ASIC Design Manual

Using Cadence NCLaunch and Silicon Ensemble,
Synopsys Chip Synthesis and PrimeTime

Version 1.0

By

Zheng Huan Qun
March 2006

Department of Electrical and Computer Engineering
National University of Singapore

Author's Note

Writing this manual has provided me with a valuable opportunity to study ASIC design, which bears significant difference from the analog design that I was accustomed to doing. During this enriching process, I gained understanding of ASIC design and learn all the EDA tools required for it. All of this commenced with a search for books on ASIC design to mastering EDA tools and finally finishing the manual after a year's effort. I strived to make the contents, to the largest extent possible, parallel to practical work. May this manual become the handy guide for our students and staff who will be doing ASIC design. I hope that you may kindly provide me with useful feedback. Please email me at elezhq@nus.edu.sg.

Zheng Huan Qun
16 March 2006

1. Introduction

This manual describes the method of ASIC design from front-end to back-end using cadence NCLaunch, cadence silicon ensemble, synopsys chip synthesis and primetime.

The manual is meant for the beginners of ASIC design. The usages of the cadence and synopsys tools are demonstrated with graphic user interface (GUI), for users understand easily and apply conveniently.

1.1 Prerequisite

Users need to know Hardware Description Language (HDL), either VHDL or Verilog, and are able to write RTL code with HDL. Users must have the knowledge of digital circuits.

1.2 Overview of ASIC Design

ASIC design flow is shown in figure 1-1. As it shows, the front-end design includes RTL coding, RTL functionality verification, synthesis, pre-layout verification and pre-layout static time analysis (STA), and the back-end design includes place and route, post-layout verification and post-layout STA.

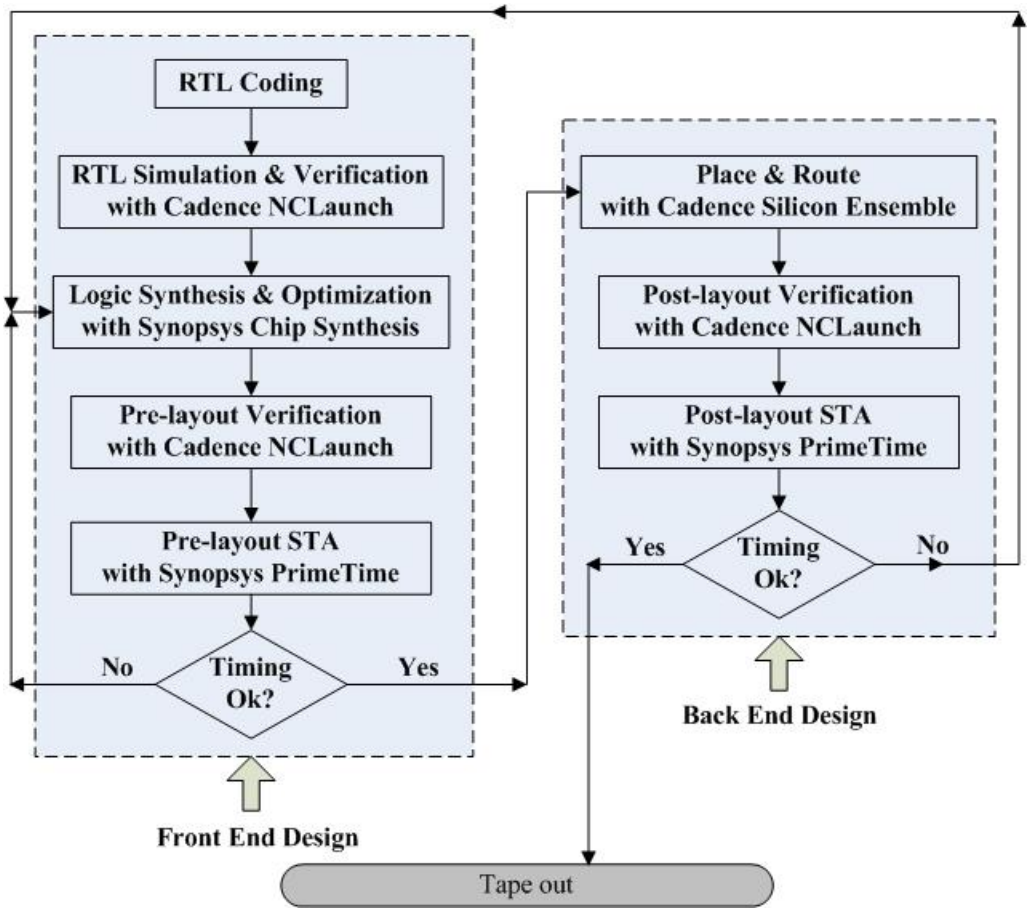


Figure 1-1 ASIC design flow.

The explanation of the flow is as follows.

- RTL coding – to code design with HDL.
- RTL Simulation & Verification – to simulate the RTL code and verify that the code is logically and functionally right.
- Logic Synthesis & Optimization – to synthesis and optimize the code and get the gate level netlist of the design.
- Pre-layout Verification – to verify that the gate level netlist satisfies the specifications of the design.
- Pre-layout STA – to do static timing analysis with standard cell delays and wire load models.
- Place & Route – to get the layout of the design.
- Post-layout Verification – to verify the layout level netlist satisfies the specifications of the design.
- Post-layout STA – to do static timing analysis with standard cell delays, net delays and parasitics.

The design can be taped out if the design satisfies its specifications after post-layout STA. If not, it has to be brought back to synthesize and optimize again. If no matter how hard the synthesis level it takes and the design still cannot meet the specifications, modifying the source (RTL) code has to be considered.

1.3 Arrangement of the Manual

RTL coding style affects the final chip synthesis results directly, so understanding the hardware implications for coding constructs is important. The hardware implications for code - *if-else*, *case* and *for loop* are described briefly in *chapter 2*. Advanced users may refer to synopsys documents or HDL books for more information.

In *chapter 3*, the usage of cadence NCLaunch is described and demonstrated. The steps to compile, elaborate and simulate a Verilog (or VHDL) design are listed in details, and the steps to save & view output data are listed in details too. The verification of RTL code using NCLaunch is demonstrated with a 32 bit adder. The method described in this chapter will be used during pre-layout and post-layout verification.

Once it is verified that the RTL code is logically and functionally right. The code is brought to synopsys chip synthesis for synthesizing and optimizing to get a gate level netlist. The method of synthesis and optimization is described, and the normal steps of running chip synthesis are listed in *chapter 4*. The whole flow is demonstrated with the 32 bit adder RTL code which has passed the verification in chapter 3.

Does the gate level netlist meet the specifications of the design? A pre-layout verification needs to be done using NCLaunch. The difference between pre-layout and RTL code verification is that the standard cell delays are considered while simulating the pre-layout netlist (the gate level netlist). The delay information is saved in a standard delay format (SDF) file which is got from chip synthesis. *Chapter 5* focuses on SDF back annotation system task. The demonstration is done with the 32 bit gate level netlist which is the output of chapter 4.

In *chapter 6*, the pre-layout STA using primetime is described. Pre-layout STA is to check the timing of the design. The method of doing STA using primetime is demonstrated with the 32 bit adder gate level netlist in this chapter.

After the design is verified, its gate level netlist can be brought to cadence silicon ensemble for place and route to get its layout. The full steps from setting up library, floor planning, cell placement, power ring creation and clock generation to route are demonstrated with the design example – 32 bit adder in *chapter 7*.

Post-layout verification is presented and demonstrated in *chapter 8*. Like pre-layout verification, SDF back annotation is used to annotate the design. The difference between post-layout and pre-layout verification is that the post-layout SDF file includes both delays of standard cells and nets while pre-layout SDF file has the standard cell delays only.

Post-layout STA using primetime is described in *chapter 9*. A SDF file including delay information of the design and a reduced standard parasitic format (RSPF) file including the parasitics are used to back annotate the design during STA. The method to back annotate the design is demonstrated with the 32 bit adder in this chapter.

1.4 Environment setup

To use the ASIC design manual, the following tools are needed,

- Cadence NCLaunch,
- Cadence Silicon Ensemble,
- Synopsis Chip Synthesis, and
- Synopsys PrimeTime.

The environment setup for using the above tools has to be done. Ask your system administrator for help.

2. Coding for Synthesis

Code that is functionally equivalent, but coded differently, will give different synthesis results. User cannot rely solely on Design Complier (DC) to fix a poorly coded design. Try to understand the "hardware" coded, to give DC the best possible starting point. The three big guidelines to write RTL code are as follows.

- Write HDL hardware descriptions and think of the topology implied by the code.
- Do not write HDL simulation models without explicit delays and file I/O.
- Isolate asynchronous logic from synchronous logic as synchronous designs run smoothly through synthesis test, simulation, and layout.

Keep in mind that writing in an RTL coding style means describing the register architecture, circuit topology and functionality between registers, and that DC optimizes logic between registers only not the register placement.

This chapter describes briefly hardware implication for some statements: *if*, *case* and *loop*. Partitioning a design is presented in this chapter, too.

2.1 *if* statements

➢ ***if-else*** statements

Code 1:	Code 2:

if (SEL=`1`) then
  SUM<=A+B;
else
  SUN<=C+D;
End if;
	if (SEL==1`b1)
	  begin
	  OP1=A;
	  OP2=B;
	  end
	else
	  begin
	  Op1=C;
	  Op2=D:
	  end
	SUM=Op1+Op2;

Code 1 construct implies multiplexing hardware figure 2-1 (a) or figure 2-1 (b). Code 2 implies figure 2-1 (b) only. Both codes are functionally same.

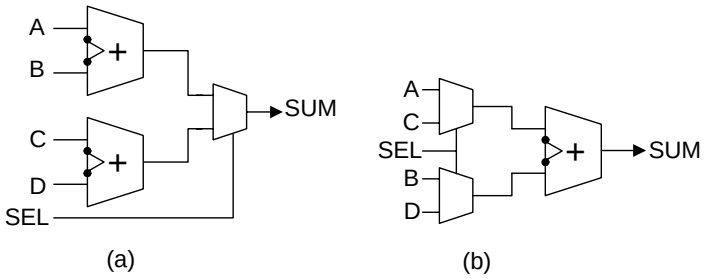


Figure 2-1 Implication of if-else.

- ***if*** statements and ***Latches***

Any signal that is not fully specified for all conditions infers a latch. Below is the code example of VHDL/Verilog.

VHDL code 1:	Verilog code 2:

LS373: process (ALE, ADBUS)
begin
  If (ALE=`1`) then
  ABUS<=ADBUS;
  end if;
end process LS373
	Always @ (ALE or ADBUS)
	begin
	  If (ALE)
	  ABUS=ADBUS;
	end

- ***if-then-elseif*** statements

VHDL and Verilog *if-elseif* statements imply priority, use only if priority checking is a circuit requirement. Priority control logic will be synthesized, resulting in a larger and possibly slower logic, if it is used. An example is shown below, and its implication is shown in figure 2-2.

```
module IPC(active[3:0], int0, int1, int2, int3)
input int0, int1, int2, int3;
output [3:0] active;
reg int0, int1, int2. int3;
reg[3:0] active;

always@(int0 or int1 or int2 or int3) begin
   active[3:0] = 4`b0000;
   if       (int0) active[0]=1`b1;
   else if (int1) active[1]=1`b1;
   else if (int2) active[1]=1`b1;
   else if (int3) active[1]=1`b1;
end; endmodule
```

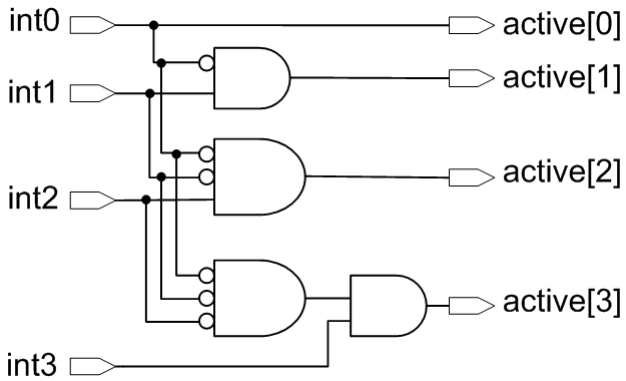


Figure 2-2 Implication of if-then-elseif.

There are cases where there is no need to use *if-then-elseif*, and they are

- when input signals have equal priority (no priority over each other), and
- when signals are mutually exclusive.

2.2 *Case* Statements

Case statements imply parallel MUX function, as shown in figure 2-3. The actual gates synthesized might not be a 4:1 MUX, and they depend on the target library used.

VHDL code:	Verilog code:
process (SEL, A, B, C, D) begin case SEL is when “00”=>OUTC <= A; when “01”=>OUTC <= B; when “10”=>OUTC <= C; when others=> OUTC <= D; end case; end process;	always@(SEL or A or B or C or D) begin case (SEL) 2`b00 : OUTC = A;   2`b01 : OUTC = B; 2`b10 : OUTC = C; default : OUTC = D; endcase end

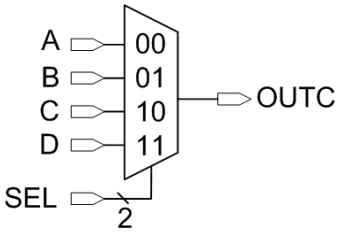


Figure 2-3 Implication of case statements.

2.3 *Loop* statements

- **Unrolling** loops

In synthesis, *for* loops are “unrolled” during translation, and then synthesized. For the code below, its hardware is shown in figure 2-4.

VHDL code:	Verilog code:

Process(a, b)
begin
  for i in 0 to 3 loop
  out(i) <= a(i) and b(3-i);
  end loop;
end process;
	integer i;
	
	always@(a or b) begin
	  for (i = 0; i <= 3; i=i+1)
	Out[i] = a[i] & b[3-i];
	end

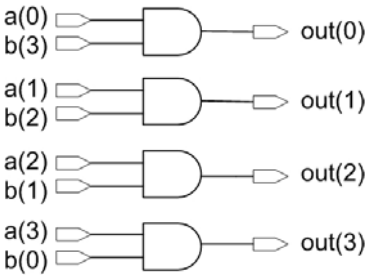


Figure 2-4 Implication of unrolled loop.

- **Tradeoffs** with loops

VHDL code:	Verilog code:

process (data)
variable sum; integer;
 	
begain
sum : = 0;
-- count the 1's
for i in 0 to 7 loop
sum := data(i) + sum;
end loop
 	
-- check parity
odd_parity <= sum mod 2;
end process
	always@(data)
	begin
	
	sum = 0;
	/*count the numer of '1's*/
	for (i = 0 ; i < 8; i = i+1)
	sum = sum + data[i];
	
	/* check if even or odd number */
	odd_parity = sum[0]
	
	end

The hardware for the above code is shown in figure 2-5.

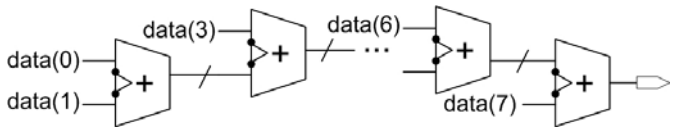


Figure 2-5 Implication of tradeoffs with loop.

- **Recoded** loop

VHDL code:	Verilog code:

process (data)
variable odd-parity : bit;
begin
odd_parity <= '0';
for i in 0 to 7 loop
odd_parity <= data(i) xor odd_parity;
end loop;
end process
	always@(data)
	begin
	for (i = 0;i <= 8; i=i+1)
	odd_parituy = ^data[i];
	end

The hardware implication of the above code is shown in figure 2-6.

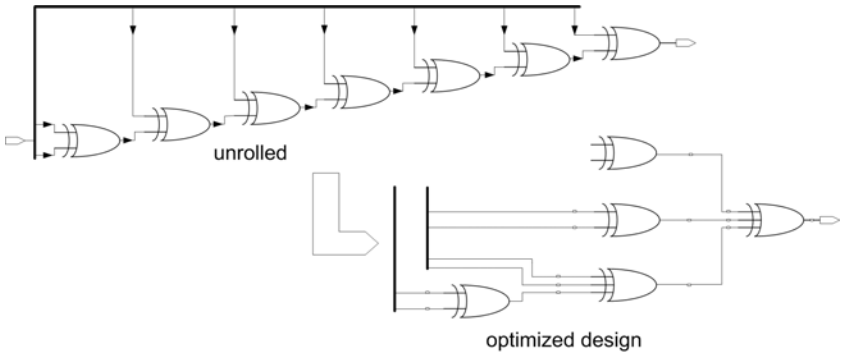


Figure 2-6 Implication of recoded loop

2.4 Partitioning a Design

Partitioning is the process of dividing complex design into smaller parts. Ideally, all partitions would be planned prior to writing any RTL. Initial partitions of a design are defined by RTL, but it can be modified using DC by the commands: *group* and *ungroup*[1]. The followings determine the partitions within the RTL description.

- *Entity* and *module* statements define hierarchical blocks,
- Inference of arithmetic circuits (+, -, *, ..) can create a new level of hierarchy, and *Process* and *Always* statements do not create hierarchy.

The partitioning strategies for synthesis are shown below.

- Don't separate combinational logic across hierarchical boundaries.
- Place hierarchy boundaries at register outputs.
- Size blocks for reasonable runtimes.
- Separate core logic, pads, clocks, asynchronous logic and JTAG.

A design with better partitioning brings

- better results: smaller and faster design,
- easier synthesis process: simplified constraints and scripts, and
- faster compiles: quicker turnaround.

Remember: always plan the partitioning of design prior to start writing RTL code.

2.5 Conclusion

This chapter lists the hardware implications for some statements, and highlights the importance of partitioning a design. Reader should keep these in mind and remember that DC optimizes logic between registers only not the register placement.

[1] Refer to Synopsys Design Compiler document for details.

3. RTL Simulation and Verification with Cadence NCLaunch

Once coded, simulation and verification should be done to verify the code and its functionality. This can be achieved with either cadence tool (NCLaunch) or synopsys tool (VCS). In this manual, NCLaunch is introduced and used.

The arrangement is as follows. In section 3.1, an introduction to NCLaunch software is presented. In section 3.2, a tutorial of using NCLaunch is preformed. The method of debugging a design is given in section 3.3. The conclusion is given in section 3.4. The whole process is demonstrated with a 32 bit adder which is coded with Verilog.

3.1 Introduction to NCLaunch

NCLaunch provides user with a graphical user interface to configure and launch cadence simulation tools: compiler, elaborator and simulator. The following concepts are described in this section, which user should be familiar before running NCLaunch.

- Invoking NCLaunch
- Single-Step and Multi-Step Modes
- Components of NCLaunch
- Exiting NCLaunch
- The NCLaunch Help Menu

3.1.1 Invoking NCLaunch

On UNIX system, invoke NCLaunch with the following commands.

% nclaunch –new[2] (first time)

% nclaunch (afterwards)

When NCLaunch starts for the first time, it prompts user to select a running mode, *single-step* and *multi-step*, as shown in figure 3-1.

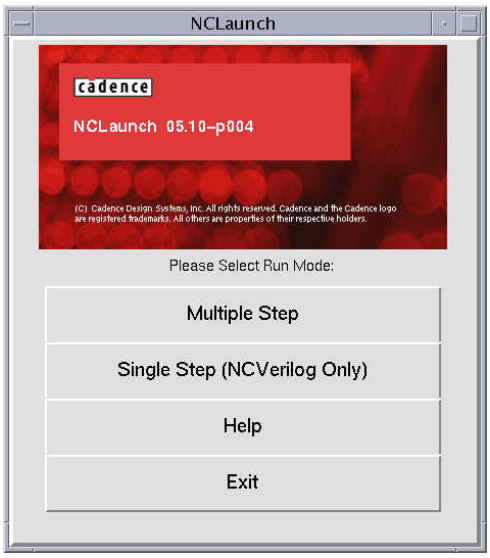


Figure 3-1 Select running mode.

[2] There are more options. Please refer to NCLunch User Guide for details

3.1.2 Single-Step and Multi-Step Modes

NCLaunch allows user to invoke the simulator in one of the following two modes:

- In multi-step mode[3], user performs separate steps to compile source files, elaborate design units, and simulate snapshots for Verilog, VHDL, and mixed-language designs. This gives user greatest control and flexibility to specify simulation options and features. Multiple step mode uses the *ncvlog* and *ncelab* commands to compile and elaborate design.
- In single-step mode[4], user compiles, elaborates, and simulates a design in one step. For designs entirely written in Verilog, this provides an easy way to select NC-Verilog options and run the simulation. Single-step mode creates everything needed to run the NC-Verilog simulator, including all directories, a cds.lib file, and an hdl.var file. Single step mode uses the *ncverilog* command to compile and elaborate design.

User can switch mode at any time by selecting **File→Switch to Multiple Step** or **File→Switch to Single Step**.

3.1.3 Components of NCLaunch

The NCLaunch main window contains a menu bar, toolbar, file browser or design area, and an I/O region. Figures 3-2 and 3-3 show the main window in multi-step mode and single-step mode respectively.

- **Menu Bar** contains the *File*, *Edit*, *Tools*, *Utilities*, *Plug-ins*, and *Help* menu choices.
- **ToolBar** consists of icons that invoke cadence NC simulation tools and utilities. The tool icons give user a shortcut to the tools, as shown in Table 3-1.
- **I/O Region and Status Bar** let user submit batch commands to simulation tools and utilities and view the output of running process. Standard output messages from running processes are displayed in blue and error messages are displayed in red.

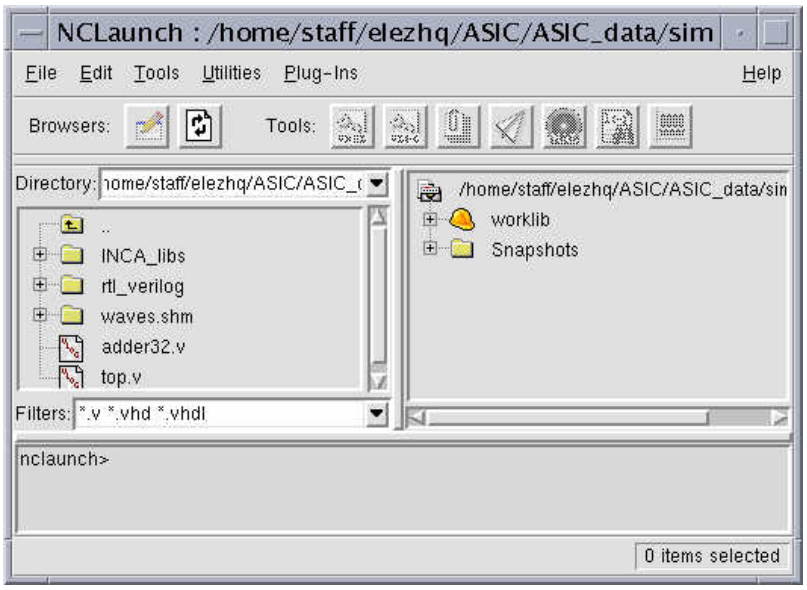


Figure 3-2 NCLaunch main window, multi-step mode.

[3, 4] For more information, refer to the NC-Verilog Simulator Help.

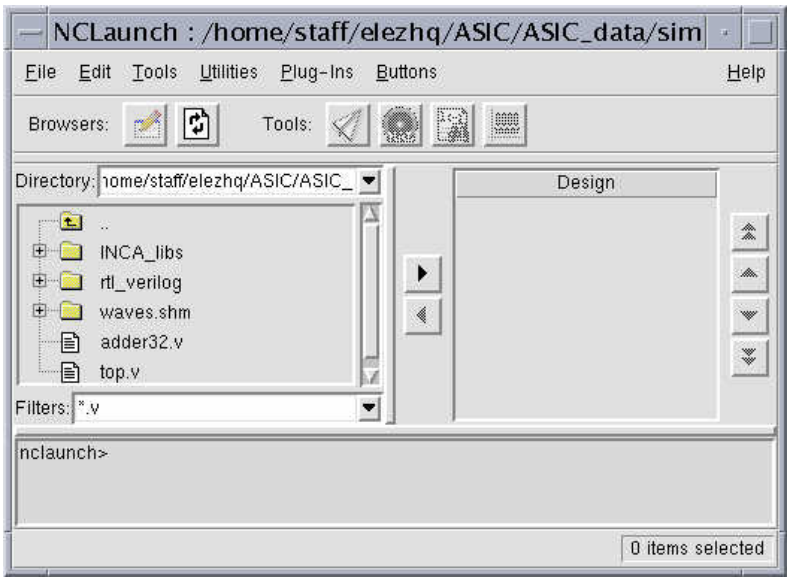


Figure 3-3 NCLaunch main window, single-step mode.

Table 3-1 Icons of the toolbar.

Icons	Function
	Edit File – by selecting a file and clicking on this icon, a text editor appears with the file's contents for review or modification.
	Refresh – Updates user's browser window.
	Compile VHDL Files (multi-step only) – compiles selected VHDL files that appear as design units under user's work library in the Library Browser.
	Compile Verilog Files (multi-step only) – compiles selected Verilog files that appear as design units under user's work library in the Library Browser.
	Elaborate Files (multi-step only) – by selecting the top level design unit and clicking on this icon, user's design is elaborated.
	Run Simulation – starts a simulation of selected design.
	Launch analysis & lint with current selection
	Browser Logfiles – launches the NCBrowse message browser to analyze selected log files.
	Waveform Viewer – starts the SimVision analysis environment with selected database files.

Figure 3-4 I/O region.

3.1.4 Exiting NCLaunch

To exit NCLaunch, select **File→Exit**.

Exiting the application does not terminate any batch jobs that user has already launched. On exit, NCLaunch saves general default settings to user's home directory, and saves design default setting to user's current working directory.

3.1.5 Environment Setup

User can run NCLaunch in one of the two modes, single step or multiple step. As mentioned in section 3.1.2, single step mode provides all the necessary setup files (cds.lib and hdl.var), while multiple step mode creates those setup files through a few steps of setting. A sample of setting environment is shown in section 3.2.1.

3.2 Tutorial of Using NCLaunch

The sample used here is a 32 bit adder, and its source code *adder32.v* and *test_adder.v* are listed in tables 3-2 and 3-3 respectively. The file *test_adder.v* which tests the function of the adder32.v is the top design.

Table 3-2 Source code of 32 bit adder.

```
//file: ~/project/rtl_verilog/adder32.v
module adder32 (a, b, cin, CLOCK, sum, cout);
input [31:0] a, b;
input cin, CLOCK;
output [31:0] sum;
output cout;
reg [31:0] sum;
reg cout;
reg [32:0] temp;
always @(a or b or cin)
begin
temp=a+b+cin;
end

always @(posedge CLOCK)
begin
{cout, sum}<=temp[32:0];
end

endmodule
```

Table 3-3 Test bench of the 32 bit adder.

```
//file: ~/project/rtl_verilog/test_adder.v
module test_adder;
reg [31:0] a, b;
reg cin, CLOCK;
wire [31:0] sum;
wire cout;
adder32 block1(a, b, cin, CLOCK, sum, cout);

//create a clock with a cycle of 100ns
initial
begin
CLOCK = 1'b0;
forever #50 CLOCK= ~CLOCK;
end

initial
begin
cin= 1'b1;
a = 32'h0000;
b = 32'h0000;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'h00000000;
    b = 32'h0000ffff;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'h0000ffff;
    b = 32'h00000000;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'h0000ffff;
    b = 32'h0000ffff;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'h00000000;
    b = 32'hffff0000;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'hffff0000;
    b = 32'h00000000;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'h0000ffff;
    b = 32'hffffffff;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'hffffffff;
    b = 32'hffffffff;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
#100 a = 32'h00000000;
    b = 32'h00000000;
      $display("%d a=%h b+%h cin+%h sum+%h cout+%h", $time, a, b, cin, sum, cout);
end

//finish the simulation at time 1000ns
initial
begin
#10000 $finish;
end
```

3.2.1 Starting NCLaunch

1. Start NCLaunch with the following command in the directory ~/project/rtl_verilog where the source files are placed.
   % nclaunch –new&
   where *–new* specifies that this is a new design. The command *nclaunch* should be used if it is not a new design. At startup, NCLaunch displays a list of modes which users can choose, as shown in figure 3-1.
2. Click on **Multiple Step**.
   As this is a new design, user must define a cds.lib and work library. NCLaunch opens the **Set Design Directory** form as figure 3-6. Do step 3 if the **Set Design Directory** form doesn't appear, otherwise skip step 3.
3. Choose **File→Set Design Directory…** from Nclaunch main window as shown in figure 3-5. The **Set Design Directory** form appears as shown in figure 3-6.

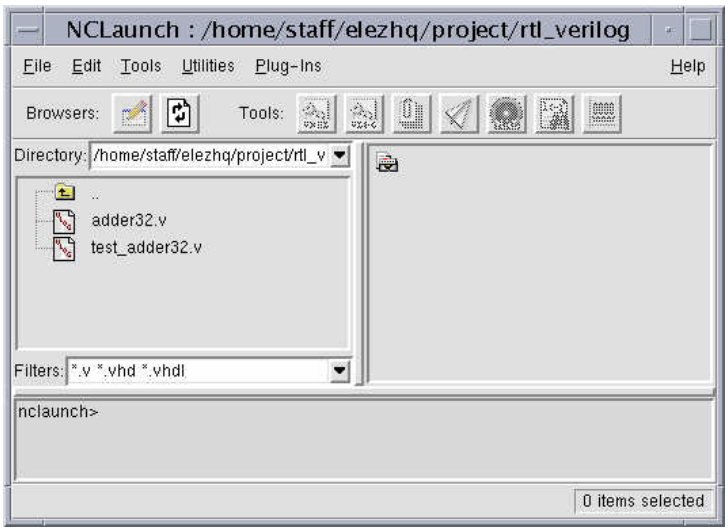


Figure 3-5 NCLaunch main window.

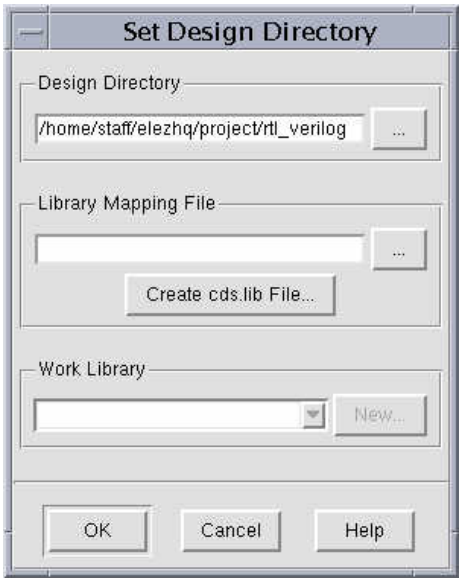


Figure 3-6 Set Design Directory form.

4. On the Set Design Directory form, click on the **Create cds.lib File** button under the **Library Mapping File** field.This opens the **Create a cds.lib file** form as shown below.

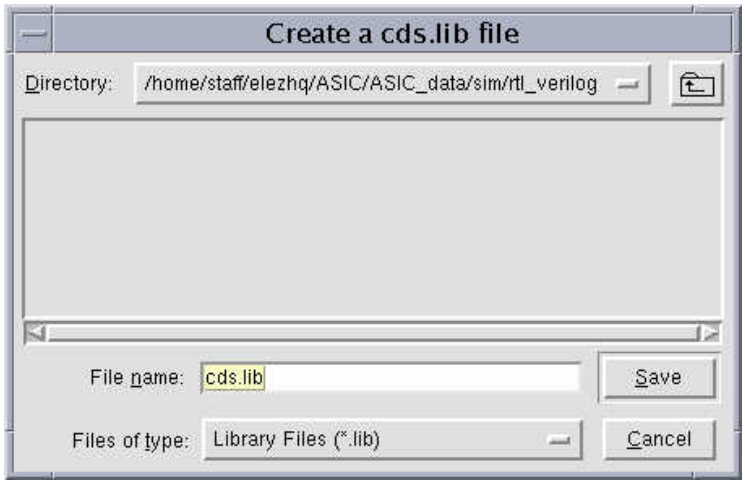


Figure 3-7 Create a cds.lib file form.

5. Click on **Save** to create a library mapping file with the default name - cds.lib. NCLaunch opens the **New cds.lib File** form, as shown in figure 3-8. This form lets user pick the libraries that the user wants to use.
   - For Verilog files, choose **Don't include any libraries**.
   - For VHDL and mixed-language designs, choose either the **default libraries** or **the IEEE pure libraries.**

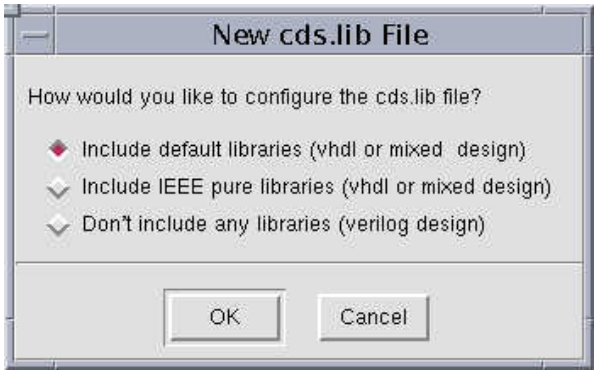


Figure 3-8 New cds.lib File Form.

6. Click on **OK** to close the New cds.lib File form. NCLaunch displays the main window as shown in figure 3-9.

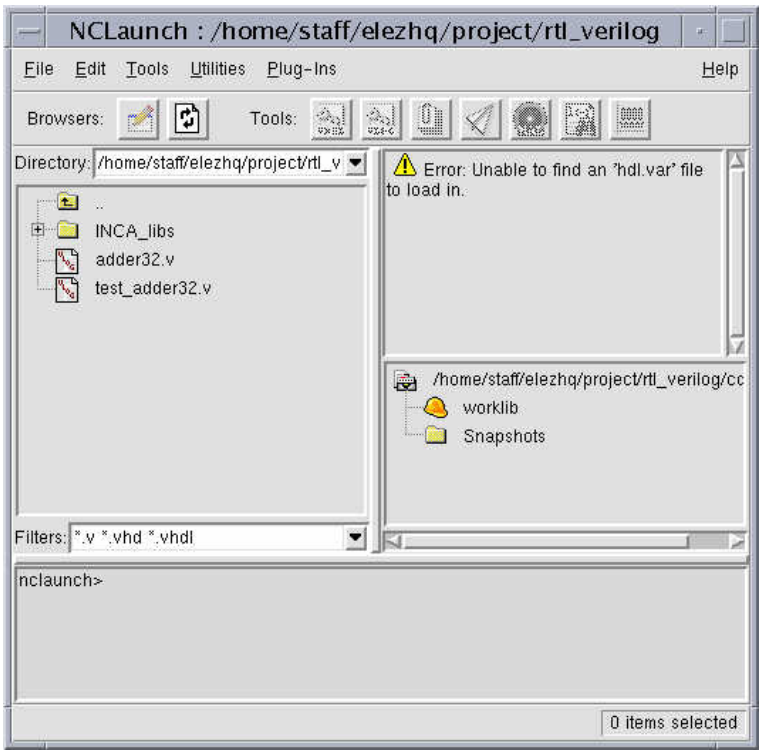


Figure 3-9 NCLaunch Main Window.

3.2.2 Compiling and Elaborating the Design

Before simulating the design, user must

- Compile the source files using the Verilog Compiler[5], and
- Elaborate the design into a snapshot using the elaborator.

A snapshot is the representation of the design that the simulator uses. The NCLaunch main window gives user access to the tools which user needs when compiling and elaborating design, as well as to several utilities. User accesses the tools and utilities by using the **Tools** or **Utilities** menu or clicking on the appropriate button on the **toolbar**[6].

The steps of compiling and elaborating are introduced in the following sub-sections.

3.2.2.1 Compiling the Design

1. Select the Verilog files that make up the design: adder32.v and test_adder.v. To select multiple files, hold down the control key and click on each filename.
2. Click on the **Verilog Compiler** button.
   The I/O area at the bottom of the window displays the *ncvlog* command that runs, and it displays the messages that the compiler generates as it compiles the design files.

3.2.2.2 Elaborating the Design

To elaborate a design, user typically expands the work library (**worklib**), select the top-level design unit, and then click on the **Elaborate** button.

---

[5] Use VHDL compiler if the source code is written in VHDL.
[6] Refer to section 3.1 for the Tools or Utilities or toolbar.

1. Expand the work library (**worklib**) by clicking on the **plus sign** next to the hardhat icon.
2. Expand the **top-level design unit**. In this example, the top-level is the Verilog testbench, **test_adder**.
3. Select the **module**.
4. Choose **Tools→Elaborator** to open the **Elaborate** form which is shown in figure 3-10.

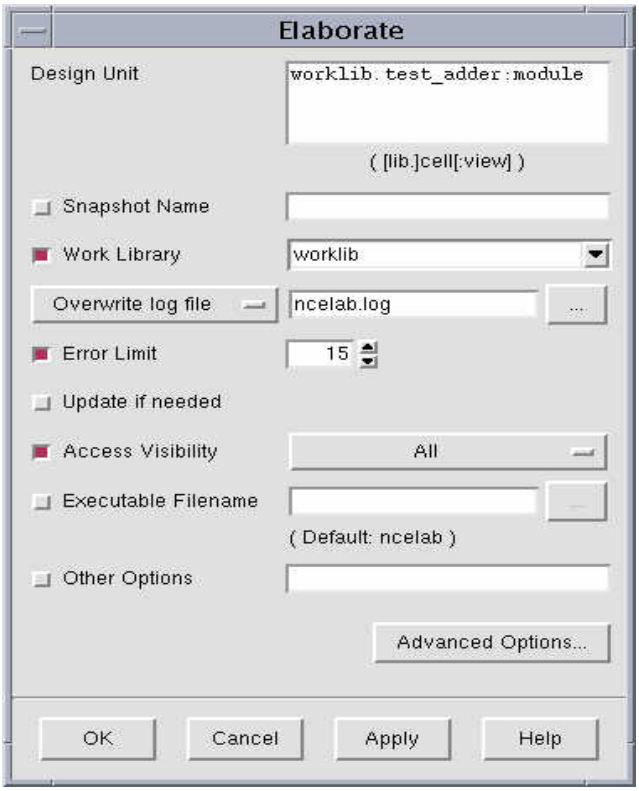


Figure 3-10 Elaborate Form.

Notice that the **Access Visibility** button is selected and that the value is set to **All**. This option provides full access (read, write, and connectivity access) to simulation objects so that user can probe objects and scopes to a simulation database and debug the design.

5. Enable the **Other Options** button and enter **-timescale 1ns/1ns** in the text field.
6. Click on **Ok** to elaborate the design.
   The I/O area at the bottom of the window displays the *ncelab* command that runs, and it displays the messages that the elaborator generates.

3.2.3 Starting the Simulator

To start the simulator:

1. Expand the **Snapshots** folder to display the snapshots that are available in the design library.
2. Select the **snapshot**, as shown in figure 3-11.

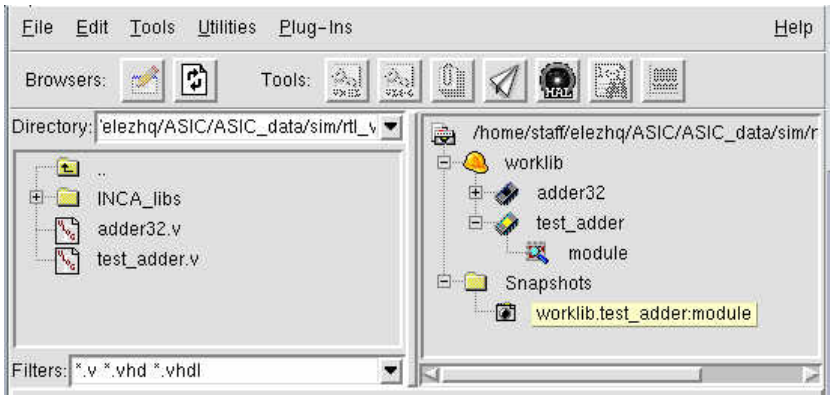


Figure 3-11 Selecting the snapshot.

3. Click on the **Simulator** button.
   The **Design Browser** and the **Console** window appear. User can access design hierarchy in the **Design Browser,** and enter '**SimVision** and **simulator'** commands in the **Console** window.

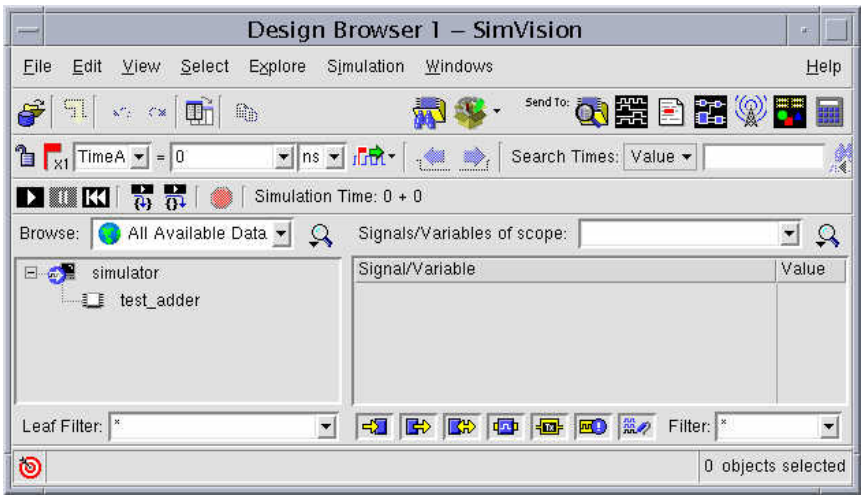


Figure 3-12 Design Browser.

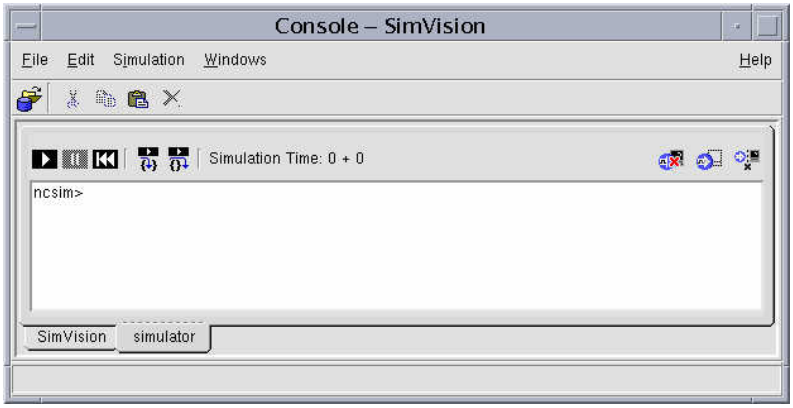


Figure 3-13 Console Window.

In the **Design Browser** sidebar on the left side of the window, **SimVision** places the simulation at the top of the hierarchy and assigns it the name **simulator**. The top-level of the design hierarchy is placed below the simulator. In this example, it is named test_adder.

At startup, the **Console** window has two tabs, as shown in figure 3-13. The **SimVision** tab lets users enter *SimVision* commands and the **simulator** tab lets users enter *simulator* commands. As simulation running, the **Console** window also displays messages from *SimVision* and the *simulator*.

4. After invoking the simulator, user can exit NCLaunch[7]. To **exit** NCLaunch, bring the NCLaunch main window to the foreground and choose **File→Exit** from the menu bar.

3.2.4 Simulating the Design

SimVision lets user choose the simulation data that user wants to save for particular objects or scopes. This can help to keep the size of simulation data files as small as possible. At a later time, user can load a simulation data file back into the Waveform window and re-examine the simulation results.

This section describes how to select simulation data to save and how to run the simulation.

3.2.4.1 Selecting the Simulation Data[8] to Save

User can save simulation data by probing the design during simulation and saving the values of the probed objects to a database. There are two types of probe commands:

- Probe[9] a specific object or objects. The values of the specified objects are saved in the database
- Probe a scope or scopes. Users can choose the type of information to save, such as the inputs to that scope, and can choose whether to probe some or all subscopes.

To probe all objects in all scopes, begin at the top module as follows.

1. In the **Design Browser**, click on the **+** ***icon*** next to top:test_adder to expand the hierarchy.
2. Select the **top** scope. The signal list on the right side of the window displays the signals for the *top* scope, as shown in figure 3-14. The signal list indicates the type of each signal – input, output, inout, internal signal, or transaction. User can use the **Leaf Filter** to choose signals which are intended to view.

[7] For more information about NCLaunch, user may refer to the NCLaunch User Guide.
[8] User can create different simulation database for individual components of design to help debugging, referring to "Managing Simulation Database" in the SimVision User Guide.
[9] About enabling, disabling, and deleting probes, or creating new probes, please refer to "Creating and Managing Probes" in SimVision User Guide.

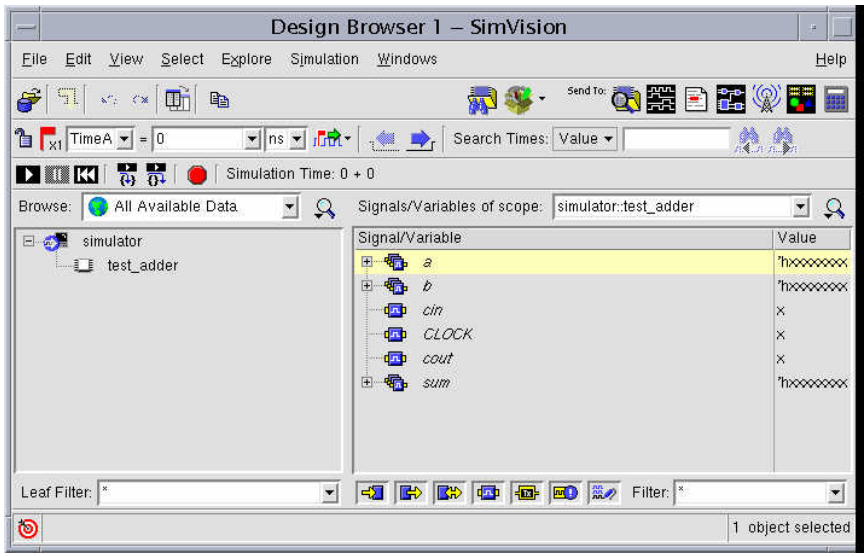


Figure 3-14 Choosing the top scope.

3. **Choose Simulation → Create Probe…** from the **menu bar**.
   SimVision opens the **Create Probe** form. This form lets user probe one or more levels of *subscope*, choose the type of signals that user wants to probe, and write the probed information to any database.

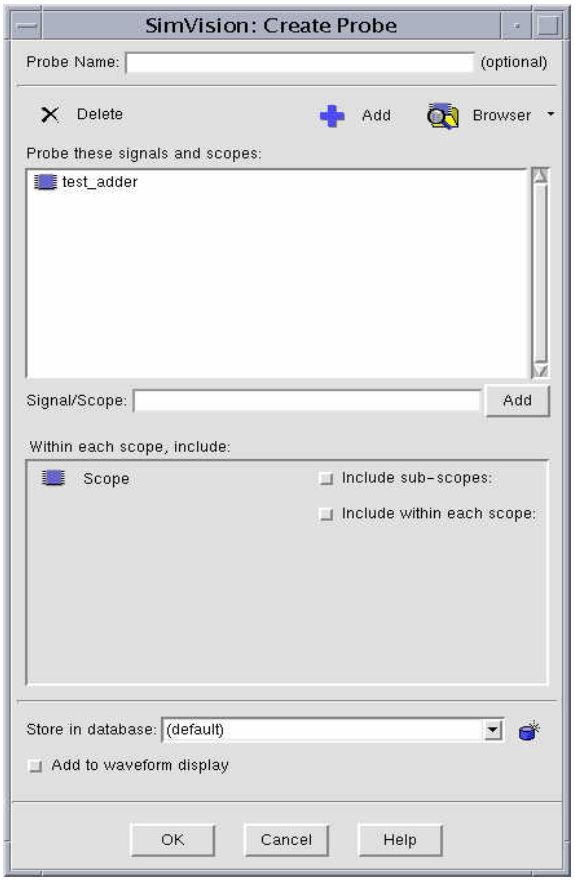


Figure 3-15 Create Probe form.

4. For this probe[10]:
   - Select **Include sub-scopes** and choose **all** from the drop-down list to include all the sub-scopes in the design.
   - Select **Include within each scope** and choose **all** from the drop-down list to include all inputs, outputs, and ports.
   - Deselect **Add** to waveform display.

   The form should have the settings shown in figure 3-16.

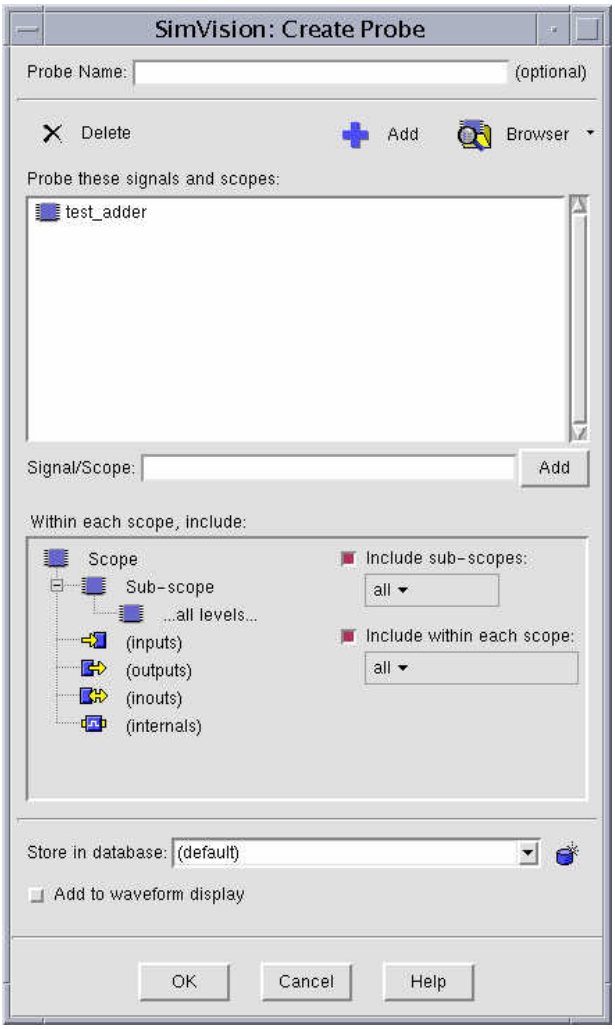


Figure 3-16 Create Probe form.

5. Click on **Ok** to close the **Create Probe** form.

3.2.4.2 Running the Simulation

To run the simulation:

1. From the **SimVision** window, choose **Simulation→Run**. SimVision simulates the design and saves the simulation data in a default database. As it runs, the simulator displays the following messages in the Console window.

[10] There are other ways to create probe(s). Please refer to NC-Verilog Simulator Help for more information.

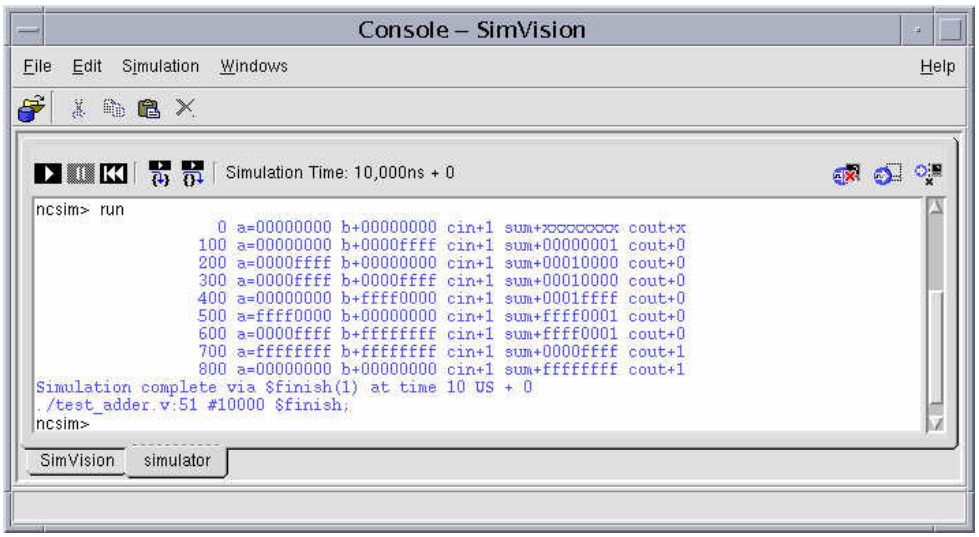


Figure 3-17 Messages of Console window.

Note:

- After completed these steps, user's working directory should contain a new directory named waves.shm. The waves.shm directory should contain two files – waves.dsn and waves.trn.
- To correct any problems if there was, restart the simulator by choosing **Simulation→Reinvoke Simulator** from the **Console** window.

3.2.5 Displaying Simulation Data[11]

Waveforms show the values of signals at any time during simulation. They can help user to understand the behavior of the design. To open a waveform window:

- Deselect[12] the **top** scope in the **Design Browser** sidebar, and then click on the **Waveform** button in the **Send to** toolbar. User can deselect the scope by pressing Control while clicking on the selected scope.

SimVision opens a blank Waveform window, as shown in figure 3-18.

[11] For more information about managing Waveform window and displaying signals, refer to SimVision User Guide.

[12] If user selects a scope before clicking the Waveform button, all of the signals in that scope are added to the Waveform window. If the scope has many signals, this may add more signals to the window, and it may take a long time to load all of those signals and their waveforms into the window.

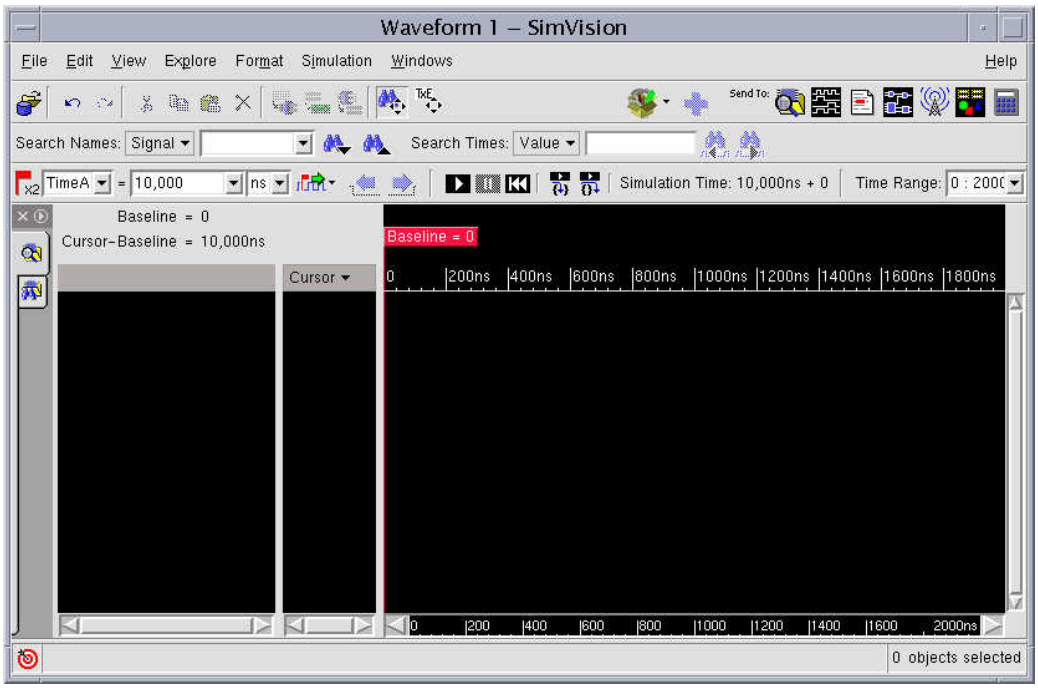


Figure 3-18 Opening a blank waveform window.

3.2.5.1 Selecting the Signals to Display

In the **Design Browser** sidebar, user can select objects from one scope at a time and send them to the Waveform window.

To select the signals that user wants to display in the Waveform window:

1. Expand the **Design Browser sidebar** by clicking on the **Expand** button in the sidebar tab of **Waveform** window. SimVision adds the sidebar to the window, as shown in figure 3-19.

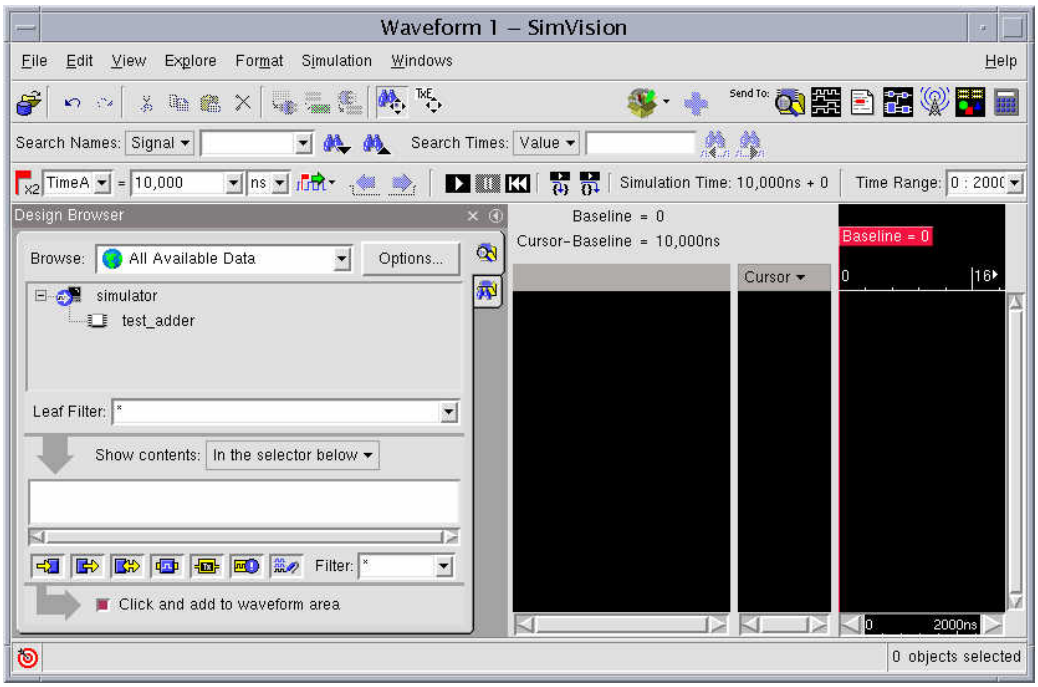


Figure 3-19 Expanding the design browser sidebar.

2. **Expend** the **test_adder** scope by clicking on the button (If there are subscopes under the **top**, clicking on the +button to expand design, and then select a scope.) The sidebar displays the signals for that scope in the selector area, as shown in figure 3-20.

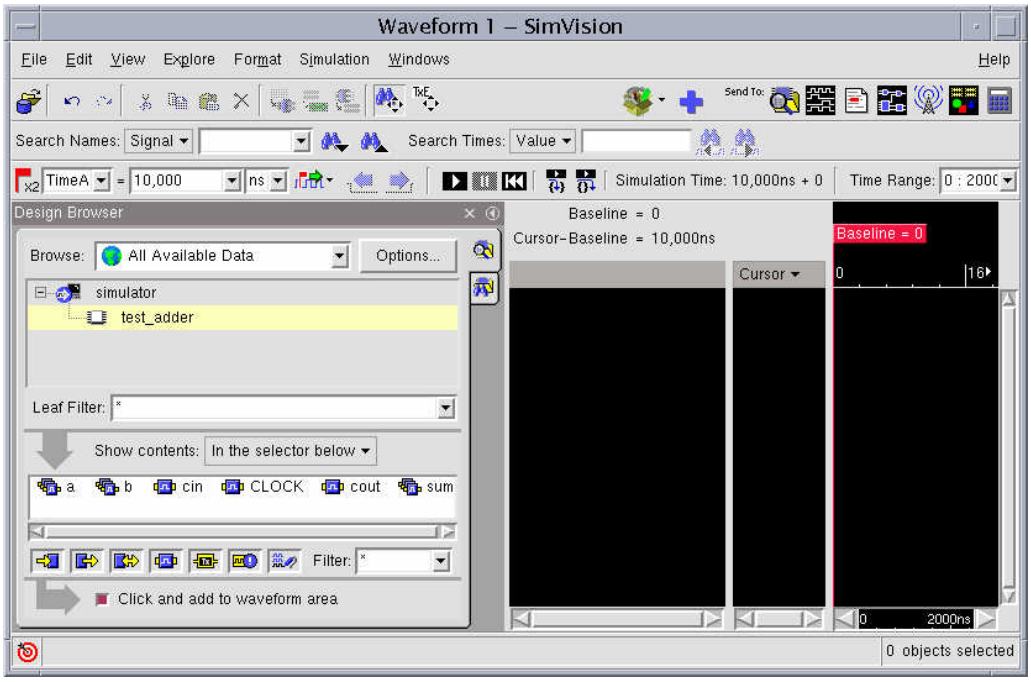


Figure 3-20 Showing the contents of the slected scope.

3. In the **selector area**, select the signals that user wants to add to the Waveform window. For this example select all the signals.

4. **Collapse** the sidebar by clicking on the **Collapse** button.
   The Waveform window displays the signals and waveforms, as shown in figure 3-21. Signal names and values are displayed on the left; their waveforms are displayed on the right.

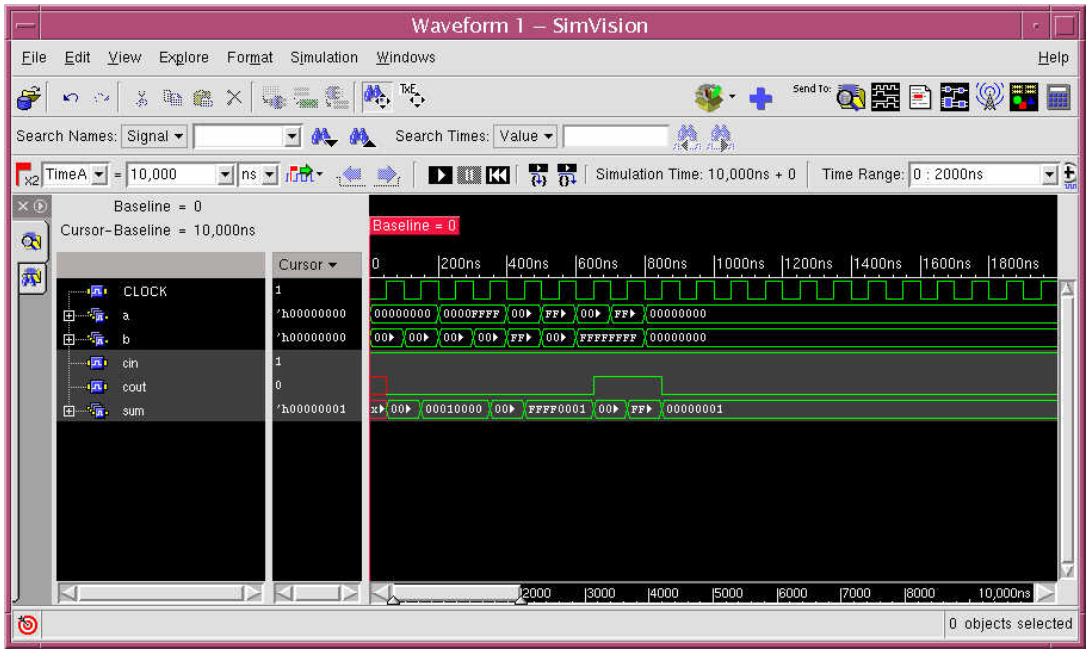


Figure 3-21 Displaying Data in the waveform window.

Note: SimVision adds the signals in the order in which user selects them, but user can rearrange them. Select the signal that user wants to move, and then press and hold the middle mouse button. As moving the cursor, SimVision displays a red insertion bar. Place the insertion bar where the user wants the signal to appear, and release the mouse button.

3.2.5.2 Moving through Simulation Time

Above the waveform data, user can see the beginning and ending times for the simulation data currently displayed. Below the waveform data, the scroll bar shows the entire simulation time. User can adjust the amount of waveform data displayed in the window by entering a new time range.

To enter a new time range:

1. For this example, enter 0:1000ns, as shown in figure 3-22, and press **Return** to apply the time range.

Figure 3-22 Entering a new time range.

2. Save these settings by selecting **Keep this range** from the **Time drop-down menu** as shown in figure 3-22, to keep the time range.
3. At any time, user can quickly return to the view by selecting it from the drop-down list.

3.2.5.3 Moving the Cursors

The **Waveform** window contains two cursors, named **TimeA** and **Baseline**. User can move these cursors to any point in simulation time and use them as reference points. User can also create any number of additional cursors. However, for this example, user needs only these two.

To move a cursor:

- Either dragging the cursor to the desired time or entering a simulation time in the cursor time text field. For this example, change the simulation time of ***TimeA*** to 800ns, as shown in figure 3-23.

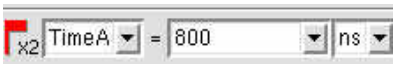


Figure 3-23 Setting the cursor time.

3.3 Debugging a Design

After analyzing the waveform data, user is able to know if it is correct. If an error happens, debugging the design is needed. Normally 4 steps have to be done to debug a design:

- Search for conditions;
- Analyze the waveform data;
- Locate an error;
- Correct source code.

The debugging method is described below. The example used is the adder32, too. The following description serves demonstration purpose only, as there is no error with the adder32 RTL code.

3.3.1 Searching for Conditions

A condition is a combination of signal values that user wants to search for in the **Waveform** window. For example, a condition that occurs whenever **cout** and **CLOCK** have the value 1, as follows:

1. Select the signals of the **cout** and **CLOCK** in the **Waveform** window and click on the **Next Edge** button until both signals have value 1.
2. Choose **Edit →Create →Condition**. SimVision opens the **Expression Calculator**, as shown in figure 3-24.

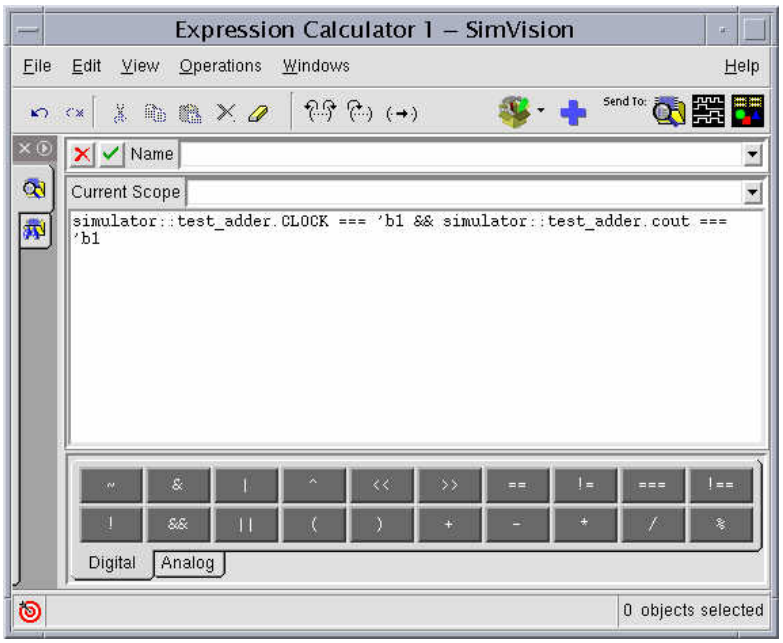


Figure 3-24 Expression Calculator.

The Expression Calculator creates a default **AND** expression. This expression is true whenever both signals have the value 1. User can edit the expression if a different condition is to be investigated.

3. Enter a name for the condition expression in the **Name** field such as cout_and_CLOCK.
4. Click on the **Waveform** button to add the condition to the Waveform window. It can be treated as a normal signal when a condition is added to the Waveform window.
5. Choose **File→Close Window** to close the **Expression Calculator** on the Expression Calculator window.
6. Search for the expression to locate where the condition occurs on the **Waveform** window.

3.3.2 Analyzing Simulation Data in the *Waveform* Window

Analyzing waveform can help user to find problems in a design. For example:

1. Set the simulation time to 800ns, as shown in section 3.2.5. There are a few ways to set the simulation time:
   - **Enter** the desired time in the cursor time field.
   - **Drag** the primary cursor until it reaches the desired time.
   - **Select** the signal to be viewed and click on the **Next Edge** button until it reaches the desired time.
2. Select a condition to be investigated and click on the **Next Edge** button to follow the sequence of events from clock cycle to clock cycle.
3. Locate the error while clicking on the Next Edge.
4. From the **Time** field, choose **Keep this View** from the drop-down list so that it is easy to go back to this view later.

3.3.3 Analyzing Simulation Data in the *Register* Window

Another way to analyze simulation results is through the **Register** window, where user can create custom views of the simulation data, including freeform text and graphical elements. A **Register** window can have several pages, each with its own view.

To create a page in a **Registe**r window:

1. From the **Waveform** window, select the signals to be analyzed, such as the a, b, cin, cout, CLOCK and sum.
2. Click on the **Register** button to send these signals to a **Register** window, as shown in figure 3-25.

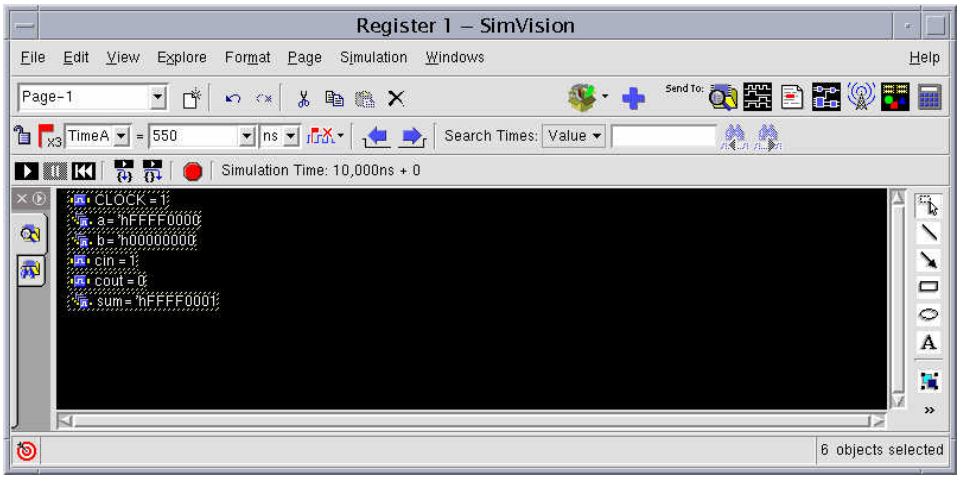


Figure 3-25 Adding signals to a Register window.

Along the right side of the window are buttons that let user draw graphical objects, add text, and manage the layout of the objects in the window. **Tool** tips pop up when placing the cursor over these buttons, telling user what functions they perform.

3. Arrange the objects any way that user likes. For example, the layout in figure 3-26 shows the relationship between the inputs and outputs.

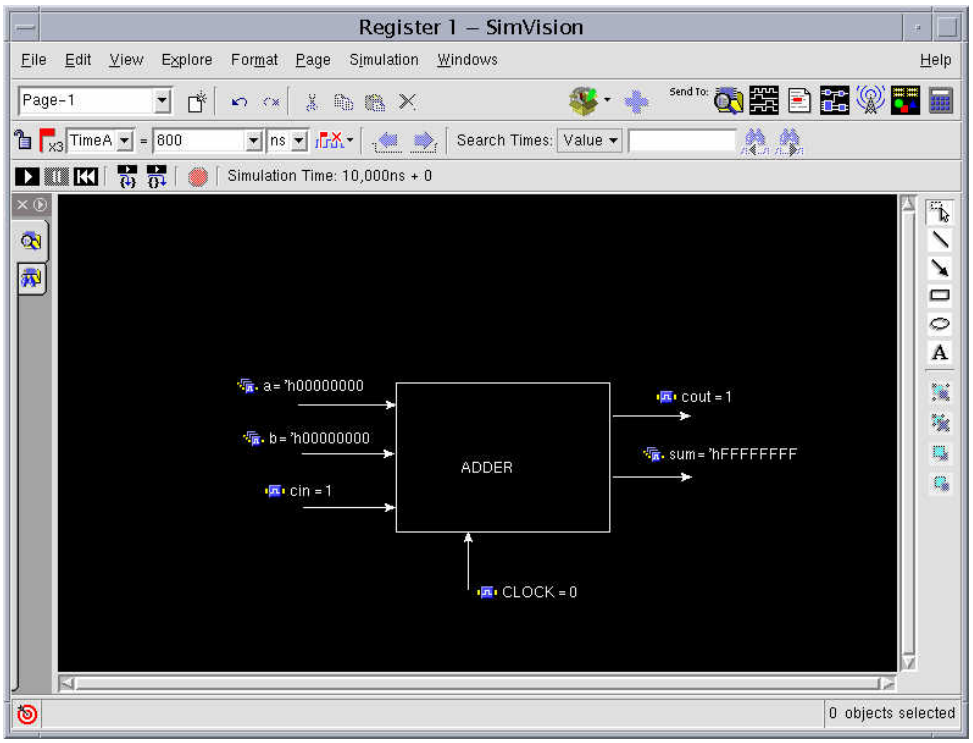


Figure 3-26 A custom layout.

4. Enter a simulation time, such as 600ns in the **Cursor TimeA** field. The Register window updates the signals to show the respective values at that time.

5. Select a signal, such as cout and click on the **Next Edge** button. The time progresses to the next edge of that signal and the **Register** window updates all of the signals to show the values at that time.

6. Click on the **Previous Edge** button to move the simulation time back to the previous edge of the selected signal.

3.3.4 Fixing an Error in the Source Code

User can use the **Signal Flow Browser** and **Source Browser** to locate the line in the source file where an error occurs.

1. In the **Waveform** window, select the variable or signal, cout, and choose **Explore→Go To→ Cause**. The **Signal Flow Browser** shows the signal user selected and the list of the signal's drivers, as shown in figure 3-27.

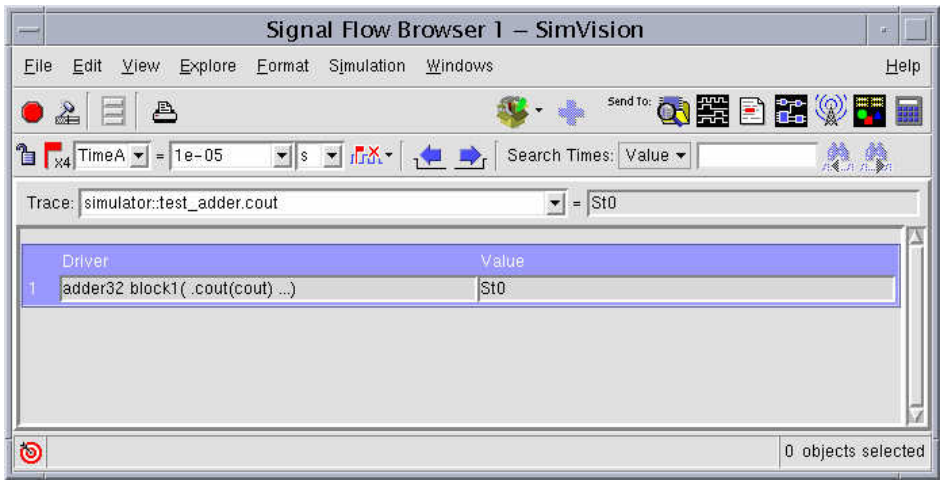


Figure 3-27 Displaying drivers in the Signal Flow Browser.

2. Open **Source Browser** by clicking on **Windows→New→Source Browser**.
3. From the **Signal Flow Browser window**, select the first driver. The Source Browser now points to the line in the source file where the first driver is set to, as shown in figure 3-28.

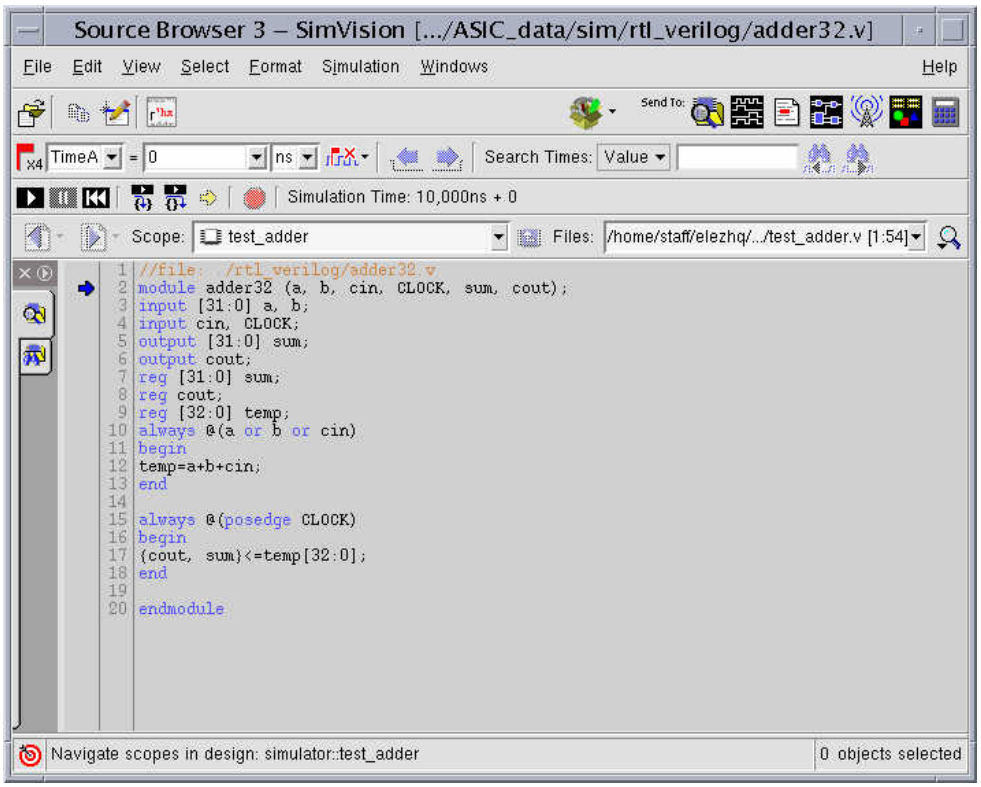


Figure 3-28 Displaying source code in the source Browser.

4. To fix an error in the design, choose **Edit→Edit File** from the **Source Browser** and make the necessary changes.
5. Save these changes to the source file.

6. Choose **Simulation→Reinvoke Simulator** from the Source browser. if the **Reinvoke** dialog box appears, click on **Yes**.
   SimVision compiles and elaborates the design, and restarts the simulator. All of the SimVision windows that were opened in the previous session are opened again.
7. In any **SimVision** window, choose **Simulation→Run** to generate new simulation data.
8. In the **Waveform** window, display the view previously saved. It can be seen that the result is correct now.

3.3.5 Ending a SimVision Session

To exit SimVision:

1. Choose **File→Exit** SimVision from any **SimVision** window.
2. If the **Waveform** window remains open, choose **File→Exit** SimVision from the **Waveform** window. SimVision displays a confirmation message.
3. Click on **Yes** to exit and close all SimVision windows.

3.4 Conclusion

In this chapter, the usage of NCLaunch is described. User can follow the steps listed in section 3.2 to compile, elaborate and simulate a design. If the results of design were incorrect, user can follow section 3.3 to debug the design and locate errors, and then re-run the design.

Next, the design can be brought to synopsys chip synthesis for synthesis and optimization.

4. Logic Synthesis and Optimization Using Synopsys Chip Synthesis (Design Compiler)

Synthesis is the transformation of an idea modeled with RTL code into a manufacturable device to carry out an intended function. Optimization means to compile a design with the design constraint file which is a description file of design specifications. If user has a design modeled with RTL code, either Verilog or VHDL, and the code has been verified with the NCLaunch previously described, it is time to use the tool - chip synthesis to synthesize and optimize the design and to get a gate level netlist of the design. The usage of chip synthesis is described in this chapter.

The arrangement of this chapter is as follows. The introduction to synthesis and optimization is presented in section 4.1. The preparations for using design compiler (DC) are described in section 4.2. Methods to fix violations are listed in section 4.3. A tutorial of using DC is shown in section 4.4. Lastly, conclusion is given in section 4.5.

4.1 Introduction to Synthesis and Optimization

Figure 4-1 shows the synthesis and optimization flow using DC, and figure 4-2 shows the project directory structure correspondingly to figure 4-1. Synthesis and optimization is an iteration process. As figure 4-1 shows, user needs to do the followings to finish an iteration of synthesis and optimization.

- Edit a *.synopsys_dc.setup* file – a set up file for DC.
- Edit a *constraint* file – a file including design specifications.
- Synthesize and optimize a design with the constraints.
- Generate and check reports to ensure that it meets the design target or specifications.

After each iteration, user needs to decide whether modifying constraint file or RTL code by checking the reports if the design result is not satisfactory, referring to figure 4-1.

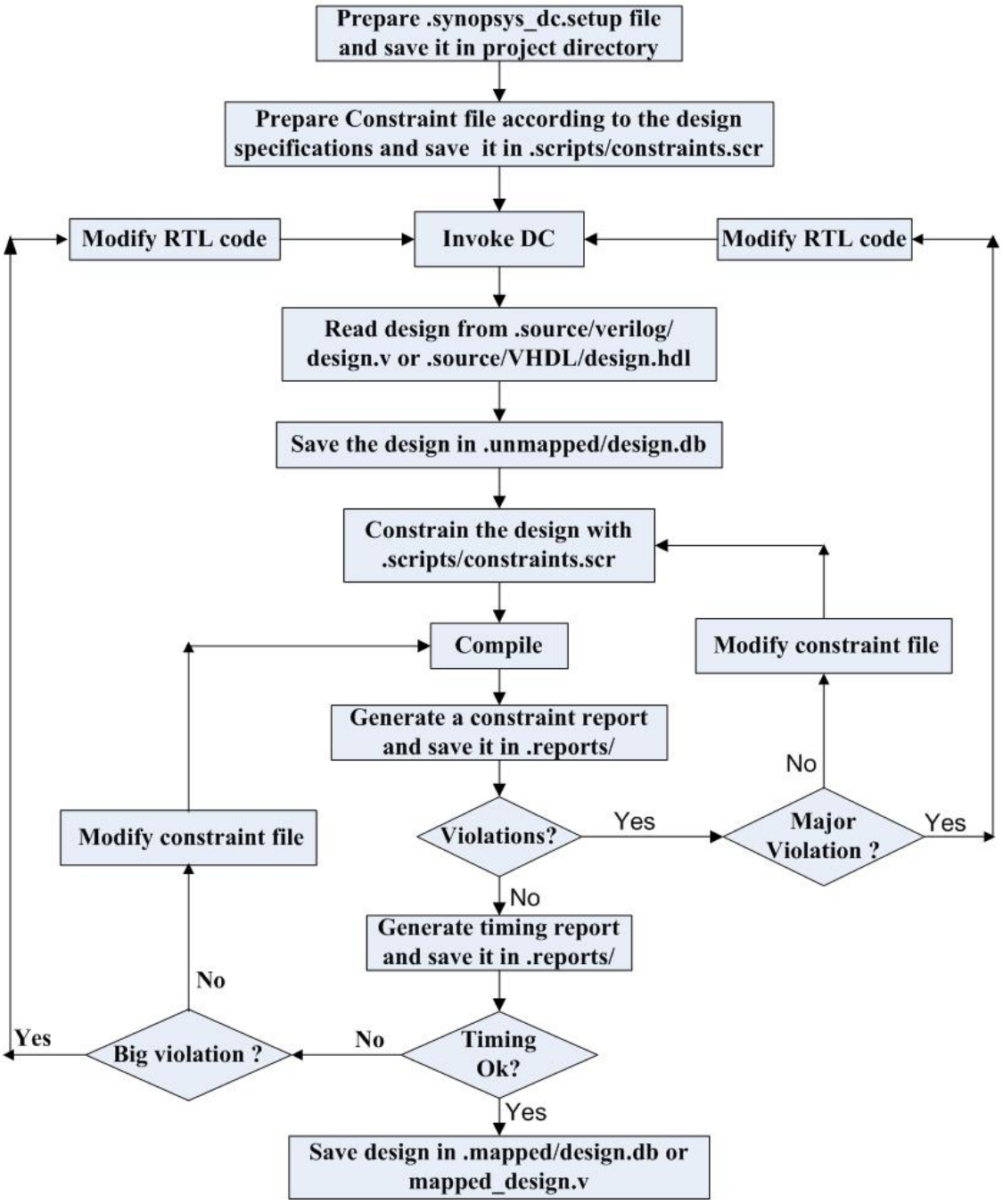


Figure 4-1 Synthesis and optimization flow.

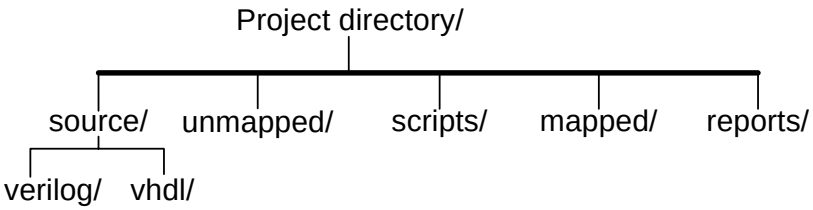


Figure 4-2 Project directory structure.

4.2 Preparations for Using Design Compiler

To use DC, user must know the followings.

- The prescriptions of the *.synopsys_dc.setup* file.
- The prescriptions of the *constraint* file.
- How to synthesize and optimize a design.
- How to generate and check reports.

These are described in this section. User has to understand them in order to synthesis and optimize design.

4.2.1 Prescriptions of the *.synopsys_dc.setup* File

The *.synopsys_dc.setup* consists of four basic commands as shown in table 4-1, where there are four variables: target library, link library, symbol library and search path. The definitions of the variables are as follows.

- **target_library -** the library used by DC for building a circuit.
- **link_library** – the library used to resolve netlist leaf-cells and sub-design references.
- **symbol_library** – defining the symbol library used by DC.
- **search_path -** defining the path where DC will search.

These variables are reserved for DC. During mapping, DC will choose functionally-correct gates from target library and calculate the timing of the circuit using vendor-supplied timing data for these gates. Referring to the second command of table 4-1, * represents DC memory. During link, DC searches the memory first and then reads the library files specified by the *link_library* variable, and DC also searches all UNIX directories defined by the *search_path* variable.

Table 4-1 *.synopsys_dc.setup* file.

set	target_library	“tech_library.db”
set	link_library	“* tech_library.db”
set	symbol_library	“tech_library.sdb”
set	search_path	“$search_path ./unmapped”

4.2.2 Prescriptions of *Constraint* File

In order to obtain optimum results from DC, designers have to methodically constrain their designs by describing the design environment, target objectives and design rules. The constraint file may contain timing and /or area information, usually derived from design specifications. DC uses these constraints to perform synthesis and tries to optimize the design with the aim of meeting target objectives.

The constraint file contains the considerations of three aspects:

- Timing Goals;
- Environmental Attributes;
- Design rules and area requirement.

These are briefly explained below.

4.2.2.1 Timing Goals

Timing goals define the timing constraints for all paths within a design, which include all input logic paths, the internal (register to register) paths, and all output paths with respect to clock. The categories and the relative commands are shown in figure 4-3. The usage and definitions of some of the commands are given below, and others can be found in DC document.

- **create_clock** defines clock source and clock period.
  Format: create_clock[13] –period *value_in_time(ns)* [get-ports *port-name*]
- **set_dont_touch_network** tells DC not to "buffer up" the clock net, even when the flip-flops load to high.
  Format: set_dont_touch_network [get-ports *port-name*]
- **set_clock_uncertainty**[14] models clock skew which is defined as the delay difference between the clock network branches. Figure 4-4 shows the clock skew which is labeled as Tu.
  Format: set_clock_uncertainty –setup *Tu_in_time (ns)* [get_clocks *clock-name*]
- **set_input_delay** constrains input paths.
  Format: set_input_delay –max or -min *value_in_time(ns)* –clock *reference-clock-name* [get_ports *ports-name*]
- **set_output_delay** defines the time it takes from the data to be available before the clock edge.
  Format: set_output_delay –max or -min *value_in_time(ns)* –clock *reference-clcok-name* [get_ports *output-port-name*]

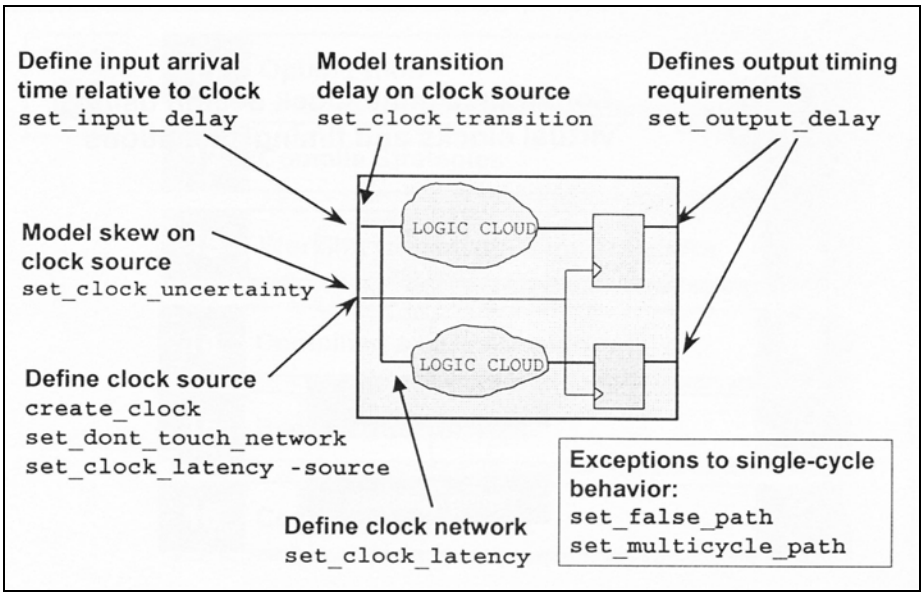


Figure 4-3 Timing goals.

[13] For multiple synchronous clocks, the format is as same as that of single clock. User may refer to DC document for more information.
[14] It also defines hold time requirements, referring to DC document for more information.

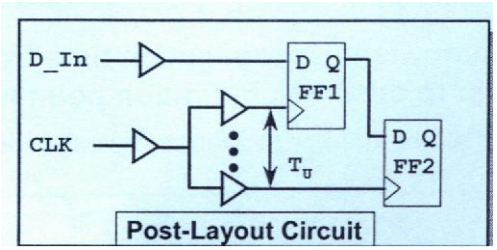


Figure 4-4 Diagram of clock skew - Tu.

4.2.2.2 Environmental Attributes

The environmental attributes define I/O port attributes and wire load models of a design, referring to figure 4-5. The definitions of the commands are given below.

- **set_load** specifies a load capacitance value on an output port.
  Format: set_load *value*[15] [get_ports *output-port-name*]
- **set_driving_cell** specifies a realistic external cell driving the input ports.
  Format: set_driving_cell –lib_cell *cell-name* –pin *pin-name* [get_ports *port-name*]
- **set_wire_load_model**[16] specifies wire load model used for gate connection in DC.
  Format: set_wire_load_model –name *wire-name*[17]
- **set_wire_load_mode** specifies what wire load mode to use for nets that cross hierarchical boundaries.
  Format: set_wire_load_mode enclosed[18] (or top)
- **set_operating_conditions** specifies the synthesis operating condition[19]: worst, normal, and best.
  Format: set_operating_condition –max WORST (or –min BEST or both) –library *library-name*

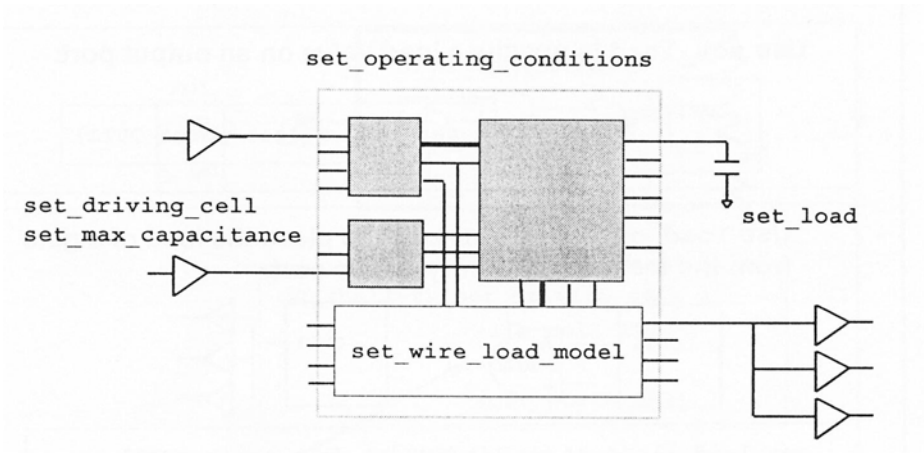


Figure 4-5 Environmental attributes.

[15] The unit of capacitance is pF.
[16] A wire load model is an estimate of a net's RC parasitics based on the net's fanout, supplied by vendor.
[17] User can use 'report_lib lib-name' to list the available wire load model.
[18] Use enclosed for sub block level connection and top for top level connection.
[19] User can use 'report_lib lib-name' to list the vendor-supplied operating conditions.

4.2.2.3 Design Rules and Area Constraints - Optional

Vendors impose design rules that restrict how many cells are connected to one another based on capacitance, transition and fanout. User may apply more conservative design rules to anticipate the interface environment and prevent the design from operating cells close to their limits, where performance degrades rapidly. DC respects design rules as highest priority of all on the following order: max_capacitance, max_transition, and max_fanout. The commands to restrict them are

- set_max_capacitance,
- set_max_transition, and
- set_max_fanout.

User can also specify area goal for a design by the command: set_max_area *area-value*. User may refer to DC reference for more information about these commands.

4.2.3 Synthesizing and Optimizing a Design

Table 4-2 shows the commands which are used to synthesize and optimize a normal design, and Table 4-3 is that for design with hierarchy. What user needs to do is simply following either table 4-2 or table 4-3 to synthesize and optimize a design. A tutorial using these commands will be given in section 4.4.

Table 4-2 Commands for a simple design.

read_vhdl *my-design.vhdl*	# read_verilog for verilog code
write –format db –hierarchy –output *.unmapped/my-design.db*	# -format vhdl or verilog
link	# link design
source *.script/constraints.scr*	# constrain the design
compile[20] -scan	# test ready compile
report_constraint[21] –all_violators	# report all violation: design rule, setup, hold and area.
report_timing[22]	# report the worst timing path
write –format db –hierarchy –output *.mapped/my-design.db*	# save final output

Table 4-3 Commands for a hierarchical design.

read_vhdl *my-design.vhdl*	# read_verilog for verilog code.
write –format db –hierarchy –output *.unmapped/my-design.db*	# -format vhdl or verilog .
link	# link design.
uniquify	# remove multiple instantiations.
source *.script/constraints.scr*	# constrain the design.
compile -scan	# test ready compile.
report_constraint –all_violators	# report all violation: design rule, setup, hold and area.
report_timing	# report worst timing path.
write –format db –hierarchy –output *.mapped/my-design.db*	# save final output

[20, 21, 22] There are other options, referring to DC document for details.

4.2.4 Generating and Checking Reports

User needs to generate reports and check the reports to see if there is any violation. There are two types of reports: report constraints and report timing. These are briefly described as follows.

4.2.4.1 Report Constraints

Constraint report shows all constraints which have been violated in a design. The violations include design rules, setup, hold and area. The command to use is *report_constraint –all_violators*. Table 4-4 is an example of part of the report. There are also other options. User may refer to DC document for other options and usage.

Table 4-4 Reporting design rule violation.

```
dc_shell-t>  report_constraint -all_violators
...
 max_transition
                          Required      Actual
 Net                     Transition   Transition       Slack
 ------------------------------------------------------------
 I_PRGRM_CNT/n184           0.50         0.69          -0.19  (VIOLATED)
 I_PRGRM_DECODE/n945        0.50         0.63          -0.13  (VIOLATED)
 Ld_Rtn_Addr                0.50         0.61          -0.11  (VIOLATED)

 max_capacitance
                          Required      Actual
 Net                    Capacitance  Capacitance      Slack
 ------------------------------------------------------------
 CurrentState[0]            0.20         0.24          -0.04  (VIOLATED)
 PC[0]                      0.20         0.24          -0.04  (VIOLATED)
 CurrentState[1]            0.20         0.24          -0.04  (VIOLATED)
```

4.2.4.2 Report Timing

Report timing shows path delay and each individual contribution to the path. The command to use is *report_timing*. The command allows user to access Synopsys DesignTime, and it will do the followings.

- The design is broken down into individual timing paths.
- Each timing path is timed out twice: once for a rising edge endpoint and once for a falling edge endpoint.
- The critical path (worst violator) for each clock group is found.
- A timing report for each clock group is echoed to the screen or a file directed by user.

A DesignTime timing report has four major sections: path information section, path delay section, path required section and summary section. Table 4-5, figures 4-6, 4-7 and 4-8 are the examples of the four major sections of a timing report. By checking the report, user is able to know if the design passes the timing goals.

Table 4-5 Path information section.

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : TT
Version: 2002.05
Date   : Fri Jun 28 16:48:52 2002
****************************************

Operating Conditions: slow_125_1.62   Library: ssc_core_slow
Wire Load Model Mode: enclosed

  Startpoint: data1 (input port clocked by clk)
  Endpoint: u4 (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  TT                 5KGATES               ssc_core_slow
```

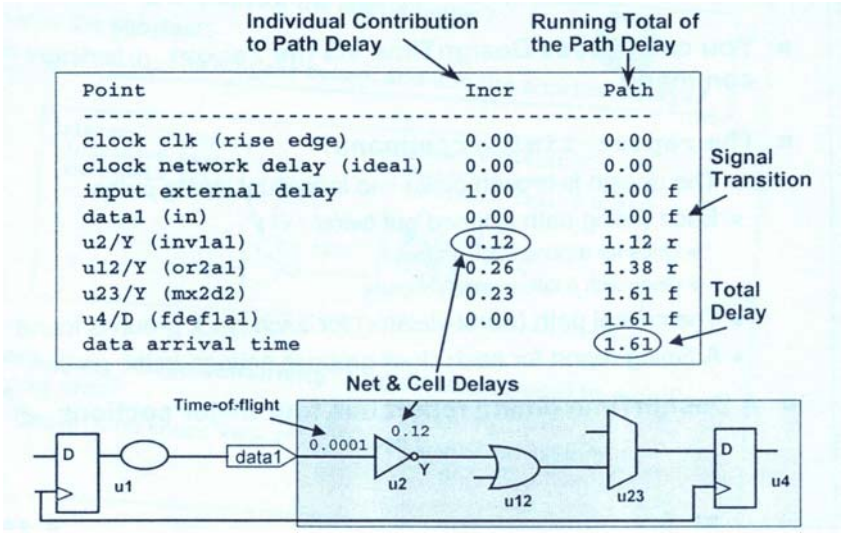


Figure 4-6 Path delay section.

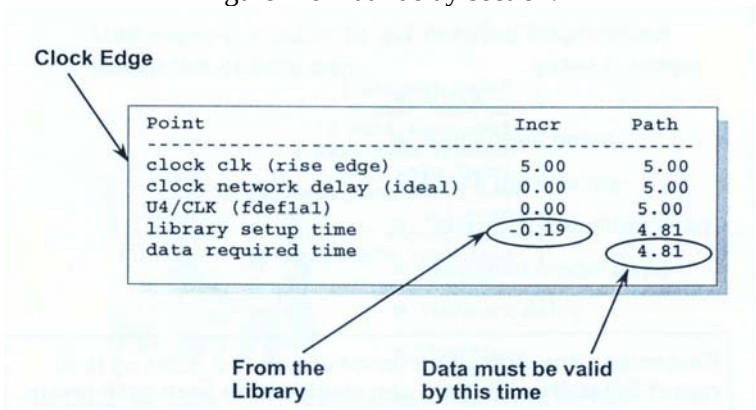


Figure 4-7 Path required section.

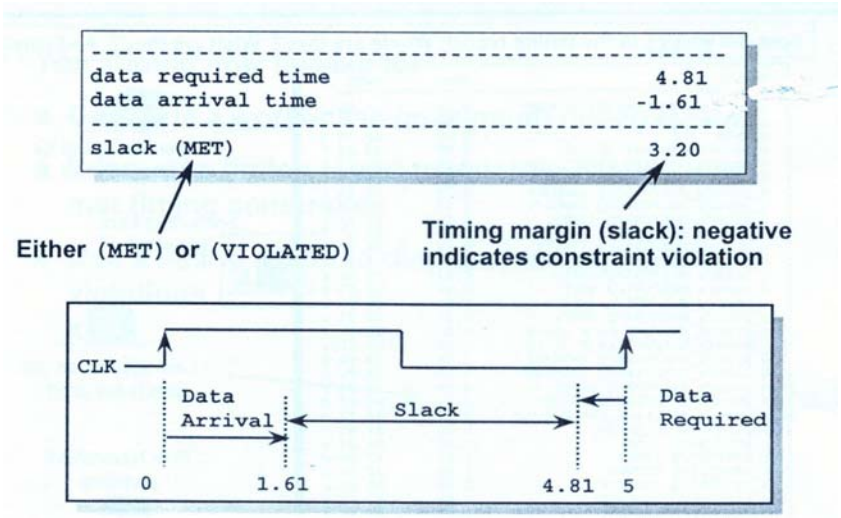


Figure 4-8 Summary section.

The command: *report_timing* has two options which are often used, *-delay max* and *–delay min*. The *report_timing –delay max* reports the worst timing path of each path group for setup[23] time constraints. The *report_timing –delay min* reports the worst timing path of each path group for hold[24] time constraints.

4.3 Methods to Fix Violations

The common methods to fix violations are

- Check and modify the constraints,
- Check the design partition,
- Re-compile using a higher effort for small violations, and
- Modify the RTL source code.

In a nutshell, there are two types of violations: design rule and timing. The methods to fix each type of violations are presented in the following sub-sections respectively.

4.3.1 Fix Design Rule Violation

Design rule violations may cause timing violations. User can use the commands

- *report_net –connections –verbose* and
- *report_timing –net* (for fanout)

to get more information of the design, and then decide if there is a need to use design rule constraints to constrain the design.

If the violations are not big, user may use the command

- *compile –scan –incr –only_design_rule*

to fix them. Executing this command, DC only adds buffers or re-size cells. It fixes only design rule violations and may fix hold time violations.

[23] Refer to figure 6-1 of chapter 6 for setup time definition.
[24] Refer to figure 6-2 of chapter 6 for hold time definition.

4.3.2 Fix Timing Violations

To fix timing violations, user can use the command

- *compile –scan –inc –map high*.

During the process, DC only accepts solutions that reduce critical path slack. The design will most likely get better or stay the same.

A successive compilation will probably not help, unless user changes something like constraints and/or structure of code, and then use the above command.

4.3.3 Other Options

A successful compilation needs skills and a well structured and partitioned design code. Besides going back to check constraints and source code, other algorithms allow user to fix the violations if the violations are not big. The options are

- creating custom path groups to allow more control over optimization, and
- using *compile_ultra*: the full strength of DC in a single command.

As for how to use them, user may refer to DC document.

4.4 Tutorial of Using Design Compiler

The tutorial uses the design - adder32 whose code has passed the verification in last chapter. Following the tutorial, user is able to know how to use DC and how to synthesize and optimize a design.

4.4.1 Preparations

1. Creating directories accordingly to figure 4-2 under project directory
2. Creating the setup file as below and save it as **.synopsys_dc.setup** in the project directory.

```
set symbol_library "c35_CORELIB.sdb c35_IOLIB_3B_4M.sdb[25]"
set target_library "c35_CORELIB.db c35_IOLIB_3B_4M.db[26]"
set link_library "* c35_CORELIB.db c35_IOLIB_3B_4M.db[27]"
set search_path ". /design_kits_installation_directory/synopsys/c35_3.3V \
/synopsys_chip_synthesis_installation_directory/libraries/syn \
/synopsys_chip_synthesis_installation_directory /dw/sim_ver\
/synopsys_chip_synthesis_installation_directory /dw\
./unmapped ./work"

define_design_lib WORK -path ./work    #optional but it is best to set it.
```

3. Creating the constraint file as below and save it as **adder32_dc_constr.scr** in the script directory.

[25,26,27] The IOLIB library ought to be omitted if IO cells are not required during synthesis.

```
current_design adder32
reset_design
create_clock -per 100 -name clk [get_ports CLOCK]

set_dont_touch_network [get_ports CLOCK]
set_clock_uncertainty -setup 0.3 [get_ports CLOCK]
set_clock_uncertainty -hold 0.3 [get_ports CLOCK]

set_operating_conditions -lib c35_CORELIB.db:c35_CORELIB -max WORST
set_wire_load_model -lib c35_CORELIB.db:c35_CORELIB -name 10k
set_wire_load_mode enclosed

set_input_delay -max 2 -clock clk [all_inputs]
set_input_delay -min 0.4 -clock clk [all_inputs]
remove_input_delay [get_ports CLOCK]
set_driving_cell -library c35_CORELIB.db:c35_CORELIB -cell BUF8 [all_inputs]
remove_driving_cell [get_ports CLOCK]

set_output_delay -max 0 -clock clk [all_outputs]
set_output_delay -min 0 -clock clk [all_outputs]

set_load 0.1 [all_outputs]
```

4.4.2 Synthesizing and Optimizing a Design

User can follow the steps listed in this section to synthesize and optimize a design.

4.4.2.1 Read and Link Design

1. Invoke DC by type **design_vision** in the project directory. The Design Vision window appears as shown in figure 4-9.
   % design_vision

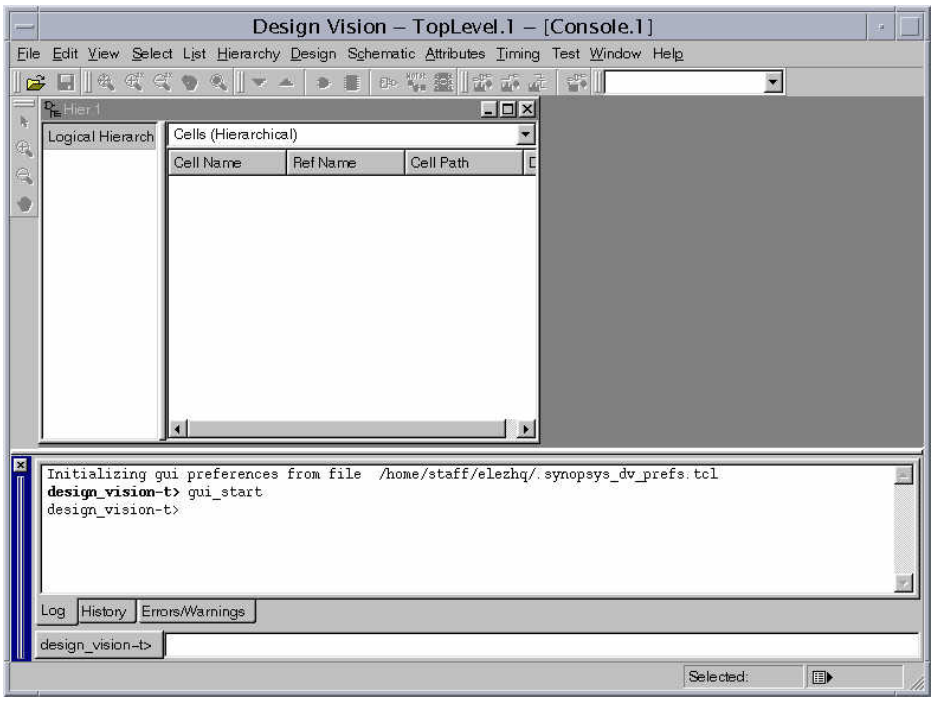


Figure 4-9 Design vision window.

There are three main parts on the window.

- the first part : pull down menu, toolbar and Hier.1 window;
- the second part: log (history and error/message) window;
- last part: command field. Users can type command here instead of using the pull down menu.

2. On **Design Vision** window, click on **File→Setup…** to check the settings. For this example, figure 4-10 shows the current settings.

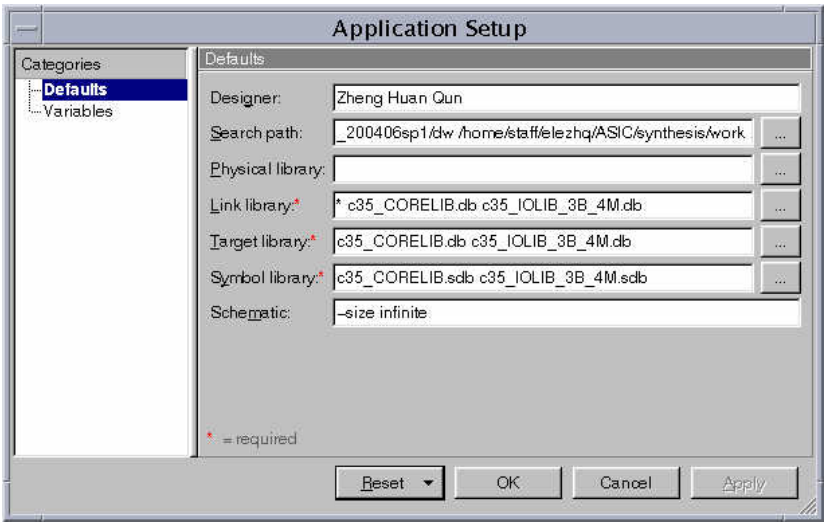


Figure 4-10 Design settings.

3. On **Design Vision** window, click on **File→Read…** to read a design. The Read Designs window appears as shown in figure 4-11.

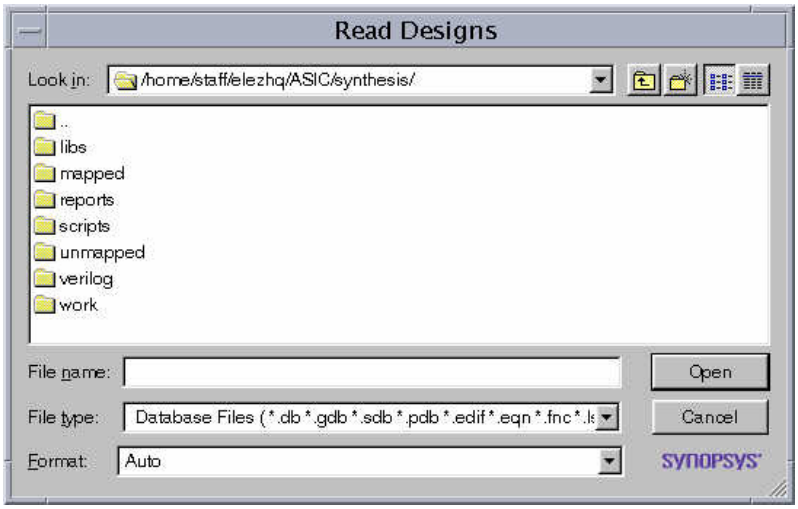


Figure 4-11 Read Designs window.

Note: the command '**read_file...**'in the log window.

4. On the **Read Designs** window, click on verilog folder (where source code is saved) and choose adder32.v.

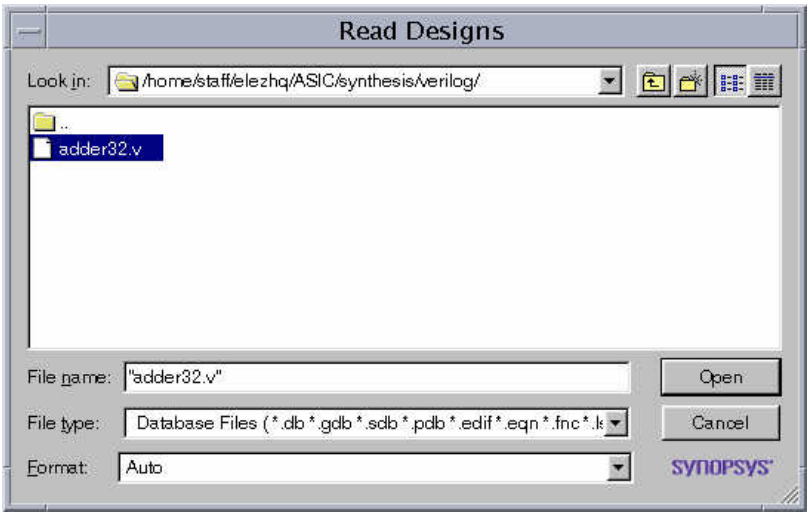


Figure 4-12 Read source code.

5. Then click on **Open** on Read Designs window. The design is read in as shown in figure 4-13. You will see an icon labeled adder… in the **Hier.1 window** under the heading **Logical Hierarchy**.

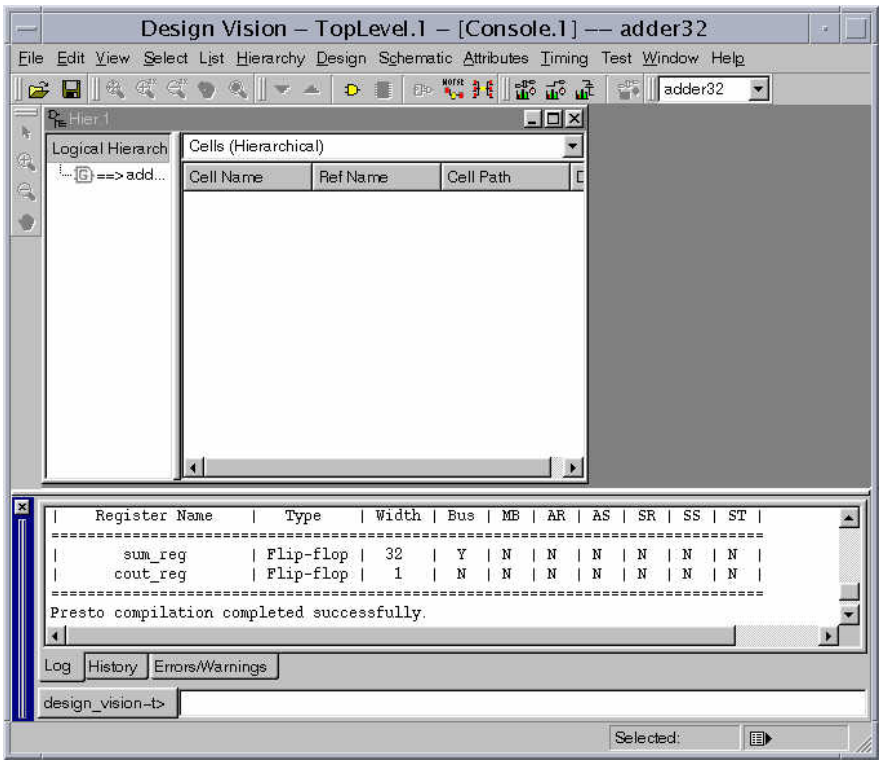


Figure 4-13 Design Vision window with the design - adder32.

The design adder32 is now in Design Compiler memory in terms of GTECH (synopsys library) components.

6. On **Design Vision** window, click on **File→Link Design…** from pull down menu. The Link Design window appears as shown in figure 4-14.

Figure 4-14 Link Design window.

7. Click on **Ok** on the **Link Design** window. The design is linked and the messages appear in the log window, as shown in figure 4-15.

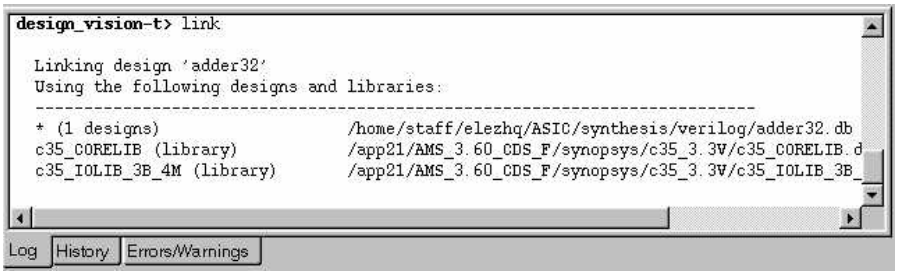
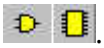


Figure 4-15 Link design message.

8. Click on the icon labeled adder… in the **Hier.1 window**. Two yellow icons will appear in the toolbar of **Design Vision** window  .
9. Push into the "Symbol View" by clicking on the  icon. A block with input and output ports attached to it appears in the symbol view window. This is referred to as the symbol view of the design. The symbol view shows the block diagram of the design.

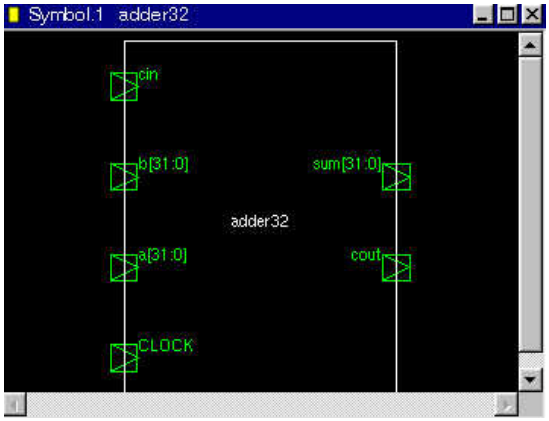


Figure 4-16 Symbol view of the design.

10. Push into the Schematic View by clicking on the  icon. Designer can check if this is similar to what is expected from the RTL code.

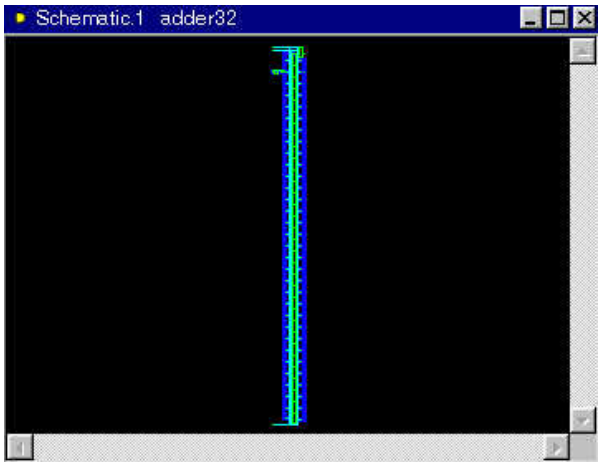


Figure 4-17 Schematic view of the design.

11. Zoom in to view the schematic by clicking on the  icon.
12. Save the design in ./unmapped/adder32_unmapped.db by clicking on **File→Save As…**. The Save Design As window appears as shown in figure 4-18.

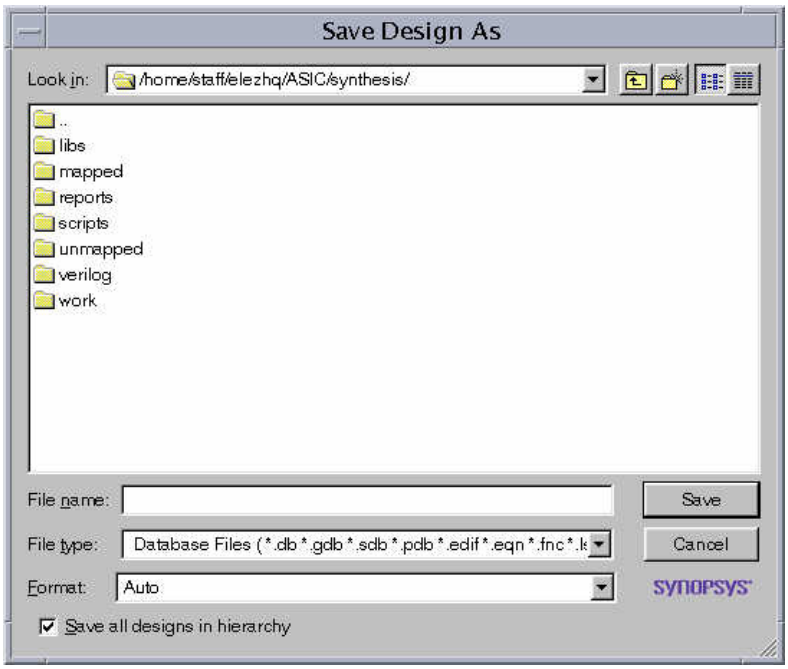


Figure 4-18 Save Design As window.

Note the command '**write …**' in the log window.

13. Click on the **unmapped folder** and type the name **adder32_unmapped** in the file name field. Then click on **Save**. Note the command in the log window.

4.4.2.2 **Constraining Design**

Just proceed to step 1 if it is continuing from section 4.4.2.1. Otherwise, **start** DC with command: design_vision, **read** the design- adder32_unmapped.db from the unmapped folder and then **link** the design by clicking on **File→Link Design…**.

1. To constrain a design, click on **File→Execute script…**. The **Execute Script File** window appears as shown in figure 4-19.

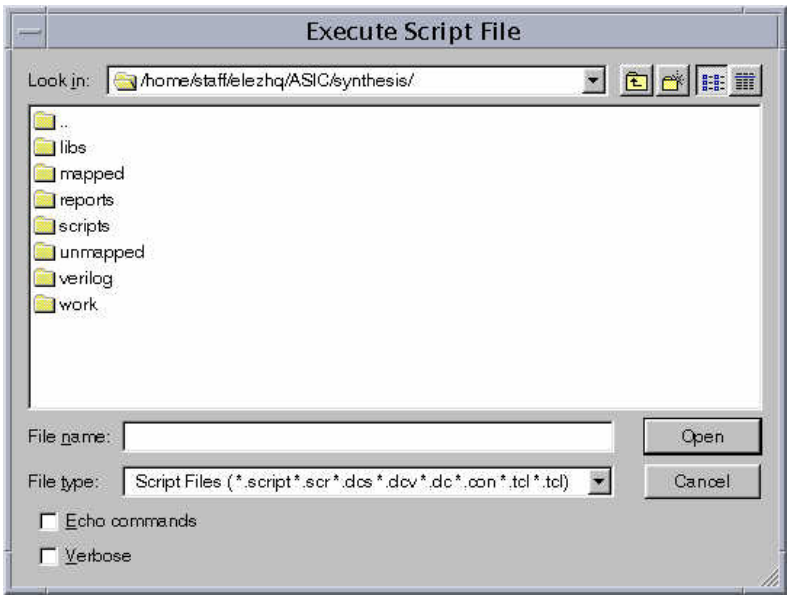


Figure 4-19 Execute Script File window.

2. Click on **the folder** where the constraints saved and click on **the constraint file** name. Then click on **Open**. The messages appear in the log window which is shown in figure 4-20.

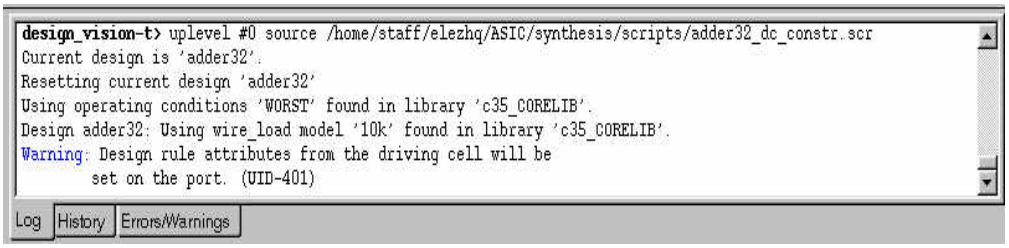


Figure 4-20 Messages of executing constraints.

Note:

- Ignore the warning messages as set_driving_cell requires specifying the output pin name because this cell has only one output pin.
- Note the command source.
- Error messages appear in red color and warning in blue color.

4.4.2.3 Compiling a Design

1. Click on **Design→Compile Design…** on the **Design Vision** window. The Compile Design window appears as shown in figure 4-21.

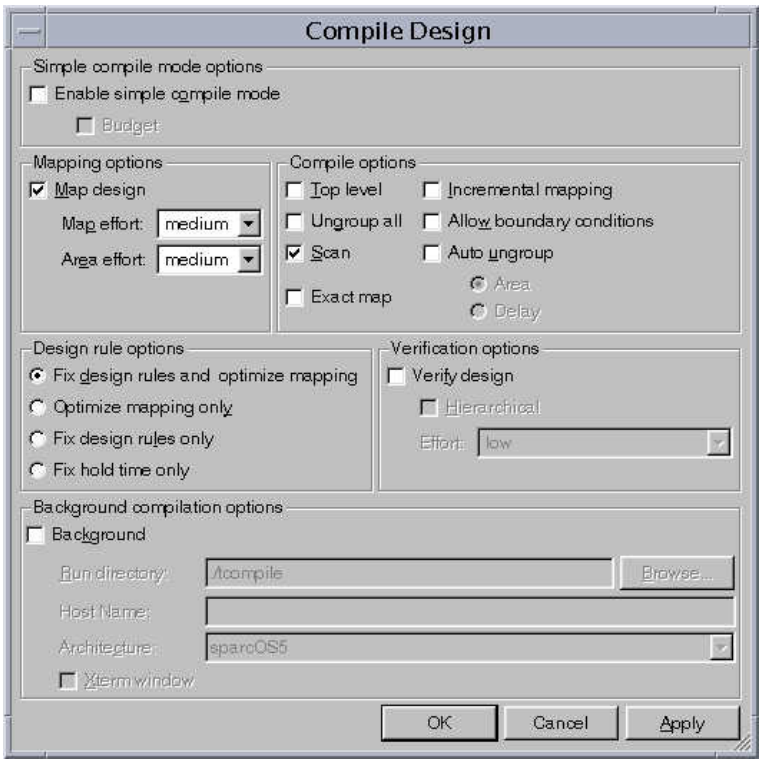


Figure 4-21 Compile Design window.

2. Click on **Ok** to start the compilation. Note the command **Compile ….** and the message 'optimization complete' in the log window. Note the change in **Hier.1** window.

```
 ELAPSED            WORST NEG TOTAL NEG  DESIGN
  TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
--------- --------- --------- --------- --------- -------------------
  0:00:21   24935.2      0.00       0.0       0.0
  0:00:21   24935.2      0.00       0.0       0.0
  0:00:21   24935.2      0.00       0.0       0.0
  0:00:21   23734.0      0.00       0.0       0.0

 Optimization Complete
 ---------------------
 Transferring design 'adder32' to database 'adder32_unmapped.db'

Current design is 'adder32'.
```

Figure 4-22 Messages in log window.

.

4.4.2.4 Generating Reports

1. Type the command '**report_constraint –all_violators**' (or type '**redirect reports/viol.rpt {report_constraint -all_violators}** to save the report') in the ***design_vision-t***> field as shown in figure 4-23 and then enter.

Figure 4-23 Command to generate violation report.

2. The violation report appears in the log window (or saved in the reports/viol.rpt) as shown in figure 4-24.

```
design_vision-t> report_constrain -all_violators
Information: Updating design information... (UID-85)

****************************************
Report : constraint
        -all_violators
Design : adder32
Version: V-2004.06-SP1
Date   : Wed Sep  7 14:00:10 2005
****************************************

This design has no violated constraints.

1
```

Figure 4-24 Constraint report.

Note: if there was violation, fix them at this stage.

3. Type the command '**report_timing**' (or type '**redirect reports/timing.rpt {report_timing}**' to save the timing report) in the ***design_vision-t***> field as shown in figure 4-25 and then enter.

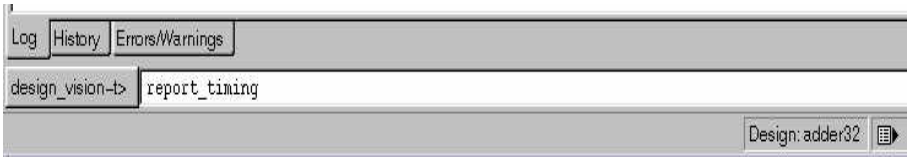


Figure 4-25 Command to generate timing report.

4. The timing report appears in the log window (or saved in reports/timing.rpt) as shown in figure 4-26.

```
add_1_root_add_12_2/U1_26/CO (ADD32)                    0.64      19.34 f
add_1_root_add_12_2/U1_27/CO (ADD32)                    0.64      19.98 f
add_1_root_add_12_2/U1_28/CO (ADD32)                    0.64      20.62 f
add_1_root_add_12_2/U1_29/CO (ADD32)                    0.64      21.26 f
add_1_root_add_12_2/U1_30/CO (ADD32)                    0.64      21.90 f
add_1_root_add_12_2/U1_31/S (ADD32)                     0.81      22.71 r
add_1_root_add_12_2/SUM[31] (adder32_DW01_add_33_0)     0.00      22.71 r
sum_reg[31]/D (DFS3)                                    0.00      22.71 r
data arrival time                                                 22.71

clock clk (rise edge)                                 100.00     100.00
clock network delay (ideal)                             0.00     100.00
clock uncertainty                                      -0.30      99.70
sum_reg[31]/C (DFS3)                                    0.00      99.70 r
library setup time                                     -0.27      99.43
data required time                                                99.43
-------------------------------------------------------------------------
data required time                                                99.43
data arrival time                                                -22.71
-------------------------------------------------------------------------
slack (MET)                                                       76.73
```

Figure 4-26 Timing report.

5. Save the design by clicking on **File→Save AS…** on the **Design Vision** window. Save it as adder32_mapped.db and adder32_mapped.v under the folder mapped.
   Note the command '**write …**' in the log window.
6. Save the standard delay format (SDF) file by type **write_sdf mapped/adder32.sdf** in the field of ***design_vision-t***>. Check if there is a file named adder32.sdf under the mapped directory.

4.4.3 Insert Pads

For a complete design, pads should be inserted to input and output pins in the end. Pads should be inserted on top design only. The steps to insert pads are

1. Load top design as the method of section 4.4.2.1.
2. Link design by type command **link** in the field of **design_vision-t**>.
3. type **set_pad_type** in the field of **design_vision-t**>.
4. type **set_port_is_pad [all_inputs]** in the field of **design_vision-t**>.
5. type **set_port_is_pad [all_outputs]** in the field of **design_vision-t**>.
6. type **insert_pads** in the field of **design_vision-t**>. After a few moment, the following message appears in the log window.

```
  Inserting IO Pads in  Design 'adder32'
  Transferring design 'adder32' to database 'adder32_mapped.db'
Current design is 'adder32'.
1
```

7. Constrain the design as section 4.4.2.2.
8. Compile design as the method of section 4.4.2.3 with the option of **top** and **scan**.
9. Next, timing report may be generated for checking.
10. Save the design as adder32_pad_mapped_top.v and adder32_pad_mapped_top.db using the steps listed in section 4.4.2.4.
11. With the command **write_sdf** to save SDF file, for functionality verification. Below is the sample.

```
design_vision-t> write_sdf mapped/adder32_pad.sdf
Information: Annotated 'cell' delays are assumed to include load delay. (UID-282)
Information: Updating design information... (UID-85)
Information: Writing timing information to file '/home/staff/elezhq/ASIC/synthesis/mapped/adder32_pad.sdf'. (WT-3)
1
```

Above is the normal procedure to insert pads on top design. If there are warnings or massage like 'insert pads terminated abnormally', use commands **get_attribute** to check IOLIB attribute and remove attribute like don't_use by issuing command **remove_attribute**. The use of command *get_attribute* and *remove_attribute* can be found from the pull down menu **Help→Man Pages** on the design vision window.

4.5 Conclusion

The method of synthesizing and optimizing a design is described in this chapter. It is well known that a successful synthesis and optimization need skills on both RTL coding and design compiler usage. User may refer to Verilog/VHDL books and synopsys DC user guide for coding styles, besides this manual. In addition to, user can refer to DC user guide and man page for more on DC commands and variables and their usage.

Synthesis and optimization are an iterative process. User may need to modify their source code if violations are not able to be corrected. Generally, it can proceed to the next step if setup time is met, no design rule violation and minor hold time violation. Fixing minor hold time violations can be done after layout with real delays back annotated. Of course, if gross hold time violations are detected after initial synthesis, they should be fixed at the pre-layout level.

After synthesis and optimization, user can proceed to NCLaunch for pre-layout verification.

5. Pre-Layout Verification With NCLaunch

Pre-layout verification with NCLaunch is described in this chapter. Recalling chapter 3, NCLaunch is just an interface where compiler is invoked to compile the source code, elaborator is invoked to elaborate the design, and finally simulator is invoked to simulate the design. Pre-layout verification with NCLaunch is almost as same as RTL verification. *The only difference is that a SDF file including delay information is involved. The SDF file has to be compiled first and then $sdf_annotate system task has to be inserted into design source files*. Compiling SDF and using *$sdf_annotate* system task will be described in the following sections.

The arrangement is as follows. The overview of SDF annotation is given in section 5.1. The *$sdf_annotation* system task is described in section 5.2. The requirements for *$sdf_annotation* system tasks are presented in section 5.3. In section 5.4, a tutorial of pre-layout verification using NCLaunch is demonstrated. Conclusion is given in section 5.5.

5.1 Overview of SDF Annotation

Refer back to chapter 3, the verification process consists of three steps, compiling, elaborating, and simulating. SDF back annotation is performed during elaborating. The elaborator recognizes *$sdf_ammotate* system tasks in design source files, and if the *$sdf_annotate* system tasks are scheduled to run at time 0 and if they meet other requirements, annotation is performed automatically.

See "*$sdf_annotate* system task" in section 5.2 for a description of the *$sdf_annotate* system task. See "requirements for *$sdf_annotate* system task" in section 5.3 for a description of the rules that apply to the *$sdf_annotate* tasks for automatic SDF annotation.

5.2 $sdf_annotate System Task

The syntax of the *$sdf_annotate* system task is as follows:

$sdf_annotate ("sdf_file", [module_instance], "config_file", "log_file", "mtm_spec", "scale_factor", "scale_type").

The "sdf_file" argument is required. All the other arguments are optional. If optional arguments are omitted, the commas that would have surrounded them must remain, unless the omitted arguments are consecutive and include the last argument. Below shows two examples. First one is that the third ("config_file") and fourth ("log_file") arguments are omitted. Second one is that the last three arguments ("mtm_spec", "scale_factor", "scale_type") are omitted.

$sdf_annotate ("mysdf.sdf", m1, , , "MAXIMUM", "1:2:3", "FROM_MTM").
$sdf_annotate ("mysdf.sdf", m1, "mysdf.config", "mysdf.log").

The definition of each item of *$sdf_annotate* system task will be listed below.

- "sdf_file"
  The name of the SDF file can be:
  - The name of the SDF source file (for example, adder32.sdf)
  - The name of the compiled SDF file (for example, adder32.sdf.X)
  - The name of a compressed or zipped SDF file (for example, adder32.sdf.gz)
  - The name of a compressed or zipped and compiled file (for example, adder32.sdf.gz.X)

The elaborator determines the format of the SDF file, and then invokes cadence *ncsdfc* utility to compile the SDF file accordingly. For small design, the format of adder32.sdf.X is often used.

- module_instance
  The SDF annotator uses the hierarchy level of the specified module instance to run the annotation. If *module_instance* is not specified, the annotator uses the module that contains the call to the *$sdf_annotate* system task as the *module_instance* for annotation.
- "config_file"
  It is the name of configuration file. The configuration file lets user control how the timing data in the SDF file is annotated. Using a configuration file is optional. The annotator uses default settings if it is not specified. Users may refer to NC-Verilog Simulator Help for the description of the configuration file if interested.
- "log_file"
  It is the name of the annotation log file. This file contains status information, warnings, and error messages from the SDF annotator. The annotator also prints warnings and error messages to standard output. By default, the annotator does not create an SDF log file. User must include this argument if the annotation specific messages are needed.
- "mtm_spec"
  Specifies the delay values that user wants to annotate. The *mtm_spec* is one of following keywords:
  - MINIMUM – annotates the minimum delay value.
  - TYPICAL – annotates the typical delay value.
  - MAXIMUM – annotates the maximum delay value.
  - TOOL_CONTROL – annotates the delay value that is specified by the command-line option – mindelays, -typdelays, or –maxdelays.

  The default for *mtm_spec* is TOOL_CONTROL. If no command-line option is specified, the default is TYPICAL. The *mtm_spec* argument overrides the *mtm* command in the configuration file.
- "scale_factor"
  Set three positive real number multipliers that the SDF annotator uses to scale the minimum, typical, and maximum timing values in the SDF file before annotating the values. The syntax of the argument is min_mult : typ_mult : max_mult. For example, "1.6:1.4:1.2". The default for *scale_factor* is 1.0:1.0:1.0, and it overrides the *scale* command in the configuration file.
- "scale_type"
  Specifies how the SDF annotator scales the timing specification. The *scale_type* is one of the following keywords:
  - FROM_MINMUM – scales from the minimum timing specification.
  - FROM_TYPICAL – scales from the typical timing specification.
  - FROM_MAXIMUM – scales from the maximum timing specification.
  - FROM_MTM – scales from the minimum, typical, and maximum timing specifications.

  The default for *scale_type* is FROM_MTM. The *scale_type* argument overrides the *scale* command in the configuration file.

In the following example, timing information in a file called my.sdf.X is used to annotate the module instance top.m1.

```
module top;
…
    circuit m1(i1, i2, i3, o1, o2, o3);
    initital
    begain
            $sdf_annotate ("my.sdf.X", m1, , ,
    "MAXIMUM",
            "1.6:1.4:1.2",  "FROM_MTM");
    end
    …
    …..
endmodule
```

5.3 Requirements for *$sdf_annotate* System Tasks

The elaborator ignores and generates a warning for any *$sdf_annotate* system task that does not satisfy the following rules:

- *$sdf_annotate* tasks must be inside an *initial* block. A *$sdf_annotate* task cannot be referenced in a task call contained in an initial block.
- Only *$sdf_annotate* tasks scheduled to run at time 0 are used for annotation.
- Delay or even control statements cannot precede *$sdf_annotate* calls
- *$sdf_annotate* calls cannot be within or follow *for, while, case, repeat,* or *wait* constructs.
- Because annotation takes place at elaboration time, and the values of variables in the design are determined at simulation time, a *$sdf_annotate* task cannot be invoked from an *if* construct with a variable expression as the condition. The expression that is used in the guard expression must evaluate to a constant.

If a *$sdf_annotate* task violates the above requirements, the elaborator generates warning messages telling user that it is ignoring the system task.

It is possible to override the default automatic SDF annotation mechanism and force annotation by writing an SDF command file and then including the command file when elaborating by using the *–sdf_cmd_file* option. For users who are interested in using the SDF command file, please refer to NC-Verilog simulator help for the use and description of SDF command file. Only automatic SDF annotation is described in this manual.

5.4 Tutorial of Pre-Layout Verification Using NCLaunch

The tutorial is divided into three parts: preparations, compiling SDF file and all the source files, elaborating the design. User can follow the tutorial to understand thoroughly the SDF back annotation topic. The example used is the design - adder32, and the design files are got from synopsys DC.

5.4.1 Preparations

1. Create a working directory: mapped_ncvlog
   % mkdir mapped_ncvlog
2. Copy the design file: adder32_pad_mapped_top.v, test_adder.v (referring to chapter 3 for this file), and SDF file: adder32_pad.sdf to the directory: mapped_ncvlog.

% cd mapped_ncvlog
% cp /the path to design files/ adder32_pad_mapped_top.v .
% cp /the path to test_adder.v/test_adder.v .
% cp /the path to SDF files/ adder32_pad.sdf .

3. Create a lib directory which holds the library files used by design.
   % mkdir lib
4. Copy the library files to the directory lib.
   % cp /path to library files/c35_CORELIB.v lib
   % cp /path to library files/c35_IOLIB_4M.v lib
   % cp /path to library files/udp.v lib
5. Modify the source file - adder32_pad_mapped_top.v to include the *$sdf_annotate* system task as follows.

```
module adder32 ( a, b, cin, CLOCK, sum, cout );
  input [31:0] a;
  input [31:0] b;
  output [31:0] sum;
  input cin, CLOCK;
  output cout;
  wire   n101, n102, n103, n104, n105, n106, n107, n108, n109, n110, n111,
         n112, n113, n114, n115, n116, n117, n118, n119, n120, n121, n122,
         n123, n124, n125, n126, n127, n128, n129, n130, n131, n132, n133,
         n134, n135, n136, n137, n138, n139, n140, n141, n142, n143, n144,
         n145, n146, n147, n148, n149, n150, n151, n152, n153, n154, n155,
         n156, n157, n158, n159, n160, n161, n162, n163, n164, n165, n199,
         n166, n167, n168, n169, n170, n171, n172, n173, n174, n175, n176,
         n177, n178, n179, n180, n181, n182, n183, n184, n185, n186, n187,
         n188, n189, n190, n191, n192, n193, n194, n195, n196, n197, n198, n1,
         n35;
  wire   [32:0] temp;

initial
begin
$sdf_annotate ("adder32_pad.sdf.X", adder32, , "sdf.log", "MAXIMUM", , "FROM_MTM");
end

  assign n35 = CLOCK;

  DFS3 cout_reg ( .C(n199), .D(temp[32]), .SD(n198), .SE(n1), .Q(n198) );
```

Figure 5-1 *$sdf_annotate* system task in the pre-layout design source file.

5.4.2 Compiling SDF File and Source Files

Refer to chapter 3, if you forgot the usage of NCLaunch.

1. Start NCLaunch in the working directory – mapped_ncvlog as follows. You should see the files as figure 5-2 in the NCLaunch window.
   % nclaunch –new&

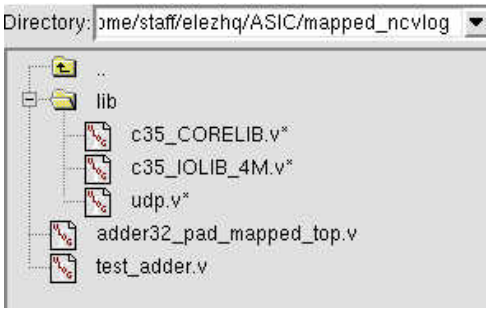


Figure 5-2 Start up of pre-layout verification.

2. Open the SDF compiler to compile SDF file by clicking on **Tools→SDF Compiler…**. The **SDF Compiler** form appears. Set the form as figure 5-3, and then click on **Advanced Options** button.

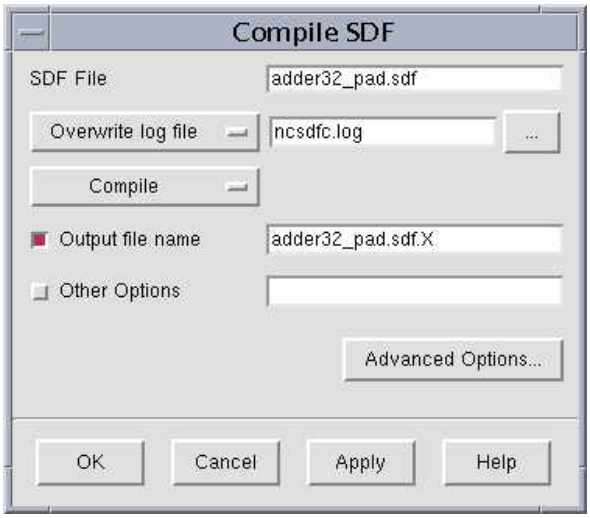


Figure 5-3 SDF compiler form.

3. Set the Advanced Options form as figure 5-4. Click on **Ok** on both forms. The SDF compiling starts and the ncsdf.log file creates in the working directory.

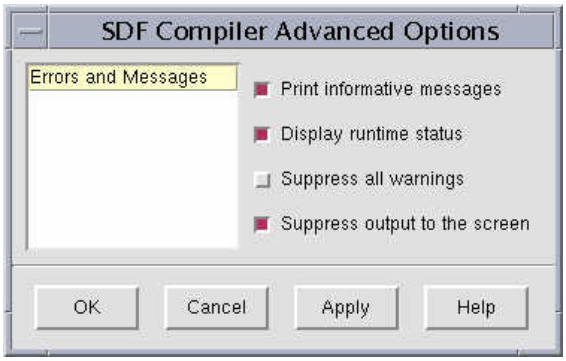


Figure 5-4 Advanced Options form.

4. Check two files created during the SDF compiling: adder32_pad.sdf.X and ncsdf.log.
5. To compile all the source files, select all the source files as figure 5-5, and then click on Verilog Compiler button  . After a few moments, the compilation completes.

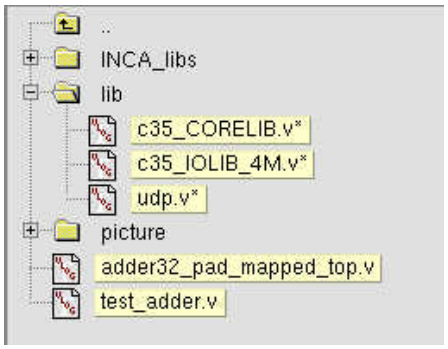


Figure 5-5 Selection of all the source files.

5.4.3 Elaborating Design

1. Click on the top module - test_adder as figure 5-6 for elaboration.

Figure 5-6 Select top module under worklib directory.

2. Click on the Elaborator button  . The **Elaborator** form appears. Set the elaborator form as figure 5-7.

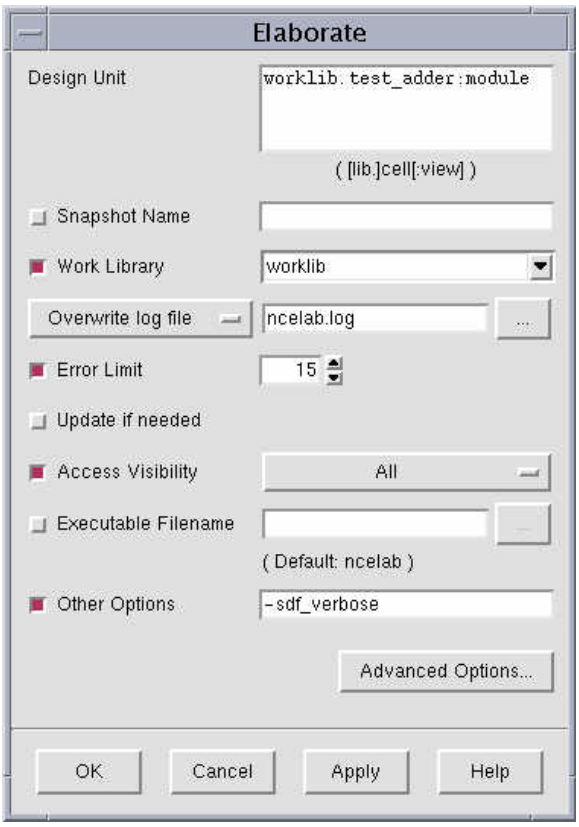


Figure 5-7 Elaborator form settings.

3. Click on the **Advanced Options** button on the elaborator form. Select **Errors and Messages** on the Advanced Options form and set the form as figure 5-8, to get more SDF annotation information.

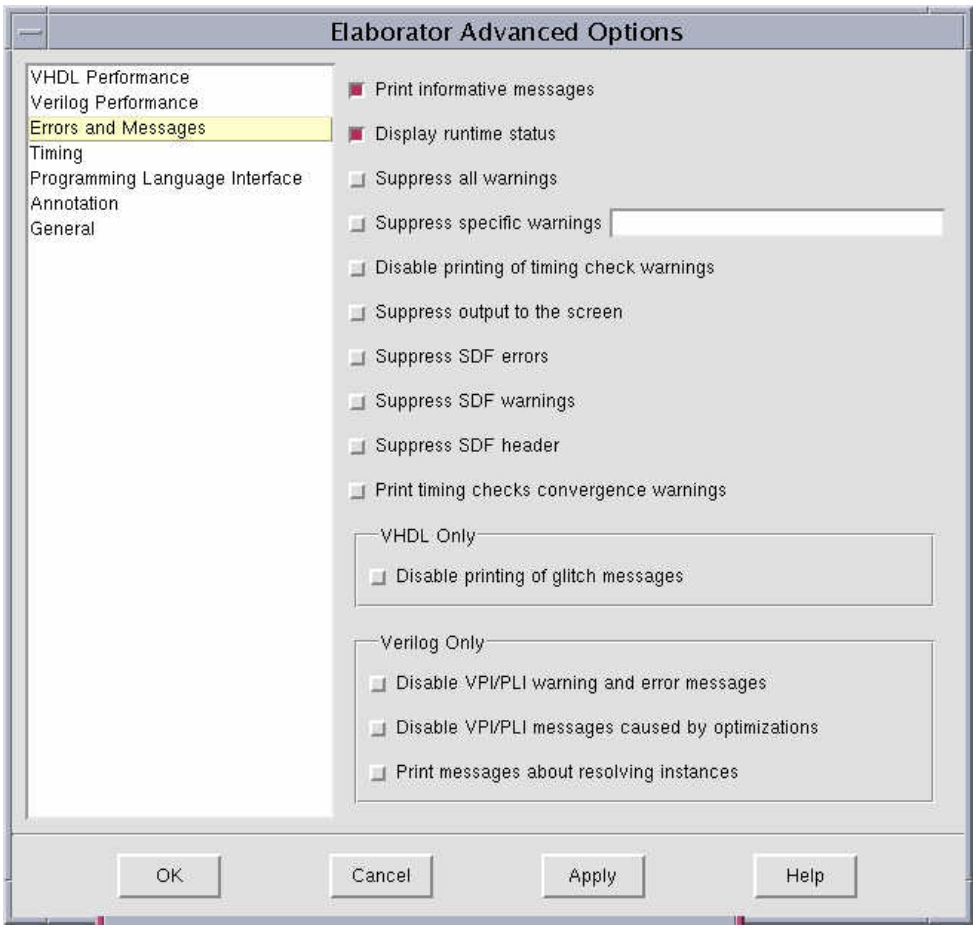


Figure 5-8 Advanced Options form settings.

4. Click on **Ok** on both forms. The elaboration starts and it completes after a moment.
5. Check the messages in the NCLaunch window. If the SDF annotation is successful, the following messages should appear.

Figure 5-9 Elaborating messages.

6. A sdf.log file should be created in the working directory. Open it to view the annotation timing data. The sdf.log file of example adder32 is shown in figure 5-10.

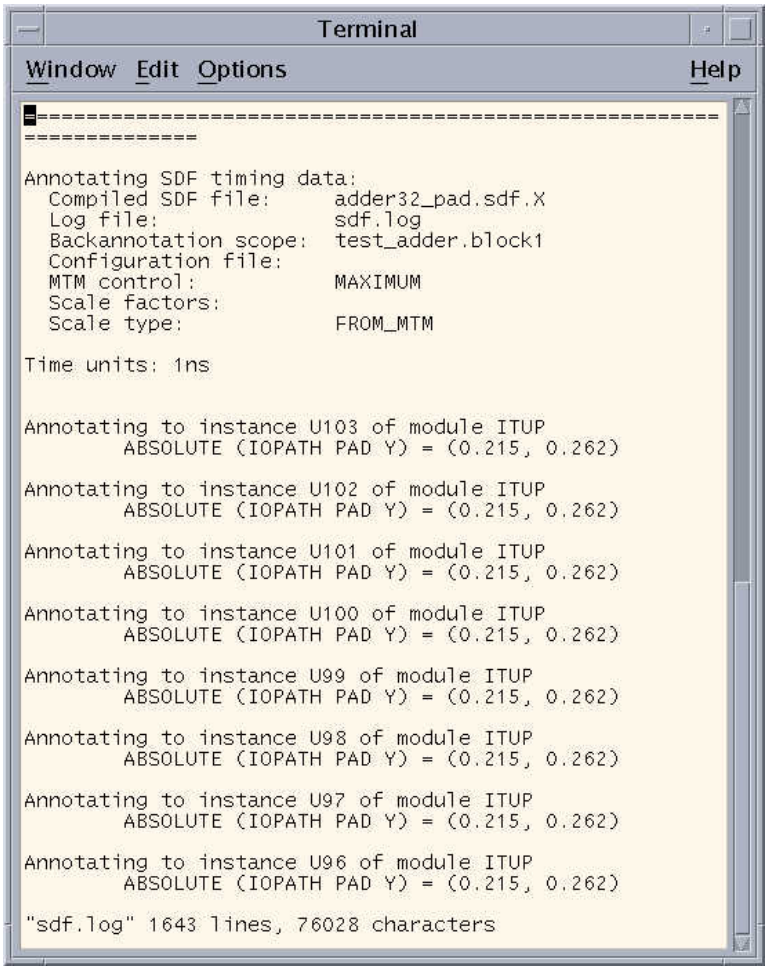


Figure 5-10 sdf. log file.

Now the elaboration finished. A snopshot – worklib:test_adder:module is created after the elaborating. Select the snapshots as figure 5-11, and start simulation to get the pre-layout simulation results.

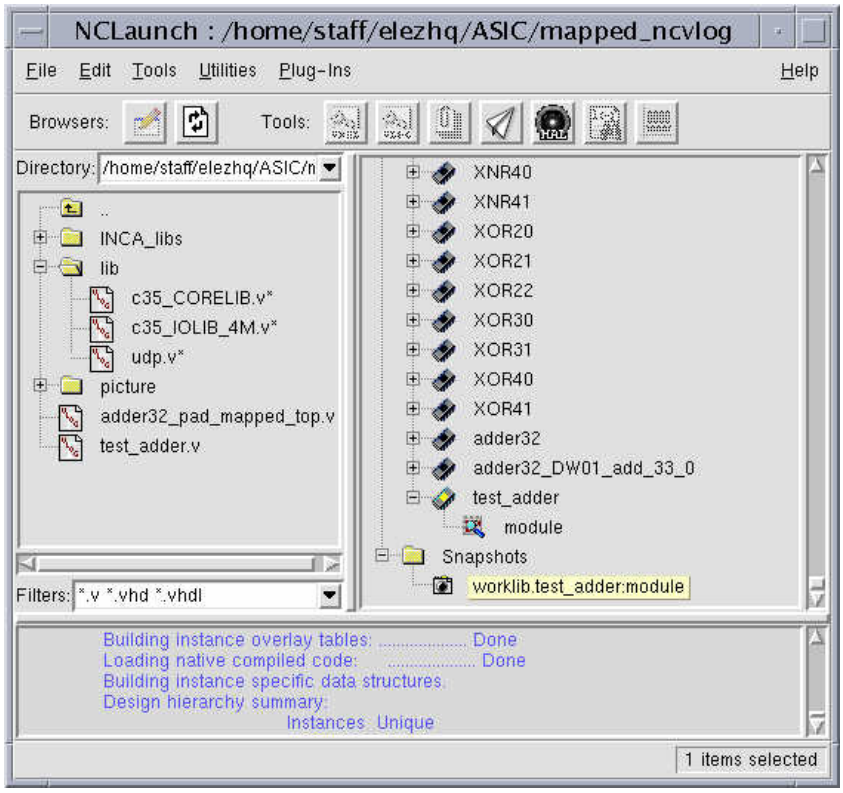


Figure 5-11 Select the snapshots to prepare for simulation.

Because simulating a pre-layout design is as same as simulating a RTL design after elaborating, the simulation methods are not repeated in this chapter. User can refer to chapter 3:

- section 3.2.3 for starting the simulator,
- section 3.2.4 for simulating the design, and
- section 3.2.5 for displaying simulation data.

5.5 Conclusion

Pre-layout verification including SDF back annotation is described in this chapter. Comparing to Chapter 3, the only difference is that the SDF file needs to be compiled first and *$sdf_annotate* system task has to be stated in the design source file. What users need to do is following the steps listed in section 5.4 and section 3.2.3, 3.2.4 and 3.2.5 of chapter 3, to finish the pre-layout verification.

Now, it is time to bring the design to next chapter for pre-layout static timing analysis if the verification is satisfactory.

6. Pre-Layout Timing Analysis Using Synopsys PrimeTime

After getting the gate level netlist from DC, it should be brought to primetime (PT) for Static Timing Analysis (STA). The purpose of STA is to investigate the design on the aspects of setup time and hold time, and to find out the paths which violate the timing targets and the sub-blocks which have no more improving space (bottleneck blocks). By analysis the information, designer is able to decide how to improve the performance of the design.

The arrangement of the chapter is as follows. An introduction to STA is presented in section 6.1. The method of reading design data is described in section 6.2. Constraining design is mentioned in section 6.3. Specifying timing exceptions are given in section 6.4. In section 6.5, checking and analyzing design is described. The types of STA are presented in section 6.6. A tutorial of using PT with the example - adder32 is demonstrated in section 6.7. Lastly, conclusion is given in section 6.8.

6.1 Introduction to Static Timing Analysis

STA verifies that every flip-flop in the design meets its setup and hold time requirements, where the definitions of the setup and hold time are shown in figures 6-1 and 6-2 respectively. STA uses SPICE characterized data stored in a technology library to verify gate level circuit timing.

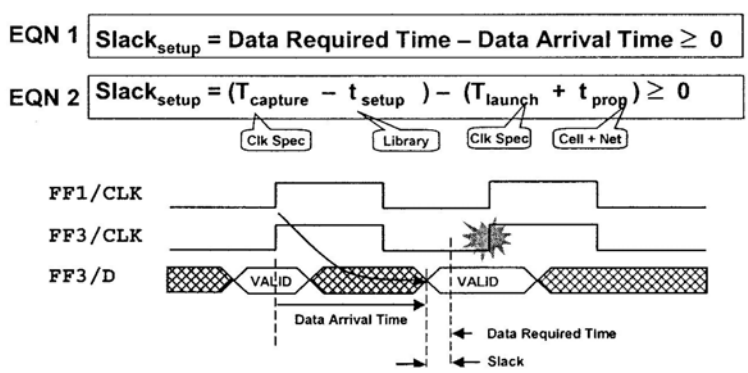


Figure 6-1 Setup definition.

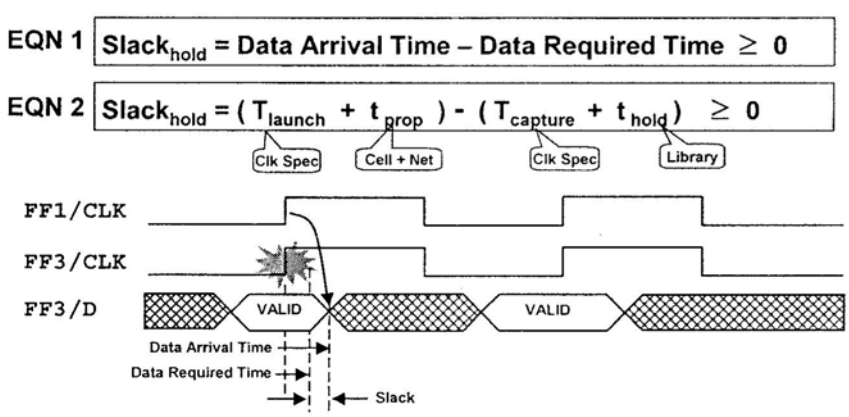


Figure 6-2 Hold definition.

There are two main steps in STA:

- The delay of each path is calculated (data arrival time),
- All path delays are checked to see if setup time and hold time (timing constraints) have been met (Slack >=0).

Each timing path has a start-point and end-point. The start-point is input ports and clock pins of flip-flops or registers, and end-point is output ports and data input pins of sequential devices. The actual path delay (data arrival time) is the sum of net and cell delays along the timing path, where the net and cell delays are provided by the technology library.

Timing and design rule are checked during STA. PT checks for setup and hold requirements of every timing path in the entire design based on specified constraints, and it checks for following design rule constraints:

- Capacitance: *max_capacitance* and *min_capacitance*
- Transition: *max_transistion* and *min_transistion*
- Fanout: *max_fanout* and *min_fanout*

STA flow is shown in figure 6-3. As the figure shows, there are five steps in STA. Each of the five steps will be described in the following sections.

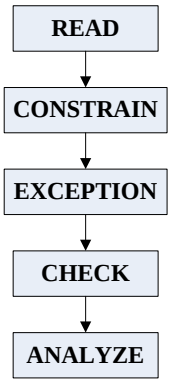


Figure 6-3 STA flow.

6.2 Reading Design Data

There are three steps to read design data.

- Set variables: *search_path* and *link_path*.
- Read design.
- Link design.

As running DC, a setup file *.synopsys_pt_setup* needs to be created, which defines the two variables: *search_path* and *link_path*. The definitions of the variables are given below:

- **search_path** defines the path which PT will search for when necessary, and
- **link_path** specifies where PT searches for designs and technology (library) files when linking the design.

A setup file defining the two variables are shown in Table 6-1, where * stands for PT memory. User can use the commands: *printvar search_path* and *printvar link_path* to verify the settings after invoking PT.

Table 6-1 *.synopsys_pt_setup* file.

set search_path	"$search_path scripts mapped reports"
set link_path	"* /path to tech installation directory/tech_lib.db"

To read, PT uses the commands:

- **read_verilog** – to read netlist in Verilog;
- **read_db** – to read netlist in db format;
- **read_vhd** – to read netlist in VHDL format.

Table 6-2 shows the method to read and link a design which has sub-designs in three different formats.

Table 6-2 Read and link design[28].

pt_shell> read_verilog "sub_design1.v,…, top.v" pt_shell> link_design top	Verilog format
pt_shell> read_db "sub_design1.db, …, top.db" pt_shell> link_design top	db format
pt_shell> read_vhd "sun_design1.vhd, …, top.vhd" pt_shell> link_design top	vhdl format

If there is error message when reading/linking design, user needs to check the *search_path* and *link_path* variables defined in the setup file. By default[29], PT will create black boxes if *link_design* couldn't solve a particular reference.

6.3 Constraining Design

PT accepts the constraint file which is used by DC. User can use the constraint file of DC, if there is no specific requirement. For user who is interested in knowing more about constraints, please refer to DC and PT documents.

6.4 Specifying Timing Exceptions

Timing exceptions are used to override the default single-cycle constraints described by *create_clock*, *set_input_delay*, and *set_output_delay*. Timing exception commands are listed in table 6-3. The usage of *set_false_path* and *set_multicycle_path* is described below, and user can refer to PT document for the usage of the rest .

Table 6-3 Timing exception commands.

set_false_path:	Removes timing constraints from timing path
set_multicycle_path:	Allows more than one clock cycle for a timing path
set_max_delay:	Specifies max and min delays on paths
set_min_delay:	
report_exceptions:	Reports current timing exceptions
reset_path:	Restores the default timing constraints on specified paths
transform_exceptions	Performs transformations on timing exceptions

[28] Mixed netlist also works, referring to PT document.
[29] It is because that the variable *link_create_black_box* is true by default, referring to PT manpage for the variable.

- **Set_false_path**

The usage is shown below, referring to figure 6-4.

pt_shell> set_false_path –from A –to R1/D (referring to figure 6-4 (a))
pt_shell> set_false_path –through R1/Q (referring to figure 6-4 (b))

(a) (b)

Figure 6-4 Circuit diagram.

- **Set_multicycle_path**[30]

The usage is shown below, referring to figure 6-5.

pt_shell> set_multicycle_path 2 –from FFA/CP –through Multiply/Out –to FFB/D

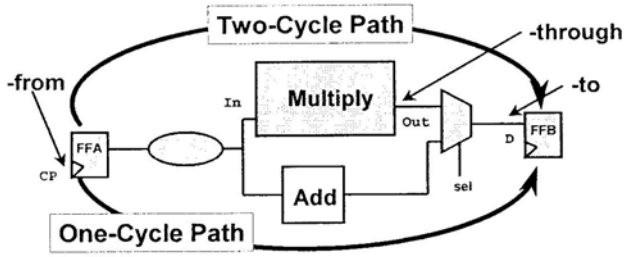


Figure 6-5 Diagram of multi-cycle path.

PT performs a default hold check at 0 if it is not specified explicitly. For examples,

```
create_clock –period 10 [get_ports CLK]
set_multicycle_path  –setup 6 –to [get_pins C_reg[*]/D]
set_multicycle_path –hold 0 –to [[get_pins C_reg[*]/D]    ← IMPLICIT
```

It can be override with the command '*set_multicycle_path –hold 5 –to [[get_pins C_reg[*]/D]*'. The command means the hold check is at 5th cycle.

6.5 Checking and Analyzing

The objectives of checking and analyzing are to

- find out the scopes of violations,
- do a complete analysis to identify timing and DRC violations,
- identify bottleneck blocks in the design as candidates for re-synthesis, and
- provide "Info Reports" for the largest violations on input paths, reg-to-reg paths and output paths.

[30] Please refer to the PT manpage for details.

6.5.1 Checking

Check design before analyzing is necessary. Check design is to assure that it is fully constrained and to identify problems with design constraints like:

- missing clock definitions,
- ports with missing input delay,
- unconstrained endpoints for setup,
- input/output delay set without a reference clock,
- combinational feedback loops, and more.

The command to check design is *check_timing -verbose.*

6.5.2 Analyzing

There are three analysis techniques. They are

- Constraint Report: *report_constraint*[31] *–all*,
- Bottleneck Report*: report_bottleneck*, and
- Timing Report: *report_timing*.

➢ **Constraint Report**

Constraint report (*report_constraint*) shows all types of violations in a design: *setup*, *hold*, *DRC*, and *pulse width*…. The default is to show the longest violation of each type. With option, the command can specify the violation what user wants to investigate. Below are the most often used options:

- **-all_violators -** showing all the violations in a design and where the violations are;
- **-all –max_delay –min_delay -** showing all the setup and hold violation;
- **-all –max_capacitance –max_fanout** shows all the DRC violations.

➢ **Bottleneck Report**

Bottleneck analysis identifies the cells (or blocks) which are involved in multiple violations. With bottleneck analysis, users are able to identify sub block(s) containing bottleneck cells, to re-synthesize the sub block(s), and then to replace the sub block(s) with newly synthesized sub block(s).

Bottleneck report command is

- *report_bottleneck –cost_type path_count*

where path_count is the default cost_type and it uses the number of violating paths through the cell as the bottleneck cost. Figure 6-6 shows an example of bottleneck analysis.

[31] It is as same as DC command but more powerful.

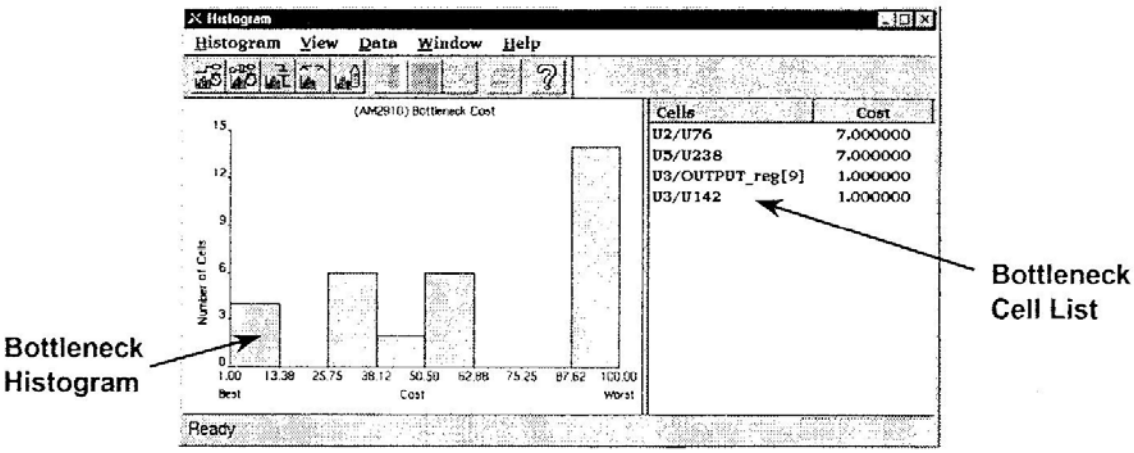


Figure 6-6 Bottleneck figure.

➢ Timing Report[32]

Timing report (*report_timing*) command finds all the individual timing paths in the design for analysis. Each path is analyzed for timing twice, once for a rising edge input and once for a falling edge input. PT organizes its timing reports by path groups. By default, the critical path (worst violator) for each clock group is found and reported.

For easy analysis, user can group timing paths with the command *group_path –name*. Examples to group timing paths are shown below. The commands are shown in table 6-4 and the circuit and grouping diagram are shown in figure 6-7. User can also group paths by clocks if design has more clocks.

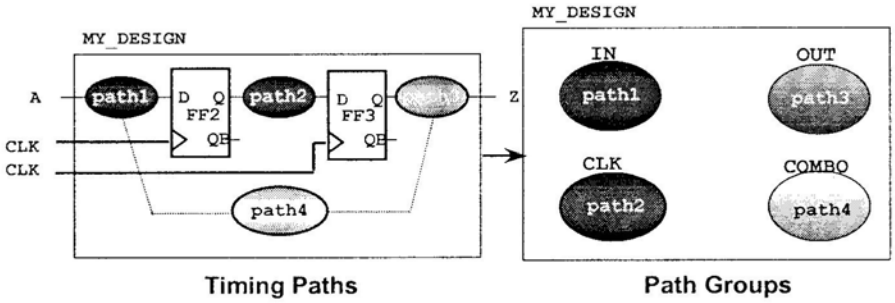


Figure 6-7 Timing paths and path groups.

Table 6-4 Commands to create groups.

```
group_path –name IN –from [all_inputs]
group_path –name OUT –to [all_outputs]
group_path –name COMBO –from [all_inputs] –to [all_outputs]
```

User can specify a path to report with the options: *-from* and *–to*. The timing report for setup and hold checks are created with the commands shown in table 6-5. Keep in mind that *report_timing* by default reports *one path* with the worst slack within each path group.

[32] It is as same as DC command but more powerful.

Table 6-5 Timing report for setup and hold checks.

report_timing –delay max	Setup check report
report_timing –delay min	Hold check report
report_timing –delay min_max	Setup and hold check report
report_timing –max_paths 10 report_timing –nworst 10	Multiple timing reports. It contains the analysis of at most 10 slowest paths.

6.6 Types of Static Timing Analysis

There are four types of analysis: *single operating condition* (OC) analysis, *best case (BC)/worst case (WC)* analysis, *on-chip variation*, and *case* analysis.

➢ **Single OC Analysis**

The command is shown below, where WORST (or BEST) is one of the conditions of library and lib_name is the library name used for design. Using WORST is for Max paths timing and using BEST is for Min paths timing.

```
pt-shell>set_operating_conditions –analysis_type single WORST(or BEST) –library
lib_name
```

➢ **Best/Worst Case Analysis**

It is used to specify both a min and a max OC. The command is shown below.

```
pt-shell>set_operating_conditions –analysis_type bc_wc –library lib_name –min
BEST –max WORST
```

As for on-chip variation and case analysis, user may refer to PT user guide.

By default, PT performs analysis based upon a single OC. If no operating conditions are specified for a design, the tool uses the default operating condition of the library which the cell is linked to. If the library does not have default operating conditions, no operating conditions are used. User can specify analysis type in the design constraint file.

6.7 Tutorial of Using PimeTime

The following steps show the flow of STA using PT. The example used is the adder32 and its netlist is got from DC.

6.7.1 Preparations

1. Use the project directory of DC.
2. Creating the setup file as below, and save it as *.synopsys_pt_setup* file in the project directory.

```
set_search_path "/path to the installation directory of foundry/synopsys/c35_3.3V\
 /path to the project directory/mapped/db"
set link_path "* c35_CORELIB.db c35_IOLIB_4M.db"
```

3. Create a constraint file as below, and save it as adder32_pt_constr.scr in the scripts directory.

```
current_design adder32
reset_design
create_clock -per 100 -name clk [get_ports CLOCK]

set_dont_touch_network [get_ports CLOCK]
set_clock_uncertainty -setup 0.3 [get_ports CLOCK]
set_clock_uncertainty -hold 0.3 [get_ports CLOCK]

[33]set_operating_conditions -analysis_type single WORST -library c35_CORELIB
set_wire_load_model -library c35_CORELIB -name 10k
set_wire_load_mode enclosed

set_input_delay -max 2 -clock clk [all_inputs]
set_input_delay -min 0.4 -clock clk [all_inputs]
remove_input_delay [get_ports CLOCK]
set_driving_cell -library c35_CORELIB -lib_cell BUF8 [all_inputs]
remove_driving_cell [get_ports CLOCK]

set_output_delay -max 0 -clock clk [all_outputs]
set_output_delay -min 0 -clock clk [all_outputs]

set_load  0.1 [all_outputs]
```

6.7.2 Invoking PrimeTime GUI and Verify Setup

1. Start primetime with the command **primetime**. Figure 6-8 shows PT GUI. Notice that there are three main part on the interface - *Logical Hierarchy*, *log* and *pt-shell>* prompt.
   %primetime&

[33] Note the difference from adder32_dc_constrc.scr.

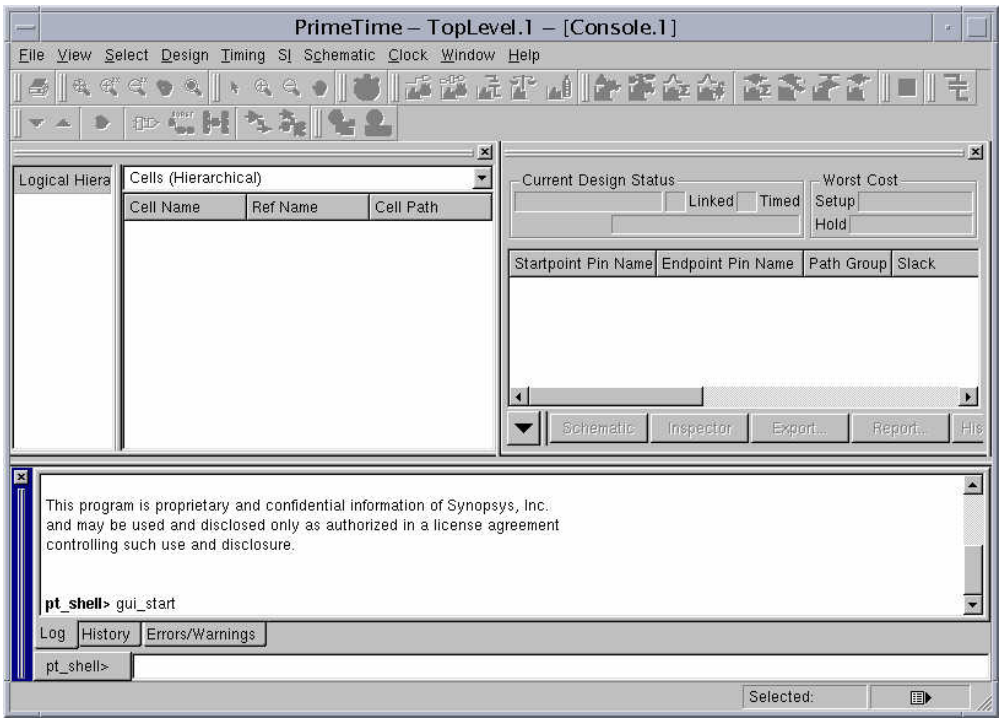


Figure 6-8 PT GUI.

2. Locate the command line (pt_shell prompt) to verify the environment with the following command.
   pt_shell>printvar link_path
   pt_shell>printvar search_path

```
pt_shell> printvar link_path
link_path          = "* c35_CORELIB.db c35_IOLIB_4M.db"
pt_shell> printvar search_path
search_path        = "/app21/AMS_3.60_CDS_F/synopsys/c35_3.3V /app22/synopsys/synthesis_200406sp1/packages/IEEE/lib /home/staff/elezhq/ASIC/synthesis/mapped/db"
```

Figure 6-9 Messages of the log area.

6.7.3 Reading, Constraining and Checking Design

1. Read the design netlist with the command: read_db. Figure 6-10 shows the message in the log area.
   pt_shell>read_db adder32_mapped.db

```
pt_shell> read_db adder32_mapped.db
Loading db file '/home/staff/elezhq/ASIC/synthesis/mapped/db/adder32_mapped.db'
1
Loading db file '/app21/AMS_3.60_CDS_F/synopsys/c35_3.3V/c35_CORELIB.db'
Loading db file '/app21/AMS_3.60_CDS_F/synopsys/c35_3.3V/c35_IOLIB_4M.db'
Linking design adder32...
Information: Issuing set_operating_conditions equivalent to timing_propagate_single_condition_min_slew setting. (PTE-037)
set_operating_conditions -analysis_type on_chip_variation  -library [get_libs {c35_CORELIB.db:c35_CORELIB}]
```

Figure 6-10 Messages of read_db .

2. Link the design with the command: link. The Message "Design 'adder32' was successfully linked" appears in the log window after the design is linked.
   pt_shell>link
3. Constrain the design with the constraint file. On the primetime window, click on **File→Execute Script….** The Execute Script File window appears as shown in figure 6-11.

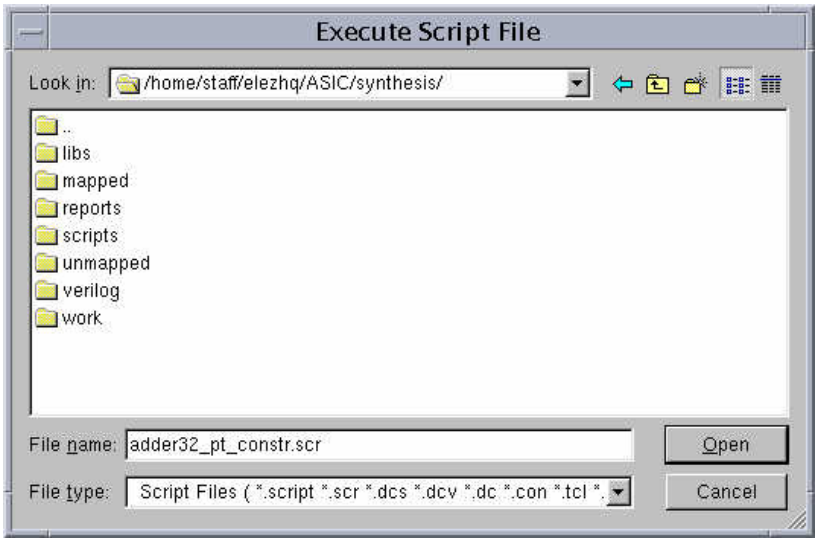


Figure 6-11 Execute Script File window.

4. On the above window, click on the folder - scripts and choose the file: adder32_pt_constr.scr and then click on **Open.**
5. Check clock applied. The report appears in the log window as shown in figure 6-12.
   pt_shell>report_clock

```
pt_shell> report_clock
****************************************
Report : clock
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Nov 10 16:10:52 2005
****************************************


Attributes:
   p - Propagated clock
   G - Generated  clock
   I - Inactive   clock

Clock        Period   Waveform        Attrs    Sources
--------------------------------------------------------------------------------
clk          100.00   {0 50}                   {CLOCK}


1
```

Figure 6-12 Messages of report_clock.

6. Check the constraints applied. Manage to solve any warning and error if there was.
   pt_shell>check_timing –verbose

```
pt_shell> check_timing -verbose
Information: Using automatic max wire load selection group 'sub_micron'. (ENV-003)
Warning: Some timing arcs have been disabled for breaking timing loops
         or because of constant propagation. Use the 'report_disable_timing'
         command to get the list of these disabled timing arcs. (PTE-003)
Information: Checking 'no_clock'.
Information: Checking 'no_input_delay'.
Information: Checking 'partial_input_delay'.
Information: Checking 'no_driving_cell'.
Information: Checking 'unconstrained_endpoints'.
Information: Checking 'unexpandable_clocks'.
Information: Checking 'generic'.
Information: Checking 'latch_fanout'.
Information: Checking 'loops'.
Information: Checking 'generated_clocks'.
check_timing succeeded.
1
```

Figure 6-13 Messages of check_timing in log area.

6.7.4 Analyzing Design

1. Analyze the design timing using the Endpoint Slack Histogram.
   On GUI, click on **Timing→Histogram→Endpoint Slack…**. The Endpoint Slack window appears as shown in figure 6-14.

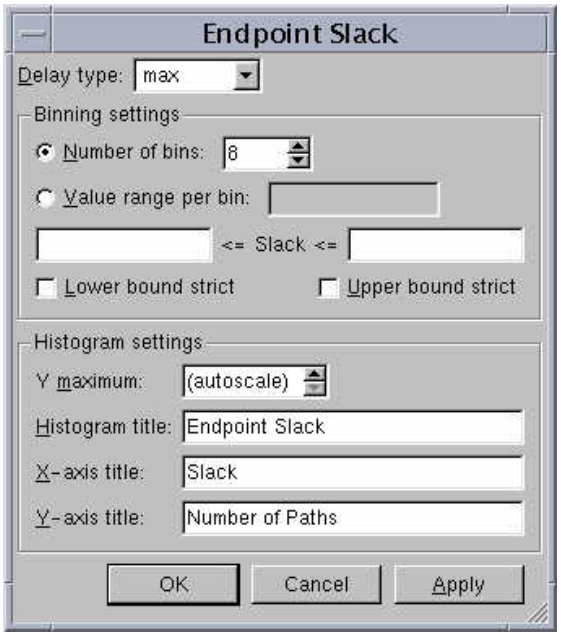


Figure 6-14 Endpoint Slack window.

2. Keep the default settings and Click on **Ok** on the endpoint slack form. The endpoint slack window appears as shown in figure 6-15.

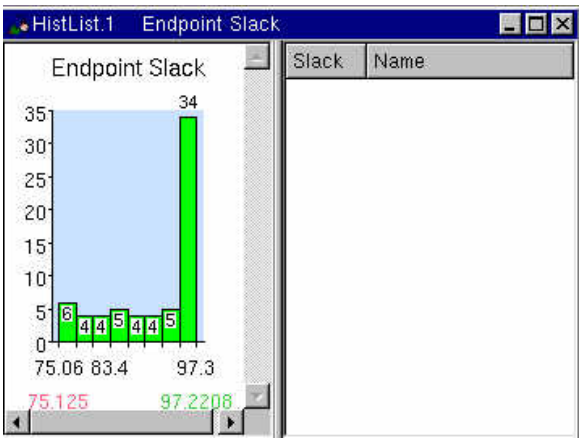


Figure 6-15 The endpoint slack.

3. Click on the left most one on the endpoint slack window. The 6 endpoints with their respective slacks appear on the right of the window, shown in figure 6-16.

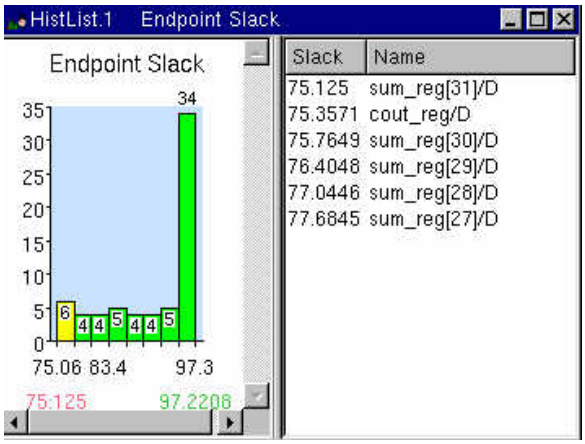


Figure 6-16 Illustration of slack histogram.

4. To investigate a path, highlight it, and then right click on your mouse and choose **Inspector for Worst Path to Selected Pin**. For example, highlight the top most entry on the right hand. The **PrimeTime TopLevelPathInspector** window appears, shown in figure 6-17.

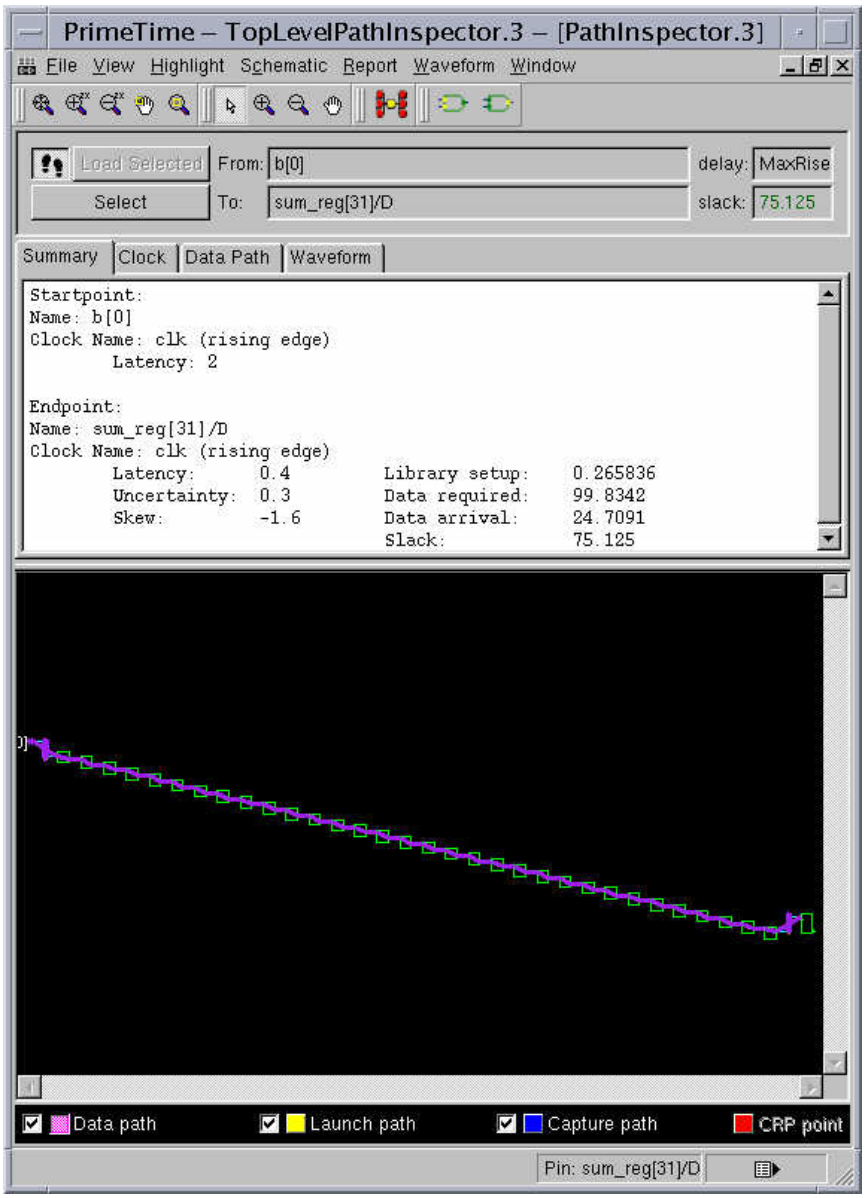


Figure 6-17 Path inspector window.

5. Click on the **Waveform** button on the path inspector window, the waveform of the specified path appears, shown in figure 6-18.

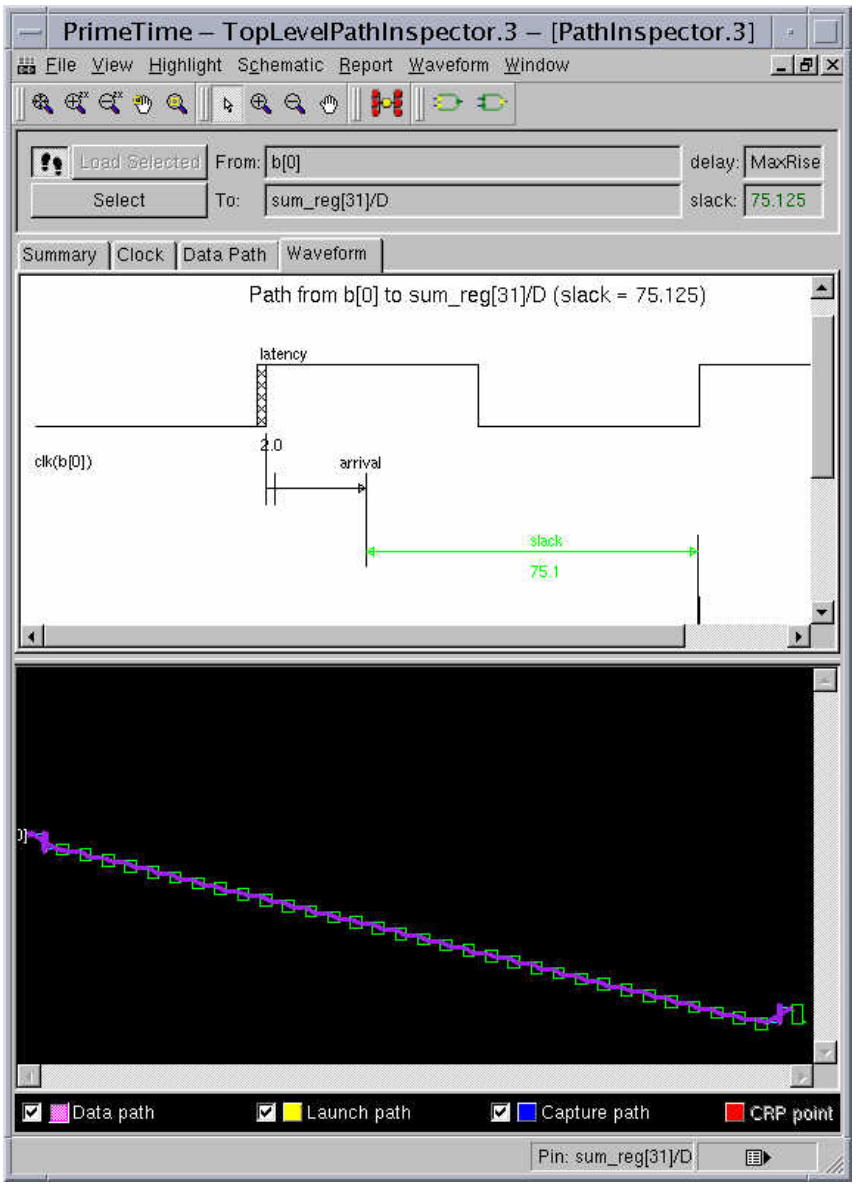
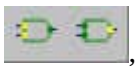


Figure 6-18 The waveform of the specified path.

6. Click on the button  , more information related to the waveform will be shown.
7. Click on the **CLOCK** and **Date Path** button, to view other information.

6.7.5 Generating Reports

1. Generate the timing report with the following command.
   pt_shell>report_timing

```
pt_shell> report_timing
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Nov 17 13:42:57 2005
****************************************


 Startpoint: b[0] (input port clocked by clk)
 Endpoint: sum_reg[31]
           (rising edge-triggered flip-flop clocked by clk)
 Path Group: clk
 Path Type: max


 Point                                    Incr      Path
 -----------------------------------------------------------------
 clock clk (rise edge)                     0.00      0.00
 clock network delay (ideal)                2.00      2.00
 input external delay                      2.00      4.00 f
```

Figure 6-19 Timing report in the log area.

2. Generate a constraint report:
   pt_shell>report_constraint –all_violators
3. Restrict the report to timing violation only, use the following command.
   pt_shell>report_constraint –all –max_delay –min_delay
4. To analyze timing bottlenecks in primetime GUI, click on
   Timing→Histogram→Timing Bottleneck.

6.7.6 Exit PrimeTime

To exit primetime, do one of the followings.

- **pt_shell>exit** or
- On primetime GUI, **File→Exit**.

6.8 Conclusion

The concept and usage of PT are described in this chapter. As mentioned, there are 4 types of STA, user needs to decide which type to use when doing STA. Exceptions and analysis type can be defined in a script file – constraint file, for easy to process.

It is known that DC can check timing, but PT is more powerful than DC for check timing. PT functions can be further explored by playing around with the pull down menu and tool icons. The online help and manpage are very helpful when there is a doubt.

Next, the design source should be brought to cadence silicon ensemble for place and route if the timing is satisfactory. Otherwise, user may need to go back to DC to further optimize the design, referring to the ASIC design flow stated in chapter 1.

7. Place and Route with Cadence Silicon Ensemble

Place and route are described in this chapter. Silicon ensemble (SE) is an area based standard cell place and route tools – no channels are used and therefore also no compaction after routing is possible. The router will not change the placement of the cells. It tries to route in the given area. User can include some blocks in the placement, but SE is mainly a standard cell router.

In this manual, designs with only standard cell are considered. The arrangement is as follows. In section 7.1, the overview of SE flow is introduced. A simple introduction of SE graphic interface and online help is given in section 7.2. The introduction to the starting scripts of AMS kits is presented in section 7.3. The tutorial of using SE to place and route with AMS design kits is demonstrated in section 7.4. Finally the conclusion is given in section 7.5.

7.1 Overview of Silicon Ensemble flow

Figure 7-1 shows the production flow. The steps to do place and route are as follows.

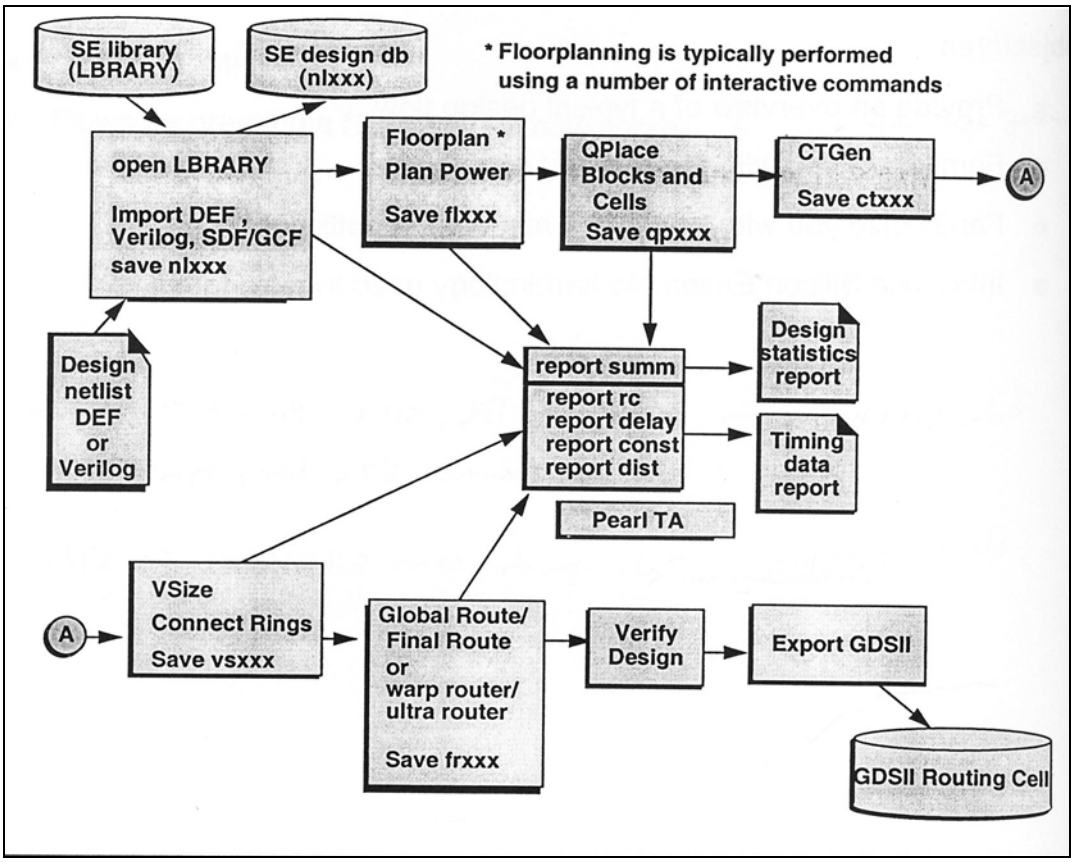


Figure 7-1 SE flow.

1. Import Library files to setup the SE library.
2. Import the design netlist (DEF and/or Verilog files).
3. Create a floorplan and pre-placement power rings.
4. Run standard cell placement with Qplace.

5. Optimize the floorplan with VSize - Optional[34]
6. Create a clock tree with CT-Gen commands.
7. Make the required power supply connections to all of the macro cells with the special router.
8. Wrouter generates a plan for signal routing and completes the inter-connection of the macro cell inputs and outputs.
9. Export parasitic RC, delays, netlist and GDSII of the routed design.

7.2 SE Graphical Interface and Online Help

7.2.2 SE Graphical Interface

Start SE with the command

seultra –m=<memSize>,

where the memSize can be 12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 150 or 200 Mbytes (depending on hardware memory limitation). The command displays the SE graphics interface window on top of other x-term windows. Figure 7-2 shows the functions of each area of the window. There are six areas in the window –Message, Artwork, Command, Context, Icon and Object Selection. To view more messages, users can enlarge the message window by dragging the widget up.

- **Message** area shows the message of each command.
- **Artwork Window** shows the design.
- **Command** field allows user to enter a command.
- **Context** area shows the relative position of the viewing area to the whole design.
- **Icon** area contains icons (commands) which are often used.
- **Object Selection** area is used for user to set what is viewable and what is selectable.

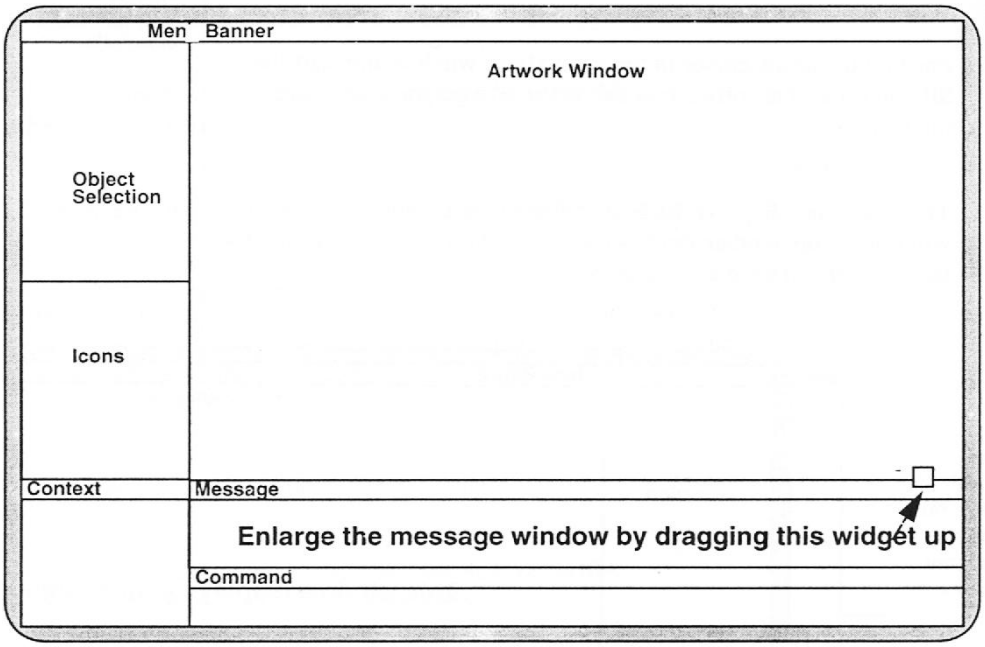


Figure 7-2 SE graphical interface.

[34] Refer to SE user guide, for floorplan optimization.

The user interface and command behavior are controlled by environmental variables. The Edit Environment (**Edit→Environment**) command allows user to

- view all variables and current settings,
- search for variables, and
- change variable settings.

7.2.2 Using Online Help

There are two types of online help:

- Help buttons;
- Form Help.

To use the help buttons, click on **Help→Getting Started** (or **Commands**) on SE window. There is also a help button at the bottom of each form. Simply clicking on **Help**, a window describing the function of the form will appear.

7.3 Introduction to the Starting Scripts of AMS Kits

AMS design kits has a script 'ams_se' to setup the environment for using AMS kits and cadence SE. The command to run the script is *ams_se –t <technology>*. The script will create the followings.

- Setup directory structure.

```
Working directory
        |_________ DB (SE database)
        |_________ VERILOG (Verilog Netlists)
        |_________ LEF (Additional LEF files)
        |_________ DEF (DEF files)
        |_________ CTGEN (CTGen Run directory)
```

- Prepare a '**se.ini**' file for initializing SE.
- Prepare **macro files**: gemma.mac, fillperi.mac, fillcore.mac.
- Prepare **gcf files** – used for importing CTLFs into SE.
- Prepare a **DEF/power_corner.def** file – template that can be used to insert power pads and corner cells into design.
- Prepare **CTGen command files**: ctgen.cmd, CTGEN/ctgen.const, ctgen_post.cmd.
- Prepare **a GDSII Map File**: gds2.map.

The above directories and files will be used in the place and route. The '.mac' file is command file. It can be executed after having started SE by clicking on **File→Execute…**. Users need to modify those '.mac' files accordingly to their design. It is best for users to have a look at the 'gemma.mac' file.

7.4 Tutorial of Using Silicon Ensemble with AMS Kits

The whole design flow will be demonstrated in this section. The sample used is the adder32 whose netlist is got from chapter 4 and has passed the pre-layout verification and pre-layout STA. The design kits used is AMS CMOS 0.35um. For users who are using other foundry, please refer to the foundry vendor for ENV setup.

7.4.1 Setup for Using SE and AMS Kits

1. Create a directory.
   % **mkdir layout**
   % **cd layout**
2. Run script: **ams_se –t c35b4** (c35b4 is technology used) in the directory. Figure 7-3 shows the output of the command.

```
ss1% ams_se -t c35b4
*** Directory DB created
*** Directory CTGEN created
*** Directory VERILOG created
*** Directory LEF created
*** Directory DEF created

*** Creating a new se.ini file...

*** Creating a new gemma.mac file...
*** Creating a new fillperi.mac file...
*** Creating a new fillperi_c.mac file...
*** Creating a new fillcore.mac file...

*** Creating a new c35b43.3V.gcf file
*** Creating a new ./DEF/power_corner_c.def file
*** Creating a new ./DEF/power_corner.def file
*** Creating a new ./ctgen.cmd file
*** Creating a new ./ctgen_post.cmd file
*** Creating a new ./ctgen.const file
File pearl5.0V.cmd not created

*** Creating a new pearl3.3V.cmd file...

*** Creating a new gds2.map file...
*** Creating a c35_cell.map file
```

Figure 7-3 Output of script - *ams_se*

3. Copy the LEF files needed by the design to the LEF directory.
   % **cp /kit_install_directory/artist/HK_C35/LEF/c35b4/c35b4.lef LEF/.**
   % **cp /kit_install_directory/artist/HK_C35/LEF/c35b4/CORELIB.lef LEF/.**
   % **cp /kit_insstall_directory/artist/HK_C35/LEF/ c35b4/IOLIB_4M.lef LEF/.**
4. Copy design netlist to VERILOG directory.

7.4.2 Loading LIBRARY

1. Start SE with the command: **seultra –m=200&.** The SE window appears as shown in figure 7-4.

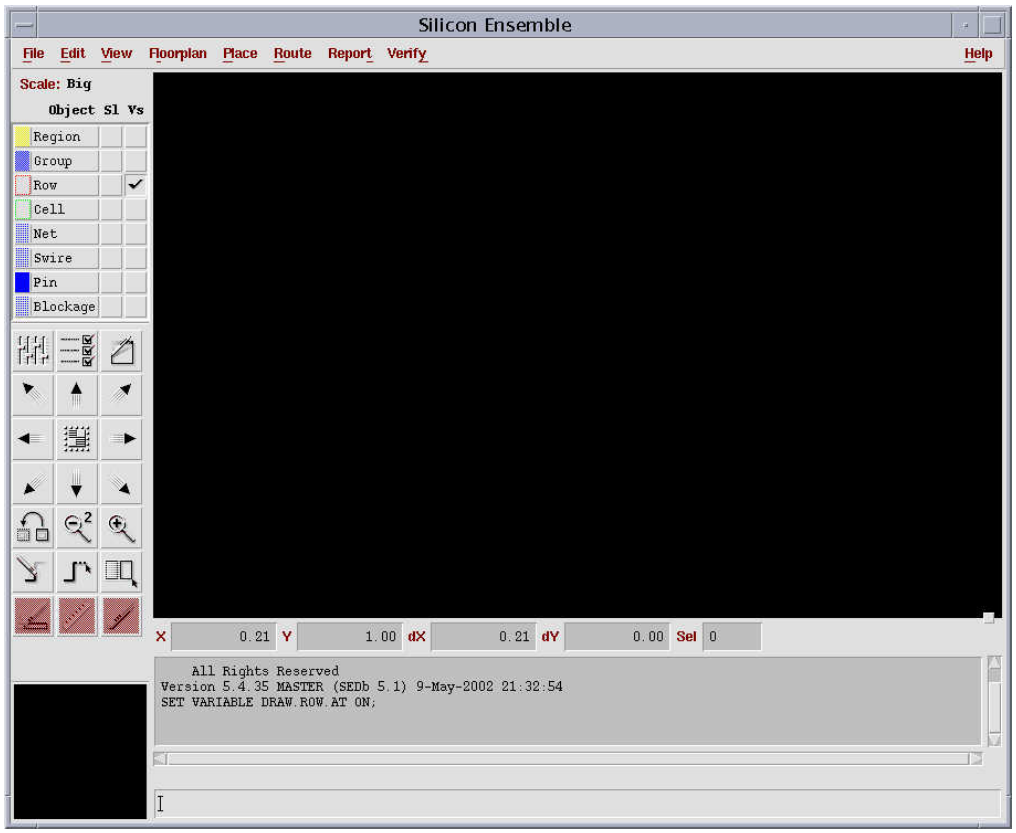


Figure 7-4 SE graphic interface.

2. To load the **LEF** files which include the descriptions of the technology used, click on **File→Import→LEF…** and the **Import LEF** form appears as shown in figure 7-5. Choose **c35b4.lef** and then click on **Ok**.

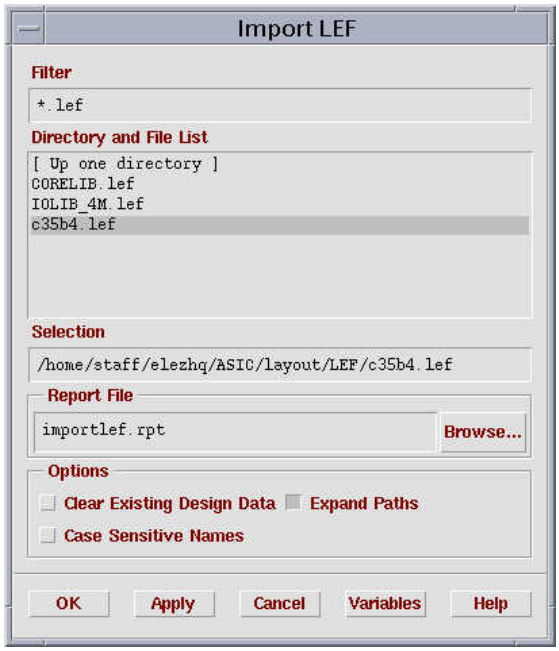


Figure 7-5 Import LEF form.

Note: repeat this step for the standard cell library needed by the design. For this example, **CORELIB.lef** and **IOLIB_4M.lef** must be imported.

3. To import the timing data of the technology, click on **File→Import→Timing Library…** and choose **c35b43.3V.gcf** as shown below. Then click on **Ok**.

Figure 7-6 Import timing data form.

4. To load the models of the digital standard cells from the foundry, click on **File→Import→Verilog…** and then click on **Browse…** on the **ImportVerilog** form. Choose the standard cells needed by the design. The models chosen for this example are shown in figure 7-7.

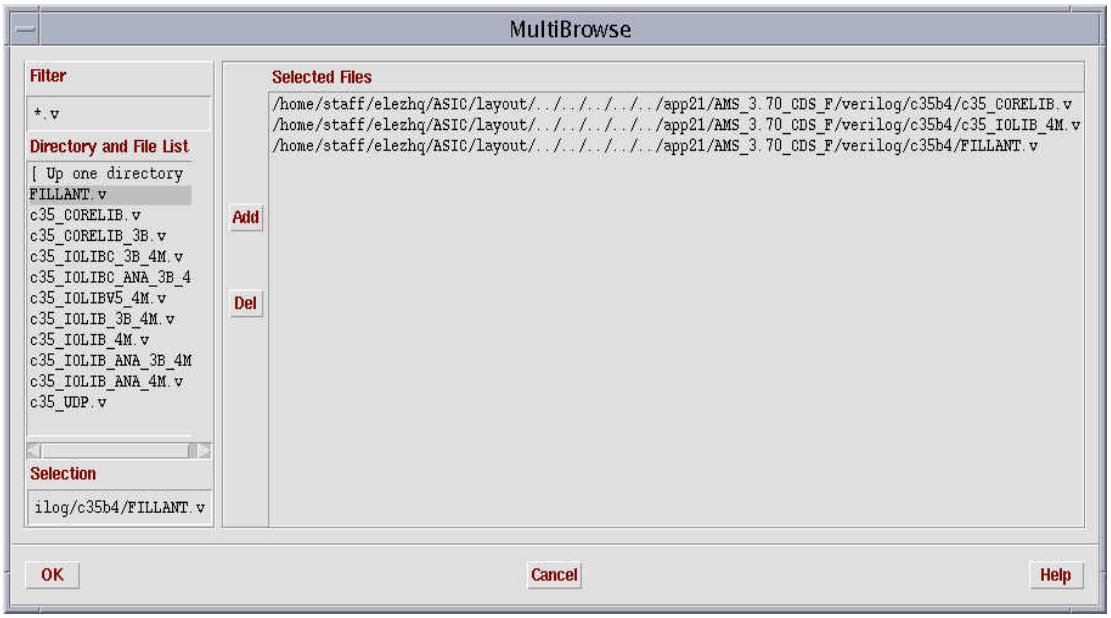


Figure 7-7 Load Verilog code of digital standard cells.

5. Click on **Ok** on the **MultiBrowse** Form.
6. Type **DesignLib** in the **Compiled Verilog Output Library** field of **ImportVerilog** form as shown in figure 7-8, and then click on **Ok**.

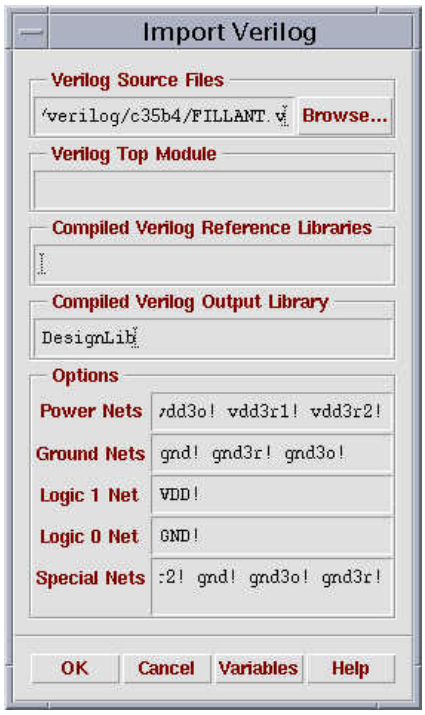


Figure 7-8 ImportVerilog form settings.

7. Save current design as Lib_adder32 by selecting **File→Save As,** and type **Lib_adder32** in the **Design Name** field as follows.

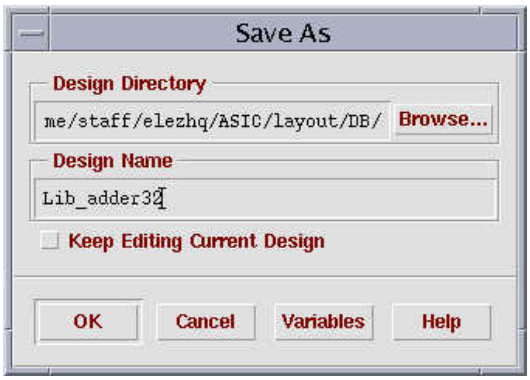


Figure 7-9 Save As form.

Lib_adder32 is the library for the design. It is saved in DB format, and can be loaded in future when necessary.

7.4.3 Importing Design and Initializing Floorplan

Start SE and load the library – Lib_adder32 saved in section 7.4.2 by clicking on **File→Open…** if SE is closed in the end of last section.

1. To load design, click on **File→Import→Verilog…**. Fill the first four fields of **ImportVerilog** form as figure 7-10, and then click on **Ok**.

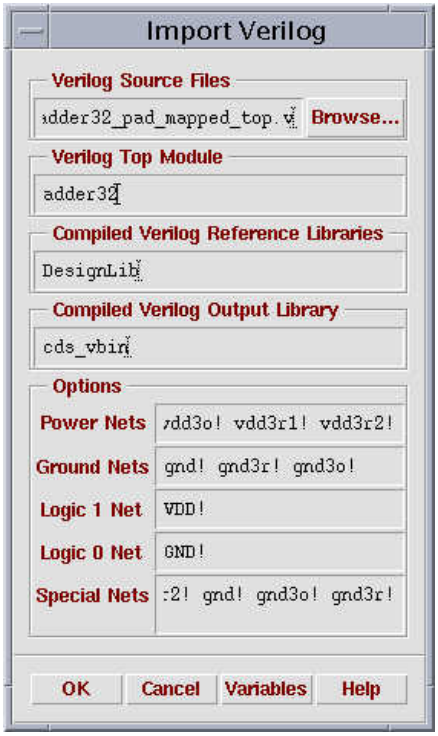


Figure 7-10 Import design.

2. To insert power pads, modify the file - **power_corner.def** which is created with the script *ams_se*. Change the lines with PWR4 and GND4 as follows.

```
#- GND3    GND3RP ;
- GND4    GND3ALLP ;
#- PWR1    VDD3IP ;
#- PWR2    VDD3OP ;
#- PWR3    VDD3RP ;
- PWR4    VDD3ALLP ;

END COMPONENTS

SPECIALNETS 7 ;

- vdd3r1! ( CORNER1 vdd3r1! ) ( CORNER2 vdd3r1! )
          ( CORNER3 vdd3r1! ) ( CORNER4 vdd3r1! )
          ( PWR4 vdd3r1! )
          ( GND4 vdd3r1! )
 ;
- vdd3r2! ( CORNER1 vdd3r2! ) ( CORNER2 vdd3r2! )
          ( CORNER3 vdd3r2! ) ( CORNER4 vdd3r2! )
          ( PWR4 vdd3r2! )
          ( GND4 vdd3r2! )
 ;
- vdd3o!  ( CORNER1 vdd3o! ) ( CORNER2 vdd3o! )
          ( CORNER3 vdd3o! ) ( CORNER4 vdd3o! )
          ( PWR4 vdd3o! )
          ( GND4 vdd3o! )
 ;
- gnd3o!  ( CORNER1 gnd3o! ) ( CORNER2 gnd3o! )
          ( CORNER3 gnd3o! ) ( CORNER4 gnd3o! )
          ( PWR4 gnd3o! )
          ( GND4 gnd3o! )
 ;
- gnd3r!  ( CORNER1 gnd3r! ) ( CORNER2 gnd3r! )
          ( CORNER3 gnd3r! ) ( CORNER4 gnd3r! )
          ( PWR4 gnd3r! )
          ( GND4 gnd3r! )
 ;
- vdd!
       ( PWR4 A )
  + SPACING MET1 800 RANGE 10000 10000000
  + SPACING MET2 800 RANGE 10000 10000000
  + SPACING MET3 800 RANGE 10000 10000000
  + SPACING MET4 800 RANGE 1000 1000000 ;
- gnd!
       ( GND4 A )
  + SPACING MET1 800 RANGE 10000 10000000
  + SPACING MET2 800 RANGE 10000 10000000
  + SPACING MET3 800 RANGE 10000 10000000
  + SPACING MET4 800 RANGE 1000 1000000 ;
```

Figure 7-11 Modification of power_corner.def file.

Note: Modify the power_corner.def file accordingly if more power pads are needed.

3. To Load the DEF file, click on **File→Import→DEF…**. Select the power_corner.def in the **Import DEF** form, and then click on **Ok**.

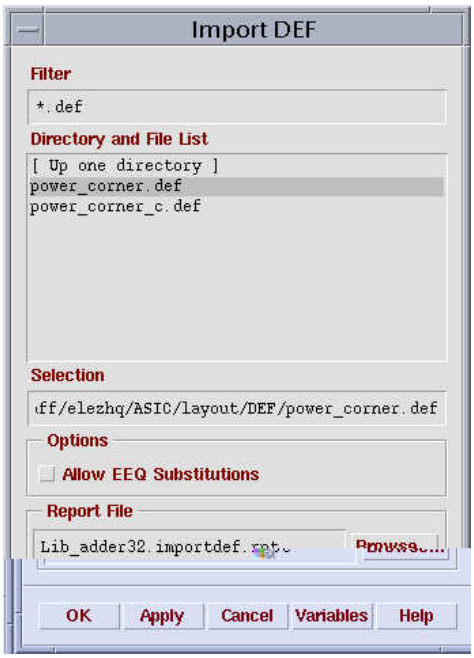


Figure 7-12 Insert power pads.

4. Click on **Floorplan→Initialize Floorplan** on the SE window, to initialize the floorplan. The **Initialize Floorplan** form appears. Change its settings and click on **Calculate** button. The form will looks like figure 7-13.

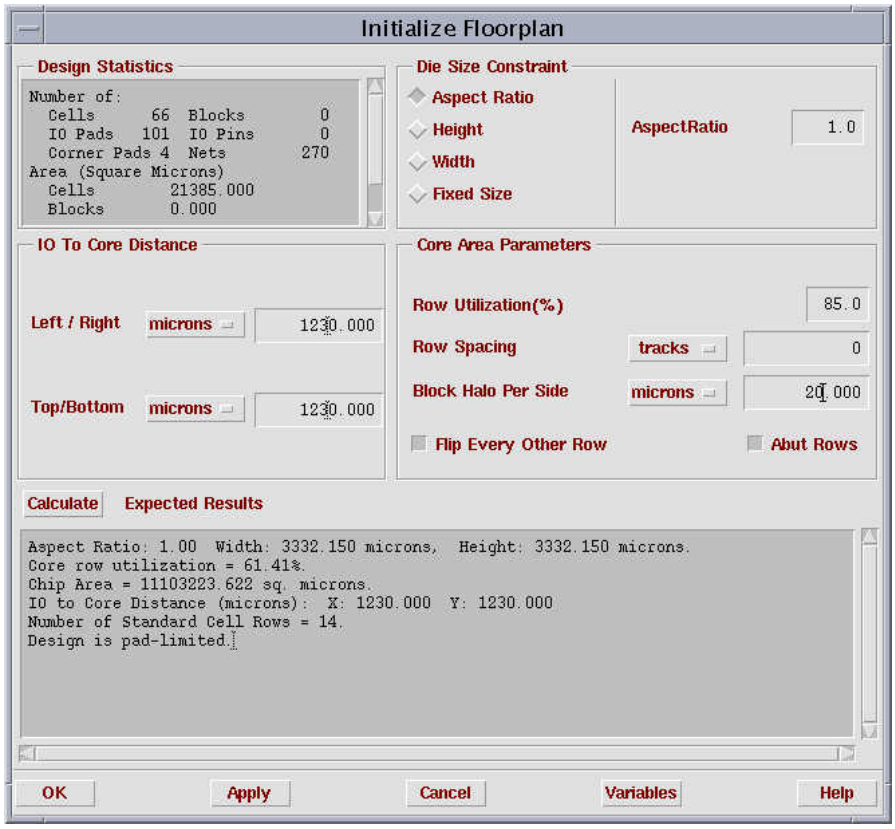


Figure 7-13 Initializing floorplan.

Note:

- It is better to get a row utilization value 0.6~0.85.
- Click on **Help** button on the form to get the explanation of the form.

5. Click on **Ok** on the **Initialization Floorplan** form. A floorplan appears in the artwork Window as shown in figure 7-14.

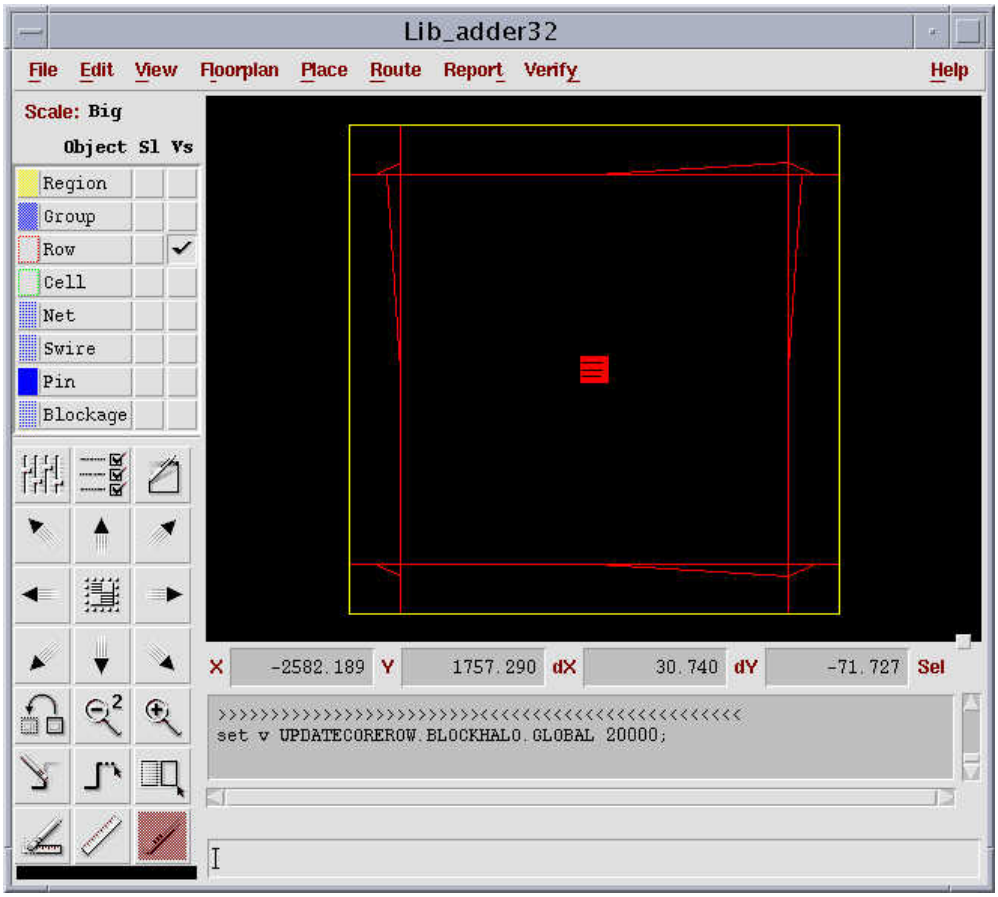


Figure 7-14 Initial floorplan.

Note:

- The example serves illustration purpose only. User should limit the number of I/O, for better performance.

6. To verify the floorplan, click on **Verify→Floorplan…** and click on **Ok** on the **Verify Floorplan** form as figure 7-15. User can change the numbers of errors and warnings.

Figure 7-15 Verify floorplan form.

7. Click on **Report→Infos…** and then click on **Ok** to save the verifying floorplan report to a file. Read the report to check if there is any error. User can optimize their floorplan with **VSize**, referring to SE user guide.

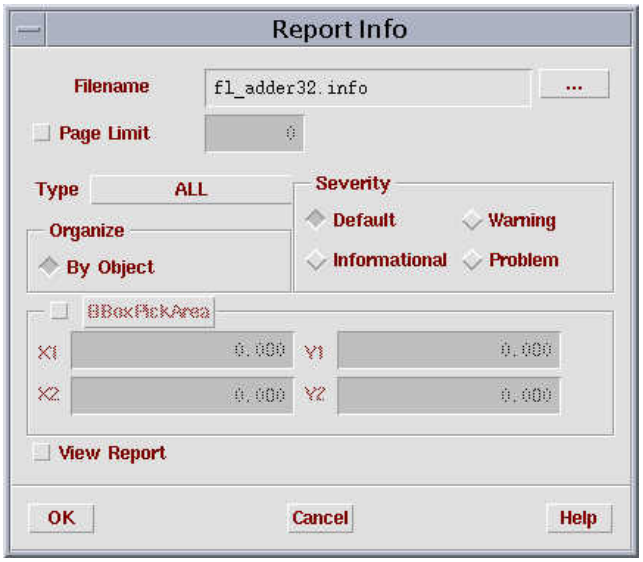


Figure 7-16 Report Info form.

8. To save the floorplan for later use, click on **File→Save As…** and type a name in the **Save As** form as follows.

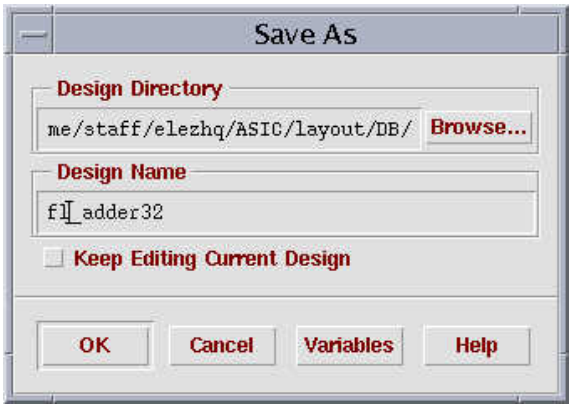


Figure 7-17 Save floorplan.

7.4.4 Viewing the Floorplan

1. To make the floorplan visible, click on the box for **Row Visibility** (Vs) in the **Object Selection** (OS) area.
2. Use the **Fit** or **Redraw** icon to set the artwork window size and cause a refresh.
3. Observe the context window. It contains two rectangles. One rectangle encloses the other rectangle. The white rectangle represents the artwork window and yellow rectangle represents the chip.
4. Use the **Zoom In** and **Zoom Out** icons to zoom into or zoom out from the chip.
5. Click on the **Display Option** icon (in the center, above the Pan icons). The Display Options form appears.

6. To make the row names viewable, do the following
   - Move the **Display Options** form to the right so that you can see the artwork window.
   - Click on **Rows** under the **Names** section (at the left center of the form), then click on **Apply**.

   The names of the rows appear above each row in the artwork window.
7. Make rows selectable in the OS area, and click on a row. Notice that the number of row selected appears in the message window.
8. Use the **Properties** icon (to the left of the Display Options icon) to view properties associated with a specific row.
9. Go back to the full view of the chip. To fill the artwork window with chip, click **Fit** icon.
10. To turn the row names off, click on **Rows** under the **Names** section (Display Options form), then click on **Ok**.

7.4.5 Power Planning

Add power strips and power rings accordingly to individual design requirements. For the design example here, only power ring is needed. Refer to the SE user guide or online help for details of power planning.

Start SE and load fl_adder32 if SE is closed in the end of last section.

1. Choose **Route→Create Ring…**, and the **Create Ring** form appears. Set the form as figure 7-18 and click on **Ok**. The design with power rings (vdd! and gnd!) appears as figure 7-19.

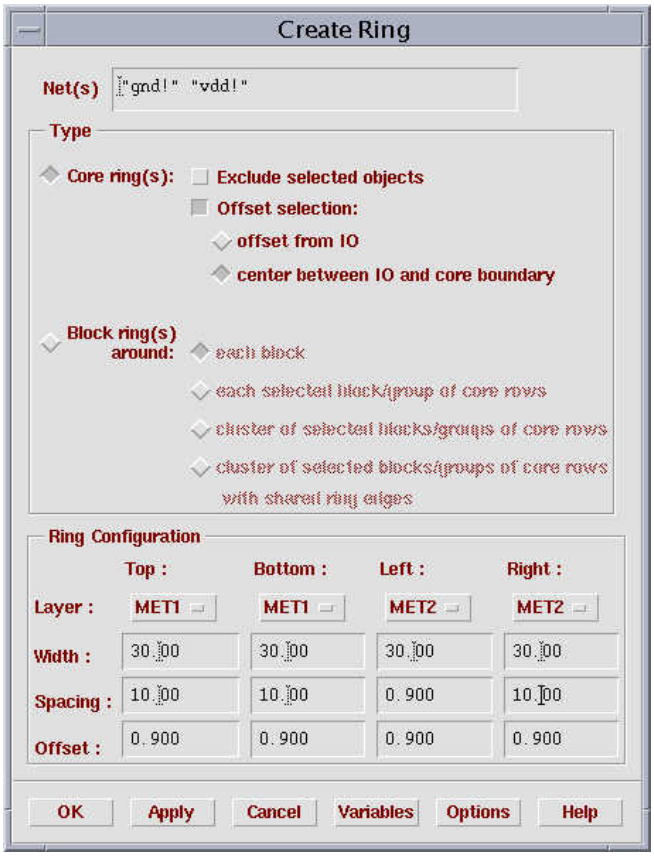


Figure 7-18 Create Ring settings.

Note:

- Users should put the nets' name as same as that of their netlists.
- The values of metal width and spacing should meet the DRC rule.
- Click on the Help button to explore more of the Create Ring form.

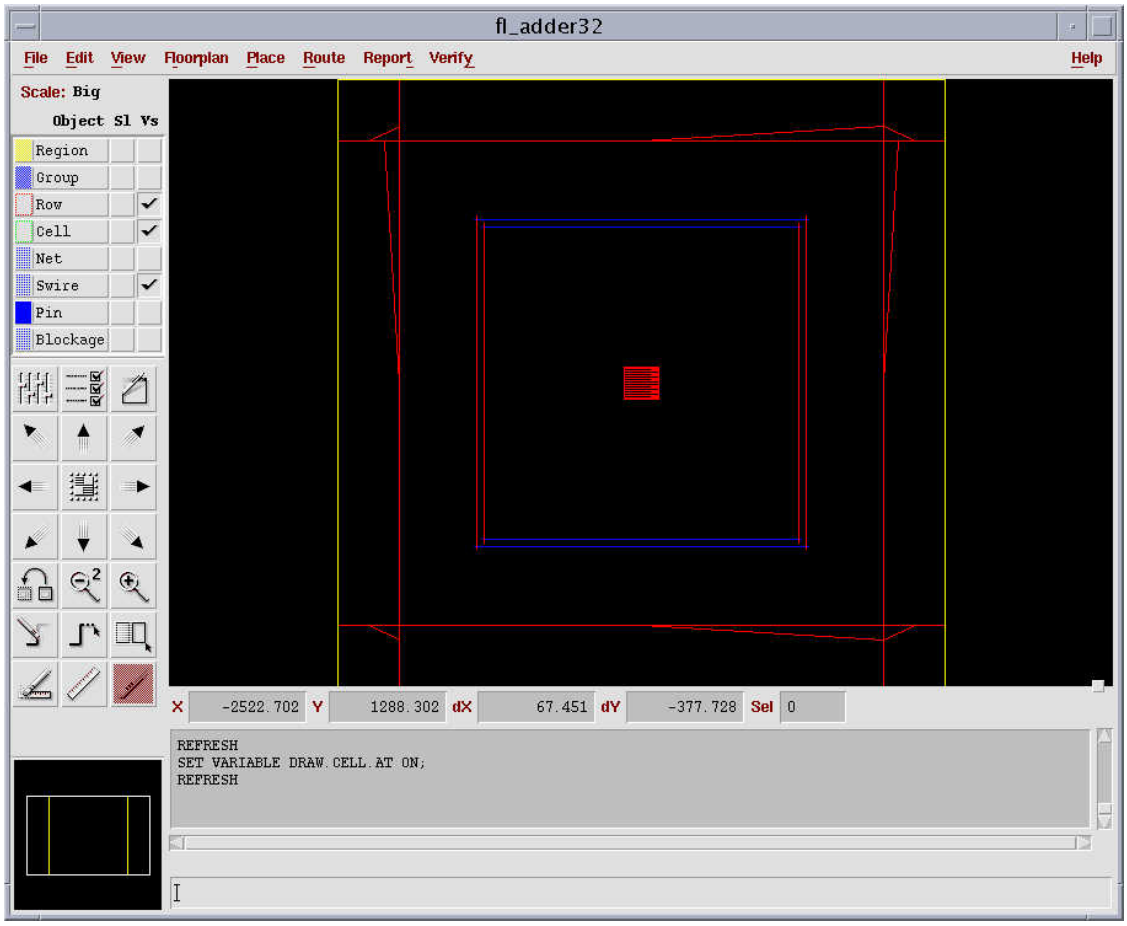


Figure 7-19 Design with power rings.

2. Save the design.

7.4.6 Place Cells

Start SE and load fl_adder32 if SE is closed in the end of last section.

1. Place Periphery (I/O) cells
   Click on **Place→IOs…**. The **Place IO** form appears. Set the form as same as figure 7-20, and then click on **Ok**. The design in the artwork window is as figure 7-21.

Figure 7-20 Place IOs.

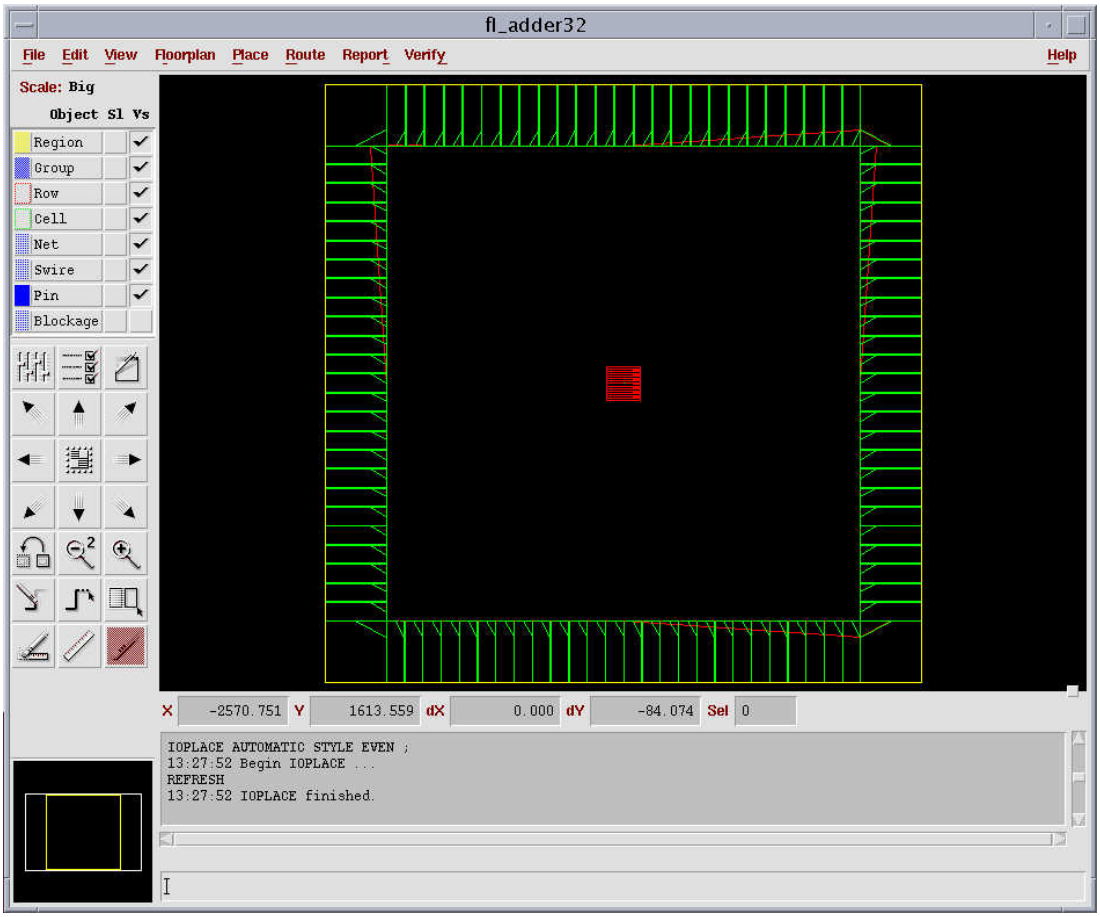


Figure 7-21 The design after placing I/O cells.

Note: User may like to refine the IO cell placement. To do it, check **I/O Constraint File** under the **Placement Mode** in the **Place IO** form, and then click on **Write**. A file named ioplace.ioc with all the IO cells' name is created. User can modify the file and change the order of the cells. Finally, run IO placement in the mode of **I/O Constraint File.**

2. Place CAP cells[35]
   Create a file named capcell.mac as figure 7-22. Click on **File→Execute…**, and choose capcell.mac on the **Execute** form as figure 7-23, and then click on **Ok**. Figure 7-24 shows the core row area which is before and after placing cap cells.

```
##-- Add Cap cells
SROUTE ADDCELL MODEL ENDCAPL PREFIX lcap
    SPIN vdd! NET vdd! SPIN gnd! NET gnd!
    AREA ( -46000000 -46000000 ) ( 46000000 46000000 ) PREENDCAP ;
SROUTE ADDCELL MODEL ENDCAPR PREFIX rcap
    SPIN vdd! NET vdd! SPIN gnd! NET gnd!
    AREA ( -46000000 -46000000 ) ( 46000000 46000000 ) POSTENDCAP ;
```

Figure 7-22 Contents of capcel.mac.

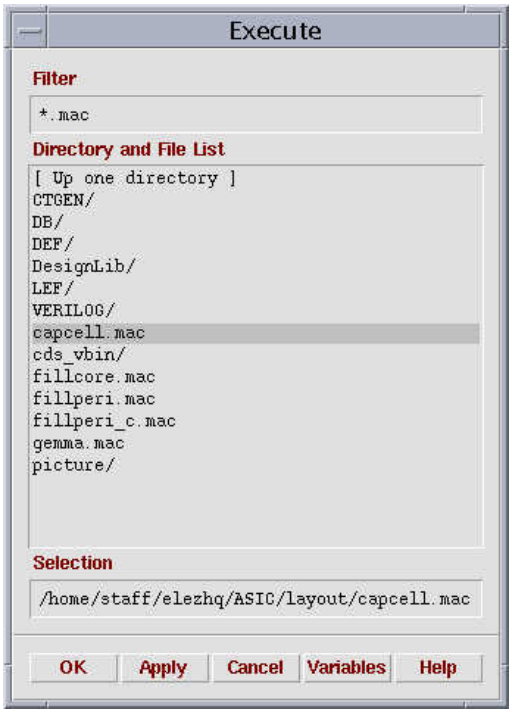
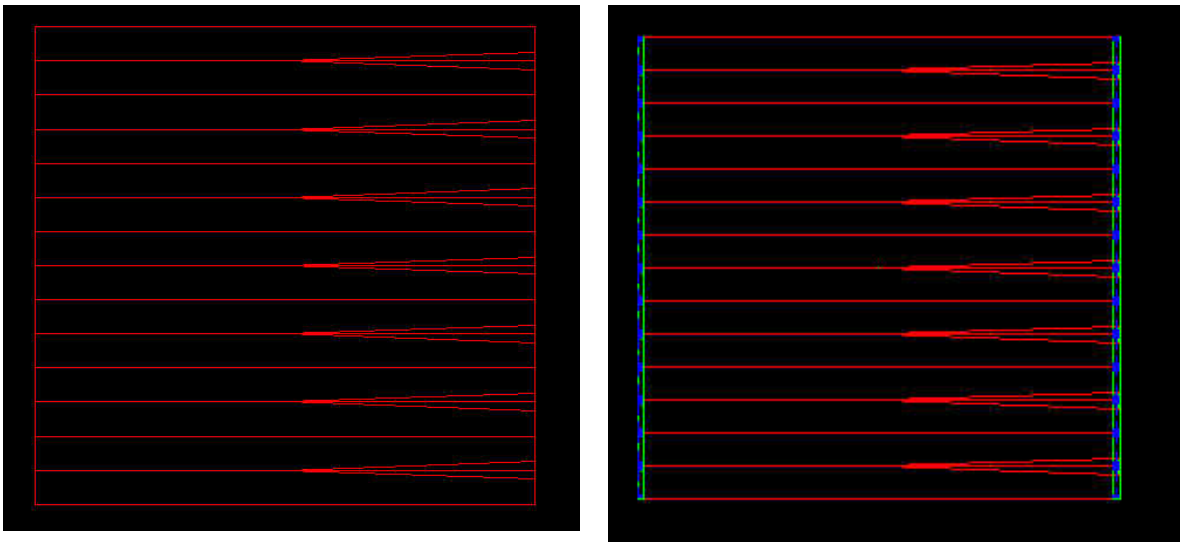


Figure 7-23 The execute form.

[35]The example is for using AMS design kits. Please refer to the respective foundry if not using AMS design kits.



(a) (b)

Figure 7-24 The core area (a) before placing the Cap cells (b) after placing the Cap cells.

3. Place standard cells
   Click on **Place→Cells…**. The **Place Cells** form appears. Set the form as figure 7-25, and then click on **Ok**.

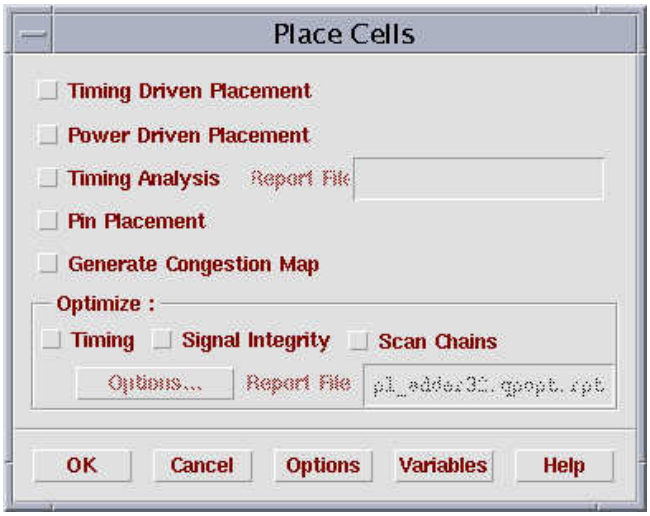


Figure 7-25 The Place Cells form.

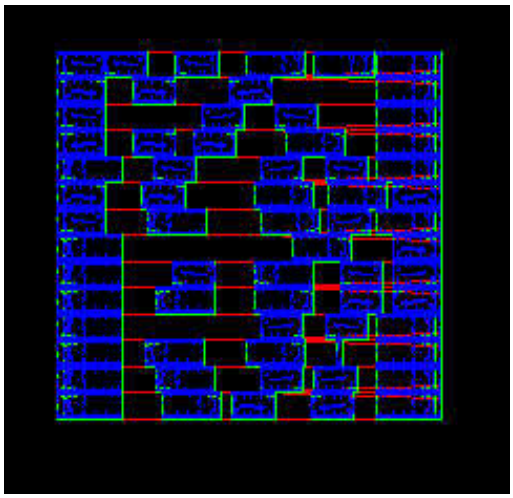


Figure 7-26 The core area after placing standard cells.

4. Save the design as pl_adder32.

7.4.7 Clock Tree Generation

Clock tree generation is described here for user reference. Load the design: pl_adder32 first.

1. Click on **Place→Clock Tree Generate (CT Gen)…**. The **CT Gen** form appears as figure 7-27.

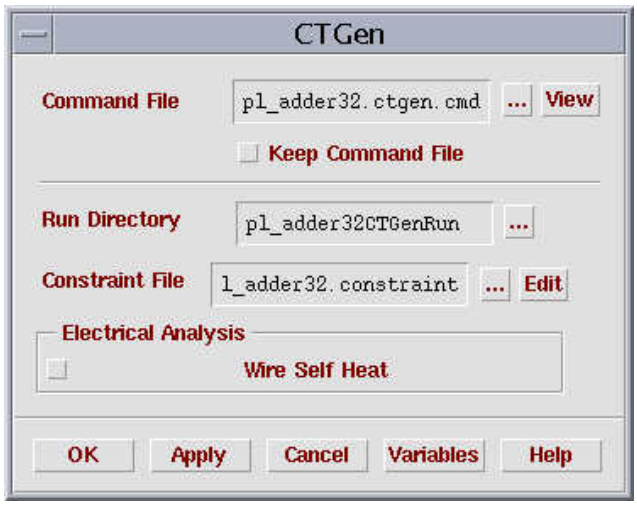


Figure 7-27 CT Gen form.

2. Click on **Edit** button on the **CT Gen** form. The **CT Gen Constraint** form appears. Set the form as figure 7-28 (user needs to consider the design clock constraint when filling up the form). Click on the Help button for more information if necessary.

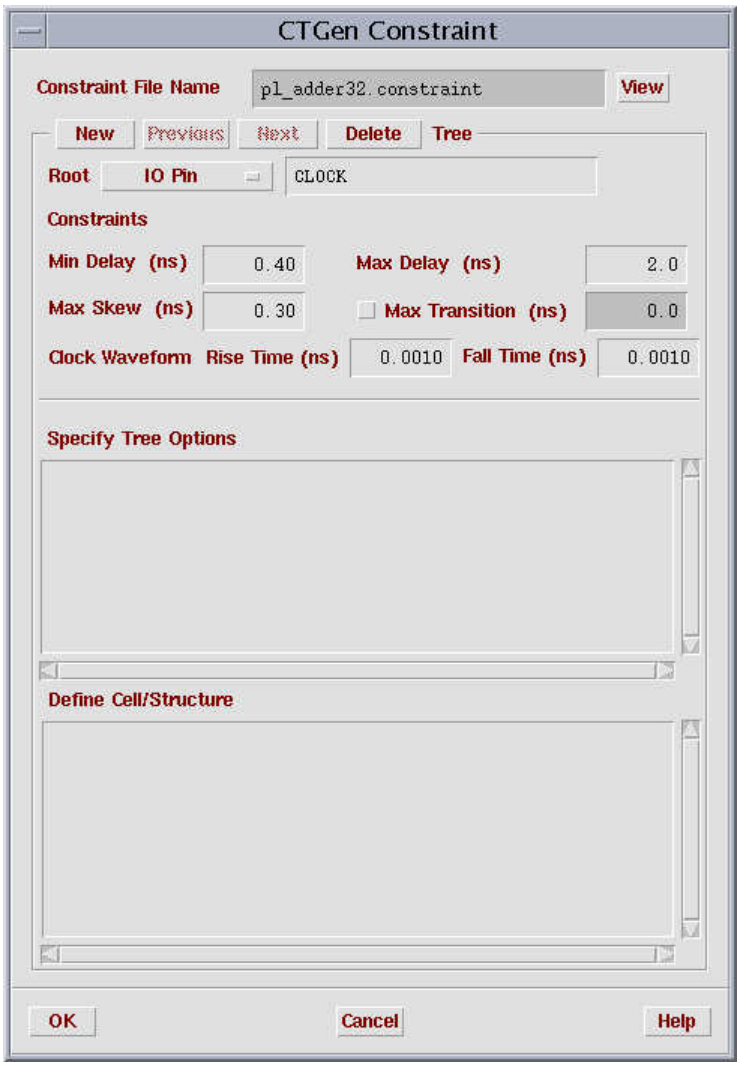


Figure 7-28 CT Gen Constraint form settings.

3. Click on **Ok** on both the **CT Gen** and **CT Gen Constraint** forms. This will create a design_nameCTGebRun (pr_adder32CTGebRun) directory and a design_name.ctgen.cmd (pr_adder32.ctgen.cmd) file.
4. Click on **Ok** on the warning window to save design. CTGen runs and executes the design_name.ctgen.cmd (pr_adder32.ctgen.cmd) file. After a few mins, the **CT Gen Results** come out as figure 7-29.

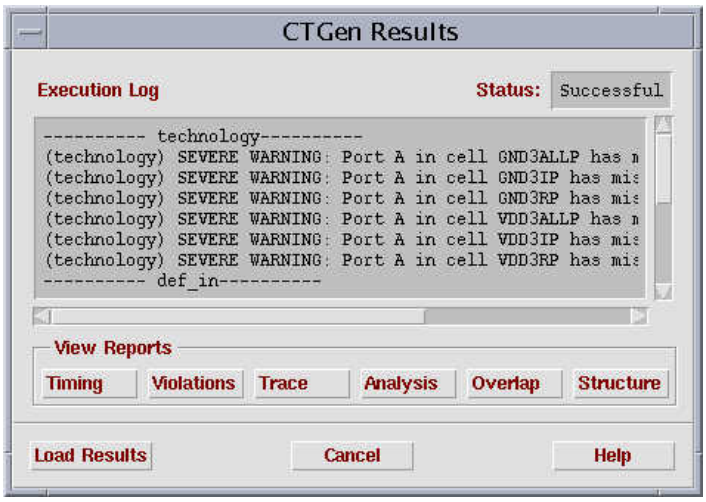


Figure 7-29 CT Gen Results form.

5. Click on **Violation** button to check if there is any violation. If there is no violation like figure 7-30, proceed to step 6. Otherwise, manage to solve it.

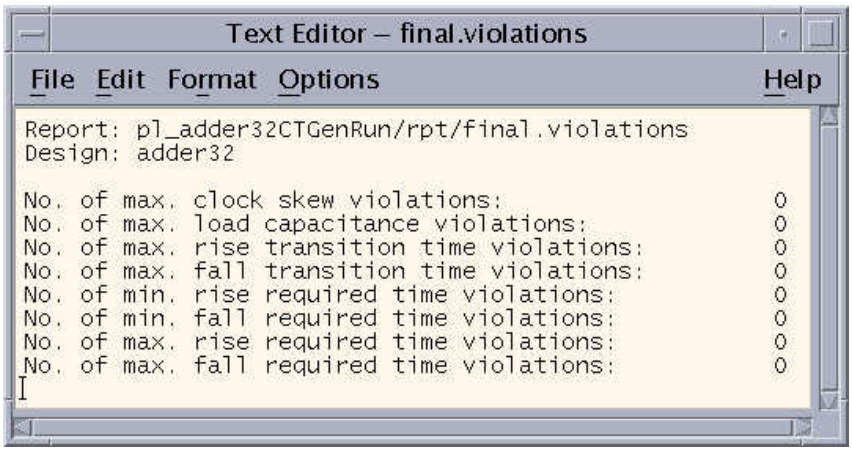


Figure 7-30 Violation report.

6. Click on the **Timing** button to check the clock timing, and the results is shown in figure 7-31. User may also view other reports by clicking on the rest buttons.

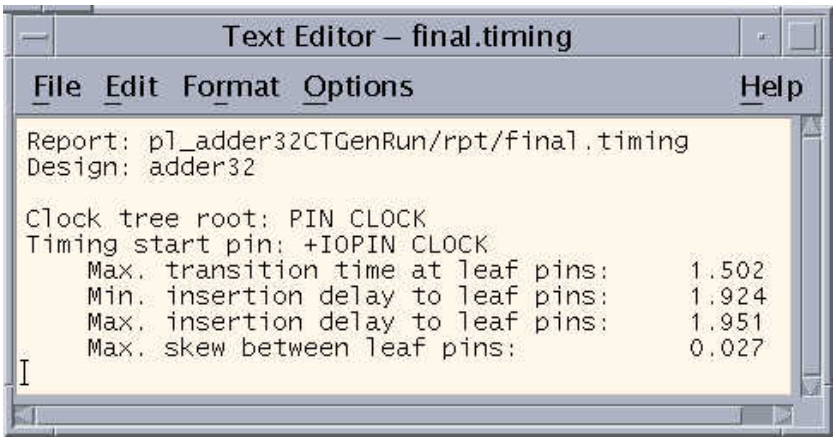


Figure 7-31 Timing report.

7. Click on **Load Results** button, and view the messages in the message area.
8. **Save** the design as **CTGen_adder32**.

7.4.8 Place Filler Cells

According to the foundry requirement, fillers have to be inserted. Load the design saved in last section.

1. Place Core Filler Cells[36]to avoid design rule violations
   Click on **File→Execute…**, and choose **fillcore.mac** on the **Execute** form as figure 7-32, and then click on **Ok**.

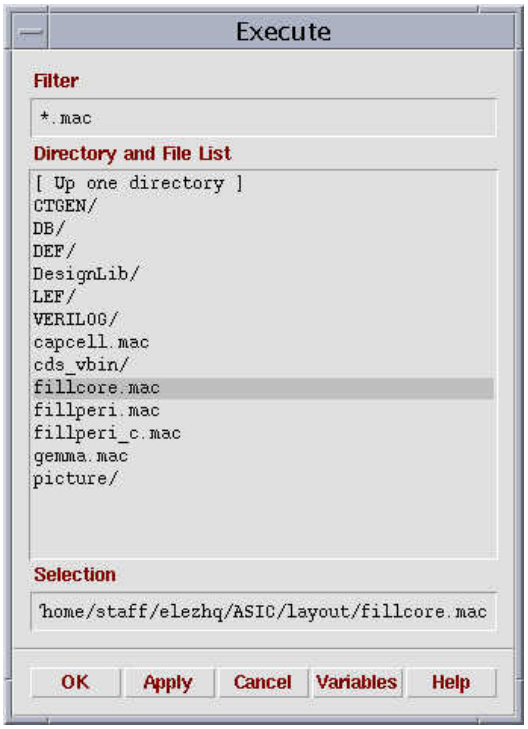


Figure 7-32 The execute form.

[36]User needs to check the foundry requirements if using other design kits.

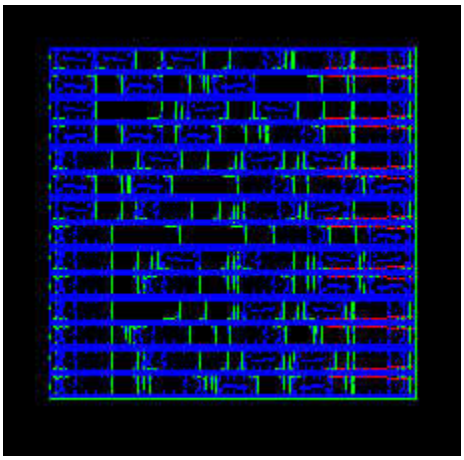


Figure 7-33 The core area after placing filler cells.

2. Place Periphery filler cells[37]
   Click on **File→Execute…**, and choose **fillperi.mac** on the **Execute** form, and then click on **Ok**. The messages appear as figure 7-34.

```
New cells with prefix fillperi and model name PERI_SPACER_01_P are added.
14:19:17 * SROUTE ADDCELL * CPU 0 0:00:01 Number of cells added: 132
Finished execution of file 'fillperi.mac'.
```

Figure 7-34 Messages of running fillperi.mac.

3. Save the design.

7.4.9 Viewing a Placed Database

7.4.9.1 Viewing Placed Cells

1. To look at the cell instances, select **cells visible** in the **Object Selection** area referring to figure 7-2, and then click on the **Redraw** icon.
   The outlines of placed macro cells appear in green.
2. Click and drag with the right mouse button to create a rectangle that encloses the cells which are to be viewed, and then zoom in by making a rectangle in the context window.
3. To look at the property list of one cell, select the cell and click on the **Properties** icon. Click on **Cancel** on the **Edit Properties** form to release the form.

7.4.9.2 Viewing Pins

1. Make **pins viewable** in the **Object Selection** area and click on the **Redraw** icon.
2. Select a pin, and then open the properties form by clicking on the **Properties** icon.
   Pins have a large number of properties.
   The NAME.CELL is the instance name of the placed cell that contains the pin.
   The NAME.NET is the name of the net the pin is connected to. The NAME.PIN is the name of the pin.

[37]User needs to check the foundry requirements if using other design kits.

3. Click on the background of the Artwork window to deselect the pin. The artwork window shows that no objects are selected.
4. Click on **Cancel** to close the Edit Properties form.
5. To go back to the big picture, click on the **Fit** icon or use **View→Recall Window** to go back to the previous artwork window setting.

7.4.9.3 Viewing Nets

1. To open the Display Options form by clicking on the **Display Options** icon, and make sure that the **Regular Nets** (under Routing) is **on**. If not, turn on **Regular Nets** and click on **Apply**.
2. Use the following steps to look at net properties:
   a. Make **nets selectable** and **visible**. Use the **Edit→Find** command to highlight a net.
   b. On the **Find** form, set **Type** to **Net**, and enter the name of a net.
   c. Click on **Select** on the Find form.
3. Click and drag the right mouse button to set the window size so it encloses the net.
4. Use the **Ctrl-q** (**Edit Properties**) to view the properties of the highlighted net.
5. Click on **Cancel** on the Edit Properties and Find forms.

7.4.10 Routing Power Nets

Load the design saved in section 7.4.8.

1. Click on **Route→Connect Ring…** to connect IO Pads, IO rings, and Follow Pins to the power rings. As there is no stripe and block in the design, set the Connect Ring form as figure 7-35 and then click on **Ok**.

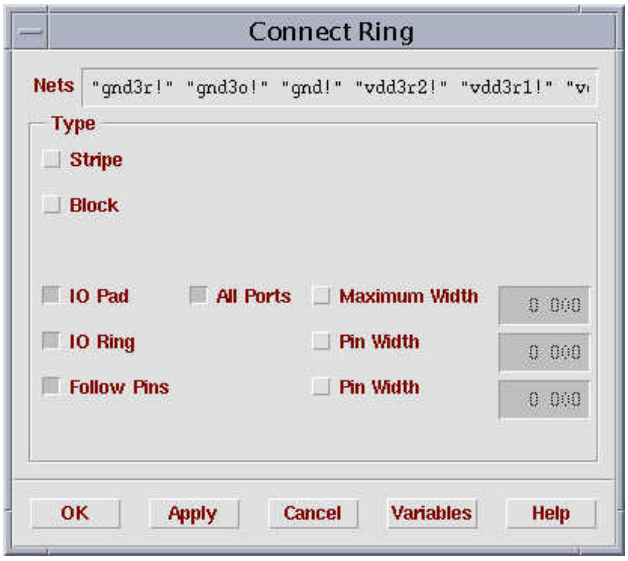


Figure 7-35 Connect Ring form settings.

Note: if your design has power stripe or block, these also need to be selected. All the selections under **Type** can be selected all in once and it can also be selected once a time. Click on **Help** button for more understanding.

2. The design after connecting rings is as figure 7-36. Notice that the power pads are not connected due to the foundry issue. They must be manually connected. Skip to step 6 if the power pads were connected.

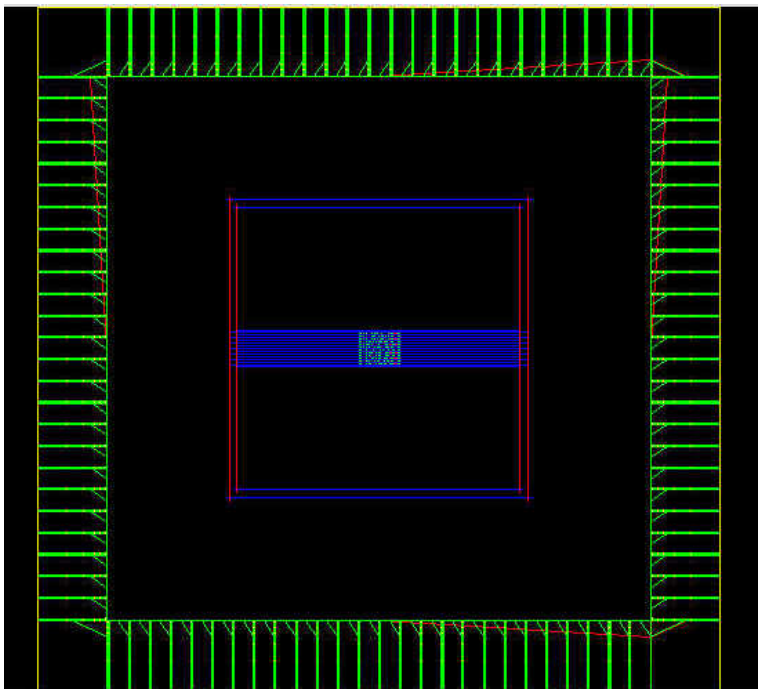


Figure 7-36 Design after connecting rings.

3. Click on **Edit→Wire→Add…**, and the **Add Wire** form appears. Click on the **Help** button for more information about the form. Set the form as figure 7-37, and then connect the vdd! pad to vdd! net manually.

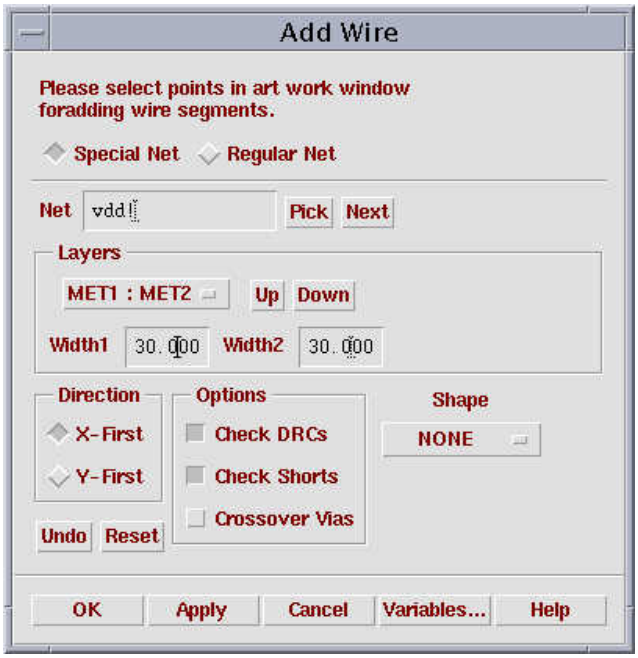


Figure 7-37 Add Wire form settings.

4. Repeat step 3 for gnd! pad and gnd! net connection. The design will look as figure 7-38 after connection.

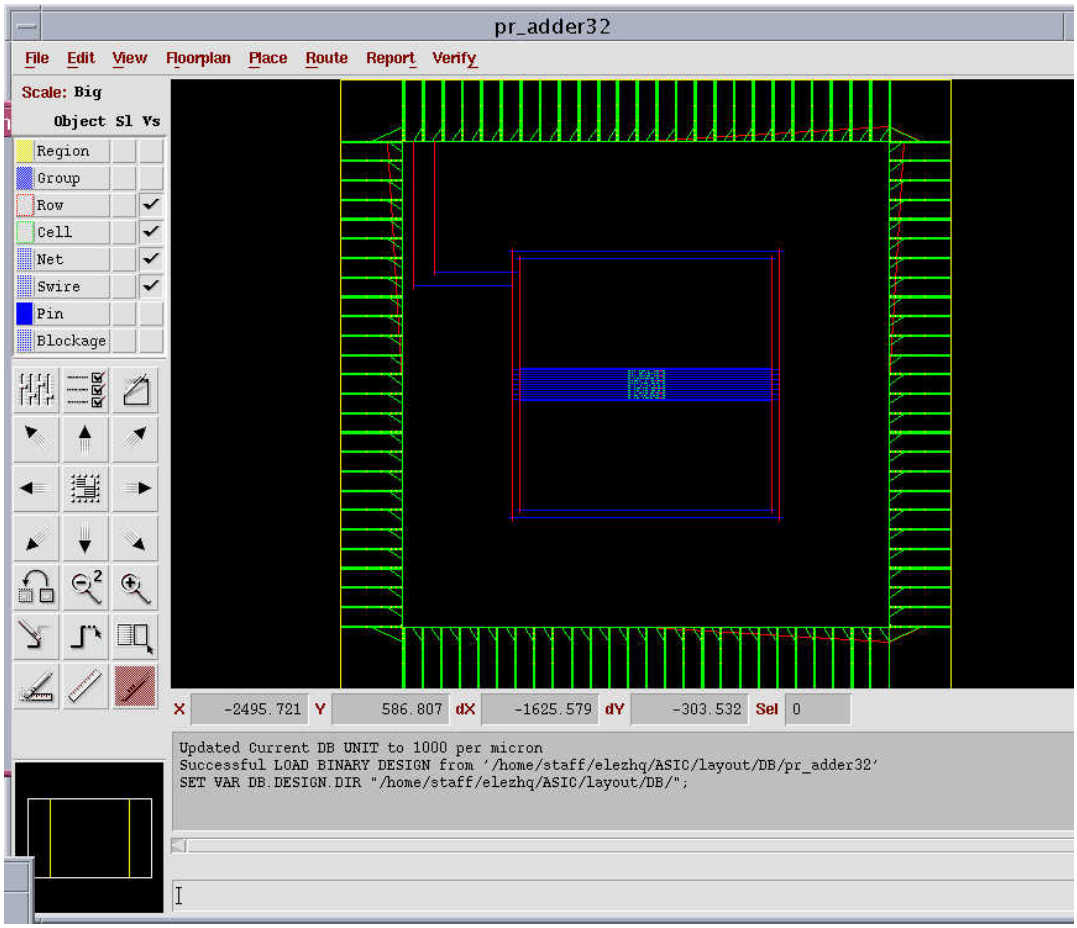


Figure 7-38 Design after connecting power pads manually.

5. Save the design as pr_adder32 by clicking on **File→ Save As…**.

7.4.11 Routing all the Nets

After power routing, global & final routing should be done. These two steps can be combined into one step with Wroute function of cadence SE software. Load the design pr_adder32.

1. Click on **Route→WRoute…**, and the WRoute form appears as figure 7-39.

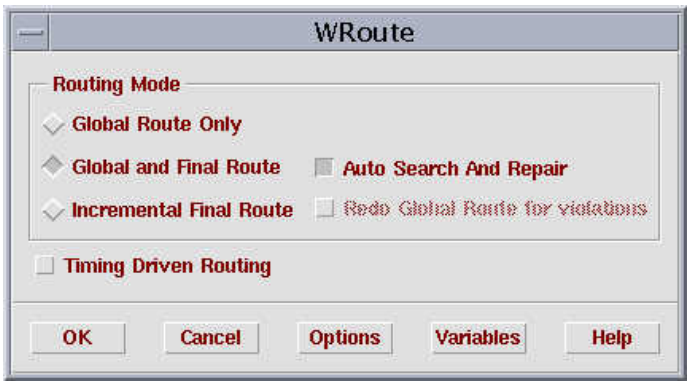


Figure 7-39 WRoute settings.

2. Click on **Ok** on the WRoute form. Click on **Help** for details of the form if necessary.
3. After a few mins, the artwork window looks as figure 7-40.

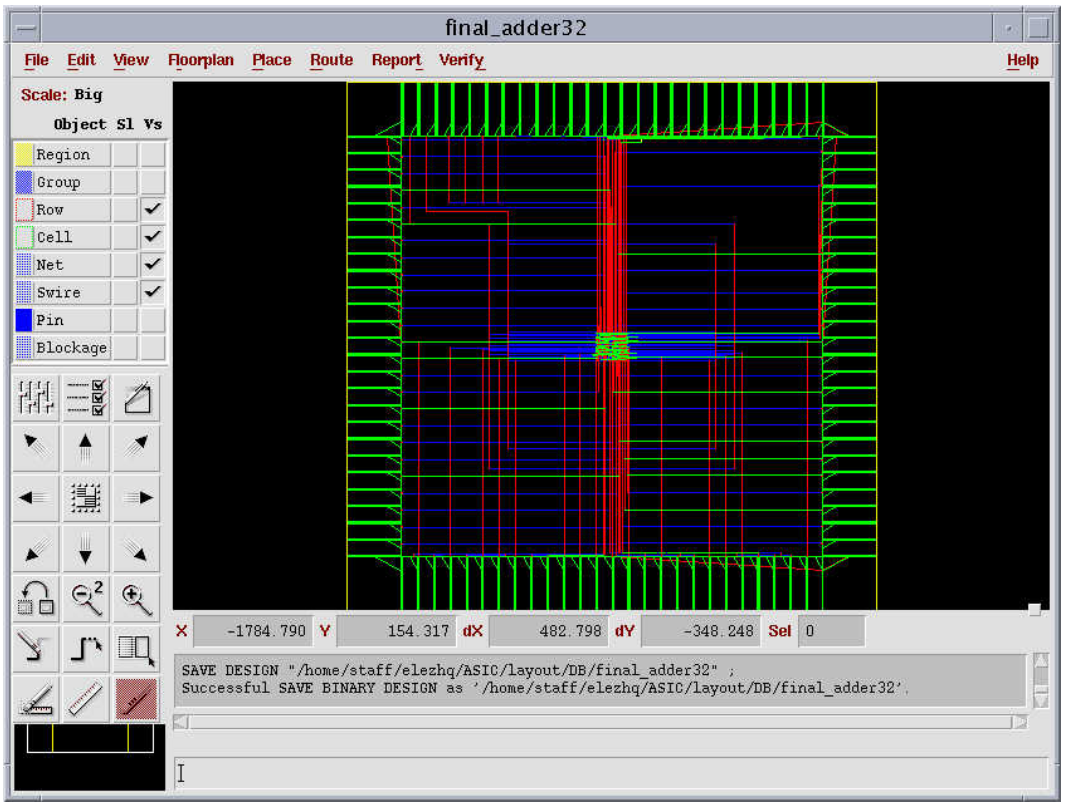


Figure 7-40 Design after WRoute.

4. Take a note of the message after routing. Manually solve any violation if there was. User can refer to section 7.4.12, to view the routings.
5. Save the design as final_adder32.

7.4.12 Viewing the Routed Design

1. Open the final routed design. On the **Display Option** form, turn on only **Rows** and **Cells&Blocks** (under Names), and then click on **Apply**.

2. The first routing step is global routing. Global routing is a plan for final or detail routing. The router divides the routing grid area of the array into GCells. This array of GCells is called the GCells grid. To see the GCell grid, click on **Global Routing** (under Grids) on the **Display Option** form and click on **Apply**. The grid appears in magenta. Zoom in to view the grid.
3. The SE **Initialize Floorplan** command creates the detail routing grid. The router uses the grid. To view it, click on **Detailed Routing** on the **Display Option** form and then click on **Apply**.
   The grid appears in yellow. Zoom in to view the grid. Each line is called a track. Notice that each GCell encloses a number of tracks.
4. After placement, the router considers the location of macro cell pins relative to the routing grid. To see pins, click on **Pins** (under **Objects**) on the **Display Options** form, and then click on **Apply**.
5. Cell layouts often include internal interconnect wiring that is not part of a pin. The router must know about these to avoid creating shorts or spacing violations. In the library, these are modeled as obstructions or blockages associated with a macro. To view the blockages of macro cells, click on **Routing Blockages** (under **Blockages**), and then click on **Apply**. The blockages appear in dark blue.
6. To see the wires created by special router, turn on **Special Nets** and **Special Net Wires** (under **Routing**). Metal 1 is dark blue and metal 2 is dark red.
7. To see the detail routing created by WRoute command, turn on **Regular Nets** and **Regular Net Wires** (under **Routing**), and then click on **Apply**. Metal 1 wires are dark blue, metal 2 wires are dark red, and metal 3 wires are dark green. Via openings between metal 2 and metal 3 are white.
8. To query a net, make **Nets** selectable, and use the **Properties** icon to look at the property of a net.
9. Use the **Fit** icon to fill the artwork window with the design.

7.4.13 Exporting Design

Follow the steps below to export/report design. Take a note of the message in the message area after click on **Ok** on each export/report form.

1. Writing parasitic RC, click on **Report→RC…** from the menu. The **Report RC** form appears. Set the form as figure 7-41 and click on **Ok**.

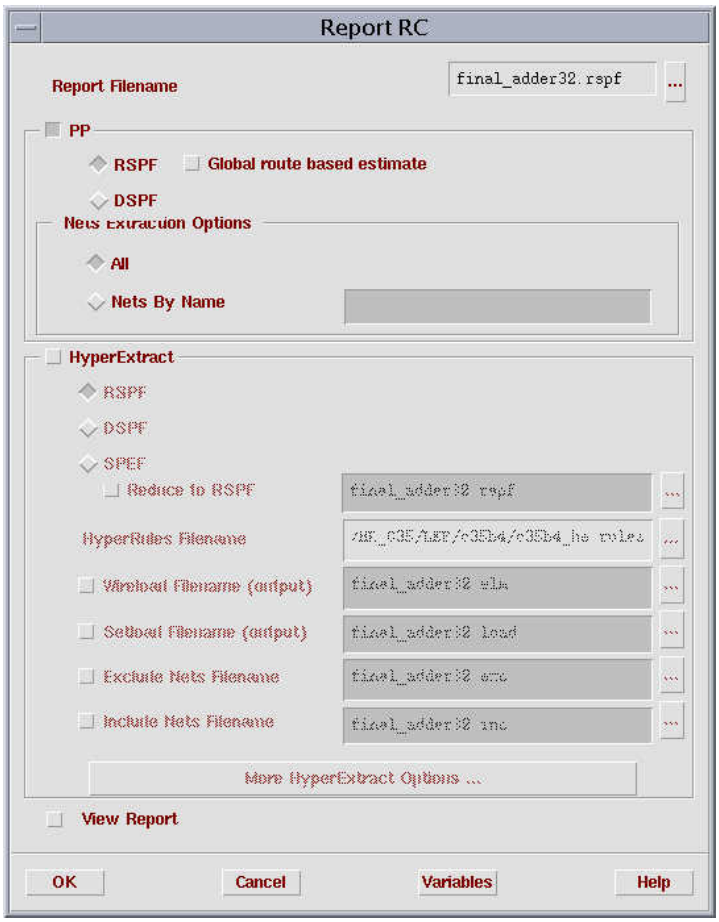


Figure 7-41 Report design RC.

2. Writing delay, click on **Report→Delay…** from the menu. The **Report Delay** form appears. Set the form as figure 7-42 and click on **Ok**.

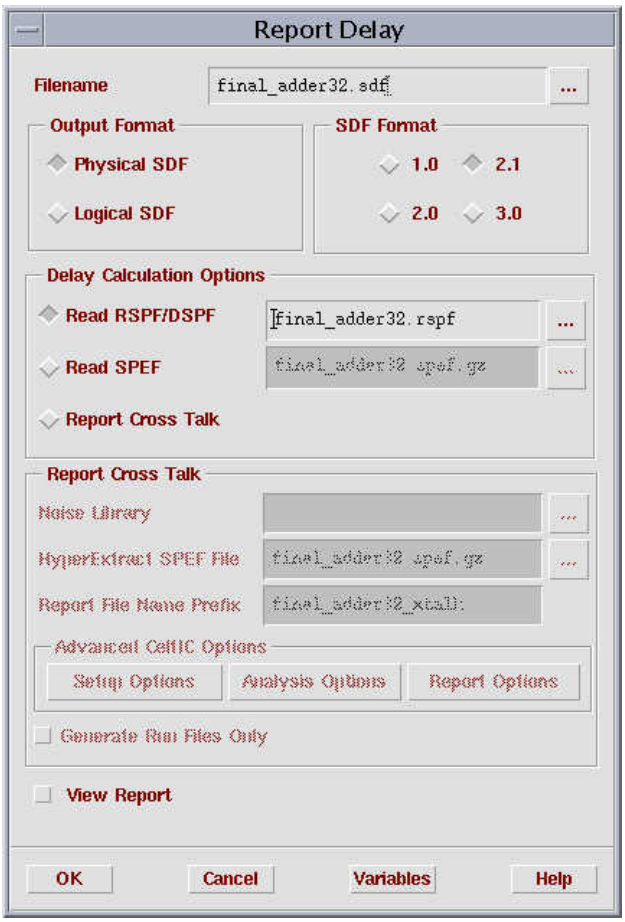


Figure 7-42 Report the delay of design.

3. Exporting design in Verilog, click on **File→Export→Verilog…** from the menu. The **Export Verilog** form appears. Set the form as figure 7-43 and click on **Ok**.

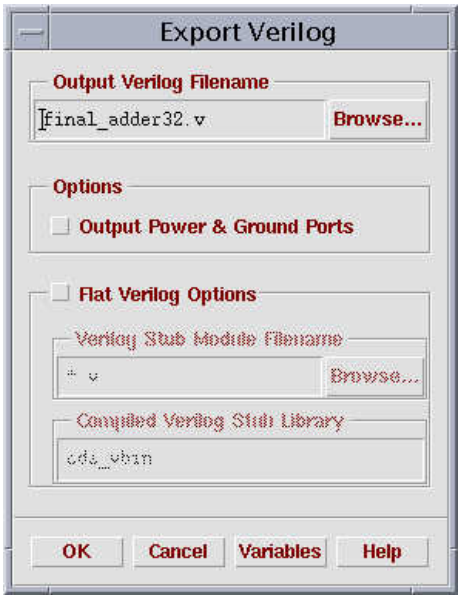


Figure 7-43 Export design in Verilog.

4. Exporting design in DEF, click on **File→Export→DEF…** from the menu. The **Export DEF** form appears. Set the form as figure 7-44 and click on **Ok**.

Figure 7-44 Export design in DEF.

5. Exporting design in GDSII, click on **File→Export→GDS II…** from the menu. The **Export GDSII** form appears. Set the form as figure 7-45 and click on **Ok**.

Figure 7-45 Export design in GDS II.

7.5 Conclusion

The digital layout flow with cadence SE is described. User may just follow the steps listed, for place and route. User can also use script to run the whole process at backend of the x-term window. For doing so, User may modify the file gemma.mac according to individual design and then start SE with the file.

User should notice that the technology used for the example is AMS 0.35um. Different design kits, the method to setup initial environment and design library would be different. User should change accordingly to vendor requirements.

The next step of ASIC design is to bring the output of layout to NCLaunch and/or primetime for post-layout verification and/or post-layout static timing analysis.

8. Post-Layout Verification with NCLaunch

The methods of post-layout and pre-layout verifications are same. The difference of simulation results between post-layout and pre-layout depends on the SDF files. The post-layout SDF file includes both delays of nets and cells but pre-layout SDF file includes only the delay of cells. As the verification methods are same, the SDF back annotation is not repeated. Only the file preparations are described in this chapter, for user knows what files should be used in the post-layout verification.

The example used is the adder32. The source and SDF files of the design are got from chapter 7, which are after place and route. The file preparations of post-layout verification are as follows:

1. Create a working directory: routed_ncvlog
   % mkdir routed_ncvlog
2. Copy the design source files: final_adder32.v, test_adder.v (referring to chapter 3 for this file) and SDF file: final_adder32.sdf to the directory: routed_ncvlog.
   % cd routed_ncvlog
   % cp /the path to design files/ final_adder32.v .
   % cp /the path to test_adder.v/test_adder.v .
   % cp /the path to SDF files/ final_adder32.sdf .
3. Create a lib directory which holds the library files used by the design.
   % mkdir lib
4. Copy the library files to the directory lib.
   % cp /path to library files/c35_CORELIB.v lib
   % cp /path to library files/c35_IOLIB_4M.v lib
   % cp /path to library files/udp.v lib
5. Modify the design source file - final_adder32.v to include the $sdf_annotate system task as figure 8-1.

```
module adder32
 ( a , b , cin , CLOCK , sum , cout  );

 input [31:0] a   ;
 input [31:0] b   ;
 input cin , CLOCK ;

 output [31:0] sum  ;
 output cout  ;
 wire [32:0] temp  ;

 wire n199  , n198  , n1  , n197  , n196  , n195  , n194  , n193  , n192  , n191  ;

 wire n190  , n189  , n188  , n187  , n186  , n185  , n184  , n183  , n182  , n181  ;

 wire n180  , n179  , n178  , n177  , n176  , n175  , n174  , n173  , n172  , n171  ;

 wire n170  , n169  , n168  , n167  , n166  , n132  , n131  , n130  , n129  , n128  ;

 wire n127  , n126  , n125  , n124  , n123  , n122  , n121  , n120  , n119  , n118  ;

 wire n117  , n116  , n115  , n114  , n113  , n112  , n111  , n110  , n109  , n108  ;

 wire n107  , n106  , n105  , n104  , n103  , n102  , n101  , n164  , n163  , n162  ;

 wire n161  , n160  , n159  , n158  , n157  , n156  , n155  , n154  , n153  , n152  ;

 wire n151  , n150  , n149  , n148  , n147  , n146  , n145  , n144  , n143  , n142  ;

 wire n141  , n140  , n139  , n138  , n137  , n136  , n135  , n134  , n133  , n165  ;

initial
begin
$sdf_annotate ("final_adder32.sdf.X", adder32, , "sdf.log", "MAXIMUM", "0.8:1.1:1.2", "FROM_MTM");
end

VDD3ALLP  PWR4 ( ) ;
```

Figure 8-1 $sdf_annotate system task in the post-layout source file.

6. Start NCLaunch in the working directory – routed_ncvlog as follows. The files of post-layout verification are shown as figure 8-2 in the NCLaunch window.
   %nclsunch –new&

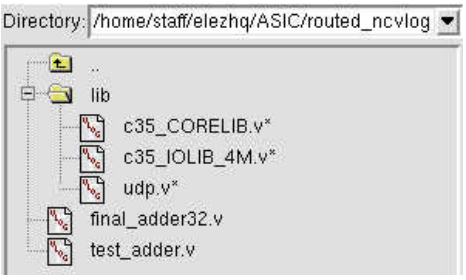


Figure 8-2 Start up of post-layout verification.

Next, users can follow the steps of section 5.4 to finish the post-layout verification.

9. Post-Layout STA with PrimeTime

Post-layout STA with primetime is described in this chapter. The data used for post-layout STA is got from cadence silicon ensemble, which is after place and route. The arrangement of the chapter is as follows. In section 9.1, the overview of post-layout STA is described. It focuses more on the physical data related topics which are not described in chapter 6 (pre-layout STA). The constraints of post-layout are given in section 9.2. A tutorial of post-layout STA with the example adder32 is demonstrated in section 9.3. Conclusion is given in section 9.4.

9.1 Overview of Post-Layout STA

Pre-layout STA with primetime is performed based on cell delays and wire load models, but post-layout STA with primetime is done with the physical data – delays and parasitics. These data and relative primetime concepts are briefly described in the following sub-sections.

9.1.1 Parasitic versus SDF

During post-layout STA, back-annotation is done with both SDF and parasitic data, but the two types of data serve different purposes.

1. SDF back-annotation is used to describe the cell and net delays. There is no cell or net delay calculation by PT – a “frozen” snapshot.
2. Parasitic back-annotation is used to describe net resistance and capacitance (RC). RCs are used to perform design rule analysis (for example, max_capacitance) and compute cell and net delays if no SDF is annotated.

It is best to always back-annotate both, for most accurate timing results. Any timing arcs missing SDF data will then use parasitic data.

9.1.2 Back-Annotation Command Summary

The following commands are used to read and back-annotate a design. Users may refer to the manpage of PT for the explanations of the commands.

- read_sdf
  - report_annotated_delay
  - report_annotated_check
  - remove_annotated_delay
  - remove_annotated_check
- read_parasitics
  - report_annotated_parasitics
  - remove_annotated_parasitics

9.1.3 List of Precedence

If SDF and parasitic data are both annotated, the order of the precedence is as follows.

- SDF.
- SPEF[38]/DSPF[39]/RSPF.
- Lumped RC.
- Wire load models.

[38] Standard Parasitic Exchange Format.
[39] Detailed Standard Parasitic Format.

9.2 Constraints of Post-Layout STA

The constraints of post-layout STA are different from that of pre-layout STA. As the design is on mask level, two Commands are no more used and they are *set_clock_uncertainty* and *set_wire_load_model*. Referring to chapter 4, *set_clock_uncertainty* models clock skew and *set_wire_load_model* specifies the wire load model used for nets. The STA on mask level, the clock skew and nets are calculated and modeled with the physical data respectively.

9.3 Tutorial of Post-Layout STA Using PrimeTime

The example used is the adder32, and its data is got from chapter 7. The method and flow of post-layout STA using primetime is shown below. The working directory of the tutorial is as same as that of doing pre-layout STA.

9.3.1 Preparations

1. Create a directory – routed to keep the design data.
   % mkdir routed
   % cp /path to design data/final_adder32.v routed
   % cp /path to design data/final_adder32.sdf routed
   % cp /path to design data/final_adder32.rspf routed
2. Modify the *.synopsys_pt_setup* file as follows

```
set_search_path "/path to the installation directory of foundry/synopsys/c35_3.3V/ \
/path to the project directory/mapped/db /path to the project directory/routed"
set link_path "* c35_CORELIB.db c35_IOLIB_4M.db"
```

3. Modify the constraint file as follows and save it as adder32_pt_routed.scr under the directory - scripts.

```
current_design adder32
link
reset_design

set_operating_conditions -analysis_type bc_wc -library c35_CORELIB
-max WORST -min BEST
set_load  0.1 [all_outputs]
set_driving_cell -library c35_CORELIB -lib_cell BUF8 [all_inputs]
remove_driving_cell [get_ports CLOCK]

#use the real clock tree generated
create_clock -per 100 -name clk [get_ports CLOCK]
set_propagated_clock [get_clocks clk]

#apply default timing constraints
set_input_delay -max 2 -clock clk [all_inputs]
set_input_delay -min 0.4 -clock clk [all_inputs]
remove_input_delay [get_ports CLOCK]
set_output_delay -max 0 -clock clk [all_outputs]
set_output_delay -min 0 -clock clk [all_outputs]
```

9.3.2 Start STA with PrimeTime

The followings are the steps to check timing with primetime.

1. Start primetime
   % primetime
2. Read design with the command - read_verilog.
   pt_shell>read_verilog ./routed/final_adder32.v
3. Execute scripts by click on **File→Execute Script…** and choose the file - adder32_pt_routed.scr under the directory - scripts.
4. Check timing and report analysis coverage before reading SDF and parasitics to ensure that there are no problem at this stage. The output is shown in figure 9-1.
   pt_shell>check_timing –verbose
   pt_shell>report_analysis_coverage

```
pt_shell> check_timing -verbose
Information: Using automatic max wire load selection group 'sub_micron'. (ENV-00
Information: Using automatic min wire load selection group 'sub_micron'. (ENV-00
Warning: Some timing arcs have been disabled for breaking timing loops
        or because of constant propagation. Use the 'report_disable_timing'
        command to get the list of these disabled timing arcs. (PTE-003)
Information: Checking 'no_clock'.
Information: Checking 'no_input_delay'.
Information: Checking 'partial_input_delay'.
Information: Checking 'no_driving_cell'.
Information: Checking 'unconstrained_endpoints'.
Information: Checking 'unexpandable_clocks'.
Information: Checking 'generic'.
Information: Checking 'latch_fanout'.
Information: Checking 'loops'.
Information: Checking 'generated_clocks'.
check_timing succeeded.
1
pt_shell> report_analysis_coverage
****************************************
Report : analysis_coverage
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Feb 16 14:54:52 2006
****************************************

Type of Check         Total              Met         Violated         Untested
--------------------------------------------------------------------------------
setup                    99          0 (  0%)        0 (  0%)        99 (100%)
hold                     99          0 (  0%)        0 (  0%)        99 (100%)
min_pulse_width          66         66 (100%)        0 (  0%)         0 (  0%)
out_setup                33         33 (100%)        0 (  0%)         0 (  0%)
out_hold                 33         33 (100%)        0 (  0%)         0 (  0%)
--------------------------------------------------------------------------------
All Checks              330        132 ( 40%)        0 (  0%)       198 ( 60%)

1
```

Figure 9-1 Messages of check_timing and report_analysis_coverage before reading physical data.

5. Read SDF file with the command - read_sdf.
   pt_shell>read_sdf ./routed/final_adder32.sdf
6. Read parasitics file with the command - read_parasitics
   pt_shell>read_parasitics ./routed/final_adder32.rspf
7. Check if all delays are annotated with the command - report_annotated_delay. The messages are shown in figure 9-2.
   pt_shell>report_annotated_delay

```
pt_shell> report_annotated_delay
****************************************
Report : annotated_delay
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Feb 16 15:14:39 2006
****************************************

                              |           |           |    NOT    |
    Delay type                |    Total  | Annotated | Annotated |
-----------------------------+-----------+-----------+-----------+
cell arcs                     |       132 |       132 |         0 |
cell arcs (unconnected)       |        33 |        33 |         0 |
internal net arcs             |        99 |        99 |         0 |
net arcs from primary inputs  |        66 |        66 |         0 |
net arcs to primary outputs   |        33 |        33 |         0 |
-----------------------------+-----------+-----------+-----------+
                              |       363 |       363 |         0 |

1
```

Figure 9-2 Messages of report annotated delay.

8. Check if all parasitics are annotated with the command - report_annotated_parasitics. Figure 9-3 shows the messages.
   pt_shell>report_anntoated_parasitics

```
pt_shell> report_annotated_parasitics
****************************************
Report : annotated_parasitics
        -internal_nets
        -boundary_nets
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Feb 16 16:13:10 2006
****************************************

                    |         |         |         |   RC    |   Not   |
Net Type            |  Total  | Lumped  |  RC pi  | network |Annotated|
--------------------+---------+---------+---------+---------+---------+
Internal nets       |     164 |       0 |     164 |       0 |       0 |
  - Driverless nets |         |       0 |       0 |       0 |       0 |
--------------------+---------+---------+---------+---------+---------+
Boundary/port nets  |      99 |       0 |      99 |       0 |       0 |
  - Driverless nets |         |       0 |       0 |       0 |       0 |
--------------------+---------+---------+---------+---------+---------+
                    |     263 |       0 |     263 |       0 |       0 |

1
```

Figure 9-3 Messages of report_annotated parasitics.

9. Check net RC with the command - report_net –verbose. Some of the messages are shown in figure 9-4.
   pt_shell>report_net -verbose

```
****************************************
Report : net
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Feb 16 15:22:07 2006
****************************************

Attributes:
    c - annotated capacitance
    r - annotated resistance

                                          Cap        Resistance
Net                  Fanout     Fanin   min/max       min/max     Pins   Attributes
-----------------------------------------------------------------------------------
CLOCK                     1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[0]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[1]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[2]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[3]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[4]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[5]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[6]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[7]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[8]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
a[9]                      1         1  4.76/4.76     0.00/0.00      2   c/c,r/r
```

Figure 9-4 Messages of report net RC.

10. Check timing and analysis coverage, referring to step 4. Figure 9-5 shows the message.

```
pt_shell> check_timing
Information: Checking 'no_clock'.
Information: Checking 'no_input_delay'.
Information: Checking 'partial_input_delay'.
Information: Checking 'no_driving_cell'.
Information: Checking 'unconstrained_endpoints'.
Information: Checking 'unexpandable_clocks'.
Information: Checking 'generic'.
Information: Checking 'latch_fanout'.
Information: Checking 'loops'.
Information: Checking 'generated_clocks'.
check_timing succeeded.
1
pt_shell> report_analysis_coverage
****************************************
Report : analysis_coverage
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Feb 16 15:27:33 2006
****************************************

Type of Check          Total              Met          Violated          Untested
-------------------------------------------------------------------------------------
setup                     99          0 (  0%)        0 (  0%)         99 (100%)
hold                      99          0 (  0%)        0 (  0%)         99 (100%)
min_pulse_width           66         66 (100%)        0 (  0%)          0 (  0%)
out_setup                 33         33 (100%)        0 (  0%)          0 (  0%)
out_hold                  33         33 (100%)        0 (  0%)          0 (  0%)
-------------------------------------------------------------------------------------
All Checks               330        132 ( 40%)        0 (  0%)        198 ( 60%)

1
```

Figure 9-5 Messages of check timing and report analysis coverage after reading physical data.

11. Generate Timing report with the command - report_timing. The timing report is shown in figure 9-6.
    pt_shell>report_timing

```
pt_shell> report_timing
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : adder32
Version: W-2004.12-SP2
Date   : Thu Feb 16 15:46:18 2006
****************************************


  Startpoint: a[0] (input port clocked by clk)
  Endpoint: sum_reg[31]
              (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Point                                                    Incr       Path
  -----------------------------------------------------------------------------
  clock clk (rise edge)                                    0.00       0.00
  clock network delay (propagated)                         2.00       2.00
  input external delay                                     2.00       4.00 f
  a[0] (in)                                                3.15 $     7.15 f
  U72/Y (ITUP)                                             0.84 *     7.98 f
  add_1_root_add_12_2/A[0] (adder32_DW01_add_33_0)         0.00 *     7.98 f
  add_1_root_add_12_2/U1_0/CO (ADD32)                      1.16 *     9.14 f
  add_1_root_add_12_2/U1_1/CO (ADD32)                      0.73 *     9.88 f
  add_1_root_add_12_2/U1_2/CO (ADD32)                      0.71 *    10.58 f
  add_1_root_add_12_2/U1_3/CO (ADD32)                      0.70 *    11.28 f
  add_1_root_add_12_2/U1_4/CO (ADD32)                      0.69 *    11.97 f
  add_1_root_add_12_2/U1_5/CO (ADD32)                      0.69 *    12.65 f
  add_1_root_add_12_2/U1_6/CO (ADD32)                      0.68 *    13.34 f
  add_1_root_add_12_2/U1_7/CO (ADD32)                      0.75 *    14.09 f
  add_1_root_add_12_2/U1_8/CO (ADD32)                      0.73 *    14.82 f
  add_1_root_add_12_2/U1_9/CO (ADD32)                      0.71 *    15.53 f
  add_1_root_add_12_2/U1_10/CO (ADD32)                     0.69 *    16.22 f
  add_1_root_add_12_2/U1_11/CO (ADD32)                     0.68 *    16.91 f
  add_1_root_add_12_2/U1_12/CO (ADD32)                     0.68 *    17.59 f
  add_1_root_add_12_2/U1_13/CO (ADD32)                     0.68 *    18.27 f
  add_1_root_add_12_2/U1_14/CO (ADD32)                     0.74 *    19.01 f
  add_1_root_add_12_2/U1_15/CO (ADD32)                     0.76 *    19.77 f
  add_1_root_add_12_2/U1_16/CO (ADD32)                     0.71 *    20.48 f
  add_1_root_add_12_2/U1_17/CO (ADD32)                     0.69 *    21.18 f
  add_1_root_add_12_2/U1_18/CO (ADD32)                     0.68 *    21.86 f
  add_1_root_add_12_2/U1_19/CO (ADD32)                     0.68 *    22.53 f
  add_1_root_add_12_2/U1_20/CO (ADD32)                     0.70 *    23.23 f
  add_1_root_add_12_2/U1_21/CO (ADD32)                     0.73 *    23.96 f
  add_1_root_add_12_2/U1_22/CO (ADD32)                     0.72 *    24.68 f
  add_1_root_add_12_2/U1_23/CO (ADD32)                     0.68 *    25.37 f
  add_1_root_add_12_2/U1_24/CO (ADD32)                     0.69 *    26.06 f
  add_1_root_add_12_2/U1_25/CO (ADD32)                     0.68 *    26.74 f
  add_1_root_add_12_2/U1_26/CO (ADD32)                     0.68 *    27.42 f
  add_1_root_add_12_2/U1_27/CO (ADD32)                     0.67 *    28.09 f
  add_1_root_add_12_2/U1_28/CO (ADD32)                     0.69 *    28.79 f
  add_1_root_add_12_2/U1_29/CO (ADD32)                     0.69 *    29.48 f
  add_1_root_add_12_2/U1_30/CO (ADD32)                     0.70 *    30.18 f
  add_1_root_add_12_2/U1_31/S (ADD32)                      0.94 *    31.12 r
  add_1_root_add_12_2/SUM[31] (adder32_DW01_add_33_0)      0.00 *    31.12 r
  sum_reg[31]/D (DFS3)                                     0.00 *    31.12 r
  data arrival time                                                  31.12

  clock clk (rise edge)                                  100.00     100.00
  clock network delay (propagated)                         1.50 *   101.50
  sum_reg[31]/C (DFS3)                                              101.50 r
  library setup time                                      -0.40 *   101.10
  data required time                                                101.10
  -----------------------------------------------------------------------------
  data required time                                                101.10
  data arrival time                                                 -31.12
  -----------------------------------------------------------------------------
  slack (MET)                                                        69.98


1
```

Figure 9-6 Post-layout STA timing report.

12. Check endpoint slack by clicking on **Timing→Histogram→Slack…**. Figure 9-7 is the endpoint slack.

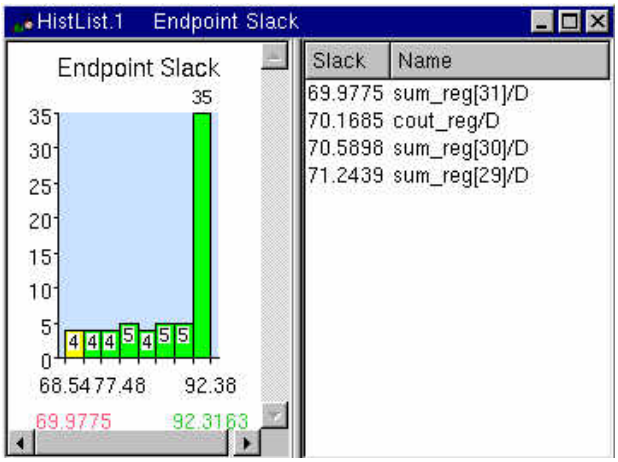


Figure 9-7 Histogram of endpoint slack.

13. Exit **primetime**. User may check others if necessary.

9.4 Conclusion

The post-layout STA using primetime is presented in this chapter. The difference between post-layout STA and pre-layout STA is focused. User may refer to chapter 6 – pre-layout STA and PT user guide while doing post-layout STA, for getting more information.

Reference

1. Advanced ASIC Chip Synthesis Using Synopsys Design Compiler and PrimeTime, Himanshu Bhatnagar, Conexant System, Inc., 1999.
2. Cadence NCLaunch User Guide, product version 5.1, September 2003.
3. Cadence NC-Verilog Simulation Help, September 2003.
4. Cadence NC-Verilog Simulator Tutorial with SimVision, April 2004.
5. Cadence NC Verilog Integration for Composer™ User Guide, October 2003.
6. Cadence Silicon Ensemble™ Place and Route Training Manual.
7. Cadence Silicon Ensemble™ Place and Route Lab Book.
8. Cadence SimVision User Guide.
9. Digital IC Design Lab Manual, Jiang Bin, May 2002.
10. Digital IC Design Lab Manual – Place & Route with I/O Pads, Xie Jiang, Dec. 2003.
11. Project Report – Design of an i80188 Microprocessor, Tan Chong Hau.
12. Synopsys Chip Synthesis, Synopsys Customer Education Services, 2003 Synopsys Inc.
13. Synopsys Chip Synthesis Workshop Lab Guide, Synopsys Customer Education Services.
14. Synopsys PrimeTime Introduction to Static Timing Analysis, Synopsys Customer Education Services, 2003 Synopsys Inc.
15. Synopsys PrimeTime Introduction to Static Timing Analysis Workshop Lab Guide, Synopsys Customer Education Services