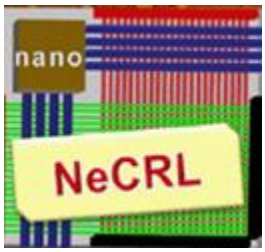


ASIC Design Flow Tutorial
Using Synopsys Tools

By
Hima Bindu Kommuru
Hamid Mahmoodi

Nano-Electronics & Computing Research Lab
School of Engineering
San Francisco State University
San Francisco, CA
Spring 2009

TABLE OF CONTENTS

What is an ASIC?

1.0 Introduction

Integrated Circuits are made from silicon wafer, with each wafer holding hundreds of die. An ASIC is an **Application Specific Integrated Circuit**. An Integrated Circuit designed is called an ASIC if we design the ASIC for the specific application. Examples of ASIC include, chip designed for a satellite, chip designed for a car, chip designed as an interface between memory and CPU etc. Examples of IC's which are not called ASIC include Memories, Microprocessors etc. The following paragraphs will describe the types of ASIC's.

1. **Full-Custom ASIC**: For this type of ASIC, the designer designs all or some of the logic cells, layout for that one chip. The designer does not used predefined gates in the design. Every part of the design is done from scratch.
2. **Standard Cell ASIC**: The designer uses predesigned logic cells such as AND gate, NOR gate, etc. These gates are called Standard Cells. The advantage of Standard Cell ASIC's is that the designers save time, money and reduce the risk by using a predesigned and pre-tested Standard Cell Library. Also each Standard Cell can be optimized individually. The Standard Cell Libraries is designed using the Full Custom Methodology, but you can use these already designed libraries in the design. This design style gives a designer the same flexibility as the Full Custom design, but reduces the risk.
3. **Gate Array ASIC**: In this type of ASIC, the transistors are predefined in the silicon wafer. The predefined pattern of transistors on the gate array is called a base array and the smallest element in the base array is called a base cell. The base cell layout is same for each logic cell, only the interconnect between the cells and inside the cells is customized. The following are the types of gate arrays:
    a. Channeled Gate Array
    b. Channelless Gate Array
    C. Structured Gate Array

When designing a chip, the following objectives are taken into consideration:

1. Speed
2. Area
3. Power
4. Time to Market

To design an ASIC, one needs to have a good understanding of the CMOS Technology. The next few sections give a basic overview of CMOS Technology.

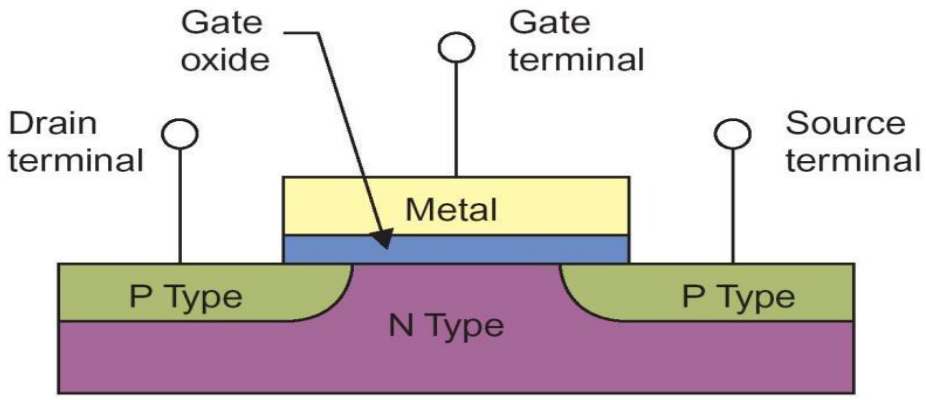
1.1 CMOS Technology

In the present decade the chips being designed are made from CMOS technology. CMOS is Complementary Metal Oxide Semiconductor. It consists of both NMOS and PMOS transistors. To understand CMOS better, we first need to know about the MOS (FET) transistor.

1.2 MOS Transistor

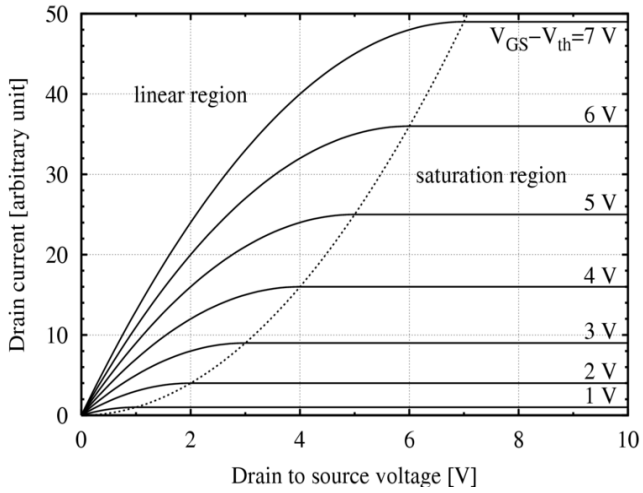
MOS stands for Metal Oxide Semiconductor field effect transistor. MOS is the basic element in the design of a large scale integrated circuit is the transistor. It is a voltage controlled device. These transistors are formed as a ``sandwich" consisting of a semiconductor layer, usually a slice, or wafer, from a single crystal of silicon; a layer of silicon dioxide (the oxide) and a layer of metal. These layers are patterned in a manner which permits transistors to be formed in the semiconductor material (the ``substrate"); The MOS transistor consists of three regions, Source, Drain and Gate. The source and drain regions are quite similar, and are labeled depending on to what they are connected. The source is the terminal, or node, which acts as the source of charge carriers; charge carriers leave the source and travel to the drain. In the case of an N channel MOSFET (NMOS), the source is the more negative of the terminals; in the case of a P channel device (PMOS), it is the more positive of the terminals. The area under the gate oxide is called the ``channel". Below is figure of a MOS Transistor.

Figure 1.2a MOS Transistor



The transistor normally needs some kind of voltage initially for the channel to form. When there is no channel formed, the transistor is said to be in the 'cut off region'. The voltage at which the transistor starts conducting (a channel begins to form between the source and the drain) is called threshold Voltage. The transistor at this point is said to be in the 'linear region'. The transistor is said to go into the 'saturation region' when there are no more charge carriers that go from the source to the drain.

Figure 1.2b Graph of Drain Current vs Drain to Source Voltage



CMOS technology is made up of both NMOS and CMOS transistors. Complementary Metal-Oxide Semiconductors (CMOS) logic devices are the most common devices used today in the high density, large number transistor count circuits found in everything from complex microprocessor integrated circuits to signal processing and communication circuits. The CMOS structure is popular because of its inherent lower power requirements, high operating clock speed, and ease of implementation at the transistor level. The complementary p-channel and n-channel transistor networks are used to connect the output of the logic device to the either the $\boldsymbol{V_{DD}}$ or $\boldsymbol{V_{SS}}$ power supply rails for a given input logic state. The MOSFET transistors can be treated as simple switches. The switch must be on (conducting) to allow current to flow between the source and drain terminals.

Example: Creating a CMOS inverter requires only one PMOS and one NMOS transistor. The NMOS transistor provides the switch connection (ON) to ground when the input is logic high. The output load capacitor gets discharged and the output is driven to a logic'0'. The PMOS transistor (ON) provides the connection to the V_{DD} power supply rail when the input to the inverter circuit is logic low. The output load capacitor gets charged to V_{DD}. The output is driven to logic '1'.

The output load capacitance of a logic gate is comprised of

a. Intrinsic Capacitance: Gate drain capacitance (of both NMOS and PMOS transistors)
b. Extrinsic Capacitance: Capacitance of connecting wires and also input capacitance of the Fan out Gates.

→In CMOS, there is only one driver, but the gate can drive as many gates as possible. In CMOS technology, the output always drives another CMOS gate input.

The charge carriers for PMOS transistors is 'holes' and charge carriers for NMOS are electrons. The mobility of electrons is two times more than that of 'holes'. Due to this the output rise and fall time is different. To make it same, the W/L ratio of the PMOS transistor is made about twice that of the NMOS transistor. This way, the PMOS and

NMOS transistors will have the same 'drive strength'. In a standard cell library, the length 'L' of a transistor is always constant. The width 'W' values are changed to have to different drive strengths for each gate. The resistance is proportional to (L/W). Therefore if the increasing the width, decreases the resistance.

1.3 Power Dissipation in CMOS IC's

The big percentage of power dissipation in CMOS IC's is due to the charging and discharging of capacitors. Majority of the low power CMOS IC designs issue is to reduce power dissipation. The main sources of power dissipation are:

1. **Dynamic Switching Power**: due to charging and discharging of circuit capacitances
   → A low to high output transition draws energy from the power supply
   → A high to low transition dissipates energy stored in CMOS transistor.
   → Given the frequency 'f', of the low-high transitions, the total power drawn would be: load capacitance*Vdd*Vdd*f
2. **Short Circuit Current**: It occurs when the rise/fall time at the input of the gate is larger than the output rise/fall time.
3. **Leakage Current Power**: It is caused by two reasons;
   a. Reverse-Bias Diode Leakage on Transistor Drains: This happens in CMOS design, when one transistor is off, and the active transistor charges up/down the drain using the bulk potential of the other transistor.
   Example: Consider an inverter with a high input voltage, output is low which means NMOS is on and PMOS is off. The bulk of PMOS is connected to VDD. Therefore there is a drain-to –bulk voltage –VDD, causing the diode leakage current.
   b. Sub-Threshold Leakage through the channel to an 'OFF' transistor/device.

1.4 CMOS Transmission Gate

A PMOS transistor is connected in parallel to a NMOS transistor to form a Transmission gate. The transmission gate just transmits the value at the input to the output. It consists of both NMOS and PMOS because, PMOS transistor transmits a strong '1' and NMOS transistor transmits a strong '0'. The advantages of using a Transmission Gate are:
1. It shows better characteristics than a switch.
2. The resistance of the circuit is reduced, since the transistors are connected in parallel.

Sequential Element

In CMOS, an element which stores a logic value (by having a feedback loop) is called a sequential element. A simplest example of a sequential element would be two inverters connected back to back. There are two types of basic sequential elements, they are:

1. **Latch:** The two inverters connected back to back, when connected to a transmission gate, with a control input, forms a latch. When the control input is high (logic '1'), the transmission gate is switched on and whatever value which was at the input 'D' passes to the output. When the control input is low, the transmission gate is off and the inverters that are connected back to back hold the

value. Latch is called a transparent latch because when the ‘D’ input changes, the output also changes accordingly.

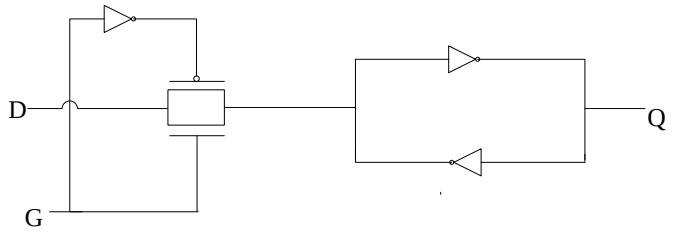


Figure 1.4a Latch

2. **Flip-Flop**: A flip flop is constructed from two latches in series. The first latch is called a Master latch and the second latch is called the slave latch. The control input to the transmission gate in this case is called a clock. The inverted version of the clock is fed to the input of the slave latch transmission gate.
    a. When the clock input is high, the transmission gate of the master latch is switched on and the input ‘D’ is latched by the 2 inverters connected back to back (basically master latch is transparent). Also, due to the inverted clock input to the transmission gate of the slave latch, the transmission gate of the slave latch is not ‘on’ and it holds the previous value.
    b. When the clock goes low, the slave part of the flip flop is switched on and will update the value at the output with what the master latch stored when the clock input was high. The slave latch will hold this new value at the output irrespective of the changes at the input of Master latch when the clock is low. When the clock goes high again, the value at the output of the slave latch is stored and step’a’ is repeated again.

The data latched by the Master latch in the flip flop happens at the rising clock edge, this type of flip flop is called positive-edge triggered flip flop. If the latching happens at negative edge of the clock, the flip flop is called negative edge triggered flip flop.

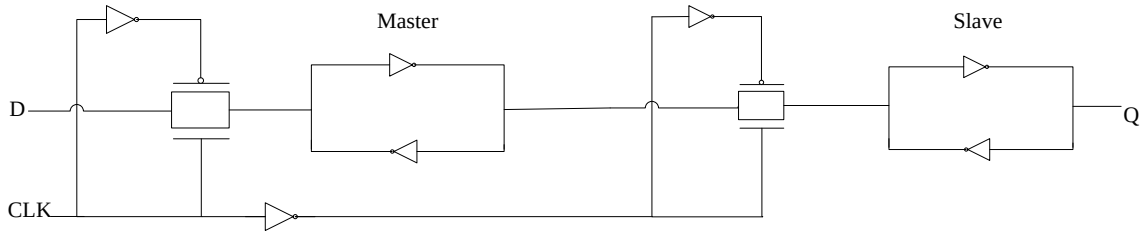


Figure 1.4b Flip-Flop

Overview of ASIC Flow

2.0 Introduction

To design a chip, one needs to have an **Idea** about what exactly one wants to design. At every step in the ASIC flow the idea conceived keeps changing forms. The first step to make the idea into a chip is to come up with the Specifications.
Specifications are nothing but

- Goals and constraints of the design.
- Functionality (what will the chip do)
- Performance figures like speed and power
- Technology constraints like size and space (physical dimensions)
- Fabrication technology and design techniques

The next step is in the flow is to come up with the **Structural and Functional Description.** It means that at this point one has to decide what kind of architecture (structure) you would want to use for the design, e.g. RISC/CISC, ALU, pipelining etc … To make it easier to design a complex system; it is normally broken down into several sub systems. The functionality of these subsystems should match the specifications. At this point, the relationship between different sub systems and with the top level system is also defined.

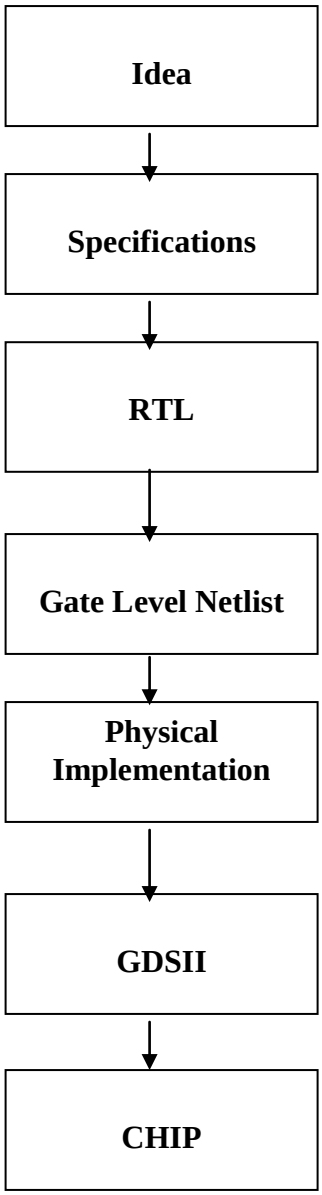
The sub systems, top level systems once defined, need to be implemented. It is implemented using logic representation (Boolean Expressions), finite state machines, Combinatorial, Sequential Logic, Schematics etc.... This step is called Logic Design / **Register Transfer Level (RTL)**. Basically the RTL describes the several sub systems. It should match the functional description. RTL is expressed usually in Verilog or VHDL. Verilog and VHDL are Hardware Description Languages. A hardware description language (HDL) is a language used to describe a digital system, for example, a network switch, a microprocessor or a memory or a simple flip-flop. This just means that, by using a HDL one can describe any hardware (digital) at any level. Functional/Logical Verification is performed at this stage to ensure the RTL designed matches the idea.

Once Functional Verification is completed, the RTL is converted into an optimized **Gate Level Netlist**. This step is called Logic/RTL synthesis. This is done by Synthesis Tools such as Design Compiler (Synopsys), Blast Create (Magma), RTL Compiler (Cadence) etc... A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as output. Standard cell library is the basic building block for today's IC design. Constraints such as timing, area, testability, and power are considered. Synthesis tools try to meet constraints, by calculating the cost of various implementations. It then tries to generate the best gate level implementation for a given set of constraints, target process. The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design. At this stage, it is also verified whether the Gate Level Conversion has been correctly performed by doing simulation.

The next step in the ASIC flow is the **Physical Implementation** of the Gate Level Netlist. The Gate level Netlist is converted into geometric representation. The geometric

representation is nothing but the layout of the design. The layout is designed according to the design rules specified in the library. The design rules are nothing but guidelines based on the limitations of the fabrication process. The Physical Implementation step consists of three sub steps; Floor planning->Placement->Routing. The file produced at the output of the Physical Implementation is the **GDSII** file. It is the file used by the foundry to fabricate the ASIC. This step is performed by tools such as Blast Fusion (Magma), IC Compiler (Synopsys), and Encounter (Cadence) Etc…Physical Verification is performed to verify whether the layout is designed according the rules.

Figure 2.a : Simple ASIC Design Flow



For any design to work at a specific speed, timing analysis has to be performed. We need to check whether the design is meeting the speed requirement mentioned in the specification. This is done by Static Timing Analysis Tool, for example Primetime (Synopsys). It validates the timing performance of a design by checking the design for all possible timing violations for example; set up, hold timing.

After Layout, Verification, Timing Analysis, the layout is ready for **Fabrication**. The layout data is converted into photo lithographic masks. After fabrication, the wafer is diced into individual chips. Each **Chip** is packaged and tested.

Synopsys Verilog Compiler Simulator (VCS) Tutorial

3.0 Introduction

Synopsys Verilog Compiler Simulator is a tool from Synopsys specifically designed to simulate and debug designs. This tutorial basically describes how to use VCS, simulate a verilog description of a design and learn to debug the design. VCS also uses VirSim, which is a graphical user interface to VCS used for debugging and viewing the waveforms.

There are three main steps in debugging the design, which are as follows

1. Compiling the Verilog/VHDL source code.
2. Running the Simulation.
3. Viewing and debugging the generated waveforms.

You can interactively do the above steps using the VCS tool. VCS first compiles the verilog source code into object files, which are nothing but C source files. VCS can compile the source code into the object files without generating assembly language files. VCS then invokes a C compiler to create an executable file. We use this executable file to simulate the design. You can use the command line to execute the binary file which creates the waveform file, or you can use VirSim.

Below is a brief overview of the VCS tool, shows you how to compile and simulate a counter. For basic concepts on verification and test bench, please refer to ***APPENDIX 3A*** at the end of this chapter.

SETUP

Before going to the tutorial Example, let's first setup up the directory.

You need to do the below 3 steps before you actually run the tool:

1. As soon as you log into your engr account, at the command prompt, please type "csh "as shown below. This changes the type of shell from bash to c-shell. All the commands work ONLY in c-shell.

[hkommuru@hafez]$csh

2. Please copy the whole directory from the below location (cp –rf source destination)

[hkommuru@hafez]$cd
[hkommuru@hafez]$ cp -rf /packages/synopsys/setup/asic_flow_setup ./

This creates directory structure as shown below. It will create a directory called **"asic_flow_setup ",** under which it creates the following directories namely

asic_flow_setup
src/ : for verilog code/source code
vcs/ : for vcs simulation ,
synth_graycounter/ : for synthesis
synth_fifo/ : for synthesis
pnr/ : for Physical design
extraction/: for extraction
pt/: for primetime
verification/: final signoff check

The **"asic_flow_setup"** directory will contain all generated content including, VCS simulation, synthesized gate-level Verilog, and final layout. In this course we will always try to keep generated content from the tools separate from our source RTL. This keeps our project directories well organized, and helps prevent us from unintentionally modifying the source RTL. There are subdirectories in the project directory for each major step in the ASIC Flow tutorial. These subdirectories contain scripts and configuration files for running the tools required for that step in the tool flow. For this tutorial we will work exclusively in the ***vcs*** directory.

3. Please source "***synopsys_setup.tcl***" which sets all the environment variables necessary to run the VCS tool.
Please source them at unix prompt as shown below

[hkommuru@hafez]$ source /packages/synopsys/setup/synopsys_setup.tcl

Please Note : You have to do steps 1 and 3 above everytime you log in.

3.1 Tutorial Example

In this tutorial, we would be using a simple counter example . Find the verilog code and testbench at the end of the tutorial.

Source code file name : counter.v
Test bench file name : counter_tb.v

Setup

3.1.1 Compiling and Simulating

NOTE: AT PRESENT THERE SEEMS TO BE A BUG IN THE TOOL, SO COMPILE AND SIMULATION IN TWO DIFFERENT STEPS IS NOT WORKING. THIS WILL BE FIXED SHORTLY. PLEASE DO STEP 3 TO SEE THE OUTPUT OF YOUR CODE. STEP 3 COMMAND PERFORMS COMPILATION AND SIMULATION IN ONE STEP.

1. In the "vcs" directory, compile the verilog source code by typing the following at the machine prompt.

[hkommuru@hafez vcs]$ cd asic_flow_setup/vcs
[hkommuru@hafez vcs]$ cp ../src/counter/ .*
[hkommuru@hafez vcs]$ vcs –f main_counter.f +v2k

Please note that the –f option means the file specified (main_counter.f) contains a list of command line options for vcs. In this case, the command line options are just a list of the verilog file names. Also note that the testbench is listed first. The below command also will have same effect .

[hkommuru@hafez vcs]$ vcs –f counter_tb.v counter.v +v2k

The +v2k option is used if you are using Verilog IEE 1364-2000 syntax; otherwise there is no need for the option. Please look at Figure 3.a for output of compile command.

Figure 3.a: vcs compile

```
                Chronologic VCS (TM)
        Version B-2008.12 -- Wed Jan 28 20:08:26 2009
          Copyright (c) 1991-2008 by Synopsys Inc.
                ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.


Warning-[ACC_CLI_ON] ACC/CLI capabilities enabled
  ACC/CLI capabilities have been enabled for the entire design. For faster
  performance enable module specific capability in pli.tab file

Parsing design file 'counter_tb.v'
Parsing design file 'counter.v'
Top Level Modules:
       timeunit
       counter_testbench
TimeScale is 1 ns / 10 ps
Starting vcs inline pass...
2 modules and 0 UDP read.
recompiling module timeunit
recompiling module counter_testbench
Both modules done.
gcc  -pipe -m32 -O -I/packages/synopsys/vcs_mx/B-2008.12/include   -c -o rmapats.o rmapats.c
if [ -x ../simv ]; then chmod -x ../simv; fi
g++  -o ../simv -melf_i386 -m32  5NrI_d.o 5NrIB_d.o IV5q_1_d.o blOS_1_d.o rmapats_mop.o rmapats.o
SIM_l.o          /packages/synopsys/vcs_mx/B-2008.12/linux/lib/libvirsim.a    /packages/synopsys/vcs_mx/B-
2008.12/linux/lib/librterrorinf.so              /packages/synopsys/vcs_mx/B-2008.12/linux/lib/libsnpsmalloc.so
```

/packages/synopsys/vcs_mx/B-2008.12/linux/lib/libvcsnew.so /packages/synopsys/vcs_mx/B-2008.12/linux/lib/ctype-stubs_32.a -ldl -lz -lm -lc -ldl
../simv up to date

VirSim B-2008.12-B Virtual Simulator Environment
Copyright (C) 1993-2005 by Synopsys, Inc.
Licensed Software. All Rights Reserved.

By default the output of compilation would be a executable binary file is named **simv**. You can specify a different name with the -o compile-time option.

For example :
vcs –f main_counter.f +v2k –o counter.simv

VCS compiles the source code on a module by module basis. You can incrementally compile your design with VCS, since VCS compiles only the modules which have changed since the last compilation.

2. Now, execute the simv command line with no arguments. You should see the output from both vcs and simulation and should produce a waveform file called counter.dump in your working directory.

[hkommuru@hafez vcs]$./counter.simv

Please see Figure 3.b for output of simv command

Figure 3.b Simulation Result

Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.
Compiler version B-2008.12; Runtime version B-2008.12; Jan 28 19:59 2009

time= 0 ns, clk=0, reset=0, out=xxxx
time= 10 ns, clk=1, reset=0, out=xxxx
time= 11 ns, clk=1, reset=1, out=xxxx
time= 20 ns, clk=0, reset=1, out=xxxx
time= 30 ns, clk=1, reset=1, out=xxxx
time= 31 ns, clk=1, reset=0, out=0000
time= 40 ns, clk=0, reset=0, out=0000
time= 50 ns, clk=1, reset=0, out=0000
time= 51 ns, clk=1, reset=0, out=0001
time= 60 ns, clk=0, reset=0, out=0001
time= 70 ns, clk=1, reset=0, out=0001
time= 71 ns, clk=1, reset=0, out=0010
time= 80 ns, clk=0, reset=0, out=0010
time= 90 ns, clk=1, reset=0, out=0010
time= 91 ns, clk=1, reset=0, out=0011
time= 100 ns, clk=0, reset=0, out=0011
time= 110 ns, clk=1, reset=0, out=0011
time= 111 ns, clk=1, reset=0, out=0100

time= 120 ns, clk=0, reset=0, out=0100
time= 130 ns, clk=1, reset=0, out=0100
time= 131 ns, clk=1, reset=0, out=0101
time= 140 ns, clk=0, reset=0, out=0101
time= 150 ns, clk=1, reset=0, out=0101
time= 151 ns, clk=1, reset=0, out=0110
time= 160 ns, clk=0, reset=0, out=0110
time= 170 ns, clk=1, reset=0, out=0110
All tests completed sucessfully

$finish called from file "counter_tb.v", line 75.
$finish at simulation time 171.0 ns
V C S S i m u l a t i o n R e p o r t
Time: 171000 ps
CPU Time: 0.020 seconds; Data structure size: 0.0Mb
Wed Jan 28 19:59:54 2009

If you look at the last page of the tutorial, you can see the testbench code, to understand the above result better.

3. You can do STEP 1 and STEP 2 in one single step below. It will compile and simulate in one single step. Please take a look at the command below:

[hkommuru@hafez vcs]$ vcs -V -R -f main_counter.f -o simv

In the above command,
-V : stands for Verbose
-R : command which tells the tool to do simulation immediately/automatically after compilation
-o : output file name , can be anything simv, counter.simv etc...
-f : specifying file

To compile and simulate your design, please write your verilog code, and copy it to the vcs directory. After copying your verilog code to the vcs directory, follow the tutorial steps to simulate and compile.

3.2 DVE TUTORIAL

DVE provides you a graphical user interface to debug your design. Using DVE you can debug the design in interactive mode or in postprocessing mode. In the interactive mode, apart from running the simulation, DVE allows you to do the following:
• View waveforms
• Trace Drivers and loads
• Schematic, and Path Schematic view
• Compare waveforms
• Execute UCLI/Tcl commands
• Set line, time, or event break points
• Line stepping

However, in post-processing mode, a VPD/VCD/EVCD file is created during simulation, and you use DVE to:
• View waveforms
• Trace Drivers and loads
• Schematic, and Path Schematic view
• Compare waveforms
Use the below command line to invoke the simulation in interactive mode using DVE:

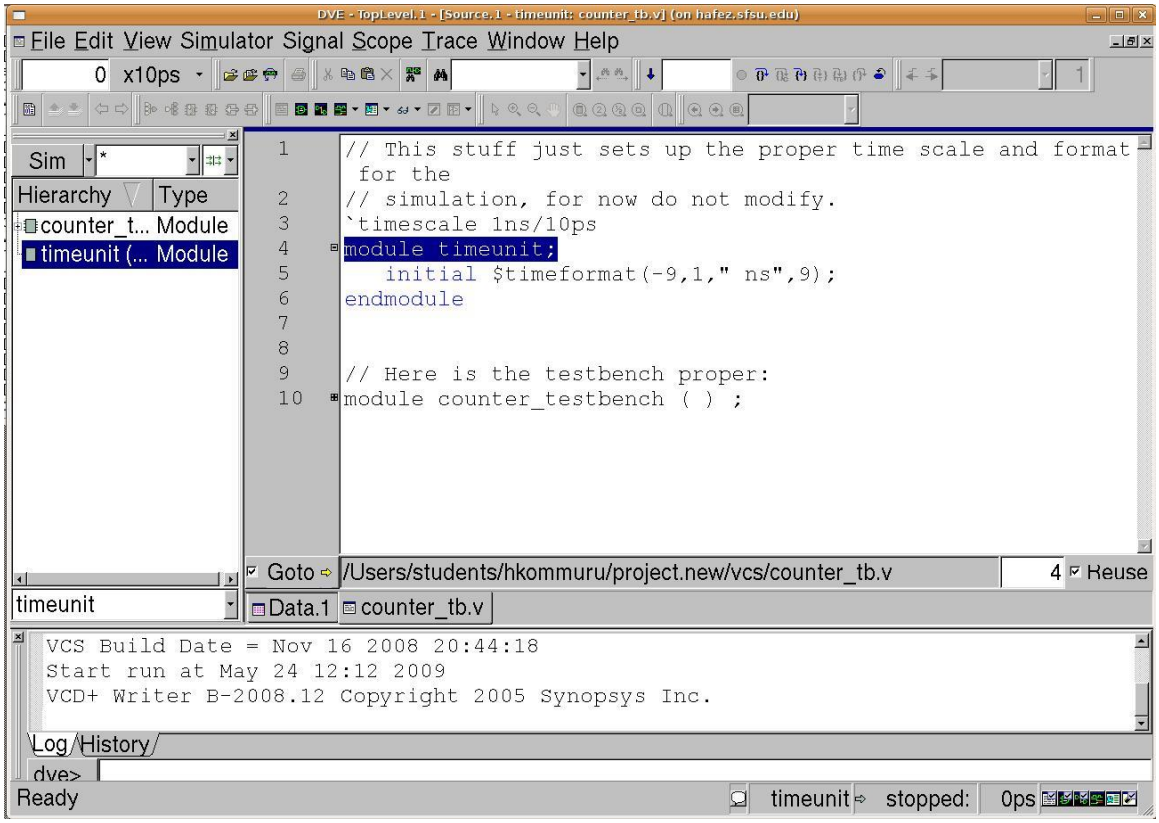
*[hkommuru@hafez vcs]$*simv –gui

A TopLevel window is a frame that displays panes and views.
• A pane can be displayed one time on each TopLevel Window. serves a specific debug purpose. Examples of panes are Hierarchy, Data, and the Console panes.
• A view can have multiple instances per TopLevel window. Examples of views are Source, Wave, List, Memory, and Schematic. Panes can be docked on any side to a TopLevel window or left floating in the area in the frame not occupied by docked panes (called the workspace).
You can use the above command or you can do everything, which is compile and simulation, open the gui in one step.

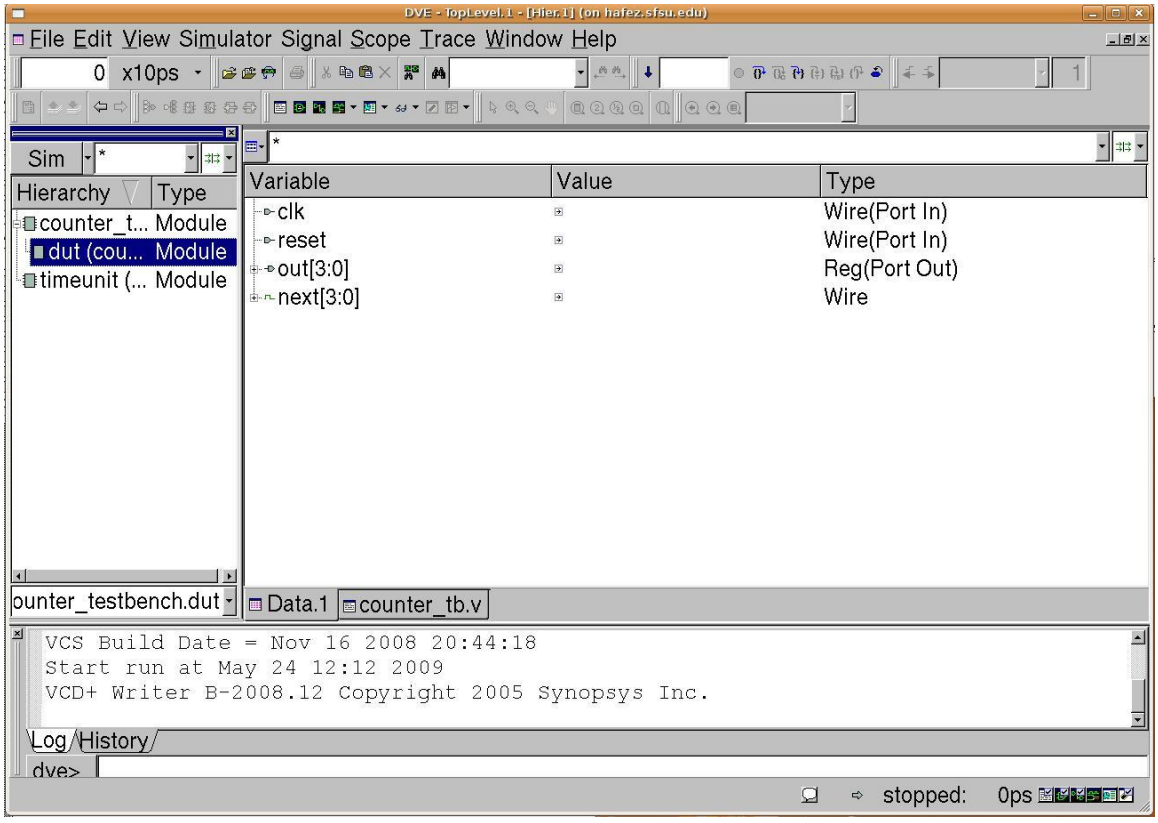
1. Invoke dve to view the waveform. At the unix prompt, type :

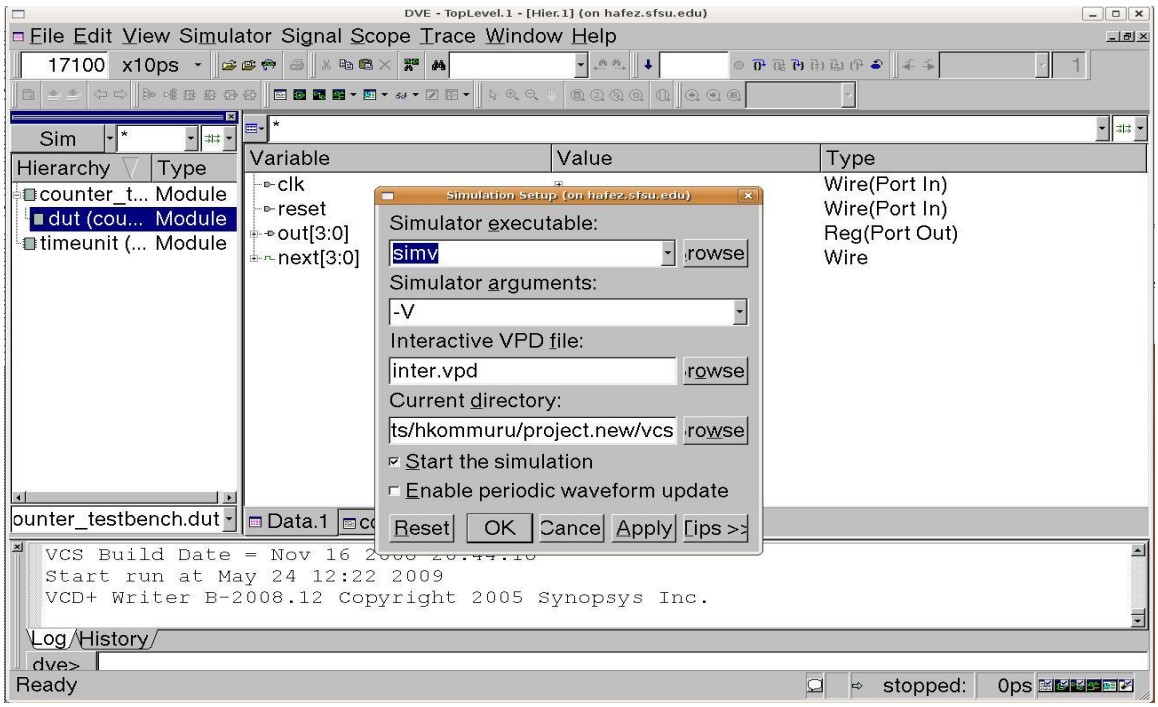
[hkommuru@hafez vcs]$ vcs -V -R -f main_counter.f -o simv -gui –debug_pp

Where debug_pp option is used to run the dve in simulation mode. Debug_pp creates a vpd file which is necessary to do simulation. The below window will open up.



2. In the above window, open up the counter module to view the whole module like below. Click on dut highlighted in blue and drag it to the data pane as shown below. All the signals in the design will show up in the data pane.

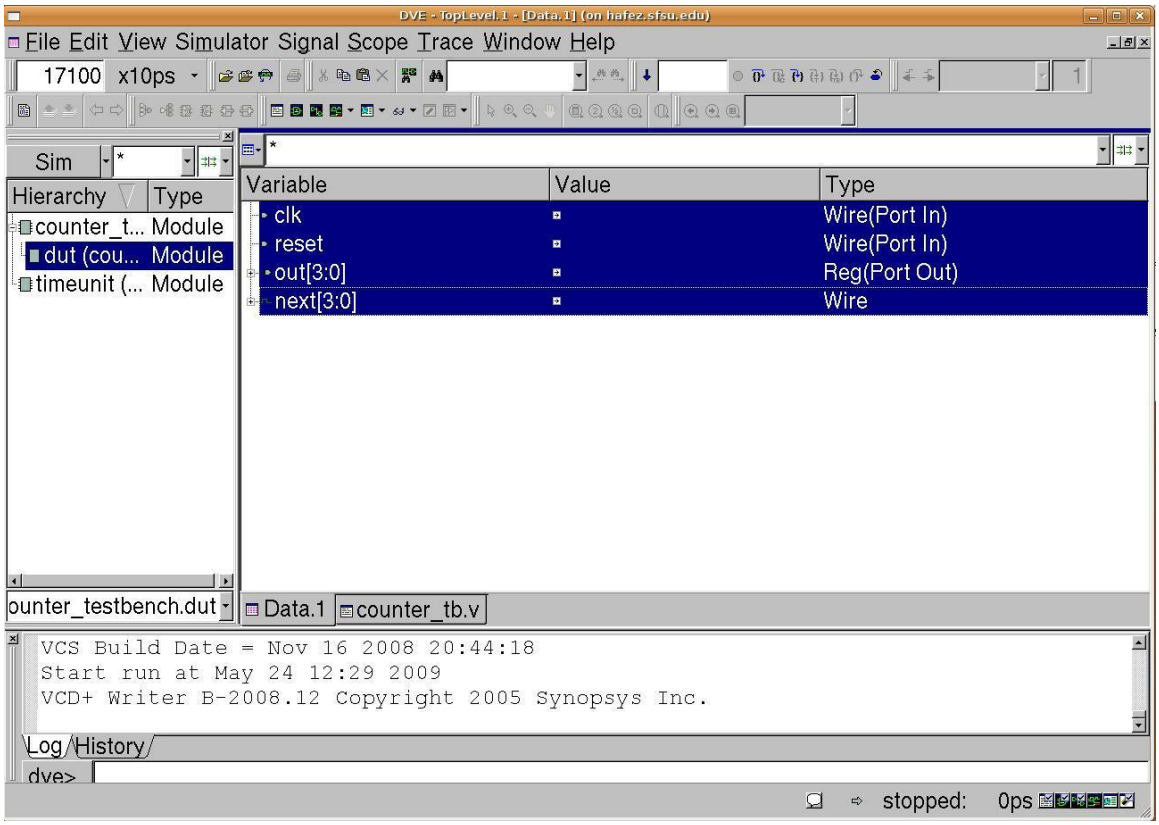




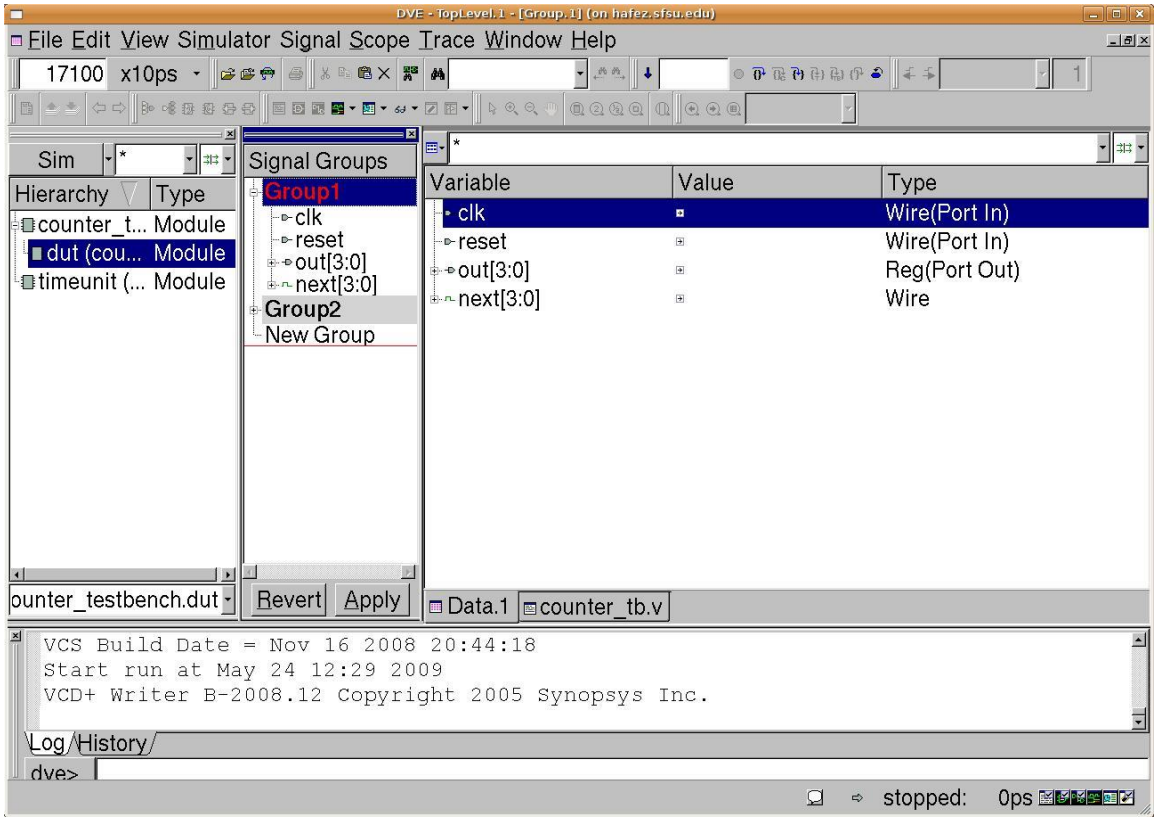
3. In this window, click on “Setup” under the “Simulator” option. A new small window will open up as shown. Inter.vpd is the file, the simulator will use to run the waveform.

The –debug_pp option in step1 creates this file. Click ok and now the step up is complete to run the simulation shown in the previous page.

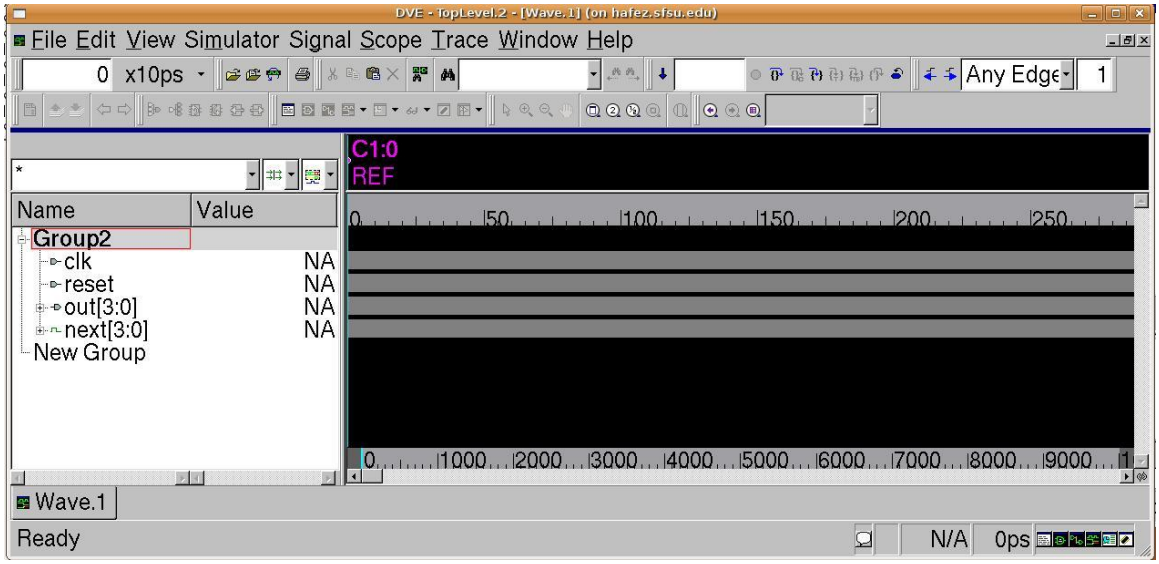
4. Now in the data pane select all the signals with the left mouse button holding the shift button so that you select as many signals you want. Click on the right mouse button to open a new window, and click on "Add to group => New group . A new window will open up showing a new group of selected signals below.



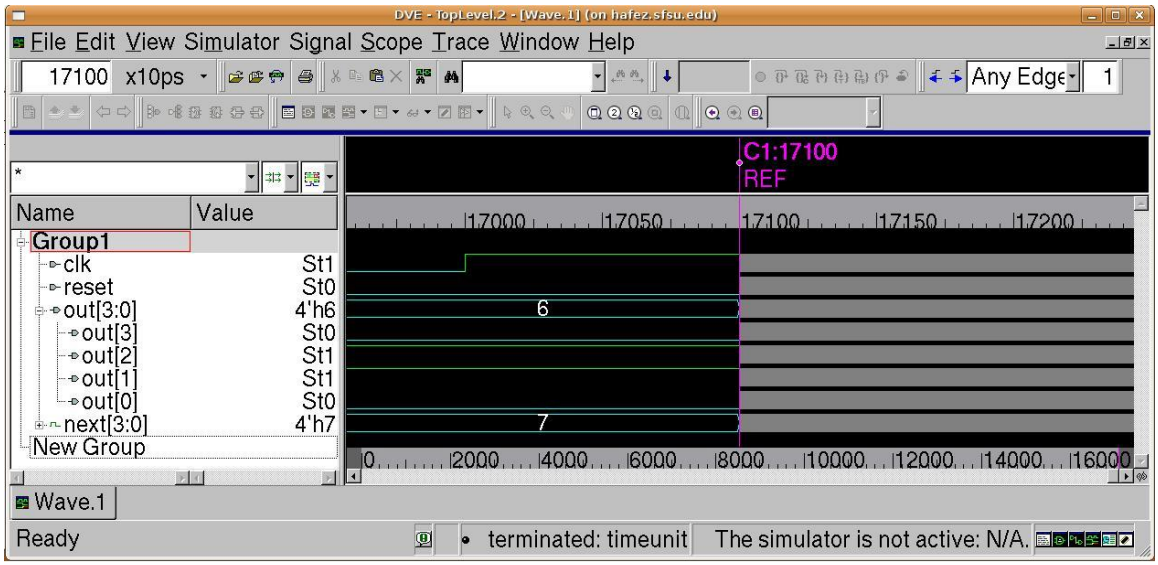
You can create any number of signal groups you want so that you can organize the way and you want to see the output of the signals .



5. 4. Now in the data pane select all the signals with the left mouse button holding the shift button so that you select as many signals you want. Click on the right mouse button to open a new window, and click on "Add to waves ➔ New wave view". A new waveform window will open with simulator at 0ns .

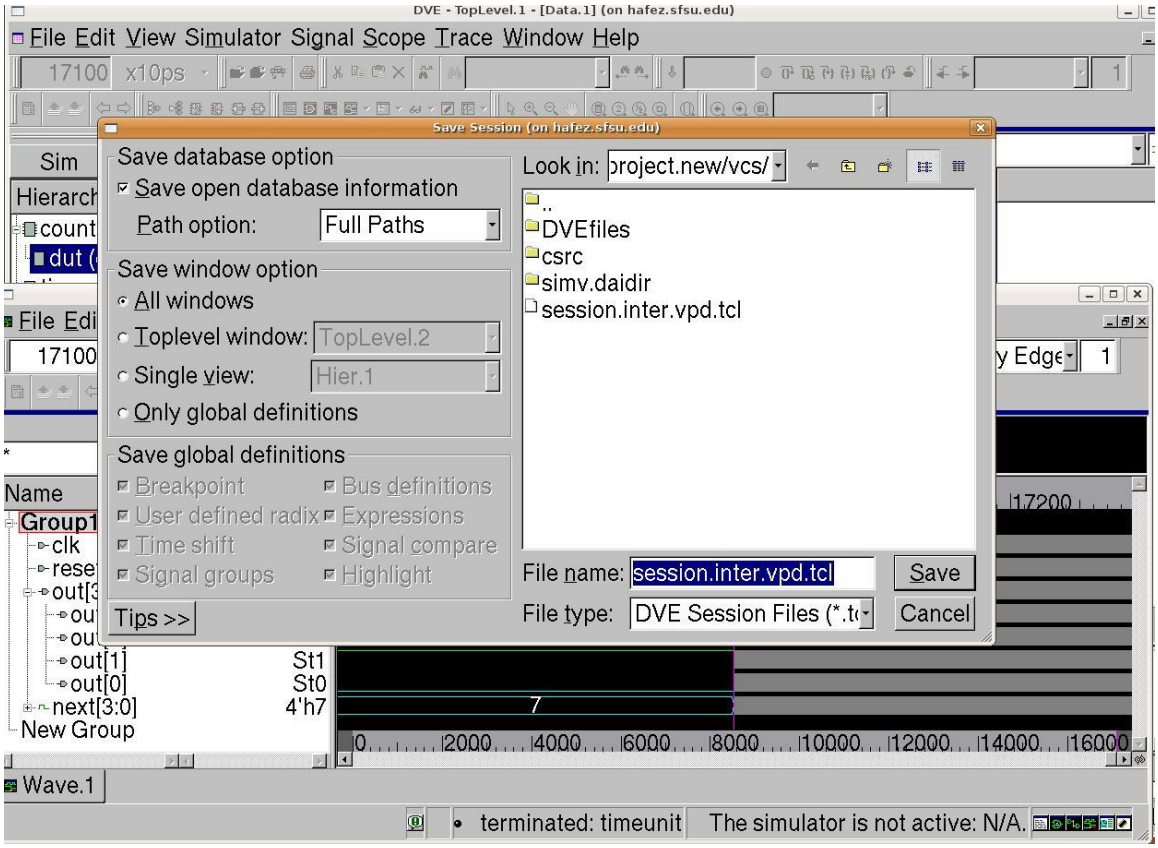


6. In the waveform window, go to “Simulator menu option” and click on “Start”. The tool now does simulation and you can verify the functionality of the design as shown below.

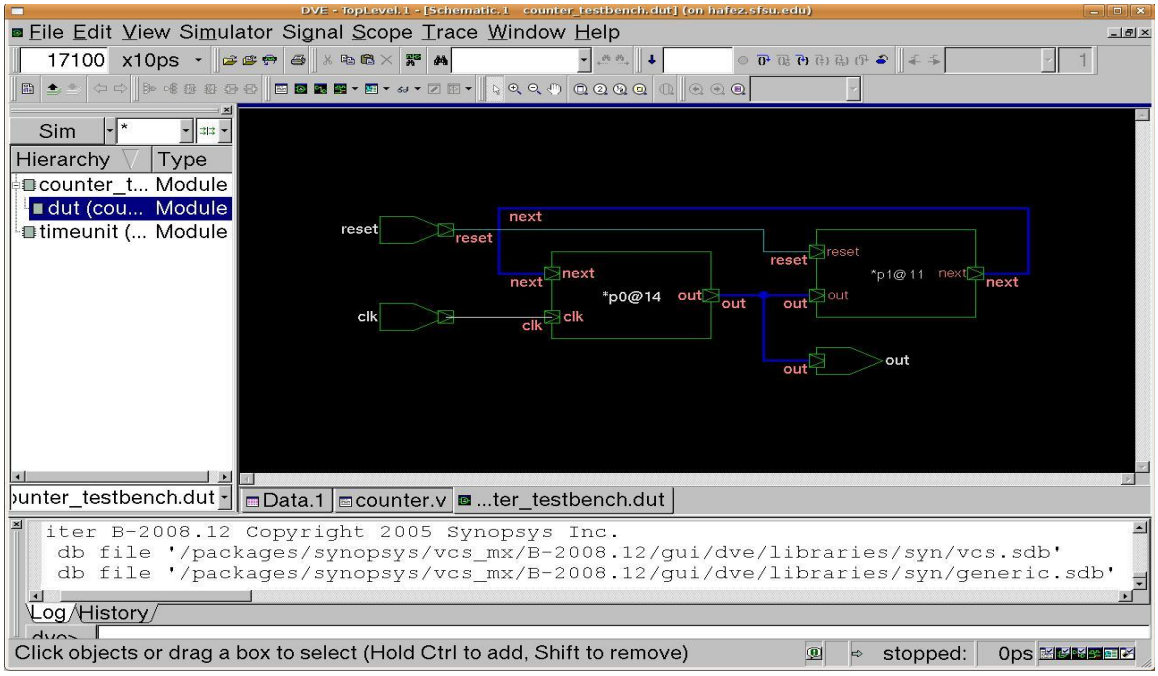


In the waveform window, the menu option View → Set Time Scale can be used to change the display unit and the display precision

7. You can save your current session and reload the same session next time or start a new session again. In the menu option , File → Save Session, the below window opens as shown below.



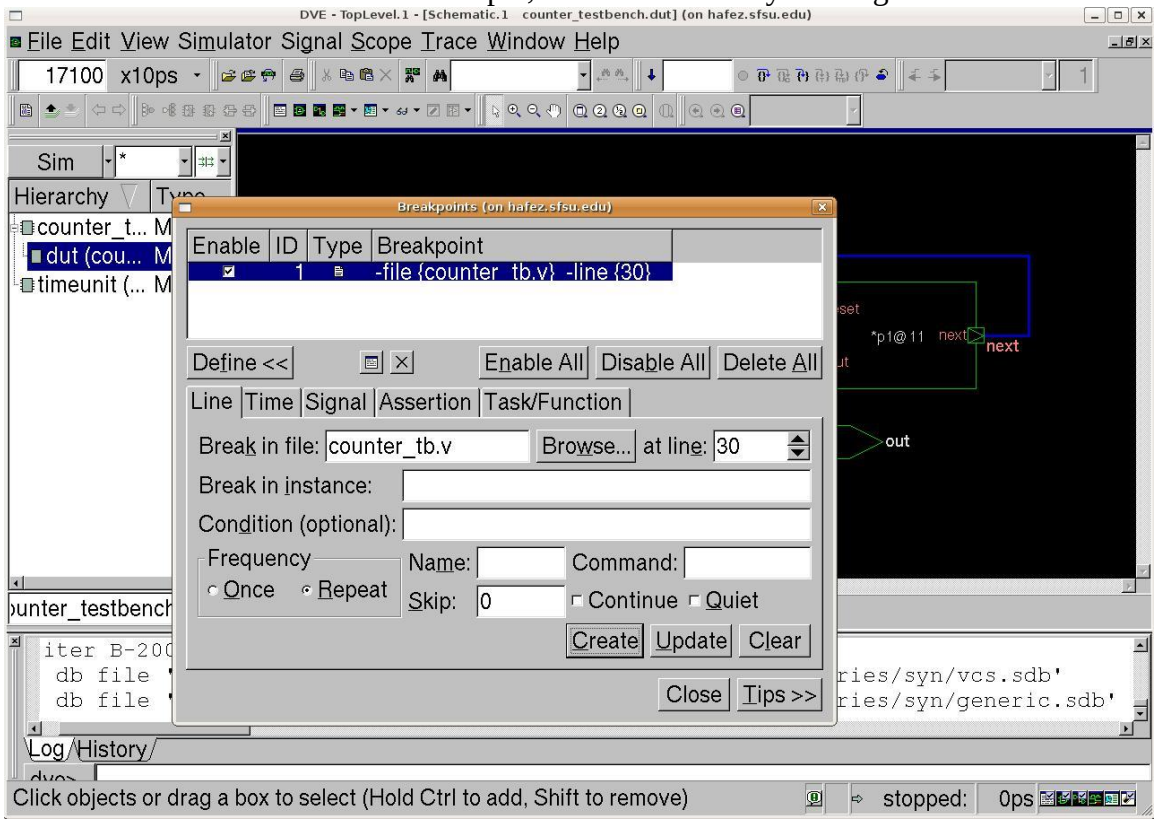
8. **For additional debugging steps,** you can go to menu option
   1. Scope → Show Source code: You can view your source code here and analyze.
   2. Scope → Show Schematic: You can view a schematic view of the design .



9. Adding Breakpoints in Simulation. To be able to add breakpoints, you have to use a additional compile option –debug_all –flag when you compile the code as shown below.

[hkommuru@hafez vcs]$ vcs -V -R -f main_counter.f -o simv -gui –debug_pp -debug_all –flag

Go to the menu option, Simulation → Breakpoints , will open up a new window as shown below. You need to do this before Step 6, i.e. before actually running the simulation.



You can browse which file and also the line number and click on "Create" button to create breakpoints.

Now when you simulate, click on Simulate → Start, it will stop at your defined breakpoint, click on Next to continue.

You can save your session again and exit after are done with debugging or in the middle of debugging your design.

Verilog Code

File : Counter.v

```
module counter ( out, clk, reset ) ;
```

```
   input        clk, reset;
   output [3:0] out;

   reg [3:0]    out;

   wire [3:0]   next;

   // This statement implements reset and increment
   assign       next = reset ? 4'b0 : (out + 4'b1);

   // This implements the flip-flops
   always @ ( posedge clk ) begin
      out <= #1 next;
   end


endmodule // counter
```

File : Counter_tb.v [Test Bench]

```
// This stuff just sets up the proper time scale and format for the
// simulation, for now do not modify.
`timescale 1ns/10ps
module timeunit;
   initial $timeformat(-9,1," ns",9);
endmodule


// Here is the testbench proper:
module counter_testbench ( ) ;

    // Test bench gets wires for all device under test (DUT) outputs:

   wire [3:0]        out;

   // Regs for all DUT inputs:
   reg               clk;
   reg               reset;


   counter dut (// (dut means device under test)
                // Outputs
                .out            (out[3:0]),
                // Inputs
                .reset          (reset),
                .clk            (clk));


  // Setup clk to automatically strobe with a period of 20.
   always #10 clk = ~clk;


   initial
     begin

        // First setup up to monitor all inputs and outputs
        $monitor ("time=%5d ns, clk=%b, reset=%b, out=%b", $time, clk,
reset, out[3:0]);
```

```
        // First initialize all registers
        clk = 1'b0;             // what happens to clk if we don't
                                // set this?;
        reset = 1'b0;

        @(posedge clk);#1;      // this  says wait for rising edge
                                // of clk and one more tic (to prevent
                                // shoot through)

        reset = 1'b1;

        @(posedge clk);#1;

        reset = 1'b0;

        // Lets watch what happens after 7 cycles
        @(posedge clk);#1;
        @(posedge clk);#1;
        @(posedge clk);#1;
        @(posedge clk);#1;
        @(posedge clk);#1;
        @(posedge clk);#1;
        @(posedge clk);#1;


        // At this point we should have a 4'b0110 coming out out
because
        // the counter should have counted for 7 cycles from 0
        if (out != 4'b0110) begin
           $display("ERROR 1: Out is not equal to 4'b0110");
           $finish;
        end


        // We got this far so all tests passed.
        $display("All tests completed sucessfully\n\n");

        $finish;
     end


   // This is to create a dump file for offline viewing.
   initial
     begin
        $dumpfile ("counter.dump");
        $dumpvars (0, counter_testbench);
     end // initial begin


endmodule // counter_testbench
```

APPENDIX 3A: Overview of RTL

3.A.1 Register Transfer Logic

RTL is expressed in Verilog or VHDL. This document will cover the basics of Verilog. Verilog is a Hardware Description Language (HDL). A hardware description language is a language used to describe a digital system example Latches, Flip-Flops, Combinatorial, Sequential Elements etc… Basically you can use Verilog to describe any kind of digital system. One can design a digital system in Verilog using any level of abstraction. The most important levels are:

- **Behavior Level:** This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. There is no regard to the structural realization of the design. Example (Use of 'always' statement in Verilog).
- **Register Transfer Level (RTL):** Designs using the Register-Transfer Level specify the characteristics of a circuit by transfer of data between the registers, and also the functionality; for example Finite State Machines. An explicit clock is used. RTL design contains exact timing possibility; and data transfer is scheduled to occur at certain times.
- **Gate level:** The system is described in terms of gates (AND, OR, NOT, NAND etc…). The signals can have only these four logic states ('0','1','X','Z'). The Gate Level design is normally not done because the output of Logic Synthesis is the gate level netlist.

Verilog allows hardware designers to express their designs at the behavioral level and not worry about the details of implementation to a later stage in the design of the chip. The design normally is written in a top-down approach. The system has a hierarchy which makes it easier to debug and design. The basic skeleton of a verilog module looks like this:

```
module example (<ports >);

input <ports>;
output <ports>;
inout <ports>;

# Data-type instantiation
#reg data-type stores values
reg <names>;

#Wire data-type connects two different pins/ports
wire <names> ;

<Instantiation>
```

end module

The modules can reference other modules to form a hierarchy. If the module contains references to each of the lower level modules, and describes the interconnections between them, a reference to a lower level module is called a *module instance*. Each instance is an independent, concurrently active copy of a module. Each module instance consists of the name of the module being instanced (e.g. NAND or INV), an instance name (unique to that instance within the current module) and a port connection list.

NAND N1 (in1, in2, out)

INV V1 (a, abar);

Instance name in the above example is 'N1 and V1' and it has to be unique. The port connection list consists of the terms in open and closed bracket ( ). The module port connections can be given in order (positional mapping), or the ports can be explicitly named as they are connected (named mapping). Named mapping is usually preferred for long connection lists as it makes errors less likely.

There are two ways to instantiate the ports:

1. Port Mapping by name : Don't have to follow order
   Example:
   INV V2 (.in (a), .out (abar));

2. Port mapping by order: Don't have to specify (.in) & (.out). The
   Example:
   AND A1 (a, b, aandb);

   If 'a' and 'b 'are the inputs and 'aandb' is the output, then the ports must be mentioned in the same order as shown above for the AND gate. One cannot write it in this way:
   AND A1 (aandb, a, b);

   It will consider 'aandb' as the input and result in an error.

   Example Verilog Code: D Flip Flop

```
module  dff ( d, clk, q , qbar)
input d, clk;
output q;

always @(posedge clk)
begin
q<=d;
qbar = !d;

end
```

endmodule

3.A.2 Digital Design

Digital Design can be broken into either Combinatorial Logic or Sequential Logic. As mentioned earlier, Hardware Description Languages are used to model RTL. RTL again is nothing but combinational and sequential logic. The most popular language used to model RTL is Verilog. The following are a few guidelines to code digital logic in Verilog:

1. Not everything written in Verilog is synthesizable. The Synthesis tool does not synthesize everything that is written. We need to make sure, that the logic implied is synthesized into what we want it to synthesize into and not anything else.
    a. Mostly, time dependant tasks are not synthesizable in Verilog. Some of the Verilog Constructs that are Non Synthesizable are task, wait, initial statements, delays, test benches etc
    b. Some of the verilog constructs that are synthesizable are assign statement, always blocks, functions etc. Please refer to next section for more detail information.
2. One can model level sensitive and also edge sensitive behavior in Verilog. This can be modeled using an always block in verilog.
    a. Every output in an 'always' block when changes and depends on the sensitivity list, becomes combinatorial circuit, basically the outputs have to be completely specified. If the outputs are not completely specified, then the logic will get synthesized to a latch. The following are a few examples to clarify this:
    b. Code which results in level sensitive behavior
    c. Code which results in edge sensitive behavior
    d. Case Statement Example
        i. casex
        ii. casez
3. Blocking and Non Blocking statements
    a. Example: Blocking assignment
    b. Example: Non Blocking assignment
4. Modeling Synchronous and Asynchronous Reset in Verilog
    a. Example: With Synchronous reset
    b. Example: With Asynchronous reset
5. Modeling State Machines in Verilog
    a. Using One Hot Encoding
    b. Using Binary Encoding

APPENDIX 3B: TEST BENCH / VERIFICATION

After designing the system, it is very vital do verify the logic designed. At the front end, this is done through simulation. In verilog, test benches are written to verify the code.

This topic deals with the whole verification process. Some basic guidelines for writing test benches:
→Test bench instantiates the top level design and provides the stimulus to the design.
→ Inputs of the design are declared as **'reg'** type. The reg data type holds a value until a new value is driven onto it in an initial or always block. The reg type can only be assigned a value in an always or initial block, and is used to apply stimulus to the inputs of the Device Under Test.
→Outputs of design declared as **'wire'** type. The wire type is a passive data type that holds a value driven on it by a port, assign statement or reg type. Wires can not be assigned values inside always and initial blocks.
→**Always and initial blocks** are two sequential control blocks that operate on reg types in a Verilog simulation. Each initial and always block executes concurrently in every module at the start of simulation. An example of an initial block is shown below

```
reg clk_50, rst_l;
initial
begin
$display($time, " << Starting the Simulation >>");
clk_50 = 1'b0; // at time 0
rst_l = 0; // reset is active
#20 rst_l = 1'b1; // at time 20 release reset
end
```

Initial blocks start executing sequentially at simulation time 0. Starting with the first line between the "begin end pair" each line executes from top to bottom until a delay is reached. When a delay is reached, the execution of this block waits until the delay time has passed and then picks up execution again. Each initial and always block executes concurrently. The initial block in the example starts by printing << Starting the Simulation >> to the screen, and initializes the reg types clk_50 and rst_l to 0 at time 0. The simulation time wheel then advances to time index 20, and the value on rst_l changes to a 1. This simple block of code initializes the clk_50 and rst_l reg types at the beginning of simulation and causes a reset pulse from low to high for 20 ns in a simulation.

→Some system tasks are called. These system tasks are ignored by the synthesis tool, so it is ok to use them. The system task variables begin with a '$' sign. Some of the system level tasks are as follows:

a. $Display: Displays text on the screen during simulation
b. $Monitor: Displays the results on the screen whenever the parameter changes.
c. $Strobe: Same as $display, but prints the text only at the end of the time step.
d. $Stop: Halts the simulation at a certain point in the code. The user can add the next set of instructions to the simulator. After $Stop, you get back to the CLI prompt.
e. $Finish: Exits the simulator
f. $Dumpvar, $Dumpfile: This dumps all the variables in a design to a file. You can dump the values at different points in the simulation.

→Tasks are a used to group a set of repetitive or related commands that would normally be contained in an initial or always block. A task can have inputs, outputs, and inouts, and can contain timing or delay elements. An example of a task is below

```
task load_count;
input [3:0] load_value;
begin
@(negedge clk_50);
$display($time, " << Loading the counter with %h >>", load_value);
load_l = 1'b0;
count_in = load_value;
@(negedge clk_50);
load_l = 1'b1;
end
endtask //of load_count
```

This task takes one 4-bit input vector, and at the negative edge of the next clk_50, it starts executing. It first prints to the screen, drives load_l low, and drives the count_in of the counter with the load_value passed to the task. At the negative edge of clk_50, the load_l signal is released. The task must be called from an initial or always block. If the simulation was extended and multiple loads were done to the counter, this task could be called multiple times with different load values.
→ The compiler directive `timescale:

```
`timescale 1 ns / 100 ps
```

This line is important in a Verilog simulation, because it sets up the time scale and operating precision for a module. It causes the unit delays to be in nanoseconds (ns) and the precision at which the simulator will round the events down to at 100 ps. This causes a #5 or #1 in a Verilog assignment to be a 5 ns or 1 ns delay respectively. The rounding of the events will be to .1ns or 100 pico seconds.

→ Verilog Test benches use a standard, which contains a description of the C language procedural interface, better known as programming language interface (PLI). We can treat PLI as a standardized simulator (Application Program Interface) API for routines written in C or C++. Most recent extensions to PLI are known as Verilog procedural interface (VPI);
→Before writing the test bench, it is important to understand the design specifications of the design, and create a list of all possible test cases.
→You can view all the signals and check to see if the signal values are correct, in the waveform viewer.
→ When designing the test bench, you can break-points at certain times, or can do simulation in a single step way, one can also have Time related breakpoints (Example: execute the simulation for 10ns and then stop)
→To test the design further, it is good to have randomized simulation. Random Simulation is nothing but supplying random combinations of valid inputs to the simulation tool and run it for a long time. When this random simulation runs for a long time, it could cover all corner cases and we can hope that it will emulate real system behavior. You can create random simulation in the test bench by using the $random variable.

→**Coverage Metric**: A way of seeing, how many possibilities exist and how many of them are executed in the simulation test bench. It is always good to have maximum coverage.

a. **Line Coverage:** It is the percentage of lines in the code, covered by the simulation tool.
b. **Condition Coverage:** It checks for all kinds of conditions in the code and also verifies to see if all the possibilities in the condition have been covered or not.
c. **State Machine Coverage:** It is the percentage of coverage, that checks to see if every sequence of the state transitions that are covered.
d. **Regression Test Suite:** This type of regression testing is done, when a new portion is added to the already verified code. The code is again tested to see if the new functionality is working and also verifies that the old code functionality has not been changed, due to the addition of the new code.

Goals of Simulation are:

1. **Functional Correctness:** To verify the functionality of the design by verifies main test cases, corner cases (Special conditions) etc…
2. **Error Handling**
3. **Performance**

Basic Steps in Simulation:

1. Compilation: During compilation, the verilog is converted to object code. It is done on a module basis.
2. Linking: This is step where module interconnectivity takes place. The object files are linked together and any kind of port mismatches (if any) occur.
3. Execution: An executable file is created and executed.

3.B.1 Test Bench Example:

The following is an example of a simple read, write, state machine design and a test bench to test the state machine.

State Machine:

```
module state_machine(sm_in,sm_clock,reset,sm_out);

parameter idle  = 2'b00;
parameter read  = 2'b01;
parameter write = 2'b11;
parameter wait  = 2'b10;
```

```
input sm_clock;
input reset;
input sm_in;
output sm_out;

reg [1:0] current_state, next_state;


always @ (posedge sm_clock)
  begin
  if (reset == 1'b1)
  current_state <= 2'b00;
  else
  current_state <= next_state;
  end


always @ (current_state or sm_in)
  begin
    // default values
    sm_out = 1'b1;
    next_state = current_state;
    case (current_state)
    idle:
      sm_out = 1'b0;
      if (sm_in)
      next_state = 2'b11;
    write:
      sm_out = 1'b0;
      if (sm_in == 1'b0)
      next_state = 2'b10;
    read:
      if (sm_in == 1'b1)
      next_state = 2'b01;
    wait:
      if (sm_in == 1'b1)
      next_state = 2'b00;
    endcase
  end

endmodule
```

Test Bench for State Machine

```
module testbench;

// parameter declaration section
// ...
parameter idle_state = 2'b00;
parameter read_state = 2'b01;
parameter write_state = 2'b11;
parameter wait_state = 2'b10;

// testbench declaration section
reg [500:1] message;
reg [500:1] state_message;
reg in1;
reg clk;
reg reset;
wire data_mux;

// instantiations
state_machine #(idle_state,
                read_state,
                write_state,
                wait_state)  st_mac (
  .sm_in       (in1),
  .sm_clock    (clk),
  .reset       (reset),
  .sm_out      (data_mux)
  );

// monitor section
always @ (st_mac.current_state)
  case (st_mac.current_state)
    idle_state : state_message = "idle";
    read_state : state_message = "read";
    write_state: state_message = "write";
    wait_state : state_message = "wait";
  endcase

// clock declaration
initial clk = 1'b0;
always #50 clk = ~clk;

// tasks
task reset_cct;
  begin
  @(posedge clk);
  message = " reset";
```

```
  @(posedge clk);
  reset = 1'b1;
  @(posedge clk);
  reset = 1'b0;
  @(posedge clk);
  @(posedge clk);
  end
endtask

task change_in1_to;
input  a;
begin
message = "change in1 task";
@ (posedge clk);
in1 = a;
end
endtask

// main task calling section
initial
begin
message = "start";
reset_cct;
change_in1_to (1'b1);
change_in1_to (1'b0);
change_in1_to (1'b1);
change_in1_to (1'b0);
change_in1_to (1'b1);
@ (posedge clk);
@ (posedge clk);
@ (posedge clk);
$stop;
end

endmodule
```

How do you simulate your design to get the real system behavior?

The following are two methods with which it id possible to achieve real system behavior and verify it.

1. FPGA Implementation: Speeds up verification and makes it more comprehensive.
2. Hardware Accelerator: It is nothing but a bunch of FPGA's implemented inside of a box. During compilation, it takes the part of the code that is synthesizable and maps it onto FPGA. Al the other non synthesizable part of the code such as test benches etc, are invoked by the simulation tools.

a. Basically RTL is mapped onto FPGA. The FPGA internally contains optimized files.
b. It translates signal transitions in the software part and signals on the FPGA and basically maps into the real signals that are there on the FPGA board.
c. This method of verification is good when there is a big design which has a lot of RTL, it also depends on the percentage of synthesizable code versus non-synthesizable code, if the amount of interaction between the codes, lesser the better.

Design Compiler Tutorial [RTL-Gate Level Synthesis]

4.0 Introduction

The Design Compiler is a synthesis tool from Synopsys Inc. In this tutorial you will learn how to perform hardware synthesis using Synopsys design compiler. In simple terms, we can say that the synthesis tool takes a RTL [Register Transfer Logic] hardware description [design written in either Verilog/VHDL], and standard cell library as input and the resulting output would be a technology dependent gate-level-netlist. The gate-level-netlist is nothing but structural representation of only standard cells based on the cells in the standard cell library. The synthesis tool internally performs many steps, which are listed below. Also below is the flowchart of synthesis process.

1. Design Compiler reads in technology libraries, DesignWare libraries, and symbol libraries to implement synthesis.

During the synthesis process, Design Compiler [DC] translates the RTL description to components extracted from the technology library and DesignWare library. The technology library consists of basic logic gates and flip-flops. The DesignWare library contains more complex cells for example adders and comparators which can be used for arithmetic building blocks. DC can automatically determine when to use Design Ware components and it can then efficiently synthesize these components into gate-level implementations.

2. Reads the RTL hardware description written in either Verilog/VHDL.
3. The synthesis tool now performs many steps including high-level RTL optimization, RTL to unoptimized Boolean logic, technology independent optimizations, and finally technology mapping to the available standard cells in the technology library, known as target library. This resulting gate-level-netlist also depends on constrains given. Constraints are the designer's specification of timing and environmental restrictions [area, power, process etc] under which synthesis is to be performed.

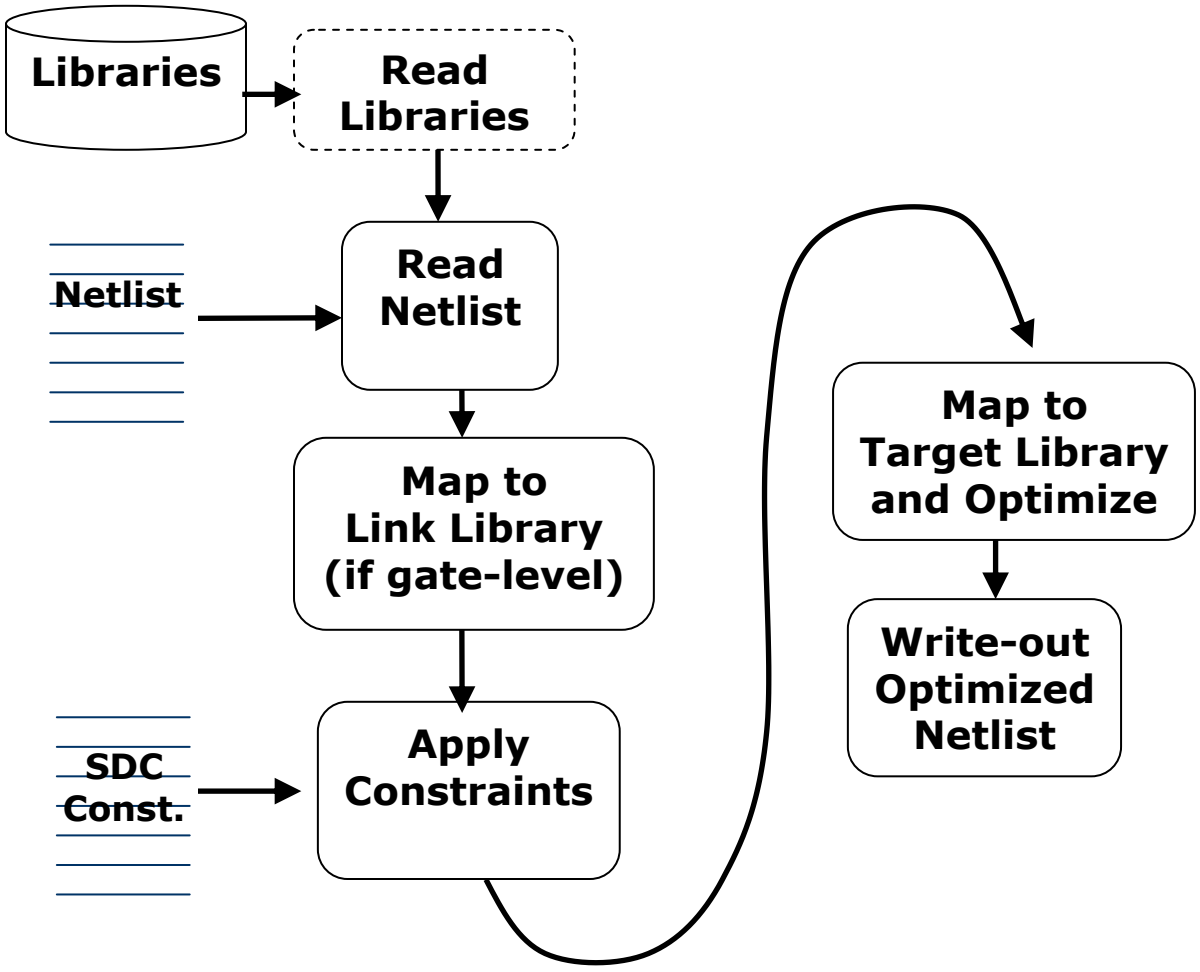
As an RTL designer, it is good to understand the target standard cell library, so that one can get a better understanding of how the RTL coded will be synthesized into gates. In this tutorial we will use Synopsys Design Compiler to read/elaborate RTL, set timing constraints, synthesize to gates, and report various QOR reports [timing/area reports etc]. Please refer ***to APPENDIX sA: Overview of RTL*** for more information.

4. After the design is optimized, it is ready for DFT [design for test/ test synthesis]. DFT is test logic; designers can integrate DFT into design during synthesis. This helps the designer to test for issues early in the design cycle and also can be used for debugging process after the chip comes back from fabrication.

In this tutorial, we will not be covering the DFT process. The synthesized design in the tutorial example is without the DFT logic. Please refer to tutorial on ***Design for Test*** for more information.

5. After test synthesis, the design is ready for the place and route tools. The Place and route tools place and physically interconnect cells in the design. Based on the physical routing, the designer can back-annotate the design with actual interconnect delays; we can use Design Compiler again to resynthesize the design for more accurate timing analysis.

Figure 4.a Synthesis Flow



While running DC, it is important to monitor/check the log files, reports, scripts etc to identity issues which might affect the area, power and performance of the design. In this

tutorial, we will learn how to read the various DC reports and also use the graphical Design Vision tool from Synopsys to analyze the synthesized design.

For Additional documentation please refer the below location, where you can get more information on the 90nm Standard Cell Library, Design Compiler, Design Vision, Design Ware Libraries etc.

4.1 BASIC SYNTHESIS GUIDELINES

4.1.1 Startup File

The Synopsys synthesis tool when invoked, through Design compiler command, reads a startup file, which must be present in the current working directory. This startup file is **synopsys_dc.setup** file. There should be two startup files present, one in the current working directory and other in the root directory in which Synopsys is installed. The local startup file in the current working directory should be used to specify individual design specifications. This file does not contain design dependent data. Its function is to load the Synopsys technology independent libraries and other parameters. The user in the startup files specifies the design dependent data. The settings provided in the current working directory override the ones specified in the root directory.

There are four important parameters that should be setup before one can start using the tool. They are:

• **search_path**

This parameter is used to specify the synthesis tool all the paths that it should search when looking for a synthesis technology library for reference during synthesis.

• **target_library**

The parameter specifies the file that contains all the logic cells that should used for mapping during synthesis. In other words, the tool during synthesis maps a design to the logic cells present in this library.

• **symbol_library**

This parameter points to the library that contains the "visual" information on the logic cells in the synthesis technology library. All logic cells have a symbolic representation and information about the symbols is stored in this library.

• **link_library**

This parameter points to the library that contains information on the logic gates in the synthesis technology library. The tool uses this library solely for reference but does not use the cells present in it for mapping as in the case of target_library.

An example on use of these four variables from a **.synopsys_dc.setup** file is given below.

search_path = ". /synopsys/libraries/syn/cell_library/libraries/syn"
target_library = class.db
link_library = class.db
symbol_library = class.db

Once these variables are setup properly, one can invoke the synthesis tool at the command prompt using any of the commands given for the two interfaces.

4.1.2 Design Objects

There are eight different types of objects categorized by Design Compiler.

Design: It corresponds to the circuit description that performs some logical function. The design may be stand-alone or may include other sub-designs. Although sub-design may be part of the design, it is treated as another design by the Synopsys.
Cell: It is the instantiated name of the sub-design in the design. In Synopsys terminology, there is no differentiation between the cell and instance; both are treated as cell.
Reference: This is the definition of the original design to which the cell or instance refers. For e.g., a leaf cell in the netlist must be referenced from the link library, which contains the functional description of the cell. Similarly an instantiated sub-design must be referenced in the design, which contains functional description of the instantiated subdesign.
Ports: These are the primary inputs, outputs or IO's of the design.
Pin: It corresponds to the inputs, outputs or IO's of the cells in the design. (Note the difference between port and pin)
Net: These are the signal names, i.e., the wires that hook up the design together by connecting ports to pins and/or pins to each other.
Clock: The port or pin that is identified as a clock source. The identification may be internal to the library or it may be done using **dc_shell** commands.
Library: Corresponds to the collection of technology specific cells that the design is targeting for synthesis; or linking for reference.

Design Entry
Before synthesis, the design must be entered into the Design Compiler (referred to as DC from now on) in the RTL format. DC provides the following two methods of design entry:

read command
analyze & elaborate commands

The **analyze & elaborate** commands are two different commands, allowing designers to initially analyze the design for syntax errors and RTL translation before building the generic logic for the design. The generic logic or GTECH components are part of Synopsys generic technology independent library. They are unmapped representation of boolean functions and serve as placeholders for the technology dependent library.

The **analyze** command also stores the result of the translation in the specified design library that maybe used later. So a design analyzed once need not be analyzed again and can be merely elaborated, thus saving time. Conversely **read** command performs the function of **analyze** and **elaborate** commands but does not store the analyzed results, therefore making the process slow by comparison.

Parameterized designs (such as usage of *generic* statement in VHDL) must use **analyze** and **elaborate** commands in order to pass required parameters, while elaborating the design. The **read** command should be used for entering pre-compiled designs or netlists in DC .

One other major difference between the two methods is that, in **analyze** and **elaborate** design entry of a design in VHDL format, one can specify different architectures during elaboration for the same analyzed design. This option is not available in the **read** command.

The commands used for both the methods in DC are as given below:
Read command:
dc_shell>read –format <format> <list of file names>
"-format" option specifies the format in which the input file is in, e.g. VHDL
Sample command for a reading "adder.vhd" file in VHDL format is given below

dc_shell>read –format vhdl adder.vhd
Analyze and Elaborate commands:

Or

dc_shell>read –format verilog adder.v
Analyze and Elaborate commands:

dc_shell>analyze -format <format> <list of file names>

dc_shell>elaborate <.syn file> -arch "<architecture >" –param "<parameter>"
.syn file is the file in which the analyzed information of the design analyzed is stored.
e.g: The adder entity in the adder.vhd has a generic parameter "width" which can be specified while elaboration. The architecture used is "beh" defined in the adder.vhd file.
The commands for analyze and elaborate are as given below:

dc_shell> analyze -format vhdl adder.vhd

dc_shell> elaborate adder –arch "beh" –param "width = 32"

4.1.3 Technology Library

Technology libraries contain the information that the synthesis tool needs to generate a netlist for a design based on the desired logical behavior and constraints on the design. The tool referring to the information provided in a particular library would make appropriate choices to build a design. The libraries contain not only the logical function of an ASIC cell, but the area of the cell, the input-to-output timing of the cell, any constraints on fanout of the cell, and the timing checks that are required for the cell.

Other information stored in the technology library may be the graphical symbol of the cell for use in creating the netlist schematic.

The **target_library**, **link_library**, and **symbol_library** parameters in the **startup** file are used to set the technology library for the synthesis tool.

The Synopsys® .lib technology library contains the following information
• Wire-load models for net length and data estimation. Wire-load models available in the technology library are statistical and hence inaccurate when estimating data.
• Operating Conditions along with scaling k-factors for different delay components to model the effects of temperature, process, and voltage on the delay numbers.
• Specific delay models like piece-wise linear, non-linear, cmos2 etc. for calculation of delay values.
For each of the technology primitive cells the following information is modeled
• Interface pin names, direction and other information.
• Functional descriptions for both combinational and sequential cells which can be modeled in Synopsys®
• Pin capacitance and drive capabilities
• Pin to pin timing
• Area

4.1.4 Register Transfer-Level Description

A Register Transfer-Level description is a style that specifies a particular design in terms or registers and combinational logic in between. This is be shown by the "register and cloud" diagram in Fig 2.0

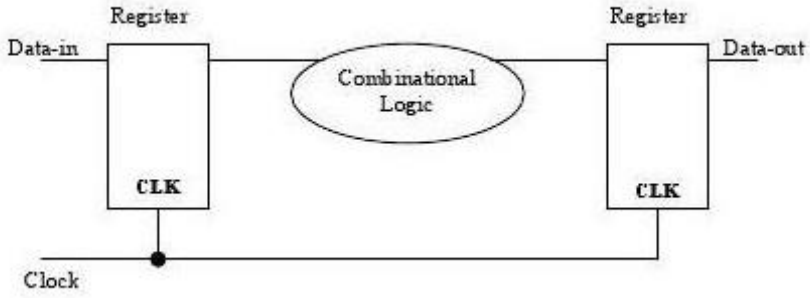


Figure 4.b. Register and cloud diagram

The registers can be described explicitly, through component instantiation, or implicitly, through inference. The combinational logic is described either by logical equations, sequential control statements (CASE, IF then ELSE, etc.), subprograms, or through concurrent statements and are represented by cloud objects in figure 2.0 between the registers.

RTL is the most popular form of high-level design specification. A good coding style would help the synthesis tool generate a design with minimal area and maximum performance.

4.1.5 General Guidelines

Following are given some guidelines which if followed might improve the performance of the synthesized logic, and produce a cleaner design that is suited for automating the synthesis process.

- Clock logic including clock gating and reset generation should be kept in one block – to be synthesized once and not touched again. This helps in a clean specification of the clock constraints. Another advantage is that the modules that are being driven by the clock logic can be constrained using the ideal clock specifications.
- No glue logic at the top: The top block is to be used only for connecting modules together. It should not contain any combinational glue logic. This removes the time consuming top-level compile, which can now be simply stitched together without undergoing additional synthesis.
- Module name should be same as the file name and one should avoid describing more that one module or entity in a single file. This avoids any confusion while compiling the files and during the synthesis.
- While coding finite state machines, the state names should be described using the enumerated types. The combinational logic for computing the next state should be in its own process, separate from the state registers. Implement the next-state combinational logic with a case statement. This helps in optimizing the logic much better and results in a cleaner design.
- Incomplete sensitivity lists must be avoided as this might result in simulation mismatches between the source RTL and the synthesized logic.
- Memory elements, latches and flip-flops: A latch is inferred when an incomplete *if* statement with a missing *else* part is specified. A flip-flop, or a register, is inferred when an edge sensitive statement is specified in the always statement for Verilog and process statement for VHDL. A latch is more troublesome than a latch as it makes static timing analysis on designs containing latches. So designers try to avoid latches and prefer flipflops more to latches.
- Multiplexer Inference: A *case* statement is used for implementing multiplexers. To prevent latch inferences in case statements the *default* part of the *case* statement should always be specified. On the other hand an if statement is used for writing priority encoders. Multiple if statements with multiple branches result in the creation of a priority encoder structure.

Ex: always @ (A, B, C)
begin
if A= 0 then D = B; end if;
if A= 1 then D = C; end if;
end

The above example infers a priority encoder with the first *if* statement given the precedence. The same code can be written using a *case* statement to implement a multiplexer as follows.

```
always @ (A, B, C)
begin
case (A) is
when 0 => D = B;
when others => D = C;
end case;
end
```

The same code can be written using *if* statement along with *elsif* statements to cover all possible branches.

 Three state buffers: A tri-state buffer is inferred whenever a high impedance (Z) is assigned to an output. Tri-state logic is generally not always recommended because it reduces testability and is difficult to optimize – since it cannot be buffered.

 Signals versus Variables in VHDL: Signal assignments are order independent, i.e. the order in which they are placed within the process statement does not have any effect on the order in which they are executed as all the signal assignments are done at the end of the process. The variable assignments on the other hand are order dependent. The signal assignments are generally used within the sequential processes and variable assignments are used within the combinational processes.

4.1.6 Design Attributes and Constraints

A designer, in order to achieve optimum results, has to methodically constrain the design, by describing the design environment, target objectives and design rules. The constraints contain timing and/or area information, usually derived from the design specifications. The synthesis tool uses these constraints to perform synthesis and tries to optimize the design with the aim of meeting target objectives.

4.1.6.1 Design Attributes

Design attributes set the environment in which a design is synthesized. The attributes specify the process parameters, I/O port attributes, and statistical wire-load models. The most common design attributes and the commands for their setting are given below:

***Load*:** Each output can specify the drive capability that determines how many loads can be driven within a particular time. Each input can have a load value specified that determines how much it will slow a particular driver. Signals that are arriving later than the clock can have an attribute that specifies this fact. The load attribute specifies how much capacitive load exists on a particular output signal. The load value is specified in the units of the technology library in terms of picofarads or standard loads, etc... The command for setting this attribute is given below:

set_load <value> <object_list>

e.g. **dc_shell> set_load 1.5 x_bus**

***Drive*:** The drive specifies the drive strength at the input port. It is specified as a resistance value. This value controls how much current a particular driver can source. The larger a driver is, i.e 0 resistance, the faster a particular path will be, but a larger driver will take more area, so the designer needs to trade off speed and area for the best performance. The command for setting the drive for a particular object is given below

set_drive <value> <object_list>
e.g. **dc_shell> set_drive 2.7 ybus**

4.1.6.2 Design Constraints

Design constraints specify the goals for the design. They consist of area and timing constraints. Depending on how the design is constrained the DC/DA tries to meet the set objectives. Realistic specification is important, because unrealistic constraints might result in excess area, increased power and/or degrading in timing. The basic commands to constrain the design are

set_max_area: This constraint specifies the maximum area a particular design should have. The value is specified in units used to describe the gate-level macro cells in the technology library.
e.g. **dc_shell> set_max_area 0**
Specifying a 0 area might result in the tool to try its best to get the design as small as possible

create_clock: This command is used to define a clock object with a particular period and waveform. The –**period** option defines the clock period, while the –**waveform** option controls the duty cycle and the starting edge of the clock. This command is applied to a pin or port, object types.

Following example specifies that a port named CLK is of type "clock" that has a period of 40 ns, with 50% duty cycle. The positive edge of the clock starts at time 0 ns, with the falling edge occurring at 20 ns. By changing the falling edge value, the duty cycle of the clock may be altered.
e.g. **dc_shell> create_clock –period 40 –waveform {0 20} CLK**

set_don't_touch_network: This is a very important command, usually used for clock networks and resets. This command is used to set a **dont_touch** property on a port, or on the net. Note setting this property will also prevent DC from buffering the net. In addition any gate coming in contact with the "don't_touch" net will also inherit the attribute.
e.g. **dc_shell> set_dont_touch_network {CLK, RST}**

set_don't_touch: This is used to set a don_touch property on the **current_design,** cells, references, or nets. This command is frequently used during hierarchical compilation of blocks for preventing the DC from optimizing the don't_touch object.
e.g. **dc_shell> set_don't_touch current_design**

current_design is the variable referencing the current working design. It can be set using the **current_design** command as follows
dc_shell>current_design <design_name>

set_input_delay: It specifies the input arrival time of a signal in relation to the clock. It is used at the input ports, to specify the time it takes for the data to be stable after the clock edge. The timing specification of the design usually contains this information, as the setup/hold time requirements for the input signals. From the top-level timing specifications the sub-level timing specifications may also be extracted.
e.g. **dc_shell> set_input_delay –max 23.0 –clock CLK {datain}**
dc_shell> set_input_delay –min 0.0 –clock CLK {datain}

The CLK has a period of 30 ns with 50% duty cycle. For the above given specification of max and min input delays for the datain with respect to CLK, the setup-time requirement for the input signal datain is 7ns, while the hold-time requirement is 0ns.

set_output_delay: This command is used at the output port, to define the time it takes for the data to be available before the clock edge. This information is usually is provided in the timing specification.
e.g. **dc_shell> set_output_delay – max 19.0 –clock CLK {dataout}**

The CLK has a period of 30 ns with 50% duty cycle. For the above given specification of max output delay for the dataout with respect to CLK, the data is valid for 11 ns after the clock edge.

set_max_delay: It defines the maximum delay required in terms of time units for a particular path. In general it is used for blocks that contain combination logic only. However it may also be used to constrain a block that is driven by multiple clocks, each with a different frequency. This command has precedence over DC derived timing requirements.
e.g. **dc_shell> set_max_delay 5 –from all_inputs() – to_all_outputs()**

set_min_delay: It defines the minimum delay required in terms of time units for a particular path.. It is the opposite of the set_max_delay command. This command has precedence over DC derived timing requirements.
e.g. **dc_shell> set_max_delay 3 –from all_inputs() – to_all_outputs()**

4.2 Tutorial Example

Setup

1. Write the Verilog Code. For the purpose of this tutorial, please consider the simple verilog code for gray counter below.

Gray Code Counter

```
// MODULE: Sequential Circuit Example: gray_counter.v
// MODULE DECLARATION
module graycount (gcc_out, reset_n, clk, en_count);

  output [2-1:0] gcc_out;        // current value of counter
  input reset_n;                                  // active-low RESET signal
  input clk;                                      // clock signal
  input en_count;                                 // counting is enabled when en_count = 1

  // SIGNAL DECLARATIONS
  reg [2-1:0] gcc_out;
// Compute new gcc_out value based on current gcc_out value
  always @(negedge reset_n or posedge clk) begin
    if (~reset_n)
      gcc_out <= 2'b00;
    else begin    // MUST be a (posedge clk) - don't need “else if (posedge clk)"
      if (en_count) begin          // check the count enable
        case (gcc_out)
          2'b00: begin         gcc_out <= 2'b01;   end
          2'b01: begin         gcc_out <= 2'b11;   end
          2'b11: begin         gcc_out <= 2'b10;   end
          default: begin       gcc_out <= 2'b00;          end
        endcase   // of case
      end // of if (en_count)
    end  // of else
  end    // of always loop for computing next gcc_out value

endmodule
```

2. As soon as you log into your engr account, at the command prompt, please type “csh “as shown below. This changes the type of shell from bash to c-shell. All the commands work ONLY in c-shell.

[hkommuru@hafez]$csh

2. Please copy the whole directory from the below location

[hkommuru@hafez]$cd
[hkommuru@hafez]$ cp –rf /packages/synopsys/setup/asic_flow_setup .

This ccreate directory structure as shown below. It will create a directory called **“asic_flow_setup ”,** under which it creates the following directories namely

asic_flow_setup

src/ : for verilog code/source code
vcs/ : for vcs simulation ,
synth_graycounter/ : for synthesis
synth_fifo/ : for synthesis
pnr/ : for Physical design
extraction/: for extraction
pt/: for primetime
verification/: final signoff check

The **"asic_flow_setup"** directory will contain all generated content including, VCS simulation, synthesized gate-level Verilog, and final layout. In this course we will always try to keep generated content from the tools separate from our source RTL. This keeps our project directories well organized, and helps prevent us from unintentionally modifying the source RTL. There are subdirectories in the project directory for each major step in the ASIC Flow tutorial. These subdirectories contain scripts and configuration files for running the tools required for that step in the tool flow. For this tutorial we will work exclusively in the ***vcs*** directory.

3. Please source "***synopsys_setup.tcl***" which sets all the environment variables necessary to run the VCS tool.
Please source them at unix prompt as shown below

[hkommuru@hafez]$ source /packages/synopsys/setup/synopsys_setup.tcl

Please Note : You have to do steps 1 and 3 above everytime you log in.

4. Please open the "dc_synth.tcl" at below location

[hkommuru@hafez]$cd
[hkommuru@hafez.sfsu.edu] $cd asic_flow_setup/synth_graycounter
[hkommuru@hafez.sfsu.edu] $cd scripts
[hkommuru@hafez.sfsu.edu] $emacs dc_synth.tcl &
[hkommuru@hafez.sfsu.edu] $cd ..

4.2.1 Synthesizing the Code

5. First we will learn how to run dc_shell manually, before we automate the scripts. Use the below command invoke dc_shell

[hkommuru@hafez.sfsu.edu] $ dc_shell-xg-t
Initializing...
dc_shell-xg-t>

Once you get the prompt above, you can run various commands to load verilog files, libraries etc. To get more information on any command you can type *"**man <command_name>*** at the prompt.

6. Type/Copy in the below commands at the command prompt, from the "dc_synth.tcl" which you have already opened in STEP 4.

dc_shell-xg-t> lappend search_path ../src/gray_counter
dc_shell-xg-t> define_design_lib WORK –path "work"
dc_shell-xg-t > set link_library [list
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_min.db
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_typ.db]
dc_shell-xg-t > set target_library [list
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db]

The command ***"lappend search path"*** tells the tool to search for the verilog code in that particular directory] to the verilog source code directory.

The next command ***"define_design_lib",*** creates a Synopsys work directory, and the la last two commands ***" set link_library "*** and ***" set target_library "*** point to the standard technology libraries we will be using. The DB files contain wireload models [Wire load modeling allows the tool to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets, calculate wire delays and circuit speeds], area and timing information for each standard cell. DC uses this information to optimize the synthesis process. For more detail information on optimization, please refer to the DC manual.

7. The next step is to load your Verilog/VHDL design into Design Compiler. The commands to load verilog are ***"analyze"*** and ***"elaborate".*** Executing these commands results in a great deal of log output as the tool elaborates some Verilog constructs and starts to infer some high-level components. Try executing the commands as follows.

dc_shell_xg-t > analyze –library WORK –format verilog gray_counter.v
dc_shell_xg-t> elaborate –architecture verilog –library WORK graycount

Notice, that the *graycount* is the name of the top module to be synthesized and not the name of the verilog file (*gray_counter.v*). You can see part of the analyze command in Figure 7.a below

Figure 4.a : Fragment of analyze command

```
dc_shell-xg-t> analyze -library WORK -format verilog gray_counter.v
Running PRESTO HDLC
Searching for ./gray_counter.v
Searching for /packages/synopsys/synthesis/X-2005.09-SP3/libraries/syn/gray_counter.v
Searching for /packages/synopsys/synthesis/X-2005.09-SP3/dw/sim_ver/gray_counter.v
Searching for ../src/gray_counter/gray_counter.v
Compiling source file ../src/gray_counter/gray_counter.v
Presto compilation completed successfully.
```

You can see Figure 7.b, which shows you a part of the elaboration, for the above gray code; the tool has inferred flipflop with 2 bit width. Please make sure that you check your design at this stage in the log to check if any latches inferred. We typically do not want latches inferred in the design.

Before DC optimizes the design, it uses Presto Verilog Compiler [for verilog code], to read in the designs; it also checks the code for the correct syntax and builds a generic technology (GTECH) netlist. DC uses this GTECH netlist to optimize the design. You could also use "read_verilog" command, which basically combines both elaborate and analyze command into one. You can use "read_verilog" as long as your design is not parameterized, meaning look at the below example of a register.

```
module dflipflop( inp, clk, outp );
parameter SIZE = 8;
input [SIZE-1:0] inp;
input clk;
output [SIZE-1:0] outp;
reg [SIZE-1:0] outp;
reg [SIZE-1:0] tmp;
always @(clk)
if (clk == 0)
tmp = inp;
else //(clk == 1)
out1 <= tmp;
endmodule
```

If you want an instance of the above register to have a bit-width of 32, use the elaborate command to specify this as follows:

```
elaborate dflipflop -param SIZE=32
```

For more information on the "elaborate" command, and how the synthesis tool infers combinational and sequential elements, please refer to ***Presto HDL Compiler Reference Manual*** found in the documentation area.

Figure 4.b Fragment of elaborate command

```
dc_shell-xg-t> elaborate   -architecture verilog -library WORK graycount
Running PRESTO HDLC

Statistics for case statements in always block at line 24 in file
        '../src/gray_counter.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            30            |    auto/auto     |
===============================================

Inferred memory devices in process
        in routine graycount line 24 in file
                '../src/gray_counter.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|     gcc_out_reg     | Flip-flop |   2   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'graycount'.
1
dc_shell-xg-t>
```

8. Next, we check to see if the design is in a good state or consistent state; meaning that there are no errors such as unconnected ports, logical constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets etc.

dc_shell-xg-t> check_design

Please go through, the check_design errors and warnings. DC cannot compile the design if there are any errors. Many of the warning's may not an issue, but it is still useful to skim through this output.

9. After the design compile is clean, we need to tell the tool the constraints, before it actually synthesizes. The tool needs to know the target frequency you want to synthesize. Take a look at the "create_clock" command below.

dc_shell-xg-t> create_clock clk -name ideal_clock1 -period 5

The above command tells the tool that the pin named clk is the clock and that your desired clock period is 5 nanoseconds. We need to set the clock period constraint carefully. If the period is unrealistically small, then the tools will spend forever trying to meet timing and ultimately fail. If the period is too large, then the tools will have no trouble but you will get a very conservative implementation.

You could also add additional constraints such as constrain the arrival of certain input signals, the drive strength of the input signals, capacitive load on the output signals etc. Below are some examples. These constraints are defined by you, the user; hence we can call them user specified constraints.
Set input constraints by defining how much time would be spent by signals arriving into your design, outside your design with respect to clock.

dc_shell-xg-t> set_input_delay 2.0 [remove_from_collection [all_inputs] clk] –clock ideal_clock1

Similarly you can define output constraints, which define how much time would be spent by signals leaving the design, outside the design, before being captured by the same clk.

dc_shell-xg-t> set_output_delay 2.0 [all_outputs] –clock ideal_clock1

Set area constraints: set maximum allowed area to 0 ☺, well it's just to instruct design compiler to use as less area as possible.

dc_shell-xg-t > set_max_area 0

Please refer to tutorial on "***Basics of Static Timing Analysis***" for more understanding of concepts of STA and for more information on the commands used in STA, please refer to the ***Primetime Manual and DC Compiler Manual*** at location /packages/synopsys/

10. Now we are ready to use the compile command to actually synthesize our design into a gate-level netlist. Two of the most important options for the compile command are the map effort and the area effort. Both of these can be set to one of *none, low, medium, or high*. They specify how much time to spend on technology mapping and area reduction.

dc_shell-xg-t> compile -map_effort medium -area_effort medium

DC will attempt to synthesize your design while still meeting the constraints. DC considers two types of constraints: user specified constraints and design rule constraints. We looked at the user specified constraints in the previous step. Design rule constraints are fixed constraints which are specified by the standard cell library. For example, there are restrictions on the loads specific gates can drive and on the transition times of certain pins. To get a better understanding of the standard cell library, please refer to Generic 90nm library documents in the below location which we are using in the tutorial.

/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/doc/databook/

Also, note that the compile command does not optimize across module boundaries. You have to use "set flatten" command to enable inter-module optimization. For more information on the compile command consult the ***Design Compiler User Guide (dc-user-guide.pdf)*** or use man compile at the DC shell prompt.

The compile command will report how the design is being optimized. You should see DC performing technology mapping, delay optimization, and area reduction. Figure 7.c shows a fragment from the compile output. Each line is an optimization pass. The area column is in units specific to the standard cell library, but for now you should just use the area numbers as a relative metric. The worst negative slack column shows how much room there is between the critical path in your design and the clock constraint. Larger negative slack values are worse since this means that your design is missing the desired clock frequency by a greater amount. Total negative slack is the sum of all negative slack across all endpoints in the design - if this is a large negative number it indicates that not only is the design not making timing, but it is possible that many paths are too slow. If the total negative slack is a small negative number, then this indicates that only a few paths are too slow. The design rule cost is an indication of how many cells violate one of the standard cell library design rules constraints.

You can use the compile command more than once, as many iterations as you want, for example, first iteration you can optimize only timing, but it might come with high area cost, for second iteration, it optimizes area, but could cause the design to no longer meet timing. There is no limit on number of iterations; however each design is different, and you need to do number of runs, to decide how many iterations it needs.

We can now use various commands to examine timing paths, display reports, and further optimize the design. Using the shell directly is useful for finding out more information about a specific command or playing with various options.

Figure 4.c: Fragment of Compile command

```
Beginning Pass 1 Mapping
------------------------
Processing 'graycount'

Updating timing information
Information: Updating design information... (UID-85)

Beginning Mapping Optimizations  (Medium effort)
-------------------------------

 ELAPSED            WORST NEG TOTAL NEG  DESIGN
  TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
--------- --------- --------- --------- --------- --------------------------
  0:00:05     103.2      0.00       0.0       0.0
  0:00:05      99.5      0.00       0.0       0.0


Beginning Delay Optimization Phase
----------------------------------

 ELAPSED            WORST NEG TOTAL NEG  DESIGN
  TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
--------- --------- --------- --------- --------- --------------------------
  0:00:05      99.5      0.00       0.0       0.0


Beginning Area-Recovery Phase  (cleanup)
-----------------------------

 ELAPSED            WORST NEG TOTAL NEG  DESIGN
  TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
--------- --------- --------- --------- --------- --------------------------
  0:00:05      99.5      0.00       0.0       0.0
  0:00:05      99.5      0.00       0.0       0.0
  0:00:05      94.0      0.00       0.0       0.0

Optimization Complete
---------------------
```

4.2.2 Interpreting the Synthesized Gate-Level Netlist and Text Reports

In addition to the actual synthesized gate-level netlist, the dc_synth.tcl also generates several text reports. Reports usually have the rpt filename suffix. The following is a list of the synthesis reports.
The synth area.rpt report contains area information for each module in the design. 7.d shows a fragment from synth_area.rpt. We can use the synth_area.rpt report to gain insight into how various modules are being implemented. We can also use the area report to measure the relative area of the various modules.

You can find all these reports in the below location for your reference.
/packages/synopsys/setup/project_dc/synth/reports/

You can also look at command.log , in the synth directory, which will list all the commands used in the current session.

synth_area.rpt - Contains area information for each module instance

Figure 4.d : Fragment of area report

```
Library(s) Used:

    saed90nm_typ (File:
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm
/Digital_Standard_Cell_Library/synopsys/models/saed90nm_typ.db)

Number of ports:                5
Number of nets:                 9
Number of cells:                5
Number of references:           3

Combinational area:         29.492001
Noncombinational area:      64.512001
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:            94.003998
Total area:                 undefined
1
```

The **synth_cells.rpt - Contains the cells list in the design ,** as you can see in Figure 4.e . From this report , you can see the breakup of each cell area in the design.

Figure 4.e: Fragment of cell area report

```
Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell               Reference       Library             Area  Attributes
--------------------------------------------------------------------------
U2                         AO22X1          saed90nm_typ    11.981000
U3                         AO22X1          saed90nm_typ    11.981000
U4                         INVX0           saed90nm_typ    5.530000
gcc_out_reg[0]             DFFARX1         saed90nm_typ    32.256001 n
gcc_out_reg[1]             DFFARX1         saed90nm_typ    32.256001 n
--------------------------------------------------------------------------
---------
Total 5 cells                                              94.003998
1
```

Synth_qor.rpt – Contains summary information on the area , timing, critical paths violations in the design. You can take a look at this report to understand the overall quality of your design. Figure 4.f shows the example. As you can see in Figure 4.f , there is no negative slack in the design that means the design is meeting timing.

Figure 4.f : Fragment of qor report

```
Timing Path Group 'ideal_clock1'
-----------------------------------
Levels of Logic:               2.00
Critical Path Length:          0.12
Critical Path Slack:           2.77
Critical Path Clk Period:      5.00
Total Negative Slack:          0.00
No. of Violating Paths:        0.00
-----------------------------------


Cell Count
-----------------------------------
Hierarchial Cell Count:           0
Hierarchial Port Count:           0
Leaf Cell Count:                  5
-----------------------------------


Area
-----------------------------------
Combinational Area:       29.492001
Noncombinational Area:    64.512001
Net Area:                  0.000000
-----------------------------------
Cell Area:                94.003998
Design Area:              94.003998


Design Rules
-----------------------------------
Total Number of Nets:             9
Nets With Violations:             0
-----------------------------------


Hostname: hafez.sfsu.edu

Compile CPU Statistics
-----------------------------------
Resource Sharing:              0.00
Logic Optimization:            0.31
Mapping Optimization:          0.33
-----------------------------------
Overall Compile Time:          3.64
```

1

synth_timing.rpt - Contains critical timing paths

You can see below an example of a timing report dumped out from synthesis . You can see at the last line of the Figure 7.f , this paths meets timing. The report lists the critical path of the design. The critical path is the slowest logic path between any two registers

and is therefore the limiting factor preventing you from decreasing the clock period constraint
(and thus increasing performance). The report is generated from a purely static worst-case timing analysis (i.e. independent of the actual signals which are active when the processor is running). In the example below, since it's a simple gray counter, the crtical path is from the port to the register.
Please note that the last column lists the cumulative delay to that node, while the middle column shows the incremental delay. You can see that the datapath is and Inverter, complex gate before it reaches the register which is 0.12ns. From our SDC constraints, we set 2ns delay on the input port. So, the total delay so far is 2.12ns. Notice, however, that the final register file flip-flop has a setup time of 0.11 ns, the clock period is 5ns. Therefore 5ns-0.11ns=4.89ns, is the time before which the register should latch the data. The critical path delay is however only 2.12ns, so there is more than enough time for the path to meet timing.

Figure 4.g: Fragment of Timing report

```
Operating Conditions: TYPICAL   Library: saed90nm_typ
Wire Load Model Mode: top

  Startpoint: en_count (input port)
  Endpoint: gcc_out_reg[1]
            (rising edge-triggered flip-flop clocked by ideal_clock1)
  Path Group: ideal_clock1
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock (input port clock) (rise edge)     0.00       0.00
  input external delay                     2.00       2.00 f
  en_count (in)                            0.00       2.00 f
  U4/QN (INVX0)                            0.02       2.02 r
  U2/Q (AO22X1)                            0.10       2.12 r
  gcc_out_reg[1]/D (DFFARX1)               0.00       2.12 r
  data arrival time                                   2.12

  clock ideal_clock1 (rise edge)           5.00       5.00
  clock network delay (ideal)              0.00       5.00
  gcc_out_reg[1]/CLK (DFFARX1)             0.00       5.00 r
  library setup time                      -0.11       4.89
  data required time                                  4.89
  -----------------------------------------------------------
  data required time                                  4.89
  data arrival time                                  -2.12
  -----------------------------------------------------------
  slack (MET)                                        2.77
```

synth_resources.rpt - Contains information on Design Ware components

```
=> In the above example, the file will be empty since the graycounter
did not need any of the complex cells.
```

```
****************************************
Report : resources
Design : graycount
Version: X-2005.09-SP3
Date   : Mon Mar  9 21:30:37 2009
****************************************


No resource sharing information to report.

No implementations to report

No multiplexors to report
```

synth_check_design.rpt - Contains output from check design command, which is clean in the above example.

Below is the gate-level netlist output of the gray counter RTL code after synthesis.

Figure 4.h : Synthesized gate-level netlist

```
module graycount ( gcc_out, reset_n, clk, en_count );
  output [1:0] gcc_out;
  input reset_n, clk, en_count;
  wire   N8, n1, n4, n5, n6;
  assign gcc_out[0] = N8;

  AO22X1 U2
( .IN1(gcc_out[1]), .IN2(n1), .IN3(en_count), .IN4(N8), .Q(n4) );
  AO22X1 U3 ( .IN1(en_count), .IN2(n6), .IN3(N8), .IN4(n1), .Q(n5) );
  INVX0 U4 ( .IN(en_count), .QN(n1) );
  DFFARX1 \gcc_out_reg[0]
( .D(n5), .CLK(clk), .RSTB(reset_n), .Q(N8) );
  DFFARX1 \gcc_out_reg[1]
( .D(n4), .CLK(clk), .RSTB(reset_n), .Q(gcc_out[1]),
        .QN(n6) );
endmodule
```

4.2.3 Synthesis Script

```
###### Synthesis Script #######

## Give the path to the verilog files and define the WORK directory

lappend search_path ../src/gray_counter
define_design_lib WORK -path "work"
```

```
## Define the library location
set                  link_library                  [                  list
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm
/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db
packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/
Digital_Standard_Cell_Library/synopsys/models/saed90nm_typ.db
packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/
Digital_Standard_Cell_Library/synopsys/models/saed90nm_min.db]

set                target_library                  [                  list
/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm
/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db ]

## read the verilog files
analyze -library WORK -format verilog gray_counter.v

elaborate   -architecture verilog -library WORK graycount

## Check if design is consistent
check_design  > reports/synth_check_design.rpt

## Create Constraints
create_clock clk -name ideal_clock1 -period 5
set_input_delay  2.0  [remove_from_collection  [all_inputs]  clk  ]  -clock
ideal_clock1
set_output_delay 2.0 [all_outputs] -clock ideal_clock1
set_max_area 0

## Compilation
## you can change medium to either low or high
compile -area_effort medium -map_effort medium


##  Below  commands  report  area  ,  cell,  qor,  resources,  and  timing
information needed to analyze the design.

  report_area > reports/synth_area.rpt
  report_cell > reports/synth_cells.rpt
  report_qor  > reports/synth_qor.rpt
  report_resources > reports/synth_resources.rpt
  report_timing -max_paths 10 > reports/synth_timing.rpt

## Dump out the constraints in an SDC file

  write_sdc  const/gray_counter.sdc

## Dump out the synthesized database and gate leedl nedtlist
  write -f ddc -hierarchy -output output/gray_counter.ddc
  write -hierarchy -format verilog -output  output/gray_counter.v


## You can play with the commands or exit

  exit
```

Note : There is another synthesis example of a FIFO in the below location for further reference. This synthesized FIFO example is used in the physical design IC Compiler Tutorial

Location :
/packages/synopsys/setup/asic_flow_setup/synth_fifo

APPENDIX 4A: SYNTHESIS OPTIMIZATION TECHNIQUES

4. A.0 Introduction

A fully optimized design is one, which has met the timing requirements and occupies the smallest area. The optimization can be done in two stages one at the code level, the other during synthesis. The optimization at the code level involves modifications to RTL code that is already been simulated and tested for its functionality. This level of modifications to the RTL code is generally avoided as sometimes it leads to inconsistencies between simulation results before and after modifications. However, there are certain standard model optimization techniques that might lead to a better synthesized
design.

4. A.1 Model Optimization

Model optimizations are important to a certain level, as the logic that is generated by the synthesis tool is sensitive to the RTL code that is provided as input. Different RTL codes generate different logic. Minor changes in the model might result in an increase or decrease in the number of synthesized gates and also change its timing characteristics. A logic optimizer reaches different endpoints for best area and best speed depending on the starting point provided by a netlist synthesized from the RTL code. The different starting points are obtained by rewriting the same HDL model using different constructs. Some of the optimizations, which can be used to modify the model for obtaining a better quality design, are listed below.

4.A.1.1 Resource Allocation

This method refers to the process of sharing a hardware resource under mutually exclusive conditions. Consider the following *if* statement.

```
if A = '1' then
E = B + C;
else
E = B + D;
end if;
```

The above code would generate two ALUs one for the addition of B+C and other for the addition B + D which are executed under mutually exclusive conditions. Therefore a single ALU can be shared for both the additions. The hardware synthesized for the above code is given below in Figure 4A.a.

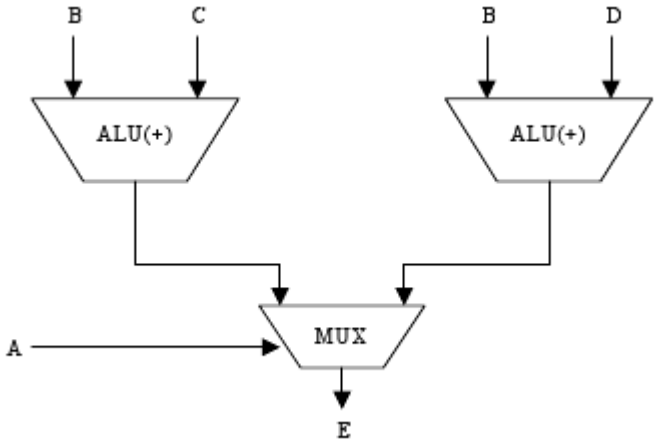


Figure 4A.a Without resource allocation.

The above code is rewritten with only on addition operator being employed. The hardware synthesized is given in Figure 4A.b.

```
if A = '1' then
temp := C; // A temporary variable introduced.
else
temp := D;
end if;
E = B + temp;
```

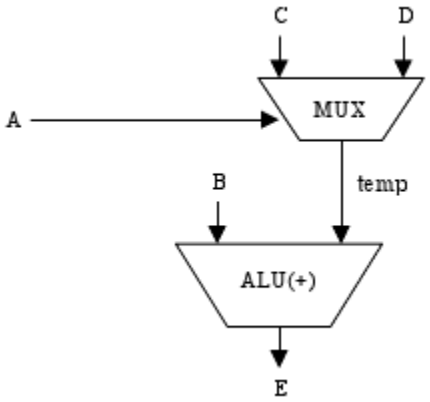


Figure 4A.b. With resource allocation.

It is clear from the figure that one ALU has been removed with one ALU being shared for both the addition operations. However a multiplexer is induced at the inputs of the ALU that contributes to the path delay. Earlier the timing path of the select signal goes through the multiplexer alone, but after resource sharing it goes through the multiplexer

and the ALU datapath, increasing its path delay. However due to resource sharing the area of the design has decreased. This is therefore a trade-off that the designer may have to make. If the design is timing-critical it would be better if no resource sharing is performed.

Common sub-expressions and Common factoring
It is often useful to identify common subexpressions and to reuse the computed values wherever possible. A simple example is given below.

```
B := R1 + R2;
…..
C <= R3 – (R1 + R2);
```

Here the subexpression R1 + R2 in the signal assignment for C can be replaced by B as given below. This might generate only one adder for the computation instead of two.

```
C <= R3 – B;
```

Common factoring is the extraction of common sub expressions in mutually-exclusive branches of an *if* or *case* statement.

```
if (test)
A <= B & (C + D);
else
J <= (C + D) | T;
end if;
```

In the above code the common factor C + D can be place out of the *if* statement, which might result in the tool generating only one adder instead of two as in the above case.

```
temp := C + D; // A temporary variable introduced.
if (test)
A <= B & temp;
else
J <= temp | T;
end if;
```

Such minor changes if made by the designer can cause the tool to synthesize better logic and also enable it to concentrate on optimizing more critical areas.

Moving Code

In certain cases an expression might be placed, within a for/while loop statement, whose value would not change through every iteration of the loop. Typically a synthesis tool handles the a for/while loop statement by unrolling it the specified number of times. In such cases redundant code might be generated for that particular expression causing additional logic to be synthesized. This could be avoided if the expression is moved outside the loop, thus optimizing the design. Such optimizations performed at a higher

level, that is, within the model, would help the optimizer to concentrate on more critical pieces of the code. An example is given below.

```
C := A + B;
……
for c in range 0 to 5 loop
……
T := C – 6;

// Assumption : C is not assigned a new value within the loop, thus the above expression would remain constant on every iteration of the loop.
……
end loop;
```

The above code would generate six subtracters for the expression when only one is necessary. Thus by modifying the code as given below we could avoid the generation of unnecessary logic.

```
C := A + B;
……
temp := C – 6; // A temporary variable is introduced
for c in range 0 to 5 loop
……
T := temp;
// Assumption : C is not assigned a new value within the loop, thus the above expression would remain constant on every iteration of the loop.
……
end loop;
```

Constant folding and Dead code elimination

The are possibilities where the designer might leave certain expressions which are constant in value. This can be avoided by computing the expressions instead of the implementing the logic and then allowing the logic optimizer to eliminate the additional logic.

Ex:

```
C := 4;
….
Y = 2 * C;
```

Computing the value of Y as 8 and assigning it directly within your code can avoid the above unnecessary code. This method is called constant folding. The other optimization, dead code elimination refers to those sections of code, which are never executed.

Ex.

```
A := 2;
```

```
B := 4;
if(A > B) then
……
end if;
```

The above *if* statement would never be executed and thus should be eliminated from the code. The logic optimizer performs these optimizations by itself, but nevertheless if the designer optimizes the code accordingly the tool optimization time would be reduced resulting in faster tool running times.

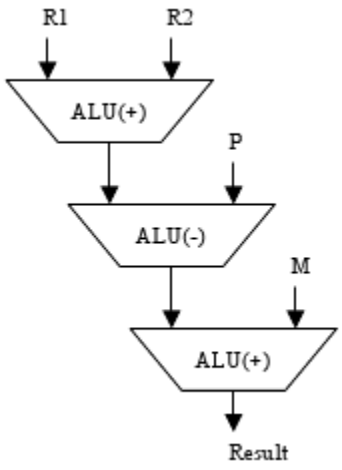
4.A.1.2 Flip-flop and Latch optimizations

Earlier in the RTL code section, it has been described how flip-flops and latches are inferred through the code by the synthesis tool. However there are only certain cases where the inference of the above two elements is necessary. The designer thus should try to eliminate all the unnecessary flip-flop and latch elements in the design. Placing only the clock sensitive signals under the edge sensitive statement can eliminate the unnecessary flip-flops. Similarly the unwanted latches can be avoided by specifying the values for the signals under all conditions of an *if/case* statement.

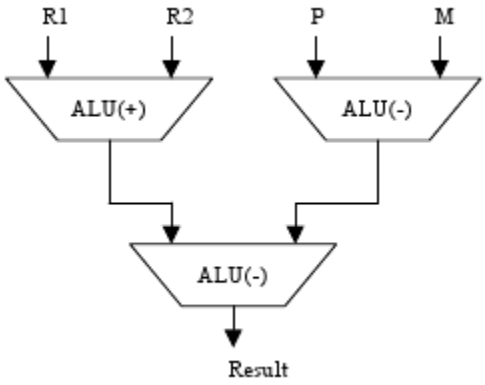
4.A.1.3 Using Parentheses

The usage of parentheses is critical to the design as the correct usage might result in better timing paths.

Ex.
Result <= R1 + R2 - P + M;
The hardware generated for the above code is as given below in Figure 4 (a).



If the expression has been written using parentheses as given below, the hardware synthesized would be as given in Figure 4 (b).
Result <= (R1 + R2) – (P - M);



It is clear that after using the parentheses the timing path for the datapath has been reduced as it does not need to go through one more ALU as in the earlier case.

4.A.1.4 Partitioning and structuring the design.

A design should always be structured and partitioned as it helps in reducing design complexity and also improves the synthesis run times since it smaller sub blocks synthesis synthesize faster. Good partitioning results in the synthesis of a good quality design. General recommendations for partitioning are given below.

- Keep related combinational logic in the same module
- Partition for design reuse.
- Separate modules according to their functionality.
- Separate structural logic from random logic.
- Limit a reasonable block size (perhaps a maximum of 10K gates per block).
- Partition the top level.
- Do not add glue-logic at the top level.
- Isolate state-machine from other logic.
- Avoid multiple clocks within a block.
- Isolate the block that is used for synchronizing the multiple clocks.

4.A.2 Optimization using Design Compiler

For the optimization of design, to achieve minimum area and maximum speed, a lot of experimentation and iterative synthesis is needed. The process of analyzing the design for speed and area to achieve the fastest logic with minimum area is termed – design space exploration.

For the sake of optimization, changing of HDL code may impact other blocks in the design or test benches. For this reason, changing the HDL code to help synthesis is less desirable and generally is avoided. It is now the designer's responsibility to minimize the area and meet the timing requirements through synthesis and optimization. The later

versions of DC, starting from DC98 have their compile flow different from previous versions. In the DC98 and later versions the timing is prioritized over area. Another difference is that DC98 performs compilation to reduce "total negative slack" instead of "worst negative slack". This ability of DC98 produces better timing results but has some impact on area. Also DC98 requires designers to specify area constraints explicitly as opposed to the previous versions that automatically handled area minimization. Generally some area cleanup is performed but better results are obtained when constraints are specified.

The DC has three different compilation strategies. It is up to user discretion to choose the most suitable compilation strategy for a design.
a) Top-down hierarchical compile method.
b) Time-budget compile method.
c) Compile-characterize-write-script-recompile (CCWSR) method.

4.A.2.1 Top-down hierarchical Compile

Prior to the release of DC98 this method was used to synthesize small designs as this method was extremely memory intensive and took a lot of time for large designs. In this method the source is compiled by reading the entire design with constraints and attributes applied, only at the top level. DC98 provided Synopsys the capability to synthesize million gate designs by tackling much larger blocks (>100K) at a time. This approach is feasible for some designs depending on the design style (single clock etc.) and other factors. One may use this technique to synthesize larger blocks at a time by grouping the sub-blocks together and flattening them to improve timing.

Advantages
 Only top level constraints are needed.
 Better results due to optimization across entire design.

Disadvantages
 Long compile time.
 Incremental changes to the sub-blocks require complete re-synthesis.
 Does not perform well, if design contains multiple clocks or generated clocks.

Time-budgeting compile.
This process is best for designs properly partitioned designs with timing specifications defined for each sub-block. Due to specifying of timing requirements for each block, multiple synthesis scripts for individual blocks are produced. The synthesis is usually performed bottom-up i.e., starting at the lowest level and going up to the top most level. This method is useful for medium to very large designs and does not require large amounts memory.

Advantages
 Design easier to manage due to individual scripts.
 Incremental changes to sub-blocks do not require complete re-synthesis.

 Can be used for any style of design, e.g. multiple and generated clocks.
Disadvantages
 Difficult to keep track of multiple scripts.
 Critical paths seen at top level may not be critical at lower level.
 Incremental compilations may be needed for fixing DRC's.

Compile-Characterize-Write-Script-Recompile
This is an advanced synthesis approach, useful for medium to very large designs that do not have good inter-block specifications defined. It requires constraints to be applied at the top level of the design, with each sub-block compiled beforehand. The subblocks are then characterized using the top-level constraints. This in effect propagates the required timing information from the top-level to the sub-blocks. Performing a **write_script** on the characterized sub-blocks generates the constraint file for each subblock.

The constraint files are then used to re-compile each block of the design.
Advantages
 Less memory intensive.
 Good quality of results because of optimization between sub-blocks of the design.
 Produces individual scripts, which may be modified by the user.
Disadvantages
 The generated scripts are not easily readable.
 It is difficult to achieve convergence between blocks
 Lower block changes might need complete re-synthesis of entire design.

Resolving Multiple instances
Before proceeding for optimization, one needs to resolve multiple instances of the sub-block of your design. This is a necessary step as Dc does not permit compilation until multiple instances are resolved.

Ex: Lets say moduleA has been synthesized. Now moduleB that has two instantiations of moduleA as U1 and U2 is being compiled. The compilation will be stopped with an error message stating that moduleA is instantiated 2 times in moduleB. There are two methods of resolving this problem.
You can set a **don_touch** attribute on moduleA before synthesizing moduleB, or **uniquify** moduleB. **uniquify** a **dc_shell** command creates unique definitions of multiple instances. So it for the above case it generates moduleA-u1 and moduleA_u2 (in VHDL), corresponding to instance U1 and U2 respectively.

4.A.2.2 Optimization Techniques

Various optimization techniques that help in achieving better area and speed for your design are given below.

Compile the design
The compilation process maps the HDL code to actual gates specified from the target library. This is done through the **compile** command. The syntax is given below :

compile –map_effort <low | medium | high>
-incremental_mapping
-in_place
-no_design_rule | -only_design_rule
-scan
The compile command by default uses the –map_effort medium option. This usually produces the best results for most of the designs. It also default settings for the structuring and flattening attributes. The map_effort high should only be used, if target objectives are not met through default compile. The -incremental_mapping is used only after initial compile as it works only at gate-level. It is used to improve timing of the logic.

Flattening and structuring
Flattening implies reducing the logic of a design to a 2-level AND/OR representation. This approach is used to optimize the design by removing all intermediate variables and parenthesis. This option is set to "false" by default. The optimization is performed in two stages. The first stage involves the flattening and structuring and the second stage involves mapping of the resulting design to actual gates, using mapping optimization techniques.

Flattening
Flattening reduces the design logic in to a two level, sum-of-products of form, with few logic levels between the input and output. This results in faster logic. It is recommended for unstructured designs with random logic. The flattened design then can be structured before final mapping optimization to reduce area. This is important as flattening has significant impact on area of the design. In general one should compile the design using default settings (flatten and structure are set as false). If timing objectives are not met flattening and structuring should be employed. It the design is still failing goals then just flatten the design without structuring it. The command for flattening is given below

set_flatten <true | false>
-design <list of designs>
-effort <low | medium | high>
-phase <true | false>
The –phase option if set to true enables the DC to compare the logic produced by inverting the equation versus the non-inverted form of the equation. Structuring The default setting for this is "true". This method adds intermediate variables that can be factored out. This enables sharing of logic that in turn results in reduction of area.
For ex.

Before structuring After structuring
$P = ax + ay + c$ $P = aI + c$
$Q = x + y + z$ $Q = I + z$
$I = x + y$

The shared logic generated might effect the total delay of the logic. Thus one should be careful enough to specify realistic timing constraints, in addition to using default settings. Structuring can be set for timing(default) or Boolean optimization. The latter helps in reducing area, but has a greater impact on timing. Thus circuits that are timing sensitive should not be structured for Boolean optimization. Good examples for Boolean optimization are random logic structures and finite state machines. The command for structuring is given below.

set_structure <true | false>
-design <list of designs>
-boolean <low | medium | high>
-timing <true | false>

If the design is not timing critical and you want to minimize for area only, then set the area constraints (**set_max_area** 0) and perform Boolean optimization. For all other case structure with respect to timing only.

Removing hierarchy
DC by default maintains the original hierarchy that is given in the RTL code. The hierarchy is a logic boundary that prevents DC from optimizing across this boundary. Unnecessary hierarchy leads to cumbersome designs and synthesis scripts and also limits the DC optimization within that boundary, without optimizing across hierarchy. To allow DC to optimize across hierarchy one can use the following commands.

dc_shell> **current_design** <design name>
dc_shell> **ungroup –flatten –all**

This allows the DC to optimize the logic separated by boundaries as one logic resulting in better timing and an optimal solution.

Optimizing for Area

DC by default tries to optimize for timing. Designs that are not timing critical but area intensive can be optimized for area. This can be done by initially compiling the design with specification of area requirements, but no timing constraints. In addition, by using the **don_touch** attribute on the high-drive strength gates that are larger in size, used by default to improve timing, one can eliminate them, thus reducing the area considerably.
Once the design is mapped to gates, the timing and area constraints should again be specified (normal synthesis) and the design re-compiled incrementally. The incremental compile ensures that DC maintains the previous structure and does not bloat the logic unnecessarily. The following points can be kept in mind for further area optimization:

- Bind all combinational logic as much as possible. If combinational logic were spreadover different blocks of the design the optimization of the logic would not be perfect resulting in large areas. So better partitioning of the design with

combinational logic not spread out among different blocks would result in better area.

- At the top level avoid any kind of glue logic. It is better to incorporate glue logic in one of the sub-components thus letting the tool to optimize the logic better.

4. A.3 Timing issues

There are two kind of timing issues that are important in a design- setup and hold timing violations.

Setup Time: It indicates the time before the clock edge during which the data should be valid i.e. it should be stable during this period and should not change. Any change during this period would trigger a setup timing violation. Figure 4A.b illustrates an example with setup time equal to 2 ns. This means that signal DATA must be valid 2 ns before the clock edge; i.e. it should not change during this 2ns period before the clock edge.

Hold Time: It indicates the time after the clock edge during which the data should be held valid i.e. it should not change but remain stable. Any change during this period would trigger a hold timing violation. Figure 4A.b illustrates an example with hold time equal to 1 ns. This means that signal DATA must be held valid 1 ns after the clock edge; i.e. it should not change during the 1 ns period after the clock edge.

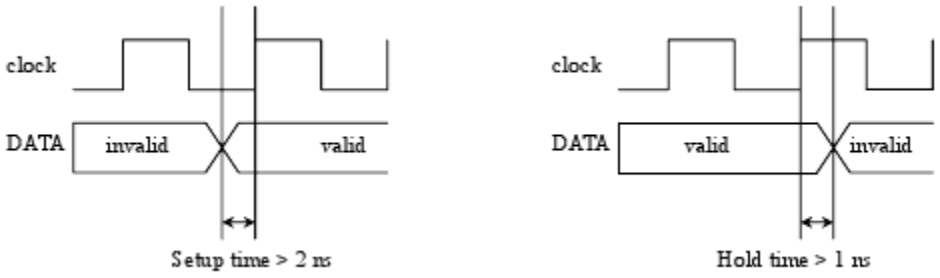


Figure 4A.b Timing diagram for setup and hold On DATA

The synthesis tool automatically runs its internal static timing analysis engine to check for setup and hold time violations for the paths, that have timing constraints set on them. It mostly uses the following two equations to check for the violations.

Tprop + Tdelay < Tclock - Tsetup (1)
Tdelay + Tprop > Thold (2)

Here Tprop is the propagation delay from input clock to output of the device in question (mostly a flip-flop); Tdelay is the propagation delay across the combinational logic through which the input arrives; Tsetup is the setup time requirement of the device;

Tclock is clock period;
Thold the hold time requirement of the device.

So if the propagation delay across the combinational logic, Tdelay is such that the equation (1) fails i.e. Tprop + Tdelay is more than Tclock – Tsetup then a setup timing violation is reported. Similarly if Tdelay + Tprop is greater than Thold then a hold timing violation is reported. In the case of the setup violation the input data arrives late due to large Tdelay across the combinational logic and thus is not valid/unstable during the setup time period triggering a violation. The flip-flop needs a certain time to read the input. During this period the data must remain stable and unchanged and any change would result in improper working of the device and thus a violation. In case of hold timing violation the data arrives faster than usual because the Tdelay + Tprop is not enough to delay the data enough. The flip-flop needs some time to store the data, during which the data should remain stable. Any change during this period would result in a violation. The data changes faster, without giving the flip-flop sufficient time to read it, thus triggering a violation.

4.A.3.1 HOW TO FIX TIMING VIOLATIONS

When the synthesis tool reports timing violations the designer needs to fix them. There are three options for the designer to fix these violations.

1) **Optimization using synthesis tool:** this is the easiest of all the other options. Few of the techniques have been discussed in the section Optimization Techniques above.

Few other techniques will be dealt with later in this section.
2) **Microarchitectural Tweaks:** This is a manual approach compared to the previous one. Here the designer should modify code to make microarchitectural changes that effect the timing of the design. Some of these techniques were discussed in the section Optimization Techniques and few new ones would be dealt with in this section.
3) **Architectural changes:** This is the last option as the designer needs to change the whole architecture of the design under consideration and would take up a long time.

Optimization using synthesis tool
The tool can be used to tweak the design for improving performance. A designer for performance optimization can employ the following ways.
a) Compilation with a map_effort high option;
b) Group critical paths together and give them a weight factor;
c) Register balancing;
d) Choose a specific implementation for a module;
e) Balancing heavy loading.

Compilation with a map_effort high
The initial compilation of a design is done with map_effort as medium when employing design constraints. This usually gives the best results with flattening and structuring options. In case the desired results are not met i.e. the design generates some timing violations then the map_effort of high can be set. This usually takes a long time to run

and thus is not used as the first option. This compilation could improve design performance by about 10% .

Group critical paths and assign a weight factor
We can use the group_path command to group critical timing paths and set a weight factor on these critical paths. The weight factor indicates the effort the tool needs to spend to optimize these paths. Larger the weight factor the more the effort. This command allows the designer to prioritize the critical paths for optimization using the weight factor.

group_path –name <group_name> -from <starting_point> -to <ending_point> -weight <value>

Register balancing
This command is particularly useful with designs that are pipelined. The command reshuffles the logic from one pipeline stage to another. This allows extra logic to be moved away from overly constrained pipeline stages to less constrained ones with additional timing. The command is simply balance_registers.

Choose a specific implementation for a module
A synthesis tool infers high-level functional modules for operators like '+', '-', '*', etc.. . however depending upon the map_effort option set, the design compiler would choose the implementation for the functional module. For example the adder has the following kinds of implementation.

a) Ripple carry – rpl
b) Carry look ahead –cla
c) Fast carry look ahead –clf
d) Simulation model –sim

The implementation type sim is only for simulation. Implementation types rpl, cla, and clf are for synthesis; clf is the faster implementation followed by cla; the slowest being rpl. If compilation of map_effort low is set the designer can manually set the implementation using the set_implementation command. Otherwise the selection will not change from current choice. If the map_effort is set to medium the design compiler would automatically choose the appropriate implementation depending upon the optimization algorithm. A choice of medium map_effort is suitable for better optimization or even a manual setting can be used for better performance results.
Balancing heavy loading Designs generally have certain nets with heavy fanout generating a heavy load on a certain point. A large load would be difficult to drive by a single net. This leads to unnecessary delays and thus timing violations. The balance_buffers command comes in hand to solve such problems. this command would make the design compiler to create buffer trees to drive the large fanout and thus balance the heavy load.

Microarchitectural Tweaks

The design can be modified for both setup timing violations as well as hold timing violations. Lets deal with setup timing violations. When a design with setup violations cannot be fixed with tool optimizations the code or microarchitectural implementation changes should be employed.
The following methods can be used for this purpose.

a) Logic duplication to generate independent paths
b) Balancing of logic between flip-flops
c) Priority decoding versus multiplex decoding
Logic duplication to generate independent paths

Consider the figure 4A.c Assuming a critical path exists from A to Q2, logic optimization on combinational logic X, Y, and Z would be difficult because X is shared with Y and Z. We can duplicate the logic X as shown in figure 4A.d. In this case Q1 and Q2 have independent paths and the path for Q2 can be optimized in a better fashion by the tool to ensure better performance.

Figure 4A.c : Logic with Q2 critical path

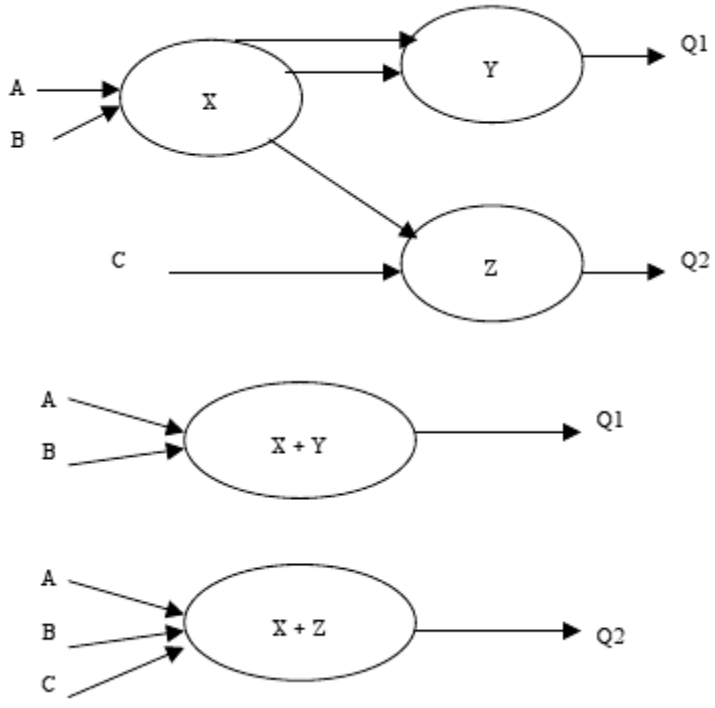


Figure 4A.d: Logic duplication allowing Q2 to be an independent path.

Logic duplication can also be used in cases where a module has one signal arriving late compared to other signals. The logic can be duplicated in front of the fast -arriving signals such that timing of all the signals is balanced. Figure 4A.e & 4A.f illustrate this fact quite well. The signal Q might generate a setup violation as it might be delayed due

to the late-arriving select signal of the multiplexer. The combinational logic present at the output could be put in front of the inputs (fast arriving). This would cause the delay due the combinational logic to be used appropriately to balance the timing of the inputs of the multiplexer and thus avoiding the setup violation for Q.

Figure 4A.e: Multiplexer with late arriving sel signal

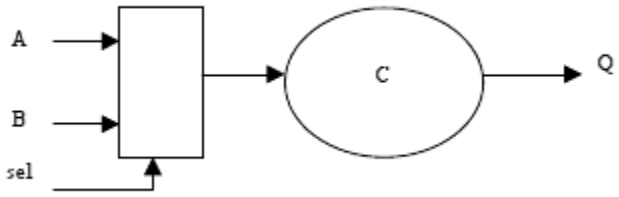
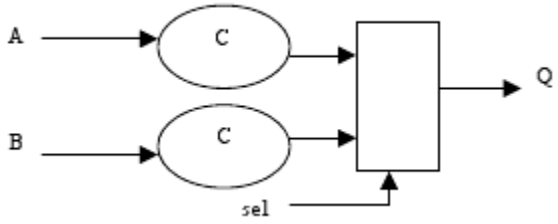


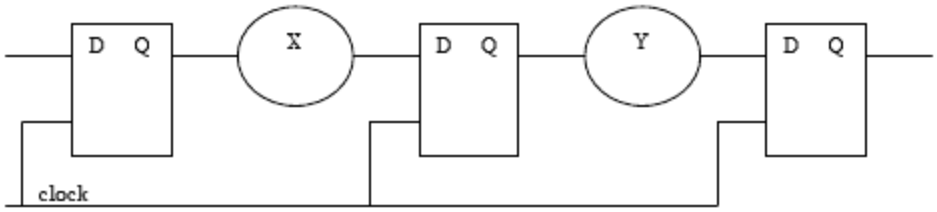
Figure 4A.f: Logic Duplication for balancing the timing between signals



Balancing of logic between flip-flops

This concept is similar to the balance_registers command we have come across in the Tool optimization section. The difference is that the designer does this at the code level. To fix setup violations in designs using pipeline stages the logic between each stage should be balanced. Consider a pipeline stage consisting of three flip-flops and two combinational logic modules in between each flip-flop. If the delay of the first logic module is such that it violates the setup time of the second flip-flop by a large margin and the delay of the second logic module is so less that the data on the third flip-flop is comfortably meeting the setup requirement. We can move part of the first logic module to the second logic module so that the setup time requirement of both the flip-flops is met. This would ensure better performance without any violations taking place. Figure 4.A.g illustrates the example.

Figure 4.A.g : Logic with pipeline stages



Priority encoding versus multiplex encoding

When a designer knows for sure that a particular input signal is arriving late then priority encoding would be a good bet. The signals arriving earlier could be given more priority and thus can be encoded before the late arriving signals.

Consider the boolean equation:
Q = A.B.C.D.E.F

It can be designed using five and gates with A, B at the first gate. The output of first gate is anded with C and output of the second gate with D and so on. This would ensure proper performance if signal F is most late arriving and A is the earliest to arrive. If propagation delay of each and gate were 1 ns this would ensure the output signal Q would be valid only 5 ns after A is valid or only 1 ns after signal H is valid. Multiplex decoding is useful if all the input signals arrive at the same time. This would ensure that the output would be valid at a faster rate. Thus multiplex decoding is faster than priority decoding if all input signals arrive at the same time. In this case for the boolean equation above the each of the two inputs would be anded parallely in the form of A.B, C.D and E.F each these outputs would then be anded again to get the final output. This would ensure Q to be valid in about 2 ns after A is valid.

Fixing Hold time violations
Hold time violations occur when signals arrive to fast causing them to change before they are read in by the devices. The best method to fix paths with hold time violations is to add buffers in those paths. The buffers generate additional delay slowing the path considerably. One has to careful while fixing hold time violations. Too many buffers would slow down the signal a lot and might result in setup violations which are a problem again.

4A.4 Verilog Synthesizable Constructs

Since it is very difficult fot the synthesis tool to find hardware with exact delays, all absolute and relative timing declarations are ignored by the tools. Also, all signals are assumed to be of maximum strength (strength 7). Boolean operations on x and z are not permitted. The constructs are classified as

- Fully supported constructs- Constructs that are supported as defined in the Verilog Language Reference Manual.
- Partially supported constructs- Constructs supported with restrictions on them
- Ignore constructs - constructs which are ignored by the synthesis tool
- Unsupported constructs- constructs which if used, may cause the synthesis tool to not accept the Verilog input or may cause different results between synthesis and simulation.

Fully supported constructs:
<module instantiation, with named and positional notations>

<integer data types, with all bases>

<identifiers>

<subranges and slices on right hand side of assignment>

<continuous assignment>

>>,<<,?:,{}
assign (procedural and declarative), begin, end, case, casex, casez, endcase
default
disable
function, endfunction
if, else, else if
input, output, inout
wire, wand, wor, tri
integer, reg
macromodule, module
parameter
supply0, supply1
task, endtask

Partially Supported Constructs

Construct	Constraints
*,/,%	when both operands constants or second operand is a power of 2
Always	only edge triggered events
For	bounded by static variables: only ise + or - to index

posedge, negedge	only with always @
primitive, endprimitive, table, endtable	Combinational and edge sensitive user defined primitives are often supported.
<=	limitations on usage with blocking statement
and,nand,or,nor,xor,xnor,buf,not,,bufif0,bufif1,notif0,notif1	Gate types supported without X or Z constructs
!, &&, ||, ~, &, |, ^, ^~, ~^, ~&, ~|, +, -, <, >, <=, >=, ++, !=	Operators supported without X or Z constructs

Ignored Constructs

<intra assignment timing controls>
<delay specifications>
scalared, vectored
small medium large
specify
time (some tools treat these as integers)
weak1, weak0, highz0, highz1, pull0, pull1
$keyword (some tools use these to set synthesis constraints)
wait (some tools support wait with a bounded condition).

Unsupported constructs

<assignment with variable used as bit select on LHS of assignment>
<global variables>
===, !==
cmos,nmos,rcmos,rnmos,pmos,rpmos
deassign
defparam
event
force
fork,join
forever,while
initial
pullup,pulldown
release
repeat
rtran,tran,tranif0
tranif1

rtranif0,rtranif1
table,endtable,primitive,endprimitive

5.0 DESIGN VISION

5.1 ANALYSIS OF GATE-LEVEL SYNTHESIZED NETLIST USING DESIGN VISION

Synopsys provides a GUI front-end to Design Compiler called Design Vision which we will use to analyze the synthesis results. You should avoid using the GUI to actually perform synthesis since we want to use scripts for this. To launch Design Vision and read in the synthesized design, move into the /project_dc/synth/ working directory and use the following commands. The command "design_vision-xg" will open up a GUI.

```
% design_vision-xg
design_vision-xg> read_file -format ddc output/gray_counter.ddc
```

You can browse your design with the hierarchical view. Right click on the gray_counter module and choose the Schematic View option [Figure 8.a], the tool will display a schematic of the synthesized logic corresponding to that module. Figure 8.b shows the schematic view for the gray counter module. You can see synthesized flip-flops in the schematic view.

Figure 5.a: Design Vision GUI

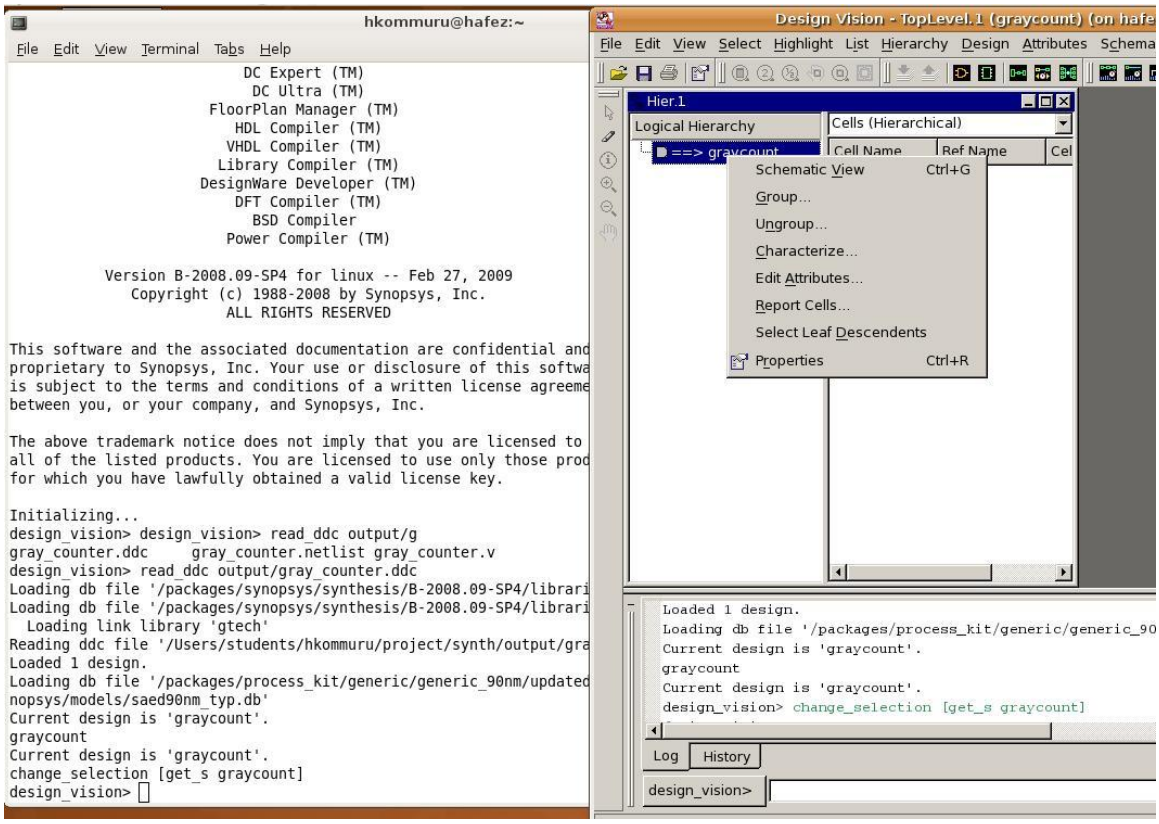
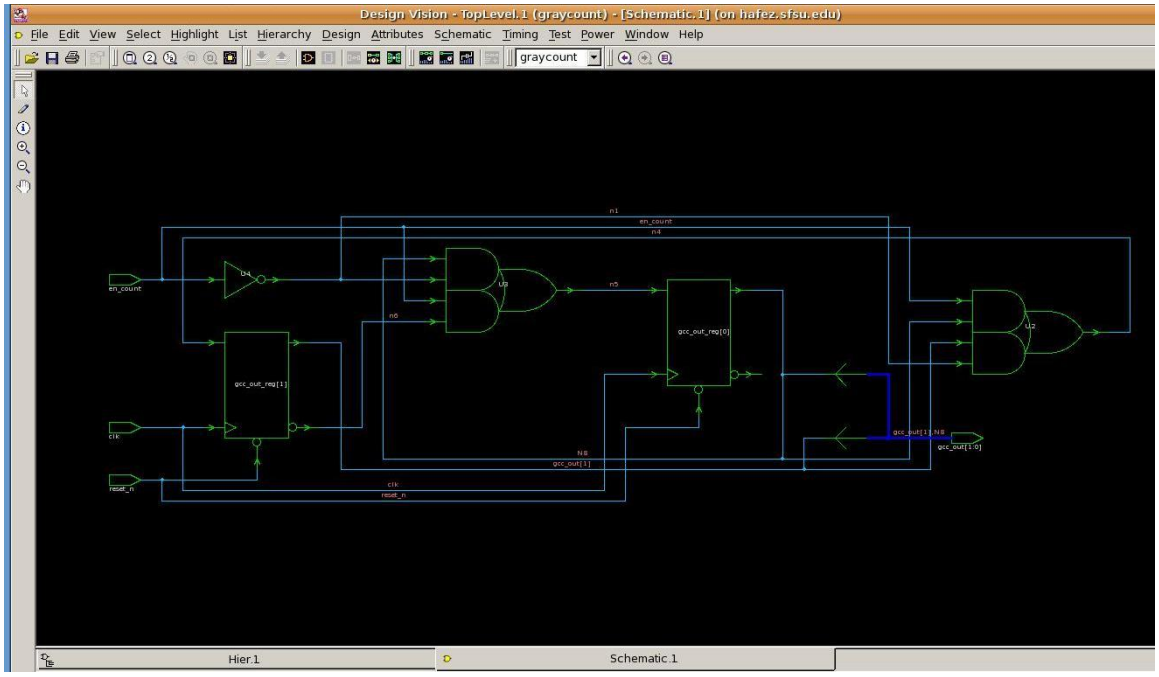
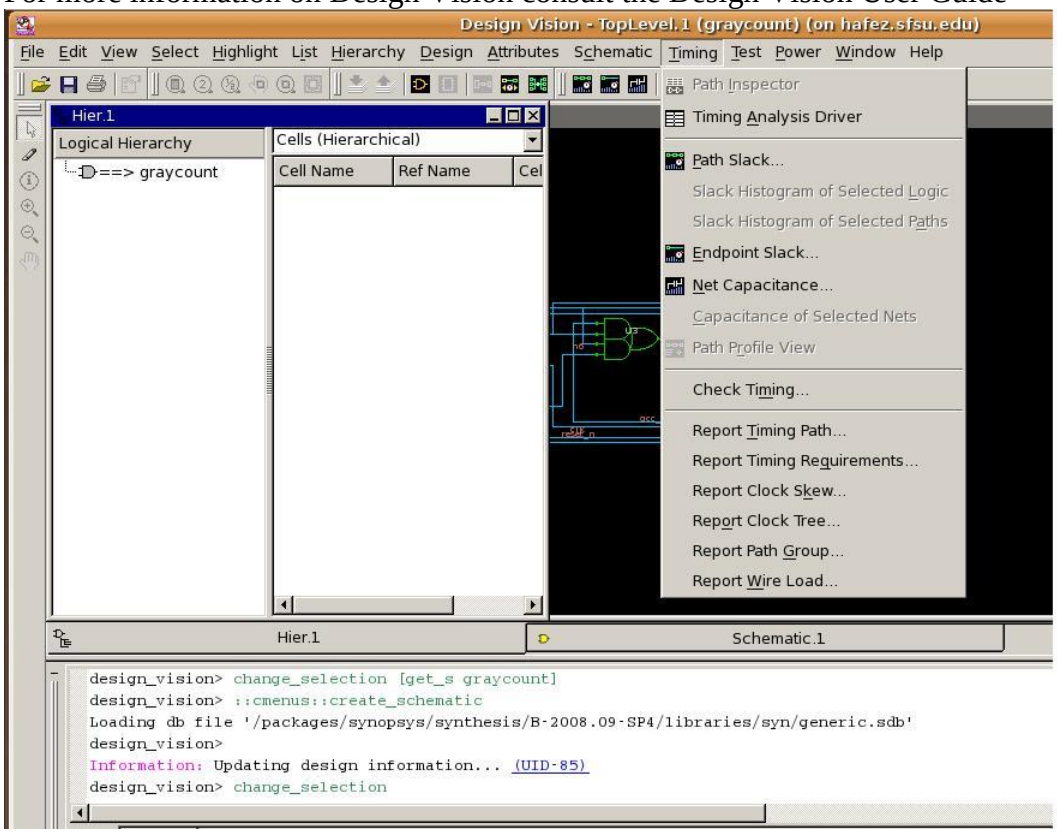


Figure 5.b: Schematic View of Synthesized Gray Counter



You can use Design Vision to examine various timing data. The Schematic ! Add Paths From/To menu option will bring up a dialog box which you can use to examine a specific path. The default options will produce a schematic of the critical path. The Timing ! Paths Slack menu option [Figure 8.c] will create a histogram of the worst case timing paths in your design. You can use this histogram to gain some intuition on how to approach a design which does not meet timing. If there area large number of paths which have a very large negative timing slack then a global solution is probably necessary, while if there are just one or two paths which are not making timing a more local approach may be sufficient. Figure 8.c and Figure 8.d shows an example of using these two features.

In the current gray_count design, there are no submodules. If there are submodules in the design, it is sometimes useful to examine the critical path through a single submodule. To do this, right click on the module in the hierarchy view and use the Characterize option. Check the timing, constraints, and connections boxes and click OK. Now choose the module from the drop down list box on the toolbar (called the Design List). Choosing Timing ! Report Timing will provide information on the critical path through that submodule given the constraints of the submodule within the overall design's context. For more information on Design Vision consult the Design Vision User Guide



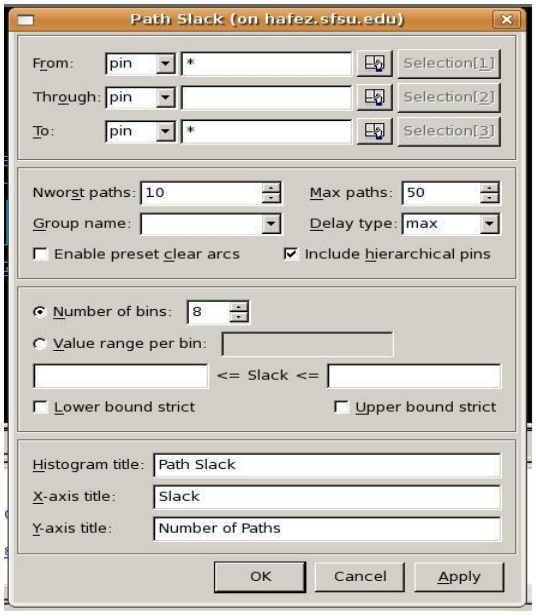
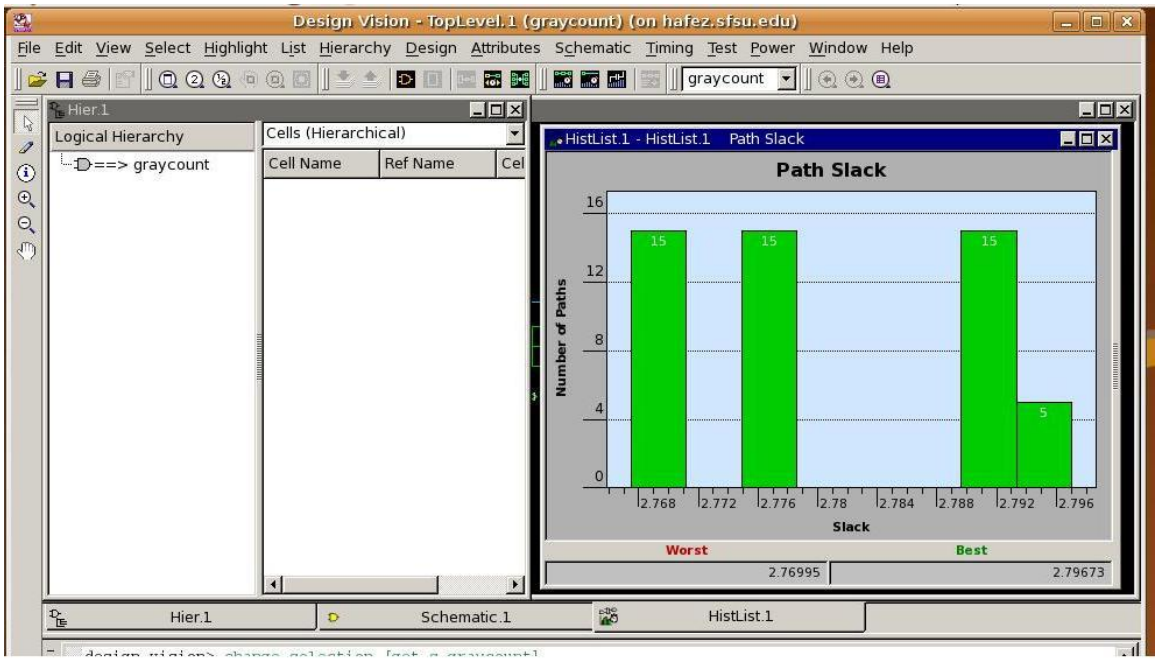


Figure 5.c Display Timing Path

Figure 5.d Histogram of Timing Paths



STATIC TIMING ANALYSIS

6.0 Introduction

Why is timing analysis important when designing a chip?

Timing is important because just designing the chip is not enough; we need to know how fast the chip is going to run, how fast the chip is going to interact with the other chips, how fast the input reaches the output etc…

Timing Analysis is a method of verifying the timing performance of a design by checking for all possible timing violations in all possible paths.

Why do we normally do Static Timing Analysis and not Dynamic Timing Analysis? What is the difference between them?

Timing Analysis can be done in both ways; static as well as dynamic. Dynamic Timing analysis requires a comprehensive set of input vectors to check the timing characteristics of the paths in the design. Basically it determines the full behavior of the circuit for a given set of input vectors. Dynamic simulation can verify the functionality of the design as well as timing requirements. For example if we have 100 inputs then we need to do 2 to the power of 100 simulations to complete the analysis. The amount of analysis is astronomical compared to static analysis.

Static Timing analysis checks every path in the design for timing violations without checking the functionality of the design. This way, one can do timing and functional analysis same time but separately. This is faster than dynamic timing simulation because there is no need to generate any kind of test vectors. That's why STA is the most popular way of doing timing analysis.

6.1 Timing Paths

The different kinds of paths when checking the timing of a design are as follows:

1. Input pin/port → Sequential Element
2. Sequential Element → Sequential Element
3. Sequential Element → Output pin/port
4. Input pin/port → Output pin/port

The static timing analysis tool performs the timing analysis in the following way:

1. STA Tool breaks the design down into a set of timing paths.
2. Calculates the propagation delay along each path.
3. Checks for timing violations (depending on the constraints e.g. clock) on the different paths and also at the input/output interface.

6.1.1 Delay Calculation of each timing path:

STA calculates the delay along each timing path by determining the Gate delay and Net delay.

Gate Delay: Amount of delay from the input to the output of a logic gate. It is calculated based on 2 parameters

a. Input Transition Time
b. Output Load Capacitance

Net Delay: Amount of delay from the output of a gate to the input of the next gate in a timing path. It depends on the following parameters

a. Parasitic Capacitance
b. Resistance of net

During STA, the tool calculates timing of the path by calculating:

1. Delay from input to output of the gate (Gate Delay).
2. Output Transition Time → (which in turn depends on Input Transition Time and Output Load Capacitance).

6.2 Timing Exceptions

Timing exceptions are nothing but constraints which don't follow the default when doing timing analysis. The different kinds of timing exceptions are:

1. **False path**: If any path does not affect the output and does not contribute to the delay of the circuit then that path is called false path.
2. **Multicycle Path**: Multicycle paths in a design are the paths that require more than one clock cycle. Therefore they require special Multicycle setup and hold-time calculations.
3. **Min/Max Path**: This path must match a delay constraint that matches a specific value. It is not an integer like the multicycle path. For example:
   Delay from one point to another max: 1.67ns; min: 1.87ns
4. **Disabled Timing Arcs**: The input to the output arc in a gate is disabled. For e.g.
   → 3 input and gate (a, b, c) and output (out). If you want you can disable the path from input 'a' to output 'out' using disable timing arc constraint.

6.3 Setting up Constraints to calculate timing*:*

To perform timing analysis we need to specify constraints. Few of the basic constraints one need to specify are:

1. **Clock Constraint**: Define the clock frequency you want your circuit to run at. This clock input controls all the timing in the chip/design.
2. **Setting Input Delay**: This delay is defined as the time taken by the signal to reach the input with respect to the clock.
3. **Setting Output Delay**: This delay is the 9delay incurred outside the particular block/pin/port with respect to the clock.
   Example: Assume Clock to be: 10ns
   If Output Delay is 5.4 ns then Input Delay would be: 10ns -5.4ns = 4.6ns

4. **Interface Timing**: It is the timing between different components/chips of a design.

6.4 Basic Timing Definitions:

- **Clock Latency**: Clock latency means delay between the clock source and the clock pin. This is called as source latency of the clock. Normally it specifies the skew between the clock generation point and the Clock pin.
- **Rise Time:** It is defined as the time it takes for a waveform to rise from 10% to 90% of its steady state value.
- **Fall time**: It is defined as the time it takes for a waveform to rise from 90% to 10% of its steady state value.
- **Clock-Q Delay**: It is the delay from rising edge of the clock to when Q (output) becomes available. It depends on
    - Input Clock transition
    - Output Load Capacitance
- **Clock Skew**: It is defined as the time difference between the clock path reference and the data path reference. The clock path reference is the delay from the main clock to the clock pin and data path reference is the delay from the main clock to the data pin of the same block. (Another way of putting it is the delay between the longest insertion delay and the smallest insertion delay.)
- **Metastability**: It is a condition caused when the logic level of a signal is in an indeterminate state.
- **Critical Path**: The clock speed is normally determined by the slowest path in the design. This is often called as 'Critical Path'.
- **Clock jitter**: It is the variation in clock edge timing between clock cycles. It is usually caused by noise.
- **Set-up Time**: It is defined as the time the data/signal has to stable before the clock edge.
- **Hold Time**: It is defined as the time the data/signal has to be stable after the clock edge.
- **Interconnect Delay**: This is delay caused by wires. Interconnect introduces three types of parasitic effects – capacitive, resistive, and inductive – all of which influence signal integrity and degrade the performance of the circuit.
- **Negative Setup time**: In certain cases, due to the excessive delay (example: caused by lot of inverters in the clock path) on the clock signal, the clock signal actually arrives later than the data signal. The actual clock edge you want your data to latch arrives later than the data signal. This is called negative set up time.
- **Negative Hold time**: It basically allows the data that was supposed to change in the next cycle, change before the present clock edge.
- **Negative Delay**: It is defined as the time taken by the 50% of output crossing to 50% of the input crossing.
- **Transition Time**: It is the time taken for the signal to go from one logic level to another logic level

- **Delay Time**: It is defined as the time taken by the 50% of input crossing to 50% of the output crossing. (50% of i/p transition level to 50% of output transition level)
- **Insertion Delay**: Delay from the clock source to that of the sequential pin.

6.5 Clock Tree Synthesis (CTS):

Clock Tree Synthesis is a process which makes sure that the clock gets distributed evenly to all sequential elements in a design. Also, if a net is a high fan out net, then we need to do load balancing. A high fan out net is a net which drives a large number of inputs. Load balancing is nothing but Clock tree Synthesis. During this process, depending on the clock skew, buffers are added to the different clock paths in the design. For example, let us consider two Flip-Flops; Launch FF and Capture FF. Launch FF is where the data is launched and Capture FF is the FF where data has to be captured. To improve setup time or hold time, the following need to be done.

a. More delays (buffers) are added on the launching side to have a better hold time.
b. More delays (buffers) are added to latching side to have a better Set up time.

→ The most efficient time to CTS is after Placement.

You can learn about CTS more detail in the Physical Design part of this tutorial.

Clock Network Delay: A set of buffers are added in between the source of the clock to the actual clock pin of the sequential element. This delay due to the addition of all these buffers is defined as the Clock Network Delay. [Clock Network Delay is added to clock period in Primetime]
Path Delay: When calculating path delay, the following has to be considered:
Clock Network Delay+ Clock-Q + (Sum of all the Gate delays and Net delays)
Global Clock skew: It is defined as the delay which is nothing but the difference between the Smallest and Longest Clock Network Delay.
Zero Skew: When the clock tree is designed such that the skew is zero, it is defined as zero skew.
Local Skew: It is defined as the skew between the launch and Capture flop. The worst skew is taken as Local Skew.
Useful Skew: When delays are added only to specific clock paths such that it improves set up time or hold time, is called useful skew.

What kind of model does the tool use to calculate the delay?
The tool uses a wire load model. It is nothing but a statistical model .It consists of a table which gives the capacitance and resistance of the net with respect to fan-out.

For more information please refer to the Primetime User Manual in the packages/synopsys/ directory.

6.6 PRIMETIME TUTORIAL EXAMPLE

6.6.1 Introduction

PrimeTime (PT) is a sign-off quality static timing analysis tool from Synopsys. Static timing analysis or STA is without a doubt the most important step in the design flow. It determines whether the design works at the required speed. PT analyzes the timing delays in the design and flags violation that must be corrected.

PT, similar to DC, provides a GUI interface along with the command-line interface. The GUI interface contains various windows that help analyze the design graphically. Although the GUI interface is a good starting point, most users quickly migrate to using the command-line interface. Therefore, I will focus solely on the command-line interface of PT.

PT is a stand-alone tool that is not integrated under the DC suite of tools. It is a separate tool, which works alongside DC. Both PT and DC have consistent commands, generate similar reports, and support common file formats. In addition PT can also generate timing assertions that DC can use for synthesis and optimization. PT's command-line interface is based on the industry standard language called Tcl. In contrast to DC's internal STA engine, PT is faster, takes up less memory, and has additional features.

6.6.2 Pre-Layout

After successful synthesis, the netlist obtained must be statically analyzed to check for timing violations. The timing violations may consist of either setup and/or hold-time violations. The design was synthesized with emphasis on maximizing the setup-time, therefore you may encounter very few setup-time violations, if any. However, the hold-time violations will generally occur at this stage. This is due to the data arriving too fast at the input of sequential cells with respect to the clock.

If the design is failing setup-time requirements, then you have no other option but to re-synthesize the design, targeting the violating path for further optimization. This may involve grouping the violating paths or over constraining the entire sub-block, which had violations. However, if the design is failing hold-time requirements, you may either fix these violations at the pre-layout level, or may postpone this step until after layout. Many designers prefer the latter approach for minor hold-time violations (also used here), since the pre-layout synthesis and timing analysis uses the statistical wire-load models and fixing the hold-time violations at the pre-layout level may result in setup-time violations for the same path, after layout. However, if the wire-load models truly reflect the post-routed delays, then it is prudent to fix the hold-time violations at this stage. In any case, it must be noted that gross hold-time violations should be fixed at the pre-layout level, in order to minimize the number of hold-time fixes, which may result after the layout.

6.6.2.1 PRE-LAYOUT CLOCK SPECIFICATION

In the pre-layout phase, the clock tree information is absent from the netlist. Therefore, it is necessary to estimate the post-route clock-tree delays upfront, during the pre-layout phase in order to perform adequate STA. In addition, the estimated clock transition should also be defined in order to prevent PT from calculating false delays (usually large) for the driven gates. The cause of large delays is usually attributed to the high fanout normally associated with the clock networks. The large fanout leads to slow input transition times computed for the clock driving the endpoint gates, which in turn results in PT computing unusually large delay values for the endpoint gates. To prevent this situation, it is recommended that a fixed clock transition value be specified at the source.
The following commands may be used to define the clock, during the prelayout phase of the design.

pt_shell> create_clock –period 20 –waveform [list 0 10] [list CLK]
pt_shell> set_clock_latency **2.5** [get_clocks CLK]
pt_shell> set_clock_transition **0.2** [get_clocks CLK]
pt_shell> set_clock_uncertainty 1.2 –setup [get_clocks CLK]
pt_shell> set_clock_uncertainty 0.5 –hold [get_clocks CLK]

The above commands specify the port CLK as type clock having a period of 20ns, the clock latency as 2.5ns, and a fixed clock transition value of 0.2ns. The clock latency value of 2.5ns signifies that the clock delay from the input port CLK to all the endpoints is fixed at 2.5ns. In addition, the 0.2ns value of the clock transition forces PT to use the 0.2ns value, instead of calculating its own. The clock skew is approximated with 1.2ns specified for the setup-time, and 0.5ns for the hold-time. Using this approach during pre-layout yields a realistic approximation to the post-layout clock network results.

6.6.3 STEPS FOR PRE-LAYOUT TIMING VALIDATION

0. The design example for the rest of this tutorial is a FIFO whose verilog code is available in asic_flow_setup/src/fifo/fifo.v . Please first run the DC synthesis on this file:

[hkommuru@hafez]$ source /packages/synopsys/setup/synopsys_setup.tcl
[hkommuru@hafez.sfsu.edu] $ cd
[hkommuru@hafez.sfsu.edu] $cd asic_flow_setup/synth_fifo
[hkommuru@hafez.sfsu.edu] $cd scripts
[hkommuru@hafez.sfsu.edu] $emacs dc_synth.tcl &
[hkommuru@hafez.sfsu.edu] $cd ..
[hkommuru@hafez.sfsu.edu] $ dc_shell-xg-t

and run the scripts in /asic_flow_setup/synth_fifo/scripts/dc_synth.tcl

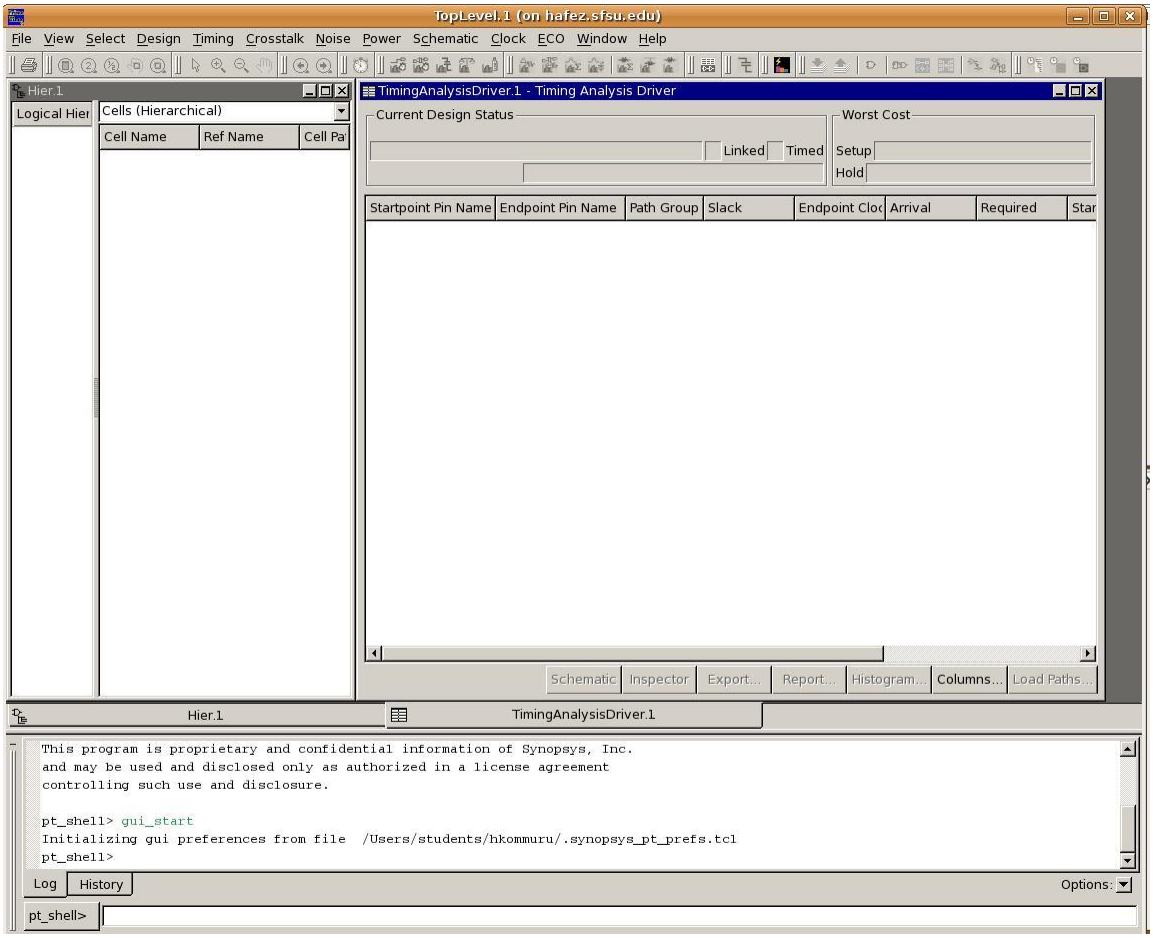
1. Please source "***synopsys_setup.tcl***" which sets all the environment variables necessary to run the Primetime . Please type csh at the unix prompt before you source the below script.

Please source the above file from the below location.

[hkommuru@hafez.sfsu.edu] $ csh
[hkommuru@hafez.sfsu.edu] $ cd
[hkommuru@hafez.sfsu.edu] $ cd /asic_flow_setup/pt
[hkommuru@hafez.sfsu.edu] $ source /packages/synopsys/setup/synopsys_setup.tcl

2. PT may be invoked in the command-line mode using the command pt_shell or in the GUI mode through the command primetime as shown below.

Command-line mode:
> pt_shell
GUI-mode:
> primetime



Before doing the next step , open the pre_layout_pt.tcl script and keep it ready which is at location /

[hkommuru@hafez.sfsu.edu]$ vi scripts/pre_layout_pt.tcl

3. Just like DC setup, you need to set the path to link_library and search_path

pt_shell > set link_library [list /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_typ.db /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max .db/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_min.db]
pt_shell > set target_library [list /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db]

4. Read the synthesized netlist

pt_shell> read_verilog ../synth_fifo/output/fifo.v

5. You need to set the top module name .

pt_shell > current_design FIFO

6. Read in the SDC from the synthesis

pt_shell> source ../synth_fifo/const/fifo.sdc

7. Now we can do the analysis of the design, as discussed in the beginning of this chapter, general, four types of analysis is performed on the design, as follows:

- From primary inputs to all flops in the design.
- From flop to flop.
- From flop to primary output of the design.
- From primary inputs to primary outputs of the design.

All four types of analysis can be accomplished by using the following commands:

pt_shell> report_timing -from [all_inputs] -max_paths 20 -to [all_registers -data_pins] > reports/timing.rpt

pt_shell> report_timing -from [all_register -clock_pins] -max_paths 20 -to [all_registers -data_pins] >> reports/timing.rpt

pt_shell> report_timing -from [all_registers -clock_pins] -max_paths 20 -to [all_outputs] >> reports/timing.rpt

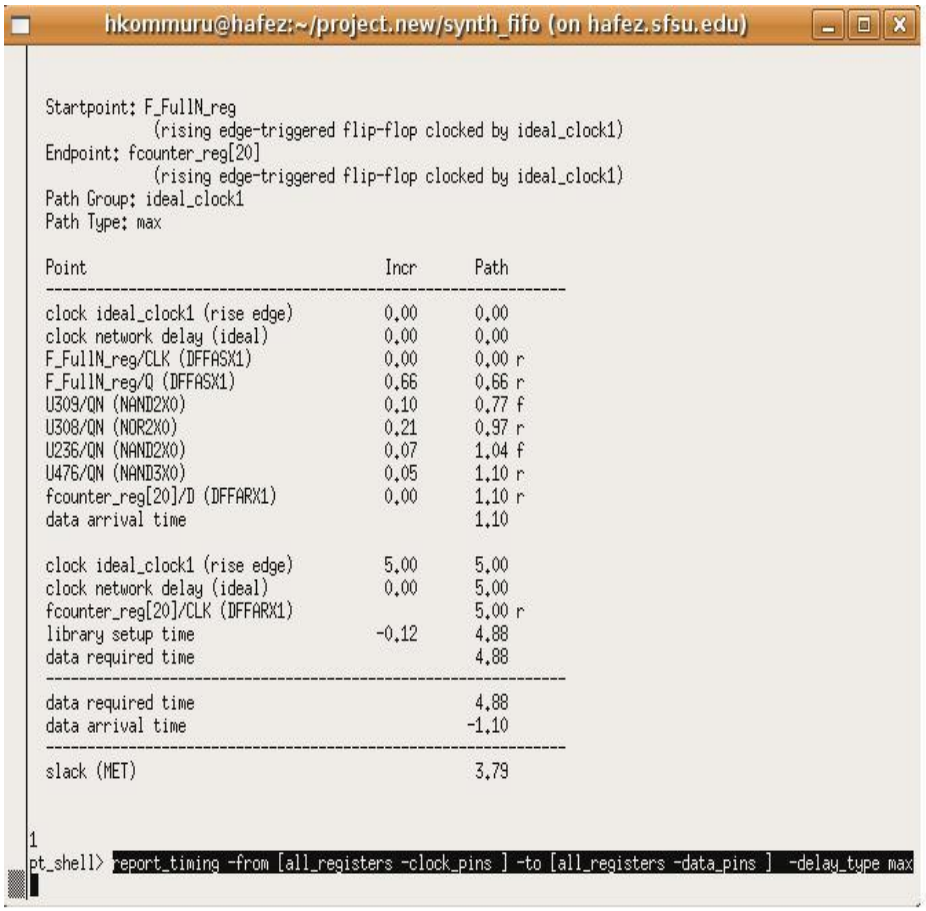
pt_shell> report_timing -from [all_inputs] -to [all_outputs] -max_paths 20 >> reports/timing.rpt

Please note that the above command write their reports to the file timing.rpt under the reports subdirectory. Please open this file to read the reports and check for any errors. Please notice that -max_paths 20 option above, gives the worst 20 paths in the design.

8. Reporting setup time and hold time. Primetime by default reports the setup time. You can report the setup or hold time by specifying the –delay_type option as shown in below figure.

pt_shell> report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins] -delay_type max >> reports/timing.rpt

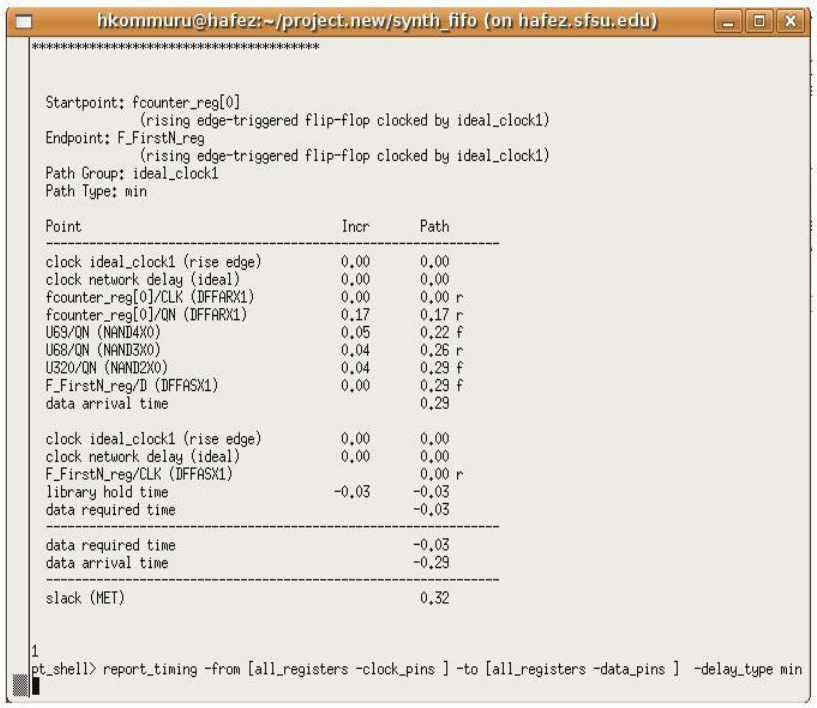
If you open and read the timing.rpt file, you will notice that the design meets the setup time.



9. Reporting hold time

pt_shell> report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins] -delay_type min >> reports/timing.rpt

If you open and read the timing.rpt file, you will notice that the design meets the setup time.



10. Reporting timing with capacitance and transition time at each level in the path

pt_shell > report_timing -transition_time -capacitance -nets -input_pins -from [all_registers -clock_pins] -to [all_registers -data_pins] > reports/timing.tran.cap.rpt

11. You can save your session and come back later if you chose to.

pt_shell > save_session output/fifo.session

Note: If the timing is not met, you need to go back to synthesis and redo to make sure the timing is clean before you proceed to the next step of the flow that is Physical Implementation.

```
hkommuru@hafez:~/projec
--------------------------------------------------------------------------------
slack (MET)                                                              3.04


Startpoint: FOutN (input port)
Endpoint: fcounter_reg[0]
          (rising edge-triggered flip-flop clocked by ideal_clock1)
Path Group: ideal_clock1
Path Type: max

Point                       Fanout    Cap      Trans      Incr      Path
--------------------------------------------------------------------------------
clock (input port clock) (rise edge)                     0.00      0.00
input external delay                                     2.00      2.00 r
FOutN (in)                                    0.00       0.00      2.00 r
FOutN (net)                     4     0.01
U339/IN (INVX0)                               0.00       0.00      2.00 r
U339/QN (INVX0)                               0.03       0.02      2.02 f
n582 (net)                      3     0.00
U315/IN2 (NAND2X0)                            0.03       0.00      2.02 f
U315/QN (NAND2X0)                             0.09       0.06      2.08 r
n116 (net)                      4     0.01
U159/IN (INVX0)                               0.09       0.00      2.08 r
U159/QN (INVX0)                               0.04       0.04      2.13 f
n583 (net)                      3     0.00
U246/IN2 (NAND2X0)                            0.04       0.00      2.13 f
U246/QN (NAND2X0)                             0.16       0.04      2.17 r
n351 (net)                      1     0.00
U146/IN2 (AND2X1)                             0.16       0.00      2.17 r
U146/Q (AND2X1)                               0.13       0.15      2.31 r
n244 (net)                     32     0.04
U168/IN2 (NAND2X0)                            0.13       0.00      2.31 r
U168/QN (NAND2X0)                             0.06       0.05      2.37 f
n443 (net)                      1     0.00
U497/IN3 (NAND3X0)                            0.06       0.00      2.37 f
U497/QN (NAND3X0)                             0.05       0.05      2.41 r
n175 (net)                      1     0.00
fcounter_reg[0]/D (DFFARX1)                   0.05       0.00      2.41 r
data arrival time                                                   2.41

clock ideal_clock1 (rise edge)                0.00       5.00      5.00
clock network delay (ideal)                              0.00      5.00
fcounter_reg[0]/CLK (DFFARX1)                                      5.00 r
library setup time                                      -0.12      4.88
data required time                                                 4.88
--------------------------------------------------------------------------------
data required time                                                 4.88
data arrival time                                                 -2.41
--------------------------------------------------------------------------------
slack (MET)                                                        2.47
```

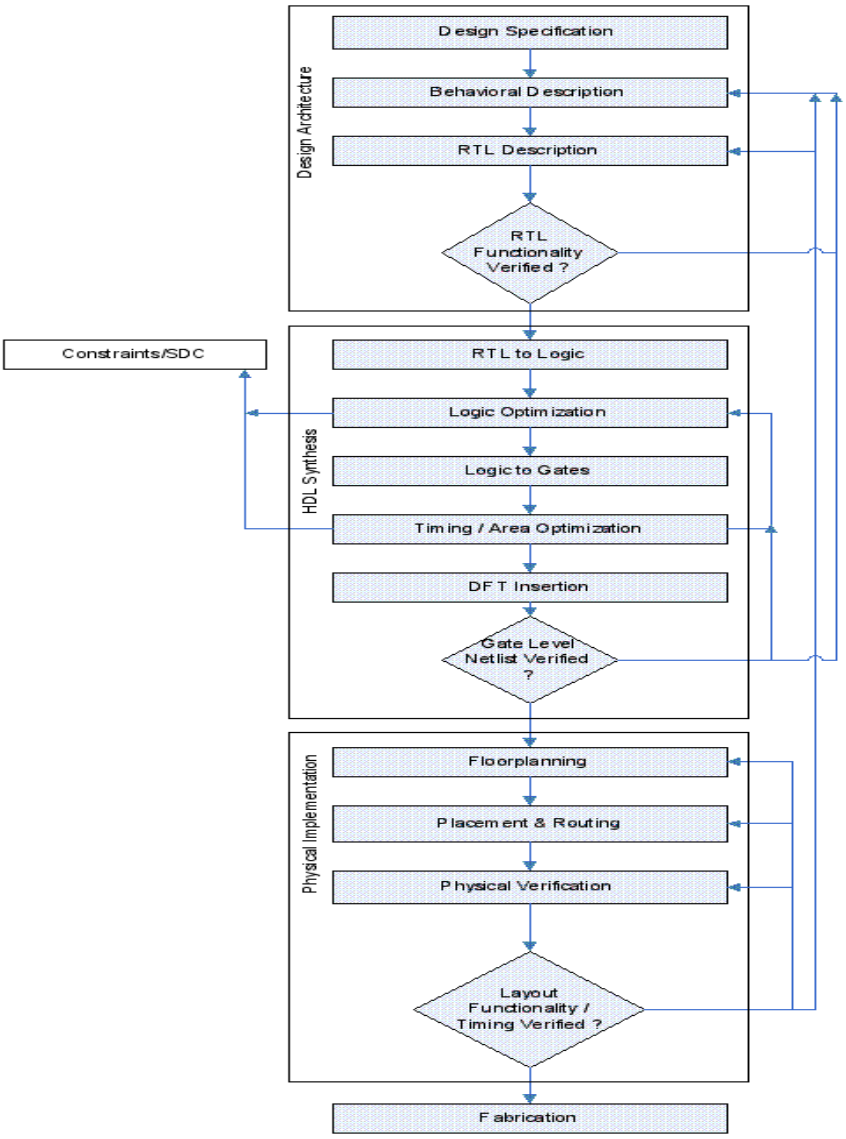
IC COMPILER TUTORIAL

8.0 Basics of Physical Implementation

8.1 Introduction

As you have seen in the beginning of the ASIC tutorial, after getting an optimized gate-level netlist, the next step is Physical implementation. Before we actually go into details of ICCompiler, which is the physical implementation tool from Synopsys, this chapter covers the necessary basic concepts needed to do physical implementation. Also, below you can see a more detailed flowchart of ASIC flow.

Figure 8.1.a : ASIC FLOW DIAGRAM



The Physical Implementation step in the ASIC flow consists of:

1. Floorplanning
2. Placement
3. Routing

This document will cover each one of the topics mentioned above one by one.

8.2 Floorplanning

At the floorplanning stage, we have a netlist which describes the design and the various blocks of the design and the interconnection between the different blocks. The netlist is the logical description of the ASIC and the floorplan is the physical description of the ASIC. Therefore, by doing floorplanning, we are mapping the logical description of the design to the physical description. The main objectives of floorplanning are to minimize

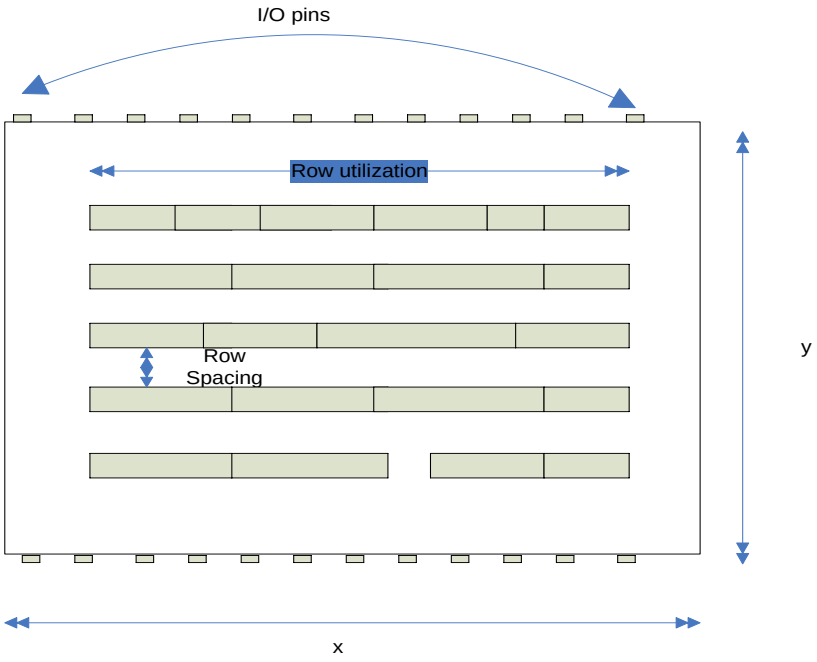
a. Area

b. Timing (delay)

During floorplanning, the following are done:

- The size of the chip is estimated.
- The various blocks in the design, are arranged on the chip.
- Pin Assignment is done.
- The I/O and Power Planning are done.
- The type of clock distribution is decided

Figure 8.2.a : Floorplan example



Floorplanning is a major step in the Physical Implementation process. The final timing, quality of the chip depends on the floorplan design. The three basic elements of chip are:

1. Standard Cells: The design is made up of standard cells.
2. I/O cells: These cells are used to help signals interact to and from the chip.
3. Macros (Memories): To store information using sequential elements takes up lot of area. A single flip flop could take up 15 to 20 transistors to store one bit. Therefore special memory elements are used which store the data efficiently and also do not occupy much space on the chip comparatively. These memory cells are called macros. Examples of memory cells include 6T SRAM (Static Dynamic Access Memory), DRAM (Dynamic Random Access Memory) etc.

The above figure shows a basic floorplan. The following is the basic floorplanning steps (and terminology):

1. **Aspect ratio (AR)**: It is defines as the ratio of the width and length of the chip. From the figure, we can say that aspect ratio is x/y. In essence, it is the shape of the rectangle used for placement of the cells, blocks. The aspect ratio should take into account the number of routing resources available. If there are more

horizontal layers, then the rectangle should be long and width should be small and vice versa if there are more vertical layers in the design.

   a. Normally, METAL1 is used up by the standard cells. Usually, odd numbered layers are horizontal layers and even numbered layers are vertical. So for a 5 layer design, AR = 2/2 = 1.
   b. Example2: For a 6 layer design, AR = 2/3 = 0.66

2. **Concept of Rows**: The standard cells in the design are placed in rows. All the rows have equal height and spacing between them. The width of the rows can vary. The standard cells in the rows get the power and ground connection from VDD and VSS rails which are placed on either side of the cell rows. Sometimes, the technology allows the rows to be flipped or abutted, so that they can share the power and ground rails.
3. **Core**: Core is defined as the inner block, which contains the standard cells and macros. There is another outer block which covers the inner block. The I/O pins are placed on the outer block.
4. **Power Planning**: Signals flow into and out off the chip, and for the chip to work, we need to supply power. A power ring is designed around the core. The power ring contains both the VDD and VSS rings. Once the ring is placed, a power mesh is designed such that the power reaches all the cells easily. The power mesh is nothing but horizontal and vertical lines on the chip. One needs to assign the metal layers through which you want the power to be routed. During power planning, the VDD and VSS rails also have to be defined.
5. **I/O Placement**: There are two types of I/O's.
   a. **Chip I/O:** The chip contains I/O pins. The chip consists of the core, which contains all the standard cells, blocks. The chip I/O placement consists of the placement of I/O pins and also the I/O pads. The placement of these I/O pads depends on the type of packaging also. (Refer document : Packaging)
   b. **Block I/O:** The core contains several blocks. Each block contains the Block I/O pins which communicate with other blocks, cells in the chip. This placement of pins can be optimized.
6. **Pin Placement**: Pin Placement is an important step in floorplanning. You may not know where to place the pins initially, but later on when you get a better idea, the pin placement can be done based on timing, congestion and utilization of the chip.
   a. **Pin Placement in Macros**: It uses up m3 layers most of the time, so the macro needs to be placed logically. The logical way is to put the macros near the boundary. If there is no connectivity between the macro pins and the boundary, then move it to another location.
7. **Concept of Utilization**: Utilization is defined as the percentage of the area that has been utilized in the chip. In the initial stages of the floorplan design, if the size of the chip is unknown, then the starting point of the floorplan design is utilization. There are three different kinds of utilizations.
   a. **Chip Level utilization:** It is the ratio of the area of standard cells, macros and the pad cells with respect to area of chip.

$$\rightarrow \frac{\text{Area (Standard Cells)} + \text{Area (Macros)} + \text{Area (Pad Cells)}}{\text{Area (chip)}}$$

b. **Floorplan Utilization**: It is defined as the ratio of the area of standard cells, macros, and the pad cells to the area of the chip minus the area of the sub floorplan.

→ $\frac{\text{Area (Standard Cells) + Area (Macros) + Area (Pad Cells)}}{\text{Area (Chip) – Area (sub floorplan)}}$

c. **Cell Row Utilization**: It is defined as the ratio of the area of the standard cells to the area of the chip minus the area of the macros and area of blockages.

→ $\frac{\text{Area (Standard Cells)}}{\text{Area (Chip) - Area (Macro) – Area (Region Blockages)}}$

8. **Macro Placement**: As a part of floorplanning, initial placement of the macros in the core is performed. Depending on how the macros are placed, the tool places the standard cells in the core. If two macros are close together, it is advisable to put placement blockages in that area. This is done to prevent the tool from putting the standard cells in the small spaces between the macros, to avoid congestion. Few of the different kinds of placement blockages are:
    a. **Standard Cell Blockage**: The tool does not put any standard cells in the area specified by the standard cell blockage.
    b. **Non Buffer Blockage**: The tool can place only buffers in the area specified by the Non Buffer Blockage.
    c. **Blockages below power lines**: It is advisable to create blockages under power lines, so that they do not cause congestion problems later. After routing, if you see an area in the design with a lot of DRC violations, place small chunks of placement blockages to ease congestion.

→After Floorplanning is complete, check for DRC (Design Rule check) violations. Most of the pre-route violations are not removed by the tool. They have to be fixed manually.

I/O Cells in the Floorplan: The I/O cells are nothing but the cells which interact in between the blocks outside of the chip and to the internal blocks of the chip. In a floorplan these I/O cells are placed in between the inner ring (core) and the outer ring (chip boundary). These I/O cells are responsible for providing voltage to the cells in the core. For example: the voltage inside the chip for 90nm technology is about 1.2 Volts. The regulator supplies the voltage to the chip (Normally around 5.5V, 3.3V etc).

The next question which comes to mind is that why is the voltage higher than the voltage inside the chip?

The regulator is basically placed on the board. It supplies voltage to different other chips on board. There is lot of resistances and capacitances present on the board. Due to this, the voltage needs to be higher. If the voltage outside is what actually the chip need inside, then the standard cells inside of the chip get less voltage than they actually need and the chip may not run at all.

So now the next question is how the chips can communicate between different voltages?

The answer lies in the I/O cells. These I/O cells are nothing but Level Shifters. Level Shifters are nothing but which convert the voltage from one level to another The Input I/O cells reduce the voltage coming form the outside to that of the voltage needed inside

the chip and output I/O cells increase the voltage which is needed outside of the chip. The I/O cells acts like a buffer as well as a level shifter.

8.3 Concept of Flattened Verilog Netlist

Most of the time, the verilog netlist is in the hierarchical form. By hierarchical I mean that the design is modeled on basis of hierarchy. The design is broken down into different sub modules. The sub modules could be divided further. This makes it easier for the logic designer to design the system. It is good to have a hierarchical netlist only until Physical Implementation. During placement and routing, it is better to have a flattened netlist. Flattening of the netlist implies that the various sub blocks of the model have basically opened up and there are no more sub blocks. There is just one top block. After you flatten the netlist you cannot differentiate between the various sub block, but the logical hierarchy of the whole design is maintained. The reason to do this is:
→ In a flat design flow, the placement and routing resources are always visible and available.
→Physical Design Engineers can perform routing optimization and can avoid congestion to achieve a good quality design optimization. If the conventional hierarchical flow is used, then it can lead to sub-optimal timing for critical paths traveling through the blocks and for critical nets routed around the blocks.
The following gives and example of a netlist in the hierarchical mode as well as the flattened netlist mode:

8.3.a Hierarchical Model:

```
module top (a , out1 )
input a;
output out1;

wire n1;

SUB1   U1 (.in (a), .out (n1))
SUB1   U2 (.in (n1), .out (out1))

endmodule

module SUB1 (  b , outb )
input b;
output outb;

wire n1, n2;

INVX1 V1 (.in (b), .out (n1))
INVX1 V2 (.in (n1), .out (n2))
```

```
INVX1 V3 (.in (n2), .out (outb))

endmodule
```

In verilog, the instance name of each module is unique. In the flattened netlist, the instance name would be the top level instance name/lower level instance name etc… Also the input and output ports of the sub modules also get lost. In the above example the input and output ports; a, out1, b and outb get lost.
The above hierarchical model, when converted to the flattened netlist, will look like this:

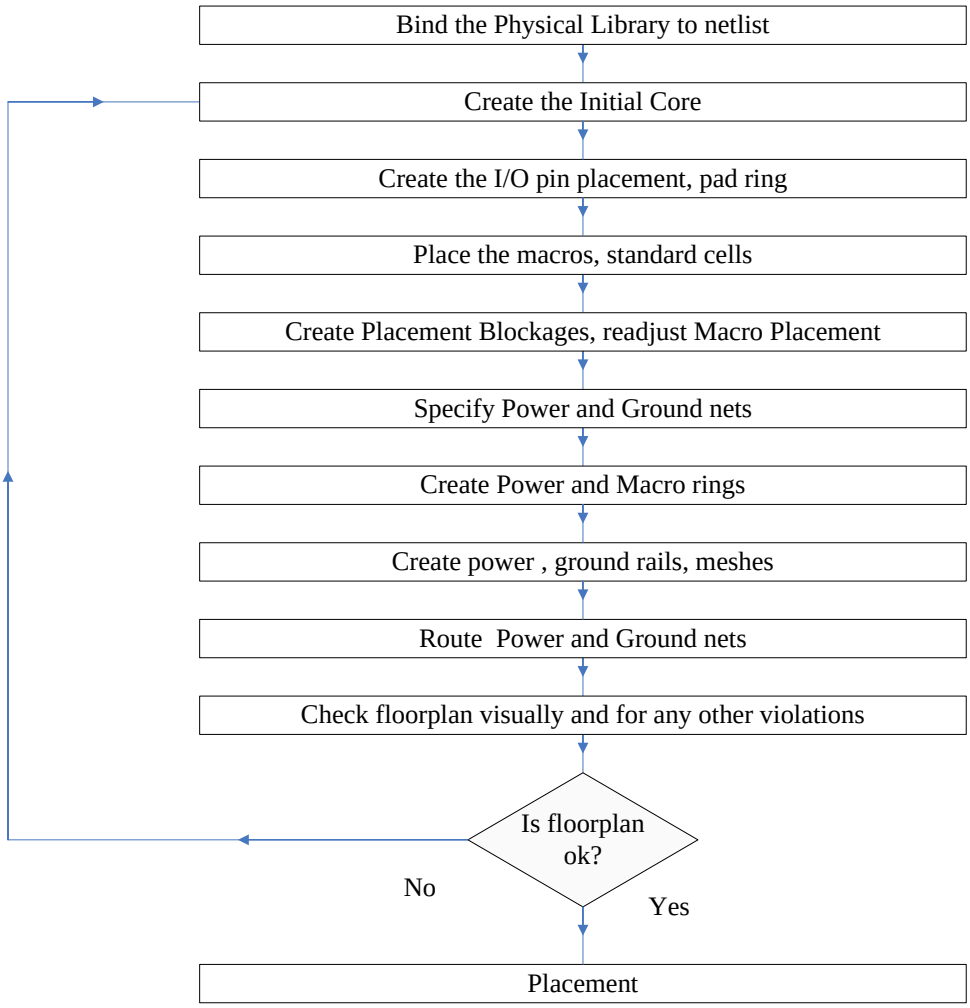
8.3.b Flattened Model:

```
module top ( in1, out1 )

input in1;
output out1;

wire topn1;

INVX1 U1/V1 (.in (in1), .out (V1/n1)
INVX1 U1/V2 (.in (V1/n1), .out (V2/n2   )
INVX1 U1/V3 (.in (V2/n2), .out (topn1   )
INVX1 U2/V1 (.in (n1), .out (V1/n1     )
INVX1 U2/V2 (.in (V1/n1), .out (V2/n2        )
INVX1 U2/V3 (.in (V2/n2), .out (    out1     )

endmodule
```

The following figure shows the summary of floorplanning:

Figure 8.c Floorplanning Flow Chart

Floorplanning Summary



8.4 Placement

Placement is a step in the Physical Implementation process where the standard cells location is defined to a particular position in a row. Space is set aside for interconnect to each logic/standard cell. After placement, we can see the accurate estimates of the capacitive loads of each standard cell must drive. The tool places these cells based on the algorithms which it uses internally. It is a process of placing the design cells in the floorplan in the most optimal way**.**

What does the Placement Algorithm want to optimize?

The main of the placement algorithm is

1. Making the chip as dense as possible (Area Constraint)
2. Minimize the total wire length (reduce the length for critical nets)

3. The number of horizontal/vertical wire segments crossing a line.

Constraints for doing the above are:

- The placement should be routable (no cell overlaps; no density overflow).
- Timing constraints are met

There are different algorithms to do placement. The most popular ones are as follows:

1. **Constructive algorithms:** This type of algorithm uses a set of rules to arrive at the optimized placement. Example: Cluster growth, min cut, etc.
2. **Iterative algorithms:** Intermediate placements are modified in an attempt to improve the cost function. It uses an already constructed placement initially and iterates on that to get a better placement.

Example: Force-directed method, etc

3. **Nondeterministic approaches:** simulated annealing, genetic algorithm, etc.

Min-Cut Algorithm

This is the most popular algorithm for placement. This method uses successive application of partitioning the block. It does the following steps:

1. Cuts the placement area into two pieces. This piece is called a bin. It counts the number of nets crossing the line. It optimizes the cost function. The cost function here would be number of net crossings. The lesser the cost function, the more optimal is the solution.
2. Swaps the logic cells between these bins to minimize the cost function.
3. Repeats the process from step 1, cutting smaller pieces until all the logic cells are placed and it finds the best placement option.

The cost function not only depends on the number of crossings but also a number of various other factors such as, distance of each net, congestion issues, signal integrity issues etc. The size of the bin can vary from a bin size equal to the base cell to a bin size that would hold several logic cells. We can start with a large bin size, to get a rough placement, and then reduce the bin size to get a final placement.

There are two steps in the Placement process.

1. Global Placement
2. Detail Placement

8.5 Routing

After the floorplanning and placement steps in the design, routing needs to be done. Routing is nothing but connecting the various blocks in the chip with one an other. Until now, the blocks were only just placed on the chip. Routing also is spilt into two steps

1. **Global routing**: It basically plans the overall connections between all the blocks and the nets. Its main aim is to minimize the total interconnect length, minimize the critical path delay. It determines the track assignments for each interconnect.

   →The chip is divided into small blocks. These small blocks are called routing bins. The size of the routing bin depends on the algorithm the tool uses. Each routing bin is also called a gcell. The size of this gcell depends on the tool.

Each gcell has a finite number of horizontal and vertical tracks. Global routing assigns nets to specific gcells but it does not define the specific tracks for each of them. The global router connects two different gcells from the centre point of each gcell.

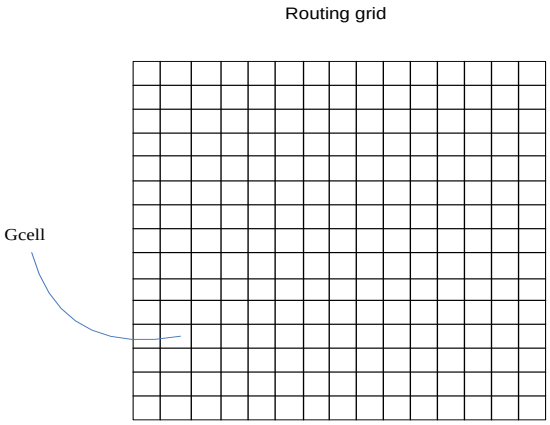
→ Track Assignment: The Global router keeps track of how many interconnections are going in each of direction. This is nothing but the routing demand. The number of routing layers that are available depend on the design and also, if the die size is more, the greater the routing tracks. Each routing layer has a minimum width spacing rule, and its own routing capacity.

For Example: For a 5 metal layer design, if Metal 1, 4, 5 are partially up for inter-cell connections, pin, VDD, VSS connections, the only layers which are routable 100% are Metal2 and Metal3. So if the routing demand goes over the routing supply, it causes **Congestion**. Congestion leads to DRC errors and slow runtime.

2. **Detailed Routing**: In this step, the actual connection between all the nets takes place. It creates the actual via and metal connections. The main objective of detailed routing is to minimize the total area, wire length, delay in the critical paths.

   →It specifies the specific tracks for the interconnection; each layer has its own routing grid, rules. During the final routing, the width, layer, and exact location of the interconnection are decided.

Figure 8.5.a : Routing grid



After detailed routing is complete, the exact length and the position of each interconnect for every net in the design is known. The parasitic capacitance, resistance can now is extracted to determine the actual delays in the design. The parasitic extraction is done by extraction tools. This information is back annotated and the timing of the design is now calculated using the actual delays by the Static Timing Analysis Tool.

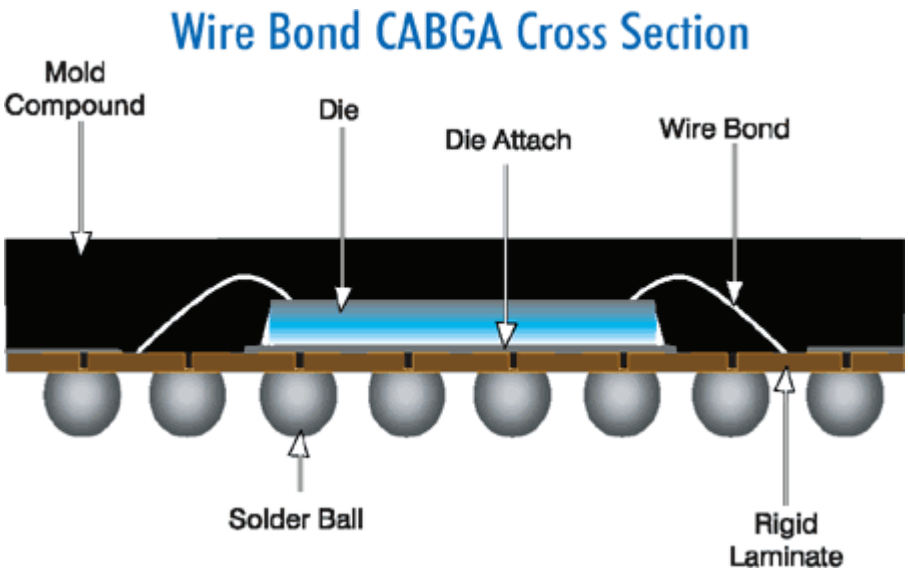
After timing is met and all other verification is performed such as LVS, etc, the design is sent to the foundry to manufacture the chip.

8.6 Packaging

Depending on the type of packaging of the chip, the I/O cells, pad cells are designed differently during the Physical Implementation. There are two types of Packaging style:

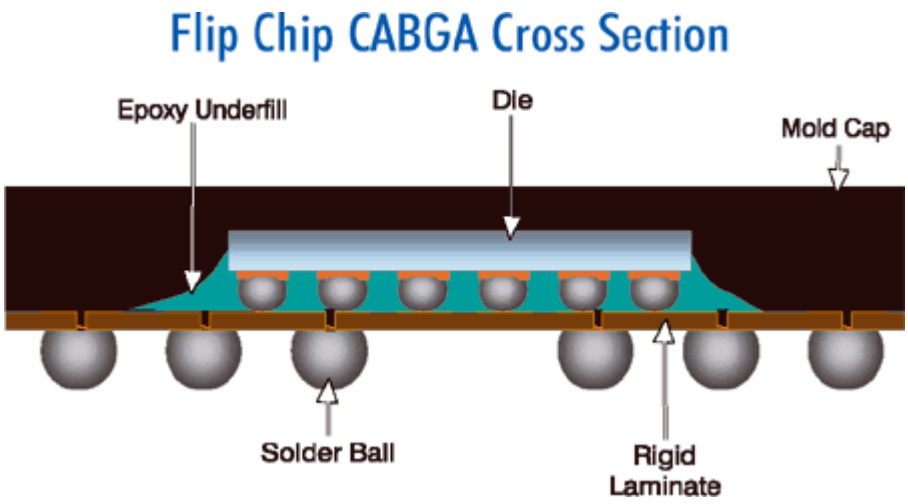
a. **Wire-bond**: The connections in this technique are real wires. The underside of the die is first fixed in the package cavity. A mixture of epoxy and a metal (aluminum, sliver or gold) is used to ensure a low electrical and thermal resistance between the die and the package. The wires are then bonded one at a time to the die and the package. Below is a illustration of Wire Bond packaging.

Figure 8.6.a : Wire Bond Example



b. **Flip-Chip**: Flip Chip describes the method of electrically connecting the die to the package carrier. This is a direct chip-attach technology, which accommodates dies that have several bond pads placed anywhere on the surfaces at the top. Solder balls are deposited on the die bond pads usually when they are still on the wafer, and at corresponding locations on the board substrate. The upside-down die (Flip-chip) is then aligned to the substrate. The advantage of this type of packaging is very short connections (low inductance) and high package density. The picture below is an illustration of flip-chip type of packaging.

Figure 8.6.b : Flip Chip Example



8.7 IC TUTORIAL EXAMPLE

8.7.1 Introduction

The physical design stage of the ASIC design flow is also known as the "place and route" stage. This is based upon the idea of physically placing the circuits, which form logic gates and represent a particular design, in such a way that the circuits can be fabricated.

This is a generic, high level description of the physical design (place/route) stage. Within the physical design stage, a complete flow is implemented as well. This flow will be described more specifically, and as stated before, several EDA companies provide software or CAD tools for this flow. Synopsys® software for the physical design process is called IC Compiler. The overall goal of this tool/software is to combine the inputs of a gate-level netlist, standard cell library, along with timing constraints to create and placed and routed layout. This layout can then be fabricated, tested, and implemented into the overall system that the chip was designed for.

The first of the main inputs into ICC are :

1. ***Gate-level netlist***, which can be in the form of Verilog or VHDL. This netlist is produced during logical synthesis, which takes place prior to the physical design stage(discussed in chapter 3).
2. The second of the main inputs into ICC is a ***standard cell library***. This is a collection of logic functions such as OR, AND, XOR, etc. The representation in the library is that of the physical shapes that will be fabricated.

This layout view or depiction of the logical function contains the drawn mask layers required to fabricate the design properly. However, the place and route tool does not require such level of detail during physical design. Only key information such as the location of metal and input/output pins for a particular logic function is needed. This representation used by ICC is considered to be the abstract version of the layout. Every desired logic function in the standard cell library will have both a layout and abstract view. Most standard cell libraries will also contain timing information about the function such as cell delay and input pin capacitance which is used to calculated output loads. This timing information comes from detailed parasitic analysis of the physical layout of each function at different process, voltage, and temperature points (PVT). This data is contained within the standard cell library and is in a format that is usable by ICC. This allows ICC to be able to perform static timing analysis during portions of the physical design process. It should be noted that the physical design engineer may or may not be involved in the creating of the standard cell library, including the layout, abstract, and timing information. However, the physical design engineer is required to understand what common information is contained within the libraries and how that information is used during physical design. Other common information about standard cell libraries is the fact that the height of each cell is constant among the different functions. This common height will aid in the placement process since they can now be linked together in rows across the design. This concept will be explained in detail during the placement stage of physical design.

3. The third of the main inputs into ICC are the ***design constraints***. These constraints are identical to those which were used during the front-end logic synthesis stage prior to physical design. These constraints are derived from the system specifications and implementation of the design being created. Common constraints among most designs include clock speeds for each clock in the design as well as any input or output delays associated with the input/output signals of the chip. These same constraints using during logic synthesis are used byICC so that timing will be considered during each stage of place and route. The constraints are specific for the given system specification of the design being implemented.

In the below IC compiler tutorial example, we will place & route the fifo design synthesized.

STEPS

1. As soon as you log into your engr account, at the command prompt, please type “csh “as shown below. This changes the type of shell from bash to c-shell. All the commands work ONLY in c-shell.

[hkommuru@hafez]$**csh**

2. Please copy the whole directory from the below location

[hkommuru@hafez]$ **cp –rf** /packages/synopsys/setup/asic_flow_setup ***./***

This ccreate directory structure as shown below. It will create a directory called **"asic_flow_setup ",** under which it creates the following directories namely

asic_flow_setup
src/ : for verilog code/source code
vcs/ : for vcs simulation for counter example
synth_graycounter/ : for synthesis of graycounter example
synth_fifo/ : for fifo synthesis
pnr_fifo/ : for Physical design of fifo design example
extraction/: for extraction
pt/: for primetime
verification/: final signoff check

The **"asic_flow_setup"** directory will contain all generated content including, VCS simulation, synthesized gate-level Verilog, and final layout. In this course we will always try to keep generated content from the tools separate from our source RTL. This keeps our project directories well organized, and helps prevent us from unintentionally modifying the source RTL. There are subdirectories in the project directory for each major step in the ASIC Flow tutorial. These subdirectories contain scripts and configuration files for running the tools required for that step in the tool flow. For this tutorial we will work exclusively in the ***vcs*** directory.

3. Please source "***synopsys_setup.tcl***" which sets all the environment variables necessary to run the VCS tool.
Please source them at unix prompt as shown below

[hkommuru@hafez]$ **source** /packages/synopsys/setup/synopsys_setup.tcl

Please Note : You have to do steps 1 and 3 above everytime you log in.

4. Go to the pnr directory .

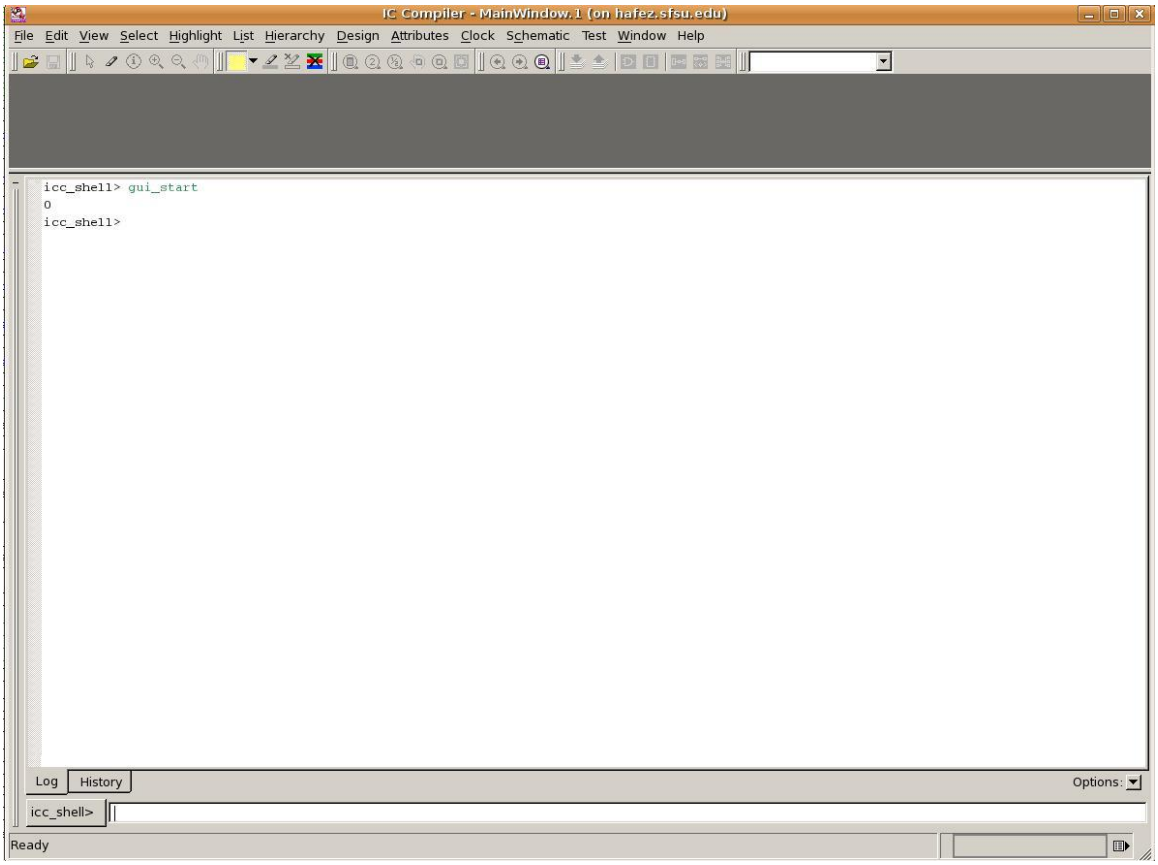
[hkommuru@hafez.sfsu.edu] $**cd** asic_flow_setup/pnr_fifo
[hkommuru@hafez.sfsu.edu] $**cd** scripts
[hkommuru@hafez.sfsu.edu] $**emacs** init_design_icc.tcl &
[hkommuru@hafez.sfsu.edu] $**cd ..**

At the unix prompt type "icc_shell ", it will open up the icc window.

[hkommuru@hafez.sfsu.edu] $**icc_shell**

Next, to open the gui, type "gui_start", it opens up gui window as shown in the next page .

icc_shell > **gui_start**



Before a design can be placed and routed within ICC, the environment for the design needs to be created. The goal of the design setup stage in the physical design flow is to prepare the design for floorplanning. The first step is to create a design library. Without a design library, the physical design process using will not work. This library contains all of the logical and physical data that will need. Therefore the design library is also referenced as the design container during physical design. One of the inputs to the design library which will make the library technology specific is the technology file.

CREATING DESIGN LIBRARY

4.a Setting up the logical libraries. The below commands will set the logical libraries and define VDD and VSS

icc_shell > **lappend search_path** “/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models”

icc_shell > **set link_library** “ * saed90nm_max.db saed90nm_min.db saed90nm_typ.db”

icc_shell > **set target_library** " saed90nm_max.db"

icc_shell > **set mw_logic0_net** VSS
icc_shell > **set mw_logic1_net** VDD

icc_shell> **set_tlu_plus_files -max_tluplus** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/star_rcxt/tluplus/saed90nm_1p9m_1t_Cmax.tluplus **-min_tluplus** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/star_rcxt/tluplus/saed90nm_1p9m_1t_Cmin.tluplus **-tech2itf_map** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/astro/tech/tech2itf.map

4.b Creating Milkway database

icc_shell> **create_mw_lib -technology** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/astro/tech/astroTechFile.tf **-mw_reference_library** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/astro/fram/saed90nm_fr/ FIFO_design.mw

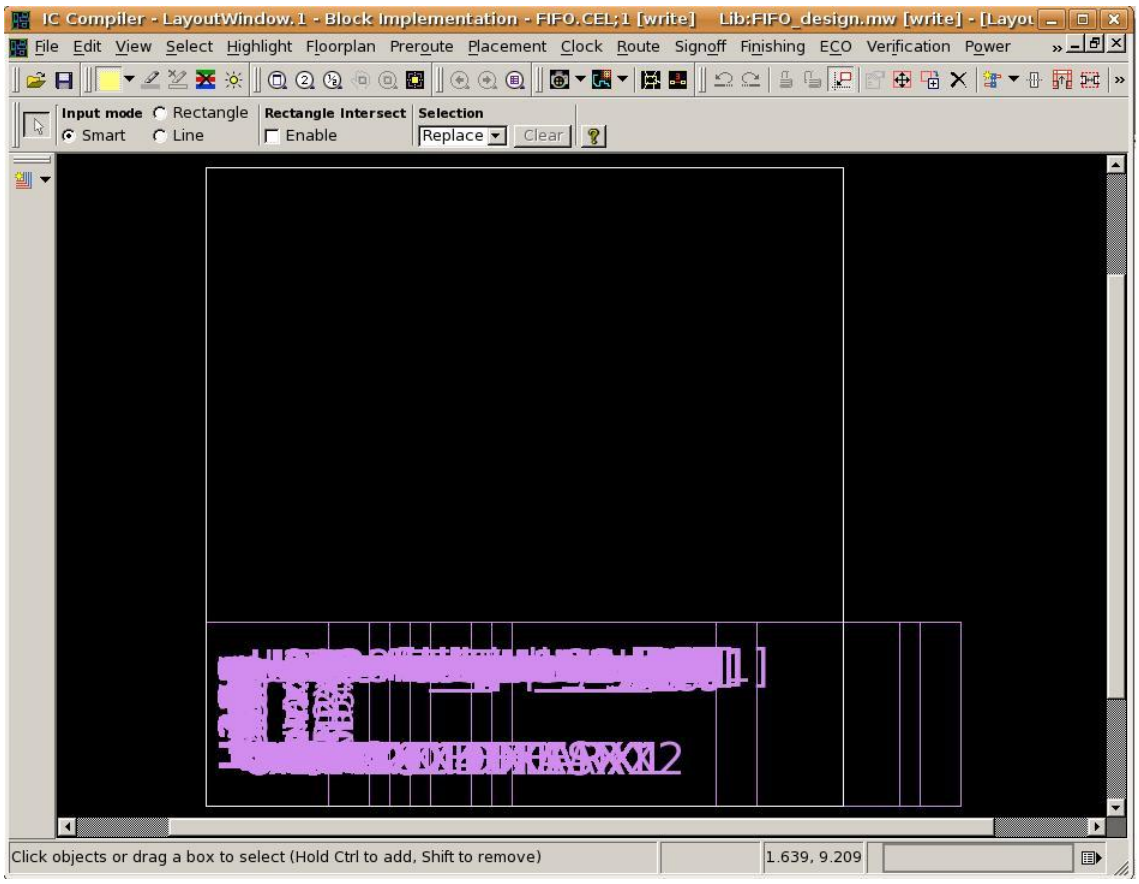
notice the space between saed90nm_fr/ and FIFO_design.mw, which is the design name. You can choose your own design name.

4.c Open the newly created Milkyway database

icc_shell > **open_mw_lib** FIFO_design.mw

4.d Read in the gate level synthesized verilog netlist. It opens up a layout window , which contains the layout information as shown below. You can see all the cells in the design at the bottom, since we have not initialized the floorplan yet or done any placement.

icc_shell > **read_verilog** ../synth_fifo/output/fifo.v



4.e Uniquify the design by using the uniquify_fp_mw_cel command. The Milkyway format does not support multiply instantiated designs. Before saving the design in Milkyway format, you must uniquify the design to remove multiple instances.

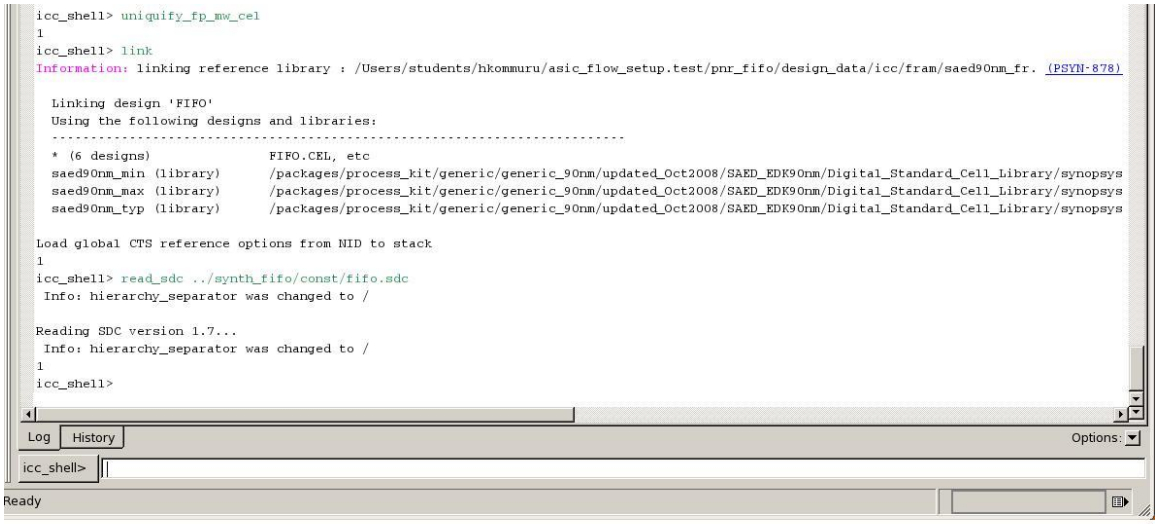
icc_shell> **uniquify_fp_mw_cel**

4.f Link the design by using the link command (or by choosing File > Import > Link Design in the GUI).

icc_shell> **link**

4.g Read the timing constraints for the design by using the read_sdc command (or by choosing File > Import > Read SDC in the GUI).

icc_shell > **read_sdc** ../synth_fifo/const/fifo.sdc



4.h Save the design.

icc_shell > **save_mw_cel –as** fifo_inital

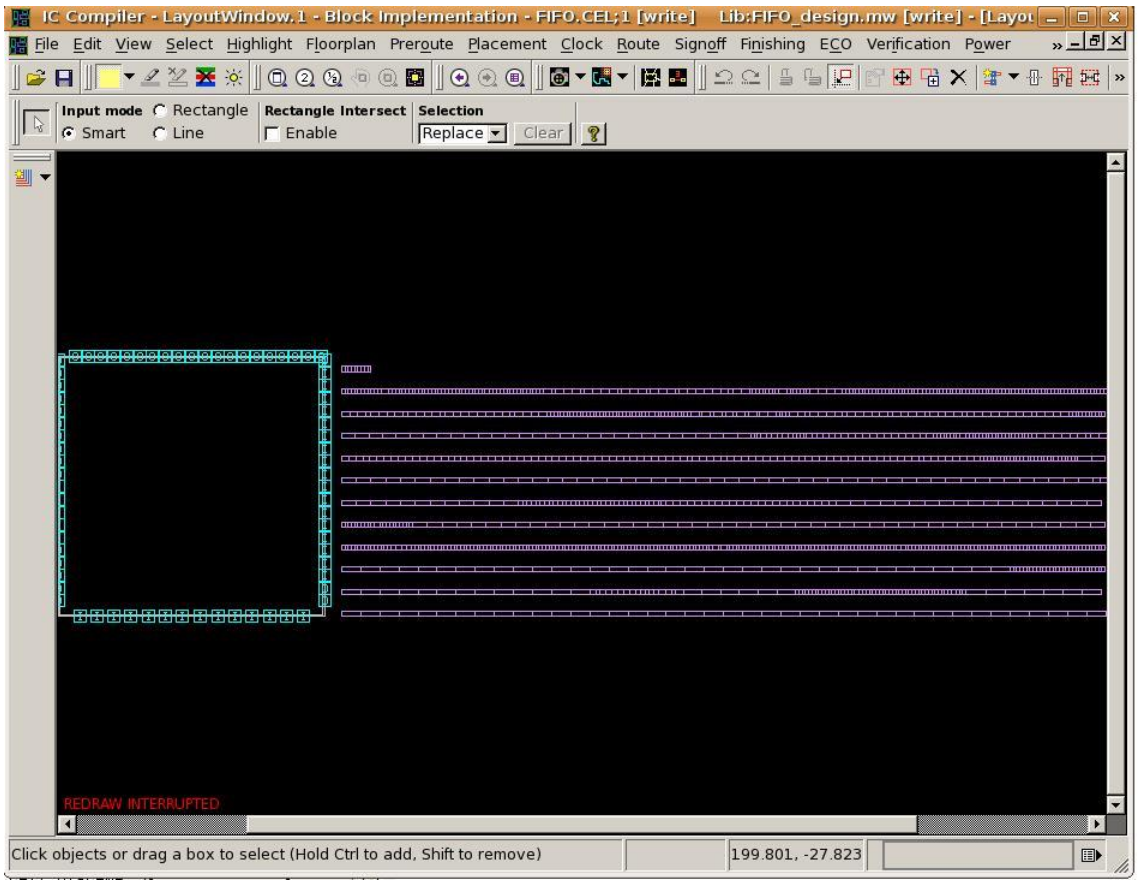
FLOORPLANNING

Open file /scripts/floorplan_icc.tcl

5. Initalize the floorplan with below command

icc_shell > **initialize_floorplan -core_utilization** 0.6 **-start_first_row -left_io2core** 5.0 **-bottom_io2core** 5.0 **-right_io2core** 5.0 **-top_io2core** 5.0 **-pin_snap**

You can see in the layout window, the floorplan size and shape. Since we are still in the floorplan stage all the cells in the design are outside of the floorplan . You can change the above options to play around with the floorplan size.



6. Connect Power and Ground pins with below command

icc_shell > **derive_pg_connection -power_net** VDD **–ground_net** VSS
icc_shell > **derive_pg_connection -power_net** VDD **–ground_net** VSS **-tie**

7. ICC automatically places the pins around the boundary of the floorplan evenly, if there are no pin constraints given. You can constrain the pins around the boundary [the blue color pins in the above figure] , using a TDF file. You can look at the file /const/fifo.tdf

You need to create the TDF file in the following format

pin PINNAME layer width height pinside [pin Order]
pin PINNAME layer width height pinside [pin Offset]

Example with pin Order
pin Clk M3 0.36 0.4 right 1
In this tutorial example, the tool is placing the pins automatically. Hence, you do not have to run this step.

8. Power Planning: First we need to create rectangular power ring around the floorplan.

##Create VSS ring

icc_shell>**create_rectangular_rings -nets** {VSS} **-left_offset** 0.5 **-left_segment_layer** M6 **-left_segment_width** 1.0 **-extend_ll -extend_lh -right_offset** 0.5 **-right_segment_layer** M6 **-right_segment_width** 1.0 **-extend_rl -extend_rh -bottom_offset** 0.5 **-bottom_segment_layer** M7 **-bottom_segment_width** 1.0 **-extend_bl -extend_bh -top_offset** 0.5 **-top_segment_layer** M7 **-top_segment_width** 1.0 **-extend_tl -extend_th**

Create VDD Ring

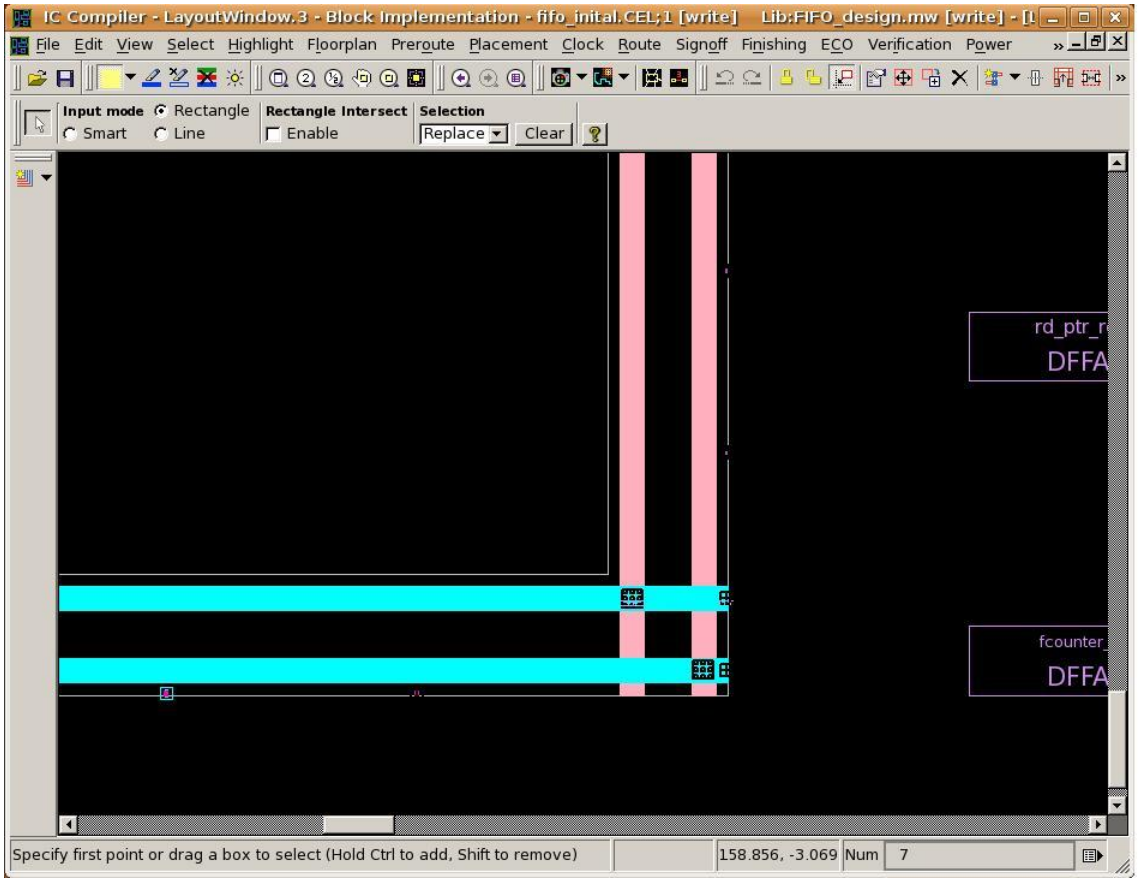
icc_shell>**create_rectangular_rings -nets** {VDD} **-left_offset** 1.8 **-left_segment_layer** M6 **-left_segment_width** 1.0 **-extend_ll -extend_lh -right_offset** 1.8 **-right_segment_layer** M6 **-right_segment_width** 1.0 **-extend_rl -extend_rh -bottom_offset** 1.8 **-bottom_segment_layer** M7 **-bottom_segment_width** 1.0 **-extend_bl -extend_bh -top_offset** 1.8 **-top_segment_layer** M7 **-top_segment_width** 1.0 **-extend_tl -extend_th**

Creates Power Strap

icc_shell>**create_power_strap -nets** { VDD } **-layer** M6 **-direction** vertical **-width** 3

icc_shell>**create_power_strap -nets** { VSS } **-layer** M6 **-direction** vertical **-width** 3

See in the figure below.



10. Save the design .

icc_shell> **save_mw_cel –as** fifo_fp

PLACEMENT

open /scripts/place_icc.tcl

During the optimization step, the place_opt command introduces buffers and inverters tofix timing and DRC violations. However, this buffering strategy is local to some critical paths.The buffers and inverters that are inserted become excess later because critical paths change during the course of optimization. You can reduce the excess buffer and inverter counts after place_opt by using the set_buffer_opt_strategy command, as shown

icc_shell> **set_buffer_opt_strategy -effort** low

This buffering strategy will not degrade the quality of results (QoR).

Also, by default, IC Compiler performs automatic high-fanout synthesis on nets with a fanout greater than or equal to 100 and does not remove existing buffer or inverter trees.

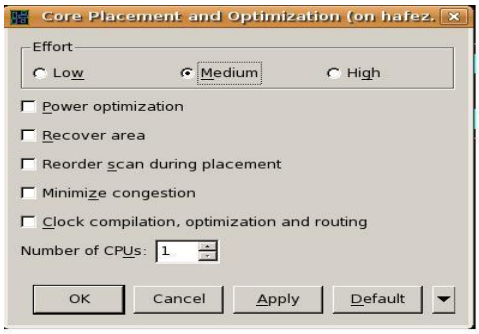
You can control the medium- and high-fanout thresholds by using the -hf_thresh and -mf_thresh options, respectively. You can control the effort used to remove existing buffer and inverter trees by using the -remove_effort option.

Setting TLUplus files:
TLUPlus models are a set of models containing advanced process effects that can be used by the parasitic extractors in Synopsys place-and-route tools for modeling. These files need to be set using the set_tlu_plus_files command as shown below:

icc_shell> **set_tlu_plus_files -max_tluplus** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/star_rcxt/tluplus/saed90nm_1p9m_1t_Cmax.tluplus **-min_tluplus** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/star_rcxt/tluplus/saed90nm_1p9m_1t_Cmin.tluplus **-tech2itf_map** /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/process/astro/tech/tech2itf.map

11. Goto Layout Window , Placement → Core Placement and Optimization . A new window opens up as shown below . There are various options, you can click on what ever option you want and say ok. The tool will do the placement. Alternatively you can also run at the command at icc_shell . Below is example with congestion option.

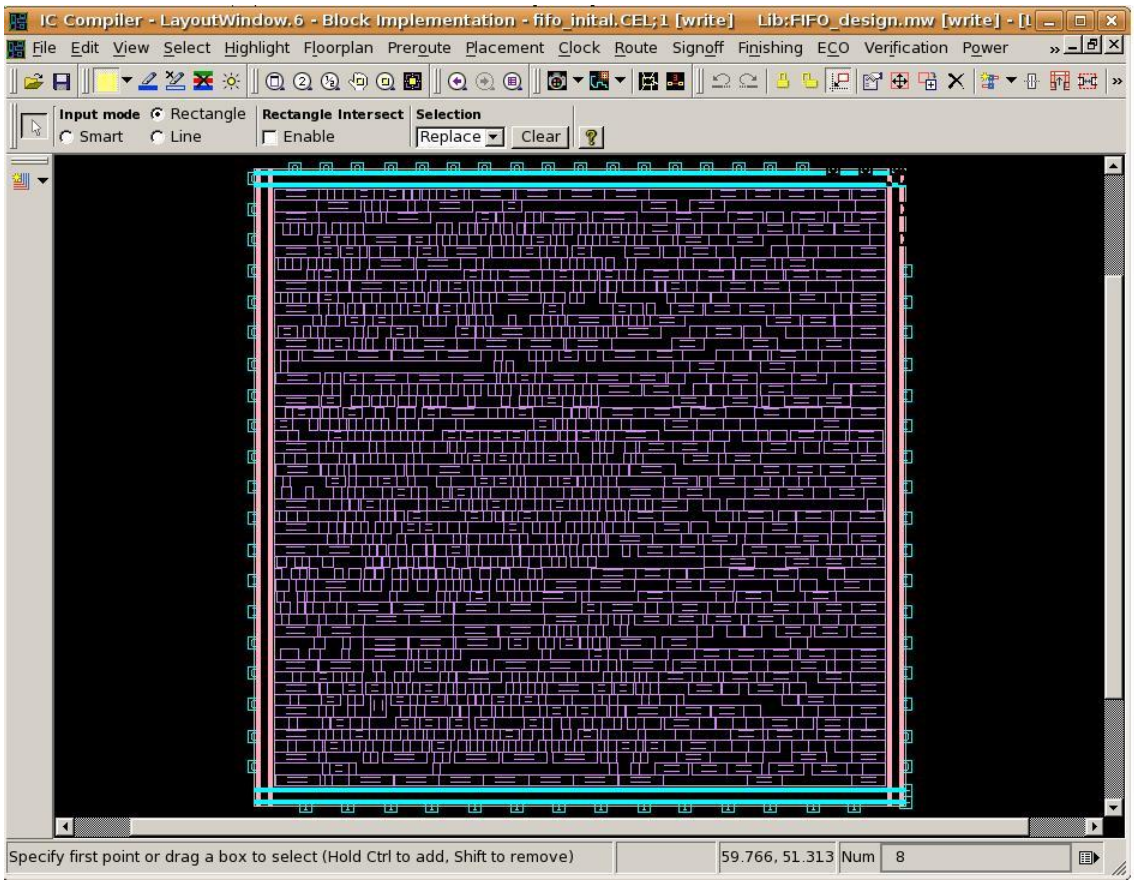
icc_shell > **place_opt –congestion**



When you want to add area recovery, execute :
place_opt -area_recovery -effort low

```
# When the design has congestion issues, you have following choices :
#   place_opt -congestion -area_recovery -effort low  # for medium effort congestion
removal
#   place_opt -effort high -congestion -area_recovery # for high eff cong removal


## What commands do you need when you want to optimize SCAN ?
#   read_def < def file >
#   check_scan_chain > $REPORTS_DIR/scan_chain_pre_ordering.rpt
#   report_scan_chain >>  $REPORTS_DIR/scan_chain_pre_ordering.rpt
#   place_opt -effort low -optimize_dft


## What commands do you need when you want to reduce leakage power ?
#   set_power_options -leakage true
#   place_opt -effort low -area_recovery -power


## What commands do you need when you want to reduce dynamic power ?
#   set_power_options -dynamic true -low_power_placement true
#   read_saif –input < saif file >
#   place_opt -effort low -area_recovery –power
#         Note : option -low_power_placement enables the register clumping algorithm in
#               place_opt, whereas the option -dynamic enables the
#               Gate Level Power Optimization (GLPO)


## When you want to do scan opto, leakage opto, dynamic opto, and you have congestion
issues,
## use all options together :
#   read_def < scan def file >
#   set_power_options -leakage true -dynamic true -low_power_placement true
#   place_opt -effort low -congestion -area_recovery -optimize_dft -power -num_cpus
```

12. After the placement is done, all the cells would be placed in the design and it would the below window.



13. You can report the following information after the placement stage.

icc_shell> **save_mw_cel –as** fifo_place

Reports

icc_shell>**report_placement_utilization** > output/fifo_place_util.rpt
icc_shell>**report_qor_snapshot** > output/fifo_place_qor_snapshot.rpt
icc_shell>**report_qor** > output/fifo_place_qor.rpt

Timing Report

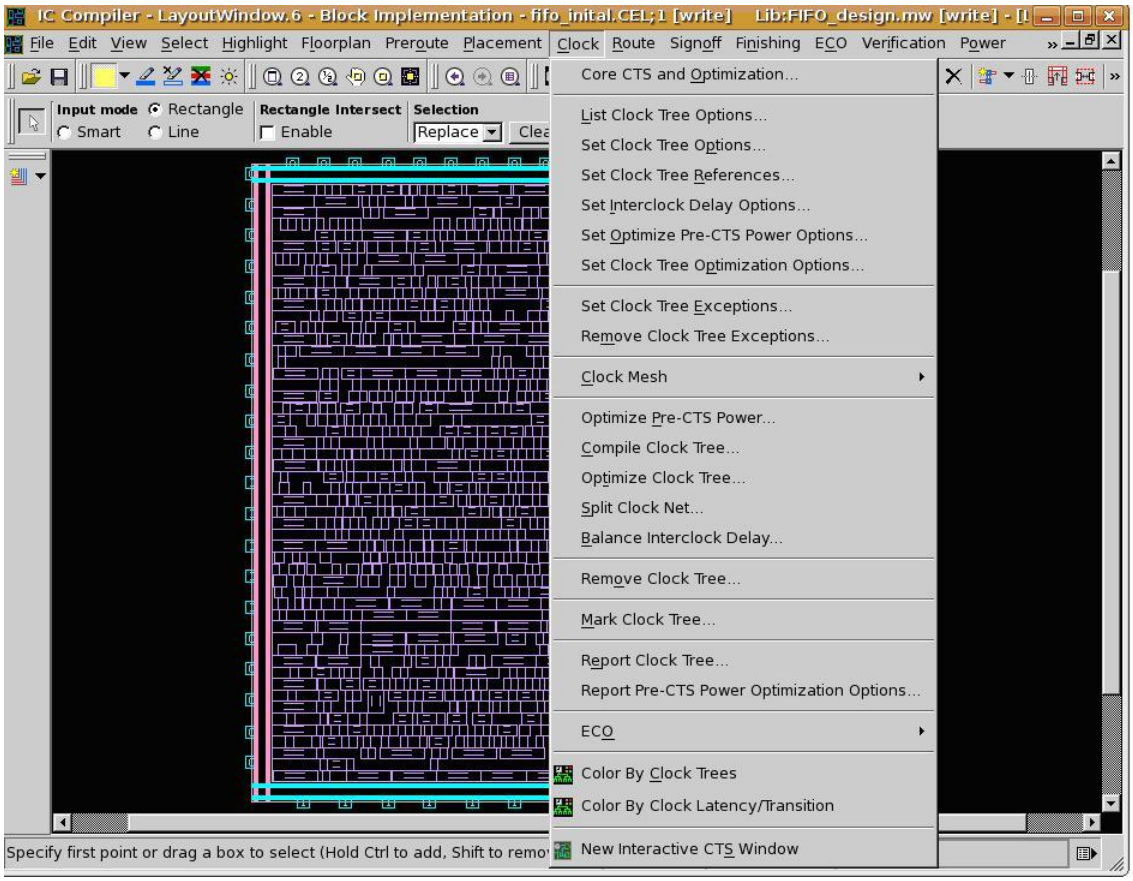
icc_shell>**report_timing** -delay max -max_paths 20 > output/fifo_place.setup.rpt
icc_shell>**report_timing** -delay min -max_paths 20 > output/fifo_place.hold.rpt

After placement, if you look at the fifo_cts.setup.rpt and fifo_cts.hold.rpt, in the reports directory, they meet timing.

CLOCK TREE SYNTHESIS

open scripts/ cts_icc.tcl

Before doing the actual cts, you can set various optimization steps. In the Layout window, click on "Clock ", you will see various options, you can set any of the options to run CTS. If you click on Clock → Core CTS and Optimization .



14. Save the Cell and report timing

icc_shell>**save_mw_cel** -as fifo_cts
icc_shell> **report_placement_utilization** > reports/fifo_cts_util.rpt
icc_shell> **report_qor_snapshot** > reports/fifo_cts_qor_snapshot.rpt
icc_shell> **report_qor** > reports/fifo_cts_qor.rpt
icc_shell> **report_timing** –max_paths 20 –delay max > reports/fifo_cts.setup.rpt
icc_shell> **report_timing** –max_paths 20 –delay min > reports/fifo_cts.hold.rpt

CTS POST OPTIMIZATION STEPS

##Check for hold time and setup time and do incremental if there are any violations
setup time fix

clock_opt -only_psyn

clock_opt -sizing

hold_time fix
clock_opt -only_hold_time

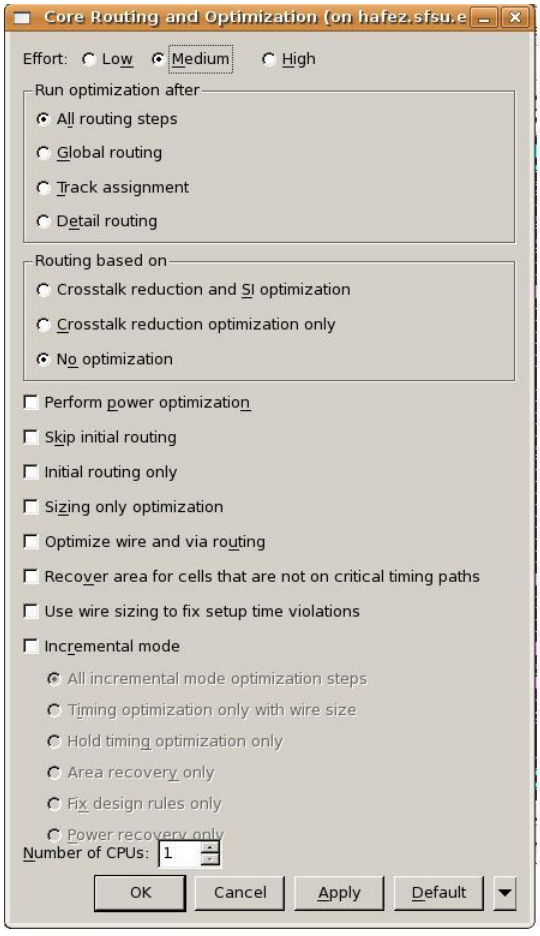
Save cel again and report paths
save_mw_cel -as fifo_cts_opt

Timing Report
report_timing -delay max -max_paths 20 > reports/fifo_cts_opt.setup.rpt
report_timing -delay min -max_paths 20 > reports/fifo_cts_opt.hold.rpt

ROUTING

open scripts/route_icc.tcl

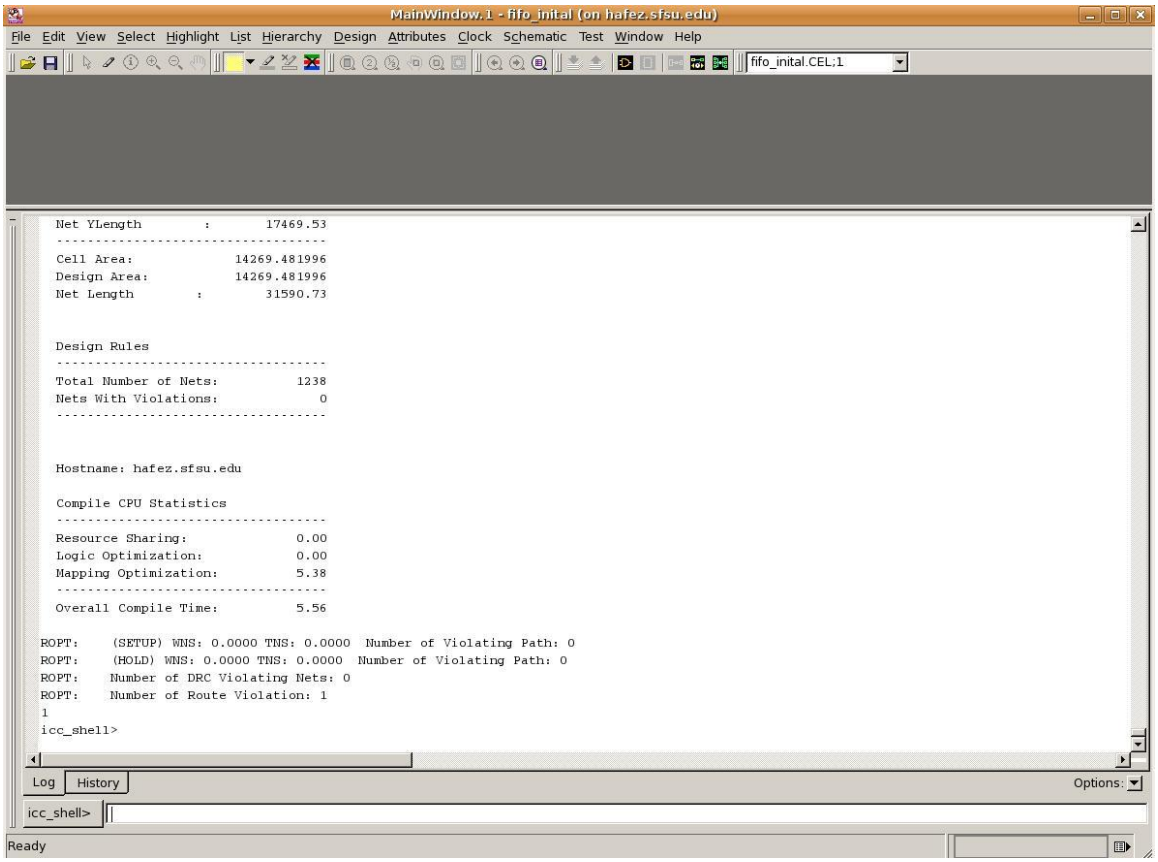
14 . In the layout window, click on Route → Core Routing and Optimization, a new window will open up as shown below

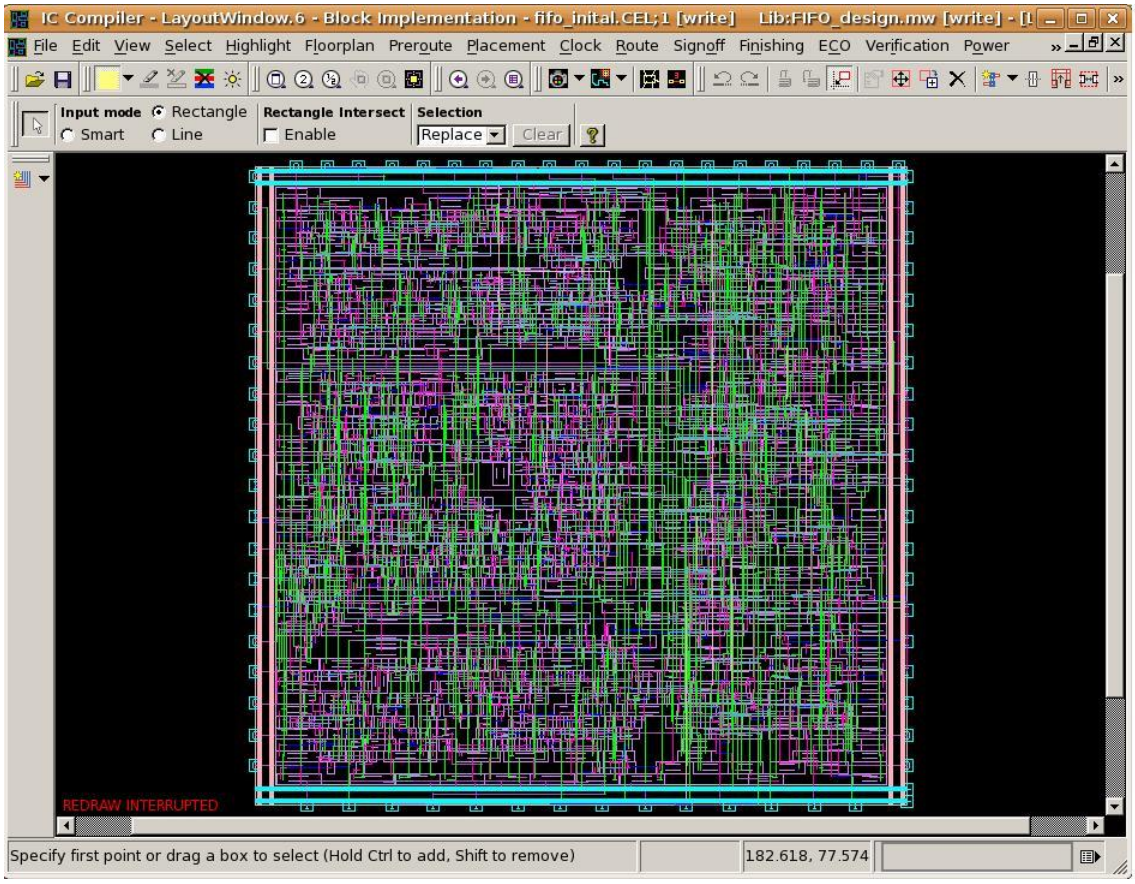


You can select various options, if you want all the routing steps in one go, or do global routing first, and then detail, and then optimization steps. It is up to you.

icc_shell> **route_opt**

Above command does not have optimizations. You can although do an incremental optimization by clicking on the incremental mode in the above window after route_opt is completed. View the shell window after routing is complete. You can see that there are no DRC violations reported, indicating that the routing is clean:



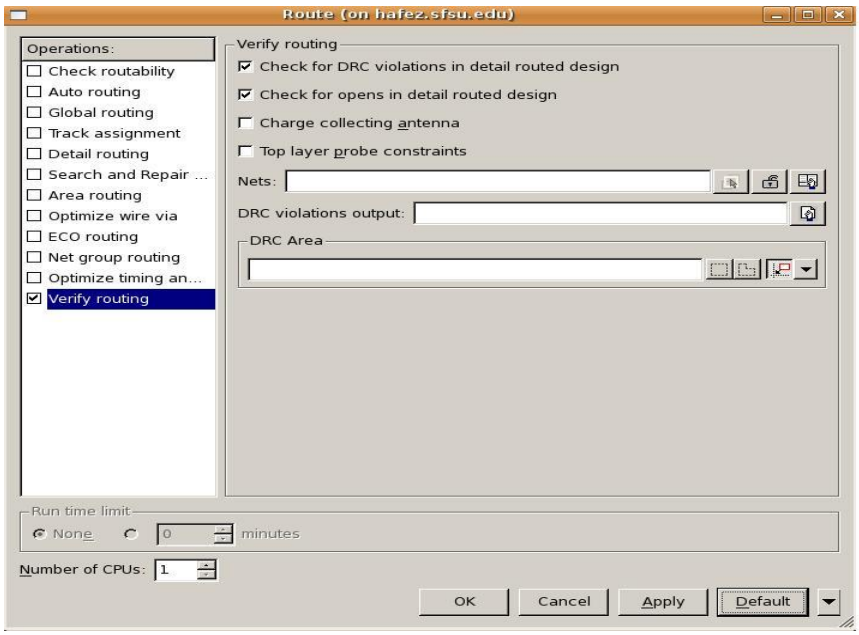


15. Save the cel and report timing

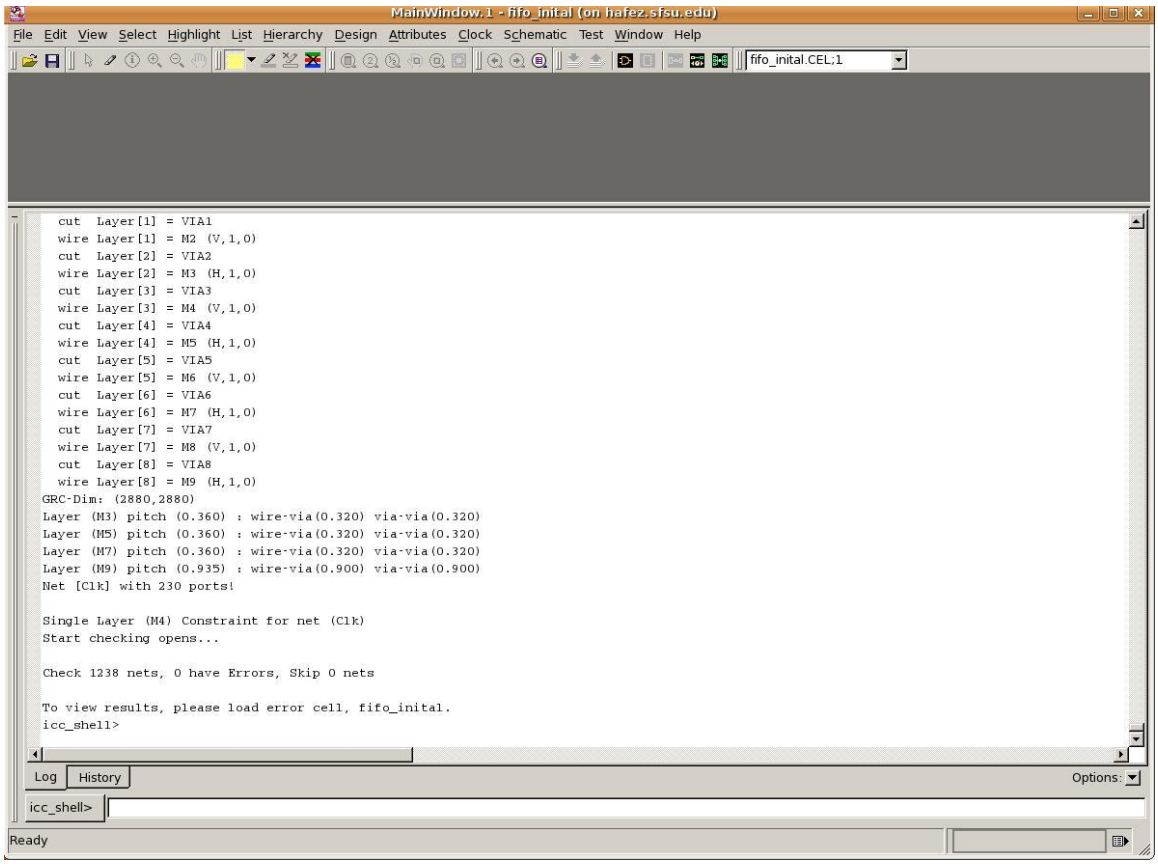
icc_shell > **save_mw_cel** –as fifo_route
icc_shell> **report_placement_utilization** > reports/fifo_route_util.rpt
icc_shell> **report_qor_snapshot** > reports/fifo_route_qor_snapshot.rpt
icc_shell> **report_qor** > reports/fifo_route_qor.rpt
icc_shell> **report_timing** –max_paths 20 –delay max > reports/fifo_route.setup.rpt
icc_shell> **report_timing** –max_paths 20 –delay min > reports/fifo_route.hold.rpt

POST ROUTE OPTIMIZATION STEPS

16. Goto Layout Window, Route → Verify Route, it opens up a new window as shown below, click ok.



The results are clean , as you can see in the window below:



If results are not clean, you might have to do post route optimization steps, like incremental route. Verify, clean, etc.

EXTRACTION

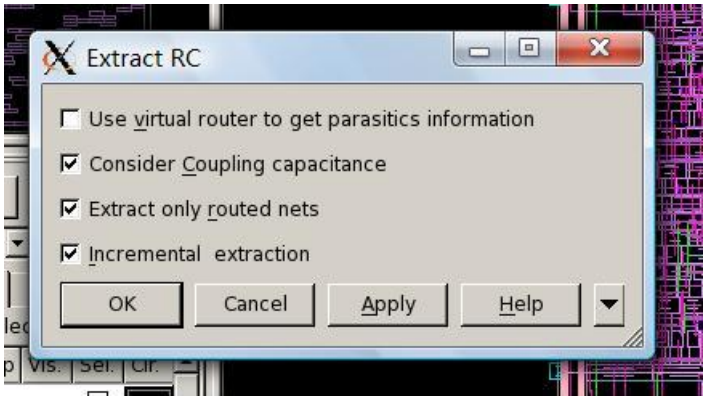
9.0 Introduction

In general, almost all layout tools are capable of extracting the layout database using various algorithms. These algorithms define the granularity and the accuracy of the extracted values. Depending upon the chosen algorithm and the desired accuracy, the following types of information may be extracted:
Detailed parasitics in DSPF or SPEF format.
Reduced parasitics in RSPF or SPEF format.
Net and cell delays in SDF format.
Net delay in SDF format + lumped parasitic capacitances.
The DSPF (Detailed Standard Parasitic Format) contains RC information of each segment (multiple R's and C's) of the routed netlist. This is the most accurate form of extraction. However, due to long extraction times on a full design, this method is not practical. This type of extraction is usually limited to critical nets and clock trees of the design.
The RSPF (Reduced Standard Parasitic Format) represents RC delays in terms of a pi model (2 C's and 1 R). The accuracy of this model is less than that of DSPF, since it does not account for multiple R's and C's associated with each segment of the net. Again, the extraction time may be significant, thus limiting the usage of this type of information. Target applications are critical nets and small blocks of the design. Both detailed and reduced parasitics can be represented by OVI's (Open Verilog International) Standard Parasitic Exchange Format (SPEF). The last two (number 3 and 4) are the most common types of extraction used by the designers. Both utilize the SDF format. However, there is major difference between the two. Number 3 uses the SDF to represent both the cell and net delays, whereas number 4 uses the SDF to represent only the net delays. The lumped parasitic capacitances are generated separately. Some layout tools generate the lumped parasitic capacitances in the Synopsys set_load format, thus facilitating direct back annotation to DC or PT.

Extraction steps for the tutorial example:

open scripts/extract_icc.tcl

17. Go to Layout Window, Route → Extract RC, it opens up a new window as shown below, click ok.



Alternatively, you can run this script on the ICC shell:

icc_shell > **extract_rc** -coupling_cap -routed_nets_only -incremental

##write parasitic to a file for delay calculations tools (e.g PrimeTime).

icc_shell > **write_parasitics** -output ./output/fifo_extracted.spef -format SPEF

The above script will produce the min and max files, which can be used for delay estimation using PrimeTime tool.

##Write Standard Delay Format (SDF) back-annotation file

icc_shell > **write_sdf** ./output/fifo_extracted.sdf

##Write out a script in Synopsys Design Constraints format

icc_shell > **write_sdc** ./output/fifo_extracted.sdc

##Write out a hierarchical Verilog file for the current design, extracted from layout

icc_shell > **write_verilog** ./output/fifo_extracted.v

The extracted verilog netlist can be used to double-check the netlist by running a simulation on it using VCS

18. Report timing

icc_shell> **report_timing** –max_paths 20 –delay max > reports/fifo_extracted.setup.rpt
icc_shell> **report_timing** –max_paths 20 –delay min > reports/fifo_extracted.hold.rpt

19. Report power

icc_shell> **report_power** > reports/fifo_power.rpt

If you open the generated fifo_power.rpt, you will notice both total dynamic power (active mode) and cell leakage power (standby mode) being reported.

20. Save the design

icc_shell> **save_mw_cel** -as fifo_extracted

Post-Layout Timing Verification Using PrimeTime

After extraction of RC parasitics (.spef file) and the enlist (.V) from layout, we can run primetime to perform timing verification on the extracted netlist. The steps will be similar to the step we used to run prime time on the verilog netlist obtained after DC synthesis, except that we need to use the post layout netlist as well as the parasitic information.

1. Please source "***synopsys_setup.tcl***" which sets all the environment variables necessary to run the Primetime . Please type csh at the unix prompt before you source the below script. Please source the above file from the below location.

[hkommuru@hafez.sfsu.edu] $ csh
[hkommuru@hafez.sfsu.edu] $ cd
[hkommuru@hafez.sfsu.edu] $ cd /asic_flow_setup/pt_post
[hkommuru@hafez.sfsu.edu] $ source /packages/synopsys/setup/synopsys_setup.tcl

2. PT may be invoked in the command-line mode using the command pt_shell or in the GUI mode through the command primetime as shown below.
Command-line mode:
> pt_shell
GUI-mode:
> primetime

Before doing the next step, open the post_layout_pt.tcl script and keep it ready which is at location /scripts/*post_layout_pt.tcl*

[hkommuru@hafez.sfsu.edu]$ vi scripts/post_layout_pt.tcl

3. Just like DC setup, you need to set the path to link_library and search_path

pt_shell > set link_library [list /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_typ.db

/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max .db/packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_min.db]
pt_shell > set target_library [list /packages/process_kit/generic/generic_90nm/updated_Oct2008/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db]

4. Read the extracted netlist

pt_shell> read_verilog ../pnr_fifo/output/fifo_extracted.v

5. You need to set the top module name.

pt_shell > current_design FIFO

6. Read extracted parasitics

pt_shell> read_parasitics -format SPEF ../pnr_fifo/output/fifo_extracted.spef.max

7. Read in the SDC from the ICC

pt_shell> read_sdc ../pnr_fifo/output/fifo_extracted.sdc

8. Now we can do the analysis of the design. Generally, four types of analysis is performed on the design, as follows:

- From primary inputs to all flops in the design.
- From flop to flop.
- From flop to primary output of the design.
- From primary inputs to primary outputs of the design.

All four types of analysis can be accomplished by using the following commands:

pt_shell> report_timing -from [all_inputs] -max_paths 20 -to [all_registers -data_pins] > reports/timing.rpt

pt_shell> report_timing -from [all_register -clock_pins] -max_paths 20 -to [all_registers -data_pins] >> reports/timing.rpt

pt_shell> report_timing -from [all_registers -clock_pins] -max_paths 20 -to [all_outputs] >> reports/timing.rpt

pt_shell> report_timing -from [all_inputs] -to [all_outputs] -max_paths 20 >> reports/timing.rpt

The results are reported to the timing.rpt file under the reports subdirectory. Please open the file to see if the design meets the timing or if there are any errors.

9. Reporting setup time and hold time. Primetime by default reports the setup time. You can report the setup or hold time by specifying the –delay_type option as shown in below figure.

pt_shell> report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins] -delay_type max >> reports/timing.rpt

The report is added to the timing.rpt file. Please read the file.

10. Reporting hold time
pt_shell> read_parasitics -format SPEF ../pnr_fifo/output/fifo_extracted.spef.min

pt_shell> report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins] -delay_type min >> reports/timing.rpt

The report is added to the timing.rpt file. Please read the file.

11. Reporting timing with capacitance and transition time at each level in the path

pt_shell > report_timing -transition_time –capacitance –nets -input_pins –from [all_registers –clock_pins] –to [all_registers –data_pins] > reports/timing.tran.cap.rpt

The report is added to the timing.tran.cap.rpt file. Please read the file for results.

12. You can save your session and come back later if you chose to.

pt_shell > save_session output/fifo.session

APPENDIX A: Design for Test

A.0 Introduction

What do we do to test the chip after it is manufactured? Why do we need DFT?
Consider a scenario where a million chips are produced. It is time consuming and an extremely costly process of trying to test each of these million chips. The design being correct does not guarantee that the manufactured chip is operational. It could have a lot of manufacturing defects. For example for 0.13nm technology, the percentage of chips that are good is about 60%. Also it is very difficult to ascertain that the chip manufactured will function correctly for all possible inputs and various other conditions. Moreover even we do test the chip; we can only see the inputs and the output pins. A designer will not be able to see what is happening at the intermediate steps before the signal reaches the output pins. To overcome this problem DFT was started. This is the second best thing we can do. It increases the probability of the chip working in a real system. To perform Design for Test, very small changes in the design are needed.

In CMOS technology, the transistor consists of both NMOS and PMOS transistors. If we take an example of and inverter, we see that PMOS is connected to VDD and NMOS is connected to VSS. Inverter inverts the given input. PMOS transistor is on when the input to the gate is zero. It charges up the capacitor to the VDD value and the opposite e happened for NMOS. So basically NMOS acts alike a pull down device and PMOS acts like a pull up device. SO sometimes what happens is that these pull up and pull down devices don't work properly in some transistors and a fault occurs. This fault is called as 'Stuck at Fault'. The logic could be stuck at zero permanently or at '1' permanently without depending on the input values. The next topic covers Test techniques.

A.1 Test Techniques

A.1.1 Issues faced during testing

1. Consider a combinatorial circuit which has N inputs. To validate the circuit, we need to exhaustively apply all possible input test vectors and observe the responses. Therefore for N inputs we need to generate 2 to the power of N inputs. If the number of inputs is too high, we still can manage to it, but it would take a long time.
Test vector: A set of input vectors to test the system.
2. Now consider a Sequential circuit, for a sequential circuit, the output of the circuit depends on the inputs as well as the state value. To test such a circuit, it would take extremely long long time and is practically impossible to do it.
To overcome the above issue, a Scan-based Methodology was introduced

A.2 Scan-Based Methodology

To begin with, there are two important concepts we need to understand. They are:

2. **Controllability**: It is the ability to control the nodes in a circuit by a set of inputs. For a given set of input and output pins, when we give the system a set of input

test vectors, we should be able to control each node we want to test. The higher the degree of controllability, the better.

3. **Observability**: It is the ease with which we can observe the changes in the nodes (gates). Like in the previous case the higher the observability, the better. What I mean by saying higher is that, we can see the desired state of the gates at the output in lesser number of cycles.

It is easy to test combinational circuits using the above set of rules. For sequential circuits, to be able to the above, we need to replace the flip flops (FF) with 'Scan-FF'. These Scan Flip Flops are a special type of flip flops; they can control and also check the logic of the circuits. There are two methodologies:

1. Partial Scan: Only some Flip-Flops are changed to Scan-FF.
2. Full Scan: Entire Flip Flops in the circuit are changed to these special Scan FF. This does mean that we can test the circuit 100%.

What does the Scan-FF consist of and how does it work?
The Scan-FF is nothing but a Multiplexer added to the Flip-Flop. We can control the input to the flip-flops by using the select pin of the multiplexer to do it. Therefore using the select pin, the circuit can run in two modes, functional mode and scan mode. In the scan mode, the select input is high and the input test vectors are passed to the system. So now we have controlled the circuit and we can therefore see how the circuit is going to behave for the specific set of input test vectors which were given. After the input vectors are in the exact place we want them to be in the circuit, the select pin is changed to low and the circuit is now in functional mode. We can now turn back the select pin to Scan Mode and the output vectors can be seen at the output pins and a new set of input test vectors are flushed into the system. We can check the output vectors to see whether they are expected results or not.
→Some Flip-Flops cannot be made scanable like Reset pins, clock gated Flip Flops
→Typical designs might have hundreds of scan chains...
→The number of scan chains depends on the clock domains. (Normally within a domain, it is not preferable to have different clocks in each scan chain)
Example: If there are 10 clock domains, then it means the minimum number of scan chains would be 10.
→Scan chains test logic of combinational gates
→Scan chains test sequential Flip-Flops through the vectors which pass through them. Any mismatch in the value of the vectors can be found.

Manufacturing tests: These include functional test and performance test. Functional test checks whether the manufactured chip is working according to what is actually designed and performance test checks the performance of the design (speed).
→ After the chip comes back from the foundry, it sits on a load board to be tested. The equipment which tests these chips is called Automatic Test Equipment (ATE).
→The socket is connected to the board. The chip is placed on a socket. A mechanical arm is used to place the chip on the socket.

→ A test program tells the ATE what kind of vectors needs to be loaded. Once the vectors are loaded, the logic is computed and the output vectors can be observed on the screen.
→The output pattern should match the expected output.
→When an error is found, you can exactly figure out which flip-flop output is not right Through the Automatic Test Pattern Generator, we can point out the erroneous FF.
→ Failure Analysis analyzes why the chip failed. This is a whole new different field.
→ Fault Coverage: It tells us that for a given set of vectors, how many faults are covered.
→ Test Coverage: It is the coverage of the testable logic; meaning how much of the circuit can be tested.
→ The latest upcoming technology tests the chips at the wafer level (called wafer sort (with probes))

A.3 Formal Verification

Formal Verification verifies the circuit without changing the logic of the circuit. It is the defacto standard used today in the industry. Some of the tools which use formal verification today in the industry are CONFORMAL (from cadence), FORMALITY (from Synopsys). Formal verification is an algorithmic-based approach to logic verification that exhaustively proves functional properties about a design. The algorithms formal verification uses are:
→ Binary Decision Diagram (BDD): It is a compact data structure of Boolean logic. It can represent the logic state encoded as a Boolean function.
→Symbolic FSM Traversal
Typically, the following are the types of formal verification:
1.**Equivalence Checking**: Verifies the functional equivalence of two designs that are at the same or different abstraction levels (e.g., RTL-to-RTL, RTL-to-Gate, or Gate-to-Gate). It checks combinatorial and sequential elements. (Basically checks if two circuits are equivalent). For sequential elements, it checks if the specific instance name occurs in both the circuits or not.
2. **Model Checking**: Verifies that the implementation satisfies the properties of the design. Model checking is used early in the design creation phase to uncover functional bugs.
Compare Point: [Cadence Nomenclature]; it is defined as any input of a sequential element and any primary output ports.

APPENDIX B: EDA Library Formats

B.1 Introduction

In the EDA industry, library is defined as a collection of cells (gates). These cells are called standard cells. There are different kinds of libraries which are used at different steps in the whole ASIC flow. All the libraries used contain standard cells. The libraries contain the description of these standard cells; like number of inputs, logic functionality, propagation delay etc. The representation of each standard cell in each library is different.

There is no such library which describes all the forms of standard cells. For example: the library used for timing analysis contains the timing information of each of the standard cells; the library used during Physical Implementation contains the geometry rules. Standard Cell Libraries determine the overall performance of the synthesized logic. The library usually contains multiple implementations of the same logic-function, differing by area and speed. For example few of the basic standard cells could be a NOR gate, NAND gate, AND gate, etc. The standard cell could be a combination of the basic gates etc. Each library can be designed to meet the specific design constraints. The flowing are some library formats used in the ASIC flow:

LEF: Library Exchange Format:
LEF is used to define an IC process and a logic cell library. For example, you would use LEF to describe a gate array: the base cells, the logic macros with their size and connectivity information, the interconnect layers and other information to set up the database that the physical design tools need.

DEF: Design Exchange format:
DEF is used to describe all the physical aspects of a particular chip design including the netlist and physical location of cells on the chip. For example, if you had a complete placement from a floorplanning tool and wanted to exchange this information with another tool, you would use DEF.

EDIF: (Electronic Design Interchange Format)
It is used to describe both schematics and layout.

GDSII: Generic Data Structures Library:
This file is used by foundry for fabricating the ASIC.

SPICE: (Simulation Program with Integrated Circuits Emphasis)
SPICE is a circuit simulator. It is used to analyze the behavior of the circuits. It is mostly used in the design of analog and mixed signal IC design. The input to SPICE is basically at netlist, and the tool analyzes the netlist information and does what it is asked to do.

PDEF: Physical Design Exchange Format:
It is a proprietary file format used by Synopsys to describe placement information and the clustering of logic cells.

SPF: Standard Parasitic Format
This file is generated by the Parasitic Extraction Tool. It contains all the parasitic information of the design such as capacitance, resistance etc. This file is generated after the layout. It is back annotated and the timing analysis of the design is performed again to get the exact timing information.

.lib: Liberty Format
This library format is from Synopsys.

.plib: Physical Liberty Format

This library format is from Synopsys. It is mainly used for Very Deep Sub Micron Technology. This library is an extension of the Liberty format; it contains extensive, advanced information needed for Physical Synthesis, Floorplanning, Routing and RC Extraction

SPEF: Standard Parasitic Exchange Format

Standard Parasitic Exchange Format (**SPEF**) is a standard for representing parasitic data of wires in a chip in ASCII format. Resistance, capacitance and inductance of wires in a chip are known as parasitic data. SPEF is used for delay calculation and ensuring signal integrity of a chip which eventually determines its speed of operation.

SDC: Synopsys Design Constraints File

SDC is a Tcl Based format-constraining file. It provides the timing information, constraints regarding the design to the tool. The tool uses the SDC file when performing timing analysis.

SDF: Standard Delay Format

Primetime writes out SDF for gate level simulation purposes. It describes the gate delay and also the interconnect delay. SDF can also be used with floorplanning and synthesis tools to backannotate the interconnect delay. A synthesis tool can us this information to improve the logic structure.

Fragment of SDF file:

(TIMESCALE 100ps) (INSTANCE B) (DELAY (ABSOLUTE

(NETDELAY net1 (0.6)))

(INSTANCE B) (DELAY (ABSOLUTE

(INTERCONNECT A.INV8.OUT B.DFF1.Q (:0.6:) (:0.6:))))

In this example the rising and falling delay is 60 ps (equal to 0.6 units multiplied by the time scale of 100 ps per unit specified in a TIMESCALE construct. The delay is specified between the output port of an inverter with instance name A.INV8 in block A and the Q input port of a D flip-flop (instance name B.DFF1) in block B