating the verification of testbench results is recommended, particularly for larger designs. Automation reduces the time required to check a design for correctness, and minimizes human error.

Several methods are commonly used to automate testbench verification:

- **Database Comparisons:** First, a database file containing expected output (a "golden vector" file) is created. Then, simulation outputs are captured and compared to the reference vectors in the golden vector file (the unix diff utility can be used to compare the ASCII database files). However, since pointers from output to input files are not provided, a disadvantage of this method is the difficulty of tracing an incorrect output to the source of the error.

Automatic Verifications

- **Waveform Comparison.** Waveform comparisons can be performed automatically or manually. The automatic method employs a testbench comparator to compare a golden waveform against the testbench output waveform.
- **Self-Checking Testbenches.** A self-checking testbench checks expected results against actual results at run time, not at the end of simulation. Since useful error-tracking information can be built into the testbench to show where a design fails, debugging time is significantly shortened. Further information on self-checking testbenches is provided in the next slide.

Automatic Verifications

Self-Checking Testbenches: Self-checking testbenches are implemented by placing a series of expected vectors in a testbench file. These vectors are compared at defined run-time intervals to actual simulation results. If actual results match expected results, the simulation succeeds. If results do not match expectations, the testbench reports the discrepancies.

Self checking Verilog test bench example

Following the instantiation of the design, expected results are specified. Later in the code, expected and actual results are compared, and the results are echoed to the terminal. If there are no mismatches, an “end of good simulation” message is displayed. If a mismatch occurs, an error is reported along with the mismatched expected and actual values.

Self checking Verilog test bench example

```
‘timescale 1 ns / 1 ps
module test_sc;
reg tbreset, tbstrtstop;
reg tbclk;
wire [6:0] onesout, tensout;
wire [9:0] tbtenthsout;
parameter cycles = 25;
reg [9:0] Data_in_t [0:cycles];
// Instantiation of the Design
stopwatch UUT (.CLK (tbclk), .RESET (tbreset), .STRTSTOP (tbstrtstop),
.ONESOUT (onesout), .TENSOUT (tensout), .TENTHSOUT
(tbtenthsout));
wire [4:0] tbonesout, tbtensout;
assign tbtensout = led2hex(tensout);
assign tbonesout = led2hex(onesout);
//EXPECTED RESULTS
```

Self checking Verilog test bench example

```
initial begin
Data_in_t[1] =10'b1111111110; Data_in_t[2] =10'b1111111101;
Data_in_t[3] =10'b1111111011; Data_in_t[4] =10'b1111110111;
Data_in_t[5] =10'b1111101111; Data_in_t[6] =10'b1111011111;
Data_in_t[7] =10'b1110111111; Data_in_t[8] =10'b1101111111;
Data_in_t[9] =10'b1011111111; Data_in_t[10]=10'b0111111111;
Data_in_t[11]=10'b1111111110; Data_in_t[12]=10'b1111111110;
Data_in_t[13]=10'b1111111101; Data_in_t[14]=10'b1111111011;
Data_in_t[15]=10'b1111110111; Data_in_t[16]=10'b1111101111;
Data_in_t[17]=10'b1111011111; Data_in_t[18]=10'b1110111111;
Data_in_t[19]=10'b1101111111; Data_in_t[20]=10'b1011111111;
Data_in_t[21]=10'b0111111111; Data_in_t[22]=10'b1111111110;
Data_in_t[23]=10'b1111111110; Data_in_t[24]=10'b1111111101;
Data_in_t[25]=10'b1111111011; end
```

Self checking Verilog test bench example

```
reg GSR;
assign glbl.GSR = GSR;
initial begin GSR = 1; // Wait till Global Reset Finished
        #100 GSR = 0; end
// Create the clock
initial begin tbclk = 0;
// Wait till Global Reset Finished, then cycle clock
#100 forever #60 tbclk = ~tbclk; end
initial begin
// Initialize All Input Ports
        tbreset = 1;
        tbstrtstop = 1;
// Apply Design Stimulus
#240 tbreset = 0; tbstrtstop = 0; #5000 tbstrtstop = 1;
#8125 tbstrtstop = 0; #500 tbstrtstop = 1; #875 tbreset = 1;
#375 tbreset = 0; #700 tbstrtstop = 0;
```

Self checking Verilog test bench example

```
#550 tbstrtstop = 1;
// simulation must be halted inside an initial statement
 #100000 $stop;
end
integer i,errors;
// Block below compares the expected vs. actual results
// at every negative clock edge.
always @ (posedge tbclk)
begin
        if (tbstrtstop)
                begin i = 0; errors = 0; end
        else
                begin
                        for (i = 1; i <= cycles; i = i + 1)
                                begin
                                        @(negedge tbclk)
// check result at negedge
```

Self checking Verilog test bench example

```
$display("Time%d ns; TBSTRTSTOP=%b; Reset=%h; Expected
TenthsOut=%b; Actual TenthsOut=%b", $stime, tbstrtstop, tbreset,
Data_in_t[i], tbtenthsout);
if ( tbtenthsout !== Data_in_t[i] )
begin $display(" ------ERROR. A mismatch has occurred-----");
        errors = errors + 1; end
end
if (errors == 0)
        $display("Simulation finished Successfully.");
else if (errors > 1)
        $display("%0d ERROR! See log above for details.",errors);
else
        $display("ERROR! See log above for details.");
        #100 $stop;
end
end
endmodule
```

FPGA Architecture and Design Flow

What is an FPGA (Field Programmable Gate Array)?

FPGAs are programmable semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects. As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements.

Basic Components in an FPGA:

The following are the basic components in an FPGA:

- **CLB (Configurable Logic Blocks)**
- **Interconnect**
- **Serial IO(IOBs) and Global Clock Buffers**
- **Memory (Embedded Block RAM Memory)**
- **Complete Clock Management Block(DCM)**

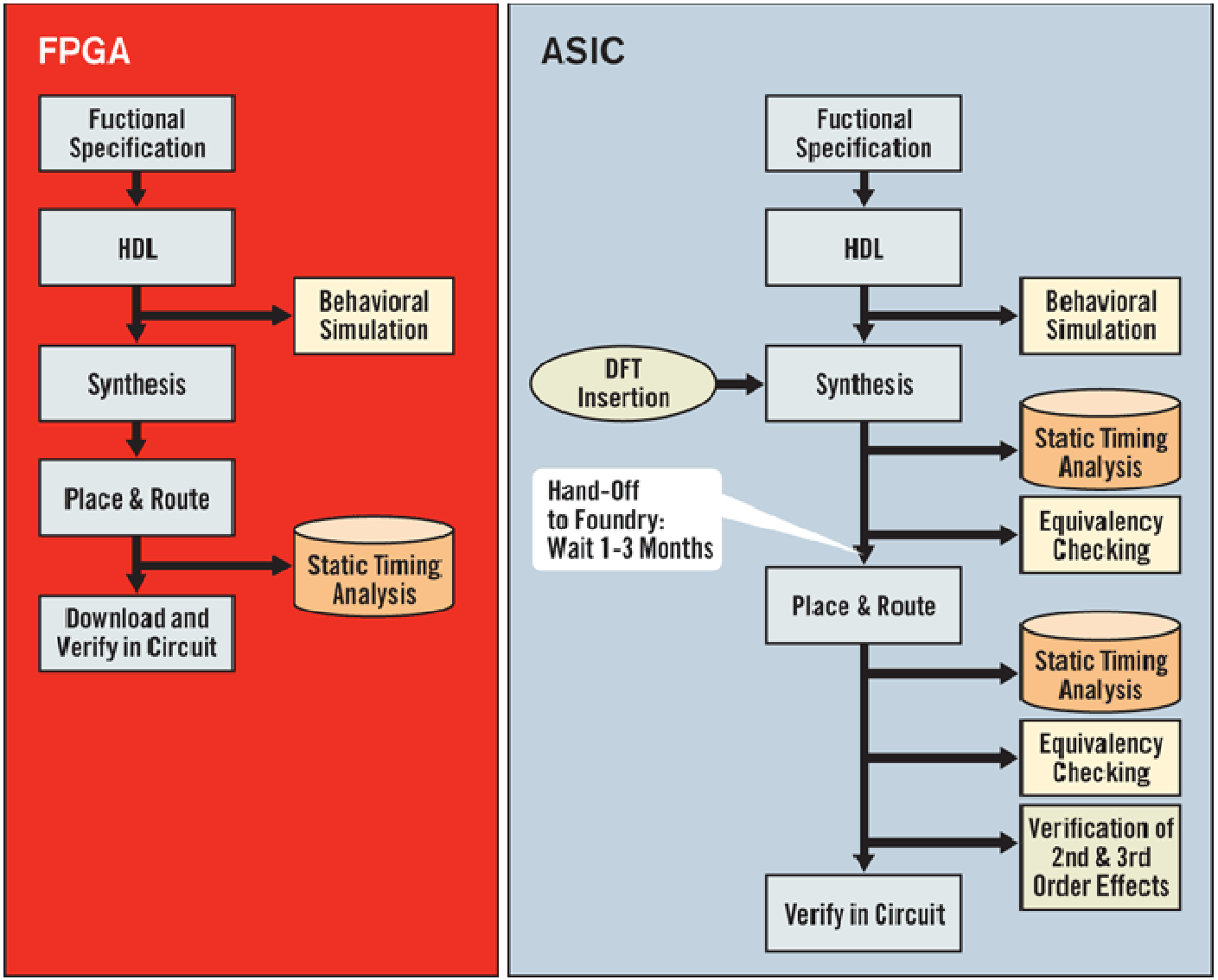
Naming Convention in Altera,Xilinx FPGA

In Xilinx FPGA an LC(Logic Cell) consist of 4-input look-up table (LUT), a D-flip-flop and some additional circuitry .In Altera LC is called LE(Logic Element). LE's and LC's used to be main building blocks in early designs and were used to express the complexity of FPGA structure. As FPGA architecture became more and more complex, manufacturers started calling their main building blocks differently. These "new" building blocks typically contain more than one LUT, more than one D-FF, and a mix of combinational, arithmetic, and register logic. The improved functionality of these "new" blocks earned them a new name. However, since they do contain the same elements as a simple LC (or LE), both manufacturers still list the equivalent number of LCs(LEs) as an important attribute in their datasheets. Altera adopted a new term ALM – Adaptive Logic Module for describing Stratix II family (1 ALM = 2.5 LEs). Xilinx uses the term CLB – Configurable Logic Block to name the basic logic block of all its FPGAs. Each CLB has 8 LCs. But since these 8 LCs provide a greater functionality than if they were separate, Xilinx now uses the unit ELC – Equivalent Logic Cell (1 ELC = 1.125 LC) to state the complexity of its FPGAs. To make it all more complicated, Xilinx introduced the term ASMBL – Advanced Silicon Modular Block (pronounced like "assemble") to describe the new feature-rich architecture of their Virtex-4 building blocks.

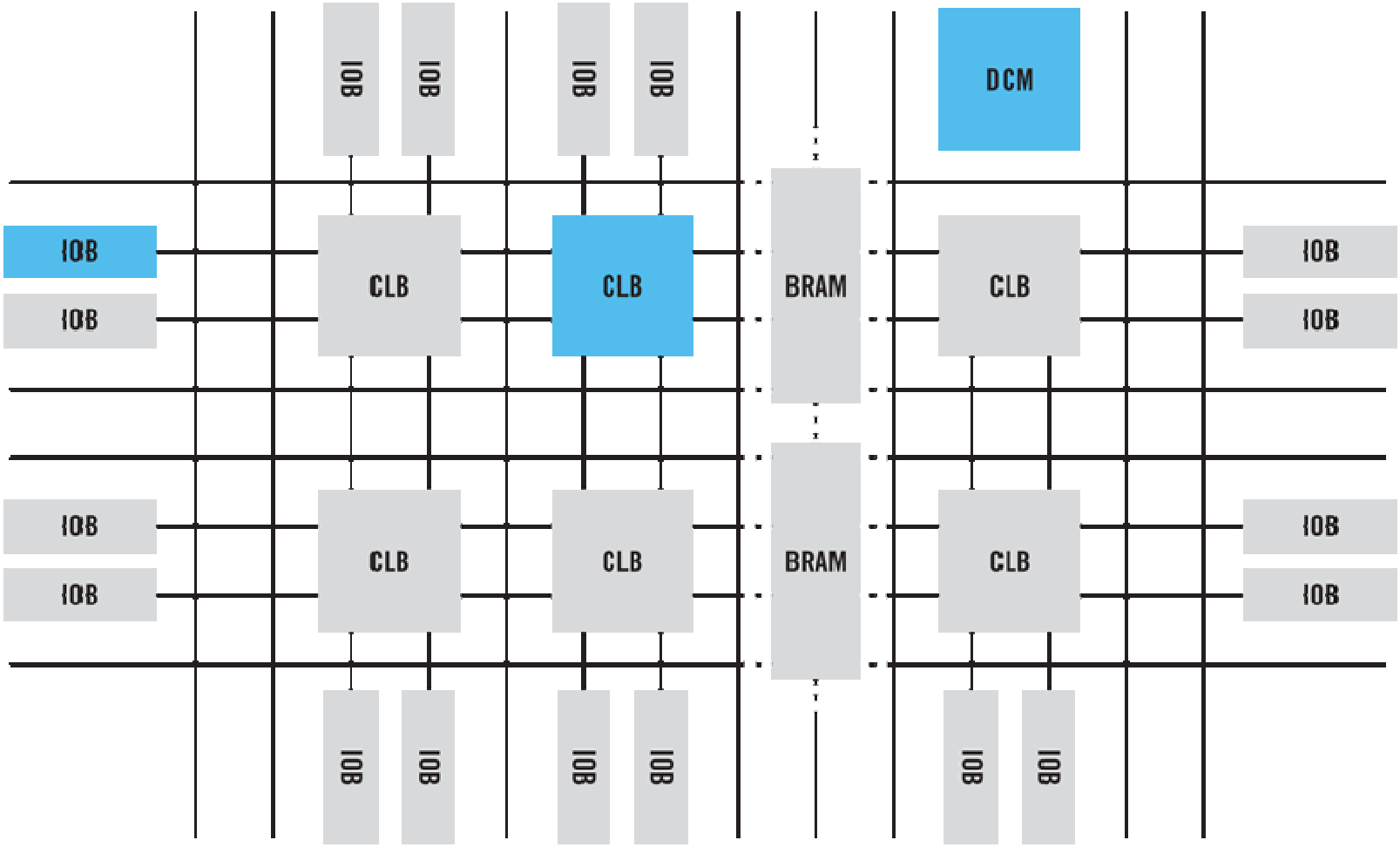
FPGA Vs ASIC

Characteritics	FPGA	ASIC
Performance	Medium	Very high
High volume unit cost	High	Low
Power consumption	High	Low
Time-to-market	Short design cycle(No layout, masks are needed)	Long design cycle
Density	Medium	Very High
Form Factor(Size)	Big	Small
Design flow complexity	Medium	Very High
Flexibility after manufacturing	High(Field Programmable remotely)	None
Complexity of test	Low	High
Re-usability	High (reprogramming)	None
Project Cycle	Predictable(No potential re-spins, wafer capacities etc.)	Hard to predict accurately

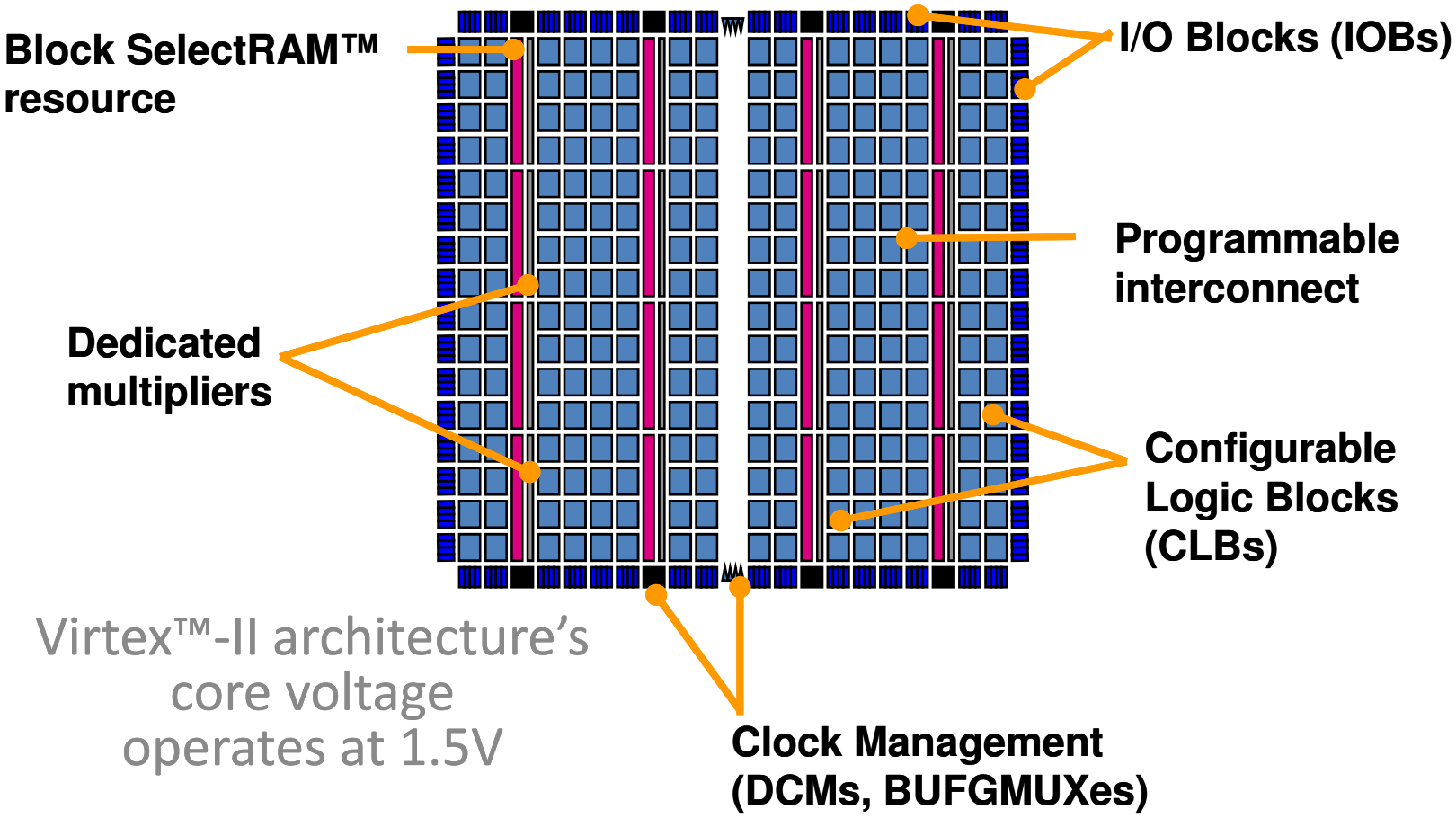
FPGA Vs ASIC Design Flow



FPGA Block Structure



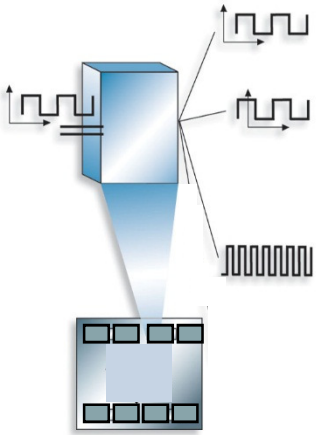
FPGA Virtex-II Architecture



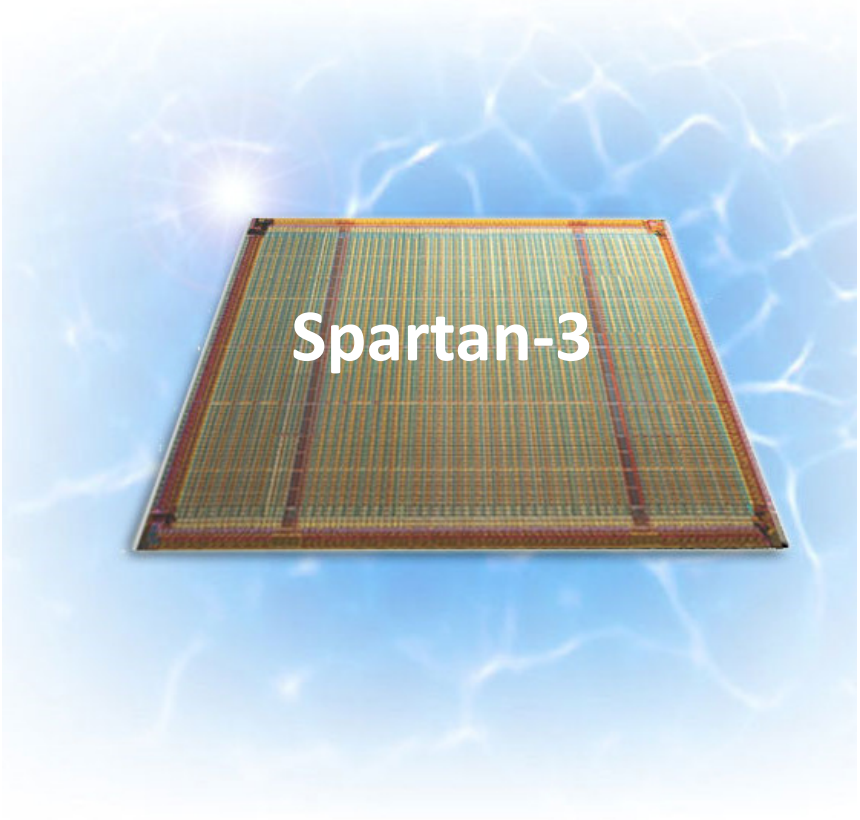
Virtex™-II architecture's core voltage operates at 1.5V

FPGA Spartan-III Solution

18x18 bit Embedded Pipelined Multipliers for efficient DSP



Up to eight on-chip Digital Clock Managers to support multiple system clocks



Configurable 18K Block RAMs + Distributed RAM

4 I/O Banks, Support for all I/O Standards including PCI, DDR333, RSDS, mini-LVDS

Xilinx Spartan-III CLB Structure

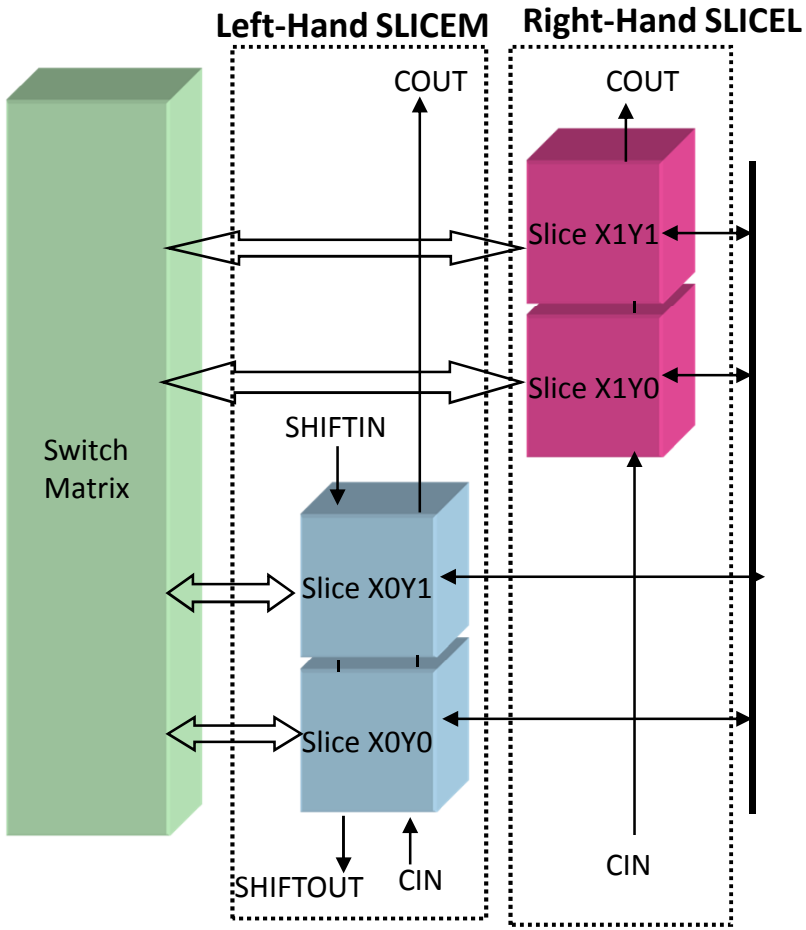
Each Spartan™-3 CLB

- contains four slices

Slices are grouped in pairs

- **Left-hand SLICEM (Memory)**
  - **LUTs can be configured as memory or SRL16**
- **Right-hand SLICEM (Logic)**
  - **LUTs can be configured as logic only**

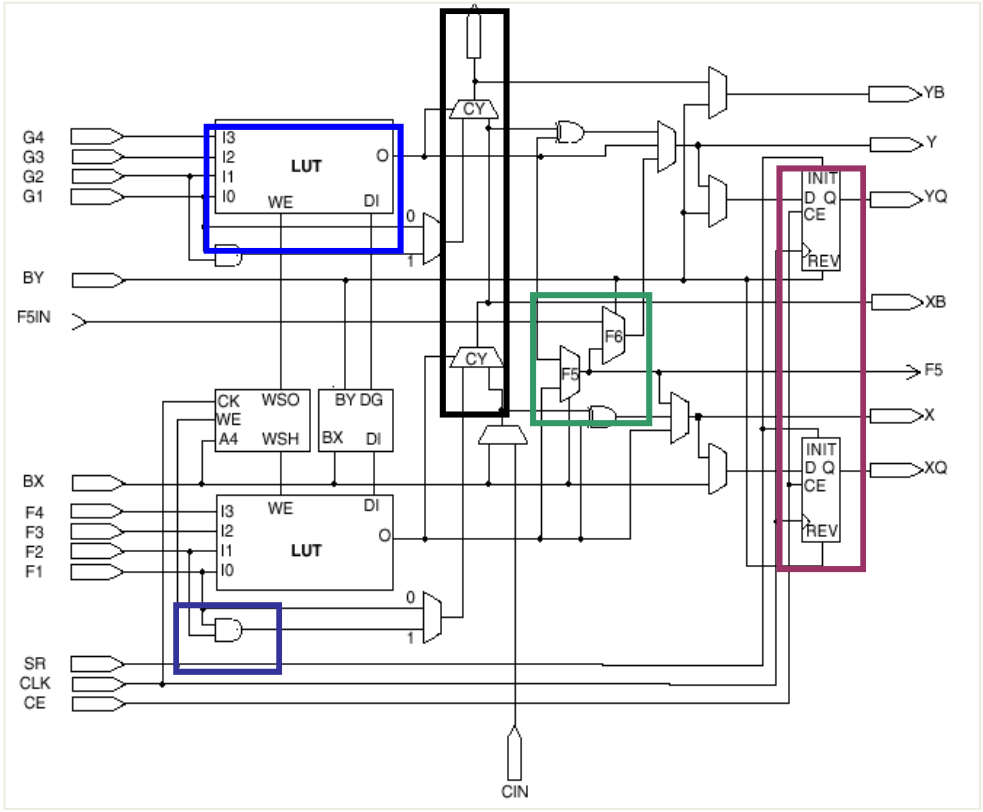
Note: A switch matrix provides access to general routing resources



Detailed CLB Slice Structure

The next few slides discuss the slice features:

- LUTs
- MUXF5, MUXF6, MUXF7, MUXF8 (only the F5 and F6 MUX are shown in this diagram)
- Carry Logic
- MULT_ANDs
- Sequential Elements



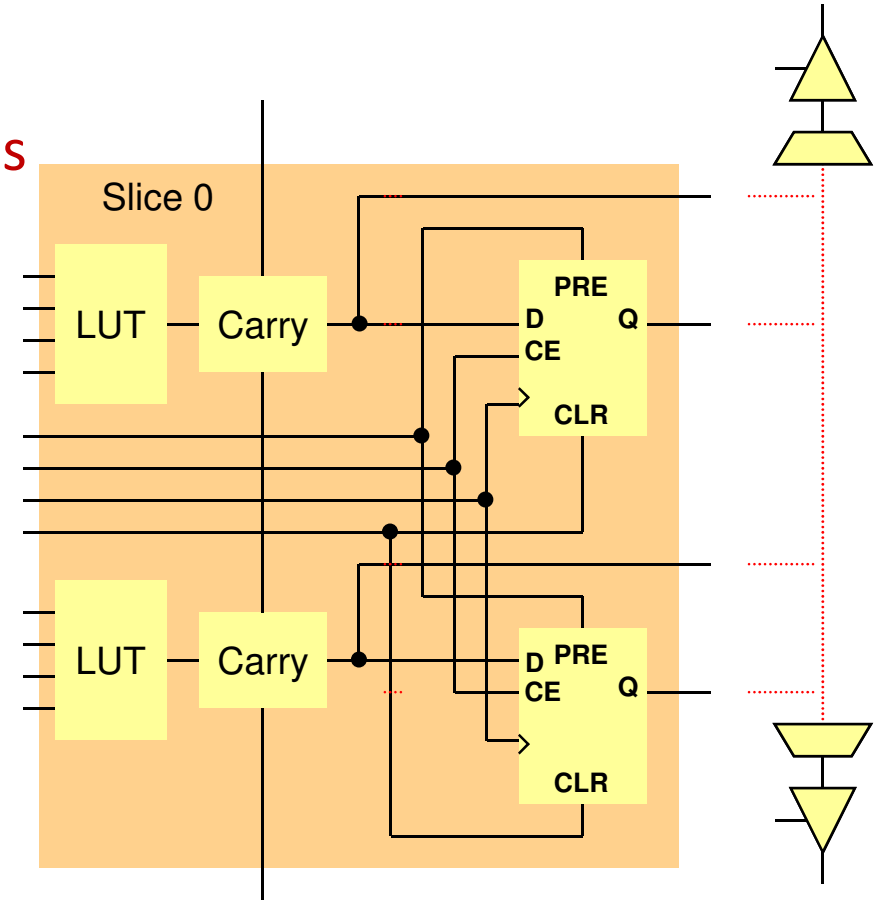
Simplified CLB Slice Structure

Each slice has four outputs:

- Two registered outputs, two non-registered outputs
- Two BUFTs associated with each CLB, accessible by all 16 CLB outputs

Carry logic runs vertically, up only:

- Two independent carry chains per CLB

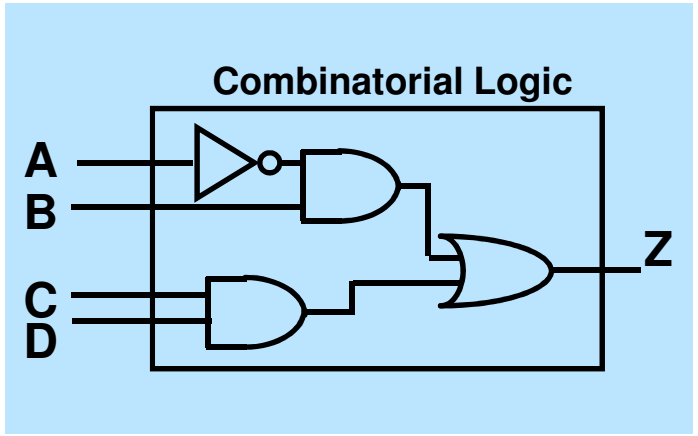


LUT (Look-Up Table) Functionality

Combinatorial logic is stored in Look-Up Tables (LUTs):

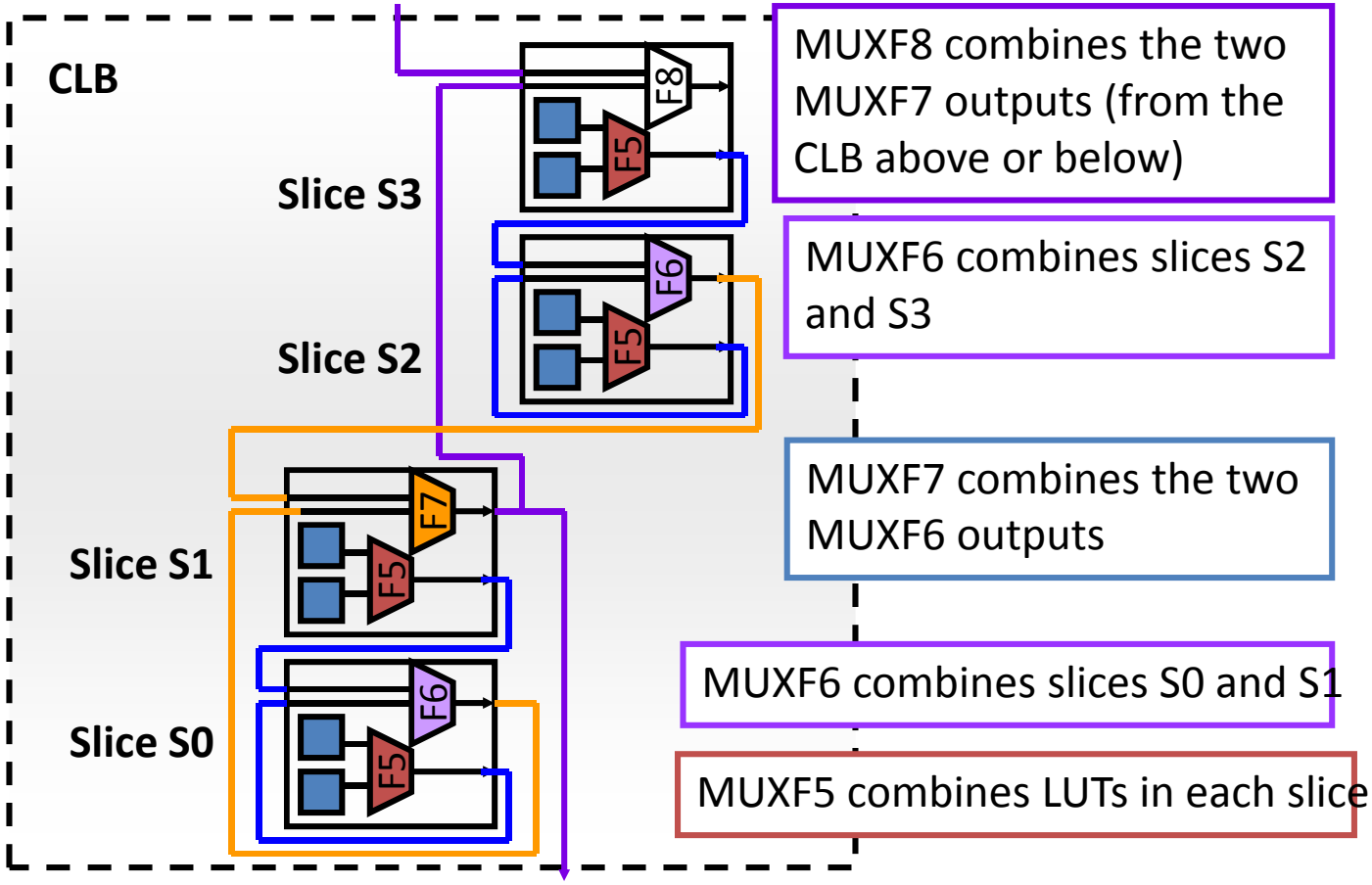
- Also called Function Generators (FGs)
- Capacity is limited by the number of inputs, not by the complexity

- **Delay through the LUT is constant**



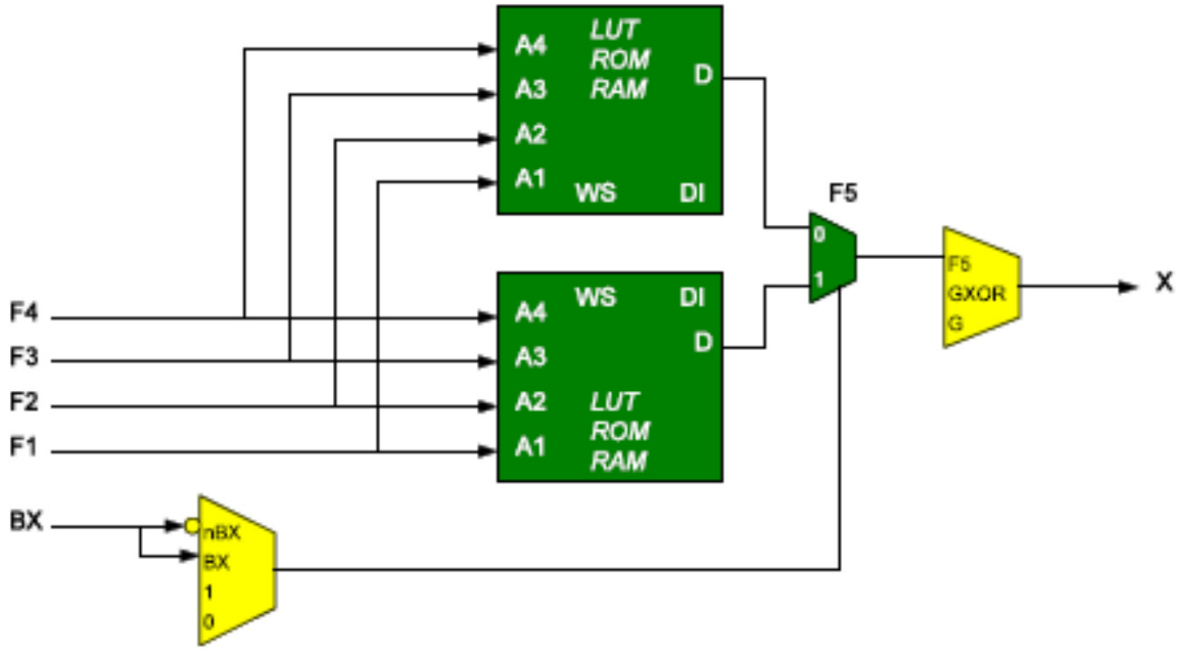
A	B	C	D	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
	.	.	.	
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Connecting Look-Up Tables



5-Input Functions implemented using two LUTs

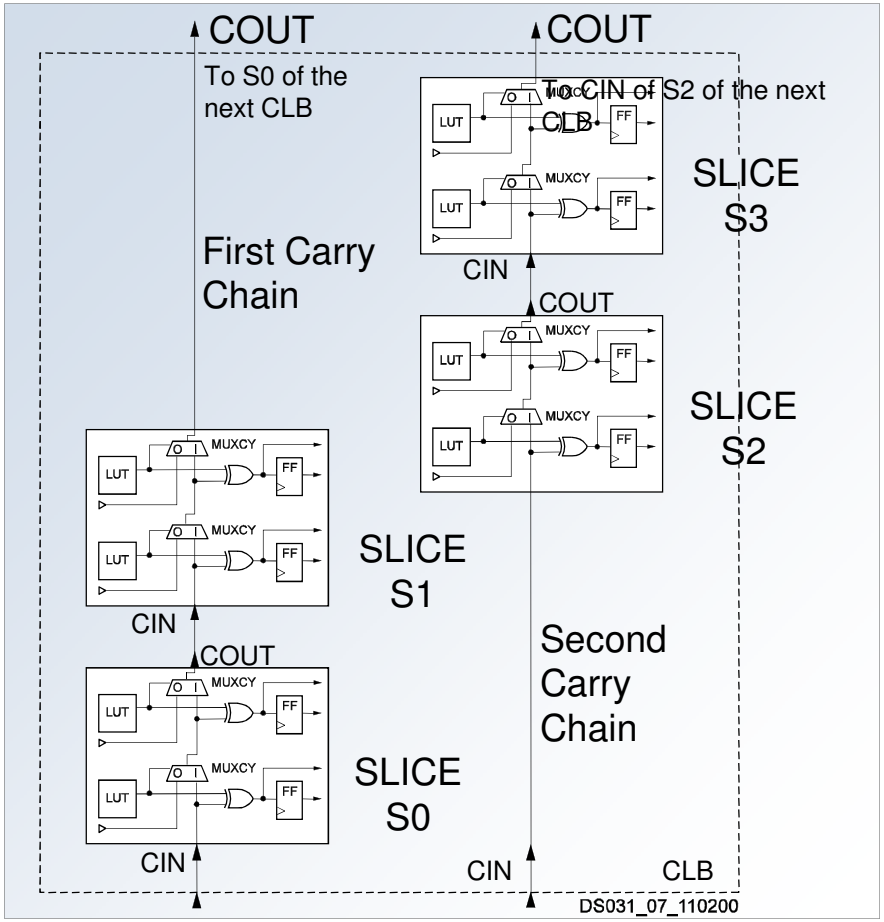
- One CLB Slice can implement any function of 5 inputs
- Logic function is partitioned between two LUTs
- F5 multiplexer selects LUT



Fast Carry Logic

Simple, fast, and complete arithmetic Logic:

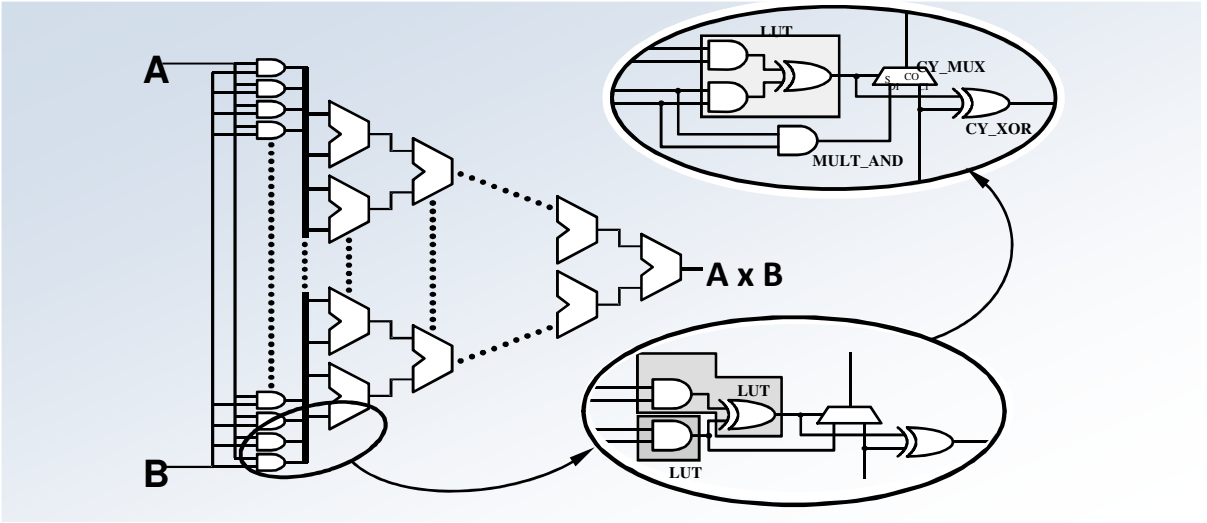
- Dedicated XOR gate for single-level sum completion
- Uses dedicated routing resources
- All synthesis tools can infer carry logic



MULT_AND Gate

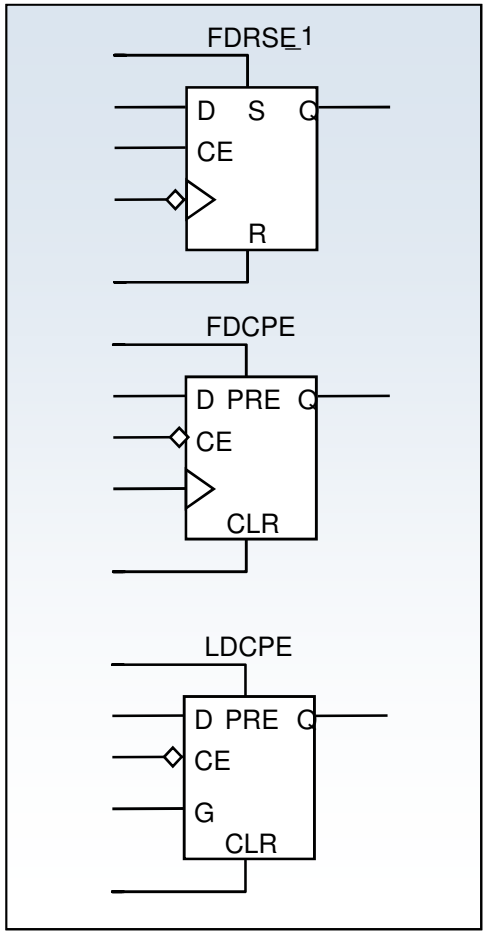
Highly efficient multiply and add implementation:

- Earlier FPGA architectures require two LUTs per bit to perform the multiplication and addition
- The MULT_AND gate enables an area reduction by performing the *multiply* and the *add* in one LUT per bit



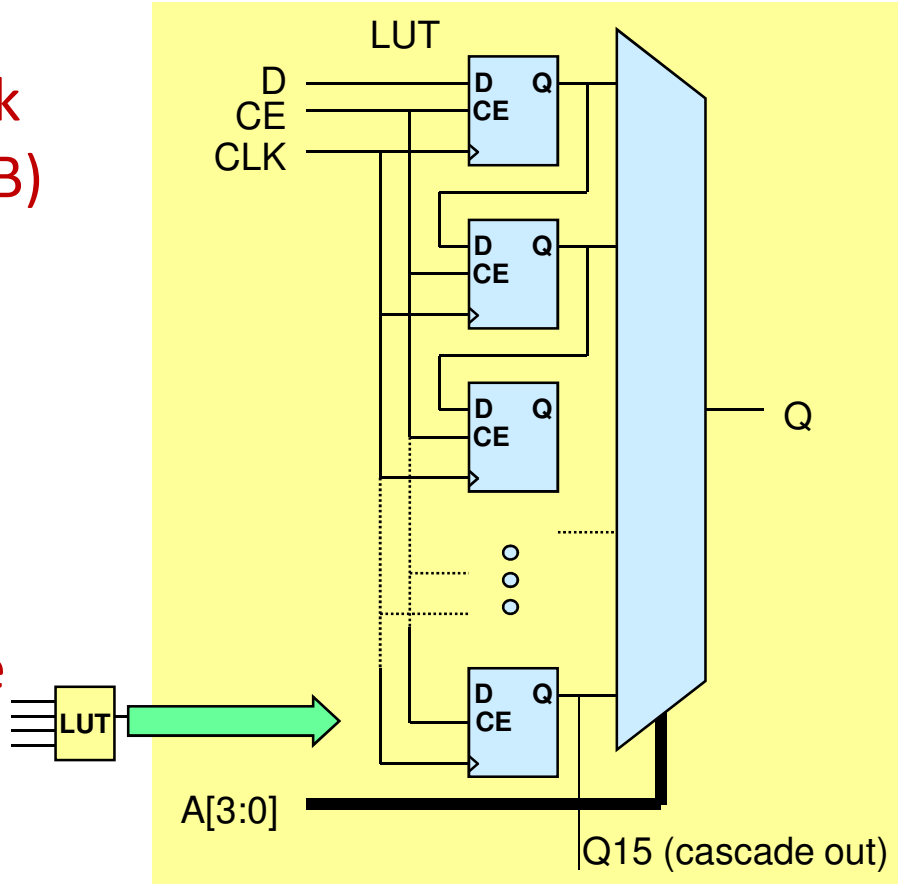
Flexible Sequential Elements

- Either flip-flops or latches
- Two in each slice; eight in each CLB
- Inputs come from LUTs or from an independent CLB input
- Separate set and reset controls
  - Can be synchronous or asynchronous
- All controls are shared within a slice
  - Control signals can be inverted locally within a slice



Shift Register LUT (SRL16CE)

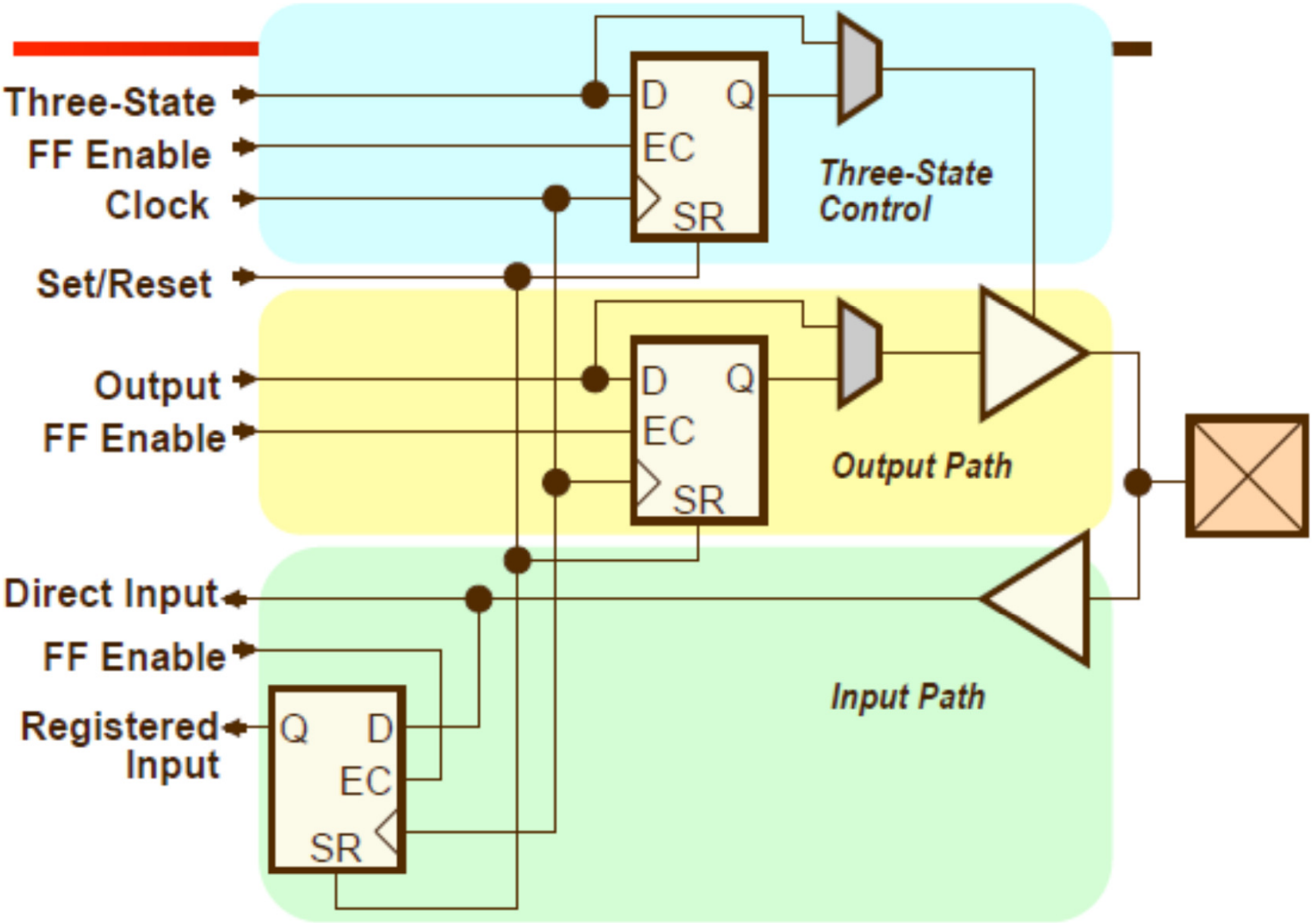
- Dynamically addressable serial shift registers
  - Maximum delay of 16 clock cycles per LUT (128 per CLB)
  - Cascade to other LUTs or CLBs for longer shift registers(Dedicated connection from Q15 to D input of the next SRL16CE)
  - Shift register length can be changed asynchronously by toggling address A



IOB Functionality

- IOB provides interface between the package pins and CLBs.
- Each IOB can work as uni- or bi-directional I/O
- Outputs can be forced into High Impedance
- Inputs and outputs can be registered
  - advised for high-performance I/O
- Inputs can be delayed

Basic I/O Block Structure

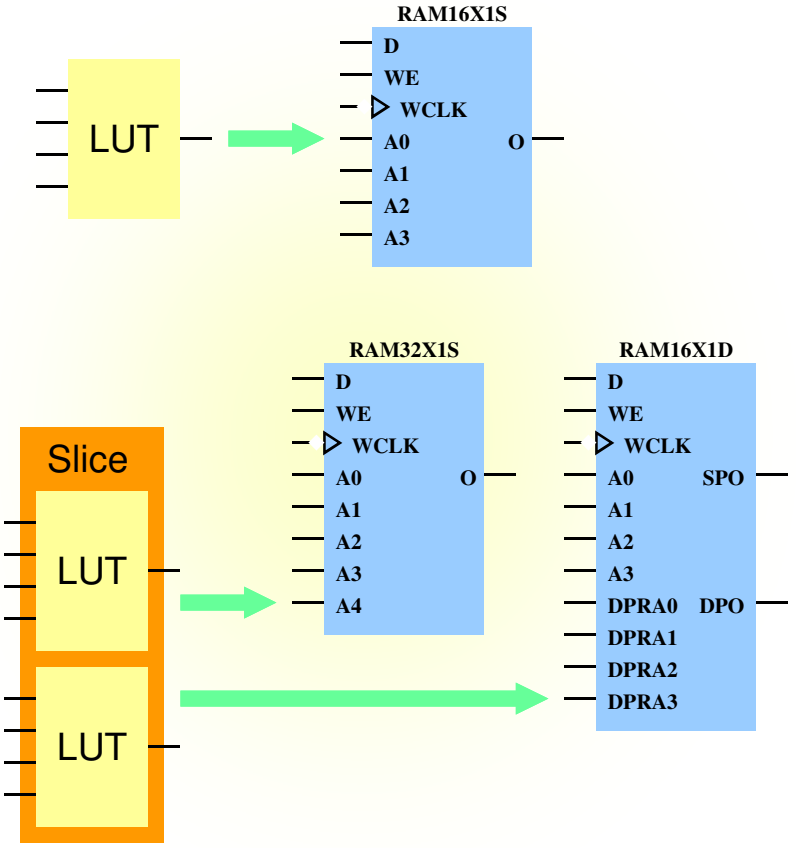


DCI(Digital Controlled Impedance)

- **DCI provides**
  - Output drivers that match the impedance of the traces
  - On-chip termination for receivers and transmitters
- **DCI advantages**
  - Improves signal integrity by eliminating stub reflections
  - Reduces board routing complexity and component count by eliminating external resistors
  - Eliminates the effects of temperature, voltage, and process variations by using an internal feedback circuit

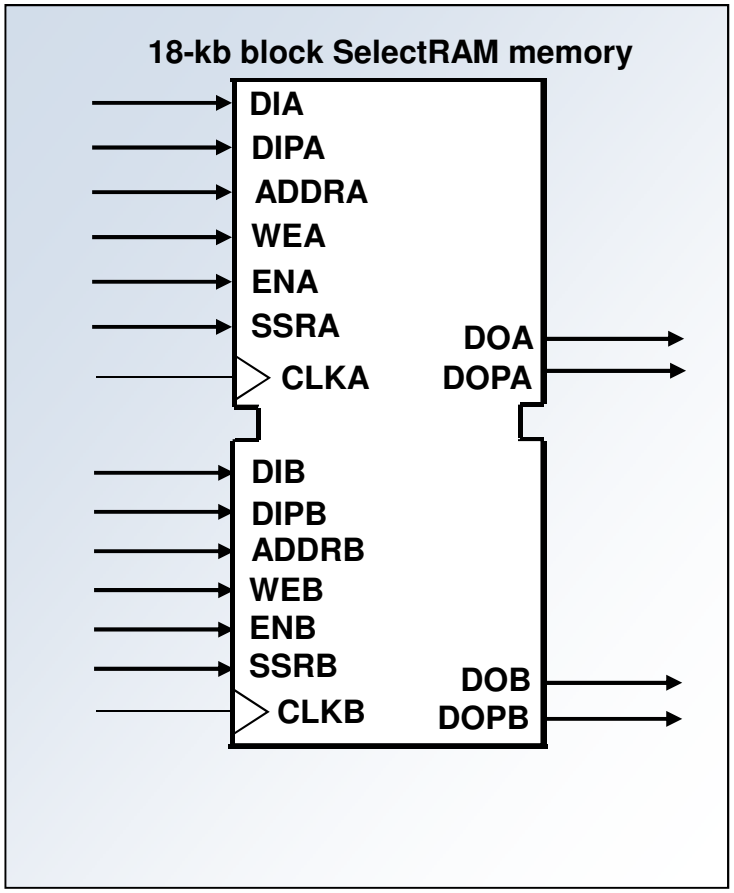
Distributed RAM

- Uses a LUT in a slice as memory
- Synchronous write
- Asynchronous read
  - Accompanying flip-flops can be used to create synchronous read
- RAM and ROM are initialized during configuration
  - Data can be written to RAM after configuration
- Emulated dual-port RAM
  - One read/write port
  - One read-only port



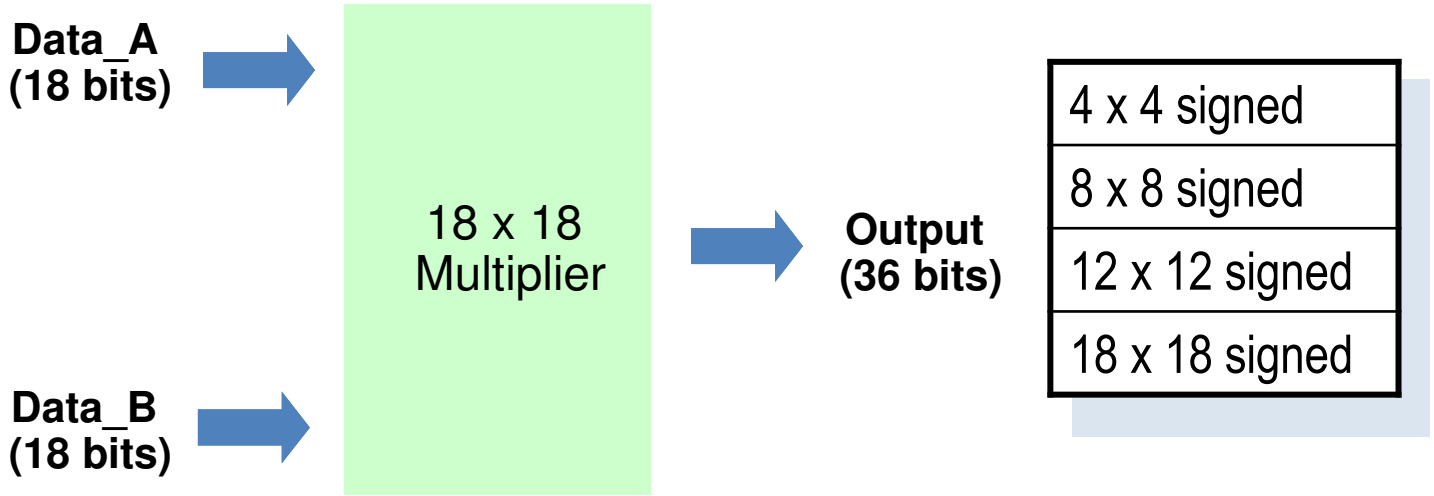
Block Selected RAM

- Up to 3.5 Mb of RAM in 18-kb blocks
  - Synchronous read and write
- True dual-port memory
  - Each port has synchronous read and write capability
  - Different clocks for each port
- Supports initial values
- Synchronous reset on output latches
- Supports parity bits
  - One parity bit per eight data bits



Dedicated Multipliers

- 18-bit twos complement signed operation
- Optimized to implement Multiply and Accumulate functions
- Multipliers are physically located next to block SelectRAM™ memory



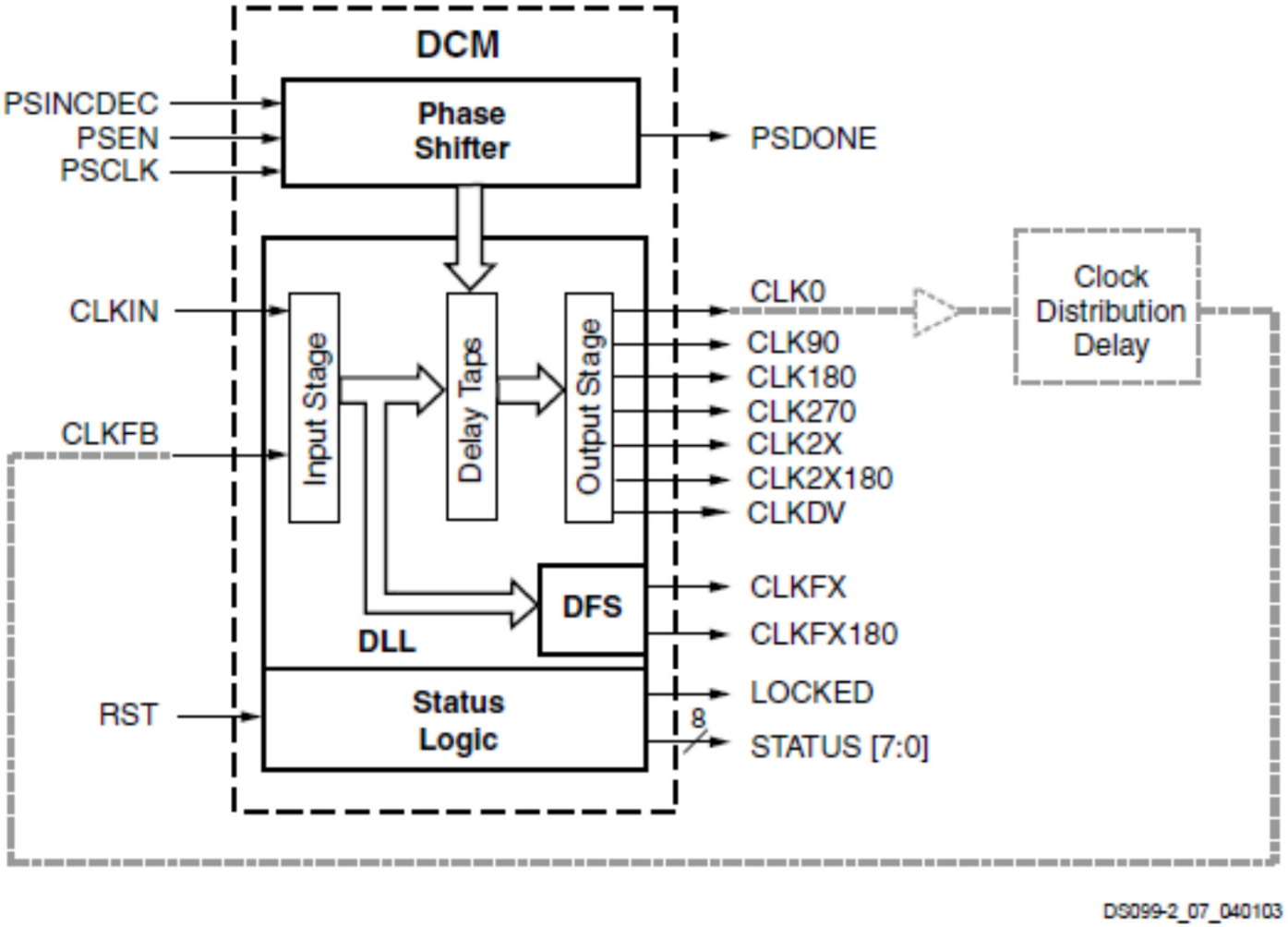
Global Clock Routing Resources

- **Sixteen dedicated global clock multiplexers**
  - Eight on the top-center of the die, eight on the bottom-center
  - Driven by a clock input pad, a DCM, or local routing
- **Global clock multiplexers provide the following:**
  - Traditional clock buffer (BUFG) function
  - Global clock enable capability (BUFGCE)
  - Glitch-free switching between clock signals (BUFGMUX)
- **Up to eight clock nets can be used in each clock region of the device**
  - Each device contains four or more clock regions

Digital Clock Manager(Clock-skew elimination, Frequency synthesis, Phase shifting)

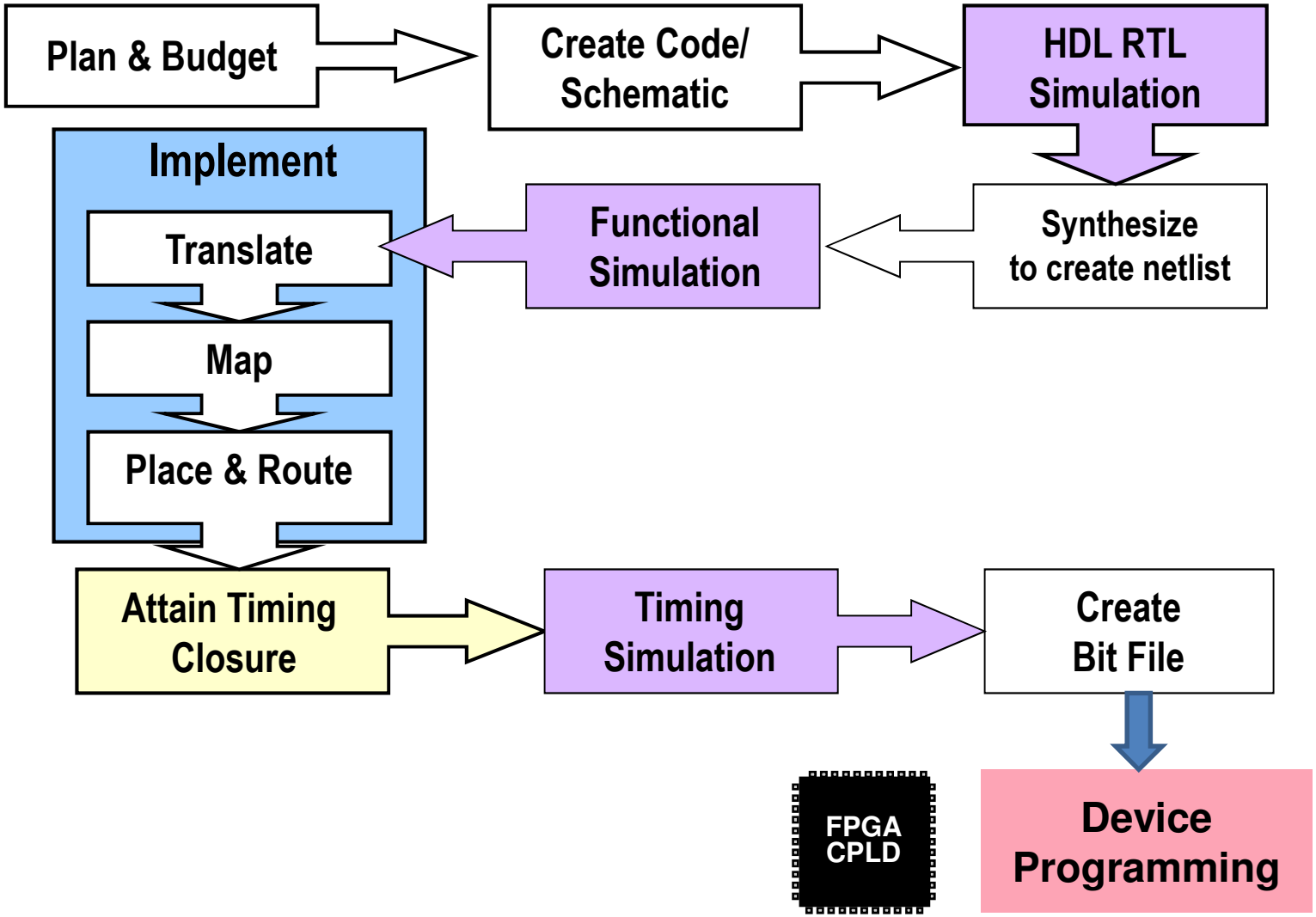
- Up to twelve DCMs per device
  - Located on the top and bottom edges of the die
  - Driven by clock input pads
- DCMs provide the following:
  - Delay-Locked Loop (DLL)
  - Digital Frequency Synthesizer (DFS)
  - Digital Phase Shifter (DPS)
- Up to four outputs of each DCM can drive onto global clock buffers
  - All DCM outputs can drive general routing

Digital Clock Manager(Clock Management)



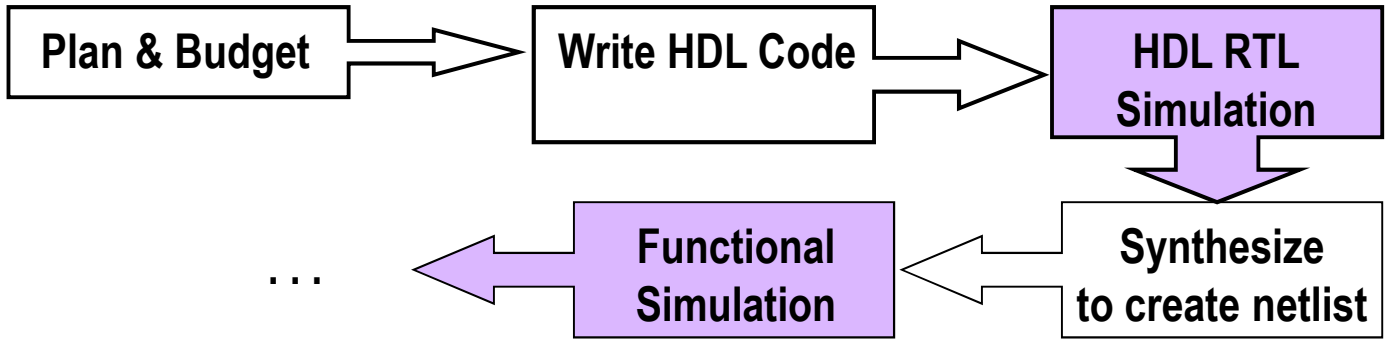
DCM functional blocks and signals

Xilinx FPGA Design Flow



Design Entry

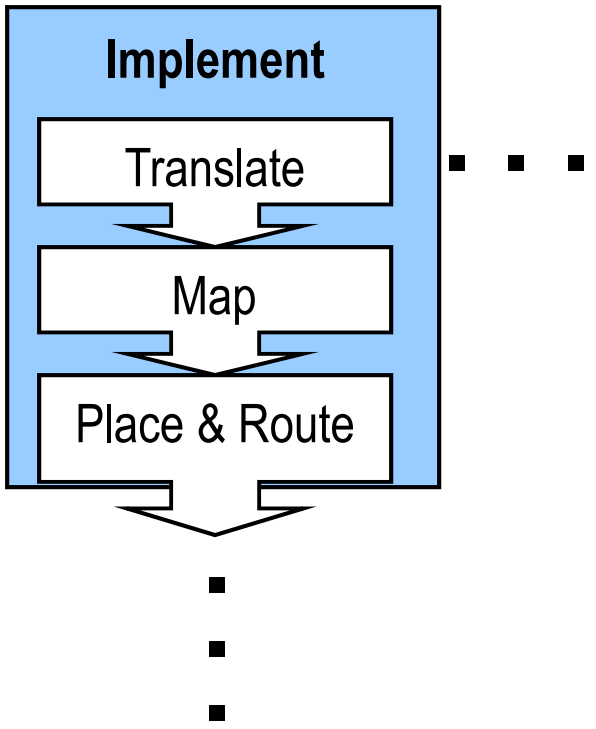
- **Plan and budget**
- **Two Design-entry methods: HDL or Schematics Entry**
  - Write HDL Code for the design
- **Whichever method you use, you will need a tool to generate an EDIF or NGC netlist to bring into the Xilinx implementation tools**
  - Popular synthesis tools: Synplify, Vivado, ISE and XST
- **Simulate the design to ensure that it works as expected!**



FPGA Implementation

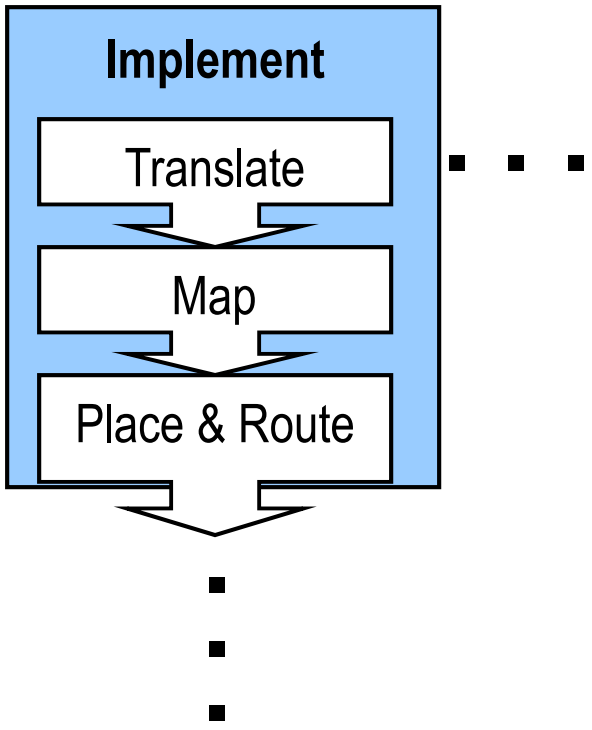
I. Translate : Translate step checks the design and ensures that netlist is consistent with chosen architecture. Translate also checks User-defined Constraint file, for any inconsistencies.

II. Map : Calculates & allocates Physical Combinational Logic Blocks (CLB) & Input Output Block (IOB) Components in Targeted Device, to Logic Element symbols in Netlist that is generated during Translation Process.



FPGA Implementation

III. Place & Route (PAR): Places CLBs into logical position and utilizes the routing resources on target device, to connect logic cells on Xilinx Product such that desired Timing Specification are met.



Selection Consideration for Xilinx Device

1. Speed Requirement : If the Maximum Frequency requirement for the design is not met with a articular Xilinx device, choose a Xilinx device of higher Speed Grade (or) switch to next generation Xilinx Product Family.

Target Device	Maximum Frequency (of Multiplier design)
Spartan :- xc3s500e-4-pq208	119 MHz
Spartan :- xc3s500e-5-pq208	137 MHz
Virtex :- xc5vlx30-2-ff324	259.639 MHz
Virtex :-xc5vlx30-3-ff324	295.683 MHz

Selection Consideration for Xilinx Device

Note : i) In Target Device Name, **-2/-3/-4/-5** indicates **Speed Grade** available for the device. (Higher the indicated Speed Grade number, higher is the maximum frequency obtained through that device)

ii) Virtex and Spartan Product Family are indicated by text "**vlx**" and "**s**" respectively, in Xilinx Device name.

2. Logic Density: Choose the device which has Gate Count and Macro-Cells to meet the Logic Density of the design.

*For e.g. Following is Device Utilization Summary of Multiplier Implementation on xc3s**500**e-5-pq208 device(Here, number **500** in device name indicates that this device has **500k** gates).*

Selection Consideration for Xilinx Device

Device utilization summary:

Number of Slices:	*198*	*out of 4656*	*4%*
Number of Slice Flip Flops:	*165*	*out of 9312*	*1%*
Number of 4 input LUTs:	*379*	*out of 9312*	*4%*
Number of IOs:	*133*		
Number of bonded IOBs:	*133*	*out of 158*	*84%*
Number of GCLKs:	*1*	*out of 24*	*4%*

(Small Designs like Multiplier can be implemented on CPLD Series having minimum number of gates, amongst available Xilinx Product families)

3. Package Type and Number of IO Pins: Based on Number of Input/Output Ports of the design, decide on device having sufficient number of IO Pins.

*For e.g. 32-bit Multiplier Design cannot be implemented using device 3s500e**cp132**-5, since, this **Chip-Scale Package** only has **92** I/O pins, whereas Multiplier design here has 133 I/O ports.*

*xc3s500e-5-**pq208** device can be used for Multiplier Design because this **Plastic Quad Flat Package** has **158** I/O Pins.*

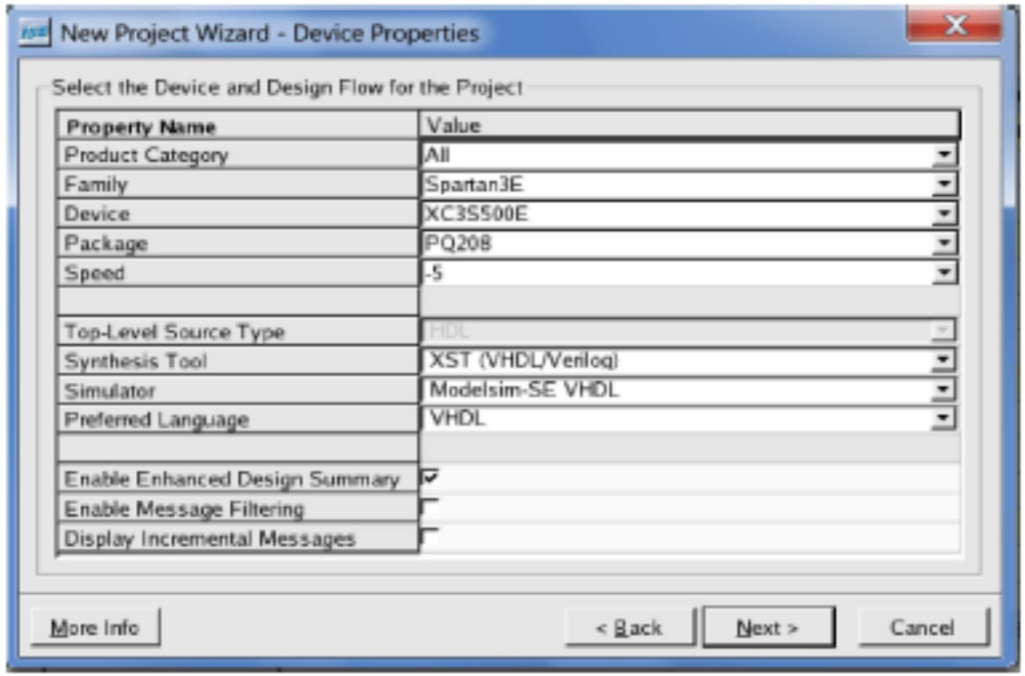
Creating new ISE Project

1. From Project Navigator, select File > New Project.

(Specify the *Project Name*, *Project Location*; and select *Top Level Source Type* as HDL)

2. Click *Next* & describe **Device Properties** in New Project Wizard. *(Based on the Selection Design Consideration mentioned earlier, select Xilinx Product Family/ Device Type/ Package Type & Speed Grade)*

Following image window indicates the Device Properties selected for Shift-Add Multiplier Implementation :-



3. Click *Next* & then click *Add Source* tab in New Project Wizard, to browse and add existing HDL source files to current ISE Project

Synthesizing Design

Synthesis Tool Functionality :

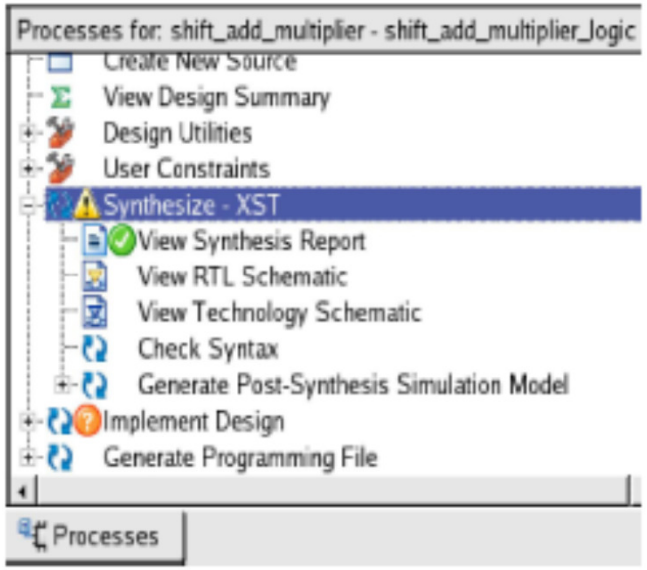
During synthesis, HDL files are translated into gates and optimized for the target architecture.

Thus, XST Synthesis tool uses design's HDL code and generates a supported netlist type (NGC) for Xilinx implementation tools, by performing following general steps :-

- **Analyze / Check Syntax** of the Source Code.
- **Compile** : Translates and optimizes the HDL code into a set of components that the synthesis tool can recognize.
- **Map** : Translates the components from the compile stage into the target technology's primitive components from UNISIM Library.

Steps to Synthesize HDL Design :

- Select top-level HDL design in the *Sources* window.
- To set Synthesis options, right-click *Synthesize - XST* in the *Processes* window; select *Properties* to display the *Process Properties* dialog box.
- With the top-level source file selected, right-click *Synthesize - XST* in the Processes window & select *Run* option.
- Synthesis Report file is stored with extension <project_name>.syr in ISE Project Directory.



Synthesizing Design – *Understanding Synthesis Options*

[Source : Xilinx Synthesis and Simulation Design Guide]

Synthesis options enables designer to modify the behaviour of the synthesis tool, to make optimizations according to the needs of design.

- ***Optimization Effort :*** constraint allows to choose synthesis optimization level as *Normal* or *High* optimization.
- ***Optimization Goal :*** constraint allows to choose synthesis optimization strategy as *Speed* or *Area*.

For e.g. In Shift Add Multiplier implementation, when Optimization Goal is selected as Area, though Number of Slices and LUTs utilized reduces by 1%, but the maximum frequency for the design also reduces by 8MHz, in comparison to Synthesis done with default Optimization Goal (i.e. Speed).

- ***Use Synthesis Constraint File :*** option allows to include (or) exclude .xcf Constraint File ,during synthesis process.
- ***Keep Hierarchy*** : is a synthesis and implementation constraint. If hierarchy is maintained during Synthesis, the Implementation tools will use this constraint to preserve the hierarchy throughout the implementation process and allow a simulation netlist to be created with the desired hierarchy. *Though preserving the hierarchy gives the advantage of fast processing; but, merging the hierarchy blocks improves the fitting results (i.e. fewer device macrocells & better frequency).*
- ***Global Optimization Goal*** : allows to optimize speed in different regions (register to register, inpad to register, register to outpad, and inpad to outpad) of the design.
- ***Write Timing Constraints*** : enables or disables propagation of timing constraints to the NGC file, which will be used during place and route, as well as synthesis optimization.
- ***Slice Utilization Ratio*** : defines the area size in absolute number or percent of total number of slices that XST must not exceed, during timing optimization.

Synthesizing Design– *Generating Post-Synthesis Simulation Model*

Requirement & Generation of Post-Synthesis Simulation Model :

To verify whether the correct functionality of the design is retained, after synthesizing it into netlist; HDL Design's equivalent simulation model can be generated, by clicking on ***Generate Post-Synthesis Simulation Model*** *option within Synthesis-XST process list.* (Netlist simulation model is generated in **netgen/synthesis** directory)

Post-Synthesis Simulation using Modelsim :

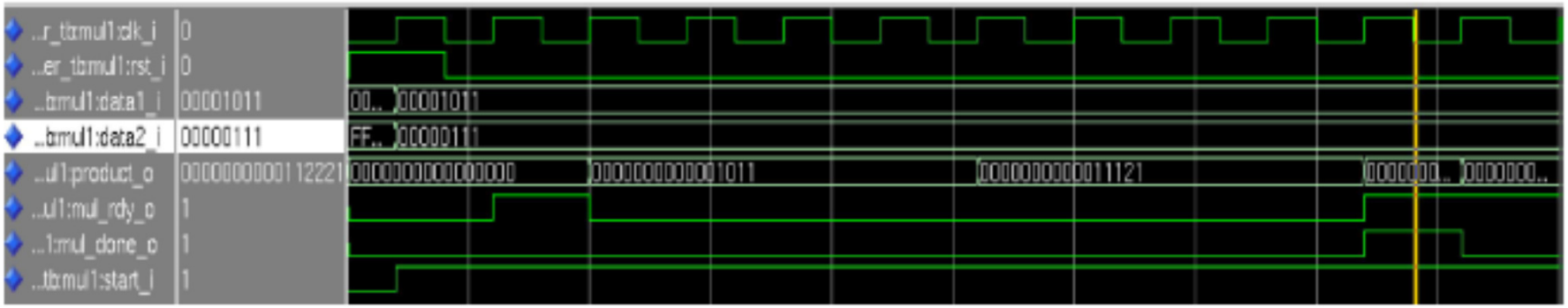
Post-Synthesis Simulation Model can be compiled & simulated using the same HDL testbench, that was used for HDL behavioral code verification.

In Modelsim, after creating a new project for Post Synthesis Simulation, include Netlist (_synthesis.vhd) from <ise_project>/netgen/synthesis directory. This netlist file is compiled along with testbench, instead of HDL behavioral Code being compiled.

Post-Synthesis Simulation Result :

For the generated Post-Synthesis Simulation Model, no standard delay file (.sdf file) is back-annotated during simulation (*Thus, UNISIM Library primitives, included in the synthesis generated netlist, do not have any delay associated with it*)

Therefore, expected Post-Synthesis Simulation result is same as Functional Verification result of HDL Design, as can be seen in following Post-Synthesis simulation waveform for Shift-Add Multiplier design :-



Specifying User Constraints

Need for Setting Constraints :

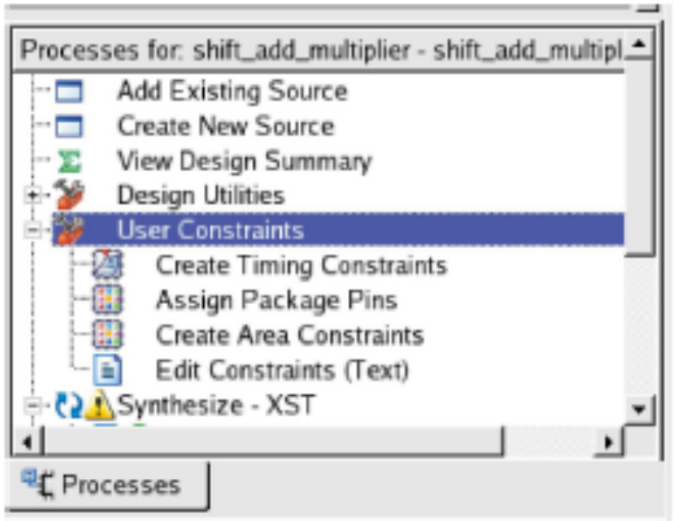
For Design to meet desired Area/Frequency Specification on FPGA, it is required to tell the Implementation Tool for what performance it should optimize the design implementation processes like Map, Place & Route. Thus, User Defined Constraints allows to specify desired Clock period/ Pad to Setup/Clock to Pad delay & assign areas to hierarchical blocks of logic.

Physical User Constraint also allows to allocate HDL design's I/O signals to specific package pins.

Adding Constraints to Design :

User can specify Constraints for ISE Design Project , either through GUI by double-clicking on following options available within *User Constraints* label in *Processes* Window:-

(i) Create Timing Constraints
(ii) Assign Package Pins
(iii) Create Area Constraints



(or) the User can specify constraints in .**ucf** file, through any text editor.

While opening Constraint Editor window, Translate step runs automatically because implementation stage must see the netlist before it can offer the user the chance to constraint sections of design.

Translating Design

Translate Process Functionality :

During translation, the NGDBuild program performs the following functions :-

- Converts input design netlists and writes results to a single merged NGD netlist. The merged netlist describes the logic in the design as well as any location and timing constraints.
- Performs timing specification and logical design rule checks.
- Adds constraints from the User Constraints File (UCF) to the merged netlist.

Steps to Translate the Design :

- Translate Process gets automatically executed while opening Consraint Editor GUI Window (or) User can right-click *Translate* option in *Processes* Window and select *Run* option.
- To set *Translate Properties*, right-click *Translate* in the *Processes* window; select *Properties* to display the Process Properties dialog box.
- Synthesis Report file is stored with extension <project_name>.**bld** in ISE Project Directory.

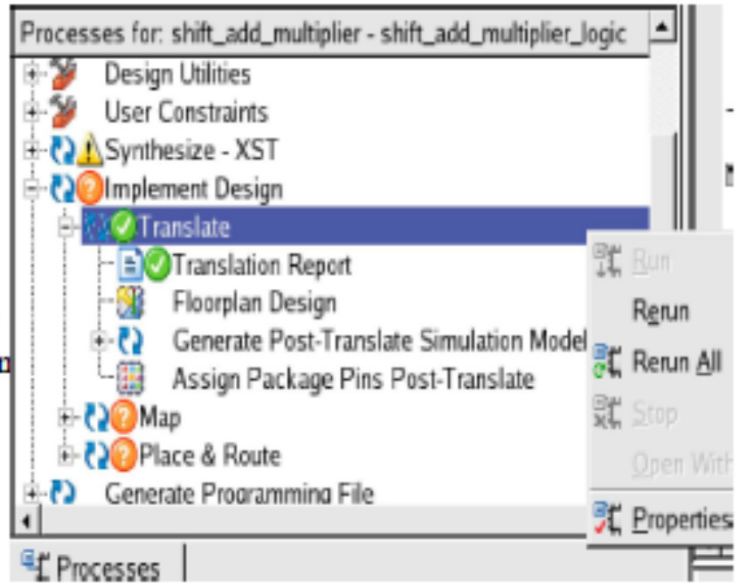
Translate Process File Types :

Translate Process uses following files as input :-

- NGC netlist file from Synthesis Process.
- UCF constraint file containing timing and layout constraints.

Translate Process creates following files as output :-

- NGD file, containing logical description of the design, expressed in terms of lower level Xilinx Primitives, with constraint applied to design
- BLD Report file shows following error in design or UCF file :-
  - Missing or untranslatable hierarchical blocks
  - Invalid or incomplete timing constraints
  - Output contention, loadless outputs, and sourceless inputs



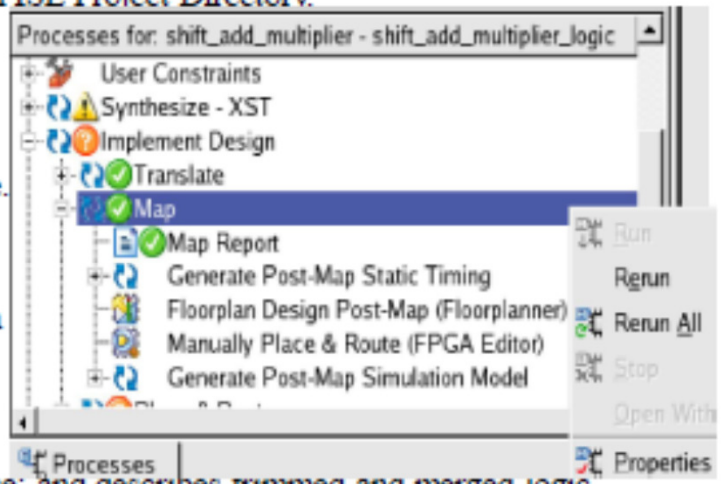
Mapping the Design

MAP Process Functionality :

- Allocates CLB and IOB resources for all basic logic elements in the design.
- Processes all location and timing constraints, performs target device optimizations, and runs a design rule check on the resulting mapped netlist.

Steps to Map the Design :

- To set Map Process Properties, right-click *Map* in the *Processes* window; select *Properties* to display the Process Properties dialog box.
- Right-click *Map* label in *Processes* Window and select *Run* option.
- Synthesis Report file is stored with extension <project_name>.**mrp** in ISE Project Directory.



Map Process File Types :

MAP Process uses NGD file, created during Translate Process, as Input file.

MAP Process creates following files as output :-

- NCD (Native Circuit Description) file containing physical description of design in terms of the components in the target Xilinx device.
- PCF (Physical Constraints File) contains constraints specified during design entry expressed in terms of physical elements.
- MRP (MAP Report File) confirms the resources used within the device; and describes trimmed and merged logic. Detailed Report also describes exactly where each portion of the design is located in the device.

Mapping Design – *Analyzing MAP Report*

Following is the sectional details of MAP Report (*.mrp*) file :-

- *Design Summary* - Summarizes the mapper run, showing the number of errors and warnings, and how many of the resources in the target device are used by the mapped design.

```
Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Logic Utilization:
  Number of Slice Flip Flops:          163 out of   9,312    1%
  Number of 4 input LUTs:              379 out of   9,312    4%
Logic Distribution:
  Number of occupied Slices:                        202 out of   4,656    4%
    Number of Slices containing only related logic:     202 out of     202  100%
    Number of Slices containing unrelated logic:          0 out of     202    0%
      *See NOTES below for an explanation of the effects of unrelated logic
Total Number of 4 input LUTs:          379 out of   9,312    4%
  Number of bonded IOBs:               133 out of     158   84%
    IOB Flip Flops:                      2
  Number of GCLKs:                       1 out of      24    4%

Total equivalent gate count for design:  4,377
Additional JTAG gate count for IOBs:  6,384
```

```
Section 4 - Removed Logic Summary
---------------------------------
   2 block(s) optimized away

Section 5 - Removed Logic
-------------------------

Optimized Block(s):
TYPE            BLOCK
GND             XST_GND
VCC             XST_VCC
```

- *Removed Logic* - Describes in detail all logic (design components and nets) removed for the following reasons, from the input NGD file when the design is mapped :-

  - The design uses only part of the logic in a library macro.
  - The design has been mapped even though it is not yet complete.
  - The mapper has optimized the design logic.
  - Unused logic has been created in error during schematic entry.

Mapping Design – *Analyzing MAP Report*

- *IOB Properties* - Lists each IOB to which the user has supplied constraints along with the applicable constraints.

```
Section 6 - IOB Properties
--------------------------

+-------------------------------------------------------------------------------------------------------------------------+
| IOB Name    | IOB Type          | Direction | IO Standard | Drive    | Slew | Reg (s)      | Resistor | IBUF/IFD |
|             |                   |           |             | Strength | Rate |              |          | Delay    |
+-------------------------------------------------------------------------------------------------------------------------+
| clk_i       | IBUF              | INPUT     | LVCMOS25    |          |      |              |          | 0 / 0    |
| data1_i<0>  | IBUF              | INPUT     | LVCMOS25    |          |      |              |          | 0 / 0    |
| data1_i<1>  | IBUF              | INPUT     | LVCMOS25    |          |      |              |          | 0 / 0    |
| data1_i<2>  | IBUF              | INPUT     | LVCMOS25    |          |      |              |          | 0 / 0    |
```

- *Timing Report* - This section, produced with *Perform Timing driven packing & Placement* option, shows information on timing constraints considered during the MAP run.

```
Section 11 - Timing Report
--------------------------
INFO:Timing:3284 - This timing report was generated using estimated delay
   information.  For accurate numbers, please refer to the post Place and Route
   timing report.
Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

------------------------------------------------------------------------------------------------------
  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
                                            |         |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------------------------------------
* TS_clk_i = PERIOD TIMEGRP "clk_i" 7.5 ns  | SETUP   |    -0.091ns|     7.591ns|       1|          91
  HIGH 50%                                  | HOLD    |     1.263ns|            |       0|           0
------------------------------------------------------------------------------------------------------


1 constraint not met.
```

Placing and Routing the Design

Place And Route Process Functionality :-

- During placement, PAR places components into sites based on factors such as constraints, the length of connections, and the available routing resources.
- After placing the design, the router performs a converging procedure for a solution that routes the design to completion and meets timing constraints.

Steps to Place and Route the Design :-

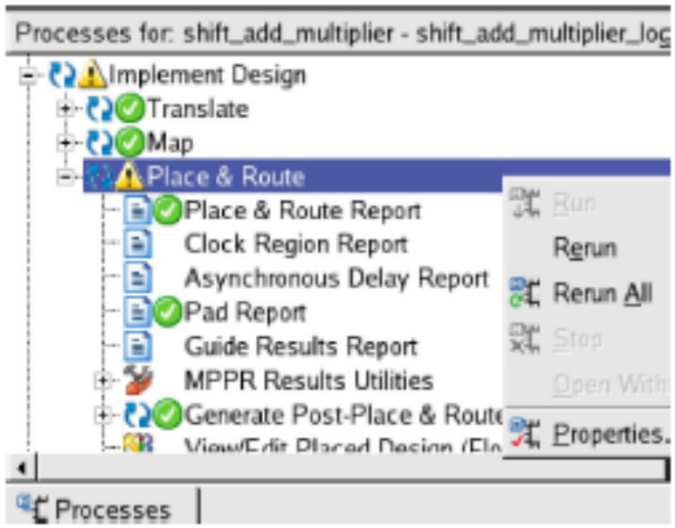
- To execute PAR in the *Processes* tab, right-click *Place & Route* under the *Implement Design* process group, and select *Run* option.
- To set *Place and Route* Properties, right-click *Place & Route* in the *Processes* window; select *Properties* to display the *Process Properties* dialog box.

PAR Process File Types :-

PAR Process uses Mapped Design (NCD) File and Physical Constraint (PCF) file created during MAP Process, as Input File.

PAR Process creates following files as output :-

- Placed and Routed NCD Design File
- PAR Report File, including summary information of all placement and routing iterations.



Placing and Routing Design – *Analyzing PAR Report*

Following is the sectional details of PAR Report (*. par*) file :-

- ***Design Summary*** - Provides a breakdown of the resources in the design and includes the Device Utilization Summary

```
Design Summary Report:

 Number of External IOBs                          133 out of 158    84%

   Number of External Input IOBs                  67

      Number of External Input IBUFs              67
        Number of LOCed External Input IBUFs      67 out of 67    100%


   Number of External Output IOBs                 66

      Number of External Output IOBs              66
        Number of LOCed External Output IOBs      66 out of 66    100%


   Number of External Bidir IOBs                   0


   Number of BUFGMUXs                          1 out of 24      4%
   Number of Slices                          202 out of 4656    4%
      Number of SLICEMs                        0 out of 2328    0%
```

- ***Clock Report*** - Lists all clocks in the design and provides information on the routing resources, number of fanout, maximum net skew for each clock, and the maximum delay. The locked column in the clock table indicates whether the clock driver (BUFGMUX) is assigned to a particular site or left floating..

Clock Net	Resource	Locked	Fanout	Net Skew(ns)	Max Delay(ns)
clk_i_BUFGP	BUFGMUX_X2Y11	No	153	0.055	0.159

Placing and *Routing* Design – *Analyzing PAR Report*

- ***Delay Summary Report*** - Summarizes the connection and pin delays for the design.

```
The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        1.211
   The MAXIMUM PIN DELAY IS:                              4.125
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   3.290

   Listing Pin Delays by value: (nsec)

    d < 1.00   < d < 2.00   < d < 3.00   < d < 4.00   < d < 5.00   d >= 5.00
   ---------   ---------   ---------   ---------   ---------   ---------
         838         648         286          46           3           0
```

- ***Timing Score*** - Lists information on timing constraints contained in the input PCF, including how many timing constraints were met.

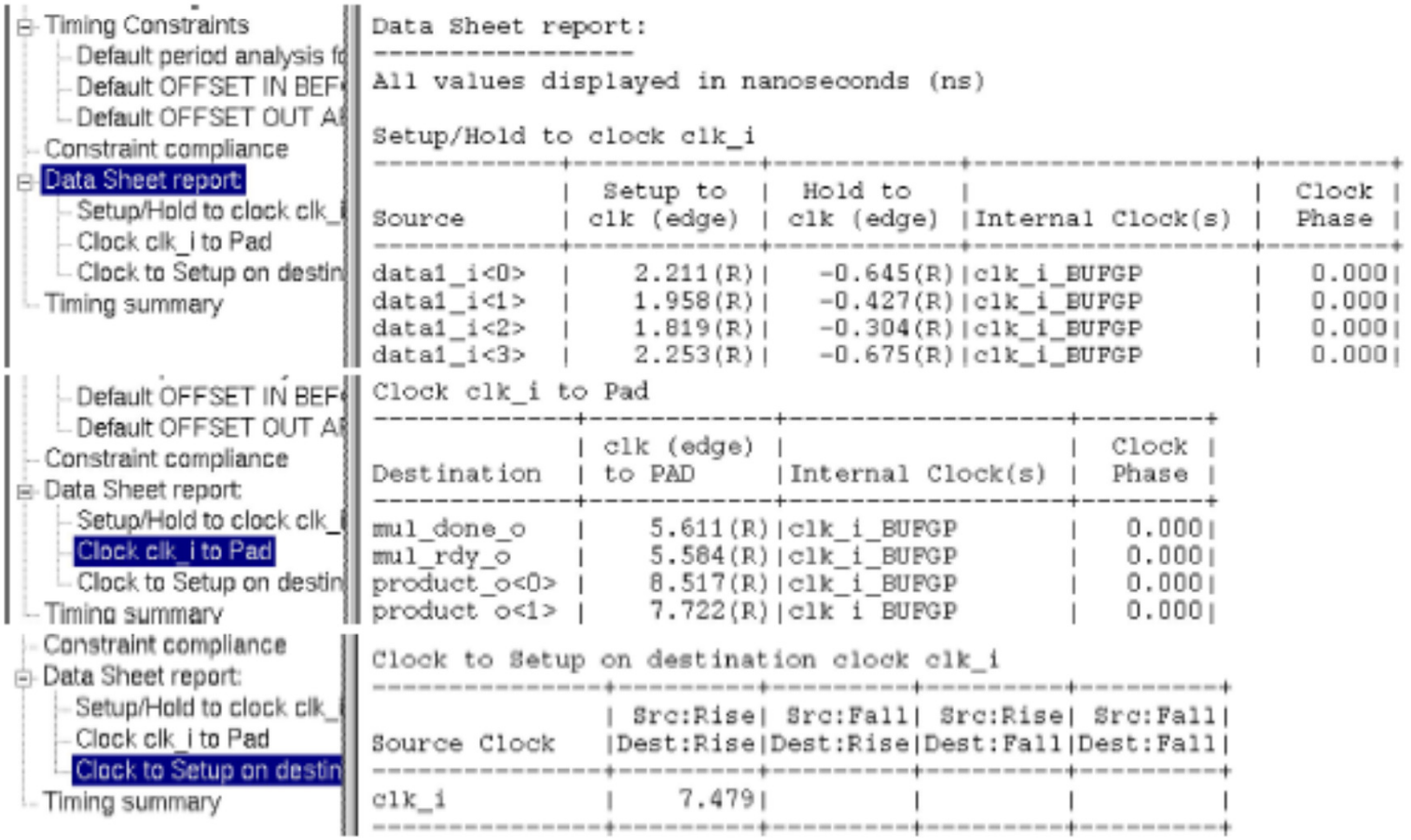
```
Timing Score: 0

Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

------------------------------------------------------------------------------------------------------
  Constraint                                 |  Check  | Worst Case |  Best Case | Timing |   Timing
                                             |         |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------------------------------------
  TS_clk_i = PERIOD TIMEGRP "clk_i" 7.5 ns   | SETUP   |     0.021ns|     7.479ns|       0|           0
  HIGH 50%                                   | HOLD    |     1.148ns|            |       0|           0
------------------------------------------------------------------------------------------------------


All constraints were met.
```

Slack value indicates the difference between the constraint and the analyzed value, with negative slack indicating an error condition.

- ***Setup Slack*** informs about the amount of setup violation seen for a path, and is calculated as :
  Setup Slack = Constraint_requirement - T_{clock_skew} - T_{data_path} - T_{su}
- ***Hold Slack*** :- Hold/Race checks are performed on register-to-register paths by taking the data path (T_{ckQ}+T_{route_total} +T_{logic_total}) and subtracting the clock skew (T_{dest_clk} - T_{src_clk}) and the register hold delay (T_h) i.e.
  Hold Slack = T_{data} - T_{skew} - T_{hold}.

Placing and Routing Design - *Post PAR Static Timing Analysis*

(ii) *Data Sheet Report* includes the source and destination PAD names, and either the propagation delay between the source and destination or the setup and hold requirements for the source relative to the destination.




```
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Setup/Hold to clock clk_i
------------+------------+------------+------------------+--------+
            |  Setup to  |  Hold to   |                  |  Clock |
Source      | clk (edge) | clk (edge) |Internal Clock(s) |  Phase |
------------+------------+------------+------------------+--------+
data1_i<0>  |    2.211(R)|   -0.645(R)|clk_i_BUFGP       |   0.000|
data1_i<1>  |    1.958(R)|   -0.427(R)|clk_i_BUFGP       |   0.000|
data1_i<2>  |    1.819(R)|   -0.304(R)|clk_i_BUFGP       |   0.000|
data1_i<3>  |    2.253(R)|   -0.675(R)|clk_i_BUFGP       |   0.000|
------------+------------+------------+------------------+--------+

Clock clk_i to Pad
------------+------------+------------------+--------+
            | clk (edge) |                  |  Clock |
Destination | to PAD     |Internal Clock(s) |  Phase |
------------+------------+------------------+--------+
mul_done_o  |    5.611(R)|clk_i_BUFGP       |   0.000|
mul_rdy_o   |    5.584(R)|clk_i_BUFGP       |   0.000|
product_o<0>|    8.517(R)|clk_i_BUFGP       |   0.000|
product_o<1>|    7.722(R)|clk_i_BUFGP       |   0.000|
------------+------------+------------------+--------+

Clock to Setup on destination clock clk_i
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk_i          |    7.479|         |         |         |
---------------+---------+---------+---------+---------+
```

Negative Hold Time indicates that data pin of the flip-flop can change ahead of the clock pin and still meet Hold Time check, due to internal data path-delay of Flip-Flop.

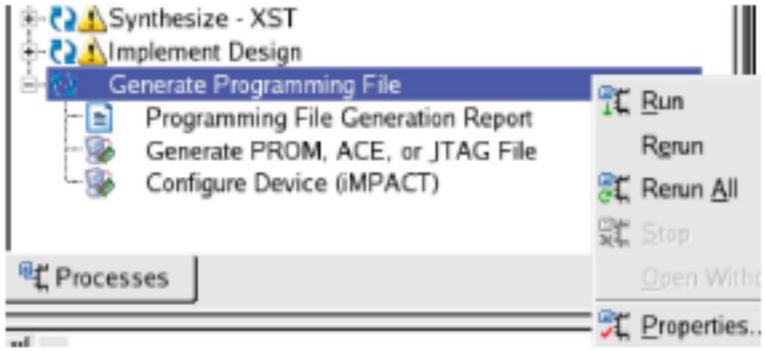
Generating BitMap Programming File

Purpose of Generating Programming File :

After design has been routed, it is required to generate the binary data which can be used to program the physical device. The Programming BIT File for FPGA Device should contain all the configuration information, defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. The binary data in the BIT file can then be downloaded into the FPGA's memory cells or it can be used to create a PROM file.

Step to Generate Programming File :

To create BitMap file, right-click *Generate Programming File* label in *Processes* window & click *Run* option.



BitGen Process File Types :

Xilinx uses *BitGen* process for generating Bitstream program. *BitGen* takes a fully routed NCD file, generated during PAR process as its input, and produces a configuration bitstream - a binary file with a *.bit* extension.