RTL-to-Gates Synthesis using Synopsys Design Compiler

6.375 Tutorial 4
February 12, 2007

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. A synthesis tool is only as good as the standard cells which it has at its disposal. 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 we will use Synopsys Design Compiler to elaborate RTL, set optimization constraints, synthesize to gates, and prepare various area and timing reports.

Synopsys provides a library called Design Ware which includes highly optimized RTL for arithmetic building blocks. For example, the Design Ware libraries contain adders, multipliers, comparators, and shifters. 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 we will learn more about what Design Ware components are available and how to best encourage DC to use them.

It is important to carefully monitor the synthesis tool to identify issues which might negatively impact the area, power, or performance of the design. We will learn how to read the various DC text reports and how to use the graphical Synopsys Design Vision tool to visualize the synthesized design.

The following documentation is located in the course locker (`/mit/6.375/doc`) and provides additional information about Design Compiler, Design Vision, the Design Ware libraries, and the Tower 0.18 μm Standard Cell Library.

- `tsl-180nm-sc-databook.pdf` - Databook for Tower 0.18 μm Standard Cell Library
- `presto-HDL-compiler.pdf` - Guide for the Verilog Complier used by DC
- `dc-user-guide.pdf` - Design Compiler user guide
- `dc-quickref.pdf` - Design Compiler quick reference
- `dc-constraints.pdf` - Design Compiler constraints and timing reference
- `dc-opt-timing-analysis.pdf` - Design Compiler optimization and timing analysis ref
- `dc-shell-basics.pdf` - Basics of using the DC shell
- `dc-tcl-user-guide.pdf` - Basics of writing TCL scripts for Design Compiler
- `design-ware-quickref.pdf` - Design Ware quick reference
- `design-ware-user-guide.pdf` - Design Ware user guide
- `design-ware-datasheets` - Directory containing datasheets on each DW component
- `dv-tutorial.pdf` - Design Vision Tutorial
- `dv-user-guide.pdf` - Design Vision User Guide

Getting started

Before using the 6.375 toolflow you must add the course locker and run the course setup script with the following two commands.

```
% add 6.375
% source /mit/6.375/setup.csh
```

For this tutorial we will be using an unpipelined SMIPSv1 processor as our example RTL design. You should create a working directory and checkout the SMIPSv1 example project from the course CVS repository using the following commands.

```
% mkdir tut4
% cd tut4
% cvs checkout examples/smipsv1-1stage-v
% cd examples/smipsv1-1stage-v
```

Before starting, take a look at the subdirectories in the `smips1-1stage-v` project directory. Figure 1 shows the system diagram which is implemented by the example code. When pushing designs through the physical toolflow we will often refer to the *core*. The core module contains everything which will be on-chip, while blocks outside the core are assume to be off-chip. For this tutorial we 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 we will not be using a real combinational memory, but instead we will use a dummy memory to emulate

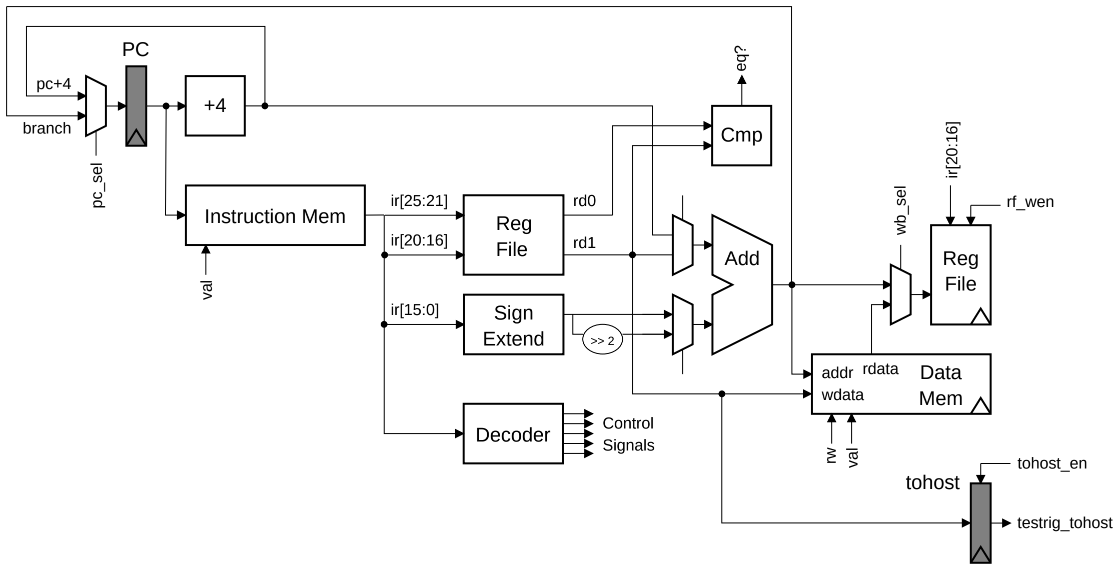


Figure 1: Block diagram for Unpipelined SMIPSv1 Processor

the 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 us illustrate more reasonable critical paths in the design. In later tutorials, we will start using memory generators which will create 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 we will always try to keep generated content separate from our source RTL. This keeps our project directories well organized, and helps prevent us from unintentionally modifying our source RTL. There are subdirectories in the `build` directory for each major step in the 6.375 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-synth` directory.

Synthesizing the Processor

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

```
% pwd
tut4/examples/smipsv1-1stage-v
% cd build/dc-synth
% mkdir build
% cd build
% dc_shell-xg-t
Initializing...
dc_shell-xg-t>
```

You should be left at the DC shell prompt from which you can 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-xg-t` prompt. We will now execute some commands to setup your environment.

```
dc_shell-xg-t> lappend search_path ../../../src
dc_shell-xg-t> lappend search_path /mit/6.375/install/vclib
dc_shell-xg-t> define_design_lib WORK -path "work"
dc_shell-xg-t> set link_library \
  [list /mit/6.375/libs/tsl180/tsl18fs120/db/tsl18fs120_typ.db]
dc_shell-xg-t> set target_library \
  [list /mit/6.375/libs/tsl180/tsl18fs120/db/tsl18fs120_typ.db]
```

These commands point to your Verilog source directory, create a Synopsys work directory, and point to the standard libraries we will be using for this class. The DB files contain wireload models and

```
Information: Building the design 'smipsProcDpathRegfile'. (HDL-193)
Inferred memory devices in process
        in routine smipsProcDpathRegfile line 27 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/23 |   32   |   32    |      5       | N  |
| smipsProcDpathRegfile/24 |   32   |   32    |      5       | N  |
===================================================================
```

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

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` and `elaborate` 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.

```
dc_shell-xg-t> analyze -library WORK -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-xg-t> elaborate smipsCore_synth -architecture verilog -library WORK
```

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 *Presto HDL Compiler Reference Manual* (`presto-HDL-compiler.pdf`) for more information on the output from the elaborate command and more generally how DC infers combinational and sequential hardware elements.

After reading our design into DC we 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. Run the `check_design` command as follows and take a look at the warnings. Many of these warning are obviously not an issue, but it is still useful to skim through this output.

```
  dc_shell-xg-t> 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 5 nanoseconds. We need to set the clock period constraint carefully. If the period is unrealistically small, then the tools will spend forever trying to meet timing and ultimately fail. If the period is too large, then the tools will have no trouble but you will get a very conservative implementation. For more information about constraints consult the *Design Compiler Constraints and Timing Reference Manual* (`dc-constraints.pdf`).

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

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

DC 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 we 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. Note that the `compile` command does not optimize across module boundaries. You can use the `set_flatten` command to enable inter-module optimization. For more information on the `compile` command consult the *Design Compiler User Guide* (`dc-user-guide.pdf`) or use `man compile` at the DC shell prompt. Run the following command and take a look at the output.

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

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, but for now you should just use the area numbers as a relative metric. The worst negative slack column shows how much room there is between the critical path in your design and the clock constraint. Larger negative slack values are worse since this means that your design is missing the desired clock frequency by a greater amount. Total negative slack is the sum of all negative slack across all endpoints in the design - if this is a large negative number it indicates that not only is the design not making timing, but it is possible that *many* paths are too slow. If the total negative slack is a small negative number, then this indicates that only a few paths are too slow. The design rule cost is 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. On the third through fifth iterations the tool is trying to increase performance and decrease the design rule cost while maintaining the same area. Once the synthesis is complete, you will get a the following warning about a very high-fanout net.

```
  Warning: Design 'smipsCore_synth' contains 1 high-fanout nets.
```

```
      ELAPSED            WORST NEG TOTAL NEG  DESIGN
       TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
     --------- --------- --------- --------- --------- -------------------------
       0:00:27   20274.9      0.00       0.0       0.0
       0:00:31   15195.4      3.22    2234.8       4.7
       0:00:34   15288.7      0.75     151.0       0.8
       0:00:40   15161.0      0.00       0.0       0.0
```

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

```
         A fanout number of 1000 will be used for delay
         calculations involving these nets. (TIM-134)
Net 'clk': 1034 load(s), 1 driver(s)
```

The synthesis tool is noting that the clock is driving 1034 gates. Normally this would be a serious problem, and it would require special steps to be taken during place+route so that the net is properly driven. However, this is the clock node and we already handle the clock specially by building a clock tree during place+route so there is no need for concern.

We can now use various commands to examine timing paths, display reports, and further optimize our design. Entering in these commands by hand can be tedious and error prone, plus doing so makes it difficult to reproduce a result. Thus we 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-xg-t> exit
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth/build
% cd ..
% rm -rf build
```

Automating Synthesis with TCL Scripts and Makefiles

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

- `Makefile` - Makefile for driving synthesis with the TCL scripts
- `synth.tcl` - Primary TCL script which contains the DC commands
- `libs.tcl` - TCL fragment which will setup various library variables
- `synth.sdc` - User specified constraints

First take a look at the `libs.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 `libs.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. We will take a closer look at it in a moment. Now examine the `synth.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 `synth.sdc` file contains various user specified constraints. This is where you constrain the clock period. We 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 we are more familiar with the various TCL scripts, we will see how to use the makefile to drive synthesis. Look inside the makefile and identify where the Verilog sources are defined. Notice that we 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 included non-synthesizable test harnesses modules. Also notice that we *must* identify the toplevel Verilog module in the design. We also specify several modules in the `dont_touch` make variable. Any modules which we 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 we are marking the dummy memories don't touch so that DC does not completely optimize away the buffer chain we 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.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% make new-build-dir
```

You should now see a new build directory named `build-<date>` where `<date>` represents the time and date. The `current` symlink always points to the most recent build directory. If you look inside the build directory, you will see the `libs.tcl`, `synth.tcl`, and `synth.sdc` 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` 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.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% cd current
% dc_shell-xg-t
dc_shell-xg-t> source libs.tcl
dc_shell-xg-t> analyze -library WORK -format verilog ${VERILOG_SRCS}
dc_shell-xg-t> elaborate ${VERILOG_TOPLEVEL} -architecture verilog -library WORK
dc_shell-xg-t> check_design
dc_shell-xg-t> source synth.sdc
dc_shell-xg-t> compile -map_effort medium -area_effort medium
dc_shell-xg-t> exit
% cd ..
```

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 our synthesis we can use the `synth` 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.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% make synth
```

You should see DC compiler start and then execute the commands located in the `synth.tcl` script. Once synthesis is finished try running `make synth` 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. Let's make a change to one of the TCL scripts. Edit `synth.sdc` and change the clock period constraint to 15 ns. Now use `make synth` 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-synth`.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% ls -l
build-2007-02-26_16-15
build-2007-02-26_16-25
build-2007-02-26_16-31
current -> build-2007-02-26_16-31
CVS
libs.tcl
Makefile
synth.sdc
synth.tcl
```

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. We 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 5 ns clock period constraint and the build directory which corresponds to a 15 ns clock period constraint.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% ln -s build-2007-02-26_16-25 build-5ns
% ln -s build-2007-02-26_16-31 build-15ns
```

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.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% rm -rf current
% make synth
```

Interpreting the Synthesized Gate-Level Netlist and Text Reports

In this section we will examine some of the output which our `synth.tcl` script generates. We will initially focus on the contents of the `build-5ns` build directory. The primary output from the synthesis scripts is the synthesized gate-level netlist which is contained in `synthesized.v`. Take a look at the gate-level netlist for the 5 ns clock constraint. Notice that the RTL module hierarchy is preserved in the gate-level netlist since we did not flatten any part of our 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 Tower 0.18 μm Standard Cell Library (`tsl-180nm-sc-databook.pdf`) to determine the function of the `mux02d1` standard cell. You should discover that this is a 2 input 1-bit mux cell. From the gate-level netlist we 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 four 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. We can compare this to the buffer insertion which occurs during place+route. After place+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 `synth.tcl` also generates several text reports. Reports usually have the `rpt` filename suffix. The following is a list of the synthesis reports.

- `synth_area.rpt` - Contains area information for each module instance
- `synth_resources.rpt` - Contains information on Design Ware components
- `synth_check_design.rpt` - Contains output from `check_design` command
- `synth_timing.rpt` - Contains critical timing paths
- `dc.log` - Log file of all output during DC run

In this section we will discuss the `synth_area.rpt` and the `synth_timing.rpt` reports. The next section will discuss the `synth_resources.rpt` report. The `synth_area.rpt` report contains area information for each module in the design. Figure 5 shows a fragment from `synth_area.rpt` for the SMIPSv1 unpipelined processor. We can use the `synth_area.rpt` report to gain insight into how various modules are being implemented. For example, we can use the area report in a similar fashion as the `synthesized.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.

We 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 we can see that register file consumes 90% of the total processor area. The area report reveals that the

```
module vcMux2_W32_3 ( in0, in1, sel, out );

  input  [31:0] in0;
  input  [31:0] in1;
  output [31:0] out;
  input         sel;

  wire   N0, n1, n2, n3, n4, n5, n6;
  assign N0 = sel;

  buffd1 U8 ( .I(N0), .Z(n4) );
  buffd1 U9 ( .I(N0), .Z(n5) );
  buffd1 U10 ( .I(N0), .Z(n6) );
  buffd1 U11 ( .I(N0), .Z(n3) );

  nr02d0 U1 ( .A1(n3), .A2(n1), .ZN(out[0]) );
  inv0d0 U2 ( .I(in0[0]), .ZN(n1) );
  nr02d0 U3 ( .A1(n3), .A2(n2), .ZN(out[1]) );
  inv0d0 U4 ( .I(in0[1]), .ZN(n2) );

  mx02d1 U5 ( .I0(in0[31]), .I1(in1[31]), .S(n6), .Z(out[31]) );
  mx02d1 U6 ( .I0(in0[29]), .I1(in1[29]), .S(n6), .Z(out[29]) );
  // ... 26 additional mx02d1 instantiations ...
  mx02d1 U37 ( .I0(in0[10]), .I1(in1[10]), .S(n4), .Z(out[10]) );
  mx02d1 U38 ( .I0(in0[9]), .I1(in1[9]), .S(n4), .Z(out[9]) );

endmodule

module vcMux2_W32_2 ( in0, in1, sel, out );

  input [31:0] in0;
  input [31:0] in1;
  output [31:0] out;
  input sel;

  wire   N0, n1, n2, n3, n4;
  assign N0 = sel;

  buffd1 U2 ( .I(N0), .Z(n1) );
  buffd1 U3 ( .I(N0), .Z(n2) );
  buffd1 U4 ( .I(N0), .Z(n3) );
  buffd1 U5 ( .I(N0), .Z(n4) );

  mx02d1 U1 ( .I0(in0[7]), .I1(in1[7]), .S(n1), .Z(out[7]) );
  mx02d1 U6 ( .I0(in0[1]), .I1(in1[1]), .S(n1), .Z(out[1]) );
  // ... 28 additional mx02d1 instantiations ...
  mx02d1 U35 ( .I0(in0[30]), .I1(in1[30]), .S(n4), .Z(out[30]) );
  mx02d1 U36 ( .I0(in0[31]), .I1(in1[31]), .S(n4), .Z(out[31]) );

endmodule
```

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

```
Design : smipsCore_synth/proc (smipsProc)
Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
smipsProcCtrl                   100.750000       1    100.750000  h, n
smipsProcDpath_pstr           12077.750000       1  12077.750000  h, n
-----------------------------------------------------------------------------
Total 2 references                                  12178.500000

Design : smipsCore_synth/proc/dpath (smipsProcDpath_pstr)
Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
buffd1             tsl18fs120_typ     1.000000       5     5.000000
smipsProcDpathRegfile         11074.250000       1  11074.250000  h, n
vcAdder_simple_W32              319.750000       1    319.750000  h
vcEQComparator_W32               96.000000       1     96.000000  h
vcInc_W32_INC00000004           113.500000       1    113.500000  h
vcMux2_W32_0                     68.250000       1     68.250000  h
vcMux2_W32_1                     68.750000       1     68.750000  h
vcMux2_W32_2                     68.000000       1     68.000000  h
vcMux2_W32_3                     67.000000       1     67.000000  h
vcRDFF_pf_W32_RESET_VALUE00001000   196.250000       1   196.250000 h, n
vcSignExtend_W_IN16_W_OUT32       1.000000       1      1.000000  h
-----------------------------------------------------------------------------
Total 11 references                                 12077.750000

Design : smipsCore_synth/proc/dpath/op0_mux (vcMux2_W32_3)
Reference          Library       Unit Area   Count    Total Area   Attributes
-----------------------------------------------------------------------------
buffd1             tsl18fs120_typ     1.000000       4     4.000000
inv0d0             tsl18fs120_typ     0.500000       2     1.000000
mx02d1             tsl18fs120_typ     2.000000      30    60.000000
nr02d0             tsl18fs120_typ     1.000000       2     2.000000
-----------------------------------------------------------------------------
Total 4 references                                     67.000000
```

Figure 5: Fragment from `synth_area.rpt`

register file is being implemented with approximately 1000 enable flip-flops and two large 32 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 6 illustrates a fragment of the timing report found in `synth_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

```
Point                                                         Incr   Path
----------------------------------------------------------------------------
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]/CP (dfnrq4)                     0.00 # 0.00 r
proc/dpath/pc_pf/q_np_reg[21]/Q (dfnrq4)                      0.25   0.25 f
proc/dpath/pc_pf/q_np[21] (vcRDFF_pf_W32_RESET_VALUE00001000)  0.00   0.25 f
proc/dpath/imemreq_bits_addr[21] (smipsProcDpath_pstr)        0.00   0.25 f
proc/imemreq_bits_addr[21] (smipsProc)                        0.00   0.25 f
dmem/imem_read_delay/row[0].bit[21].delay/Z (bufbdk)          0.16   0.41 f
...
dmem/imem_read_delay/row[3].bit[21].delay/Z (bufbdk)          0.16   0.87 f
proc/imemresp_bits_data[21] (smipsProc)                       0.00   0.87 f
proc/ctrl/imemresp_bits_data[21] (smipsProcCtrl)              0.00   0.87 f
proc/ctrl/rf_raddr0[0] (smipsProcCtrl)                        0.00   0.87 f
proc/dpath/rf_raddr0[0] (smipsProcDpath_pstr)                 0.00   0.87 f
proc/dpath/rfile/raddr0[0] (smipsProcDpathRegfile)            0.00   0.87 f
...
proc/dpath/rfile/rdata0[5] (smipsProcDpathRegfile)            0.00   1.92 f
proc/dpath/op1_mux/in1[5] (vcMux2_W32_2)                      0.00   1.92 f
proc/dpath/op1_mux/U11/Z (mx02d1)                             0.15   2.07 f
proc/dpath/op1_mux/out[5] (vcMux2_W32_2)                      0.00   2.07 f
proc/dpath/adder/in1[5] (vcAdder_simple_W32)                  0.00   2.07 f
proc/dpath/adder/add_29/B[5] (vcAdder_simple_W32_DW01_add_0)   0.00   2.07 f
...
proc/dpath/adder/add_29/SUM[31] (vcAdder_simple_W32_DW01_add_0) 0.00   3.71 f
proc/dpath/adder/out[31] (vcAdder_simple_W32)                 0.00   3.71 f
proc/dpath/dmemreq_bits_addr[31] (smipsProcDpath_pstr)        0.00   3.71 f
proc/dmemreq_bits_addr[31] (smipsProc)                        0.00   3.71 f
...
proc/dmemresp_bits_data[31] (smipsProc)                       0.00   4.45 f
proc/dpath/dmemresp_bits_data[31] (smipsProcDpath_pstr)       0.00   4.45 f
proc/dpath/wb_mux/in1[31] (vcMux2_W32_1)                      0.00   4.45 f
proc/dpath/wb_mux/U2/Z (mx02d2)                               0.16   4.61 f
proc/dpath/wb_mux/out[31] (vcMux2_W32_1)                      0.00   4.61 f
proc/dpath/rfile/wdata_p[31] (smipsProcDpathRegfile)          0.00   4.61 f
...
proc/dpath/rfile/registers_reg[10][31]/D (denrq1)             0.00   4.80 f
data arrival time                                                    4.80

clock ideal_clock1 (rise edge)                                5.00   5.00
clock network delay (ideal)                                   0.00   5.00
proc/dpath/rfile/registers_reg[10][31]/CP (denrq1)            0.00   5.00 r
library setup time                                           -0.18   4.82
data required time                                                   4.82
----------------------------------------------------------------------------
data required time                                                   4.82
data arrival time                                                   -4.80
----------------------------------------------------------------------------
slack (MET)                                                          0.01
```

Figure 6: Fragment from `synth_timing.rpt`

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. We 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 10 in the register file. The large buffers in the memory (the `bufbdk` cell in the `dmem` module) model the combinational delay through these memories. We can use the delay column to get a feel for how much each module contributes to the critical path: the combinational memories contribute about 0.6 ns; the register file read contributes about 1.1 ns; the adder contributes 1.7 ns; and the register file write requires 0.2 ns.

The critical path takes a total of `4.82`ns which is less than the `5`ns clock period constraint. Notice, however, that the final register file flip-flop has a setup time of 0.18 ns. So the critical path plus the setup time (`4.82`ns + `0.18`ns = `5`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* (`design-ware-quickref.pdf`). The components we will be using in the class are the Building Block IP described in Chapter 2.

The `synth_area.rpt` report can help us 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 (`/mit/6.375/doc/design-ware-datasheets/dw01_addpdf`). The data sheets contain information on the different component implementation types. For example, DC can use a ripple-carry adder, a carry-lookahead adder, or a parallel-prefix adder. The `synth_resources.rpt` report contains information on which implementation was chosen for each Design Ware component. Figure 7 shows a fragment from `synth_resources.rpt` which indicates that the adder uses a `pparch` implementation. From the `dw01_add` datasheet we know that the `pparch` implementation is a delay-optimized flexible parallel-prefix adder. Compare this to what is generated with the 15 ns clock constraint. Look at the `synth_resources.rpt` file in the `build-15ns` directory. Figure 8 shows that with the much slower clock period constraint, DC has chosen to use a simple ripple-carry adder (`rpl`). Although the ripple-carry adder is slower than the 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 included 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 $(SYNOPSYS)/dw/sim_ver +libext+.v+
```

We suggest only using direct instantiation as a last resort since it 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.

```
****************************************
Design : smipsCore_synth/proc/dpath/adder (vcAdder_simple_W32)

Resource Sharing Report
|          |              |            | Contained      |                      |
| Resource | Module       | Parameters | Resources      | Contained Operations |
===============================================================================
| r242     | DW01_add     | width=32   |                | add_29               |

Implementation Report
|                   |                  | Current          | Set              |
| Cell              | Module           | Implementation   | Implementation   |
=============================================================================
| add_29            | DW01_add         | pparch           |                  |
```

Figure 7: Fragment from `synth_resources.rpt` for 5 ns clock period

```
****************************************
Design : smipsCore_synth/proc/dpath/adder (vcAdder_simple_W32)

Resource Sharing Report
|          |              |            | Contained      |                      |
| Resource | Module       | Parameters | Resources      | Contained Operations |
===============================================================================
| r242     | DW01_add     | width=32   |                | add_29               |

Implementation Report
|                   |                  | Current          | Set              |
| Cell              | Module           | Implementation   | Implementation   |
=============================================================================
| add_29            | DW01_add         | rpl              |                  |
```

Figure 8: Fragment from `synth_resources.rpt` for 15 ns 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 we will use to analyze the synthesis results. You should avoid using the GUI to actually perform synthesis since we want to use scripts for this. To launch Design Vision and read in our synthesized design, move into the appropriate working directory and use the following commands.

```
% pwd
tut4/examples/smipsv1-1stage-v/build/dc-synth
% cd current
% design_vision-xg
design_vision-xg> source libs.tcl
design_vision-xg> read_file -format ddc synthesized.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 9 shows the schematic view for the datapath adder module with the 15 ns clock constraint. Notice the ripple-carry structure of the adder.

You can use Design Vision to examine various timing data. The *Schematic → Add Paths From/To* menu option will bring up a dialog box which you can use to examine a specific path. The default options will produce a schematic of the critical path. The *Timing → Paths Slack* menu option 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. Figure 10 shows an example of using these two features.

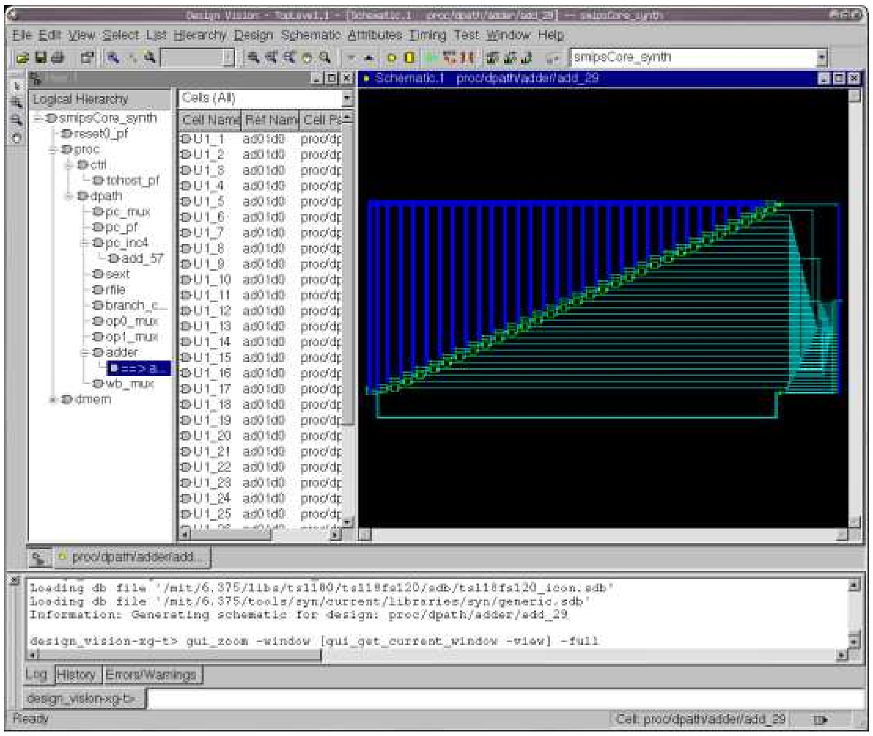
It is sometimes useful to examine the critical path through a single submodule. To do this, right click on the module in the hierarchy view and use the *Characterize* option. Check the timing, constraints, and connections boxes and click OK. Now choose the module from the drop down list box on the toolbar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

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

Review

The following sequence of commands will setup the 6.375 toolflow, checkout the SMIPSv1 processor example, and synthesize the design.

```
% add 6.375
% source /mit/6.375/setup.csh
% mkdir tut4
% cd tut4
% cvs checkout examples/smipsv1-1stage-v
% cd examples/smipsv1-1stage-v/build/dc-synth
% make
```

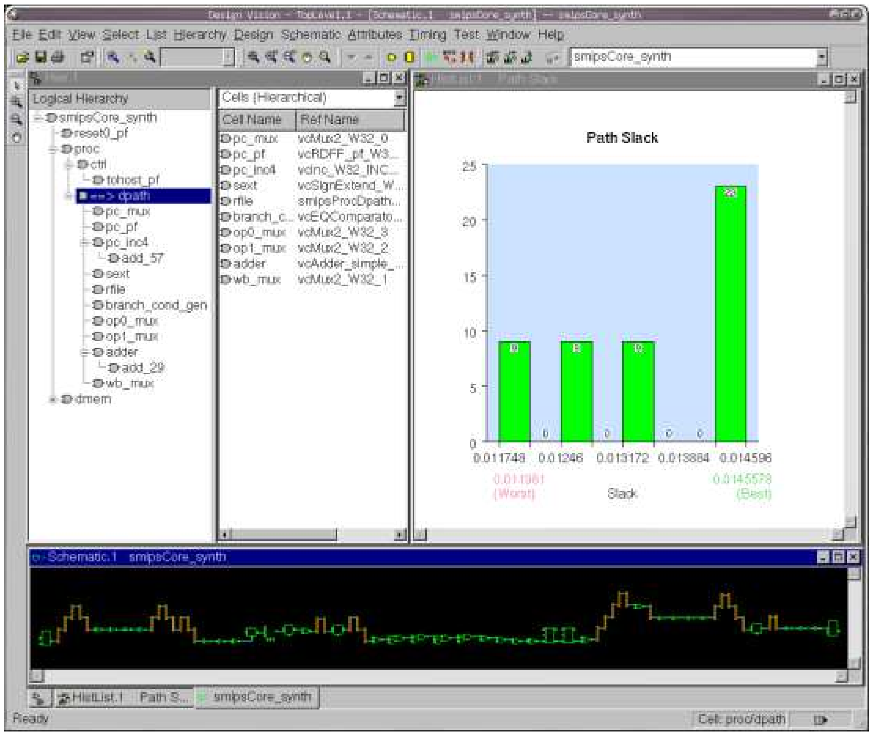


Cell Name	Ref Name	Cell Pa
U1_1	ad01d0	proc/dp
U1_2	ad01d0	proc/dp
U1_3	ad01d0	proc/dp
U1_4	ad01d0	proc/dp
U1_5	ad01d0	proc/dp
U1_6	ad01d0	proc/dp
U1_7	ad01d0	proc/dp
U1_8	ad01d0	proc/dp
U1_9	ad01d0	proc/dp
U1_10	ad01d0	proc/dp
U1_11	ad01d0	proc/dp
U1_12	ad01d0	proc/dp
U1_13	ad01d0	proc/dp
U1_14	ad01d0	proc/dp
U1_15	ad01d0	proc/dp
U1_16	ad01d0	proc/dp
U1_17	ad01d0	proc/dp
U1_18	ad01d0	proc/dp
U1_19	ad01d0	proc/dp
U1_20	ad01d0	proc/dp
U1_21	ad01d0	proc/dp
U1_22	ad01d0	proc/dp
U1_23	ad01d0	proc/dp
U1_24	ad01d0	proc/dp
U1_25	ad01d0	proc/dp

```
Loading db file '/mit/6.375/libs/tsl180/tsl18fs120/sdb/tsl18fs120_icon.sdb'
Loading db file '/mit/6.375/tools/syn/current/libraries/syn/generic.sdb'
Information: Generating schematic for design: proc/dpath/adder/add_29

design_vision-xg-t> gui_zoom -window [gui_get_current_window -view] -full
```

Figure 9: Screen shot of a schematic view in Design Vision



Cell Name	Ref Name
pc_mux	vcMux2_W32_0
pc_pf	vcRDFF_pf_W3...
pc_inc4	vcInc_W32_INC...
sext	vcSignExtend_W...
rfile	smipsProcDpath...
branch_c...	vcEQComparato...
op0_mux	vcMux2_W32_3
op1_mux	vcMux2_W32_2
adder	vcAdder_simple_...
wb_mux	vcMux2_W32_1

Figure 10: Screen shot of timing results in Design Vision