Chapter 8

Verilog Synthesis

SYNTHESIS is the process of taking a behavioral Verilog file and converting it to a structural file using cells from a standard cell library. That is, the behavior that is captured by the Verilog program is *synthesized* into a circuit that behaves in the same way. The synthesized circuit is described as a collection of cells from the *target cell library*. This Verilog file is known as **structural** because it is strictly structural instantiations of cells. It is the Verilog text equivalent of a schematic. This structural file can be used as the starting point for the backend tools which will place those cells on the chip and route the wire connections between them.

There are three different behavioral synthesis tools that are usable in our flow, and one schematic-based helper application. They are:

Synopsys Design Compiler: This is the flagship synthesis tool from Synopsys. It comes in two flavors: **dc_shell** which has a TCL shell-command interface and **design_vision** which is a gui window/menu driven version. The **dc_shell** version is often driven by writing scripts and executing those scripts on new designs.

TCL is Tool Command Language and is a standard syntax for providing input commands to tools.

Synopsys Module Compiler: This is a specialty synthesis tool from Synopsys that is specifically for synthesis of arithmetic circuits. It has more knowledge of different arithmetic circuit variants than **design compiler**, including many fast adder variants and even floating point models. It also has its own circuit specification language that you can use to describe complex arithmetic operations. It uses the same target library database at **design compiler**.

Cadence Build Gates: This is the primary synthesis tool from Cadence. It uses the **.lib** file directly as the cell database and it is also usually driven from scripts.

Cadence to Synopsys Interface (CSI): This is a tool integrated with the Composer schematic capture tool that lets you take schematics that use a combination of standard cells and behavioral Verilog (wrapped in Composer symbols) and output that schematic as a structural Verilog file.

8.1 Synopsys dc_shell Synthesis

Add some intro stuff here

8.1.1 Basic Synthesis

In order to make use of Synopsys synthesis you need (at least) the following files:

.synopsys_dc.setup: This is the setup file for **design compiler** and **module compiler**. It can be copied from

/uusoc/facility/cad_common/local/class/6710/synopsys

Note that it has a dot as the first character in the file name. You should copy this file into the directory from which you plan to run the Synopsys tools.

a cell database file: This is the binary **.db** file that contains the cell information for the target library that you would like your behavior compiled to. Details of cell characterization and file formats for characterized libraries are in Chapter 7. It's helpful if you either have this file in the directory from which you are running Synopsys, or make a link to it in that directory if it really lives elsewhere.

A behavioral Verilog file: This is the file whose behavior you would like to synthesize into a circuit. It can contain purely behavioral code, or a mixture of behavioral and structural (cell instantiations) code. This should also be either in or linked to the directory you use for running **Synopsys**

If you have these three files you can run very basic synthesis using a class script. This script, named `beh2str`, converts a behavioral Verilog file into a structural Verilog file in a very simplified way. It does no fancy optimization, and it only works for a single behavioral Verilog file as input (no multiple-file designs). It's not really designed for final synthesis, it's just deigned as a quick and dirty script that you can use to check and see how things are working, and for initial investigations into small designs. To

```
/* Behavioral Model of an Inverter */
module INV_Test(INV_in,INV_out);
   input INV_in;
   output INV_out;
       assign INV_out = ~INV_in;
endmodule
```

Figure 8.1: Verilog behavioral description of an inverter

use the script you don't need to know anything about what **design compiler** is actually doing. For more advanced synthesis you'll want much more direct control. But, this is a good introduction to the process of synthesis, and a good way to see what the synthesis engine does to different ways of expressing things in Verilog.

Tiny Example: An Inverter

As an example, consider the extremely basic piece of Verilog code in Figurẽreffig:synth-inv which describes the behavior of a single inverter. I'll start by making a new directory in which to run the synthesis tools. I'll call it **$HOME/IC_CAD/synth** and I'll put the file from Figurẽreffig:synth-inv (named **inv-behv.v**) in that directory. I'll also put a copy (or a link) of a **.db** file of a target library in that directory. (I'll use **UofU_Digital_v1_1.db** but you can use a file which describes your own cell library). Finally I'll put a copy of the **.synopsys_dc.setup** file in this directory. If you make a directory called **WORK** in your **synth** directory, then some of the files that get generated in the synthesis process will go there instead of messing up your **synth** directory. I recommend doing this.

Once I have all these files in my **synth** directory I can connect to that directory and fire up the basic synthesis script. The usage information is shown in Figure 8.2. The command I'll use is:

```
beh2str inv-behv.v inv-struct.v UofU_Digital_v1_1.db
```

This will fire up **design compiler** with the right arguments and produce a file called **inv-struct.v** as output. This output file is seen in Figure 8.3. As you might hope, the tool has synthesized the behavior of the inverter into a single inverter cell from the target library.

Small Example: A Finite State Machine

As a (very) slightly larger example, consider the Verilog description of a simple four-state finite state machine in Figure 8.4. This FSM description

```
> behv2str

CORRECT>beh2str (y|n|e|a)? yes
beh2str - Synthesises a verilog RTL code to a structural code
          based on the synopsys technology library specified.
Usage   : beh2str f1 f2 f3
           f1 is the input verilog RTL file
           f2 is the output verilog structural file
           f3 is the compiled synopsys technology library file
```

Figure 8.2: Usage information for the **beh2str** script

```
module INV_Test ( INV_in, INV_out );
  input INV_in;
  output INV_out;

  invX1 U2 ( .A(INV_in), .Y(INV_out) );
endmodule
```

Figure 8.3: Synthesized inverter using a cell from the target library

uses parameters to define state encodings, and defines a state register in an **always** statement. The register will have an active-high clock and an active-low asynchronous clear. The state transition logic is defined with a case statement, and the output is defined with a continuous assignment.

If this state machine is synthesized using the **beh2str** script with the command

```
beh2str moore.v moore-struct.v UofU_Digital_v1_1.db
```

this results in the structural file shown in Figure 8.5. Note that internal wires have been defined by the synthesis procedure. Also note that for unknown reasons the synthesis procedure choose to use a **dff_qb** cell for state register bit 1 (**state_reg_1_**) even though it didn't use the **QB** output. I have no explanation for this, other than that the **beh2str** script is extremely basic and doesn't apply many optimizations.

8.1.2 Scripted Synthesis

If you look "under the hood" of the **beh2str** script you find that it is a wrapper that calls the much more general Synopsys **dc_shell** interface of **design compiler** with a very simple script. The script, shown in Figure 8.2, shows a very basic version of a general synthesis flow. All Synopsys scripts are

```
module moore (clk, clr, insig, outsig);
input clk, clr, insig;
output outsig;

// define state encodings as parameters
parameter [1:0] s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;

// define reg vars for state register and next_state logic
reg [1:0] state, next_state;

//define state register (with asynchronous active-low clear)
always @(posedge clk or negedge clr)
begin
        if (clr == 0) state = s0;
        else state = next_state;
end

// define combinational logic for next_state
always @(insig or state)
begin
        case (state)
                s0: if (insig) next_state = s1;
                    else next_state = s0;
                s1: if (insig) next_state = s2;
                    else next_state = s1;
                s2: if (insig) next_state = s3;
                    else next_state = s2;
                s3: if (insig) next_state = s1;
                    else next_state = s0;
        endcase
end

// now set the outsig. This could also be done in an always
// block... but in that case, outsig would have to be
// defined as a reg.
assign outsig = ((state == s1) || (state == s3));

endmodule
```

Figure 8.4: Simple State Machine

```
module moore ( clk, clr, insig, outsig );
  input clk, clr, insig;
  output outsig;
  wire   n6, n7, n8, n9;
  wire   [1:0] next_state;

  dff state_reg_0_ ( .D(next_state[0]), .G(clk), .CLR(clr), .Q(outsig) );
  dff_qb state_reg_1_ ( .D(next_state[1]), .G(clk), .CLR(clr), .Q(n6) );
  mux2_inv U7 ( .A(n7), .B(outsig), .S(n6), .Y(next_state[1]) );
  nand2 U8 ( .A(outsig), .B(insig), .Y(n7) );
  xor2 U9 ( .A(insig), .B(n8), .Y(next_state[0]) );
  nor2 U10 ( .A(n6), .B(n9), .Y(n8) );
  invX1 U11 ( .A(outsig), .Y(n9) );
endmodule
```

Figure 8.5: Result of running **beh2str** on **moore.v**

written in a scripting language called **Tool Command Language** or **TCL**. **TCL** is a standard language syntax for writing tool scripts that is used by most CAD tools, and certainly by most tools from Synopsys and **Cadence**. There is a basic **TCL** tutorial linked to the class web site, and eventually there will be a written version in an appendix to this text.

By looking at the **beh2str** script you can see the basic steps in a synthesis script. You can also see some basic **TCL** syntax for **dc_shell**. Variables are set using the **set** keyword. Lists are generated with a **list** command, or built by concatenating values with the **concat** command. **UNIX** environment variables are accessed using the **getenv** command. Other commands are specific to **dc_shell**. There are hundreds of variables that control different aspects of the synthesis process, and about as many commands. The **beh2str** script uses just about the bare minimum of these commands. The more advanced script shown later in this chapter uses a few more commands, but even that only scratches the surface of the opportunities for controlling the synthesis process.

The basic steps in **beh2str** are:

1. Inform the tools which cell libraries you would like to use. The **target_library** is the cell library, or a list of libraries, whose cells you would like to use in your final circuit. The **link_library** is a list of libraries that the synthesis tool can use during the linking phase of synthesis. This phase resolves design references by linking the design to library components. As such it includes libraries that contain more complex primitives than are in the **target_library**. The **synthetic_library** is set in the **.synopsys_dc.setup** file and points to a set of high level macros provided by Synopsys in their **DesignWare** package.

2. Read in the behavioral Verilog.

3. Set constraints and provide other information that will guide the synthesis. In general this section will have many more commands in a more advanced script. In this basic script the only constraint is to set a variable that forces the structural circuit to have buffer circuits for nets that might otherwise be implemented with a simple **assign** statement (i.e. nets with no logic that just pass through a module). Some downstream tools don't consider **assign** statements, even those with only a single variable on the right hand side, to be structural.

4. Compile the behavioral Verilog into structural Verilog using the constraints and conditions, and using the libraries specified earlier in the script. In this example hierarchy is flattened with the **ungroup_all** command.

```
# beh2str script
set target_library [list [getenv "LIBFILE"]]
set link_library [concat [concat "*" $target_library] $synthetic_library]

read_file -f verilog [getenv "INFILE"]

#/* This command will fix the problem of having             */
#/* assign statements left in your structural file.          */
set_fix_multiple_port_nets -all -buffer_constants

compile -ungroup_all
check_design

#/* always do change_names before write...                   */
redirect change_names { change_names -rules verilog -hierarchy -verbose }

write -f verilog -output [getenv "OUTFILE"]
quit
```

Figure 8.6: **beh2str.tcl** basic synthesis command script

5. Apply a **dc_shell** rewriting rule that makes sure that the output is correct Verilog syntax

6. Write the synthesized result to a structural Verilog file.

These steps are the basic synthesis flow that is used for this simple synthesis script, and for a more complex script. Of course, you could start up the **dc_shell** command line interface and type each of these commands to the shell one at a time, but it's almost always easier to put the commands in a script and execute from that script. The command to start the **dc_shell** tool with the command line interface is

`syn-dc`

All arguments that you give to that command will be passed through to the **dc_shell** program. So, if you wrote a script of your own synthesis commands you could execute that with the following command.

`syn-dc -f <scriptname>`

You can use `syn-dc -help` for a usage message, and if you start the tool with `syn-dc` you cay type `help` at the shell prompt for a long list of all available commands. This shell accepts commands in **TCL** syntax, and each command generally has a help message of its own. So, for example, typing `write_file -help` to **dc_shell** will give detailed documentation on the **write_file** command.

The basic sequence of a generic Synopsys synthesis flow, along with commands to consider at each step, is as follows (this is a more elaborate version of what you saw in Figure 8.6). Note that you don't have to write

these scripts from scratch. There is a class example of a relatively full-featured synthesis script that you can use.

Write behavioral Verilog: First, of course, you need to develop your design as a set of Verilog files. The description can use a combination of behavioral and structural Verilog, but remember that you can only use the "synthesis subset" of Verilog, and any structural references must come from your eventual **target library** or from a Synopsys-provided library such as the **DesignWare** cells (more about them later).

Specify Paths and Libraries: These are paths and libraries that Synopsys uses.

- **search_path:** This is the search path that **design compiler** uses to search for libraries and other files and it is set in the **.synopsys_dc.setup** file. You can override that in your script by setting the **search_path** variable there. At a minimum you probably want this path to contain **.** (your current directory) and the **/libraries/syn** directory in the **Synopsys** installation directory for the generic libraries. You should also include any other directories that hold database files of interest to your synthesis.
- **target_library:** This is the library, or list of libraries, that you want **design compiler** to target as the result of the synthesis. It's your cell library in **.db** format.
- **synthetic_library:** This is the list of libraries that contain information about pre-defined structures. The most common example is the **DesignWare** libraries from Synopsys that contain information about a host of datapath structures that **design compiler** can use. Include at least **dw_foundation.sldb** on this list.
- **link_library:** A list of libraries that you want your design linked to. This is typically a list of ***** (meaning your own module descriptions in your Verilog code), your **target_library** list and your **synthetic_library** list.
- **symbol_library:** This is a list of libraries that have graphical symbol information for showing the gates of the design in the graphical design tool **design vision**. This is typically just the built-in Synopsys generic library **generic.sdb**.

Read in the Verilog code: If you have a single Verilog file you can read it with the one-step **read_file** command, but in general (and required if you have more than one Verilog file in your input) you should use the sequence of **analyze** and **elaborate**.

analyze -format verilog -lib WORK <files> This command parses all the input files and puts the semi-digested versions into a directory called **WORK**.

elaborate <top-module-name> -lib WORK -update This command compiles the input files into an intermediate technology independent internal form. As part of this step all the inferred memory devices are discovered and reported.

Set the Operating Environment: In this phase you tell **design compiler** which operating conditions, wire load models, etc to use from the **.db** file, and also define the input and output drive expected to and from the module. Typically the operating conditions (worst, typ, best) are set in the **.db** file so you don't have to change it, as is the wire load model. If your **.lib** (and thus your **.db**) file has multiple operating conditions defined in it, here's you chance to pick one.

Other commands to consider at this point are: **set_drive**, **set_driving_cell**, **set_load**, **set_load_cell**, and **set_fanout_load**. You can get details of the syntax of these commands from the **design compiler** shell. There are also examples in the class generic script. If you don't set the driving cell or the drive **design compiler** will assume there is infinite drive available on the inputs. This may or may not be what you want to assume. I typically set the input driving cell to be a 4X inverter, and the output load to be driving the input of a 4x inverter. The D and Q signals from a DFF are another good choice.

Set the Design Constraints: This is where you tell **design compiler** how fast you want the synthesized circuit to run, how big it should be, and other constraints. This is a critical section. It's where you set speed and area goals for synthesis and it determines how hard **design compiler** tries to optimize things. Commands include: **create_clock**, **set_clock_latency**, **set_propagated_clock**, **set_clock_uncertainty**, **set_clock_transition**, **set_input_delay**, **set_output_delay**, and **set_max_area**.

The most important of these commands are:

create_clock This is the command to use to set your speed goal for the synthesis. If you have a clock signal in your design, use that signal as the clock signal (which should be the obvious thing to do). The period you set is the speed goal that **design compiler** tries to hit. Think about this carefully! Too aggressive a speed goal will cause **design compiler** to spend a *long* time trying to meet the goal and then failing. Too conservative a goal will be easily met, but with a very conservative design. You can also set a **virtual** clock if your design is combinational. This is a name you use for a fake clock that is used just to set the speed goal for synthesis. See the example of this command in the class script.

set_max_area This sets the area goal. Speed is always the primary goal for synthesis. But, after speed is achieved, **design compiler** will try to optimize for a smaller circuit. This is one situation where it's common to set the max area goal to 0 to force **design compiler** to try to make as small a design as possible.

Compile (synthesize/optimize) your design: In this step you call the **compile** command to synthesize your circuit, subject to your constraints. The newest version of **design compiler** has a *mega command* called **compile_ultra** which runs through a Synopsys-approved set of compilation procedures. You should probably use this one unless you have a good reason not to. The other choice is the plain **compile** command which has lots of switches you can read about in **design compiler** documentation.

Analyze and report results: The **check_design** command will check the result to make sure nothing funny has happened. You can also report the area, timing, power, and other results as analyzed by **design_compiler**. The commands you might use here are:

write -format verilog This command will write out the synthesized structural Verilog. Before you issue this command you should always issue the **change_names -rules verilog** command which will make sure that correct Verilog syntax is used. Why this isn't an automatic part of the **write** command I don't know.

write_rep This generates a synthesis report that describes (among other things) the critical path of the design and whether the synthesis has achieved the speed target.

write_ddc This writes a Synopsys formatted binary database file that you can read in to either **design compiler** or **design vision** for further processing.

write_sdf This writes a *standard delay format* file that you can use to back-annotate your simulations with extracted timings from the synthesized circuit.

write_sdc This writes a constraints file that is used to pass the constraints that you set in your synthesis script on to other tools like the place and route tool. It's expecially important for the clock tree synthesis phase of the place and route tool.

write_pow This writes a report file that describes the power your design will dissipate as best as **design compiler** can tell.

A much more advanced script that demonstrates this flow is available in the **/uusoc/facility/cad_common/local/class/6710/synopsys** directory. It's called **syn-script.tcl**. This is a much more general script for synthesis that

you can use for many of your final synthesis tasks, and as a starting point if you'd like to use other features. The script is shown in three separate Figures, but should be kept in a single file for execution. The first part of the script, shown in Figure 8.7, is where you set values specific to your synthesis task. Note that this script assumes that the following variables are set in your **.synopsys_dc.setup** file (which they will be if you make sure that you've linked the class version to your synthesis directory):

SynopsysInstall: The path to the main synopsys installation directory

synthetic_library: The path to the **DesignWare** files

symbol_library: The path to a library of generic logic symbols for making schematics

In this first part of the **syn-script.tcl** file you need to modify things for your specific synthesis task. You should look at each line carefully, and in particular you should change everything that has "!!" in front and back to the correct values for your synthesis task. Some comments about the things to set follow:

- You need to set the name of your **.db** file as the **target_library**, or make this a list if you have multiple libraries with cell descriptions.
- You also need to list all of your Verilog behavioral files. The examples have all been with a single Verilog file, but in general a larger design will most likely use multiple files.
- The **basename** is the basename that will be used for the output files. An extra descriptor will be appended to each output file to identify them.
- **myclk** is the name of your clock signal. If your design has no clock (i.e. it's combinational not sequential) you can use a **virtual clock** for purposes of defining a speed target. Synopsys uses the timing of the clock signal to define a speed goal for the synthesis. A **virtual clock** is a name not attached to any wire in your circuit that can be used for this speed goal if you don't actually have a clock.
- The **useUltra** switch defines whether to use "ultra mode" for compiling or not. Unless you have very specific reasons to drive the synthesis directly, "ultra mode" will probably give you the best results.
- The timing section is where you set speed goals for the synthesis. The numbers are in ns. A period of **10** would set a speed goal of 100MHz, for example.

```
#/* search path should include directories with memory .db files */
#/* as well as the standard cells                                */
set search_path [list . \
[format "%s%s"  SynopsysInstall /libraries/syn] \
[format "%s%s"  SynopsysInstall /dw/sim_ver] \
!!your-library-path-goes-here!!]

#/* target library list should include all target .db files     */
set target_library [list !!your-library-name!!.db]

#/* synthetic_library is set in .synopsys_dc.setup to be        */
#/* the dw_foundation library.                                 */
set link_library [concat  [concat  "*" $target_library] $synthetic_library]

#/* below are parameters that you will want to set for each design */

#/* list of all HDL files in the design                          */
set myfiles [list !!all-your-files!!]
set fileFormat verilog    ;# verilog or VHDL
set basename !!basename!! ;# choose a basename for the output files
set myclk !!clk!!         ;# The name of your clock
set virtual 0             ;# 1 if virtual clock, 0 if real clock

#/* compiler switches...                                         */
set useUltra 1          ;# 1 for compile_ultra, 0 for compile
                         #mapEffort, useUngroup are for
                         #non-ultra compile...
set mapEffort1 medium   ;# First pass - low, medium, or high
set mapEffort2 medium   ;# second pass - low, medium, or high
set useUngroup 1        ;# 0 if no flatten, 1 if flatten

#/* Timing and loading information                               */
set myperiod_ns !!10!!   ;# desired clock period (sets speed goal)
set myindelay_ns !!0.5!! ;# delay from clock to inputs valid
set myoutdelay_ns !!0.5!!;# delay from clock to output valid
set myinputbuf !!invX4!! ;# name of cell driving the inputs
set myloadcell !!UofU_Digital/invX4/A!! ;# name of pin that outputs drive
set mylibrary !!UofU_Digital!! ;# name of library the cell comes from

#/* Control the writing of result files                          */
set runname   struct    ;# Name appended to output files

#/* the following control which output files you want. They     */
#/* should be set to 1 if you want the file, 0 if not           */
set write_v 1           ;# compiled structural Verilog file
set write_db 0          ;# compiled file in db format (obsolete)
set write_ddc 0         ;# compiled file in ddc format (XG-mode)
set write_sdf 0         ;# sdf file for back-annotated timing sim
set write_sdc 0         ;# sdc constraint file for place and route
set write_rep 1         ;# report file from compilation
set write_pow 0         ;# report file for power estimate
```

Figure 8.7: Part 1 of the **syn-script.tcl** synthesis script

- The other delays define how inputs from other circuits and outputs to other circuits will behave. That is, how will this synthesized circuit connect to other circuits.

- The **myinputbuf** variable should be set to the name of the cell in your library that is an example of what would be driving the external inputs to your circuit, and the **myloadcell** variable should be the name of the pin on a cell in your library that represents the output load of an external output. The example in Figure 8.7 references the **A** input of the **invX4** inverter in the **UofU_Digital** library (also defined as the **target_library**. These must be names of cells and cell inputs in one of your target libraries.

- The **write** flags define which outputs should be generated by the synthesis process. You almost certainly want to generate a structural Verilog file. You will usually also want at least a report file for timing and area reports. The other output files are used for other phases of the total flow. The **ddc** file is the Synopsys binary database format. You can save the synthesized circuit in **.ddc** format for ease of reading it back in to Synopsys for further processing. The **.sdf** and **.sdc** files are timing and constraint files that are used later in the flow. The power report uses power information in the **.db** file to generate a very rough estimate of power usage by your design.

.

The second part of the **syn-script.tcl** is seen in Figure 8.8. It contains the synthesis commands that use the variables you set in the first part of the file. You shouldn't have to modify anything in this part of the file unless you'd like to change how the synthesis proceeds. Note that the **read** command from **beh2str** has been replaced with a two-step process of **analyze** and **elaborate**. This is because if you have multiple Verilog files to synthesize, you need to analyze them all first before combining them together and synthesizing them. The other commands are documented in the script. You can see that constraints are set according to your information in part1. Finally the design is compiled, checked, and violations are checked.

The third part of the **syn-script.tcl** file is where the outputs are written out. It's pretty straightforward. Note that when you write the structural Verilog output you also **change_names** to make sure that the output is in correct Verilog syntax. You'd think that this would be part of the **write** command, but it's not. The output file names are constructed from **basename** and **runname** which are set in the first part of the file.

```
# analyze and elaborate the files
analyze -format $fileFormat -lib WORK $myfiles
elaborate $basename -lib WORK -update
current_design $basename

# The link command makes sure that all the required design
# parts are linked together.
# The uniquify command makes unique copies of replicated modules.
link
uniquify

# now you can create clocks for the design
if {  $virtual == 0 } {
    create_clock -period $myperiod_ns $myclk
} else {
    create_clock -period $myperiod_ns -name $myclk
}

# Set the driving cell for all inputs except the clock
# The clock has infinte drive by default. This is usually
# what you want for synthesis because you will use other
# tools (like SOC Encounter) to build the clock tree
# (or define it by hand).
set_driving_cell  -library $mylibrary -lib_cell $myinputbuf  \
    [remove_from_collection [all_inputs] $myclk]

# set the input and output delay relative to myclk
set_input_delay $myindelay_ns -clock $myclk \
    [remove_from_collection [all_inputs] $myclk]
set_output_delay $myoutdelay_ns -clock $myclk [all_outputs]

# set the load of the circuit outputs in terms of the load
# of the next cell that they will drive, also try to fix
# hold time issues
set_load [load_of $myloadcell] [all_outputs]
set_fix_hold $myclk

# This command will fix the problem of having
# assign statements left in your structural file.
# But, it will insert pairs of inverters for feedthroughs!
set_fix_multiple_port_nets -all -buffer_constants

# now compile the design with given mapping effort
# and do a second compile with incremental mapping
# or use the compile_ultra meta-command
if {  $useUltra == 1 } {
    compile_ultra
} else {
    if {  $useUngroup == 1 } {
        compile -ungoup_all -map_effort $mapEffort1
        compile -incremental_mapping -map_effort $mapEffort2
    } else {
        compile -map_effort $mapEffort1
        compile -incremental_mapping -map_effort $mapEffort2
    }
}

# Check things for errors
check_design
report_constraint -all_violators
```

Figure 8.8: Part 2 of the **syn-script.tcl** synthesis script

```
set filebase [format "%s%s" [format "%s%s" $basename "_"] $runname]

# structural (synthesized) file as verilog
if {  $write_v == 1 } {
    set filename [format "%s%s" $filebase ".v"]
    redirect change_names \
        { change_names -rules verilog -hierarchy -verbose }
    write -format verilog -hierarchy -output $filename
}

# write out the sdf file for back-annotated verilog sim
# This file can be large!
if {  $write_sdf == 1 } {
    set filename [format "%s%s" $filebase ".sdf"]
    write_sdf -version 1.0 $filename
}

# this is the timing constraints file generated from the
# conditions above - used in the place and route program
if {  $write_sdc == 1 } {
    set filename [format "%s%s" $filebase ".sdc"]
    write_sdc $filename
}

# synopsys database format in case you want to read this
# synthesized result back in to synopsys later (Obsolete db format)
if {  $write_db == 1 } {
    set filename [format "%s%s" $filebase ".db"]
    write -format db -hierarchy -o $filename
}

# synopsys database format in case you want to read this
# synthesized result back in to synopsys later in XG mode (ddc format)
if {  $write_ddc == 1 } {
    set filename [format "%s%s" $filebase ".ddc"]
    write -format ddc -hierarchy -o $filename
}

# report on the results from synthesis
# note that > makes a new file and >> appends to a file
if {  $write_rep == 1 } {
    set filename [format "%s%s" $filebase ".rep"]
    redirect $filename { report_timing }
    redirect -append $filename { report_area }
}

# report the power estimate from synthesis.
if {  $write_pow == 1 } {
    set filename [format "%s%s" $filebase ".pow"]
    redirect $filename { report_power }
}

quit
```

Figure 8.9: Part 3 of the **syn-script.tcl** synthesis script

```
module moore ( clk, clr, insig, outsig );
  input clk, clr, insig;
  output outsig;
  wire   n2, n3, n4;
  wire   [1:1] state;
  wire   [1:0] next_state;

  dff state_reg_0_ ( .D(next_state[0]), .G(clk), .CLR(clr), .Q(outsig) );
  dff state_reg_1_ ( .D(next_state[1]), .G(clk), .CLR(clr), .Q(state[1]) );
  mux2_inv U3 ( .A(n2), .B(outsig), .S(state[1]), .Y(next_state[1]) );
  nand2 U4 ( .A(outsig), .B(insig), .Y(n2) );
  xor2 U5 ( .A(insig), .B(n3), .Y(next_state[0]) );
  nor2 U6 ( .A(state[1]), .B(n4), .Y(n3) );
  invX4 U7 ( .A(outsig), .Y(n4) );
endmodule
```

Figure 8.10: Result of running **syn-dc** with the **syn-script.tcl** on **moore.v**

Small Example: A Finite State Machine

As an example, if the **moore.v** Verilog file in Figure 8.4 was compiled with this script, I would set the **target_library** to **UofU_Digital.db**, include **moore.v** in the **myfiles** list, use a **basename** of **moore**, a **myclk** of **clk**, and use the ultra mode with **useUltra**. With a speed goal of **10**ns period and the input buffer and output load set as in the example, the result is seen in Figure 8.10. Note that it's almost the same as the simple result in Figure 8.5 (which would expect for such a simple Verilog behavioral description), but there are differences. This version of the synthesis uses the same **dff** cells for both state variables, and the inverter producing the **outsig** output has been sized up to an **invX4**.

During the synthesis procedure a lot of output is produced by **dc_shell**. You should not ignore this output! You really need to look at it to make sure that the synthesis procedure isn't complaining about something in your behavioral Verilog, or in your libraries!

One place that you really need to pay attention is in the **Inferred memory devices** section. This is in the elaboration phase of the synthesis where all the memory (register and flip flop) devices are inferred from the behavioral code. In the case of this simple finite state machine the inferred memory is described as seen in Figure 8.11. You can see that the synthesis process inferred a **flip-flop** memory (as opposed to a gate latch) with a width of **2**. The other features with **Y/N** switches define the features of the memory. In this case the **flip-flop** has an **AR** switch which means Asynchronous Reset. The other possibilities are Asynchronous Set, Synchronous Reset, Synchronous Set, and Synchronous Toggle. One reason it's critical to pay attention to the inferred memories is that it is easy to write Verilog code that will result in an inferred memory when you meant the construct to be combinational.

```
Inferred memory devices in process
        in routine moore line 14 in file
                './moore.v'.
===============================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|      state_reg      | Flip-flop |   2   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
```

Figure 8.11: Inferred memory from the **moore.v** example

```
link

  Linking design 'moore'
  Using the following designs and libraries:
  --------------------------------------------------------------------------
  moore                       /home/elb/IC_CAD/syn-f06/moore.db
  UofU_Digital_v1_1 (library) /home/elb/IC_CAD/syn-f06/UofU_Digital.db
  dw_foundation.sldb (library) /uusoc/.../SYN-F06/libraries/syn/dw_foundation.sldb
```

Figure 8.12: Link information from the **dc_shell** synthesis process (with **dw_foundation** path shortened)

Another section of the compilation log that you might want to pay attention to is the **link** information. This tells you which designs and libraries your design has been linked against and lets you make sure that you're using the libraries you want to be using. The **link** information for the **moore.v** example is shown in Figure 8.12. The **dw_foundation** link library is the Synopsys **DesignWare** library that was defined in the **.synopsys_dc.setup** file.

The final timing report and area report are both contained in the **moore_struct.rep** file and are shown in Figures 8.13 and 8.14. The timing report (shown in Figure 8.13) tells you how well **dc_shell** did in compiling your design for speed. It lists the worst-case timing path in the circuit in terms of the delay at each step in the path. Then it compares that with the speed target you set in the script to see if the resulting design is fast enough. All the timing information is from your cell library in the **.db** file. In the case of the **moore.v** example you can see that the worst case path takes **2.30ns** in this synthesized circuit. The required time set in the script is **10ns**, minus the **0.2ns** setup time defined in the library for the flip flops. So, in order to meet timing, the synthesized circuit must have a worst case critical path of less than **9.98ns**. The actual worst case path according to **dc_shell** is **2.30ns** so the timing is **met** with **7.69ns** of slack.

If the timing was not met, the slack would tell you by how much you need to improve the speed of the worst case path to meet the timing goal. If I reset the speed goal to a **1ns** period with **0.1ns** input and output delays

to try to make a very fast (1GHz) circuit rerun the synthesis, the critical path timing will be reduced to **1.90ns** because **dc_shell** works harder in the optimization phase, but that doesn't meet the required arrival time of **0.98ns** so the slack is violated by **-0.98ns**. That is, the circuit will not run at the target speed.

The area report is shown in Figure 8.14. It tells us that the design has four ports: three inputs (insig, clk, and clr), and one output (outsig). There are seven cells used, and 10 total nets. It also estimates the area of the final circuit using the area estimates in the **.db** file. This area does not take placement and routing into account so it's just approximate.

8.1.3 Design Vision GUI

A "tall thin" designer is someone who knows something about the entire flow from top to bottom. This is as oppose to a "short fat" designer who knows everything about one layer of the design flow, but not much about the other layers.

Running **dc_shell** with a script (like **syn-script.tcl**) is the most common way to use the synthesis tool in industry. What usually happens is that a CAD support group designs scripts for particular designs and the Verilog design group will develop the behavioral model of the design and then just run the scripts that are developed by the CAD team. But, there are times when you as a "tall thin" designer want to run the tool interactively and graphically and see what's happening at each step. For this, you would use the graphical GUI interface to **design compiler** called **design_vision**.

To start **design compiler** with the **design_vision** GUI use the `syn-dv` script. This will read your **.synopsys_dc.setup** file and then open a GUI interface where you can perform the synthesis steps. Each step in the **syn-script.tcl** can be performed separately either by typing the command into the command line interface of **design_vision** or by using the menus. The main **design_vision** window looks like that in Figure 8.15.

Small Example: A Finite State Machine

Using the same small Moore-style finite state machine as before (shown in Figure 8.4) I can read this design into **design vision** using the **File → Analyze** and **File → Elaborate** menu commands. After the elaboration step the behavioral Verilog has been elaborated into an initial circuit. This circuit, shown in Figure 8.16, is mapped to a generic set of internally defined gates that are not related to any library in particular. It is this step where memory devices are inferred from the behavioral Verilog code. You can see the inferred memory information in the **Log** window.

Now you can set constraints on the design using the menu choices or by typing the constraint-setting commands into the shell. One of the most important is the definition of the clock which sets the speed goal for synthesis.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : moore
Version: Y-2006.06
Date   : Mon Sep 25 15:52:13 2006
****************************************

Operating Conditions: typical   Library: UofU_Digital_v1_1
Wire Load Model Mode: enclosed

  Startpoint: state_reg_0_
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: state_reg_0_
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  moore              5k                    UofU_Digital_v1_1

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.00       0.00
  state_reg_0_/G (dff)                     0.00       0.00 r
  state_reg_0_/Q (dff)                     1.31       1.31 f
  U7/Y (invX2)                             0.28       1.59 r
  U6/Y (nor2)                              0.30       1.89 f
  U5/Y (xor2)                              0.40       2.29 r
  state_reg_0_/D (dff)                     0.00       2.30 r
  data arrival time                                   2.30

  clock clk (rise edge)                   10.00      10.00
  clock network delay (ideal)              0.00      10.00
  state_reg_0_/G (dff)                     0.00      10.00 r
  library setup time                      -0.02       9.98
  data required time                                  9.98
  -----------------------------------------------------------
  data required time                                  9.98
  data arrival time                                  -2.30
  -----------------------------------------------------------
  slack (MET)                                         7.69
```

Figure 8.13: Timing report for the **moore.v** synthesis using **syn-script.tcl**

```
****************************************
Report : area
Design : moore
Version: Y-2006.06
Date   : Mon Sep 25 15:24:12 2006
****************************************

Library(s) Used:

    UofU_Digital_v1_1 (File: /home/elb/IC_CAD/syn-f06/UofU_Digital.db)

Number of ports:                4
Number of nets:                10
Number of cells:                7
Number of references:           6

Combinational area:       1231.199951
Noncombinational area:    1944.000000
Net Interconnect area:    8470.000000

Total cell area:          3175.199951
Total area:               11645.200195
```

Figure 8.14: Area report for the **moore.v** synthesis using **syn-script.tcl**

You can set the clock by selecting the **clk** signal in the schematic view, and then selecting **Attributes** → **Specify Clock** from the menu. An example of a clock definition with a period of **10ns** and a symmetric waveform with the rising edge of the clock at **5ns** and the falling edge at **10ns** is shown in Figure 8.17.

After your desired constraints are set, you can compile the design with the **Design** → **Compile Ultra** menu. After the design is compiled and mapped to your target library the schematic is updated to reflect the new synthesized and mapped circuit as seen in Figure 8.18. This file can now be written using the **Edit** → **Save As** menu. If you choose a filename with a **.v** extension you will get the structural Verilog view. If you choose a file name with a **.ddc** extension you will get a Synopsys database file. You can also write out report files with the **Design** → **Report ...** menus.

Perhaps the most interesting thing you can do with **design vision** is use the graphical display to highlight critical paths in your design. You can use the **Timing** → **Report Timing Paths** menu command to generate a timing report for the worst case path (if you leave the paths blank in the dialog box), or for a specific path in your circuit (if you fill them in). You can also obtain timing slack information for all path endpoints in the design using endpoint slack histograms. These histograms show a distribution of the timing slack values for all endpoints in the design giving an overall picture of how the design is meeting timing requirements. Use the **Timing** → **Endpoint Slack** command to generate this window. You can choose how many histogram bins to use. The **endpoint slack histogram** for the **moore** example isn't all

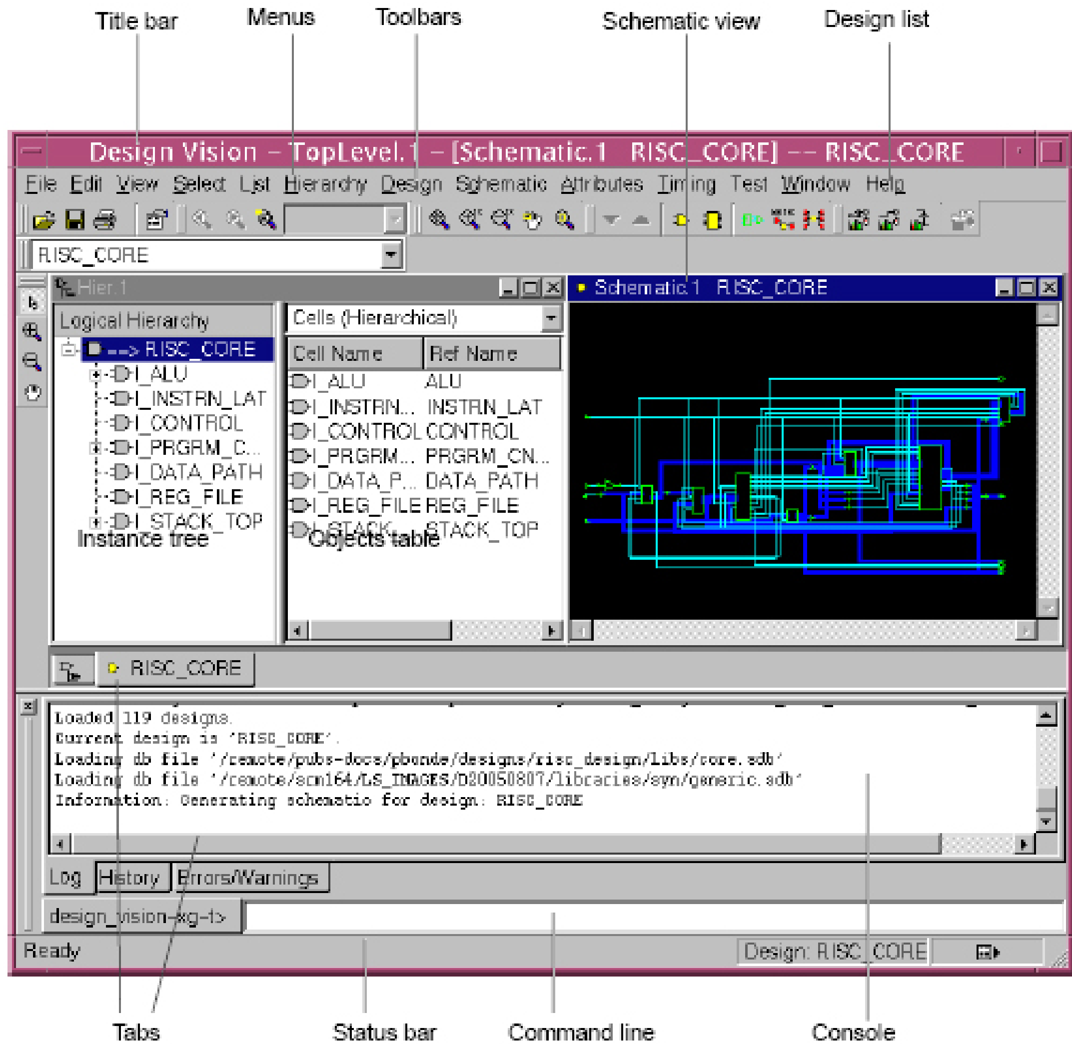

Figure 8.15: General view of the **Design Vision** GUI

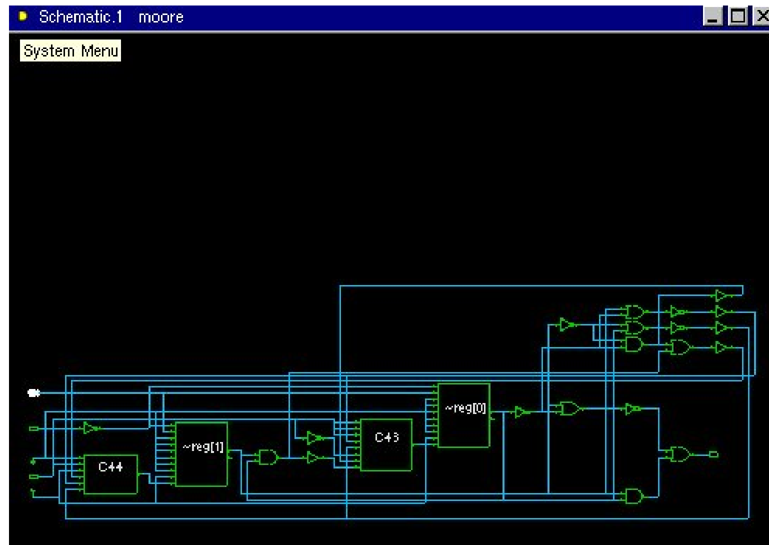

Figure 8.16: Initial mapping of the **moore.v** example

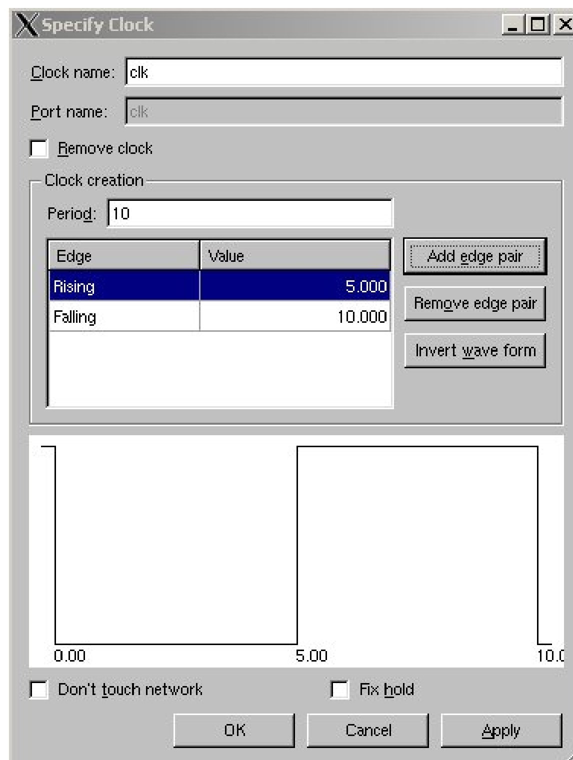

Figure 8.17: Clock definition in **Design Vision**

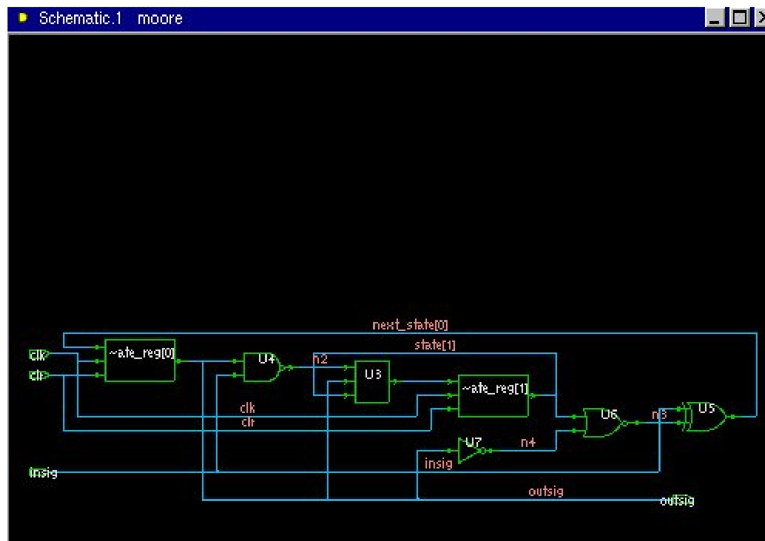

Figure 8.18: Final mapping of the **moore.v** example

that interesting since there are only two paths of interest in the circuit (one ending at each state bit). The result is shown in Figure 8.19. You can see which path corresponds to each bar in the histogram by clicking on it. You can also look at net capacitance, and general path slack (i.e. not ending at points in the circuit. See Figures 8.20 and 8.21 for examples. Clicking on a path in a slack window will highlight that path in the schematic as seen in the Figures. You can also use the **Highlight** menu to highlight all sorts of things about the circuit including the critical path. It's a pretty slick interface. You should play around with it to discover things it can do (especially on more complex circuits than the **moore** machine).

Note that you can also use **design vision** to explore a circuit graphically that has been compiled with a script. Make sure that your script writes a Synopsys database file (a **.ddc** file), and then you can fire up **design vision** with `syn-dv` and read in the compiled file as a **.ddc** file. You can then explore the timing and other features of the compiled circuit with **design vision**.

8.1.4 DesignWare Building Blocks

Although you can write Verilog code to describe almost any behavior you want to describe, there are some behaviors and structures that are so common you might ask "Isn't there a pre-designed version out there somewhere so I don't have to do it from scratch?" In fact, there are. The Synopsys **DesignWare** package is a large set of library building blocks that are pre-

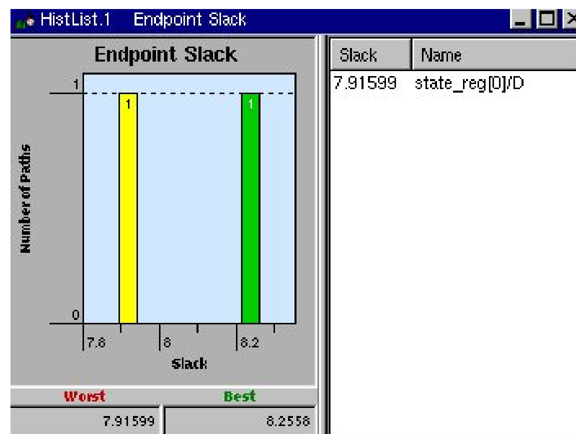

Figure 8.19: Endpoint slack for the two endpoints in **moore**

designed by Synopsys for use with **design compiler**. They are generally quite parameterizable so you can tailor them to your particular application. They are instantiated either by structural instantiation of a particular component from the **DesignWare** library in your Verilog code, or by writing your code in such a way that **design compiler** can easily figure out that it should us a **DesignWare** component for that circuit.

The full set of **DesignWare** building block IP components grows with each release of the **design compiler** tool. The current set includes arbiters, datapath components (adders, subbtractors, shifters, incrementers, decrementers, multipliers, dividers, etc.), floating point arithmetic operations, parity and CRC generators, FIR filters, clock-domain crossing circuits, encoders, decoders, counters, FIFOs, and even flip-flop based RAMs. The full set of components can be seen on the Synopsys web site at **http://synopsys.com/dw/buildingblock.php**. On this site you'll find datasheets for each component that describe how to use the component in Verilog code. For example, if you want a decementer in your design, you can use the code shown in Figure 8.22. The **width** parameter controls how wide the synthesized decrementer is, and the reference to **DW01_dec** pulls that model from the **DesignWare** library. Now when you use **decrementer** in other Verilog modules, it will be implemented with the **DesignWare** version. Check the Synopsys **DesignWare** web site for a full list of modules and datasheets that describe how to use them. If I synthesize the 8-bit decrementer with

I modified the class syn-script.tcl with the information needed to synthesize the decrementer.

```
syn-dc -f dec-script.tcl
```

(using the **Uofu_Digital_v1_1** library as a target), I get the structural code shown in Figure 8.23.

Even if you don't directly instantiate a **DesignWare** model, if **design**

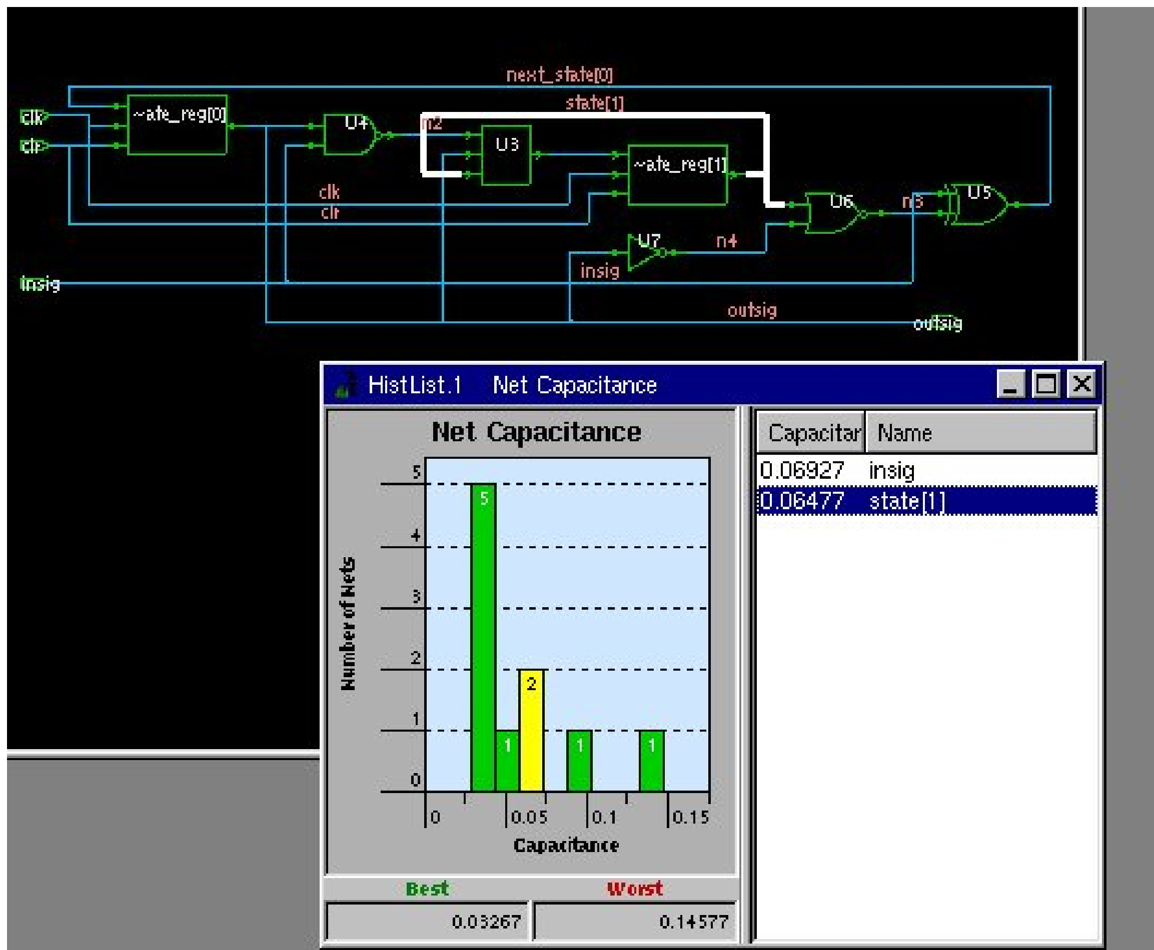

Figure 8.20: Wiring capacitance histogram with highlighted path

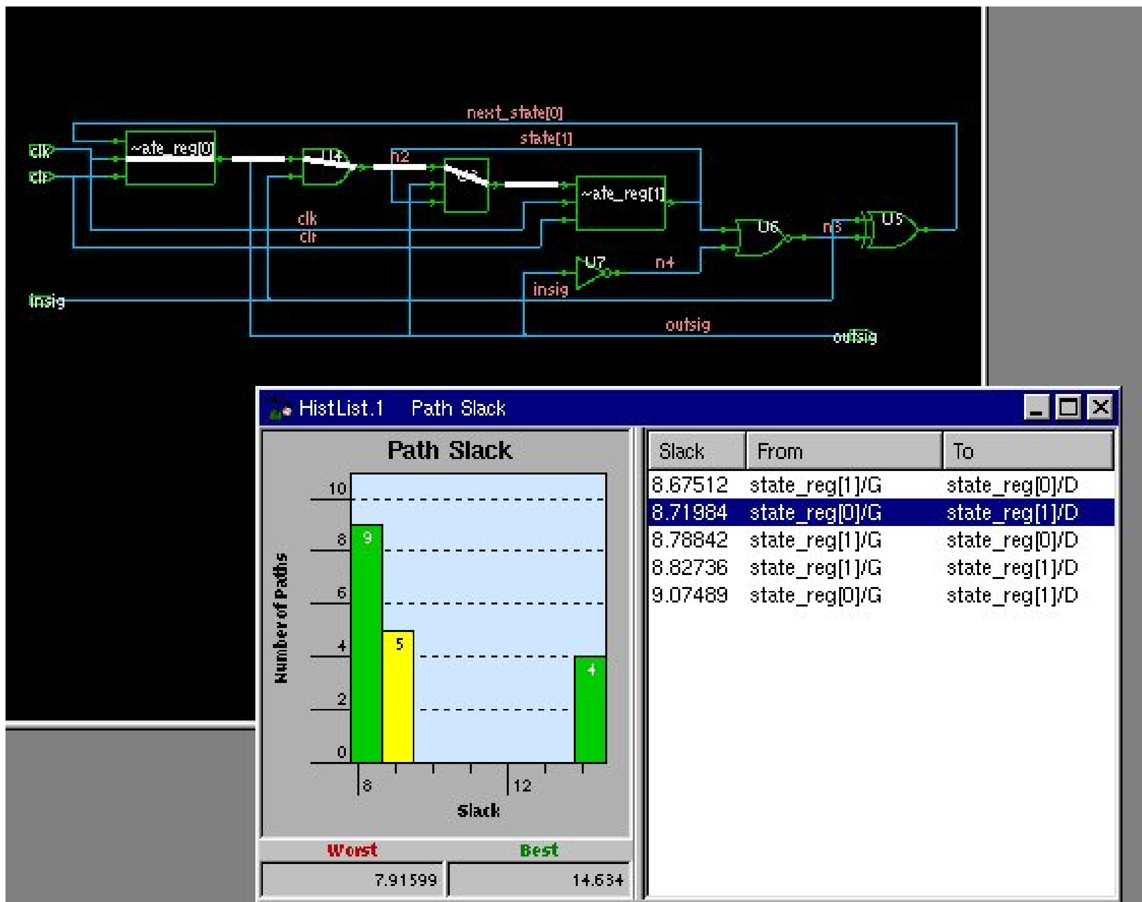

Figure 8.21: Timing path histogram with highlighted path

```
module decrementer( inst_A, SUM_inst );

  parameter width = 8;

  input [width-1 : 0] inst_A;
  output [width-1 : 0] SUM_inst;

  // Instance of DW01_dec
  DW01_dec #(width)
    U1 ( .A(inst_A), .SUM(SUM_inst) );

endmodule
```

Figure 8.22: **DesignWare** 9 bit decrementer instantiated in Verilog code

```
module decrementer ( inst_A, SUM_inst );
  input [7:0] inst_A;
  output [7:0] SUM_inst;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13;

  nand2 U1 ( .A(n1), .B(SUM_inst[0]), .Y(n3) );
  OAI U2 ( .A(SUM_inst[0]), .B(n1), .C(n3), .Y(SUM_inst[1]) );
  nor2 U3 ( .A(inst_A[2]), .B(n3), .Y(n4) );
  AOI U4 ( .A(n3), .B(inst_A[2]), .C(n4), .Y(n2) );
  nand2 U5 ( .A(n4), .B(n5), .Y(n6) );
  OAI U6 ( .A(n4), .B(n5), .C(n13), .Y(SUM_inst[3]) );
  xnor2 U7 ( .A(inst_A[4]), .B(n13), .Y(SUM_inst[4]) );
  nor2 U8 ( .A(inst_A[4]), .B(n6), .Y(n10) );
  nor2 U9 ( .A(inst_A[4]), .B(inst_A[3]), .Y(n7) );
  nand3 U10 ( .A(n7), .B(n4), .C(n9), .Y(n8) );
  OAI U11 ( .A(n10), .B(n9), .C(n8), .Y(SUM_inst[5]) );
  xnor2 U12 ( .A(inst_A[6]), .B(n8), .Y(SUM_inst[6]) );
  nand2 U13 ( .A(n10), .B(n9), .Y(n11) );
  nor2 U14 ( .A(inst_A[6]), .B(n11), .Y(n12) );
  xor2 U15 ( .A(inst_A[7]), .B(n12), .Y(SUM_inst[7]) );
  bufX4 U16 ( .A(n6), .Y(n13) );
  invX1 U17 ( .A(inst_A[1]), .Y(n1) );
  invX1 U18 ( .A(inst_A[0]), .Y(SUM_inst[0]) );
  invX1 U19 ( .A(inst_A[3]), .Y(n5) );
  invX1 U20 ( .A(inst_A[5]), .Y(n9) );
  invX1 U21 ( .A(n2), .Y(SUM_inst[2]) );
endmodule
```

Figure 8.23: Structural code for the 8-bit **DesignWare** decrementer

compiler runs across a piece of behavioral Verilog code that it thinks it can best implement with a **DesignWare** module, that's what it will use. There's a good chance you would get a **DesignWare** circuit if you had an assignment of the form

assign sum = in - 1;

in your code (or the equivalent inside of an **always** block). That's what is happening if you get sub-modules in your synthesized circuit with names that myeteriously start with **DW**. Of course, you have to have **dw_foundation.sldb** in your **synthetic_library** list for this to happen. The flip side of this, of course, is that if you don't want **design compiler** to have these libraries available, you should remove that synthetic database file from that list.

8.1.5 Module Compiler

Module compiler is a separate tool specifically for synthesizing arithmetic circuits. It uses the same cell library database as **dc_shell** but has more information about building efficient arithmetic structures, including floating point units. More details are coming!

There's some evidence that the **Module Compiler** tool is no longer rel-

evant. It looks like the **DesignWare** macros that used to be restricted to **Module Compiler** are now available for general **design compiler**. I'll document **module compiler** procedures here, but you may want to see if you can do what you need to do directly in **design compiler** with the same (or even better) results.

8.2 Cadence BuildGates Synthesis

8.2.1 Basic Synthesis

This is Cadence's version of the generic synthesis tool. Some people report better results for this tool than for dc_shell. I'm sure it depends on your circuit and your familiarity with the tools. More details are coming!

8.2.2 Scripted Synthesis

More advanced scripted synthesis

8.3 Importing Structural Verilog into Cadence

Once you have generated structural Verilog from Synopsys or from **Cadence**, one thing you might want to do is import that structural Verilog into Cadence **Composer** as a schematic and as a symbol so that you can use it in other schematics. This is known as *importing* Verilog into **Composer**. To import Verilog first decide which Cadence library you want to import the structural Verilog into. you may want to make a new library (make sure to attach it to the **UofU AMI C5N** technology library). Once you know which library you want to import the circuit into, Use the **CIW** menu **File** → **Import** → **Verilog...**. I'll use the decrementer from Figure 8.23 for this example. Fill in the fields:

Target Library Name: The library you want to read the Verilog description into. In this case I'll use a new library that I created named **decrementer**.

Reference Libraries: These are the libraries that have the cells from the cell libraries in them. In this case they will be **example** and **basic**. You will use your own library in place of **example**.

Verilog Files to Import: The structural Verilog from your synthesis process. In this case it's **dec_structr.v** from my use of Synopsys **design compiler**.

-v Options: This is the Verilog file that has Verilog descriptions of all the library cells. In this case I'm using **example.v**. You'll use the file from your own library.

The dialog box looks like that in Figure 8.24. You can click on **OK** to generate a new schematic view based on the structural Verilog. Strangely this will result in some warnings in the **CIW** related to bin files deep inside the Cadence **IC 5.1.41** directory, but it doesn't seem to cause problems. You now have a schematic (Figure 8.25) and symbol (Figure 8.26) of the decrementer. The log file of the Verilog import process should show that all the cell instances are taken from the cell library (**example** in this case).

8.4 Cadence to Synopsys (CSI) Schematic/Netlist Interface

Generating structural netlists from schematics - Although this is billed as an interface between Cadence and Synopsys, it is really a way to generate a structural netlist from a schematic. If you have a schematic with standard cells gates, this will generate a netlist that only goes down to those gates, and not descend all the way to the transistors as would happen for simulation. More details are coming!

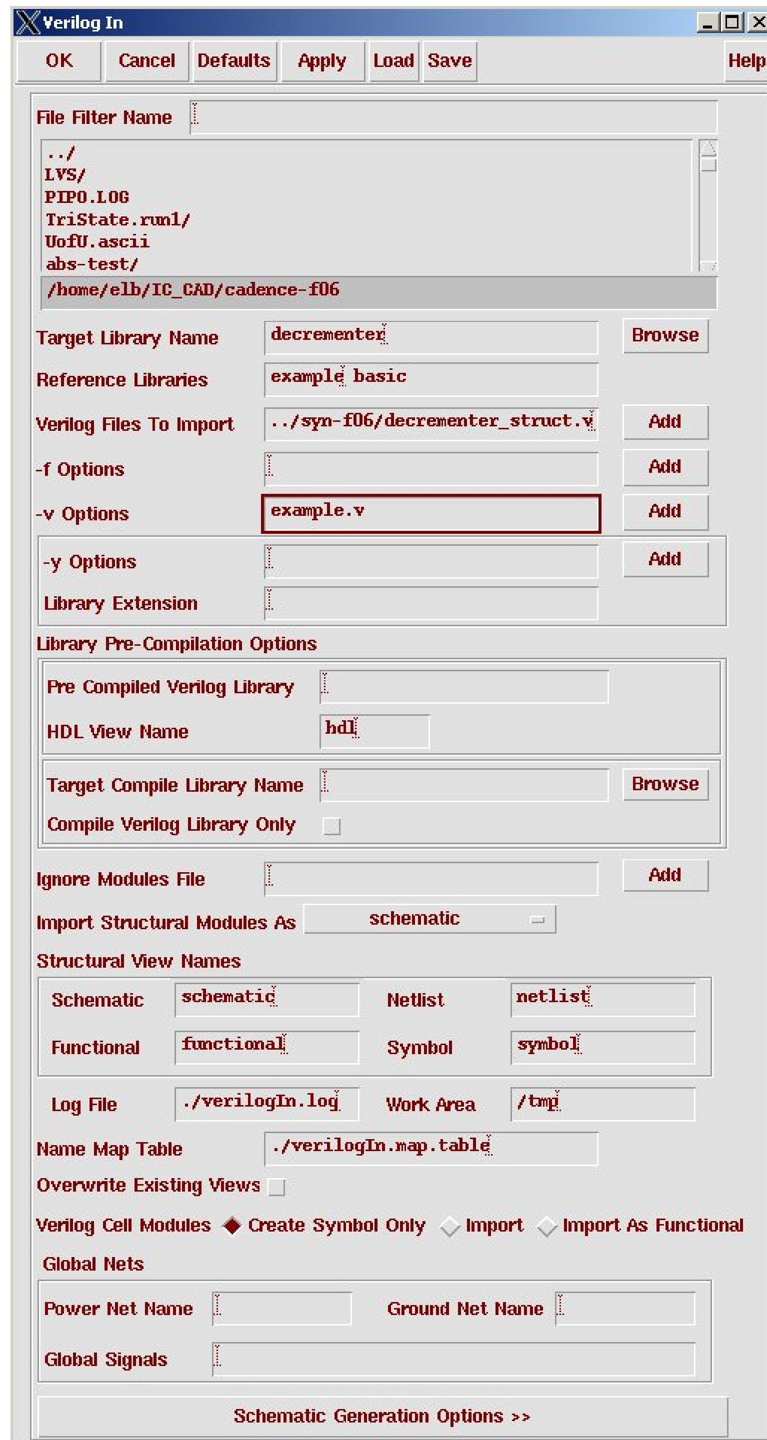

Figure 8.24: Dialog box for importing structural Verilog into a new schematic view

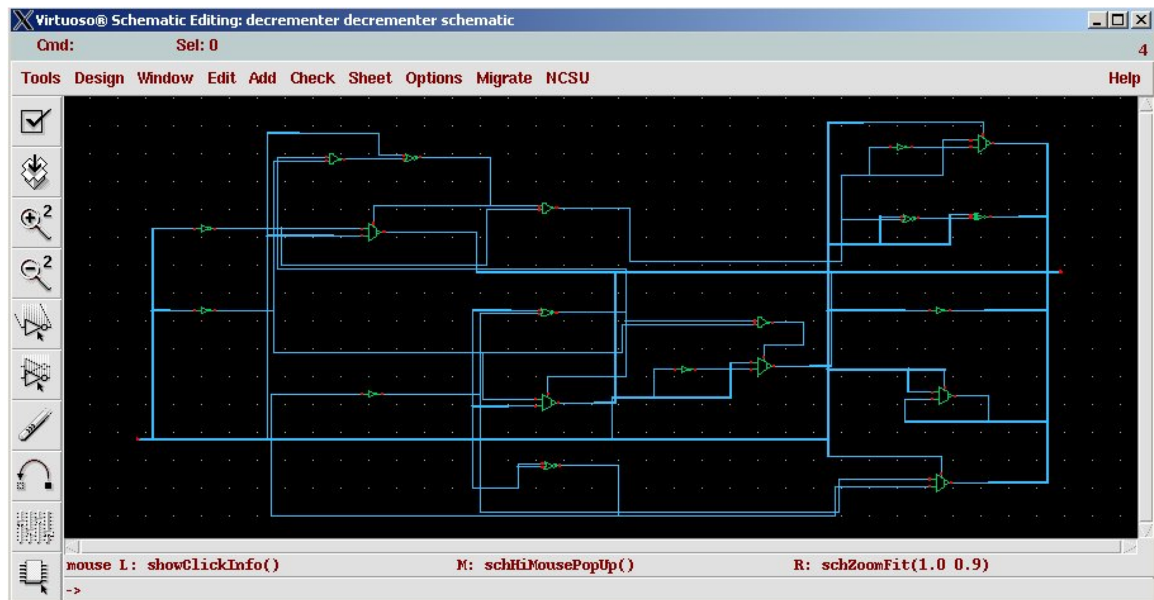

Figure 8.25: Schematic that results from importing the decrementer into **Composer**

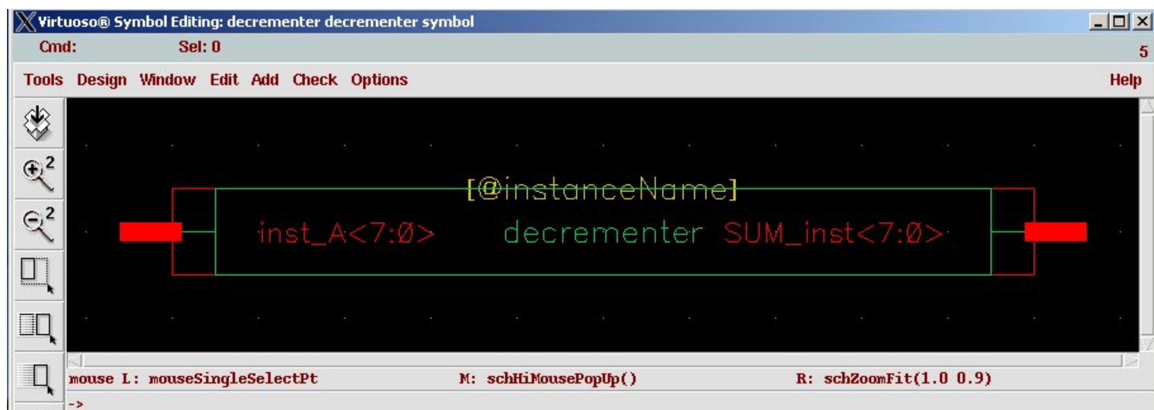

Figure 8.26: Symbol that is created for the decrementer