

RTL-to-Gates Synthesis using Synopsys Design Compiler

CS250 Tutorial 5 (Version 092509a)

September 25, 2009

Yunsup Lee

In this tutorial you will gain experience using Synopsys Design Compiler (DC) to perform hardware synthesis. A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as output. The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design. Internally, a synthesis tool performs many steps including high-level RTL optimizations, RTL to unoptimized boolean logic, technology independent optimizations, and finally technology mapping to the available standard cells. Good RTL designers will familiarize themselves with the target standard cell library so that they can develop a solid intuition on how their RTL will be synthesized into gates. In this tutorial you will use Synopsys Design Compiler to elaborate RTL, set optimization constraints, synthesize to gates, and prepare various area and timing reports. You will also learn how to read the various DC text reports and how to use the graphical Synopsys Design Vision tool to visualize the synthesized design.

Synopsys provides a library called Design Ware which includes highly optimized RTL for arithmetic building blocks. DC can automatically determine when to use Design Ware components and it can then efficiently synthesize these components into gate-level implementations. In this tutorial you will learn more about what Design Ware components are available and how to best encourage DC to use them.

The following documentation is located in the course locker (`~cs250/docs/manuals`) and provides additional information about Design Compiler, Design Vision, the Design Ware libraries, and the Synopsys 90nm Standard Cell Library.

- `dc-user-guide.pdf` - Design Compiler User Guide
- `dc-quick-reference.pdf` - Design Compiler Quick Reference
- `dc-user-guide-cli.pdf` - Design Compiler Command-Line Interface Guide
- `dc-user-guide-tcl.pdf` - Using Tcl With Synopsys Tools
- `dc-user-guide-tco.pdf` - Synopsys Timing Constraints and Optimization User Guide
- `dc-reference-manual-opt.pdf` - Design Compiler Optimization Reference Manual
- `dc-reference-manual-presto-verilog.pdf` - HDL Compiler Reference Manual
- `dc-application-note-sdc.pdf` - Synopsys Design Constraints Format Application Note
- `dc_dv-user-guide.pdf` - Design Vision User Guide
- `dc_dv-tutorial.pdf` - Design Compiler Tutorial Using Design Vision
- `designware-intro.pdf` - DesignWare Building Block IP Documentation Overview
- `designware-user-guide.pdf` - DesignWare Building Block IP
- `designware-quick-reference.pdf` - DesignWare Building Block IP Quick Reference
- `designware-datasheets` - Directory containing datasheets on each DW component
- `synopsys-90nm-databook-stdcells.pdf` - Digital Standard Cell Library Databook
- `synopsys-90nm-databook-memories.pdf` - Memory Databook
- `synopsys-90nm-databook-opensparc.pdf` - OpenSparc Megacell Databook

Getting started

Before using the CS250 toolflow you must run the course setup script with the following command.

```
% source ~cs250/tools/cs250.bashrc
```

For this tutorial you will be using an unpipelined SMIPSV1 processor as your example RTL design. You should create a working directory and copy files from the course locker using the following commands.

```
% mkdir tut5
% cd tut5
% TUT5_ROOT='pwd'
% cp -R ~cs250/examples/v-smipsv1-1stage/* $TUT5_ROOT
```

Before starting, take a look at the subdirectories in the project directory. Figure 1 shows the system diagram which is implemented by the example code. When pushing designs through the physical toolflow you will often refer to the *core*. The core module contains everything which will be on-chip, while blocks outside the core are assumed to be off-chip. For this tutorial you are assuming that the processor and a *combinational memory* are located within the core. A combinational memory means that the read address is specified at the beginning of the cycle, and the read data returns during the same cycle. Building large combinational memories is relatively inefficient. It is much more common to use *synchronous memories*. A synchronous memory means that the read address is specified at the end of a cycle, and the read data returns during the next cycle. From Figure 1 it should be clear that the unpipelined SMIPSV1 processor requires combinational memories (or else it would turn into a four stage pipeline). **For this tutorial you will not be using a real combinational memory, but instead you will use a dummy memory to emulate the**

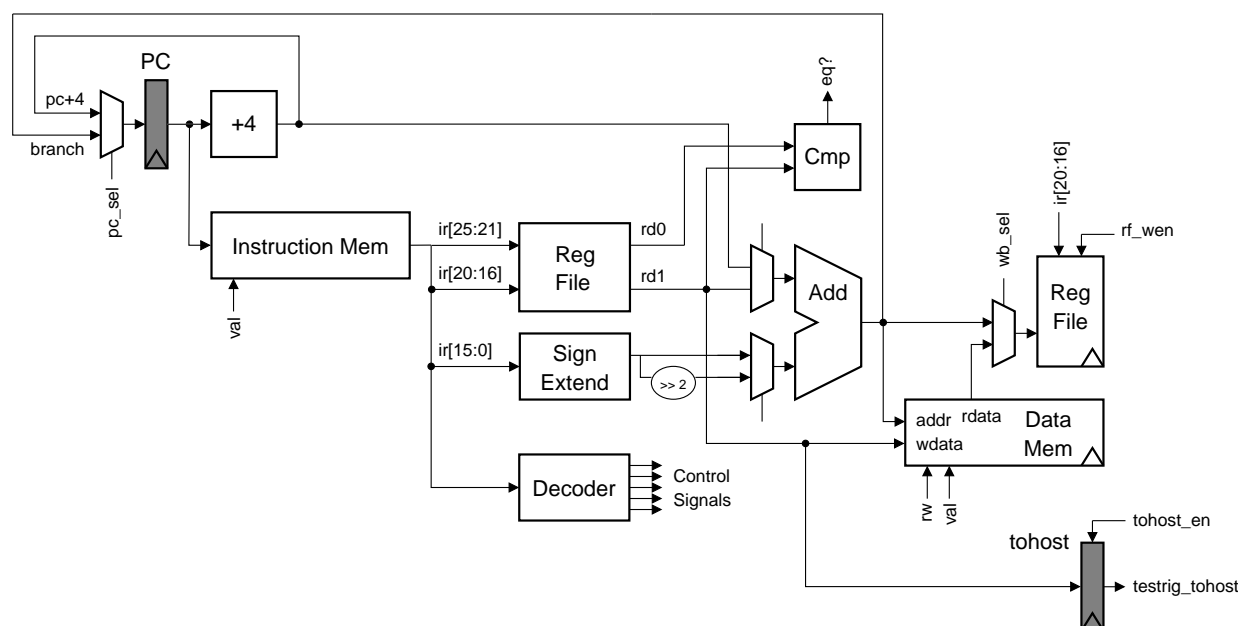


Figure 1: Block diagram for Unpipelined SMIPSV1 Processor

combinational delay through the memory. Examine the source code in `src` and compare `smipsCore_rtl` with `smipsCore_synth`. The `smipsCore_rtl` module is used for simulating the RTL of the SMIPSV1 processor and it includes a functional model for a large on-chip combinational memory. The `smipsCore_synth` module is used for synthesizing the SMIPSV1 processor and it uses a dummy memory. The dummy memory combinationally connects the memory request bus to the memory response bus with a series of standard-cell buffers. Obviously, this is not functionally correct, but it will help you illustrate more reasonable critical paths in the design. In later tutorials, you will start using synchronous on-chip SRAMs.

Now examine the `build` directory. This directory will contain all generated content including simulators, synthesized gate-level Verilog, and final layout. In this course you will always try to keep generated content separate from your source RTL. This keeps your project directories well organized, and helps prevent you from unintentionally modifying your source RTL. There are sub-directories in the `build` directory for each major step in the CS250 toolflow. These subdirectories contain scripts and configuration files for running the tools required for that step in the toolflow. For this tutorial we will work exclusively in the `dc-syn`. Notice that there are two makefiles in the `dc-syn` directory. Since we are using dummy memory go ahead and make a symlink to `Makefile.dummy`.

```
% cd $TUT5_ROOT/build/dc-syn
% ln -s Makefile.dummy Makefile
```

Synthesizing the Processor

You will begin by running several DC commands manually before learning how you can automate the tool with scripts. DC can generate a large number of output files, so you will be running DC within a build directory beneath `dc-syn`. Use the following commands to create a build directory for DC and to start the DC shell.

```
% cd $TUT5_ROOT/build/dc-syn
% mkdir manual
% cd manual
% dc_shell-xg-t
...
Initializing...
dc_shell>
```

You should be left at the DC shell prompt from which you can execute various commands to load in your design, specify constraints, synthesize your design, print reports, etc. You can get more information about a specific command by entering `man <command>` at the `dc_shell` prompt. You will now execute some commands to setup your environment.

```
dc_shell> set_app_var search_path \
    "~/cs250/stdcells/synopsys-90nm/default/db/cells \
    ~/cs250/install/vclib \
    ../../../../src"
dc_shell> set_app_var target_library "cells.db"
dc_shell> set_app_var synthetic_library "dw_foundation.sldb"
```

```
dc_shell> set_app_var link_library "*" $target_library $synthetic_library"
dc_shell> set_app_var alib_library_analysis_path \
    "/home/ff/cs250/stdcells/synopsys-90nm/default/alib"
dc_shell> define_design_lib WORK -path "./work"
```

These commands point to your Verilog source directory, create a Synopsys work directory, and point to the standard libraries you will be using for this class. The DB files contain wireload models and timing/area information for each standard cell. DC will use this information to try and optimize the synthesis process. You can now load your Verilog design into Design Compiler with the `analyze`, `elaborate`, and `link` commands. Executing these commands will result in a great deal of log output as the tool elaborates some Verilog constructs and starts to infer some high-level components. Try executing the commands as follows. To prevent the dummy memory being optimized away, we tell DC not to touch `imem_read_delay` and `dmem_read_delay`.

```
dc_shell> analyze -format verilog \
    "vcMuxes.v vcStateElements.v vcRAMs.v vcArith.v \
    smipsInst.v smipsProcCtrl.v smipsProcDpathRegfile.v
    smipsProcDpath_pstr.v smipsProc.v smipsCore_synth.v"
dc_shell> elaborate "smipsCore_synth"
dc_shell> link
dc_shell> set_dont_touch "dmem/imem_read_delay dmem/dmem_read_delay"
```

Take a closer look at the output during elaboration. DC will report all state inferences. This is a good way to verify that latches and flip-flops are not being accidentally inferred. You should be able to check that the only inferred state elements are the PC, the `tohost` register, a one-bit reset register, and the register file. DC will also note information about inferred muxes. Figure 2 shows a fragment from the elaboration output text. From this output you can see that DC is inferring 32-bit flip-flops for the register file and two 32 input 32-bit muxes for the register file read ports. See the *HDL Compiler Presto Verilog Reference Manual* ([dc-reference-manual-presto-verilog.pdf](#)) for more information on the output from the `elaborate` command and more generally how DC infers combinational and sequential hardware elements.

After reading your design into DC you can use the `check_design` command to check that the design is consistent. A consistent design is one which does not contain any errors such as unconnected ports, constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets, connection class violations, or recursive hierarchy definitions. You will not be able to synthesize your design until you eliminate any errors. Many of these warning are obviously not an issue, but it is still useful to skim through this output.

```
dc_shell> check_design
```

Before you can synthesize your design, you must specify some constraints; most importantly you must tell the tool your target clock period. The following commands tell the tool that the pin named `clk` is the clock and that your desired clock period is 2 nanoseconds. You need to set the clock period constraint carefully. If the period is unrealistically small, then the tools will spend forever trying to meet timing and ultimately fail. If the period is too large, then the tools will have no trouble but you will get a very conservative implementation. For more information about constraints consult the *Synopsys Timing Constraints and Optimization User Guide* ([dc-user-guide-tco.pdf](#)).

```
dc_shell> create_clock clk -name ideal_clock1 -period 2
```

Now you are ready to use the `compile_ultra` command to actually synthesize your design into a gate-level netlist. `-no_autoungroup` is specified in order to preserve the hierarchy during synthesis or disable inter-module optimizations. With no options given, `compile_ultra` command optimizes across module boundaries. Set `compile_ultra_ungroup_dw` to false to prevent ungrouping Design Ware hierarchies as well as `compile_seqmap_propagate_constants` to false to disable DC's constant propagation optimization by running this command. Because of emulating the gate delay of the combinational memory by putting dummy buffers in, DC is going to optimize away a lot of gates, and as a result it will be hard to understand the synthesis reports. However, in a real situation, you would like to use synthesis with both options turned on. For more information on the `compile_ultra` command consult the *Design Compiler User Guide* (`dc-user-guide.pdf`) or use `man compile_ultra` at the DC shell prompt. Run the following command and take a look at the output.

DC will attempt to synthesize your design while still meeting the constraints. DC considers two types of constraints: *user specified constraints* and *design rule constraints*. User specified constraints can be used to constrain the clock period (as you saw with the `create_clock` command) but they can also be used to constrain the arrival of certain input signals, the drive strength of the input signals, and the capacitive load on the output signals. Design rule constraints are fixed constraints which are specified by the standard cell library. For example, there are restrictions on the loads specific gates can drive and on the transition times of certain pins. For more information consult *Synopsys Design Constraints Format Application Note* (`dc-application-note-sdc.pdf`).

```
dc_shell> set_app_var compile_ultra_ungroup_dw false
dc_shell> set_app_var compile_seqmap_propagate_constants false
dc_shell> compile_ultra -no_autoungroup
```

The compile command will report how the design is being optimized. You should see DC performing technology mapping, delay optimization, and area reduction. Figure 3 shows a fragment from the compile output. Each line is an optimization pass. The area column is in units specific to the standard cell library which is um^2 , but for now you should just use the area numbers as a relative

```
in routine smipsProcDpathRegfile line 26 in file
'../../../../src/smipsProcDpathRegfile.v'.
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
| registers_reg | Flip-flop | 32 | Y | N | N | N | N | N | N |
...
| registers_reg | Flip-flop | 32 | Y | N | N | N | N | N | N |
=====
Statistics for MUX_OPs
=====
| block name/line | Inputs | Outputs | # sel inputs | MB |
=====
| smipsProcDpathRegfile/22 | 32 | 32 | 5 | N |
| smipsProcDpathRegfile/23 | 32 | 32 | 5 | N |
=====
```

Figure 2: Output from the Design Compiler `elaborate` command

metric. The worst negative slack column shows how much room there is between the critical path in your design and the clock constraint. Larger negative slack values are worse since this means that your design is missing the desired clock frequency by a greater amount. Total negative slack is the sum of all negative slack across all endpoints in the design - if this is a large negative number it indicates that not only is the design not making timing, but it is possible that *many* paths are too slow. If the total negative slack is a small negative number, then this indicates that only a few paths are too slow. The design rule cost is a indication of how many cells violate one of the standard cell library design rules constraints. Figure 3 shows that on the first iteration, the tool makes timing but at a high area cost, so on the second iteration it optimizes area but this causes the design to no longer meet timing. The tool continues to optimize until it meets the constraints.

```
...
Beginning Delay Optimization Phase
-----
```

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:05:45	98690.3	0.45	382.2	0.0	
0:05:53	99117.5	0.43	376.0	0.0	
0:05:59	99455.9	0.42	362.3	0.0	
...					
0:11:55	124976.8	0.04	0.9	0.0	
0:12:01	124885.5	0.00	0.0	0.0	
0:12:01	124885.5	0.00	0.0	0.0	

```
Beginning Area-Recovery Phase (max_area 0)
-----
```

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:12:01	124885.5	0.00	0.0	0.0	
...					
0:13:07	116502.9	0.00	0.0	0.0	
0:13:07	116496.8	0.00	0.0	0.0	
...					

Figure 3: Output from the Design Compiler `compile_ultra` command

You can now use various commands to examine timing paths, display reports, and further optimize your design. Entering in these commands by hand can be tedious and error prone, plus doing so makes it difficult to reproduce a result. Thus you will mostly use TCL scripts to control the tool. Even so, using the shell directly is useful for finding out more information about a specific command or playing with various options.

Before continuing, exit the DC shell and delete your build directory with the following commands.

```
dc_shell> exit
% cd $TUT5_ROOT/build/dc-syn
% rm -rf manual
```

Automating Synthesis with TCL Scripts and Makefiles

In this section you will examine how to use various TCL scripts and makefiles to automate the synthesis process. There are four files in the `build/dc-syn` directory.

- `Makefile` - Makefile for driving synthesis with the TCL scripts
- `dc_scripts/dc.tcl` - Primary TCL script which contains the DC commands
- `scripts/dc_setup.tcl` - TCL fragment which will setup various library variables
- `constraints.tcl` - User specified constraints

First take a look at the `dc_setup.tcl` script. You will see that it sets up several library variables, creates the search path, and instructs DC to use a work directory. The first line of the `dc_setup.tcl` script loads the `make_generated_vars.tcl` script. This script is generated by the makefile and it contains variables which are defined by the makefile and used by the TCL scripts. You will take a closer look at it in a moment. Now examine the `dc.tcl` script. You will see many familiar commands which we executed by hand in the previous section. You will also see some new commands. Take a closer look at the bottom of this TCL script where we write out several text reports. Remember that you can get more information on any command by using `man <command>` at the DC shell prompt. The `constraints.tcl` file contains various user specified constraints. This is where you constrain the clock period. You also specify that DC should assume that minimum sized inverters are driving the inputs to the design and that the outputs must drive 4 fF of capacitance.

Now that you are more familiar with the various TCL scripts, you will see how to use the makefile to drive synthesis. Look inside the makefile and identify where the Verilog sources are defined. Notice that you are using `smipsCore_synth.v` instead of `smipsCore_rtl.v` and that the test harness is not included. You should only list those Verilog files which are part of the core; do not include non-synthesizable test harnesses modules. Also notice that we *must* identify the toplevel Verilog module in the design. You also specify several modules in the `dont_touch` make variable. Any modules which you list here will be marked with the DC `set_dont_touch` command. DC will not optimize any modules which are marked don't touch. In this tutorial you are marking the dummy memories don't touch so that DC does not completely optimize away the buffer chain you are using to model the combinational delay through the memory. The build rules in the makefile will create new build directories, copy the TCL scripts into these build directories, and then run DC. Use the following make target to create a new build directory.

```
% cd $TUT5_ROOT/build/dc-syn
% make new-build-dir
```

You should now see a new build directory named `build-<date>` where `<date>` represents the time and date. The `current-dc` symlink always points to the most recent build directory. If you look inside the build directory, you will see the `dc_setup.tcl`, `dc.tcl`, and `constraints.tcl` scripts but you will also see an additional `make_generated_vars.tcl` script. Various variables inside `make_generated_vars.tcl` are used to specify the search path, which Verilog files to read in, which modules should be marked don't touch, the toplevel Verilog name, etc. After using `make new-build-dir` you can `cd` into the `current-dc` directory, start the DC shell, and run DC commands by hand. For example, the following sequence will perform the same steps as in the previous section.

```
% cd $TUT5_ROOT/build/dc-syn
% cd current-dc
% dc_shell-xg-t
dc_shell> source dc_setup.tcl
dc_shell> define_design_lib WORK -path ./work
dc_shell> analyze -format verilog ${RTL_SOURCE_FILES}
dc_shell> elaborate ${DESIGN_NAME}
dc_shell> link
dc_shell> source constraints.tcl
dc_shell> compile_ultra -no_autoungroup
dc_shell> exit
```

The `new-build-dir` make target is useful when you want to conveniently run through some DC commands by hand to try them out. To completely automate your synthesis you can use the `dc` make target (which is also the default make target). For example, the following commands will automatically synthesize the design and save several text reports to the build directory.

```
% cd $TUT5_ROOT/build/dc-syn
% make dc
```

You should see DC compiler start and then execute the commands located in the `dc.tcl` script. Once synthesis is finished try running `make dc` again. The makefile will detect that nothing has changed (i.e. the Verilog source files and DC scripts are the same) and so it does nothing. Make a change to one of the TCL scripts. Edit `constraints.tcl` and change the clock period constraint to 10 ns. Now use `make dc` to resynthesize the design. Since a TCL script has changed, make will correctly run DC again. Take a look at the current contents of `dc-syn`.

```
% cd $TUT5_ROOT/build/dc-syn
% ls -l
drwxr-xr-x 6 cs250 cs250 4096 2009-08-31 21:27 build-dc-2009-08-31_21-17
drwxr-xr-x 6 cs250 cs250 4096 2009-08-31 21:30 build-dc-2009-08-31_21-28
-rw-r--r-- 1 cs250 cs250 1109 2009-08-31 21:28 constraints.tcl
lrwxrwxrwx 1 cs250 cs250 25 2009-08-31 21:28 current-dc -> build-dc-2009-08-31_21-28
drwxr-xr-x 2 cs250 cs250 4096 2009-08-31 20:36 dc_scripts
drwxr-xr-x 2 cs250 cs250 4096 2009-08-14 16:49 fm_scripts
lrwxrwxrwx 1 cs250 cs250 14 2009-08-31 21:07 Makefile -> Makefile.dummy
-rw-r--r-- 1 cs250 cs250 3911 2009-08-31 21:06 Makefile.combinational
-rw-r--r-- 1 cs250 cs250 4121 2009-08-31 21:01 Makefile.dummy
drwxr-xr-x 2 cs250 cs250 4096 2009-08-31 20:48 scripts
```

Notice that the makefile does *not* overwrite build directories. It always creates new build directories. This makes it easy to change your synthesis scripts or source Verilog, resynthesize your design, and compare your results to previous designs. You can use symlinks to keep track of what various build directories correspond to. For example, the following commands label the build directory which corresponds to a 2 ns clock period constraint and the build directory which corresponds to a 10 ns clock period constraint.

```
% cd $TUT5_ROOT/build/dc-syn
% ln -s build-dc-2009-08-31_21-17/ build-2ns
% ln -s build-dc-2009-08-31_21-28/ build-10ns
```


Every so often you should delete old build directories to save space. The `make clean` command will delete *all* build directories so use it carefully. Sometimes you want to really force the makefile to resynthesize the design but for some reason it may not work properly. To force a resynthesis without doing a `make clean` simply remove the `current` symlink. For example, the following commands will force a resynthesis without actually changing any of the source TCL scripts or Verilog.

```
% cd $TUT5_ROOT/build/dc-syn
% rm -f current
% make dc
```

Interpreting the Synthesized Gate-Level Netlist and Text Reports

In this section you will examine some of the output which our `dc.tcl` script generates. You will initially focus on the contents of the `build-10ns` build directory. The primary output from the synthesis scripts is the synthesized gate-level netlist which is contained in `results/smipsCore_synth.mapped.v`. Take a look at the gate-level netlist for the 10ns clock constraint. Notice that the RTL module hierarchy is preserved in the gate-level netlist since you did not flatten any part of your design. Find the four two-input multiplexers in the gate-level netlist by searching for `vcMux2`. Although the same two-input mux was instantiated four times in the design (the PC mux, the ALU operand muxes, and the writeback mux), DC has optimized each multiplexer differently. Figure 4 shows the gate-level netlist for two of the synthesized multiplexers.

Use the databook for the Synopsys 90nm Standard Cell Library (`synopsys-90nm-databook-stdcells.pdf`) to determine the function of the `MUX21X1` standard cell. You should discover that this is a 2 input 1-bit mux cell. From the gate-level netlist you can determine that these are the operand muxes for the ALU and that `vcMux2_W32_3` is used to select between the two sign-extension options. Notice that the `vcMux2_W32_2` mux uses 32 mux cells, while `vcMux2_W32_3` uses only 30 mux cells. DC has discovered that the low-order two bits of one of the inputs to the `vcMux2_W32_3` mux are always zero (this corresponds to the two zeros which are inserted after shifting the sign-extension two bits to the left). So DC has replaced mux cells with an inverter-nor combination for these two low-order bits. Also notice that both mux modules include an extra buffers. Carefully tracing the netlist shows that these buffers are used to drive the select lines to the mux cells. DC does some very rough buffer insertion, but DC's primitive wireload models usually result in very conservative buffering. You can compare this to the buffer insertion which occurs during place and route. After place and route the tools are able to use much better wire modeling and as a result produce much better buffer insertion.

In addition to the actual synthesized gate-level netlist, the `dc.tcl` also generates several reports. Reports usually have the `rpt` filename suffix. The following is a list of the synthesis reports.

- `reports/*.mapped.area.rpt` - Area information for each module instance
- `reports/*.mapped.power.rpt` - Power information for each module instance
- `reports/*.mapped.resources.rpt` - Information on Design Ware components
- `reports/*.mapped.reference.rpt` - Information on references
- `reports/*.mapped.timing.rpt` - Contains critical timing paths
- `reports/*.mapped.qor.rpt` - QoR (Quality of Result) information and statistics
- `log/dc.log` - Log file of all output during DC run

```

module vcMux2_W32_0_MUX_OP_2_1_32_2 ( ... );
  input ...;
  output ...;
  wire  n1, n2, n3;

  NBUFFX2 U1 ( .IN(S0), .Q(n1) );
  NOR2X0 U2 ( .IN1(n1), .IN2(n3), .QN(Z_30) );
  NOR2X0 U3 ( .IN1(n1), .IN2(n2), .QN(Z_31) );
  MUX21X1 U4 ( .IN1(D0_2), .IN2(D1_2), .S(S0), .Q(Z_2) );
  MUX21X1 U5 ( .IN1(D0_3), .IN2(D1_3), .S(S0), .Q(Z_3) );
  ... 26 additional MUX21X1 instantiations ...
  MUX21X1 U30 ( .IN1(D0_28), .IN2(D1_28), .S(n1), .Q(Z_28) );
  MUX21X1 U31 ( .IN1(D0_29), .IN2(D1_29), .S(n1), .Q(Z_29) );
  INVX0 U32 ( .IN(D0_31), .QN(n2) );
  INVX0 U33 ( .IN(D0_30), .QN(n3) );
  MUX21X1 U34 ( .IN1(D0_0), .IN2(D1_0), .S(S0), .Q(Z_0) );
  MUX21X1 U35 ( .IN1(D0_1), .IN2(D1_1), .S(S0), .Q(Z_1) );
endmodule

module vcMux2_W32_3 ( in0, in1, sel, out );
  input [31:0] in0;
  input [31:0] in1;
  output [31:0] out;
  input sel;

  vcMux2_W32_0_MUX_OP_2_1_32_2 C40 ( ... );
endmodule

module vcMux2_W32_0_MUX_OP_2_1_32_1 ( ... );
  input ...;
  output ...;
  wire  n1;

  NBUFFX2 U1 ( .IN(S0), .Q(n1) );
  MUX21X1 U2 ( .IN1(D0_7), .IN2(D1_7), .S(n1), .Q(Z_7) );
  MUX21X1 U3 ( .IN1(D0_8), .IN2(D1_8), .S(n1), .Q(Z_8) );
  ... 28 additional MUX21X1 instantiations ...
  MUX21X1 U32 ( .IN1(D0_2), .IN2(D1_2), .S(S0), .Q(Z_2) );
  MUX21X1 U33 ( .IN1(D0_3), .IN2(D1_3), .S(S0), .Q(Z_3) );
endmodule

module vcMux2_W32_2 ( in0, in1, sel, out );
  input [31:0] in0;
  input [31:0] in1;
  output [31:0] out;
  input sel;

  vcMux2_W32_0_MUX_OP_2_1_32_1 C40 ( ... );
endmodule

```

Figure 4: Gate-Level Netlist for Two Synthesized 32 Input 32-bit Muxes

In this section you will discuss the `area.rpt`, `timing.rpt`, and the `reference.rpt` reports. The next section will discuss the `resources.rpt` report. The `area.rpt` report contains area information for each module in the design. Figure 5 shows a fragment from `area.rpt` for the SMIPSV1 unpipelined processor. You can use the `reference.rpt` report (Figure 6) to gain insight into how various modules are being implemented. For example, you can use the reference report in a similar fashion as the `results/smipsCore_synth.mapped.v` gate-level netlist to see that the `vcMux2_W32_3` module includes only 30 mux cells and uses bit-level optimizations for the remaining two bits.

You can also use the area report to measure the relative area of the various modules. The report clearly shows that the majority of the processor area is in the datapath. More specifically you can see that register file consumes 85% of the total processor area. The reference report reveals that the register file is being implemented with approximately 1000 enable flip-flops and input muxes (for the read ports). This is a very inefficient way to implement a register file, but it is the best the synthesizer can do. Real ASIC designers rarely synthesize memories and instead turn to *memory generators*. A memory generator is a tool which takes an abstract description of the memory block as input and produces a memory in formats suitable for various tools. Memory generators use custom cells and procedural place+route to achieve an implementation which can be an order of magnitude better in terms of performance and area than synthesized memories.

Figure 7 illustrates a fragment of the timing report found in `timing.rpt`. The report lists the *critical path* of the design. The critical path is the slowest logic path between any two registers and is therefore the limiting factor preventing you from decreasing the clock period constraint (and thus increasing performance). The report is generated from a purely static worst-case timing analysis (i.e. independent of the actual signals which are active when the processor is running). The first column lists various nodes in the design. Note that several nodes internal to higher level modules have been cut out to save space. The last column lists the cumulative delay to that node, while the middle column shows the incremental delay. You can see that the critical path starts at bit 21 of the PC register; goes through the combinational read of the instruction memory; goes through the read address of the register file and out the read data port; goes through the operand mux; through the adder; out the data memory address port and back in the data memory response port; through the writeback mux; and finally ends at bit 31 of register 1 in the register file. The large buffers in the memory (the `A0BUF1` cell in the `dmem` module) model the combinational delay through these memories. You can use the delay column to get a feel for how much each module contributes to the critical path: the combinational memories contribute about 0.2 ns; the register file read contributes about 1.2 ns; the adder contributes 4.5 ns; and the write back mux requires 0.5 ns.

The critical path takes a total of 7.94 ns which is less than the 10 ns clock period constraint. Notice, however, that the final register file flip-flop has a setup time of 0.09 ns. So the critical path plus the setup time ($7.94 \text{ ns} + 0.09 \text{ ns} = 8.03 \text{ ns}$) is just fast enough to meet the clock period constraint.

Synopsys Design Ware Libraries

Synopsys provides a library of commonly used arithmetic components as highly optimized building blocks. This library is called Design Ware and DC will automatically use Design Ware components when it can. To get a feel for what type of components are available, take a look at the *Design Ware Quick Reference Guide* (`designware-quick-reference.pdf`). The components you will be using in the class are the Building Block IP described in Chapter 2.

Hierarchical cell	Global cell area		Local cell area		
	Absolute Total	Percent Total	Combi- national	Noncombi- national	Black boxes
smipsCore_synth	109322.2266	100.0	0.0000	0.0000	0.0000
dmem	5937.3125	5.4	260.2711	0.0000	0.0000
dmem/dmem_read_delay	2838.5305	2.6	2838.5305	0.0000	0.0000
dmem/imem_read_delay	2838.5305	2.6	2838.5305	0.0000	0.0000
proc	103359.8906	94.5	0.0000	0.0000	0.0000
proc/ctrl	902.4932	0.8	594.6781	0.0000	0.0000
proc/ctrl/tohost_pf	307.8150	0.3	108.7510	199.0640	0.0000
proc/dpath	102457.2656	93.7	93.0850	0.0000	0.0000
proc/dpath/adder	3842.9329	3.5	0.0000	0.0000	0.0000
proc/dpath/adder/add_x_28_1	3842.9329	3.5	3842.9329	0.0000	0.0000
proc/dpath/branch_cond_gen	740.9798	0.7	740.9798	0.0000	0.0000
proc/dpath/op0_mux	393.4940	0.4	0.0000	0.0000	0.0000
proc/dpath/op0_mux/C40	393.4940	0.4	393.4940	0.0000	0.0000
proc/dpath/op1_mux	1005.2824	0.9	0.0000	0.0000	0.0000
proc/dpath/op1_mux/C40	1005.2824	0.9	1005.2824	0.0000	0.0000
proc/dpath/pc_inc4	939.8579	0.9	0.0000	0.0000	0.0000
proc/dpath/pc_inc4/add_x_56_1	939.8579	0.9	939.8579	0.0000	0.0000
proc/dpath/pc_mux	690.4892	0.6	0.0000	0.0000	0.0000
proc/dpath/pc_mux/C40	690.4892	0.6	690.4892	0.0000	0.0000
proc/dpath/pc_pf	1271.3844	1.2	285.3800	986.0039	0.0000
proc/dpath/rfile	92731.5703	84.8	11953.2773	54706.9766	0.0000
proc/dpath/rfile/C4408	17474.4590	16.0	17474.4590	0.0000	0.0000
proc/dpath/rfile/C4409	8594.5879	7.9	8594.5879	0.0000	0.0000
proc/dpath/sext	171.4300	0.2	171.4300	0.0000	0.0000
proc/dpath/wb_mux	576.2928	0.5	0.0000	0.0000	0.0000
proc/dpath/wb_mux/C40	576.2928	0.5	576.2928	0.0000	0.0000
reset0_pf	24.8830	0.0	0.0000	24.8830	0.0000
Total			53402.3086	55916.9258	0.0000

Figure 5: Fragment from smipsCore_synth.mapped.area.rpt

```
*****
Design: vcMux2_W32_3 -> vcMux2_W32_0_MUX_OP_2_1_32_2
*****
```

Reference	Library	Unit Area	Count	Total Area
Attributes				
IN VX0	saed90nm_typ	5.530000	2	11.060000
MUX21X1	saed90nm_typ	11.059000	30	331.770000
NBUFFX2	saed90nm_typ	5.530000	1	5.530000
NOR2X0	saed90nm_typ	5.530000	2	11.060000
Total 4 references				359.420002

Figure 6: Fragment from smipsCore_synth.mapped.reference.rpt

Point	Fanout	Trans	Incr	Path
<hr/>				
clock ideal_clock1 (rise edge)			0.00	0.00
clock network delay (ideal)			0.00	0.00
proc/dpath/pc_pf/q_np_reg_21_/CLK (DFFX1)		0.00	0.00 #	0.00 r
proc/dpath/pc_pf/q_np_reg_21_/Q (DFFX1)		0.05	0.21	0.21 r
proc/dpath/pc_pf/q_np[21] (vcRDFF_pf_32_00001000)			0.00	0.21 r
proc/dpath/imemreq_bits_addr[21] (smipsProcDpath_pstr)			0.00	0.21 r
proc/imemreq_bits_addr[21] (smipsProc)			0.00	0.21 r
dmem/imemreq_bits_addr[21] (smipsDummyMemory_DELAY4)			0.00	0.21 r
dmem/imem_read_delay/in[21] (smipsDelayChain_WIDTH32_DELAY4)			0.00	0.21 r
dmem/imem_read_delay/row_0__bit_21__delay/Q (AOBUF1)	0.03		0.06	0.26 r
...				
dmem/imem_read_delay/row_3__bit_21__delay/Q (AOBUF1)	0.04		0.06	0.43 r
dmem/imem_read_delay/out[21] (smipsDelayChain_WIDTH32_DELAY4)			0.00	0.43 r
dmem/imemresp_bits_data[21] (smipsDummyMemory_DELAY4)			0.00	0.43 r
proc/imemresp_bits_data[21] (smipsProc)			0.00	0.43 r
proc/ctrl/imemresp_bits_data[21] (smipsProcCtrl)			0.00	0.43 r
proc/ctrl/U57/Q (NBUFFX32)	0.66		0.51	0.93 r
proc/ctrl/rf_raddr0[0] (smipsProcCtrl)			0.00	0.93 r
proc/dpath/rf_raddr0[0] (smipsProcDpath_pstr)			0.00	0.93 r
proc/dpath/rfile/raddr0[0] (smipsProcDpathRegfile)			0.00	0.93 r
...				
proc/dpath/rfile/rdata0[0] (smipsProcDpathRegfile)			0.00	2.10 r
proc/dpath/op1_mux/in1[0] (vcMux2_W32_2)			0.00	2.10 r
...				
proc/dpath/op1_mux/out[0] (vcMux2_W32_2)			0.00	2.19 r
proc/dpath/adder/in1[0] (vcAdder_simple_W32)			0.00	2.19 r
proc/dpath/adder/add_x_28_1/B[0] (vcAdder_simple_W32_DW01_add_0)			0.00	2.19 r
...				
proc/dpath/adder/add_x_28_1/SUM[31] (vcAdder_simple_W32_DW01_add_0)			0.00	6.92 r
proc/dpath/adder/out[31] (vcAdder_simple_W32)			0.00	6.92 r
proc/dpath/dmemreq_bits_addr[31] (smipsProcDpath_pstr)			0.00	6.92 r
proc/dmemreq_bits_addr[31] (smipsProc)			0.00	6.92 r
...				
proc/dmemresp_bits_data[31] (smipsProc)			0.00	7.21 r
proc/dpath/dmemresp_bits_data[31] (smipsProcDpath_pstr)			0.00	7.21 r
proc/dpath/wb_mux/in1[31] (vcMux2_W32_1)			0.00	7.21 r
...				
proc/dpath/wb_mux/out[31] (vcMux2_W32_1)			0.00	7.76 r
proc/dpath/rfile/wdata_p[31] (smipsProcDpathRegfile)			0.00	7.76 r
proc/dpath/rfile/U106/Q (MUX21X1)	0.11		0.18	7.94 r
proc/dpath/rfile/registers_reg_1__31_/D (DFFX1)	0.11		0.00	7.94 r
data arrival time				7.94
<hr/>				
clock ideal_clock1 (rise edge)			10.00	10.00
clock network delay (ideal)			0.00	10.00
proc/dpath/rfile/registers_reg_1__31_/CLK (DFFX1)			0.00	10.00 r
library setup time			-0.09	9.91
data required time				9.91
<hr/>				
data required time				9.91
data arrival time				-7.94
<hr/>				
slack (MET)				1.97

Figure 7: Fragment from synth_timing.rpt

The `reference.rpt` report can help you determine when DC is using Design Ware components. For example, if you look at the `vcAdder_simple_W32` module in `synth_area.rpt` you will see that it contains a single module named `vcAdder_simple_W32_DW01_add_0` which was not present in our original RTL module hierarchy. The `DW01_add` in the module name indicates that this is a Design Ware adder. To find out more information about this component you can refer to the corresponding Design Ware datasheet located in the locker (`~cs250/docs/manuals/designware-datasheets/dw01_add.pdf`). The data sheets contain information on the different component implementation types. For example, DC can use a ripple-carry adder, a carry-lookahead adder, delay-optimized flexible parallel-prefix adder, or an area-optimized flexible parallel-prefix adder. The `resources.rpt` report contains information on which implementation was chosen for each Design Ware component. Figure 8 shows a fragment from `resources.rpt` which indicates that the adder uses a `apparch` implementation. The `apparch` implementation is an area-optimized flexible parallel-prefix adder. Compare this to what is generated with the 2ns clock constraint. Look at the `resources.rpt` file in the `build-2ns` directory. Figure 9 shows that with the much faster clock period constraint, DC has chosen to use a delay-optimized flexible parallel-prefix adder (`pparch`). Although the area-optimized flexible parallel-prefix adder is slower than the delay-optimized parallel-prefix adder, it is still fast enough to meet the clock period constraint and it uses significantly less area.

There are two ways to use Design Ware components: inference or instantiation. For each component the corresponding datasheet outlines the appropriate Verilog RTL which should result in DC inferring that Design Ware component. Note that sometimes DC decides not to use a Design Ware component because it can do other optimizations which result in a better implementation. If you *really* want to try and force DC to use a specific Design Ware component you can instantiate the component directly. If you use direct instantiation you will need to include the appropriate Verilog model so that VCS can simulate the component. You can do this by adding the following command line parameter to VCS.

```
-y $DC_HOME/dw/sim_ver +libext+.v+
```

We suggest only using direct instantiation as a last resort since it creates a dependency between your high-level design and the Design Ware libraries, and it limits the options available to Design Compiler during synthesis.

=====				
Cell	Module	Parameters	Contained Operations	
=====				
add_x_28_1	DW01_add	width=32	add_28	
=====				
...				
=====				
		Current	Set	
Cell	Module	Implementation	Implementation	
=====				
add_x_28_1	DW01_add	apparch		
=====				

Figure 8: Fragment from `resources.rpt` for 10 ns clock period

```
=====
```

Cell	Module	Parameters	Contained Operations
add_x_28_1	DW01_add	width=32	add_28

```
=====
```

...

```
=====
```

Cell	Module	Current Implementation	Set Implementation
add_x_28_1	DW01_add	pparch	

```
=====
```

Figure 9: Fragment from `resources.rpt` for 2ns clock period

Using Design Vision to Analyze the Synthesized Gate-Level Netlist

Synopsys provides a GUI front-end to Design Compiler called Design Vision which you will use to analyze the synthesis results. You should avoid using the GUI to actually perform synthesis since you want to use scripts for this. To launch Design Vision and read in your synthesized design, move into the appropriate working directory and use the following commands.

```
% cd $TUT5_ROOT/build/dc-syn
% cd current
% design_vision-xg
design_vision> source dc_setup.tcl
design_vision> read_ddc results/smipsCore_synth.mapped.ddc
```

You can browse your design with the hierarchical view. If you right click on a module and choose the *Schematic View* option, the tool will display a schematic of the synthesized logic corresponding to that module. Figure 10 shows the schematic view for the datapath adder module with the 10 ns clock constraint. Notice the ripple-carry structure of the adder.

You can use Design Vision to examine various timing data. The *Timing > Paths Slack* menu option will create a histogram of the worst case timing paths in your design. You can use this histogram to gain some intuition on how to approach a design which does not meet timing. If there are a large number of paths which have a very large negative timing slack then a global solution is probably necessary, while if there are just one or two paths which are not making timing a more local approach may be sufficient. You can click on a bin and the tool will report critical paths in the bin. Figure 12 shows an example of using these two features. Go ahead and right click to choose *Path Inspector*. Figure ?? shows the actual components on the critical path.

It is sometimes useful to examine the critical path through a single submodule. To do this, right click on the module in the hierarchy view and use the *Characterize* option. Check the timing, constraints, and connections boxes and click OK. Now choose the module from the drop down list box on the toolbar (called the *Design List*). Choosing *Timing > Report Timing Path* will provide information on the critical path through that submodule given the constraints of the submodule within the overall design's context.

For more information on Design Vision consult the *Design Vision User Guide* (`dc_dv-user-guide.pdf`).

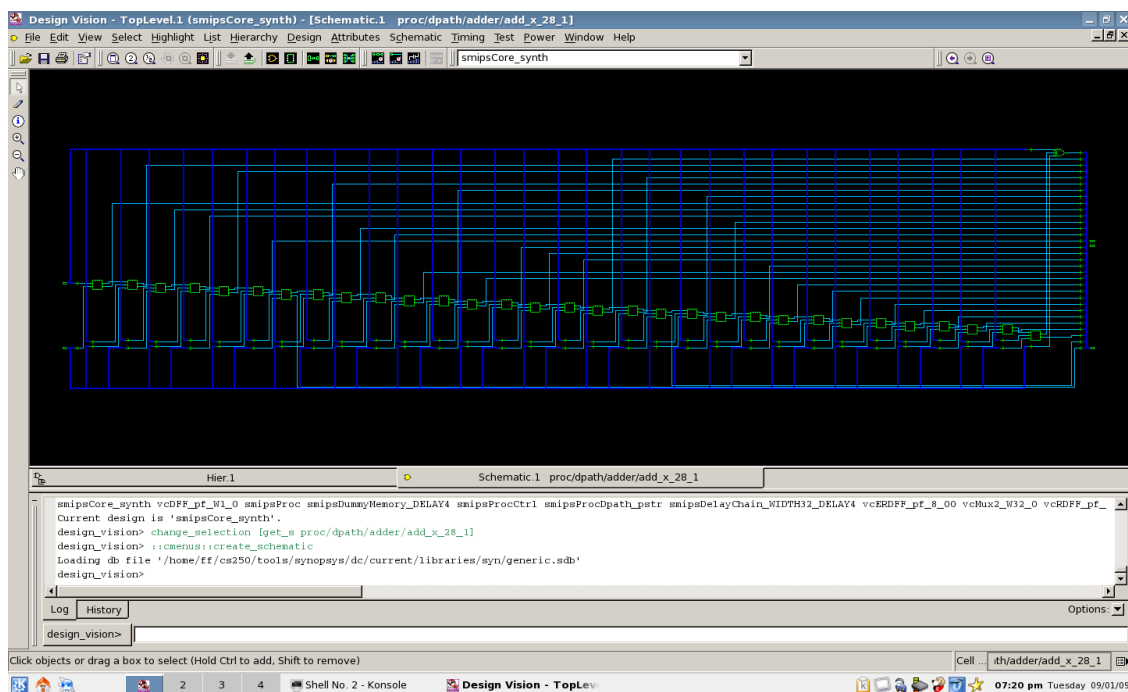


Figure 10: Screen shot of a schematic view in Design Vision

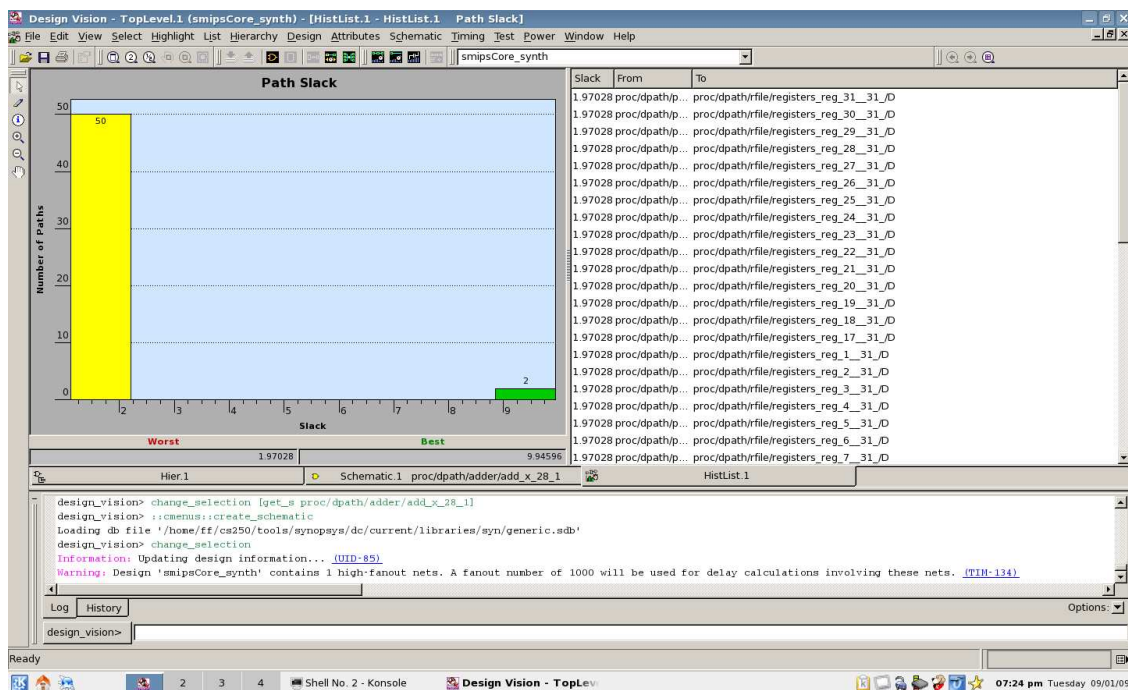


Figure 11: Screen shot of timing results in Design Vision

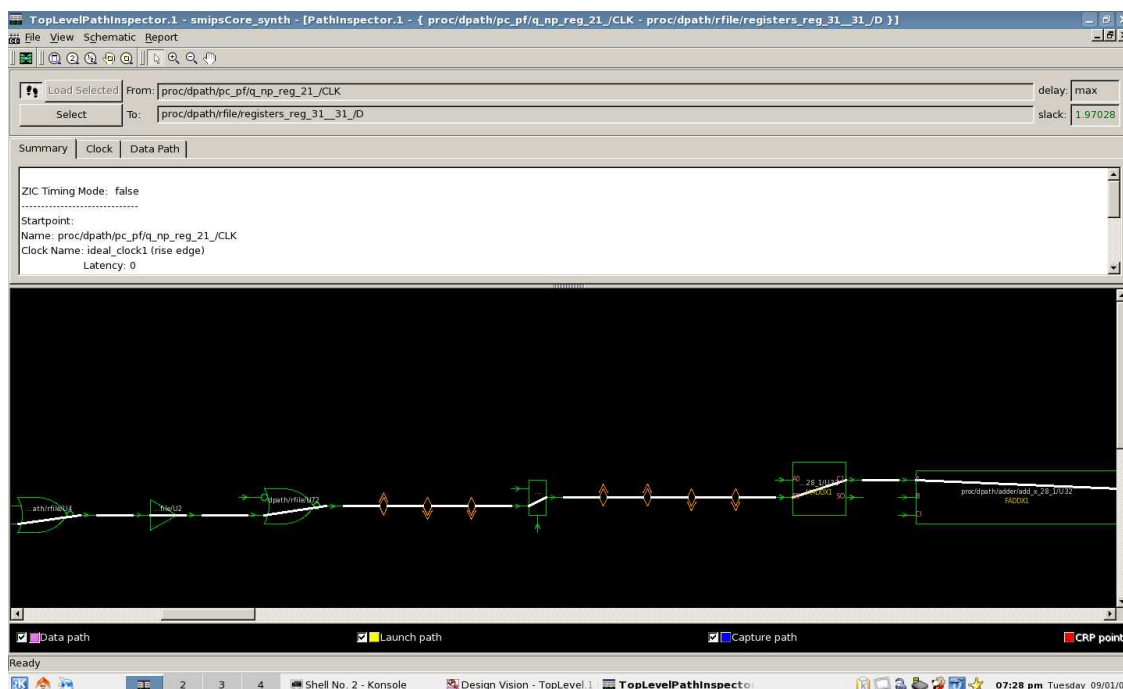


Figure 12: Screen shot of the path inspector in Design Vision

Review

The following sequence of commands will setup the CS250 toolflow, checkout the SMIPsv1 processor example, and synthesize the design.

```
% source ~cs250/tools/cs250.bashrc
% mkdir tut5
% cd tut5
% cp -R ~cs250/examples/v-smipsv1-1stage/* .
% cd build/dc-syn
% make
```

Acknowledgements

Many people have contributed to versions of this tutorial over the years. The tutorial was originally developed for 6.375 Complex Digital Systems course at Massachusetts Institute of Technology by Christopher Batten. Contributors include: Krste Asanović, John Lazzaro, Yunsup Lee, and John Wawrzynek. Versions of this tutorial have been used in the following courses:

- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CS250 VLSI Systems Design (2009) - University of California at Berkeley