RTL SIMULATION & SYNTHESIS WITH PLDS

Lecture Notes

M.TECH(VLSI & ES)
(I YEAR – I SEM)
(2022-2023)

Prepared by

Dr.Anup Dey, Professor

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
Department of Electronics and Communication Engineering

Recognized under 2(f) and 12 (B) of UGC ACT 1956
(Affiliated to JNTUH, Hyderabad, Approved by AICTE-Accredited by NBA &NAAC–'A'Grade-ISO9001:2015 Certified) Maisammaguda,Dhulapally(PostVia.Kompally),Secunderabad–500100,Telangana State, India

RTL SIMULATION AND SYNTHESIS WITH PLDS

Course Objectives:

- To learn High-Level Design Methodology and overview of design flow
- To learn the coding skills relevant to synthesis of logic circuits.
- To understand importance of libraries in synthesis flow
- To design logic to meet specifications and optimization
- To understand the design constrains related to FSM

Unit I

High-Level Design Methodology Overview: ASIC Design Flow Using Synthesis, HDL Coding, RTL Behavioral and Gate-Level Simulation, Logic Synthesis, Design for Testability, Design Re-Use, Behavioral Synthesis & Concepts. Design Analyzer and Design compiler, Target Library, Link Library, and Symbol Library, Cell names, Instance names, and VHDL Libraries in the Synthesis Environment, Synthesis, Optimization and Compile, Classic Scenarios

Unit II

VHDL/Verilog Coding for Synthesis: General HDL Coding Issues, VHDL vs. Verilog: The Language Issue, Finite State Machines, HDL Coding Examples, Classic Scenarios.

Unit III

Links to Layout, Motivation for Links to Layout Floor planning, Link to Layout Flow Using Floorplan Manager, Creating Wire Load Models After Back-Annotation Re-Optimizing Designs After P&R. Design for Testability: Introduction to Test Synthesis, Test Synthesis Using Test Compiler

Unit IV

Constraining and Optimizing Designs: Synthesis Background, Clock Specification for Synthesis, Design Compiler Timing Reports, Commonly Used Design, Compiler Commands, Strategies for Compiling Designs, Typical Scenarios When Optimizing Designs, Guidelines for Logic Synthesis, Classic Scenarios.

Unit V

Constraining and Optimizing Designs for FSM: Finite State Machine (FSM) Synthesis, Fixing Min Delay Violations Technology Translation, Translating Designs with Black-Box Cells, Pad Synthesis, Classic Scenarios

Text Book

1. Kurup Pran, Taher Abbasi, Logic Synthesis using Synopsys, 2/e, Pearson Education, 2007.

References

1. VHDL for Logic Synthesis, Third Edition. Andrew Rushton. © 2011 John Wiley & Sons, Ltd. Published 2011 by John Wiley & Sons, Ltd.
2. Weng Fook Lee, VHDL Coding and Logic Synthesis with Synopsys, Academic Press, 2000
3. Morris Mano, Michael D. Ciletti, Digital Design , 4/e, Prentice Hall of India, 2008
4. Himanshu Bhatnagar, Advanced ASIC Chip Synthesis, Springer Science, 2013

Course Outcomes:

After successful completion of the course, the student will be able to

- Understand Synthesis Flow and Optimization
- Understand Synthesizable Coding Concepts
- Analyze Physical Design Concepts
- Understand Efficient Way of Giving Constrains
- Analyze Constraining and Optimizing Designs For FSM

Unit I
High-Level Design Methodology Overview

ASIC Design Flow Using Synthesis, HDL Coding, RTL Behavioral and Gate-Level Simulation, Logic Synthesis, Design for Testability, Design Re-Use, Behavioral Synthesis & Concepts. Design Analyzer and Design compiler, Target Library, Link Library, and Symbol Library, Cell names, Instance names, and VHDL Libraries in the Synthesis Environment, Synthesis, Optimization and Compile, Classic Scenarios

Introduction

In today's world, faster and less costly ASIC chips are being designed at a much quicker rate than before. ASIC designers are able to design much more efficiently than before.Designers are constantly under pressure to come up with faster performing designs, but with fewer resources.This has led to the development of many EDA tools that help designers to complete a design in a much shorter time frame. These EDA tools are based on the concept of designing ASIC components utilizing Hardware Description Language (HDL). Today, a designer does not need to spend much time manually drawing the circuitry involved in a design but instead can write synthesizable HDL code. A common form of HDL code used in the ASIC industry for synthesis is Very High-Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog.

Synthesizable VHDL can be used as a form of input in synthesis tools such as Synopsys's Design Compiler. This new methodology of design is a great asset to designers, as it increases both productivity and efficiency.

1)To software developers, RTL may mean *register transfer language*. An example is the generation of an intermediate file format.

2)To microprocessor designers, RTL may be conceived as a pseudo-code description , dataflow between different elements of the processor.

3)To FPGA designers, RTL stands for *register transfer level*, a relatively low level of abstraction .

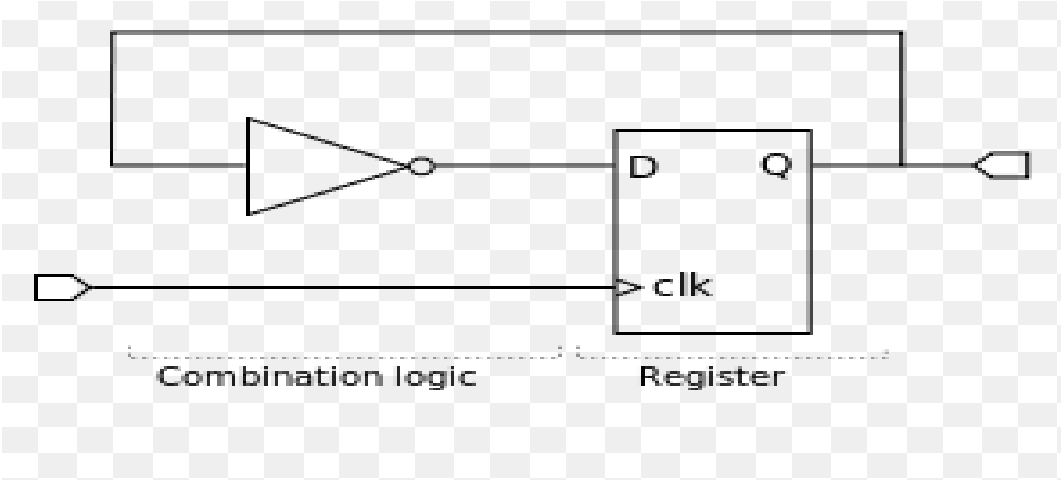
RTL can also be used to mean a Hardware Description Language(VHDL, Verilog, System C), where “RTL” code is a lower level of abstraction than “Behavioral Level” code.

Definition

The VHDL language standards committee “The register transfer level of modeling circuits in VHDL for use with register transfer level synthesis. Register transfer level is a level of description of a digital design in which the clocked behavior of the design is expressly described in terms of data transfers between storage elements in sequential logic, which may be implied, and combinatorial logic, which may represent any computing or arithmetic logic unit logic. RTL modelling allows design hierarchy that represents a structural description of other RTL models.”

RTL -Register Transfer Level

Digital circuit as registers + combinational logic (the logic is the 'transfer' between registers). This is higher level (and also a lot more convenient) than gate level, or transistor level.



A synchronous circuit consists of two kinds of elements: registers (Sequential logic) and combinational logic. Registers (usually implemented as D flip flop) synchronize the circuit's operation to the edges of the clock signal, and are the only elements in the circuit that have memory properties. Combinational

logic performs all the logical functions in the circuit and it typically consists of logic gates.

For example, a very simple synchronous circuit is shown in the figure. The inverter is connected from the output, Q, of a register to the register's input, D, to create a circuit that changes its state on each rising edge of the clock, clk. In this circuit, the combinational logic consists of the inverter.

When designing digital integrated circuits with a Hardware Description Language (HDL), the designs are usually engineered at a higher level of abstraction than transistor level (logic families)or logic gate level. In HDLs the designer declares the registers (which roughly correspond to variables in computer programming languages), and describes the combinational logic by using constructs that are familiar from programming languages such as if-then-else and arithmetic operations. This level is called *register-transfer level*. The term refers to the fact that RTL focuses on describing the flow of signals between registers.

RTL Simulation

Register Transfer Level (RTL) simulation and verification is one of the initial steps that was done. This step ensures that the design is logically correct and without major timing errors. It is advantageous to perform this step, especially in the early stages of the design, because long synthesis and place-and-route times can be avoided when an error is discovered at this stage. This step also eliminates all the syntax errors from your VHDL code.Synopsys simulation tools to perform RTL verification.

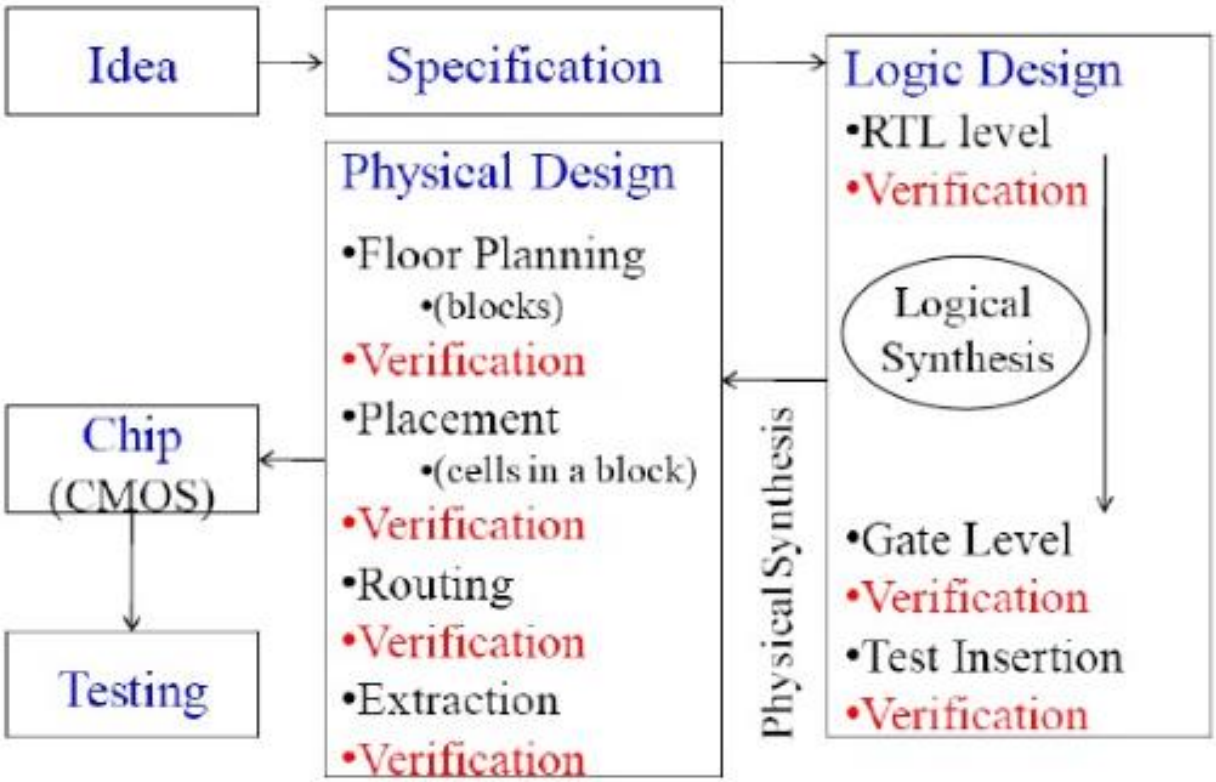
Programmable logic device

A programmable logic device (PLD) is an electronic component used to build reconfigurable digital circuits. Unlike Integrated circuits (IC) which consist of logic gates and have a fixed function, a PLD has an undefined function at the time of manufacture.–Before the PLD can be used in a circuit it must be programmed (reconfigured) by using a specialized program

Logic synthesis

The process of translating and mapping RTL code written in HDL (such as Verilog or VHDL) into technology specific gate level representation is logic

synthesis.Its a process by which an RTL model of a design is automatically turned into a transistor-level schematic netlist by a standard EDA tool.Abstract specification of desired circuit behavior, typically at Register transfer level (RTL), is turned into a design implementation in terms of logic gates, by a computer program called a *synthesis tool*.Common examples include synthesis of designs specified in Hardware Description Languages.Logic synthesis is one aspect of electonic design automation.

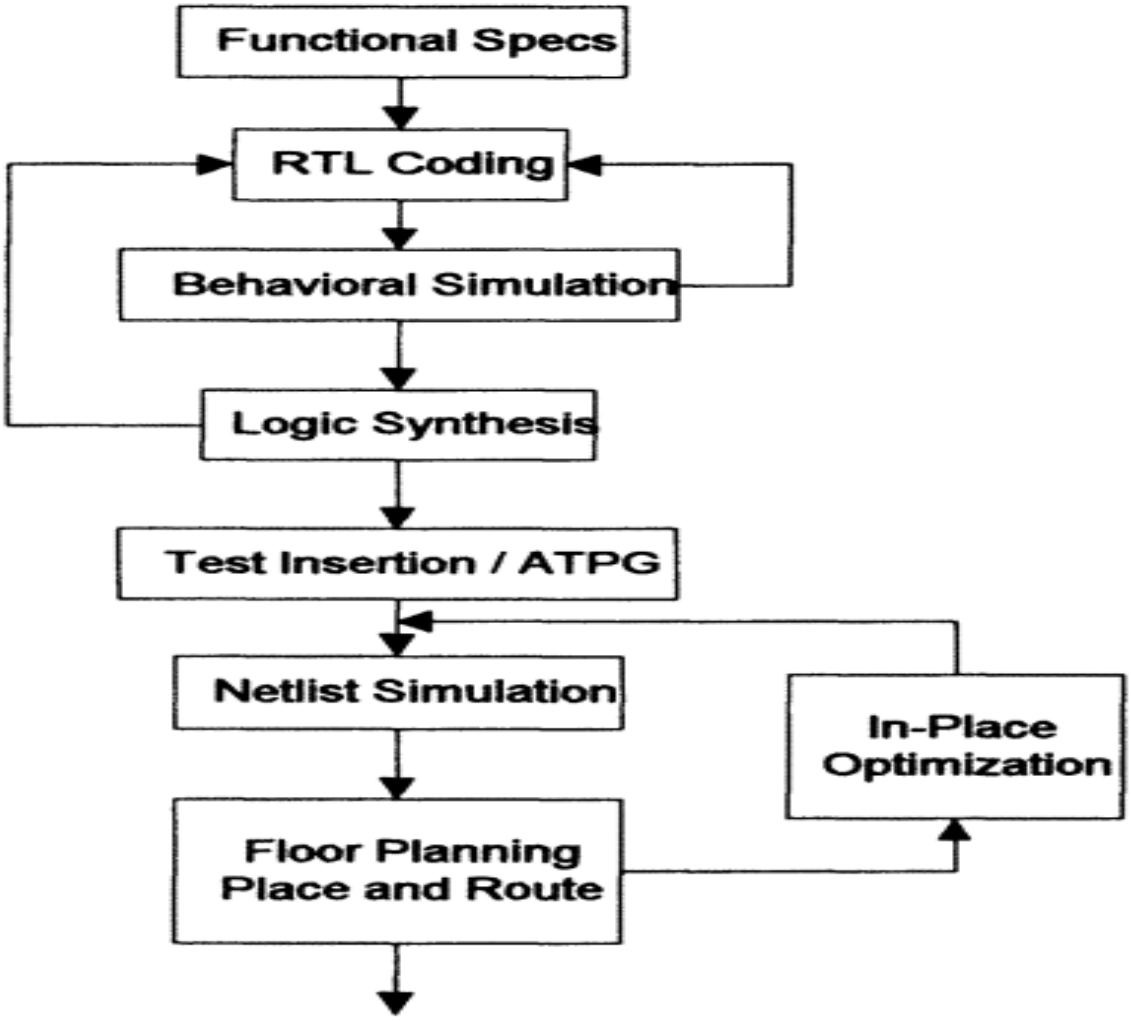


ASIC Design Flow Using Synthesis

Major advances in fabrication technology have made possible high integration, large gate count ASICs. Hardware description languages and logic synthesis have had a significant impact on the design process of these ASICs. With the adoption of HDL-based design there has emerged a highlevel design flow based on synthesis.The most commonly used HDLs today are VHDL and Verilog. The desired functionality of a design is first captured in HDL code, usually Verilog or VHDL. This step is complex, particularly for IC designers who are accustomed to schematic capture tools. This is further compounded by the fact that this code then must be synthesized into an optimal design which

meets the functional requirements of the initial specification. The synthesis based ASIC design flow can be divided into the following steps:

1. Functional Specification ofthe design
2. HOL Coding in VHDL/Verilog RTL.
3. RTL/Behavioral or Functional simulation of the HOL.
4. Logic Synthesis
5. Test Insertion and ATPG.
6. Post-synthesis or gate-level simulation.
7. Floorplanning / Place and Route.



The above seven steps are usually iterative as shown in Figure. For example, on performing a functional simulation of the source HDL code, one might find that the code does not exactly match the desired functional behavior. In such cases, one must return to modify the source code.

Also, after synthesis, it is possible that the netlist does not meet the timing requirements ofthe clock. This implies that one must either modify the source HDL, or attempt alternate synthesis strategies.Similarly, after performing place and route, it is required to back annotate delay values to incorporate real-world delays. This is followed by *in-place optimization (lPO)* of the netlist to meet routing delays.

The functional specification is always the first step in an ASIC design process. Designers in particular, are extremely familiar with formulating the specifications of a design. Most often the design specification is followed by a block level diagram of the entire ASIC. The block level diagram of the ASIC is usually done using graphical design entry tools like the *Synopsys Simulation Graphical Environment (SGE), Composer (Cadence) etc.*

After a block level schematic capture of the design, the next step involves HDL coding. The style of HDL coding often has a direct impact on the results the synthesis tool delivers. A sound knowledge of the working of the synthesis tool will help the designer write synthesizable code better. For example, one common problem arises due to partitioning of designs. Typically, designers partition designs based on functionality. During the integration of different modules in synthesis, one might find a large amount of logic in the critical path. This critical path most often traverses several hierarchical boundaries. In a typical design team scenario, these blocks are usually designed by different engineers, thereby compounding the problem. Logic synthesis provides the best results when the critical path lies in one hierarchical block as opposed to traversing multiple hierarchical blocks. In such situations, it is often required to modify the hierarchy in the source HDL code and re-optimize the design or modify the design hierarchy through Synopsys tool specific scripts.

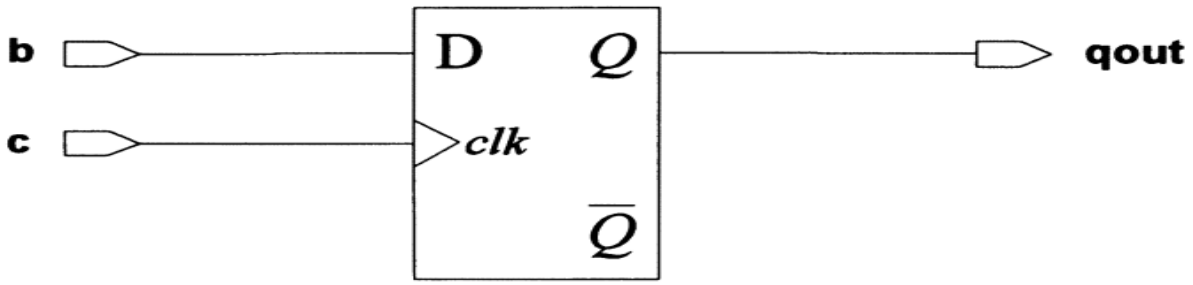
Another coding tip is to *ensure that all outputs ofsub-blocks/modules are registered outputs.* In synthesis, this helps to estimate the input delays and helps avoid intricate time budgeting. Also, it is recommended that logic blocks such as ROMs and random logic each be grouped into separate hierarchical blocks.

HDL Coding

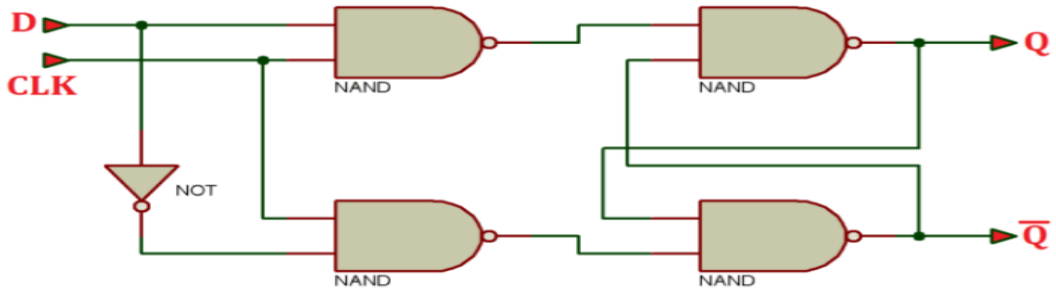
HDL code can be behavioral or RTL. In the synthesis domain, the latter is usually considered to be the synthesizable fonn of HDL code. Examples of HDL code which infer on synthesis a D-flip flop, an AND gate are in synthesizable RTL code format.

Example 1:

D Flip flop



The D flip flop circuit



Truth table

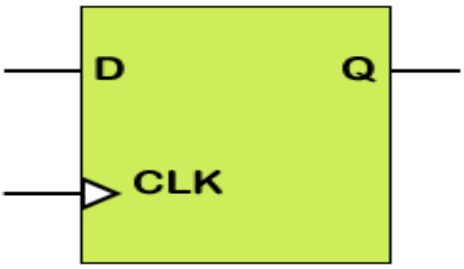
CLK	D	Q	Q(not)
↑	0	0	1
↑	1	1	0

VHDL Code

```
entity flipflop is
        port (b,clk : in bit;
                qout : out bit);
end flipflop;
architecture test of flipflop is
begin
        process
        begin
                wait until clk'event and clk = '1' ;
                qout <= b ;


        end process;
end test;
```

Verilog Code

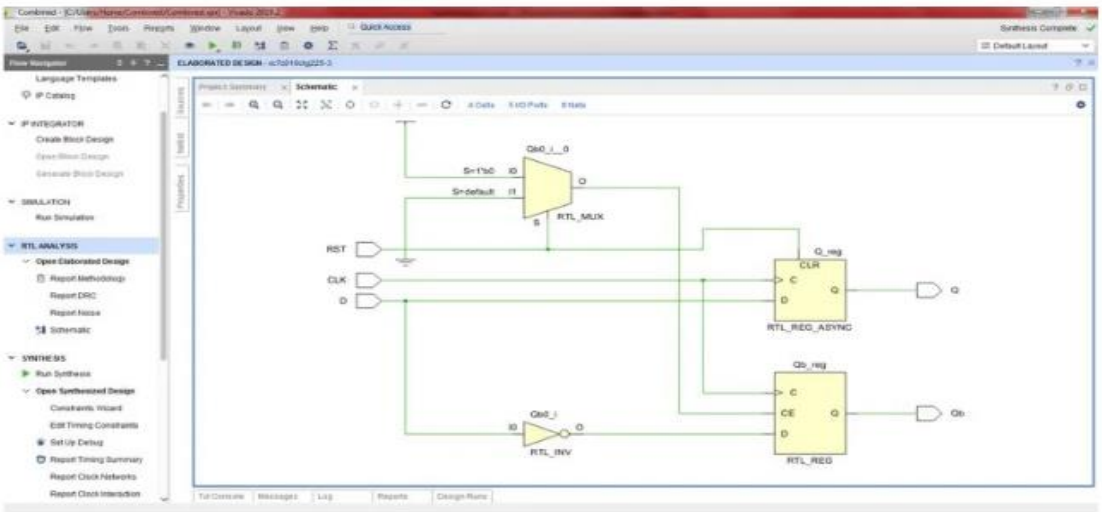
```
module flipflop(b,clk,qout);
input b,clk;
output qout;
reg out;
always@ (posedge clk)
        qout <= b;
endmodule
```



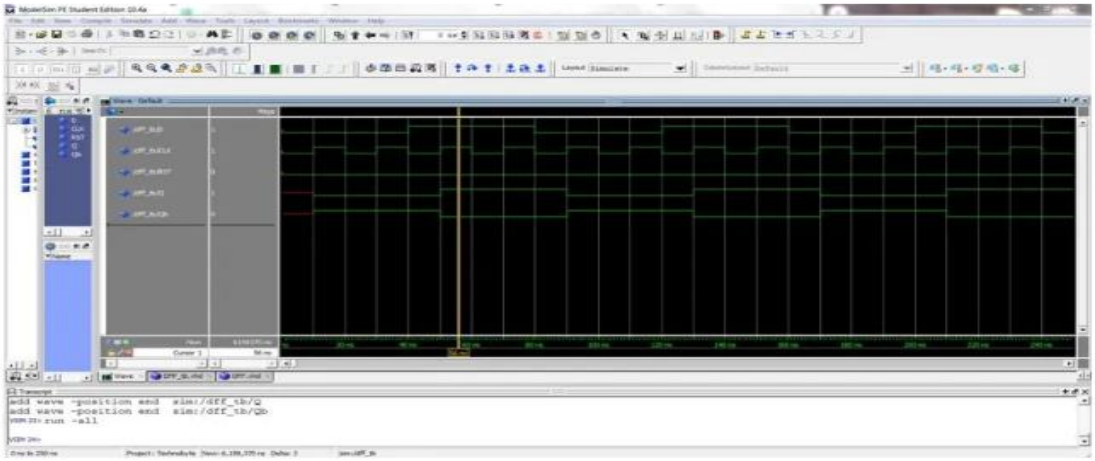
Rising-Edge D Flip-Flop

Falling-Edge D Flip-Flop

RTL Schematic



Simulation Waveform

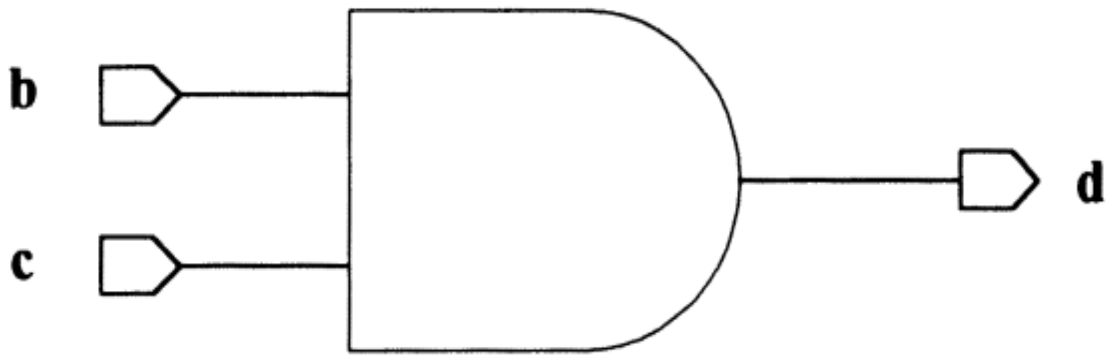


Example shows VHDL and Verilog code which when synthesized infers a positive edge triggered flip flop. If one desires a negative edge triggered flip flop, the obvious change to make would be to replace the posedge declaration in the code with negedge in the Verilog code (or clock=' l' by clock='O' in VHDL). While this might appear to be an obvious solution, inferring a negative edge triggered flip flop is largely dependent on an appropriate library cell being available.

Instead, the tool would infer a positive edge triggered flip flop with an inverter driving the clock pin of the flip flop. One can always instantiate a negative edge triggered flip flop (using structural HDL code) from the ASIC vendor library to circumvent the problem. In a larger design, which has both positive and negative edge triggered flip flops, it is recommended that all the positive edge triggered flip flops be grouped into a separate level of hierarchy and all the negative edge triggered flip flops be grouped into another level ofhierarchy. This makes the debug process and timing analysis during synthesis simpler.

Example 2

AND gate



AND gate's truth table

A	B	Y(A *and* B)
0	0	0
0	1	0
1	0	0
1	1	1

VHDL

```
entity AND2 is
```

```
        port (b,c : in bit;
                d : out bit);
end AND2;
architecture test of AND2 is
begin
        process
        begin
        if ( c = '1') then
                d <= b ;
        else
                d <= '0' ;
        end if;
        end process;
end test;
```

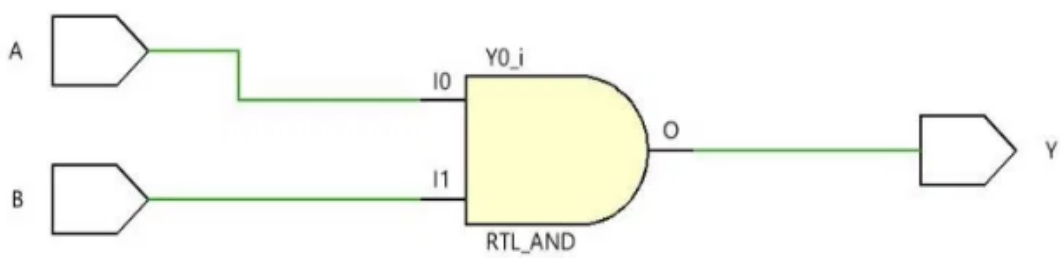
Verilog

```
module and2(a,b,out);
input a,b;
output out;
reg out;
always@ (a or b)
        if (a == 1)
                out = b;
        else
                out = 0;
endmodule
```

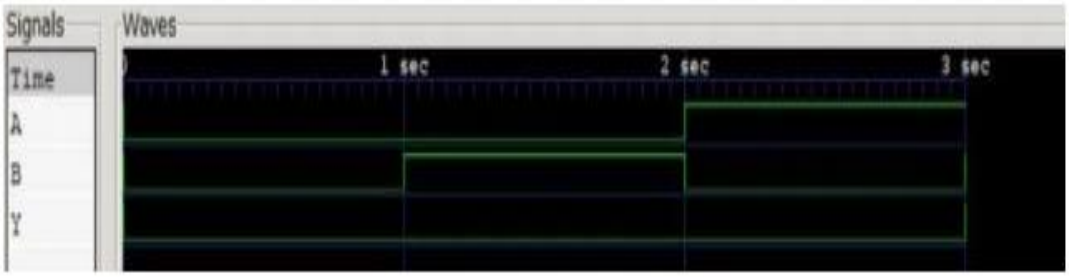
```
module AND_2_behavioral (output reg Y, input A, B);
```

```
Then we write,

always @ (A or B) begin
   .....
end
```

RTL schematic of the AND gate



Simulation Waveform

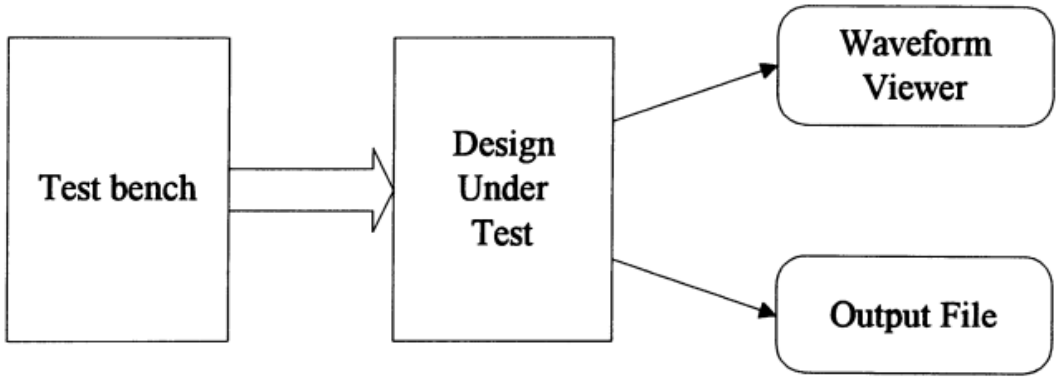


Above example infers an AND gate when synthesized. In this case, the HDL code exactly matches the logic inferred. In general, to infer an AND gate, the recommended coding style for synthesis is using (out = a&b;) instead of the if statement.

RTL Behavioral and Gate-Level Simulation

After the design has been captured in HDL, it is essential to verify that the code matches the required functionality, prior to synthesis. We call this step the *pre-synthesis behavioral simulation* of the HDL. This can be performed by simply assigning specific values to input signals, performing simulation runs and viewing the waveforms in a graphical simulation tool. An alternative is to write a testbench.

The testbench can be considered as an HDL block whose outputs provide the stimuli for the design to be simulated. In general, it is recommended that one write a testbench for simulation to simplify the postsynthesis simulation step. The same testbench can then be used for post synthesis simulation and the results of the two compared.



There are two possible ways of simulating a design using a testbench as shown in Figure. One can write a testbench where all the stimuli for the different signals are provided in the HDL code. Then one would use a graphical front end of a simulation tool to view the waveforms. It is also possible to provide the stimuli for the different signals and pipe the outputs to a file.

If one has another file of expected results, one can quite easily compare the two files to ensure that the two match.
In order to simulate a synthesized gate-level netlist, VHDL simulation models of the technology library cells are required. These can be of three kinds - unit delay structural model (UDSM), full-timing structural model (FTSM), fulltiming behavioral model (FTBM) or full-timing optimized gate-level simulation (FTGS). While UDSM and FTSM are used for functional verification, the FTBM is used for accurate, detailed timing verification and FTGS library for fast, sign-off-quality timing verification.

Unit Delay Structural Model (UDSM) - In this model of a technology library, all combinational cells have a rise/fall delay of Ins, while all sequential cells have a rise/fall delay of 2 ns.

Full-Timing Structural Model (FTSM) - This model includes transport wire delays and pin-to-pin delays on a zero delay functional network. Timing constraint violations are reported as warning messages.
Full-Timing Behavioral Model (FTBM) - This delay model is used for detailed timing verification. Transport wire delays and pin-to-pin delays are included in this delay model.

Full-timing optimized gate-level simulation (FTGS) - These models include transport wire delays and pin-to-pin delays in the delay model. In addition to warning messages, the Simulator can schedule X output values for timing constraint violations and circuit hazards. One can use the FTGS library for fast,sign-off-quality timing verification.

If one has the ASIC vendor library (the Synopsys *.db* file) for synthesis and the Synopsys Library Compiler, the above VHDL simulation models can be automatically created using the *liban* utility. The liban utility creates two files from the ASIC vendor library in Synopsys*db* (database) format. For example, given the technology library 'hlylib.db':the *liban* utlity generates an encrypted VHDL (mylib.vhd.E) containing simulation models with timing delays, and a VHDL package*(mylib_components.vhd)* containing the component declarations for all the cells in the ASIC vendor library.

Logic Synthesis

Synthesis, as referred to in present-day IC design, can be broadly divided into *logic synthesis* and *high-level synthesis.*

High-level synthesis is closer to what some refer to as 'behavioral synthesis': High-level synthesis involves synthesis of logic from behavioral descriptions. Synopsys *Behavioral Compiler* is specially targeted towards behavioral synthesis. Logic synthesis on the other hand synthesizes logic from register transfer level (RTL) descriptions. In the Synopsys domain, DC capabilities such as arithmetic optimization, implementation selection, resource sharing, in place optimization and critical path re-synthesis are referred to as *high level optimization.* High level optimization must not be confused with behavioral synthesis.

The logic synthesis process consists of two steps - translation and optimization. Translation involves transforming a HDL (RTL) description to gates, while optimization involves selecting the optimal combination of ASIC technology library cells to achieve the required functionality.

Design For Testability

With the increasing complexity of ASIC designs, the cost of testing designs has become a fairly substantial component of the overall manufacturing and maintenance cost. The cost of testing is an outcome of several cost components such as test generation cost, testing time, automatic test equipment cost etc.

Design for testability (DFT) techniques have been used in recent years to reduce the cost of testing by defining testability criteria early in the design cycle.

The most popular DFT technique in ASIC design is the *Scan Design Technique.* Scan techniques involve replacing sequential elements in the design with equivalent scan cells. There exist different styles of scan cells. Based on the *scan style* selected the design is required to meet certain design rules. The most commonly used scan style is the multiplexed flip flop.

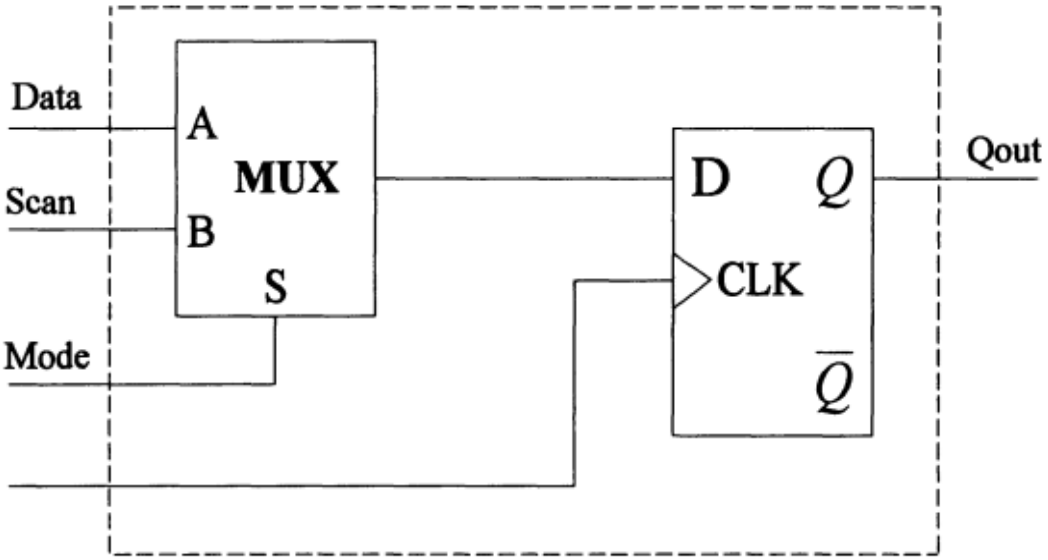


Figure *Muxed Scan Flip Flop*

A muxed scan flip-flop, as the name indicates, consists of a mux and a flipflop. The output of the mux drives the data input of the flip-flop and the

select input is controlled by the test mode pin; the inputs to the mux are the data input and the test input as shown in the Figure. Sequential cells are replaced by scan equivalents to achieve the primary requirement of testability, namely, observability and controllability.

A scan cell has two modes of operation, namely, the normal mode and the test mode. During the normal mode it behaves like the sequential cell it replaces, but in the test mode the scan input is loaded into the scan cell on the active edge of the clock transition.

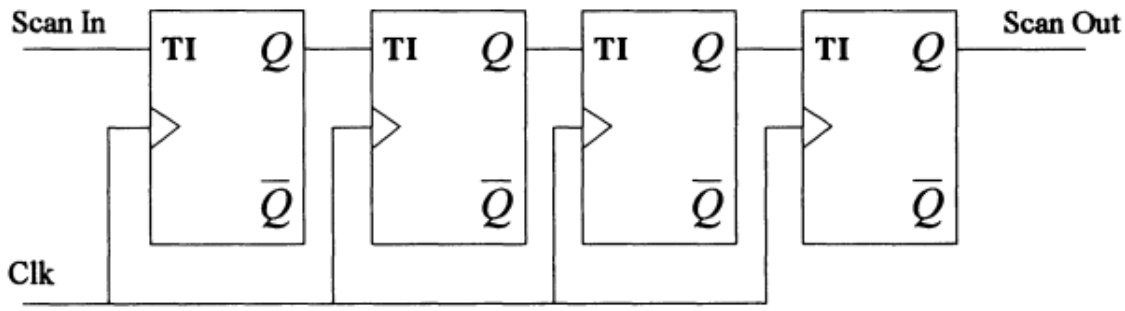


Figure. Scan chain connected to form a shift register

While on-chip testability is being increasingly adopted in ASIC design, the extent to which it is implemented is often dependent on the target application, the area overhead, and the required fault coverage. Hence, within the scope of scan design, there exist two possible test methodologies :

1. Full scan
2. Partial scan

If all the sequential cells are replaced by scan cells, then it is called a *Full Scan test methodology*. In this case, the *Automatic Test Pattern Generation (ATPG)* algorithm is combinational.

In a Partial Scan test methodology only some ofthe sequential cells are replaced by scan cells. In this case, the ATPG algorithm is sequential. Also, for partial scan it is required that the decision be made regarding which sequential cells are to be replaced by scan cells. This critical decision is usually based on the desired fault coverage and the

available silicon area.

Once the test methodology has been specified as either partial or full scan, and the scan style declared, the Synopsys Test Compiler (*TC*) automatically replaces the sequential cells by scan cells. This replacement of sequential cells with scan equivalents occurs if the target technology library has scan equivalent cells of the required scan style, and provided no test design rule violations exist.

It is possible that some flip flops in the design cannot be replaced by a scan equivalent due to design rule violations or user specified controls. In such cases, if full scan is the adopted methodology, then all the non-scan cells are interpreted as *black boxes*. This implies that all the faults associated with the black-box are untestable and hence results in reduced overall fault coverage. After scan insertion, the TC generates the test patterns for the design and the fault coverage achieved is calculated. The fault coverage is calculated as a percentage of the testable faults upon the total number of possible stuck-at-O and stuck-at-I faults.

In general, full scan designs tend to achieve a higher fault coverage than partial scan designs. The amount of fault coverage for a partial scan design is related to the number of scannable registers in the design.After ATPG, the test vectors must be formatted in one of the formats supported by the simulator on which the test vectors are to be simulated.

Design Re-Use

Several design houses rely on re-use of large blocks of designs when building newer versions of existing chips. In some cases, it might involve re-targeting an existing design to a new technology library. For others, it might involve minor tweaks to existing designs. In general, the strategy used to realize these changes has a significant impact on the turn around time.

Design re-use is an effective means to achieve fast turn around on complex designs. A clear advantage in time is gained by re-use of designs from a library of parts rather

than designing from scratch. The upcoming generation of complex systems will require a widespread availability of re-usable parts. Synthesis provides a very efficient and flexible mechanism to build a library of re-usable components. This is essentially the Synopsys DesignWare (DW) approach

DesignWare Component Libraries

Synopsys provides DesignWare component libraries with existing re-usable parts. Version 3.0b of DC consists of three DesignWare libraries namely, ALU, Advanced-Math and the Sequential families. The number of these libraries is certain to increase with subsequent versions of the software.

Internal to the DC, these libraries are referenced *as* DWOI, DW02 and DW03 respectively. In addition, there exists the generic GTECH library. When a source HDL is read into DC, the design is converted to a netlist of GTECH components and inferred designware parts.

The GTECH library, like the DW libraries, is a technology independent library that aids users develop technology independent parts. The GTECH library called “gtech.db" contains common logic elements such *as* basic logic gates and flip flops. In addition, the gtech.db also contains a half adder and a full adder.

The DW libraries contain relatively more complex cells such *as* adders, subtracters, shifters, FIFOs, counters, comparators and decoders. These parts are *parametrizable, synthesizable, testable and technology independent* making them easily usable in a HDL design flow. Moreover, these parts have simulation models (VHDL models only, not Verilog) provided with the libraries, thus substantially improving the time required for both HDL coding and functional simulation. These parts serve *as* off-theshelf design modules which can be used *as* functional sub-blocks in larger designs. Moreover, the synthesis tool ensures that when DW parts are used, high-level optimization features such *as* implementation selection, arithmetic optimization and resource sharing are automatically turned on.

Designing Using DesignWare Components

Consider the case of a Universal Asynchronous Receiver Transmitter (UART). The UART is used in almost all designs which involve serial to

parallel and parallel to serial data conversion, and transmitting and receiving of data. A UART can be used in a number of designs provided it is sufficiently flexible.

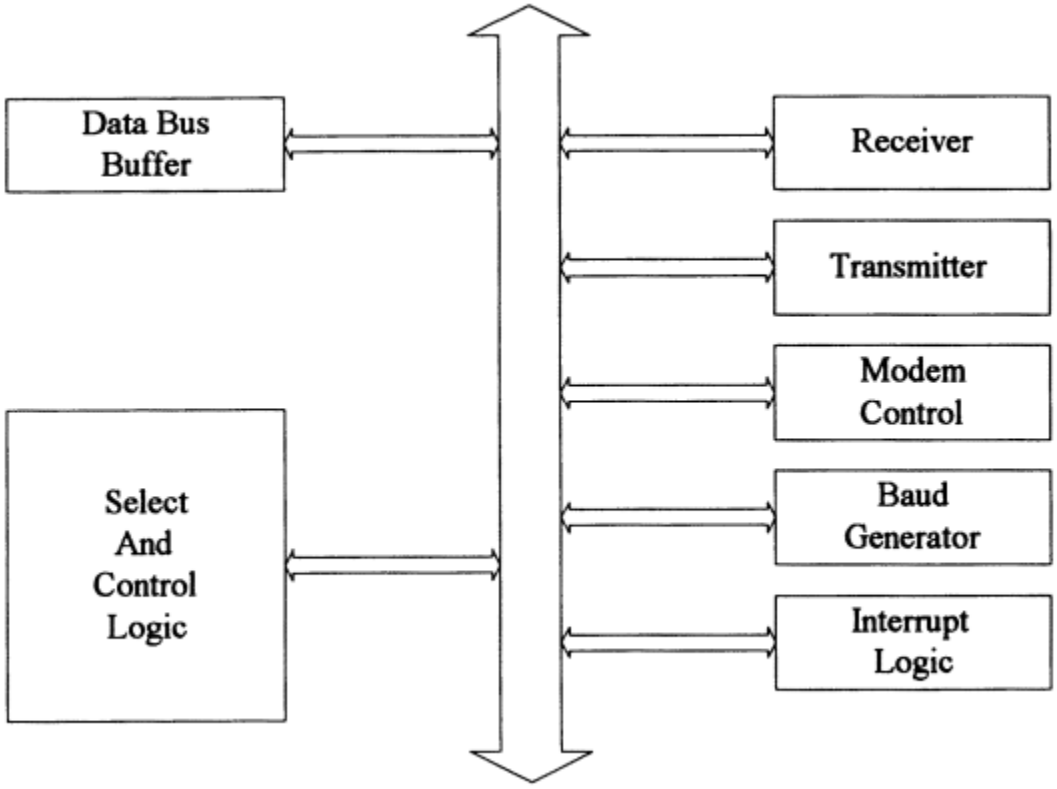


Figure . Hierarchical Overview of UART

Figure shows a block diagram of the hierarchical overview of the UART. The UART requires several registers capable of performing shift operations, several counters, decoders and FIFOs, The Synopsys DesignWare libraries have pre-existing components namely, *DW03_UPDN_CTR* (up down counter), *DW03_FIFO_S_DF* (FIFO), *DWOl_DECODE* (a decoder), *DWO*3*_SHFT_REG* (shift register) which can be used in designing a UART,

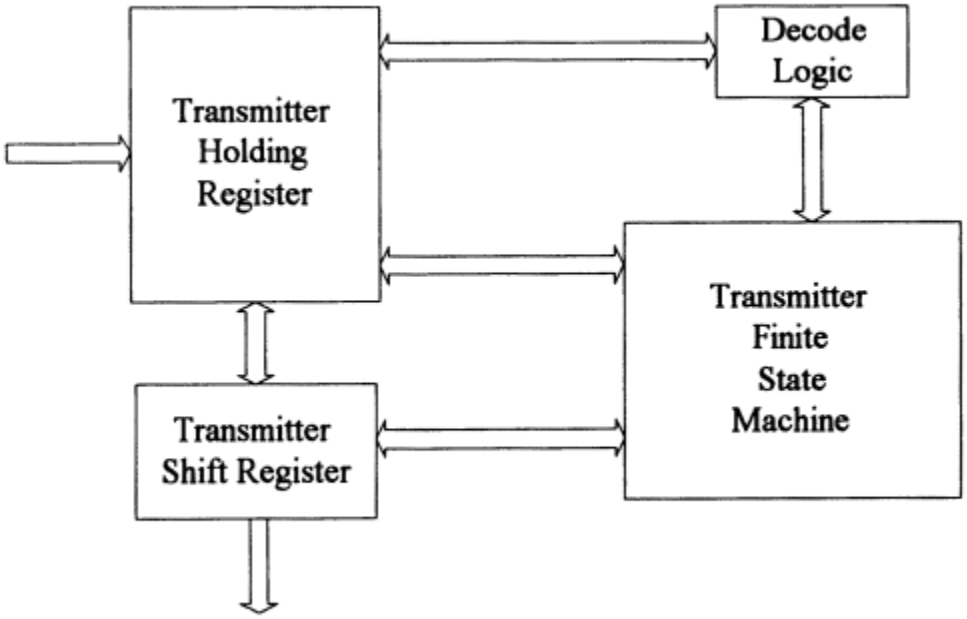


Figure. *Transmitter Block of UART*

Figure shows the transmitter sub-block of the UART. The transmitter holding register can be built using the *DW03_FIFO_S_DF*. The *DW03_DECODE* can be used in the decode logic and the shift register *(DW03_SHFT_REG)* in the transmitter shift register block. The transmitter FSM is the only additional logic that is required to be designed.

Behavioral Synthesis & Concepts.

Design Analyzer and Design compiler

Design Analyzer (DA) is the graphical front end of the Synopsys synthesis tool. Design Compiler or dc_shell is the command line interface for the same synthesis tool. In most cases, designers begin using the graphical front end, and once they are comfortable with the commands and the Design Compiler terminology, they prefer to use the command line (dc_shell) interface. At that stage, the DA is generally used only to view schematics and their critical paths. The command line interface is identified by the following dc_shell prompt:

```
dc_shell >
```

If the environment variable SYNOPSYS has been set to the synopsys root directory, then typing:

```
$SYNOPSYS/sparc/syn/bin/dc_shell
```

should invoke the DC and show the dc_shell prompt. If not running on a sparc, then use the corresponding architecture. A similar prompt can be seen from the Design Analyzer command window. The command window can be invoked from the Design Analyzer from the Setup -> Command Window pull down menu.

```
design_analyzer >
```

When using the Design Analyzer, the command window helps the user understand the commands executed when using the menus in the DA. The DA, in tum, can be invoked by typing the following:

```
$SYNOPSYS/sparc/syn/bin/design_analyzer
```

Startup Files

The DC, when invoked, reads the *.synopsys_dc. setup file.* The synopsys directory tree has a system wide *.synopsys_dc. setup* file. This file is located in $SYNOPSYS/admin/setup directory. In addition to this system wide file, the user can have a local *.synopsys_dc.setup* file in the current working directory or in the home directory.

In general, the .synopsys_dc.setup file is used to specify certain commonly used variables like the *target_library, link_library* and the *search_path.* The *.synopsys_dc. setup* file in the current working directory has the highest precedence, followed by the one in the user's home directory and finally, the system wide file. Shown below is a sample *.synopsys_dc.setup* file.

Example

```
search_path     = search_path+{".",./lib",./vhdl",./script}
target_library  = {target.db}
link_library    = {link.db}
symbol_library  = {symbol.sdb}
```

DC follows the paths in the *search_path* variable from left to right. For example, if a *link_library* file, *link.db* exists in the lib directory and in the vhdl directory, then the link.db file found in the *lib* directory is used.

If the libraries are assigned correctly and the *search_path* indicates the location of these files, then, on invoking the DA, the Setup -> Default pull down menu should indicate the specified target, link and symbol libraries.

```
dc_shell > include .synopsys_dc.setup
design_analyzer > include .synopsys_dc.setup
```

The *target_library*, *link_library* and *search_path* can be specified from the DA, Setup --> Defaults menu. To verify the current value of any variable, use the list <variable_name> at the dc_shell prompt as shown below.

```
dc_shell > list target_library
```

Target Library, Link Library, and Symbol Library

Target library is the ASIC vendor library whose cells are used to generate a netlist for the design described in HDL during synthesis. The HDL code is “mapped" to cells from this library.

The link library is used when a design is already in the form of a netlist or when the source HDL has cells instantiated from the technology library.The netlist is a design described using technology library cells. The link library which is specified by setting the *link_library* variable, indicates to the DC, the library in which the descriptions of these cells are available.Similarly, the *target_library* variable and the *symbol_library* variable help specify the target and symbol libraries

respectively.

Example :

Example shows a simple Verilog netlist written out by the DC. This example should help to better understand the link library concept. The netlist has four instances (U5, U6, U7, US) of the IVA library cell. If one wished to read the above netlist into DC and execute a command like *report_timing*, then, the link library declared by the *link_library* variable must have a description for the IVA library cell. If DC is unable to find a description for the IVA cell in the link library, the tool will be 'unable to resolve the reference IVA".

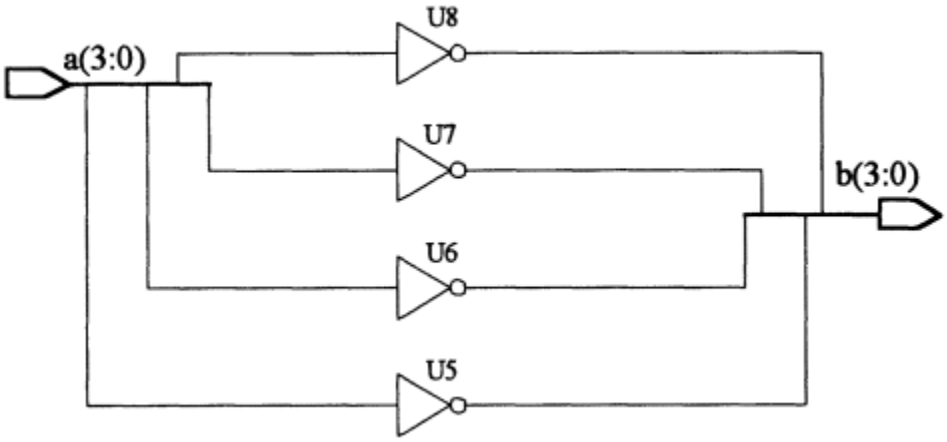


Figure. Netlist showing instances

```
/* Verilog netlist */

module bus( a, b );
input  [3:0] a;
output [3:0] b;
   IVA U7 ( .A(a[2]), .Z(b[2]) );
   IVA U8 ( .A(a[3]), .Z(b[3]) );
   IVA U5 ( .A(a[0]), .Z(b[0]) );
   IVA U6 ( .A(a[1]), .Z(b[1]) );
endmodule
```

Symbol libraries are pictorial representations of library cells. If one wishes to view the schematic of a design in DA, then the tool requires a symbol library which contains actual graphic representations of all the library cells. The tool performs a one to one mapping ofthe cells in the netlist to cells in the symbol library, when the user attempts to view the netlist representation.

For example, on reading the above netlist into DC, followed by the link command, the tool looks for the IVA cell in the link library to 'resolve the reference" IVA. Once it finds IVA in the *link_library*, the tool then looks for the IVA symbol in the symbol library. Viewing the schematic in the DA can be done by double clicking on the design icon. However, the equivalent command executed is the *create_ schematic* command. If it is unable to find cells in the symbol library, DA uses the generic symbol library (generic.sdb) to create schematics.

The technology and symbol libraries must exactly match in “case"for the cell names and their pin names. In other words, if the pin names ofthe IVA cell in the technology library do not match the pin names of the cell IVA in the symbol library, the tool will not be able to use the IVA symbol. In such a scenario, the tool uses symbols from the synopsys default generic library.

The *compare_lib* command in DC, is a fast check mechanism to determine any differences between the symbol library and the technology library that might exist.

```
dc_shell > compare_lib <target_library>  <symbol_library>
```

The link mechanism can be forced to be case insensitive by setting the following dc_shell variable to false.

```
dc_shell > link_force_case = false
```

The target and link libraries are of .db extension while *symbol_libraries* are of .sdb extension. Technology libraries are generated by the Synopsys Library Compiler from .lib files. These in tum are text files created by the ASIC vendor in Synopsys Library Compiler syntax. The ASIC vendor provides the user with .db and the .sdb files.

```
dc_shell > read_lib my_lib.lib
dc_shell > write_lib my_lib.db
```

Symbol libraries are created from .slib files.

```
dc_shell > read_lib my_lib.slib
dc_shell > write_lib my_lib.sdb
```

Cell names, Instance names

In DC terminology, cell names and instance names are the same. For example,if a design uses an IVA library cell, then, the tool provides it with an instance name (or cell name) such as *V6*. IVA is the reference and *VI* is an instance of the IVA reference, which in tum is a library cell.

VHDL Example

```
entity top is
        port (a : in bit;
                b : out bit);
end top;
architecture test of top is
component sub1
        a : in std_logic ;
        z : out std_logic ;
end component;
begin
U1 : sub1 port map (a => a, z=> b);
end test;
```

Verilog Exanple

```
module top( a, b );
input  [3:0] a;
output [3:0] b;
   sub1 U1 ( .A(a), .Z(b) );
endmodule
```

A sub design *sub1* is instantiated in a hierarchical block. When sub-designs are instantiated in higher level designs in a hierarchy, the name of the sub-design is the reference name *(sub I)*, while the instance name (or cell name) is the name assigned in the HDL code of the top level (UI in this case) design during instantiation.

Two useful commands are *report_reference* and *report_cell.* These are helpful when one is required to find the references and instances inferred in a

design after compile. For example, if a design comprises two library cells, each of which is used three times, *report_cell* command will list all the six instances and point to the corresponding reference, while the report_reference command will list just two references and the number of times each reference is used.

The *report_reference* shows just one reference, while the *report_cell* shows four instances or cells.

```
dc_shell > report_cell
```

VHDL Libraries in the Synthesis Environment

The VHDL language supports libraries. That is, frequently used functions, and component declarations are stored in packages and these packages are analyzed into libraries. The packages are then called via the “use" clause in VHDL. A package must be analyzed prior to being used in a another design. The package can be a part ofthe VHDL code or a separate VHDL file.

If the package is a separate file, then it must be analyzed prior to being used in a design. In general, it is recommended that one maintain separate package files and declare them using the “use" clause when required in VHDL design files. Since Verilog does not have a configuration management mechanism like VHDL, this is not applicable to Verilog

```
package my_pack is
type fsm_states is (state1, state2, state3, state4);
end my_pack;

// VHDL  file using my_pack
library States;
use States.my_pack.all;
```

Analyzing (using the analyze command) a design described in VHDL in DC, results in an intennediate format of files with .syn, .sim and .mra extensions. Example shows a package *my_pack*, a

VHDL design file that requires the *my_pack* package and the dc_shell script. The synthesis tool provides a mechanism by which the user can map a design library to a UNIX directory. Maintaining libraries is a good design practice as it simplifies the process of managing files.

```
/*dc_shell Script*/

dc_shell > define_design_lib States -path ./lib
dc_shell > analyze -format vhdl my_pack.vhd -lib States
```

Example shows the dc_shell script that maps the States VHDL library to the UNIX directory 'lib" in the current working directory. On executing the analyze command, the intennediate files are placed in the “lib" directory. This is extremely useful because it prevents the working directory from being cluttered with files. If the analyze command was used without the -lib States option, then by default, the intermediate files are written to the work library.

The work library is mapped to the current working directory by default. To over-ride this default, one must use the *define_design_lib* command to map the work library to a particular unix directory. In general, it is recommended that one create a directory called “Work" in the current working directory and map the “work" library to it. To verify the above steps, execute the *report_design_lib* command at the dc_shell prompt:

```
dc_shell > report_design_lib States
```

VHDL Compiler does not support configuration declarations. Hence for synthesis one cannot have different entities in a design analyzed in different VHDL design libraries. Since packages are made visible by the “"use"clause, they can be analyzed into different design libraries. To analyze design entities in different VHDL design libraries one must elaborate them individually and link their db files. Say for example you have a top level entity “top" which instantiates an entity ‘leaf’: Say, design entity leaf is analyzed into design library lib1 and top into the work library. Then the entity leaf in the library lib I must be elaborated using the following command

```
dc_shell > elaborate leaf -library lib1
```

This shall create a db file for the design leaf which can be saved and used when linking in the design top

Synthesis, Optimization and Compile

Synthesis is the the process of achieving an optimal gate level netlist from HDL code. Therefore, synthesis includes both reading in the source HDL and optimization of the code.

Optimization, on the other hand, is a step in the synthesis process which ensures the best possible combination of library cells which meet the functional, area and speed requirements of the target design. Compile is the process which executes the optimization step.

In DC, “compile" is the command which executes the optimization. After the source VHDL or Verilog has been read into DC, on executing the compile command, a netlist for the source HDL is generated. Before executing the compile step, the target library must be specified, if it is not already specified in the *.synopsys_dc. setup* file.

Optimization constraints must be specified prior to compile. Compile has a number of options, including low, medium and high map efforts. The default option is a medium effort compile. In general, if one were merely running tests to check the logic inferred on compile, one should use the low map effort since it takes the least run time. The medium effort is recommended in most cases. The high map effort takes significantly longer compile run time.

```
read -format vhdl test.vhd
include constraints.scr
compile
write -f vhdl test -output test_netlist.vhd
```

Example shows a simple dc_shell script to read in a design, compile, and write out a netlist of the design in VHDL. The file *constraints.scr* must contain timing and area constraints.
The "help" command at the dc_shell prompt as shown below:

```
dc_shell > help report_reference
```

Classic Scenarios

Few classic scenarios faced by designers when using DC are

Case 1: You are linking a design and DC issues one of these warnings:

Warning: Unable to resolve reference xxx in yw. (LlNK-5)

Warning: Design test has 3 unresolved references.(UID-341)

Solution: These warnings could be because of one of the following reasons.

1.The *search_path* variable is incorrectly specified. Check the search_path variable to verify that the UNIX paths to the target and link library have been specified correctly. At the prompt use the following command:

```
dc_shell >  list search_path
```

2. The reference xxx is a subdesign that is instantiated in yyy with instance name u1.xxx does not have a db file which exists in one of the directories in the *search_path.* Read in the source HDL for xxx before reading in the HDL for yyy. If a db file already exists for reference xxx, modify the search_path variable to include the path to the *db* file.

Case 2: DC issues one of these warnings when executing a report.
Warning: Can't resolve reference LDl for cell ul (UID-233)

Warning: Can't find symbol ..."

This is similar to case 1. Once again, check to see if the *search_path* does include the path to the link library and the target library. If it does include the path to the link library and target library, then check to see if an instantiated

component in the HDL does exist in the *link_library*, *symbol_library* and *target_library*.Use the command shown below to verify that the LD1 library cell exists in the technology library libA.

```
dc_shell > find (cell, libA/LD1)
```

If the 'find"command results in an Error message, then use the *list -libraries*
command to find the exact name of the library. The library name could be libA.db instead of libA.
Every instantiated component must be referenced to a sub-design or a library cell.

If it is a library cell, then the target technology library or the link library must have a description for it. If it is a sub design, then the source HDL for the sub design must be read into DC, prior to reading in this file. If a *db* file exists for this sub-design, then the search path must include a path to this *db* file.

Case 3: DC issues the following error on reading in the source VHDL into DC. The library declaration in the VHDL file is as shown below.

```
library test ;
use test.pack.all;

Error: The library 'test' is not mapped to a directory. (LBR-6)
```

Solution: This Error occurs when the VHDL design library "test" is not mapped to a valid unix directory. Use the following command at the dc_shell prompt

```
dc_shell >  define_design_lib test -path <the unix path to library files>
```

Case 4: DC issues the following warning message when reading in a VHDL file.
Warning: The library 'test is mapped to the directory 'lib' which is not writable. The strict VHDL analyzer will not be able to be invoked. (HDL-213)

Solution: This error occurs when one has executed the *define_design_lib test -path ./Ilb* command but there does not exist a directory *lib* in the specified location, or it exists but the user does not have write permission in that directory.

Case 5: During compile DC issues the following warning:
Warning: The cache_write directory xxx is not writable. (SYNOPT-ll)
Solution: Reading and writing to the cache is controlled by the *cacheJead* and *cache_write* variables. Ensure that these variables point to your home directory or to any shared cache that might exist for your design team.

Case 6: When reading in the design database *(db* file), DC issues the following error. What could be the cause?
Error: db file is corrupted. (EXPT-18)

Solution: It likely that you are using different versions of Synopsys tools. In other words, the db file was generated in version 3.1b of the DC while you are trying to read the db file now into v3.0b. In short, db files are backward compatible but not forward compatible. 3.Ox generated db files can be read into subsequent versions ofthe DC like 3.lx, but not vice-versa. To check the version of DC, use the following command:

```
dc_shell > list product_version
```

Case 7: In DA, the schematic shows mere boxes instead of the actual symbols for gates.
Solution: Ensure that the symbol library (.sdb file extension) for the technology library is available and specified by the *symbol_library* variable. Also, verify that the *search_path* variable in your *.synopsys_dc. setup* file includes the path to where the symbol library file is located. After doing the above, execute the following steps to verify:

```
dc_shell > read -f db <your_sdb_file>
dc_shell > create_schematic /* Or select View -> Recreate in DA*/
```

If DA still shows boxes instead of actual symbols, it is likely that the symbol library and the technology library do not match in case for the pin names or cell names.

Unit II
VHDL/Verilog Coding for Synthesis

General HDL Coding Issues, VHDL vs. Verilog: The Language Issue, Finite State Machines, HDL Coding Examples, Classic Scenarios.

Introduction

The design issues in the coding of state machines like state encoding, registered outputs, synchronous resets, asynchronous resets and "fail-safe" behavior of state machines are crucial for effective synthesis of state machines. The design and synthesis of clocked synchronous state machines using the DC is discussed.

Finite State Machines

A finite state machine (FSM) consists of a current state (P) and a next state (N), inputs (I) and outputs (0). State Machines can be classified as *Mealy* or *Moore Machines* depending on how the outputs are generated.

Mealy Machines

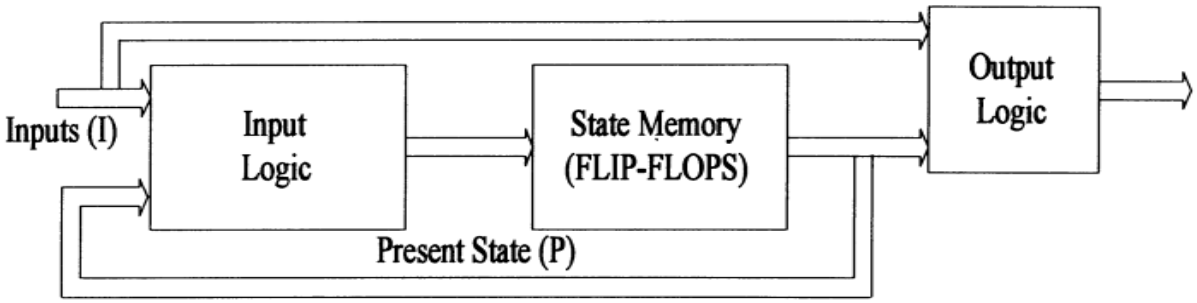


Figure *Mealy Machine*

A sequential state machine whose outputs depend on both the current state and the inputs is called a Mealy machine.

Functionality can be expressed as,

Next state (N) = function [current state (P), Inputs (I)]

Outputs (0) =function [current state (P), Inputs (I)]

Moore machine

A Moore machine is one in which the outputs are a function of only the current state and independent of the inputs

Functionality can be expressed as,

Next state (N) =function [current state (P), Inputs (I)]

Outputs (0) = function [current state (P)]

The next state logic and the output logic are purely combinational while the present state consists of sequential memory elements (flip-flops). Each active clock transition causes a change of state from the present state to the next state.

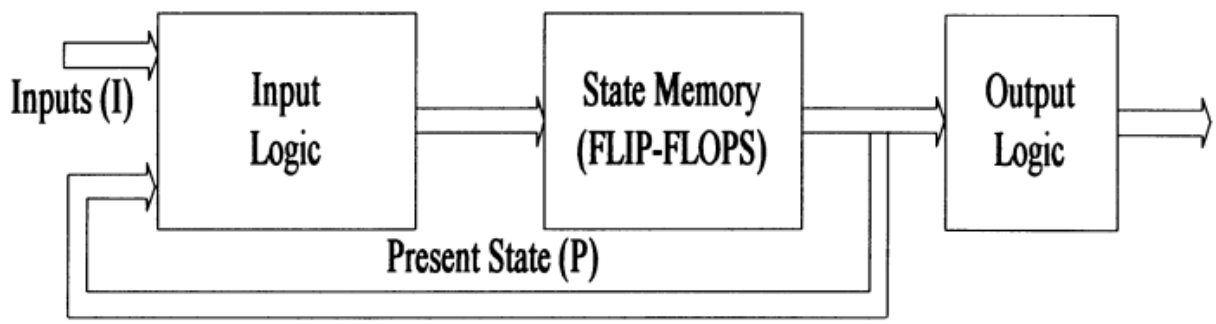


Figure *Moore Machine*

State Encoding

The concepts of current state and next state are vital to any state machine. Flip flops in a state machine serve as memory elements keeping track of the current state. Each possible state of the state machine can be assigned a unique binary code. This is called state encoding.

At any given instant, the current state of the state machine is determined by the binary values in the flip flops and their corresponding encoding. Thus n flip flops will encode a maximum of 2^n states. Alternatively, one can assign one flip flop to each state. Thus n flip-flops will represent n states. This is called the *One-hot* method of encoding. Since the state machine can only be in one state at a given

time, the outputs of only one of the flip flops is true and hence the name One-hot. The use of one flip flop for each state could result in greater silicon area.

The advantage of this method lies in that no combinational logic is required to decode the values of the current state in the state flip flops, since each state has only one flip flop. This makes the one-hot state machine the fastest implementation.

HDL Coding Examples

Consider a Mealy machine and its possible ways of coding the same in both VHDL and Verilog.Consider a Mealy machine with one input (X) and one output *(Z)*.

When X =0, the current state of the state machine remains unchanged and output Z remains at O.

When X =1, the state machine makes a transition from one state to the next binary state i.e.., 00 -> 01 -> lO -> 11 -> 00...

The output Z is equal to 1, only when the state is 11 and the input X is equal to 1, else Z is equal to 0 as shown in the state transition table and state transition diagram.

Input (X)	PRESENT_STATE	NEXT_STATE	Output (Z)
0	Q_i	Q_i	0
1	Q_i	Q_{i+1}	1 (when Q_3) else 0

Table 2-1. *State Transition Table*

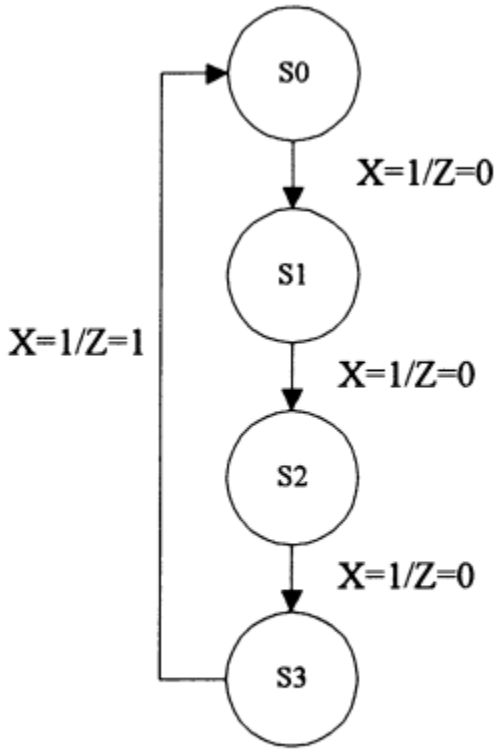


Figure *State Transition Diagram*

Coding Example 1:

VHDl Example of state machine

```
package states is
type state is (S0, S1, S2, S3); -- state can take one of these values.
end states;
use work.states.all;
entity test is
        port (X, clock : in bit;
            Z : out bit);

end test;

architecture trial of test is
signal ST : state;
begin
process
begin
        wait until clock' event and clock = '1';
        if  X='0' then Z = 0;
```

```
        else
                case ST is
                when S0 => ST <= S1 ; Z <= '0';
                when S1 => ST <= S2 ;Z <= '0';
                when S2 => ST <= S3; Z <= '0';
                when S3 => ST <= S0 ;Z <= '1';
                end case;
        end if;
end process;
end trial;
```

Verilog Example of state machine

```
module fsm1 (X, clock, Z);
input X, clock;
output Z;
reg Z ;
// state can take one of these values
parameter S0 = 2'b00, S1 = 2'b01,S2 = 2'b10,S3 = 2'b11;
reg [1:0]  ST;
/* Sequential logic of FSM */
always @(posedge clock)
begin
   Z = 1'b0;
  if (X == 1'b0) begin
        Z = 1'b0;
        ST = ST;
        end
   else
        case (ST)
          S0: ST = S1;
```

```
        S1: ST = S2;
        S2: ST = S3;
        S3: begin
            ST = S0;
            Z = 1'b1;
          end
      endcase
  end
  endmodule
```

Example shows the state machine using a case statement and a wait statement. In the event of the input X being 0, the output Z always remains 0 and no state transition occurs. The DC interprets that the state remains unchanged, if not mentioned. In other words, DC will maintain the current state.

Coding Example 2:

VHDl Example

```
package states is
type state is (S0, S1, S2, S3); -- state can take one of these values.
end states;
use work.states.all;
entity test is
        port (X, clock : in bit;
            Z : out bit);
end test;
architecture trial of test is
signal CURRENT_STATE, NEXT_STATE: state;
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of trial : architecture is "CURRENT_STATE";
begin
```

```
COMB : process (CURRENT_STATE, X)
begin
        case CURRENT_STATE is
        when S0 =>
        if  X = '0' then
                Z <= '0';
                NEXT_STATE <= S0 ;
        else
                Z <= '0';
                NEXT_STATE <= S1;
        end if;
        when S1 =>
        if  X = '0' then
                Z <= '0';
                NEXT_STATE <= S1 ;
        else
                Z <= '0';
                NEXT_STATE <= S2;
        end if;

      when S2 =>
      if  X = '0' then
              Z <= '0';
              NEXT_STATE <= S2 ;
      else
              Z <= '0';
              NEXT_STATE <= S3;
      end if;
      when S3 =>
      if  X = '0' then
              Z <= '0';
```

```
                NEXT_STATE <= S3 ;
        else
                Z <= '1';
                NEXT_STATE <= S0;
        end if;
        end case;
end process;
SYNCH : process
begin
wait until clock' event and clock='1';
                CURRENT_STATE <= NEXT_STATE;
end process;
end trial;
```

Verilog example

```
module fsm (X, clock, Z);
input X, clock;
output Z;
reg Z ;
parameter [1:0]
        S0 = 2'b00,
        S1 = 2'b01,
        S2 = 2'b10,
        S3 = 2'b11;
reg [1:0]  CURRENT_STATE, NEXT_STATE;
//synopsys state_vector CURRENT_STATE
always @(posedge clock)
begin
     CURRENT_STATE = NEXT_STATE ;
end
```

```
always @(CURRENT_STATE or X)
begin
  case (CURRENT_STATE)
 S0: if (X == 1'b0) begin
       Z = 1'b0;
       NEXT_STATE = CURRENT_STATE;
       end
      else begin
       Z = 1'b0;
       NEXT_STATE = S1;
       end
 S1: if (X == 1'b0) begin
       Z = 1'b0;
       NEXT_STATE = CURRENT_STATE;
       end
      else begin
       Z = 1'b0;
       NEXT_STATE = S2;
       end

 S2: if (X == 1'b0) begin
      Z = 1'b0;
      NEXT_STATE = CURRENT_STATE;
      end
     else begin
      Z = 1'b0;
      NEXT_STATE = S3;
      end
 S3: if (X == 1'b0) begin
      Z = 1'b0;
      NEXT_STATE = CURRENT_STATE;
```

```
        end
      else begin
        Z = 1'b1;
        NEXT_STATE = S0;
        end
  endcase
  end
```

Example shows another approach to coding the same state machine. This form of coding tells DC that the design is a state machine without having to set the state vectors after reading in the design. This is possible by use of the *state_ vector_ attribute.*The *state_vector* attribute on the architecture is assigned a value which is the name of the state signal. The design has been divided into two separate processes. The first process COMB, describes the combinational part of the design, while the second process SYNCH, describes the synchronous part of the design.

Registered Outputs

Outputs when generated by combinational logic could result in glitches. To avoid glitches in the outputs, designers infer registered outputs. Notice that in coding example 1 the output Z is a registered output while in Example 2, the output Z is not registered.One of the advantages of the approach shown in example 2 is that it separates the synchronous part of the design from the combinational parts.This style of coding provides the designer complete control over the combinational and sequential parts of the design, making the debug process less complicated. Hence, the recommended FSM coding style for synthesis is to use two separate processes, one for the combinational and the other for the sequential parts of the design.

Enumerated Types and Enum_encoding

Another approach to coding FSMs in VHDL is by the use of *enumerated types* with the use of the *enum_encoding* attribute. In this approach, one must declare the list of all possible values of the type state (say, SO, SI, S2, and S3 as in example 1). The values (SO, SI, S2, S3) can be either an

identifier (sequence of letters, underscores and numbers) or a character literal ('A', 'B'). The VHDL Compiler does default encoding of the enumerated literals. That is, by default, the enumeration values are encoded into bit_vectors, the first enumerated literal being assigned 0, the next 1 and so on depending on the number of values. Minimum number of bits are used in the encoding. For example, to encode four states two bits are used. To override the default enumeration encodings, the *enum_encoding attribute* can be used. The *enum_encoding* attribute must immediately follow the type declaration and must be a string containing a series of vectors, one for each enumerated literal in the type declaration. Again, the first vector corresponds to the encoding for the first enumeration literal and so on.

Coding Example 3:

VHDL Example

```
entity test is
        port (X, clock : in bit;
            Z : out bit);
end test;
architecture trial of test is
type state is (S0, S1, S2, S3);
attribute ENUM_ENCODING : string;
attribute ENUM_ENCODING of state: type is "00 01 10 11";
signal CURRENT_STATE, NEXT_STATE: state;
signal  Z_int : bit;
begin
-- COMB and SYNCH processes same as in example 2.2
```

Verilog Example

```
module fsm (X, clock, Z);
input X, clock;
output Z;
reg Z ;
parameter [1:0] // synopsys enum states
   S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
reg [1:0]  /* synopsys enum states */ CURRENT_STATE, NEXT_STATE;
// combinational and sequential blocks same as in example 2.2
endmodule
```

Coding example 3 shows the use of *enumerated types* with the use of the *enum_encoding* attribute. The declaration, "type state is (SO, Sl, S2, S3),: defines the list of all possible values of the type state. By default, the enumeration values are encoded into bit_vectors, the first enumerated literal SO being assigned 0, the S I being assigned a 1 and so on. By using the *enum_encoding* attribute, the encoding of the different states is declared in the code. This approach to coding FSMs is the recommended approach. The combinational and sequential parts are in separate processes. Further, by the use of *enum_encoding*, one has control over the states and their encodings.

One-hot Encoding

The fastest FSM implementation is the one-hot method of encoding .In DC, the user will have to declare the FSM encoding style using the *set_fsm_encoding_style* command.

Coding Example 4:

VHDL Example

```
package states is
type state is (S0, S1, S2, S3); -- state can take one of these values.
  attribute ENUM_ENCODING : STRING;
  attribute ENUM_ENCODING of state : type is "0001 0010 0100 1000";
end states;
library work;
use work.states.all ;
entity test is
          port (X, clock : in bit;
               Z : out bit);
end test;
architecture trial of test is
  attribute STATE_VECTOR : string;
  signal ST : state;
  attribute STATE_VECTOR of trial : architecture is "ST";
begin
-- COMB and SYNCH processes same as in example 2.2
end trial;
```

Verilog Example

```
module fsm (X, clock, Z);
input X, clock;
output Z;
reg Z ;
parameter [3:0] // synopsys enum states
   S0 = 4'b0001, S1 = 4'b0010, S2 = 4'b0100, S3 = 4'b1000;
```

```
reg [3:0]  /* synopsys enum states */ CURRENT_STATE, NEXT_STATE;
//synopsys state_vector CURRENT_STATE
// combinational and sequential blocks same as in example 2.2
endmodule
```

Coding example 4 shows the same state machine described in example 1 using both the *state_vector* attribute and the *enum_encoding* attribute. *Enum_encoding* has been used such that the first flip-flop, when on, implies the state SO, the second flip-flop implies state SI, and so on. In general, the one-hot encoding style involves the use of one flip-flop for each state, the current state being determined by the flip-flop which is on.

General HDL Coding Issues

Discussed some basic issues related to HDL coding for synthesis such as VHDL types, unwanted latches, variables and signals and priority encoding. For a certain desired functionality, it is often possible to code HDL in a number of different ways. However, there are several guidelines that one can follow to develop a consistent coding style for synthesis.

VHDL Types

It is recommended that *std_logic* types be used for port declarations in the entity. This convention avoids the need for conversion functions when integrating different levels of synthesised netlist back into the design hierarchy. The *std_logic* type is declared the IEEE *std_logic*_1164 *package.* Some of the examples in this chapter use the type 'bit" for the sake of simplicity and easy understanding.The type 'buffer" can be used when an output must be used internally.

Once declared as a buffer, all references to the particular output port must be declared as buffer throughout the hierarchy. This is often overlooked and one can run into problems during integration of different blocks. For the sake of consistency, it is recommended that one avoid the use of the type buffer.

```
entity buf is
port (a : in std_logic ;
        b: in std_logic ;
        c: out std_logic );


end buf ;
architecture test of buf is
signal c_int : std_logic ;
-- without c_int signal c will have to be declared as buffer
begin
process
        begin
        c_int <= a + b + c_int ;
end process;
        c <= c_int ;
end buf ;
```

Coding example shows an effective way to avoid the use of buffer types using internal signal declarations.

Signals and Variables

During simulation, variables are updated immediately unlike signals which require a delta time before being updated. Variables tend to simulate faster than signals but could mask glitches that might affect the functionality of the design. Further, variables tend to generate unexpected results during simulation.All signals that are being read in a process when not declared in the process sensitivity list, cause the DC to issue a warning.

Priority Encoding Structure

Coding with 'If' statements causes priority encoding logic to be inferred. In other words, DC assumes that the first ifcondition has a greater priority than the second and so on. To avoid this one must use 'base" statement.

Unwanted latches

Ensure that all signals are initialized. Further, when using case statements or nested if statements ensure that they are fully defined. A full specification will prevent latches from being inferred.

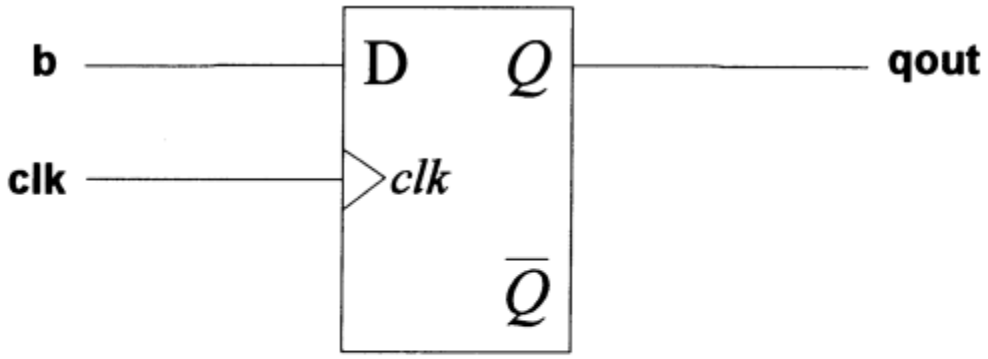


Figure *Latch*

VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
        port ( clk   : in std_logic;
               d     : in std_logic_vector (2 downto 0 );
               q     : out std_logic_vector (2 downto 0 ) );
end test;
architecture behavior of test is
begin
        process (clk, d)
        begin
                if (clk = '1') then
                        q <= d;


                end if ;
        end process;
end behavior;
```

Verillog Example

```
module latch(clk,q,d);
input clk, d;
output q;
reg q;
always @ (clk or q)
        if (clk == 1)
            q = d;
endmodule
```

In the above example,notice that the expected result when clk is not equal to 'I' is not specified. DC interprets this to mean that 'When clk=1 condition is not satisfied, retain the previous value of q': Hence,latches are inferred. After reading in the HDL, one does not have to compile the design to realize that unwanted latches have been inferred.

Parallel Case Directives

By definition, in VHDL, all the branches of a *case* statement are mutually exclusive. This means that in VHDL a *case* statement will synthesize to a mux rather than a priority encoder structure.

In Verilog, it is not the same for a *case* statement. The HDL Compiler interprets the conditions of the *case* statement to be mutually exclusive, only if the code explicitly implies it. Then, the *case* statement is parallel and will result in a mux being synthesized.

```
always @( w or x) begin
case (2'b11)
w:
   b = 10 ;
x:
   b = 01 ;
endcase
end
```

The case statement in example does not imply mutually exclusive conditions because the values of inputs w and x cannot be determined. However, if only one of the inputs is equal to 1 at a given time, then one can use the *//synopsys parallel_ case* directive to avoid synthesizing a priority encoder.

Full Case Directives

VHDL requires that a case statement be exhaustive. In other words, all combinations of the case expression must be covered. In Verilog this is not true and it is not required to enumerate all possible combinations of the case expression. But if all combinations are not covered, Synopsys HDL Compiler will infer latches for the conditionally driven variables. If one does not specify all possible branches and one or more branches can never occur, one can declare a case statement as full case with the *//synopsys full_case* directive.

```
input [1:0] a;
always @(a or w or z) begin
case (a)
2'b11:
   b = w ;
2'b00:
   b = z ;
endcase
end
```

Example shows a case statement which infers a latch for 'b' if the full_case directive was not specified. The *full_case* and *parallel_case* directives are specified after the case statement in the HDL code.

Unit III Links to Layout

Motivation for Links to Layout Floor planning, Link to Layout Flow Using Floorplan Manager, Creating Wire Load Models After Back-Annotation Re-Optimizing Designs After P&R. Design for Testability: Introduction to Test Synthesis, Test Synthesis Using Test Compiler

Motivation for Links to Layout Floor planning

Floor planning

A floorplanning is the process of placing blocks/macros in the chip/core area, thereby determining the routing areas between them. Floorplan determines the size of die and creates wire tracks for placement of standard cells. It. creates power ground(PG) connections.

RTL-level floor planning significantly reduces the front-end to back-end iterations and reduces total design turn-around time, with up to 10x faster synthesis run times compared to traditional synthesis tools and 100+ million gates design capacity to tackle the growing design sizes with ease with a compact memory

Need for floorplanning

Today's increasingly large and complex digital integrated circuit (IC) and system-on-chip (SoC) designs often contain tens of millions of logic gates. Ensuring that these designs will function as required demands the use of chip-level floorplanning.

In reality, all high-end chip designs employ some form of floorplanning activity early in the design flow. This commences with the design being partitioned into functional blocks, each of which are assigned very approximate gate-count and area values based on experience with previous designs. These area values are then used to create a rudimentary initial floorplan. Once the initial floorplan has been determined, preliminary timing budgets and constraints are assigned to the functional blocks, each of which is handed over to one or more RTL design engineers.

In the case of conventional flows, only very crude timing and area estimates are available at the RTL level. It is only after the physically-aware synthesis and in-place optimization (IPO) steps have been performed — and placed gate-level netlists are available for each functional block — that meaningful area and timing estimates are available. In turn, it is only when meaningful area and timing estimates are available that meaningful floorplanning activities can take place.

The end result is very expensive and time-consuming iterations. Obviously, the most cost-effective approach — in terms of engineering resources and time-to-market — is to start performing accurate floorplanning as early as possible in the design cycle.

Problems with conventional flows

There are three main stages in a conventional flow .

Stage 1: First of all we have the system architects who partition the design into functional blocks, associate estimated gate-counts and areas with these blocks, and establish an initial floorplan with associated chip-level and block-level timing constraints.

Stage 2: Next we have a group RTL design engineers, each of whom deals with their own block. Each RTL block will eventually equate to around 400K gates.

Once a block of RTL has been created, InTime contends that synthesis and IPO will take about 7 hours followed by 3 hours to perform timing analysis and generate a timing report (about 10 hours total).

Stage 3: Finally, we have the system integrators who take all of the blocks from the RTL engineers along with their more accurate gate-count, area, and timing values — use these to generate a more accurate floorplan, and then use this to perform chip-level synthesis/IPO and timing analysis.

Using implementation tools to perform these activities makes the cycle times through conventional flows too long and problematic.

RTL floorplanning Procedure

In its simplest form, RTL floorplanning refers to the ability to take RTL that is ready for synthesis following functional signoff, and to use this RTL to provide gate-count, area, and timing estimations that are sufficiently accurate to perform meaningful floorplanning analysis.

In order to satisfy the requirements for RTL floorplanning, InTime has two related applications called Time Planner and Time Director. Time Planner is used by system architects and system integrators to perform chip-level floorplanning and analysis functions. Time Director is used by RTL design engineers to provide gate-count, area, and timing estimations at the block level.

Stage 1: The flow starts as the system architect uses Time Planner to establish the initial floorplan and to generate the associated chip and block-level timing budgets and constraints. At this stage of the process, Time Planner will accept gate-count, area, and timing estimates for blocks for which RTL is not yet available, and it will generate gate-count, area, and timing predictions for blocks whose RTL is available.

Stage 2: As for a conventional flow, the RTL design engineers create and functionally verify the RTL corresponding to their blocks. In the conventional flow, however, the engineers would now have to run compute-intensive and time-consuming implementation tools that take about 10 hours per iteration in order to obtain accurate timing information.

Stage 3: As each RTL block is completed, it is handed over to the system integrator, who starts to create more accurate floorplan representations of the chip-level design. By means of Time Planner, the system integrator can control the shapes of the various blocks, ranging from simply modifying the aspect ratios of rectangular blocks to creating more complex contours such as 'L', 'T', and 'U'-shaped blocks.

RTL Floorplanning significantly decreases the loading on engineering resources, dramatically reduces the chip's design cycle time, and can result in higher-performance chips.

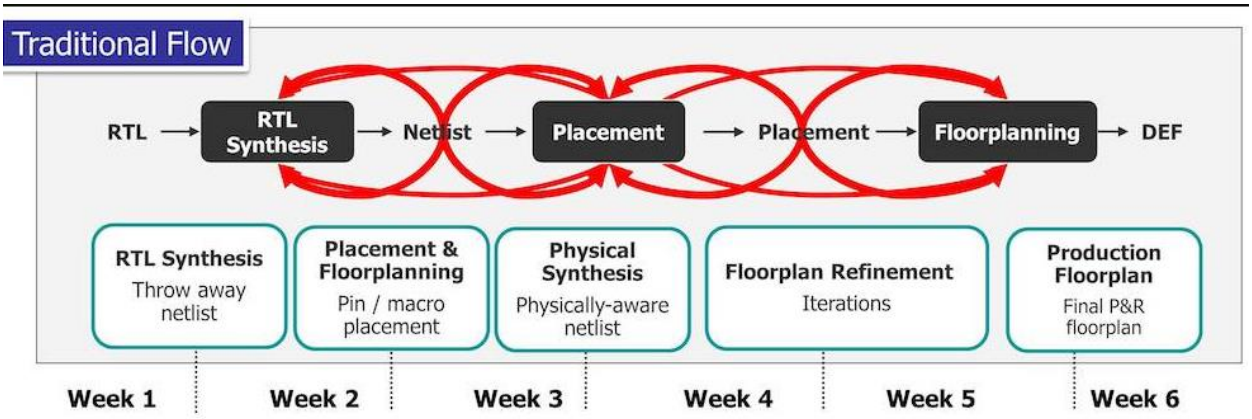
RTL floorplanning working

After the circuit partitioning phase, the area occupied by each block (subcircuit) can be estimated, possible shapes of the blocks can be ascertained and the number of terminals (pins) required by each block is known. In addition, the netlist specifying the connections between the blocks is also available. In order to complete the layout, we need to assign a specific shape to a block and arrange the blocks on the layout surface and interconnect their pins according to the netlist. The arrangement of blocks is done in two phases; Floorplanning phase, which consists of planning and sizing of blocks and interconnect and the Placement

phase, which assign a specific location to blocks. The interconnection is completed in the routing phase. In the placement phase, blocks are positioned on a layout surface, in a such a fashion that no two blocks are overlapping and enough space is left on the layout surface to complete the interconnections.

The blocks are positioned so as to minimize the total area of the layout. In addition, the locations of pins on each block are also determined. The input to the Floorplanning phase is a set of blocks, the area of each block, possible shapes of each block and the number of terminals for each block and the netlist.

If the layout of the circuit within a block has been completed then the dimensions (shape) of the block are also known. The blocks for which the dimensions are known are called fixed blocks and the blocks for which dimensions are yet to be determined are called flexible blocks. Thus we need to determine an appropriate shape for each block (if shape is not known), location of each block on the layout surface, and determine the locations of pins on the boundary of the blocks. The problem of assigning locations to fixed blocks on a layout surface is called the Placement problem. If some or all of the blocks are flexible then the problem is called the Floorplanning problem. Hence, the placement problem is a restricted version of the floorplanning problem. If one asks for planning of the interconnect in addition to floorplanning, then it is referred to as the chip planning problem . Thus floorplanning is a restricted version of chip planning problem.



Link to Layout Flow Using Floorplan Manager

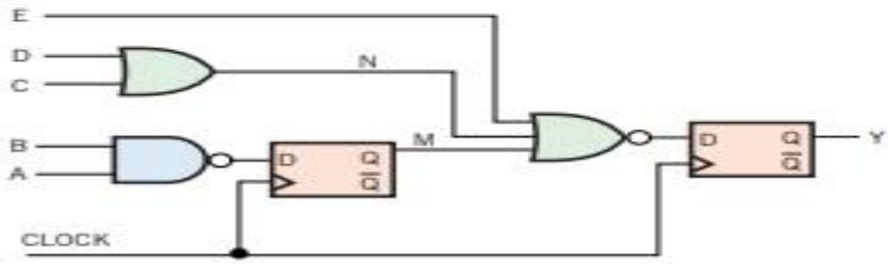
Engineers like to make design decisions as early in the design process as possible. Good decisions early on help define design parameters and eliminate incorrect design paths. A decade ago, commercial logic-synthesis tools from companies such as Synopsys focused on digital-chip analysis and design planning at the gate level. Analysis at the gate level is sufficient for design complexities to around 50,000 to 100,000 gates. Unfortunately, system-on-a-chip (SOC) complexities reaching into the tens of millions of gates have made gate-level design planning inadequate. Making design decisions at the RTL, before synthesis, is desirable. However, without the structural information that is part of a gate-level design description, it is hard to estimate design parameters, such as on-chip timing delays, power dissipation, and chip size.

RTL simulators provide information about a chip's speed but are more useful for functional verification and do not give the type of design-planning information. Designing at an RTL and a gate level are very different. RTL-design descriptions, such as the Verilog example in Figure a, include logic operation on a clock-cycle basis along with an implied design architecture. A logic-synthesis tool takes the RTL description and converts the design to a gate-level description (Figure b). Synthesis preserves the architecture and attempts to meet user-defined constraints, such as area and timing, in the gate-level description.

The RTL design is technology-independent; it includes no process information or information about what design library you will use to implement the design. Logic synthesis creates a gate-level description using cell-library information. Logic synthesis uses a targeted library, with its implicit target-process information, to determine which library elements are available for the design and to synthesize a circuit that meets your design constraints. Although logic-synthesis tools have made possible an orders-of-magnitude improvement in design productivity directly leading to SOC-design feasibility,inherent problems exist in today's typical logic-synthesisbased chip design, resulting from timing models used during different design stages.

```
module FLIP_FLOP_COMB (Clock, A, B, C, D, E, Y);
    input Clock, A, B, C, D, E;
    output Y;

    reg M, N;
    reg Y;

always@(C or D)
    N = (C | D);

always @(posedge Clock)
    begin
        M <= ! (A & B);
        Y  = ! (N | M | E);
    end

endmodule
```

(a)



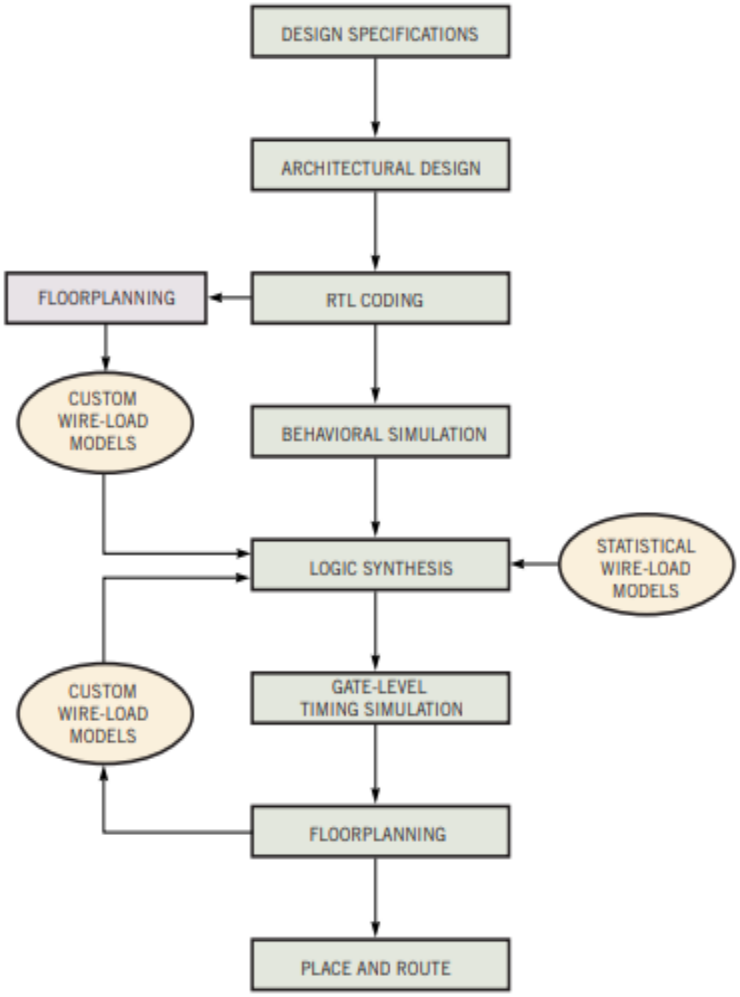
(b)

Below figure 2 shows a typical synthesis-based design flow. When you invoke a synthesis tool, it has no concept of the design's physical implementation.This model uses estimates for parasitic-interconnect and load-dependent delays that are average values based on previous designs using this technology.

Although statistical wire-load models may have been adequate for most designs greater than 0.5 mm, with deepsubmicron processes at 0.35 mm and smaller, these models are inaccurate. After you physically implement a design with place-and-route tools, the resulting logic may have very different timing characteristics, resulting in either a waste of silicon or a design that fails to meet timing requirements. The former problem wastes money; the latter definitely means redesign, resynthesis, and another place-and-route run.

Synthesis and place-and-route iterations are expensive in time spent and in financial cost, both in real money and in "lost-opportunity time." Floorplanning tools you use before or after synthesis can create better wire-load models for the synthesis tool. These "custom wire-load models" are based on placement data that the floorplanner creates. Because they are design-specific, custom wire-load models are more accurate than statistical models although still not as accurate as back-annotated parasitic data you obtain from an actual placed-and-routed chip.

RTL performance-estimation tools need to have some type of floorplanning capability to be able to predict electrical performance with any reasonable accuracy.



At the RTL, you have a description of a design's behavior.You get structural information only after logic synthesis. True topological data comes only after physical implementation. The design's constraints, which include speed, power dissipation, signal-integrity effects, and reliability, depend on process, cell-library population, and design placement and routing. RTL estimation of these parameters is a daunting task.

` In addition, RTL estimation helps you decide which cell library to use for your design. You can supply information to the logic-synthesis tool that can help achieve timing convergence and minimize synthesis and place-and-route iterations. Finally, you can get an estimate of chip size for a specific process technology that,

along with speed and power estimates, helps you decide which chip package to use and gives an indication of chip cost.

Back-Annotation

Back-annotation is the process by which resistance, capacitance and delay information after place and route are specified back into the DC. The "set load" and "set resistance" commands can be used to back-annotate capacitance and resistance values. The "Standard Delay Fonnat" (SDF) is a de facto standard for back-annotating delay values. The main motivation for developing SDF was to represent intrinsic delays, interconnect delays, loading delays, timing checks, and timing constraints in an abstract tool independent model.

Standard Delay Format (SDF)

The DC can both read in SDF as well as output SDF. The SDF file written out from the DC can be read into Synopsys VSS (and other simulation tools) for simulation. Shown below are the steps to write SDF from DC after one has initially read in the source VHDL.

```
compile
change_names -rules vhdl
write -f vhdl current_design -hier -output netlist.vhd
write_timing -format SDF
```

If we are using a Verilog simulator, then one can write out SDF without any name changes, and the Verilog netlist from DC for simulation. Also, after place and route, if one can generate layout delays in SDF, DC provides a means to back-annotate the SDF information. This can be done using the *read_timing* command as shown below.

```
read -format db design.db /* read in hierarchical db of design */
current_design = top /* set the current design to top level of hierarchy */
read_timing -format SDF delays.sdf /* read SDF delay file */
```

After the floorplanning is complete, and before the design is passed tophysical layout (place and route), the design's timing behavior can be verified once more within the synthesis environment. This time, the more accurate net parasitics (capacitance), net delays, and cell delays are used in place of the values estimated by DC. The estimates (provided by a floorplanner) for the wire capacitance, net delays, and cell delays can be back-annotated into DC.

DC supports the following SDF constructs:
• INTERCONNECT and PORT for net delay
• IOPATH for cell delay
• SETUP, HOLD, SETUPHOLD for timing checks.
The DC always uses the INTERCONNECT construct for net delays in the SDF file written out by *write_timing*. When reading in an SDF file, if the PORT construct is used for net delays, it is first converted to an INTERCONNECT construct and then annotated to the design.

```
(CELL
 (CELLTYPE "FD1")
 (INSTANCE BITS_SEEN_regx0x)
 (DELAY
   (ABSOLUTE
   (IOPATH CP Q (2.031:2.031:2.031) (2.553:2.553:2.553))
   (IOPATH CP QN (2.963:2.963:2.963) (2.926:2.926:2.926))
   )
 )
 (TIMINGCHECK
   (WIDTH (posedge CP) (1.500:1.500:1.500))
   (WIDTH (negedge CP) (1.500:1.500:1.500))
   (SETUP D CP (0.800:0.800:0.800))
   (HOLD D CP (0.400:0.400:0.400))
 )
)
```

The above example shows the IOPATH, SETUP, HOLD and INTERCONNECT constructs from an SDF file. This example shows the cell delay for the library cell FDI having instance_name BITS_SEENJegxOx. The IOPATH construct shows the minimum, typical and maximum delays between the pins CP and Q as well as CP and QN. Also, the timing check constructs SETUP and HOLD are shown.

Users may specify which value (minimum, typical, maximum) should be read by DC using the following dc shell variables.

```
dc_shell> sdfin_fall_net_delay_type = maximum
dc_shell> sdfin_rise_net_delay_type = maximum
dc_shell> sdfin_fall_cell_delay_type = maximum
dc_shell> sdfin_rise_cell_delay_type = maximum
```

Design Compiler Input/Output formats is shown below

Format	Description	File extension
db	Synopsys internal database	.db
edif	Electronic Design Interchange	.edif
lsi	LSI Logic Corporation (NDL) netlist	.net
equation	Synopsys equation	.fnc
mentor	Mentor NETED do	.neted
pla	Berkeley (Espresso) PLA	.pla
st	Synopsys State Table	.st
TDL	Tegas Design Language (TDL) netlist	.tdl
Verilog	Cadence Design Systems, Inc.	.v
VHDL	IEEE Standard VHDL	.vhd
SDF	Standard Delay Format (Cadence)	.SDF
XNF	Xilinx Netlist	.xnf

Table 7-1. DC input/output formats and file extensions.

Design for Testability

Introduction to Test Synthesis

The ever-increasing density of ASICs, the whole-sale switch to surface-mount technology, and the growing interest in multi-chip modules (MCM), have resulted in testable designs becoming a greater priority. Thus far, designers have considered testability as an issue which comes into play at the very end of the design cycle. However, in the ASIC design flow based on synthesis, it is essential that designers develop a test strategy and address testability issues concurrently with other activities in the design cycle.

Logic synthesis results in a netlist which usually contains sequential non-scan cells and other combinational gates from the technology library. The primary objective of test synthesis is to improve the observability and controllability of the design. In other words,one must be able to detect stuck-at-O and stuck-at-l faults in the design.

Scan technique is the widely used DFT technique, and more importantly, is supported by most test synthesis tools. This technique involves replacing the sequential non-scan cells by scan cells of the scan style chosen by the designer.

Full Scan and *Partial Scan* are the two design methodologies supported by most test tools. In a full scan methodology, all the sequential cells in the netlist are replaced by scan cells. On the other hand, in a partial scan methodology, only some of the sequential non-scan cells are replaced by scan cells. The choice of the non-scan cells to be replaced by scan equivalents, is made based on area and timing constraints required by the design.Full Scan designs in general, achieve a higher fault coverage when compared to partial scan designs. The fault coverage of a partial scan design is dependent on the number of non-scan sequential cells replaced by scan cells.Further, design specific characteristics like sequential depth and sequential feedback loops, impact the fault coverage achieved using partial scan.

Scan Styles

The four commonly used scan styles are as follows:

1. Multiplexed Flip-flop
2. Level Sensitive Scan Design

3. Clocked Scan Cell
4. Auxiliary LSSD cell

Multiplexed Flip-flop

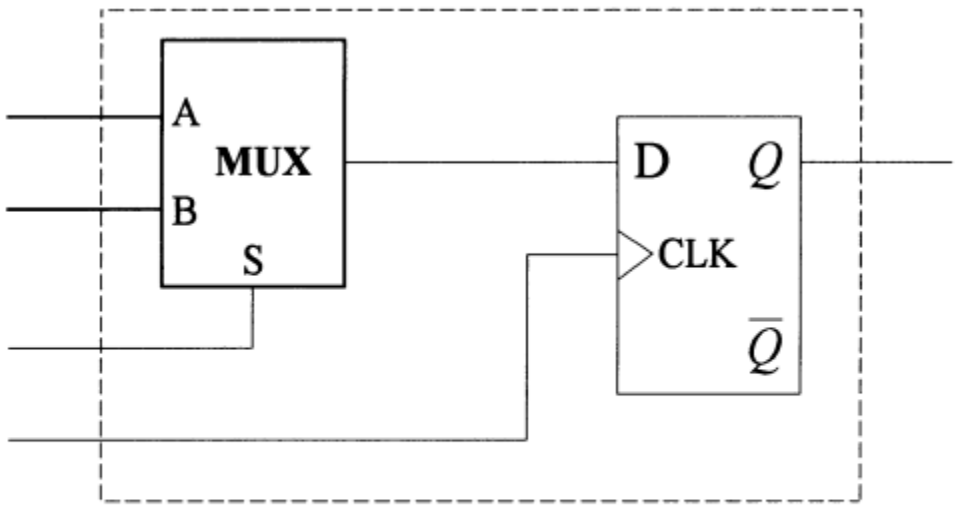


Figure *Muxed Scan Flop*

Figure shows a multiplexed flip-flop scan cell. Consder the multiplexed flip-flop scan style in which this scan style is supported by most ASIC vendors. For a multiplexed flip-flop scan style the scan ports required are *the scan-in, scan-enable and the scan-out ports.* The normal clock is used in the test mode in this scan style.

Scan Insertion

Scan cells have two different modes of operation : *The Normal Mode and The Scan Mode.* In the normal mode, the scan cell's functionality is same as that of the sequential non-scan cell. In the scan mode, the scan cells are linked in the form of a shift register.

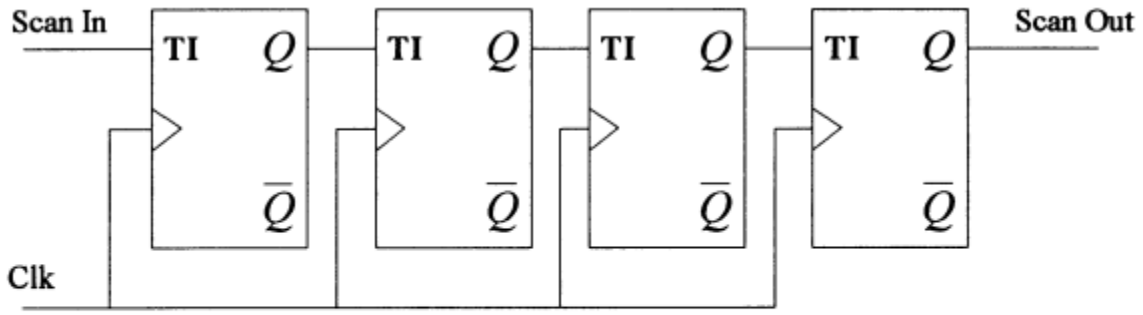


Figure *Scan cells linked to form a scan chain*

When scan cells are linked to form a *scan chain* as shown in above Figure ,all the scan cells are controllable and observable. Since shifting of data into the scan chain is perfonned serially, it takes N clock cycles to shift in a pattern into the scan chain, where N is the maximum length of the scan chain.Configurations with multiple scan chains are supported by most synthesis tools.

Scan insertion results in design overheads such as, the use of extra scan ports, an increase in silicon area due to use of scan flops, and greater timing delays due to the insertion of the scan cells for the sequential non-scan cells.It is possible to reduce the port overheads by sharing the scan ports with functional ports.

After inserting the scan logic in the design, the ATPG algorithm is used to generate test patterns. Full scan ATPG algorithm is combinational, while the partial scan ATPG algorithm is sequential. Test patterns can be generated in the format supported by the simulator used to simulate the test vectors. The common test fonnats are: VHDL, Verilog, and TSSI.

ASIC Vendor Issues

The test strategy used for a design is almost completely dependent on the requirements of the ASIC vendor. In other words, the requirements of the ASIC vendor must be taken into consideration prior to deciding on the test synthesis strategy.

Some of the critical issues which involve the ASIC vendor are

1. What scan style does the ASIC vendor support ?

ASIC vendors usually support only some scan styles and not all the available scan styles. Most ASIC vendors support the multiplexed scan flip-flop style.

2. How many clocks are supported by the tester when in test mode? Is there a limit on the number of wavefonns supported by the tester?
3. Is there a limit on the number of scan chains allowed ?Most ASIC vendors impose a restriction on the number of scan chains. Is there a limit on the length ofthe scan chain?
4. Is sharing of functional ports with *test_sean_in* and *test_sean_out* ports supported?
5. Does the vendor require that during scan-shift all the outputs have no switching i.e. all outputs are three-state outputs.
6. Does the vendor library support automatic pad synthesis?
7. What is the format of the test vectors required by the vendor?
8. Does the vendor accept parallel vectors or serial vectors for sign-off simulation ?
9. What is the maximum number of scan bits supported by the vendor's tester?
The total number of scan bits is simply the number of scan vectors multiplied by the number of flip-flops in the scan chain.
10. Do the formatted vector files require a specific naming convention ?
11. Is there a limit on the size ofthe vector files ?

Test Synthesis Using Test Compiler

Commonly Used TC commands

The definitions of basic TC commands should help to understand the TC flow with regard to actual dc_shell commands.

check_test: This command infers a default test protocol and performs a DRC check by simulating the test protocol. One must execute the *check_test* command before scan insertion as well as after scan insertion.

create test clock : This command is similar to the *create clock* command for the DC. TC automatically infers clocks during *check_test* by backtracking from the clock pins of registers. The *create_test_clock* command is used to specify the waveform and clock period in the test mode.

insert_test : The *insert_test* command replaces the non-scan sequential cells with scan equivalent cells and connects the scan cells to form a scan chain.

create_testyatterns : This command is used to generate the test patterns for the specified design. The command also writes out a *.vdb* file in the current working

directory.

Identifying Scan Ports

The TC uses the *signal_type* attribute to identify scan ports. Functional ports can be identified as scan ports, by assigning this attribute using the *set_signal_ type* command. The TC creates scan ports automatically if no functional ports are identified with the *signal_type* attribute. In the muxed flip-flop scan style, where normal clock is used as test clock, one must not associate a *signal_type* attribute, "test_clock" with the clock port.

Test Synthesis Flow Using the Test Compiler

The steps involved in test synthesis using the TC are outlined.

1. Read in the HDL code of the entire design into DC. In the example dc_shell command shown below, *VHDL_FILES* is a variable which can be assigned to a number of vhdl files

```
VHDL_FILES = {A.vhd B.vhd C.vhd TOP_LEVEL.vhd}
read -f vhdl VHDL_FILES
```

2. Set your *current_design* to the top level and specify the test methodology and scan-style.

```
current_design TOP_LEVEL
set_test_methodology full_scan
set_scan_style multiplexed_flip_flop
```

3. Another simpler alternative is to use the 'Test Smart Compile" approach. In this approach the user must specify the scan style before compile. The Test-Smart compile is turned on by specifying both the scan style (using *set_scan_style* command) and the test methodology (using *set_test_methodology* command) before *compile*.

4. Synthesize your design after specifying area and timing constraints.

```
include constraints.scr
compile
```

5. Specify the timing related test attributes as shown below.

```
test_default_period = 1000.0
test_default_delay = 50.0
test_default_bidir_delay = 550.0
test_default_strobe = 950.0
create_test_clock clock_port_list -waveform {450.0 550.0}
```

6. Analyze the testability of the design prior to scan insertion using the *check_test* command. A default test protocol is inferred and simulated on executing the *check_test* command.

```
check_test
```

7. Save the design database in *db* format prior to inserting scan.

8.Set the *current_design* to each of the different sub-designs and specify the scan chain allocation.

9. Set the *current_design* back to the top level current_design TOP_LEVEL.
10. Perform Scan Insertion using the *insert_test* command. The *-scan_chains* options implies the number of scan chains.

11. The next phase involves testability analysis after scan insertion. Analyze the testability of your design using the *check_test* command.

```
check_test
```

12. Execute ATPG on a sample fault list to check for any ATPG conflicts which might exist. The command shown below generates test patterns for 5% of the faults in the design:

```
create_test_patterns -sample 5
```

13. The next step is the JTAG synthesis phase. Group all the core logic except, three-state cells associated with three-state and bi-directional ports, into a separate level of hierarchy.

```
group -design_name Core -cell_name top filter( find(cell,"*") -except
{three_state_cell_list})
```

The variable *three_state_cell_list*, is a user-defined variable which lists the instances of three-state cells.

14. Set *current_design* to the TOP LEVEL ofthe design hierarchy.

```
current_design TOP_LEVEL
```

15. Specify the order of the boundary scan register (BSR) cells using the *set_jtag_port_routing_order* command.

```
set_jtag_port_routing_order {list_of_ports}
```

16. Perform ITAG insertion with the required options. Use the *–no_pads* option ifthe ASIC vendor library does not have pad cells.

```
insert_jtag -no_pads
```

17. Perform testabilty analysis after ITAG insertion using the *check_test* command.

```
check_test
```

18. Save the *db* file after JTAG synthesis.

```
write -f db -hier current_design -output after_jtag.db
```

19. Group all the JTAG logic into a separate level of hierarchy, and assign a *test_dont_fault* attribute on them, to avoid being considered in the fault coverage calculation. The design consists ofthe core instance surrounded by all the JTAG logic and three-state buffers.

20. In order to control and specify the characteristics of the desired pad cell, use the *set_pad_type* command. The DC inserts pads for all ports in the design which have the 'port_is""pad" attribute. This attribute can be applied using the *set_port_is_pad* command.

21. Execute ATPG using the following command. This creates a *.vdb* file which is a binary vector file.

22. Finally, generate the test vectors in the required fonnat.

```
write_test -format <format supported by TC>
```

Unit IV

Constraining and Optimizing Designs

Synthesis Background, Clock Specification for Synthesis, Design Compiler Timing Reports, Commonly Used Design, Compiler Commands, Strategies for Compiling Designs, Typical Scenarios When Optimizing Designs, Guidelines for Logic Synthesis, Classic Scenarios.

Introduction

After a design has been described in HDL and functionally simulated, the next step involves logic synthesis using DC. The core of the synthesis process is the constraints specified on designs and the timing reports generated by DC.

Synthesis Background

The Design Compiler attempts to meet two basic constraints or goals for optimization in the following order of priority

1. Optimization Constraints
2. Design Rule Constraints

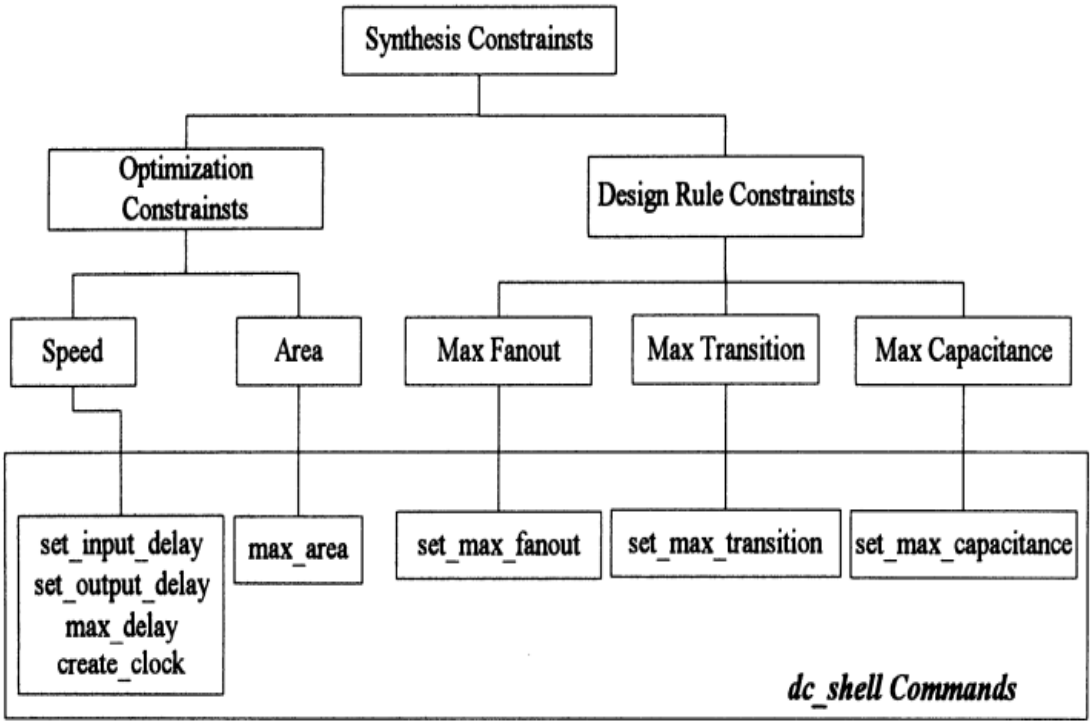


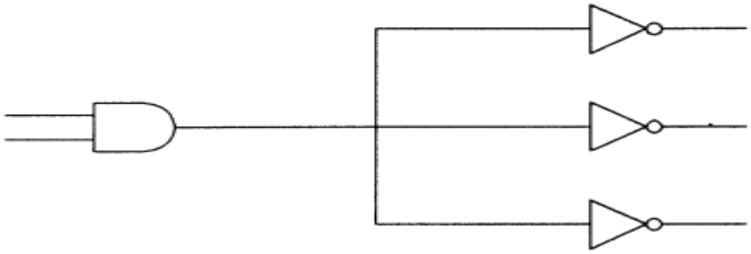
Figure 1. *Synthesis constraints and commands*

Figure 1 shows the two types of synthesis constraints and the related dc_shell commands. Optimization constraints are user specified constraints.The two optimization constraints are speed and area constraints. In addition to optimization constraints, the synthesis tool is required to meet another set of constraints called Design Rule Constraints (DRC). DRC are constraints imposed upon the design by requirements specified in the target ASIC vendor library.

Design Rule Constraints (DRC)

Max_fanout, max transition, max_capacitance are the three design rule constraints.

Consider an example



Max_fanout

Example shows the output pin of an AND gate driving the input pins of three inverters as shown in above Figure. The input pins of each of these three gates has a *fanout_load* attribute specified in the library. The sum of the fanout loads of each of the three input pins must not exceed the *Max_fanout* of the output pin of the AND gate. It is typically an integer, although each of these *fanout_load* values implies a certain standard load. One can find the fanout load on a specific input pin of a library cell (say, AND2 in library libA) using the following dc_shell command:

- **de shell> get attribute find(pin, "libA/AND2/i") fanout load**

Instead, if the library has a default *fanout_load* attribute set on the technology library, we can find this value using the following command

- **dc_shell > get_attribute IibA default_ fanout_load**

Max transition

Max_transition is the longest time for a transition from logic level 0 to 1, or vice-versa, for an entire design or for a specific net in a design. To be more specific, it is the RC time which is the product of the resistance (R) and the capacitive load (C). In the DC terminology, *max_transition* can be defined as the product of rise/fall resistance and the capacitive load on a net. When the user specifies a *max_transition* constraint in addition to the one already specified in the technology library,the more restrictive constraint will apply.

For example, if the library has a *max_transition* of 5 and the user were to specify a *max_transition* of 3, then the DC will try to meet a *max_transition* requirement of 3.

Max_capacitance

The *max_transition* design rule constraint does not provide a direct control over the actual capacitance of nets. The *max_capacitance* design rule constraint was introduced to provide a means to limit capacitance directly.This constraint behaves similarly to *max_transition,* but the cost is based on the total capacitance of the net instead of the transition time. The *max_capacitance* constraint is fully independent, so one can use it in conjunction with *max_transition. Max_capacitance* attribute can be specified on designs or ports. *MaxJransition, maxJanout* and *max_capacitance* can be used to control buffering in a design.*MaxJanout, max_transition* and *max_capacitance* constraints can be specified using the following commands :

```
set_max_transition <value> <design_name/port-name>
set_max_transition 5 test  /*where test is the name of a design */

set_max_fanout 5  <port_name/design_name>
set_max_capacitance 5 <port_name/design_name>
```

Optimization Constraints

Speed and area constraints as specified by the user are the optimization constraints. The speed constraints are specific delay constraints. One can specify timing constraints from one specific port/pin in the design to another provided such a timing path exists between the two specified points. In general, detailed timing constraints help get the best results from synthesis. To specify a max delay of 0 and expect the fastest design is not the best optimization strategy. Similarly for area, specify the expected area or a lower value than expected.

Specifying all clocks in the design using the *create_clock* command will constrain all synchronous paths in the design. To constrain the asynchronous paths in the

design, one can use the *max_delay* and *min_delay* commands.

Prior to version 3.0a of DC, *max_delay* and *min_delay* commands were used to specify timing constraints. But with 3.0a and subsequent versions, the recommended methodology is to use *set_input_delay* and *set_output_delay* commands instead. Only for asynchronous paths, must one use the *max_delay* and *min_delay* commands to specify point to point delays.

Max_delay constraints are imposed by explicit usage of the *max_delay* commands or implicitly due to clocks specified by the *create_clock* command. Similarly, the min_delay constraints are imposed by explicit *min_delay* commands or implicitly due to hold time requirements. However, DC fixes hold time requirements only when specified by the *fix_hold* command. The optimization script specifies constraints using the following dc_shell commands:

```
create_clock
set_input_delay
set_output_delay

set_driving_cell
set_load
```

Cost Functions

The synthesis tool performs optimization by minimizing cost functions – one for design rule costs and the other for optimization costs. These cost values are usually displayed during optimization. The optimization cost function consists of four parts in the following order of importance:

1. Max Delay Cost
2. Min Delay Cost
3. Max power Cost
4. Max area Cost.

Max Delay Cost

The Max Delay cost carries the greatest weight in cost calculations. It is the sum of the products of the worst violators and the weight in each *path group.*In

general, all the paths constrained by a clock are grouped into one path group. Thus, each clock in the design creates a separate path group.

All the remaining paths are grouped into the *default path group.* If no clocks are specified, then all paths default to the default path group. Since the synthesis tool is primarily path based, it is possible to attach different weights to different path groups. For example, consider a design with three clocks. If the weightage of each path group was the default of 1, and if the worst violation in each group was 1,2 and 3 respectively, then the max delay cost calculation is as follows:

Max_Delay Cost = (1 x 1) + (1 x 2) + (1 x 3) = 6.0

Min Delay Cost

Min Delay Cost is second in priority after max delay in cost calculation. The min delay cost calculation is independent of path groups. It is the sum of all the worst *min_delay* violators. The *min_delay* violation is calculated as the difference between the expected delay and the actual delay. A violation occurs when the expected delay is greater than the actual delay. Since *min_delay* is unaffected by path groups, the weightage assigned to paths has no effect on min delay calculations.

For example, a design with three paths with *min_delay* violations of 1, 2 and 3 will have a min delay cost of:

Min_Delay = 1 + 2 + 3 = 6.0

Max power Cost

Max power cost is only in the case of ECL technology. It is simply the difference between the current power and the max power specified. A violation implies that the former exceeds the later. Synopsys recently introduced a Power estimation capability for CMOS technology.

Max area Cost.

Max Area cost has the least priority in cost calculation. By default, the tool does not optimize for area once the timing constraints are met.

If explicit area constraints are specified, then DC performs area optimization.

If no explicit area constraints are specified, then area optimization occurs only if timing constraints are not met. Since synthesis results are dependent to a large extent on a number of factors such as

constraints, libraries and coding styles, optimization of a design is an iterative process.

Clock Specification for Synthesis

Clocks and clock delays are extremely important in applying constraints to a design. Practically all delays, particularly in synchronous designs are dependent on the clock. DC considers all clock network delays to be ideal.
If the design has a gated clock, the tool would not consider the delay through the gates leading to the clock, by default. One can override this default behavior by using *set_clock_skew* command to obtain non-zero clock network delay.

With the release 3.0a of Synopsys DC, the clock set up and hold check behavior was modified. *Clocks_at* command was replaced by the *create clock* command. Unlike earlier versions of the DC when 'time" was relative to zero, beginning with version 3.0a, timing for sequential paths is considered relative to clock edges. The DC automatically finds the relevant setup/hold relations by expanding the clock waveforms.

If one desires non single-cycle behavior, the *set_multicycleyath* commands must be used. One must define each clock in the design using the *create_clock* command. Clock trees are not usually synthesized using the DC.

In the event of a hand instantiated clock tree, during synthesis, one must place a *dont_touch* attribute on the clock network using the *dont_touch_network* command. This command ensures that the entire clock network in the design inherits a *dont touch* attribute.

Design Compiler Timing Reports

DC reports timing delays from clocks-to-clocks. In other words, DC reports timing from synchronous logic to synchronous logic or the logic between sequential cells. The DC timing report, by default, lists only the worst path in each path group. Each clock declared by the *create_clock* command creates a separate path group.

The *report_timing* command shows the path from primary input/clock pin to the primary output/data pin unless internal startpoints/endpoints have been

created at pins internal to the design by the explicit usage of *set_input_delay* and *set_output_delay* commands. When executed with the *–max_paths* 5 option the *report_timing* command causes the tool to report the 5 worst paths in each path group. Consider a simple example to explain timing reports generated by DC. The effect of timing constraints such as *set_input_ delay* and *set_output_delay* on DC timing reports is discussed.

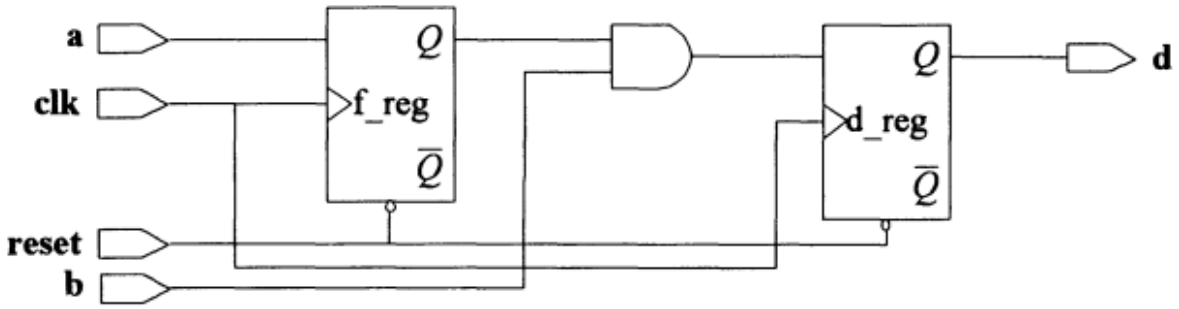


Figure *Design showing flip-flop delays*

Let us analyze these timing reports. The report gives the point in the design, which is usually a port or a pin of a library cell, the incremental delay through the cell (listed in the “Incr" column), and the 'Path" delay (listed under the 'Path" column) or the delay in the path upto that point. In other words, the path delays are calculated by adding up the incremental delays

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
f_reg/CP (FD2)	0.00	0.00 r
f_reg/Q (FD2)	1.42	1.42 f
U33/Z (AN2)	0.82	2.24 f
d_reg/D (FD2)	0.00	2.24 f
data arrival time		2.24
clock clk (rise edge)	5.00	5.00
clock network delay (ideal)	0.00	5.00
d_reg/CP (FD2)	0.00	5.00 r
library setup time	-0.85	4.15
data required time		4.15
data required time		4.15
data arrival time		-2.24
slack (MET)		1.91

Consider the first path beginning at the first sequential element f_reg and ending at the next sequential element d_reg in Figure. The rising edges of the clock are at 0 and 5 ns. So for f_reg assuming no clock network delays (which is

the default condition), the clock rise occurs at 0 ns, the clock to Q delay of FD2 flop is 1.42, the delay through AND gate is 0.82, giving a data arrival time of 2.24 at the data pin of d_reg. The register d_reg has its first rising edge at O. At this stage data from f_reg had not yet arrived. However,for the next rising edge at 5, the situation is different since data from f_reg arrived at 2.24 ns. Since the rising edge is at 5ns and the library has a setup requirement of 0.85 for FD2 flop, the latest a signal can arrive to avoid setup time violations is 5 - 0.85 = 4.l5. This implies that the constraint has been met with a positive slack of 1.91 ns.

Point	Incr	Path
clock (input port clock) (rise edge)	0.00	0.00
input external delay	0.00	0.00 r
a (in)	0.00	0.00 r
f_reg/D (FD2)	0.00	0.00 r
data arrival time		0.00
clock clk (rise edge)	5.00	5.00
clock network delay (ideal)	0.00	5.00
f_reg/CP (FD2)	0.00	5.00 r
library setup time	-0.85	4.15
data required time		4.15

data required time	4.15
data arrival time	0.00
---	---
slack (MET)	4.15

By specifying a clock period of 5 ns, we have *implicitly placed a max_delay* constraint of 4.15 from clock pin of f_reg to data pin of d_reg. In the second report, data arrives at 0 ns, and the clock requires that data arrive latest by 4.15 ns implying that setup time is met with a slack of 4.15 ns.

Report after setting an output delay

The timing report after setting an output delay of 2 ns on output port d using the *set_output_delay* command is shown below. Data must arrive 2 ns earlier at the output port d in order to meet timing requirements defined by flip-flops external to the port d.

Point	Incr	Path
---	---	---
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
d_reg/CP (FD2)	0.00	0.00 r
d_reg/Q (FD2)	1.37	1.37 f

```
d (out)                                  0.00     1.37 f
data arrival time                                 1.37

clock clk (rise edge)                    5.00     5.00
clock network delay (ideal)              0.00     5.00
output external delay                   -2.00     3.00
data required time                                3.00
-------------------------------------------------------------------
data required time                                3.00
data arrival time                                -1.37
-------------------------------------------------------------------
slack (MET)                                       1.63
```

Timing Report after setting an input delay constraint

The following report was generated after specifying an input delay of 3 ns on the input port A using the *set_input_delay* command. *Set_input_delay* is similar to *set_output_delay*, except that it accounts for timing delays at the input. For example, an *input_delay* of 3 ns on input port 'A', implies that relative to the rising edge of clock, elk, there is a delay of 3 ns, due to logic or otherwise prior to the port 'A'.

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
input external delay	3.00	3.00 r
a (in)	0.00	3.00 r
f_reg/D (FD2)	0.00	3.00 r
data arrival time		3.00
clock clk (rise edge)	5.00	5.00
clock network delay (ideal)	0.00	5.00
f_reg/CP (FD2)	0.00	5.00 r
library setup time	-0.85	4.15
data required time		4.15
data required time		4.15
data arrival time		-3.00
slack (MET)		1.15

Commonly Used Design Compiler Commands

Few basic DC commands and switches and their usage are discussed.

1. **dont_touch**

This is a very useful command, particularly when dealing with hierarchical designs. After one has specified the constraints and compiled a design to achieve the required results, it is often required that this design not be reoptimized when used in a larger design. In such cases, one would specify a *dont_touch* attribute on the instance of that design in the higher level design.

For example, say block A has been optimized to satisfaction and has been used in another design TOP as shown in Figure.

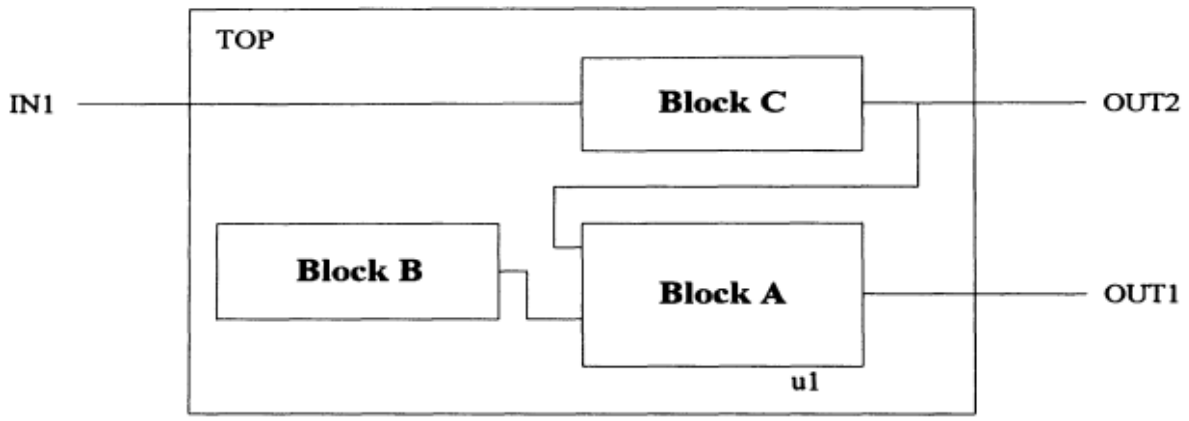


Figure . *dont_touch in design with hierarchical blocks*

Say the instance name of block A in TOP is ul, then the following dc_script performs the *dont_touch* step:

```
dc_shell > current_design = TOP
dc_shell > dont_touch u1  /*or alternatively, dont_touch find (cell,  u1) */
```

If we place a *dont_ touch* on Block A then the command is

```
dc_shell > current_design = BlockB
dc_shell > dont_touch find (design, BlockA)
```

The dont_touch attribute can be removed using the *remove attribute* command.

```
dc_shell > remove_attribute find (design, A) dont_touch
dc_shell > remove_attribute findcell, u1) dont_touch
```

2. **Flattening and Structuring**

Flattening a design essentially means converting the combinational logic into a two-level sum of products form. This is usually done to improve the speed of the design. For a design with over 20 inputs, flattening is almost never completed by the DC. If the number of inputs is less than ten, then flattening is more likely to be completed.

In the event of flattening not being completed by the DC, it simply proceeds to the next step after issuing a message that 'flattening is too expensive...': Thus it is not recommended that users simply read in a large netlist

and expect the synthesis tool to execute the flatten operation but instead use it judiciously.

Example

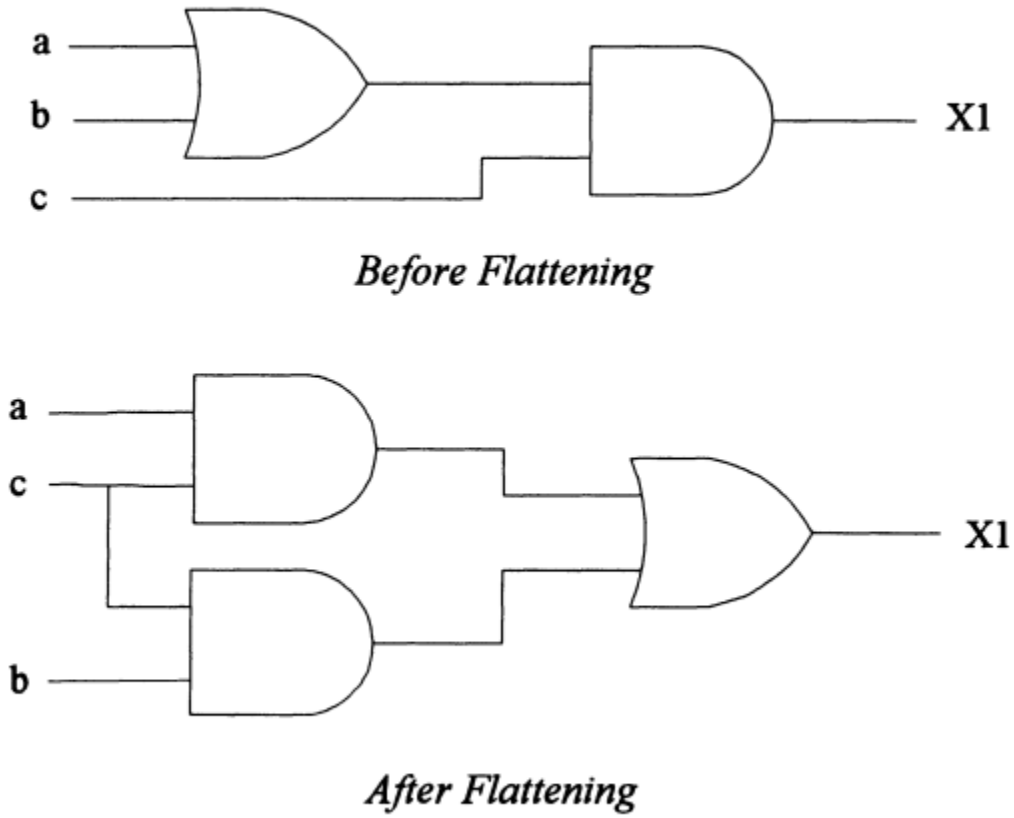


Figure *Gate structures before and after flattening*

Y1 = a + b ;

X1 = Y1 . c ;

On flattening the above equations

X1 = ac + bc ;

Structuring on the other hand, is used to improve the area or gate count of a design. It involves the addition of intermediate terms which are then shared by different outputs. In this sense, it can be considered as a reverse process of flattening.

Structuring is of two kinds, namely, *timing driven and boolean structuring.*

While timing driven structuring is executed by default, the latter is not. Timing driven structuring takes into account time delays when structuring the design, while Boolean structuring does not. Further boolean structuring results in a 2X to 4X increase in compile time. This can be very significant when dealing with large designs.

X1 = ab + ad ;

X2 = bc + cd ;

After structuring,

Y1 = b + d ;

X1 = a. Y1 ;

X2 = c. Y2;

Flattening and structuring can be specified using the following DC commands.

```
dc_shell > set_flatten true
dc_shell > set_structure -timing true
```

3. **Ungroup and Group**

The ungroup and group commands are used to remove and create levels of hierarchy in a design respectively. The ungroup command, when executed, causes the instances in the previously existing level of hierarchy to inherit the names of the level which was ungrouped. For example, if a sub-block A in hierarchical design TOP is ungrouped, an instance u2 previously contained in A will have the new instance name al/u2, where al is the instance name of the reference sub-block A. The same naming convention applies to nets in the design ungrouped. One can recursively ungroup the different levels in a hierarchy by executing the ungroup command with the *flatten* option.

The group command is the inverse operation of ungroup. One can group several instances in a design into a new level of hierarchy and specify the design name (the reference name) and the instance name of this design in the level of hierarchy above this newly created instance.

4. ***dont_use***

When one wishes to prevent the DC from inferring certain cells in the technology library, the "*dont_use* " command must be used. The cells with the

dont_use attribute are not used or ignored during optimization. For example, place a *dont_use* attribute on the library cell NAND2 in library libB as shown below:

```
dc_shell > dont_use libB/NAND2
```

5. **prefer**

The "**prefer** " command changes the priority of cells chosen by the Design Compiler during technology translation. Technology translation is essentially the process of mapping a netlist from one technology library to another. This command assigns the prefer attribute to the specified cells. For example, one can place a prefer attribute on the library cell IVB in library libB. This causes the DC to infer the IVB cell each time a cell of that functionality is required.

```
dc_shell >  prefer libB/IVB
```

6. ***set_default_register_type***

set_defaultJegister_type command specifies the default flip flop or latch to be used from the target library during technology translation.One can force DC to select a particular latch or flip flop from the target library by using the same command with a -exact option as shown below.

```
dc_shell > set_default_register_type -flip_flop -exact FDN
```

7. **Characterize**

The characterize command is used extensively in hierarchical designs. For example, consider a design TOP with two sub-blocks sub1 and sub2. Let us assume that both subl and sub2 have been compiled individually and have met their constraints. However, when instantiated in TOP, subl and sub2 have different constraints depending on constraints on TOP and the logic surrounding subl and sub2 in TOP.

The characterize command helps capture the constraints imposed on the sub design by the surrounding logic.

Strategies for Compiling Designs

The basic DC commands and the timing reports generated by DC is discussed. What is the best approach to compiling a design is the problem. The

compile strategy adopted is very much design dependent. It is possible to follow some general guidelines for compiling a design.

1. Capturing the entire design in one large HDL file, reading that file into DC, specifying the following constraint,

```
max_delay 0 -from all_inputs() -to all_outputs()
```

followed by executing the miraculous compile.

2. Dividing the design into too many hierarchical sub blocks. This is the other extreme of the strategy 1. This is not recommended for two reasons.
Firstly, managing the design with several sub-blocks can be rather cumbersome.

Secondly, optimization across hierarchical boundaries is not as effective as optimization within a block.

Typical Scenarios when Optimizing Designs

All the different strategies that one can experiment with when optimizing design using DC to be discussed. Assume that with just one sub-block of design and not the entire design.

***Scenario* 1**-- You have a design written in HDL. You have a very limited idea of the timing requirements. You simply wish to attain the fastest possible design.

A simple strategy to realize the optimal design is to experiment first with a default, medium effort compile, specifying absolutely no constraints before the compile step. This should give you a feel for the timing/area performance of your block. Then you specify your approximate timing (clocks and point to point timing constraints, if any) requirements.
A large number is used so that DC lists all the paths that fail to meet timing requirements in the design. If a number of paths are violated by a large margin, then you know right away that meeting your timing is likely to be a difficult/impossible task. On the other hand, if very few paths violate timing, then the next step would be to execute another compile with the default medium effort. Then, re-asses your paths in the report. If you see serious timing delays or very little improvement over the first timing report, then one or more of these must be attempted.
• Re-assess your code and consider alternate design partitioning.

• The technology library does not have cells to meet your timing.
• Timing requirements must be more realistic with regard to the capabilities of cells in the technology library.
• Identify any functional false paths or multi-cycle paths that might exist and specify them.

***Scenario* 2** -- You have written your source code, you know the detailed timing requirements, from characterize or otherwise.

Assess your results. Use the *group_path* command to assign higher weightage to paths which show greater violations. Use the *compile_default_critical_range* variable. The final step could be an incremental compile. This is used only to make very minimal improvements in timing, usually less than 2 ns. In general, the more specific you can be in specifying constraints, the better the synthesis results.

***Scenario* 3** -- You have fairly accurate timing requirements, but your main motive is to improve rather than merely meet the requirements. You are confident from knowledge of your library cells and earlier compile iterations that DC can meet timing, but your intent is to get the fastest possible design.

If constraints are already close to being met, then specify tighter constraints.
compile
You now meet timing but wish to improve upon this.
Now specify tighter constraints -- faster clock or tighter *max_delay* constraints for asynchronous paths. Execute report_timing again, they should now violate your delay constraint. Do not specify unrealistic constraints, like *max_delay* 0 for instance. Instead, gradually tighten constraints.

***Scenario* 4** -- Area is extremely critical in your design. While you think you could meet timing, area is an issue you would like to monitor right from the very start of your synthesis process. Given below are some tips for effective area optimization:

• Prior to the initial compile one must try and specify very accurate constraints to prevent DC from overkill ofnon-critical paths.
• After synthesis, execute the *check_design* command. Analyze the results to make sure there is no unused logic in design. Useful details about the design such as unconnected ports, feedthroughs, and multiple drivers are provided by this

command.
• Use the *report_resources* command to check implementations of resources in the designs and also on how many resources are inferred. There might be scope for sharing of resources by modifying the HDL
code.
• You could try ungrouping the hierarchy. Although this might improve area, it might make place and route task extremely difficult.
• Flatten appropriate unstructured random logic blocks using the *set_flatten* command on these blocks.

In short, synthesizing a design is an iterative process which can be aided by intuition. This intuition can be refined from extensive usage of the tool, analysis of the results, and a fair knowledge of the capabilities of the cells in the technology library

Guidelines for Logic Synthesis

The guidelines suggested here are not 'hard and fast" rules for effective synthesis. These are applicable in most cases and exceptions to these guidelines are likely.

1. For better results from synthesis, specify accurate point to point delays for asynchronous paths. Use the *create_clock* and *group_path* commands to constrain synchronous paths in the design. In general, the synthesis tool is tailored towards path optimization. Hence, it responds better to a greater detail of constraints.
2. Try to register outputs of the different design modules. This saves the designer from having to perform painstaking time budgeting. Constraining different hierarchical modules becomes easier for two reasons.
   The drive strength on the inputs to a block is equal to the drive strength of the average flip flop. Secondly, the input delays are equal to the path delays through a flip flop, given that the outputs of the driving hierarchical block are registered.
   3. Separate negative and positive edge flip-flops into separate hierarchical blocks. In other words, avoid having both kinds of flops in the same hierarchical module. This makes the debug process and timing analysis during synthesis much simpler. Moreover, this can help simplify test

insertion.

4. Group finite state machines and optimize them separately. State machine extraction and optimization process is more effective when the fsm is isolated. The group -fsm command can be used to achieve this.

5. The recommended size of a module for synthesis is in the range 250-5000. There are bound to be exceptions to this generalized recommendation.

6. Avoid having too many hierarchical blocks. Optimization across hierarchical boundaries is far less effective than when the boundaries do not exist. On the other hand having a large flat design with no hierarchy is not the solution.

7. Try to capture logic in the critical path into a separate level of hierarchy.DC does a better job of optimization when the critical path does not traverse hierarchical boundaries. This can be done by ungrouping existing blocks and re-grouping them using dc_shell scripts.

8. Compile Time: If your compile time is too long, then it is most likely due to one of the following reasons:

• You are using high map effort. Try the default medium effort. This is the recommended compile effort and hence is the default. The compile time for high map effort is dependent on the machine configuration and the size of the design.

• Your design is too large and must be broken down into smaller hierarchical modules.

• You have declared false paths which traverse hierarchical boundaries or any path exceptions specified in the design such as *set_multicycle* paths.

• You have glue logic at the top level of your design. Consider incorporating this into hierarchical sub modules using the ungroup/group commands.

• You are trying to flatten a design which is not appropriate for flattening. In general, use the 'flatten" switch only for random logic. For a design with over twenty inputs, flattening is almost never completed. If the number of inputs is less than ten, then flattening is more likely to complete.

• You have boolean optimization turned on. Again, this is appropriate only for random logic. If you do have random logic in your design, consider grouping it into a separate level of hierarchy and compile it separately with the flatten or boolean structuring switch turned on.

9. For datapath logic, consider the option of instantiating logic (like gates and muxes) or inferring them through user developed DesignWare libraries

10. Partitioning the design is extremely crucial to get the best out of synthesis. Identify signals with large fanouts and attempt to group the driving logic with the logic being driven into one hierarchical block.
11. It is always advisable to perform a preliminary round of synthesis and place and route so as to identify any serious issues which may require re-writing the HDL code.

Classic Scenarios.

Case 1 : You wish to find all the clocks defined in your design and their clock periods within a dc_shell script file. Using this information, you then wish to specify some constraints and attributes related to the clocks.

Solution: This can be done using the following commands

```
find (clock, "*")
/* This should find all the clock objects created in the design */
get_attribute find (clock, clk) period
/*This should find the value of the attribute "period" for the clock object
"clk" */
```

Case 2: Can one specify *dont_care* conditions for the condition branches of a case statement?

Solution: A typical scenario is when one cares only about certain inputs in a particular state but not the other inputs. DC does not support *dont_cares* for case statement conditions because of simulation mismatches. In the simulation world, a string to string matching is performed and this applies to
the *dont_care* conditions as well.

Case 3: The DC is unable to meet the timing for the path which is the worst violator. However, it does not seem to improve on other paths in the design which most certainly can be improved by merely swapping cells in those paths.

Solution : By default, DC creates a default path group and a clock group for each clock created. The default path group contains paths that do not terminate at a

clock. Only the worst violator in each path group affects the synthesis cost function. This can be changed by using the *group_path* command or modifying the value of the *compile_default_critical_range* variable from the default of 0.0 to a larger value. In general, set the *compile_default_critical_range variable* only in the last compile step. In other words, set constraints and perform one or more compile steps until the DC does not seem to improve its results. Then set this variable to a value (usually 2 or 3) then re-compile.
Setting this to a large value can increase the compile time significantly. The *group_path* command can be used to create explicitly a path group and specify the weight and critical_range of that group. No path can exist in more than one path group.

Case 4: A top level module has a few submodules and a hand-crafted clock circuitry at the top level. You wish to synthesize this design to gates but leave the clock logic at the top level intact. How does one go about accomplishing this?

Solution : The *dont_touch_network* command will propagate the *dont_touch* attribute throughout the hierarchy for the clock network. Since this command specifically works for clock networks, it is required that a clock object be defined (using *create_clock)* before this command is used. If the intention is to only maintain the clock logic at the top level the following script can be used to set the dont_touch attribute on all leaf cells at the top level of the design which constitute the clock circuitry.

```
/* Finding instances of all hierarchical cells at top level */
filter var1 "@is_hierarchical == true"
var2 = dc_shell_status
var3 = var1 - var2
dont_touch var3
```

Case 5: How does one find all the cells of a particular reference in a hierarchical design? In other words, you have a hierarchical design with the FDI (flip-flop) library cell used several times and you wish to get an actual count to identify if it is worth requesting a special low drive cell of the same functionality.

Solution: The simplest way would be to ungroup the design from the top level and use the *report_reference* command. Alternatively, if one prefers not to ungroup the design, a script which finds all the cell instances which reference the FD1 should accomplish the same. Then the total number of
cells in this list is counted.

Case 6: When are resource sharing decisions made - Is it during elaborate or during compile? Which tool license is used when resource sharing decisions are made, HDL Compiler or DC?

Solution: Resource sharing is done during the first compile and the license used is the HDL-Compiler license. One can actually prevent an HDL Compiler license from being used during compile. Another way to accomplish the same is to execute the *replace_synthetic* command before compile. This will, however, disable the high level optimizations that occurs during compile (including timing-driven resource sharing). This can impact the quality of results.

Case 7: A design has an address bus 32 bits wide of which only 2 bits go into a module. You create an extra level of hierarchy in DC using the group command and only 2 bits of the address needed go into the newly created module. DC brings in a1l32 bits into the module and does not connect the top 30. Is there a way to get rid of the unused bus ports?

Solution: One can remove unused ports using the *remove_port* command as shown below:

```
dc_shell > remove_port "a(26)"
```

The port name should be enclosed in quotes. However, there is a potential problem with this approach. Once you have removed these unused ports, the top level will not be able to link to the lower level. The reference in the top level will still include the unused ports, but the tool will not be able to find
these ports in the lower level since you have removed them.

Case 8: In a state machine process, if a state is supposed to remain the same under a certain condition, does the user have to explicitly write next_state < =

currentstate;Since nothing new is assigned to it shouldn't it maintain the state even if not specified?

Solution: If you do not have the *next_state* < = *current_state* statement in the combinational process statement, the DC will infer latches for the *next_state* signal

Case 9: Is there a way to control instance names inferred by DC during synthesis?

Solution: No, there is no way to control instance names except by adding pre-fixes and suffixes. This can be achieved using the following variables:

```
compile_instance_name_prefix = "U"
compile_instance_name_suffix = "S"
```

Unit V
Constraining and Optimizing Designs for FSM

Finite State Machine (FSM) Synthesis, Fixing Min Delay Violations Technology Translation, Translating Designs with Black-Box Cells, Pad Synthesis, Classic Scenarios

Finite State Machine (FSM) Synthesis

Finite State Machine synthesis involves a number of steps. The steps involved between writing the source code and generating the required state table representation of the state machine are as follows.

First the HDL source code is mapped to cells from a target technology library. Then the flip-flops in the design which hold the current state of the FSM must be identified.

The next step involves assigning specific codes to different states. This is followed by a grouping of the state flip-flops and their associated combinational logic into a separate level of hierarchy. Grouping helps to isolate the FSM from the rest of the design. Once grouped into a separate level of hierarchy, this sub-design can now be represented as a state table.

```
/*SCRIPT TO READ, COMPILE AND EXTRACT FSM */

read -f vhdl mealy.vhd
create_clock -period 10 clock -waveform {0 5}
compile -map_effort low
set_fsm_state_vector {ST_reg[0] ST_reg[1]}
set_fsm_encoding { "S0=2#00" "S1=2#10" "S2=2#01" "S3=2#11" }
group -fsm -design_name eg1_fsm
current_design = eg1_fsm
report_fsm
extract
report_fsm
```

The script reads the VHDL code into DC, compiles the design and extracts the state machine. The compile command maps the HDL code to target technology library cells. At this stage, DC is unaware that the VHDL description is a state machine. To verify if the DC understands that the design is a state machine, the *report_fsm* command can be used. Identifying the state vectors using the *set_fsm_state_vector* command, tells the DC that the design is state machine. Further, setting the state vectors helps the tool differentiate the state flip-flops from the flip-flops used for registered outputs. The DC by default, assigns the name, signal name (ST in this example) followed by _reg[i] (ST_reg[i] for this example), where i is the number of state vector bits, to the state vectors. In this case, it takes the values 0 and 1 since only two bits are essential.

The *set_fsm_encoding* command allows the designer control over the state encoding. While several encoding styles for FSMs exist, we will discuss the auto (default encoding style) encoding styles. The encoding style can be assigned using the *set_fsm_encoding_style* command. The *group-fsm* command groups the state flip-flops and the associated combinational logic into a separate level of hierarchy. On extraction, the state machine can be written out in state table format.

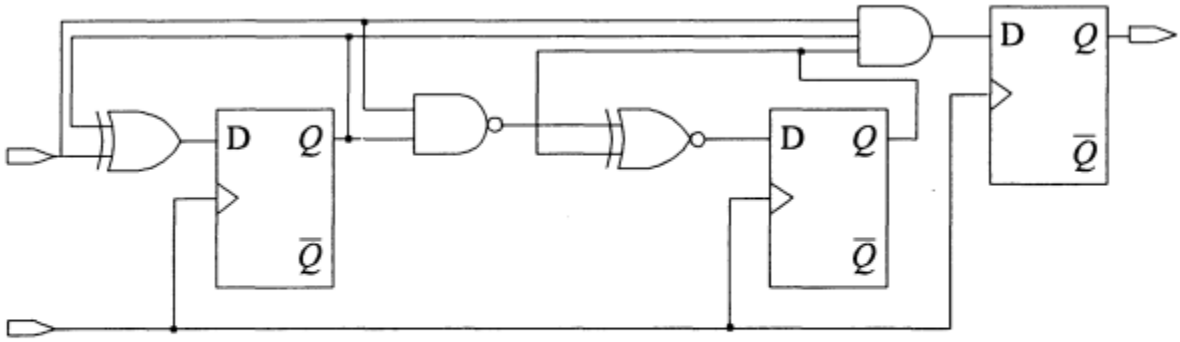


Figure *Top level design after synthesis*

Figure shows the top level design generated from the VHDL code. Notice that there are three flip-flops.

Procedure for FSM synthesis

1. Before a state machine has been extracted and after the group command,DC sometimes fails to group some of the surrounding logic which would then have made the state machine logically more optimal. The cells which are grouped are those in the transitive fan in/out of the state vector cells.After grouping, one might find two inputs to this grouped level of hierarchy which are the opposite (inverted) of each other. In this case, one must use the *characterize -connections* command with the *current_design* set to the top level, so that the connection attributes are passed on to the newly grouped level of hierarchy.

dc_shell script to characterize is as follows:

```
current_design = TOP
characterize -connections fsm_block
current_design = fsm_block
extract
```

2. Once the state machine is extracted, the design can be written out in state machine format or to the original RTL VHDL format by the following steps:

```
current_design = fsm_block
write -f vhdl fsm_block -o rtl.vhd
```

3. While the flip-flops inferred in the examples are all D-flip-flops, it is possible to force DC to map to specific flip-flops from the *target_library* using the *set_register_type –flip_floP cell_name* or the *set_register_type-flip_floP cell_name –exact* commands.

4. When the concerned nets are inputs to the state machine,one is to set the variable, *write_name_nets_same_as_ports* to true (this is false by default) before writing the design in EDIF. Then read the EDIF back into DC and follow the steps till extraction of the FSM. After reading in the edif file,the ports and the nets connected to them should have same names. It is advisable to do a *compare_design* between the new design read in and the old design in memory, to ensure that no changes occurred during the write step.

5. reduce_ fsm and set_ fsm_ minimize are two commands users tend to confuse. reduce fsm is more a command while set fsm minimize is more of a switch. reduce_ fsm should be executed after the extract command to reduce the transition logic between states. The *set_fsm_minimize* is turned on prior to compile so that the tool infers the minimum number of states required for the fsm.

6. Last the efforts must be made to clearly partition the design into control logic and data path elements. Reading in a large netlist and executing the extract command is not an effective methodology.

Fixing Min Delay Violations

Once the *max_delay* requirements imposed due to the setup constraints for the sequential cells have been met, DC then attempts to fix the minimum path delay requirements. Since the path delays are the maximum in the 'Worst case" timing analysis or 'Worst case" operating conditions, max delay requirements must be met in the "worst case" operating conditions.

The minimum delay requirements are set by the hold constraints for the sequential cells. Hold time problems are caused due to short delay paths between registers which cause the data signal to propagate through two adjacent flip-flops on a single clock edge. Since path delays are the shortest under 'best-case" operating conditions, hold time problems are maximum in these conditions. Hence,

hold violations have to be fixed under these conditions.

One approach to go about fixing both the setup and hold constraints is a two pass compile approach. In the first pass compile, fix the setup violations under the “Worst-case" operating conditions. Then set the operating conditions to 'best-case" for the second pass compile. Use the 'fix_hold" command to set an attribute 'fix_hold" on the clock objects for which hold constraints have to be met. The second pass compile should be with the compile switch "-only_design_rules" turned on. This should fix all the hold violations in your design. Also since under the 'best-case" operating conditions, the *max_delay* paths will have excessive positive slack, hold constraints maybe fixed at the cost of setup constraints. Such a situation can be avoided by adjusting the constraints such that the critical paths in the design appear critical under best-case conditions.

One of the ways to achieve that is by specifying a negative uncertainty on the clock by using the "set_clock_skew -minus_uncertainty" command. Hold time problems will generally occur in shift register structures or scan chains. Since by default DC treats the clock as ideal with no path delays, one must account for the network delay by using the command "set_clock_skew -propagated".

Technology Translation

Conversion of a design netlist from one technology library to another is called technology translation. This powerful capability helps compare performance across different ASIC vendor libraries. However, this utility has its limitations and works best with combinational logic.

Technology libraries differ in the cells they contain and in area and timing. Hence, after technology translation, optimization must be performed on the design to meet the original design constraints.

Technology *Translation in DC*

In DC, technology translation is performed by the *translate* command. In order to perform translation from one technology to another, the first requirement is the availability of both the existing library to which the netlist has been mapped and the *target_library*. Shown below are the steps involved in translating a design top from technology libA to technology library libB.

```
dc_shell > current_design = top
dc_shell > target_library = libB
dc_shell > link_library = libA
dc_shell > search_path = search_path + "path to the two libraries"
dc_shell > translate
```

The translate command replaces each cell in the design with the closest matching functional cell from the *target_library*. In case such a matching cell is not found then it is converted to a cell from the generic library. The *dont_use*, *dont_touch*, *set_default_register_type* and the *prefer* command are useful commands that affect the translation process.

Translating Designs with Black-Box Cells

When translating designs from one library to another, the DC performs an instance by instance functional comparison. In other words, translation requires that a cell in the current technology library and the target technology library both have the same functionality, if an instance of the cell is to be translated. The *translate* command does not translate black-box cells during technology translation.

However, there exists a simple trick to translate a black box cell. Black-box cells are cells with a function attribute which cannot currently be described in the Synopsys Library Compiler syntax or those which do not have a function attribute specified. Such cells have the 'b' attribute attached to them implying a black box cell. The *report_lib*command can be used to identify all the attributes on the cells in the library as shown below:

```
dc_shell > report_lib libA
```

It is possible that the design prior to translation has one such cell instantiated in it and the target library does contain an identical cell. Since the DC does not see any functionality described, it is unable to translate this particular instance. A design with black-box cells can be translated by the following steps:

1.Identify the black-box cells in your design and then find the equivalent cells in the *target_library*.
2. Create a translation library for these black-box cells. For example, if your netlist has a black-box cell 'mem"and your *target_library* contains an equivalent cell 'mem_new': then create a translation library which is essentially a module that instantiates the target cell mem_new, but with the same interface as the mem cell as shown in below Verilog example.

Verilog Translation Library

```
module mem (D0, D1, D2, D3, clk, Q0, Q1, Q2, Q3);
input D0, D1, D2, D3, clk;
output Q0, Q1, Q2, Q3;
mem_new I1 (.I0(D0), .I1(D1), .I2(D2), .I3(D3), .G(clk), .Q0(Q0), .Q1(Q1),
.Q1(Q1), .Q2(Q2), .Q3(Q3));
endmodule
```

3. After the translation library has been created, convert the design to the db format using the *read* and the *write* commands. You have now created a block around the cell in the *target_library* with an interface similar to the interface ofthe black-box cell in the current library netlist. Assuming that your translation library is called *translation. db*, your original library *orlginal.db*, and your new target technology library *new.db*, set the *link_library* variable as follows:

```
dc_shell >link_library ={translation.db original.db new.db}
```

Also, ensure that the *search_path* variable points to all the directories containing these libraries.
4. Execute the link command to translate the black-box cells in the netlist to the library *new.db*. During the link operation, the DC checks the *link_library* for cells beginning with the *translation.db* library, followed by *original.db* and finally, the *new.db*. On finding the mem design in *translation.db*, it links to the newly created design.

During translation, the mem cell is nothing but a sub-block with the mem_new instantiated in it, and mem_new is a cell in the *target_library* new.db. This level of hierarchy can later be removed with the *ungroup* command.

If there is no exact equivalent cell in the target library, you can create a structural model of the black box cell using primitives from the target technology library. Then, as in the above case, create a translation library with the same interface as the black box.

Pad Synthesis

Adding pads to your design is an essential part of the design process. One option is to instantiate pads after the core ofthe design has been implemented and simulated.

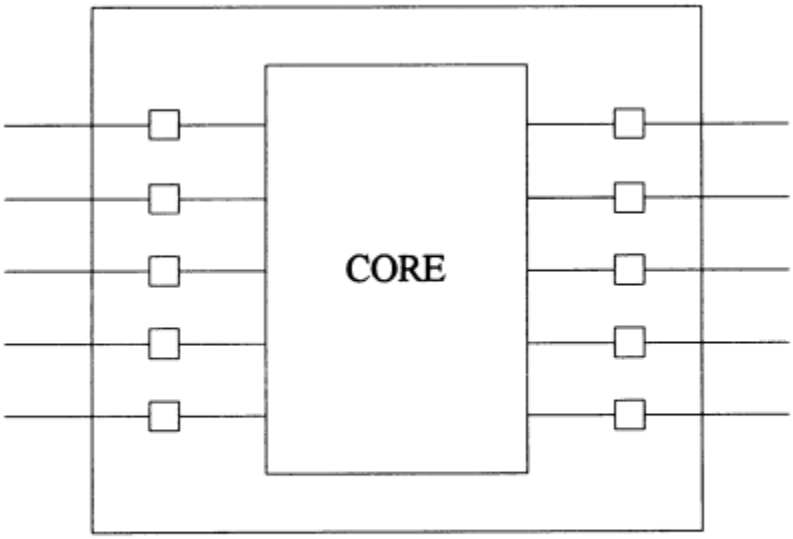


Figure *ASIC showing core and pad cells*

Figure shows an ASIC core with the pad cells. DC provides a means for automatic pad insertion. However, this is entirely dependent on the ASIC vendor library having appropriately modeled pad cells.

A pad cell in the Synopsys library is one which has the *pad_cell* attribute set to true. Also, one or more ofthe pins ofthe pad cell will have the *is _pad* attribute set on them. Hence, the first step to attempting pad synthesis is to ensure

that the technology library has pad cells modeled appropriately.The following commands can be used to determine all the pad cells in the technology library.

```
dc_shell > filter find (cell,libA.db/*) "@pad_cell == true"
```

If DC issues a message that it is unable to find the library libA.db, execute the *list -libraries* command at the dc_shell prompt. This command should list the UNIX file name and the actual name of the library. Having detennined the pad cells available in the library, the next step is to find the pin on the pad cell (say padA) that has the *is_pad* attribute. This can be done using the following command.

```
dc_shell> filter find (pin,"libA.db/padA/*") "@is_pad == true"
Performing filter on port 'A'.
Performing filter on port 'GZ'.
Performing filter on port 'Y'.
{"Y"}
```

Having determined the pad cells available in the technology library, the next step involves pad insertion. This is done using the *insert_pads* command. However, if we wish to control the kind of pad cell inserted by the DC, this can be achieved using the *set_pad_type* command. This command controls the attributes and properties of the pad cell synthesized by DC. To provide greater control, the *set_pad_type* command has a *"-exact' option* which helps the user explicitly specify the pad cell to be inserted from the library.

The *insert_pads* command does not bus together inputs into the same pad using bused pad cells. Such pad cells will have to be instantiated. The DC does not map to pad cells during the regular *compile* if the pad cells have required attributes.

Classic Scenarios

Case 1: We are performing trial compile runs. We do not wish that wire loads be considered in these trial runs. Can one prevent the DC from selecting a wire_load model for a design, or does it default to a particular wire load model?

Solution: By default, the DC automatically selects a *wire_load* model if not explicitly specified using the *set_wire_load* command. This can be prevented by setting the variable *auto_wire_load_selection* to 'false': Also, if the ASIC vendor library has the attribute *default_wire_load* set to a particular wire_load model, then the following command must be used to remove the *default_wire_load* attribute:

```
dc_shell > remove_attribute library_name default_wire_load
```

Case 2: Our design has a number of internally generated signals which drive the enable pins of latches. For example, we have a state machine generated signal which drives the enable pin of a latch. The output of the latch drives a block of combinational logic, which in tum drives a primary output as shown in below Figure. The time delay in the signal reaching the primary output is dependent on how soon the enable signal can be generated, and the delay through the combinational logic, after the data is latched. We wish to constrain the entire path along the enable line to the primary output.

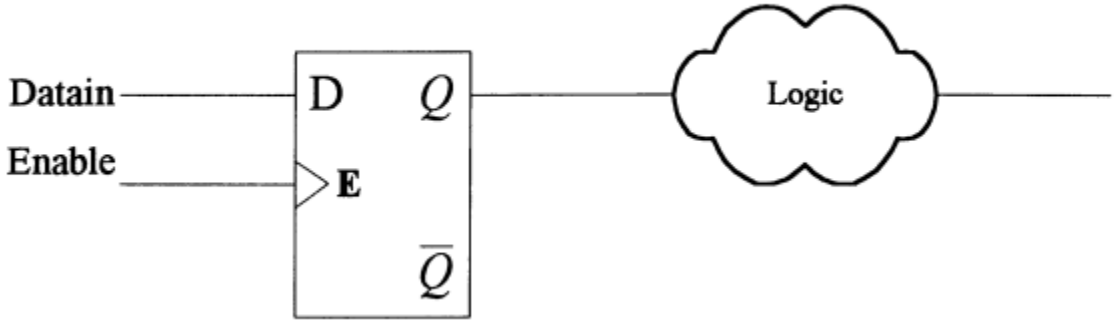


Figure. *Combinational logic driving primary output*

Solution: There is no constraints on the enable (clock) line since there are no setup requirements on the clock pin. Consider a two step constraint approach using the *set_output_delay* and the *max_delay* commands. The path from the enable pin of the latch to the primary output can be constrained using the

set_output_delay command. The path to the enable pin of the latch can be constrained using the *max_delay* command.

Case 3: You have a hierarchical design as shown in below Figure. The *current_design* is set to TOP, the top level of your design. You execute the *report_constraints -all_violators* command and find a number of *max_fanout* violations. You believe that characterizing a particular subblock A and re-compiling that block, should fix a large number of *max_fanout* violations. However, after characterizing the subdesign A and compiling A, you find that none of the fanout violations seen at the top level were fixed.

```
current_design = TOP
characterize U1 /* where u1 is an instance of the sub-design, A */
current_design = A
compile
```

Solution: The fanout violations seen at the top level were not fixed on compiling the characterized sub-block because no *fanout_load* values were applied to the output ports of the lower level. In other words, characterize does not capture the *fanout_load* drive capability required by the output ports E, F and G in above Figure. The values that were applied by characterize, were load values which are not taken into account when fixing *max_fanout* violations.Characterize command will capture this information if one were to use the *characterize-constraints* command instead of just *characterize.* This will ensure that the *fanout_load* values are passed down in addition to the load values on the nets.

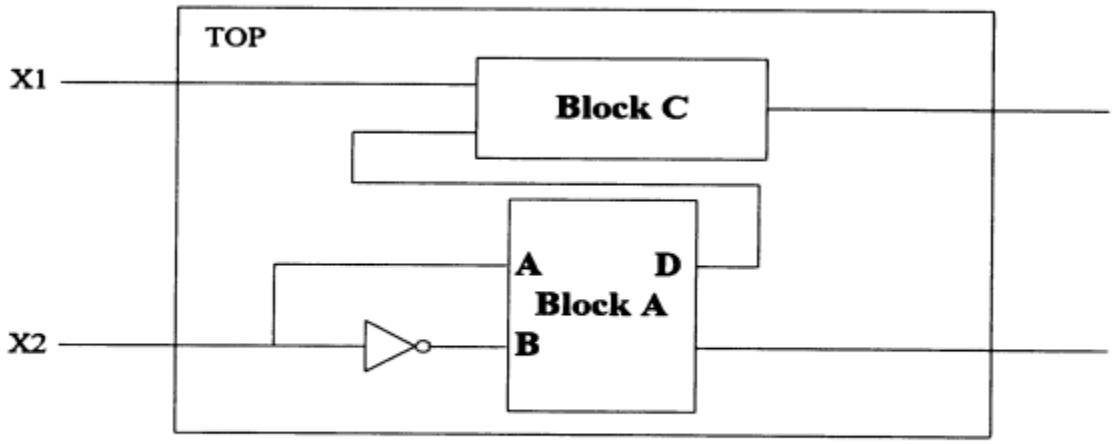


Figure *Design with opposite inputs driving a sub-block*

The *characterize* command also has another useful option, namely, *connections.* This is useful in a scenario where two inputs are identical except that one of them is an inversion of the other as shown in above Figure. Input X2 is inverted and drives Block A at pins A and B. This information is captured when Block A is characterized with the *-connections* option. One can explicitly specify that two ports are opposite of each other using the *set_opposite* command.

Case 4: We wish to find all data pins of latches in your design. Is there a single command which will accomplish this ?

Solution: This can be accomplished by the following command

```
dc_shell > all_registers (-level_sensitive_devices -data_pins)
```

Case 5: We have several instances in your design which have *dont_touch* attributes placed on them. We now wish to ungroup them, but are unable to remove the *don't_touch* attribute on an instance using the *remove_attribute* command.

Solution: It is likely that the instance has inherited the *dont_touch* attribute from its reference. If this is indeed the case, we should first remove the don't touch attribute from the reference. Use the *remove_attribute* command with the *find* command as follows

```
dc_shell > remove_attribute find (design,xxx) dont_touch
dc_shell > remove_attribute find (reference, yyy) dont_touch
```

Case 6: Our technology library has a *default_max_fanout* specified. But the DC on synthesis does not seem to buffer your clock line accordingly.

Solution: The *default_max_fanout* attribute in a library does not direct DC to buffer input ports. Since DC does not have any information on the cell driving the input ports it does not buffer the line. We should *set_max_fanout* on either the design or the input ports that we wish to buffer, and then optimize using the following command.

```
dc_shell > compile -only_design_rules
```

Case 7: After inserting pads using the *insert_pads* command you find clock pads inserted for some of the inputs.

Solution: Clock pads should normally be inserted only for the ports with a clock object created on it. However, they might be inserted on other ports if those ports are part of clock gating logic. If the pads are being inserted on a compiled netlist that contains clock enable buffers, then those ports connected to the clock enable buffers may have clock pads inserted on them also. For other regular inputs, clock pads should not be used. This problem can be avoided by specifying the *set_pad_type -no_clock* attribute on all inputs, except the clock input, prior to pad insertion.