ASIC Implementation of a Two-Stage MIPS Processor

6.884 Laboratory 2
February 22, 2005 - Version 20050225

In the first lab assignment, you built and tested an RTL model of a two-stage pipelined MIPS processor. In the second lab assignment, you will be using various commercial EDA tools to synthesize, place, and route your design. After producing a preliminary ASIC implementation, you will attempt to optimize your design to increase performance and/or decrease area. The primary objective of this lab is to introduce you to the tools you will be using in your final projects, as well as to give you some intuition into how high-level hardware descriptions are transformed into layout.

The deliverables for this lab are (a) your optimized Verilog source and all of the scripts necessary to completely generate your ASIC implementation, and (b) a short one-page lab report (see Section 5 for details on exactly what you need to turn in). The lab assignment is due via CVS at the start of class on Monday, February 28.

Before starting this lab, it is recommended that you revisit the Verilog model you wrote in the first lab. Take some time to clean up your code, add comments, and enforce a consistent naming scheme. You will find as you work through this lab assignment that having a more extensive module hierarchy can be very advantageous; initially we will be preserving module boundaries throughout the toolflow which means that you will be able to obtain performance and area results for each module. It will be much more difficult to gain any intuition about the performance or area of a specific assign statement or always block within a module. Thus you might want to consider breaking your design into smaller pieces. For example, if your entire ALU datapath is in one module, you might want to create separate submodules for the adder/subtracter unit, shifter unit, and the logic unit. Unfortunately, preserving the module hierarchy throughout the toolflow means that the CAD tools will not be able to optimize across module boundaries. If you are concerned about this you can explicitly instruct the CAD tools to *flatten* a portion of the module hierarchy during the synthesis process. Flattening during synthesis is a much better approach than lumping large amounts of Verilog into a single module yourself.

Figure 1 illustrates the 6.884 ASIC toolflow we will be using for the second lab. You should already be familiar with the simulation path from the first lab. We will use Synopsys Design Compiler to *synthesize* the design. Synthesis is the process of transforming a higher-level behavioral or dataflow model into a lower gate-level model. For this lab assignment, Design Compiler will take your RTL model of the MIPS processor as input along with a description of the standard cell gate library, and it will produce a Verilog netlist of standard cell gates. Although the gate-level netlist is at a low-level functionally, it is still relatively abstract in terms of the spatial placement and physical connectivity between the gates. We will use Cadence Encounter to *place and route* the design. Placement is the process by which each standard cell is positioned on the chip, while routing involves wiring the cells together using the various metal layers. Notice that you will be receiving feedback on the performance and area of your design after both synthesis and place+route - the results from synthesis are less realistic but are generated relatively rapidly, while the results from place+route

are more realistic but require much more time to generate. Place+route for your two-stage MIPS processor will take on the order of 15 minutes, but for your projects it could take up to an hour.

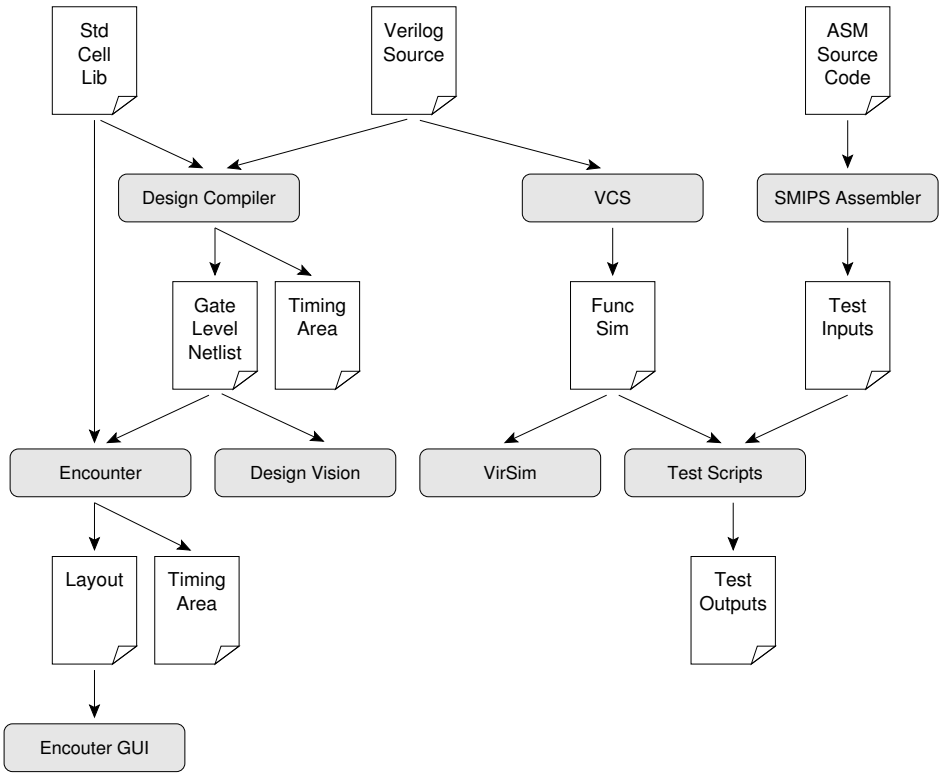

Figure 1: 6.884 Toolflow for Lab 1 and Lab 2

1 Setup

For the second lab assignment you will continue to work with your `mips2stage` CVS project. We recommend that you start with a fresh checkout for the second lab assignment. You can use the following CVS command to checkout your project.

```
% cvs checkout 2005-spring/<username>/mips2stage
```

You will need to update the makefiles and the config files in your project in order to be able to use the synthesis and place+route toolflow. We will be distributing harnesses for labs in the locker from now on so you will not be using the export command as you did in the first assignment. Instead copy the mips2stage harness tarball into a temporary directory and untar it. Then copy the various config, makefiles, and tests into your project as you see fit. You can get the tarball and untar it with the following commands.

```
% mkdir temp
% cd temp
% tar -xzvf /mit/6.884/lab-harnesses/mips2stage-harness.tgz
```

If you did not make any changes to the makefiles and the mips2stage.mk make fragment (besides listing your Verilog source files and assembly test files), then you might want to directly overwrite these files with the new versions using the following command.

```
% pwd
.../2005-spring/<username>/mips2stage
% tar --overwrite -xzvf /mit/6.884/lab-harnesses/mips2stage-harness.tgz
```

You will need to modify `config/mips2stage.mk` to correctly list your Verilog source files and any additional assembly test files. The new infrastructure includes makefile targets for synthesis and place+route as well as additional scripts that these tools will need. The rules for building tests are now make fragments in the config directory. It is important to note that the `mips2stage.mk` make fragment now separates the Verilog test harness from the Verilog RTL. This is because the Verilog test harness should not be used as input into the synthesis and place+route tools. Please list *all* of your Verilog source (not just the toplevel file) and also correctly identify the toplevel RTL module (which is most likely `mips_cpu`).

The `mips2stage.mk` make fragment lists five scripts which are used by the synthesis and place+route tools. The default `mips2stage.mk` make fragment uses the `config/template*` scripts, but you are free to point your `mips2stage.mk` to your own scripts if you want to customize them.

Before beginning the second lab, you should rerun your tests just to verify that your model is still functionally correct. You can do this with the following commands.

```
% pwd
.../2005-spring/<username>/mips2stage
% mkdir build
% cd build
% ../configure.pl ../config/mips2stage.mk
% make run-tests
```

We will be using the two-stage MIPS processor found in the examples directory of CVS throughout this document to illustrate various concepts.

2 Synthesizing the Two-Stage MIPS Processor

Synthesis is the process of transforming a higher-level behavioral or dataflow model into a lower gate-level model (see slides from Lecture 5 for more on synthesis). We will be using Synopsys Design Compiler to perform this transformation. We will be running Design Compiler in a subdirectory of your build directory to keep things organized so to start the Design Compiler shell use the following commands.

```
% pwd
.../2005-spring/<username>/mips2stage/build
% mkdir synth
% cd synth
% dc_shell-xg-t
Initializing...
dc_shell-xg-t>
```

You are left at the Design Compiler 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>` on the `dc_shell-xg-t` prompt. We will now execute some commands to setup your environment.

```
dc_shell-xg-t> lappend search_path ../../verilog
dc_shell-xg-t> define_design_lib WORK -path "work"
dc_shell-xg-t> set link_library \
  [list /mit/6.884/libs/tsmc/130/lib/db/tcb013ghpwc.db]
dc_shell-xg-t> set target_library \
  [list /mit/6.884/libs/tsmc/130/lib/db/tcb013ghpwc.db]
```

These commands point to your Verilog source directory, create a Synopsys design directory, and point to the standard libraries we will be using for this class. 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 constructs. Try executing the commands as follows.

```
dc_shell-xg-t> analyze -library WORK -format verilog { ../../verilog/mips_cpu.v }
dc_shell-xg-t> elaborate mips_cpu -architecture verilog -library WORK
```

Obviously, you might need to use a different filename depending on your exact naming scheme. You may get some errors if the tool has trouble synthesizing your design. You may have used some Verilog constructs which are not synthesizable and thus you will need to make some changes. Please make a note of what changes were required for your lab report. You will probably also get some warnings. Eliminating all of the warnings is not necessary, but they can reveal common mistakes so review them carefully. Before you can synthesize your design, you must specify some constraints; most importantly you must tell the tool what the target clock period is. The following commands tell the tool that the pin named `clk` is the clock and that your desired clock period is 7 nanoseconds. If you named your clock something different than `clk` you will need to use the correct name. You should 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 the tools will have no trouble but you will get a very conservative implementation.

```
dc_shell-xg-t> create_clock clk -name ideal_clock1 -period 7
dc_shell-xg-t> compile
```

It will take a few minutes as Design Compiler performs the synthesis and then it will display some output similar to what is shown in Figure 2. Each line is an optimization pass. The area column

```
 ELAPSED            WORST NEG TOTAL NEG  DESIGN
  TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
--------- --------- --------- --------- --------- -------------------------
  0:01:00  103628.0      0.00       0.0       2.1
  0:01:07   76941.4      1.15     190.4       1.3
  0:01:12   77214.7      0.76      23.5       1.6
  0:01:22   77360.7      0.35      24.6       1.1
  0:01:27   77499.9      0.00       0.0       0.0
```

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

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 edge. A positive number means your design is faster than the constrained clock period by that amount, while a negative number means that your design did not meet timing and is slower than the constrained clock period by that 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; these violations can be caused by gates which have too large of a fanout (larger numbers are worse). Figure 2 shows that on the first iteration, the tool makes timing but at a high area cost, so on the second iteration it optimizes area. On the third through fifth iterations the tool is trying to further reduce the area and design rule cost while still making timing. There is no point in synthesizing an implementation which is faster than the constrained clock period, so the tool trades some performance for decreased area and design rule cost. You will probably get a warning similar to the one shown below about a very high-fanout net.

```
Warning: Design 'mips_cpu' contains 1 high-fanout nets.
         A fanout number of 1000 will be used for delay
         calculations involving these nets. (TIM-134)
Net 'clk': 1097 load(s), 1 driver(s)
```

The synthesis tool is noting that the clock is driving 1097 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 driver and H-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. We will use the `template.sdc`, `template-syn.setup`, and `template-syn.tcl` scripts located in the `config` directory as templates for the scripts we feed into Design Compiler. The makefile takes care of substituting the name of your toplevel module,

Verilog source, and other options into these templates and then copying then into the Design Compiler working directory. You should feel free to modify these scripts, but do so in the `config` directory, not in the build directory. Remember that everything in the build directory should be able to be generated - even the various tool scripts. If you look in the templates you will see `__NAME__` style tokens - these are replaced by the makefile so avoid changing them. Spend a few minutes taking a look through the scipts. The `template.sdc` script is of particular importance, since this is where you specify various timing constraints for your design. The `create_clock` line sets a constraint on the clock period and Design Compiler will do its best to produce a design which meets this constraint. To use the makefile to perform a synthesis run first exit the Design Compiler shell with the `exit` command; then return to the build directory and use the following commands (we use a `make clean` to remove the synth directory you were previously working in).

```
% pwd
.../2005-spring/<username>/mips2stage/build
% make clean
% make synth
```

When the synthesis is done, it will write the final synthesized Verilog to a file called `synthesized.v` in the `synth` directory. Take a look through this file to get a feel for what a gate-level netlist looks like. The synthesis script will also write several reports in the `synth` directory which can be used to learn more about the area and performance of your synthesized design. These files include `dc.log`, `synth_area.rpt`, and `synth_timing.rpt`. The `dc.log` contains the same output you saw on the console and there can be some valuable information in it on what types of structures were inferred from the Verilog source. Figure 3 illustrates a fragment from the `synth_area.rpt` report. The report shows how much area is required for each of the hierarchical instances, and it also shows which standard cells were used to actually implement a given module. For example, we can see that the `branch_tests` module was implemented using xor and or gates. The modules shown in all capitals are the standard cells and more information about these gates can be found in the standard cell databook located at `/mit/6.884/doc/tsmc-130nm-sc-databook.pdf`. Use the databook to determine the function of the the `ND2XD2` cell and the difference between the four types of xor gates. The Verilog source for the `branch_tests` module is shown below. Can you guess what logic structure the synthesizer built to implement the module? This is a good example of how a more extensive module hierarchy can help give you insight into the synthesizer results.

```
module branch_tests ( input [31:0] in0, in1, output beq, bsign );
  assign beq   = (in0 == in1);
  assign bsign = in0[31];
endmodule
```

Let's look closer at the area breakdown for the toplevel module. Notice that the execution unit accounts 86% of the total area. The area breakdown for the execution unit (not shown in the figure) reveals that the register file alone accounts for almost 60% of the total area and that most of this area is due to 992 D flip-flops (31 registers which are each 32 bits wide) and two large 32 input muxes. The flip-flops include enable signals for the write port and the two muxes implement the two 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*

```
****************************************
Report : reference
Design : mips_cpu
Version: V-2004.06-SP2
Date   : Mon Feb 21 20:54:02 2005
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
--------------------------------------------------------------------------------
BUFFD0             tcb013ghpwc     5.092200      1      5.092200
DFQD2              tcb013ghpwc    27.158400      1     27.158400  n
control                          916.596130       1    916.596130  h
cp0                             2430.677002       1   2430.677002  h, n
exec                          108644.335938       1 108644.335938  h, n
fetch                          13674.268555       1  13674.268555  h, n
--------------------------------------------------------------------------------
Total 6 references                                  125698.125000

****************************************
Report : reference
Design : mips_cpu/exec_unit/btests (branch_tests)
Version: V-2004.06-SP2
Date   : Mon Feb 21 20:54:02 2005
****************************************

Reference          Library       Unit Area   Count    Total Area   Attributes
--------------------------------------------------------------------------------
BUFFD0             tcb013ghpwc     5.092200     18    91.659596
NR2XD2             tcb013ghpwc    15.276600      1    15.276600
OR4D1              tcb013ghpwc    11.881800     10   118.817997
XOR2D0             tcb013ghpwc    13.579200     24   325.900795
XOR2D1             tcb013ghpwc    13.579200      6    81.475199
XOR2D2             tcb013ghpwc    15.276600      1    15.276600
XOR2D4             tcb013ghpwc    28.855801      1    28.855801
--------------------------------------------------------------------------------
Total 7 references                                     677.262573
```

Figure 3: Example fragment from the `synth_area.rpt`

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. Later in the course we will have a simple memory generator for you to use, but for now simply begin to appreciate the negative impacts of a synthesized register file.

Figure 4 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

```
Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock ideal_clock1 (rise edge)                           0.00       0.00
clock network delay (ideal)                              0.00       0.00
exec_unit/ir_reg[26]/CP (DFQD2)                          0.00 #     0.00 r
exec_unit/ir_reg[26]/Q (DFQD2)                           0.27       0.27 f
exec_unit/ir[26] (exec)                                  0.00       0.27 f
control_unit/ir[26] (control)                            0.00       0.27 f
...
control_unit/cs[9] (control)                             0.00       1.37 f
exec_unit/csig[9] (exec)                                 0.00       1.37 f
exec_unit/alu_b_mux/sel[1] (mux4_width32_1)              0.00       1.37 f
...
exec_unit/alu_b_mux/out[8] (mux4_width32_1)              0.00       2.37 r
exec_unit/addsub/alu_b[8] (alu_addsub)                   0.00       2.37 r
...
exec_unit/addsub/result[31] (alu_addsub)                 0.00       5.27 r
exec_unit/wb_mux/in1[31] (mux8_width32)                  0.00       5.27 r
...
exec_unit/wb_mux/out[31] (mux8_width32)                  0.00       5.79 r
exec_unit/rfile/wd[31] (regfile)                         0.00       5.79 r
exec_unit/rfile/registers_reg[0][31]/D (EDFQD4)          0.00       5.79 r
data arrival time                                                   5.79

clock ideal_clock1 (rise edge)                           7.00       7.00
clock network delay (ideal)                              0.00       7.00
clock uncertainty                                       -1.00       6.00
exec_unit/rfile/registers_reg[0][31]/CP (EDFQD4)         0.00       6.00 r
library setup time                                      -0.18       5.82
data required time                                                  5.82
-----------------------------------------------------------------------------
data required time                                                  5.82
data arrival time                                                  -5.79
-----------------------------------------------------------------------------
slack (MET)                                                         0.03
```

Figure 4: Example fragment from the `synth_timing.rpt`

(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 I have cut out many of the nodes internal to the higher level modules. We can see that the critical path starts at bit 26 of the instruction register, goes through the combinational control logic, through the mux select of one of the alu input muxes, through the `addsub` module, through the writeback mux, and finally ends at the register file. The last column lists the cumulative delay to that node, while the middle column shows the incremental delay. We can see that the control logic contributes almost 1.1 ns of delay, the alu input mux contributes 1 ns of delay, the `addsub` module contributes 2.9 ns of delay, and finally the writeback mux contributes 0.52 ns of delay. The critical path takes a total of `5.79`ns which is

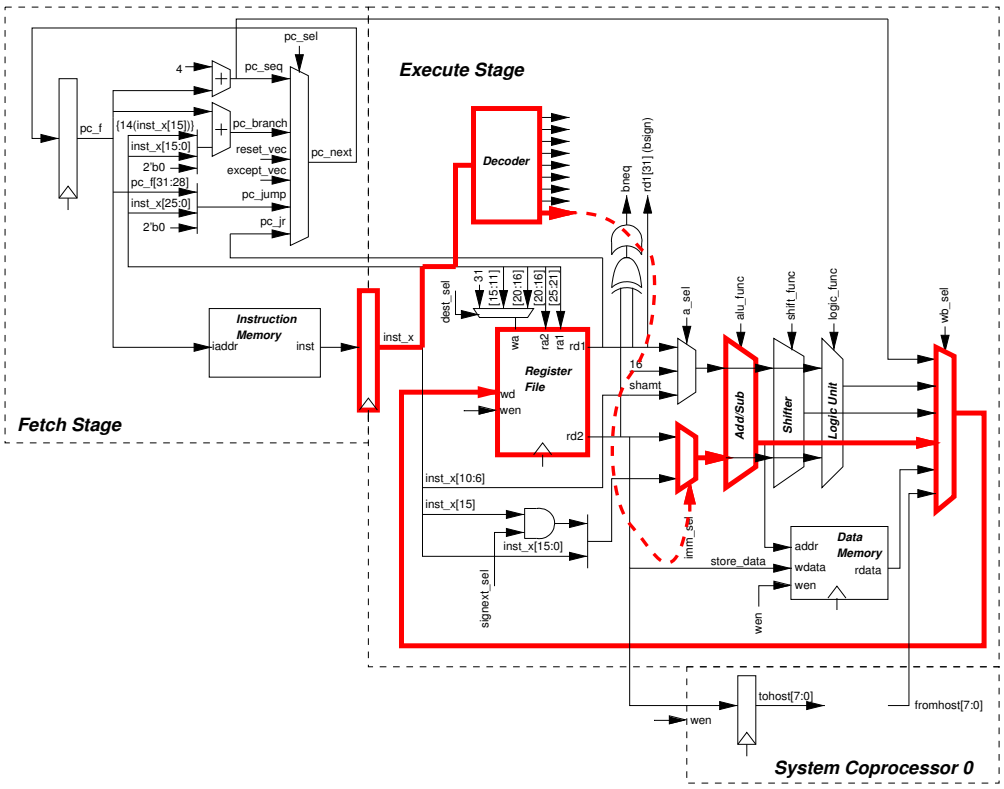

Figure 5: Critical path through MIPS Processor

plenty fast to meet the 7ns clock period constraint. Notice, however, that there is a nanosecond of clock uncertainty. This was specified in the `template-syn.tcl` script so that the synthesis tool would be forced to generate a more conservative implementation. We do this because the timing analysis used by Design Compiler ignores wire delay, and thus after place+route the critical path will definitely get slower. The nanosecond of uncertainty helps give us some extra slack to make up for this wire delay. Also notice that it is not enough for the critical path to be faster than 6 ns - the register file's setup time reduces the effective clock period by another 0.18 ns (see the slides from Lecture 6 for more on clocking). The final line of the report indicates that the critical path makes timing with 0.03 ns to spare. Figure 5 illustrates the critical path. You are free to try for a more aggressive design by reducing the target clock period specified in `template.sdc` or the clock uncertainty specified in `template-syn.tcl`. It is important to note that for this lab assignment there is essentially no delay through the memories. Furthermore, Design Compiler does not know that the memory address output is even connected to the memory data input and thus the static timing analysis on these paths is relatively useless. Later in the course, when we integrate a memory generator, we will be better able to model these paths.

You will want to save `synth_area.rpt`, `synth_timing.rpt`, and `dc.log` before continuing on with the lab. These files will be useful when writing your lab report.

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 move into the Design Compiler working directory and use the following command.

```
% pwd
.../2005-spring/<username>/mips2stage/build/synth
% design_vision
```

Load your design with the *File* → *Read* menu option and select the `synthesized.db` file. 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 6 shows the schematic view for the `branch_tests` module. The synthesizer has used 32 xor gates and a tree of or/nor gates to create a comparator for the *bneq* control signal. The tool has chosen xor gates with different drive strengths to help optimize performance through the logic (see slides from Lecture 3 for more information on how gate sizing impacts performance).

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 7 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 Paths* will provide information on the critical path through that submodule given the constraints of the submodule within the overall design's context.

Design Compiler and Design Vision are very sophisticated tools and the only way to become competent with using them is to make extensive use of the relevant documentation. The following list identifies what documentation is available in the course locker. Also remember that the standard cell databook is a valuable resource.

- `dc-user-guide.pdf` - Design Compiler User Guide
- `presto-HDL-compiler.pdf` - Guide for the Verilog Complier used by DC
- `dc-quick-reference.pdf` - Design Compiler Quick Reference
- `dc-command-line-guide.pdf` - Design Compiler Command Line Ref
- `dc-constraints.pdf` - Design Compiler Constraints and Timing Ref
- `dc-opt-timing-analysis.pdf` - Design Compiler Optimization and Timing Analysis Ref
- `dv-tutorial.pdf` - Design Vision Tutorial
- `dv-user-guide.pdf` - Design Vision User Guide

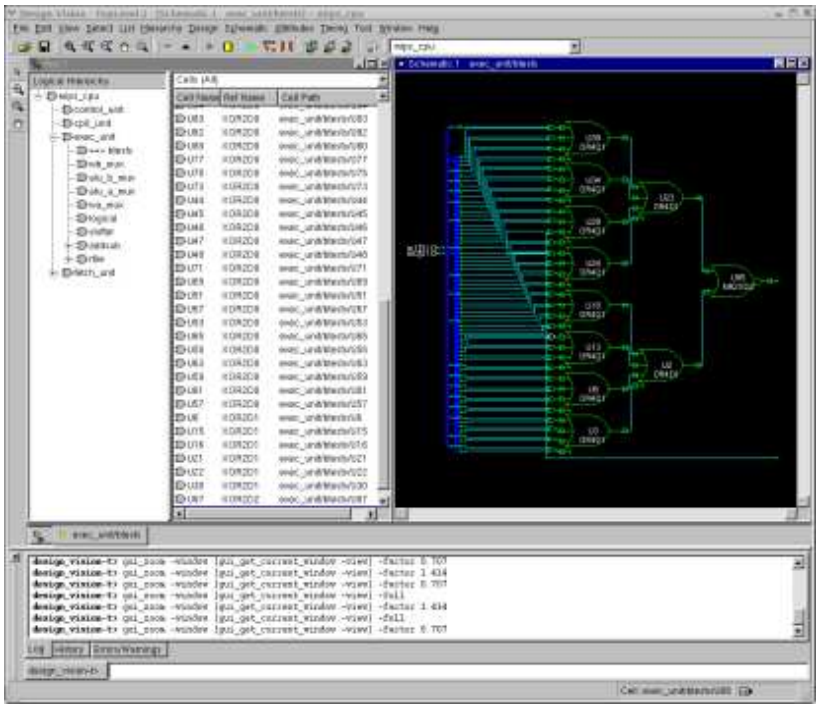

Figure 6: Screen shot of a schematic view in Design Vision

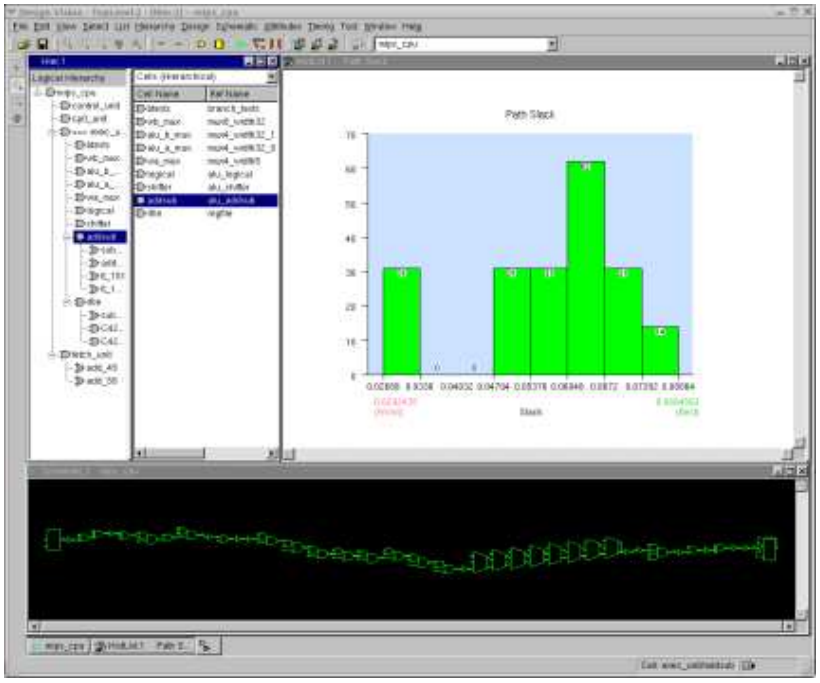

Figure 7: Screen shot of a timing results in Design Vision

3 Placing and Routing the Two-Stage MIPS Processor

This section describes how to use Cadence Encounter to *place and route* the design you synthesized in the previous section. As with synthesis, we will run Encounter in a subdirectory of the build directory to keep things organized. Like Design Compiler, Encounter uses its own shell and a user is able to execute commands directly from that shell. For now we will use the makefile and various scripts to control Encounter, but it is important for you to realize that you can always work directly from the Encounter shell. For example, to find out what a certain Encounter command does you can use the following commands.

```
% pwd
.../2005-spring/<username>/mips2stage/build
% mkdir pr
% cd pr
% encounter -nowin
encounter 1> man reportGateCount
...
% exit
```

The makefile will take care of copying the `template-pr.setup` and `template-pr.tcl` scripts from the `config` directory into the appropriate Encounter working directory. You can edit these templates to control what Encounter does. Encounter also references the `template.sdc` constraint file to load in the design constraints. It is important to use the same constraint file for both synthesis and place+route (the makefile ensures this). To place+route your design simply use the following command.

```
% pwd
.../2005-spring/<username>/mips2stage/build
% make pr
```

Place+route will probably take significantly longer than synthesis. When Encounter is finished you can see the final results by examining the `pr_area.rpt`, `pr_timing_slacks.rpt`, and `pr_timing_violations.rpt` reports located in the `pr` subdirectory. Compare the results from after synthesis to those from after place+route. The area might change due to further logic optimizations,

```
# Analysis mode: -setup -skew -caseAnalysis -async -noClkSrcPath
# reportSlacks -outfile pr_timing_slacks.rpt
# Format: clock  timeReq  slackR/slackF  setupR/setupF  instName/pinName   # cycle(s)
#
ideal_clock1(R) 7.000 -0.096/0.408 0.080/0.067 fetch_unit/pc_reg[16]/D 1
ideal_clock1(R) 7.000 -0.091/0.171 0.171/0.158 fetch_unit/pc_reg[28]/D 1
ideal_clock1(R) 7.000 -0.046/0.028 0.164/0.167 exec_unit/rfile/registers_reg[14][31]/D 1
ideal_clock1(R) 7.000 -0.045/0.028 0.164/0.167 exec_unit/rfile/registers_reg[1][31]/D  1
```

Figure 8: Example fragment from the `synth_timing.rpt`

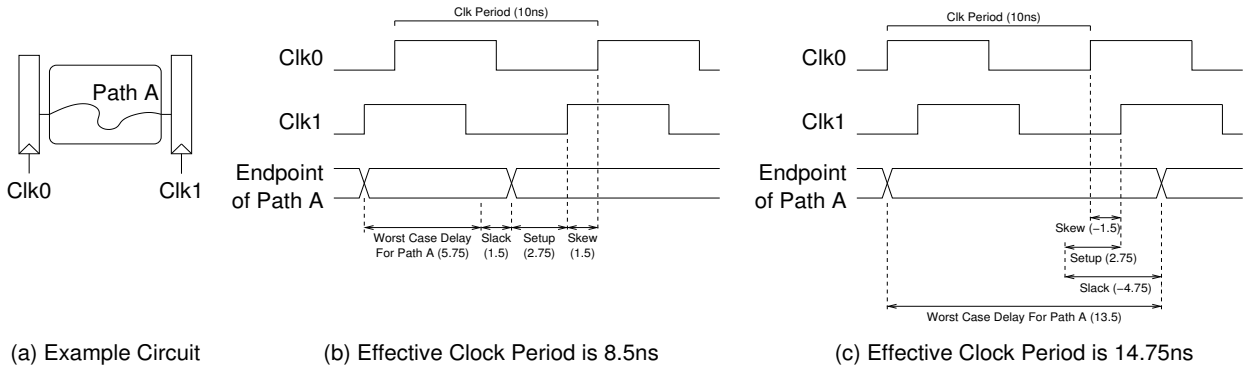

Figure 9: Determining your hardware's effective clock period

and the timing might change due to the influence of wire delay. Encounter also inserts buffers into your design to optimize drive strengths, and these buffers can increase delay and area. The area for the `mips_cpu` module listed in `pr_area.rpt` is the final area of your design and is the one you should provide in your lab report. The `pr_timing_slacks.rpt` report lists the slack for various endpoints, but it does not show the actual paths. Take a look at this report to see if your design met timing - just because it met timing after synthesis does not mean it will make timing after place+route! Figure 8 shows the slack report corresponding to the design synthesized in the previous section. The first column is the start of the path (i.e. the rising clock edge) and the final column is the end of the path. The second column is the constraint; in this case the only constraint is the clock period. The third column shows the slack for that path for both the rising and falling edge. A positive number means that the path made timing and a negative number means that the corresponding path did *not* make timing by the listed amount in nanoseconds. So in Figure 8 you can see that the design did not make timing by 0.096 ns on a path which ends at the PC. The `pr_timing_violations.rpt` report provides more information about the exact paths which are failing timing and it is similar in format to the `synth_timing.rpt` timing report we looked at earlier. The sum column lists the cumulative delay to that point in the path, while the delta column lists the incremental change in delay between points. In this case examining the `pr_timing_violations.rpt` report shows that the critical path is now from the IR register, through the register file, through the branch tests, through the control unit, and back to the fetch unit. Notice that the critical path is different from the one determined after synthesis! This is for two reasons: (a) the results after synthesis do not take into account wire delay and (b) Encounter does additional optimizations and buffer insertion.

Even though this design did not make the 7 ns clock period constraint it is still a valid piece of hardware which will operate correctly with some clock period (it is just slower than 7 ns). Similarly, a design which makes the timing constraint but does so with a positive slack can run *faster* than the constrained clock period. For this lab we are more concerned about the *effective* clock period of your design as opposed to the clock constraint you set before synthesis. The effective clock period is simply the clock period constraint *minus the worst slack* ($T_{clk} - T_{slack}$). `pr_timing_slacks.rpt` and `pr_timing_violations.rpt` are both sorted by slack so that the path with the worst slack is listed first. To determine the effective clock period for your design simply choose the smaller of the rising and falling edge slacks. Figure 9 illustrates two examples: one with positive slack and one with negative slack. For the initial synthesis and place+route of the example two-stage mips core

we see from Figure 8 that the worse case slack is 7.096 ns, and thus the effective clock period is $(7 - (-0.096)) = 7.096$ or approximately 140.9 MHz. This is the performance number you would provide in your lab report. Note that just because the design did not make timing at 7 ns, this does not mean it cannot go faster. If we set the clock period constraint to 6 ns it might result in a design with an effective clock period of 6.2 ns. Lower clock period constraints force the tools to work harder and they may or may not do better. You should experiment with this very important parameter.

We can use the Encounter GUI to view the design after place and route. To use the Encounter GUI move into the place+route working directory and use the following commands. You should open your design using the Encounter shell prompt and not actually through the GUI.

```
% pwd
.../2005-spring/<username>/mips2stage/build/pr
% encounter
encounter 1> restoreDesign placedrouted.dat mips_cpu
```

Encounter will take a minute or so to load the standard cell libraries and then it will show you your chip. Figure 10 shows the chip for the example two-stage mips pipeline. The large metal regions around the edge of your design make up the power/ground ring. You can use the color pallet to select which layers are displayed. There are several pallets - the small thin button under *All Colors* switches between two pallets and just clicking the *All Colors* button will bring up a dialog box with all of the various layers. Use *Control-R* to redraw the screen after changing which layers are visible. For example, Figure 11 shows a close up of some of the lower layers of metal for just a few cells on the chip. Each standard cell is a fixed height and a variable width; the routing between cells happens on the metal layers above the standard cells (see the slides from Lecture 1 for more on ASIC design using standard cells).

Use the *Edit* → *Select By Name* menu option to highlight cells or nets with a given name. For example, try searching for nets with the name `clk*` to see the clock tree and try searching for instances with the name `FILL*` to see the filler cells. Filler cells are cells which contain no logic and are just there to wire up power, ground, and the wells. Too many filler cells is an inefficient use of area and means Encounter is having trouble performing place+route.

Click on the middle of the three view buttons found in the lower lefthand portion of the GUI to see how modules map to the chip. Use *Tools* → *Design Browser* to bring up a module hierarchy browser. If you choose a submodule of the toplevel module then it will be highlighted on the chip display. If you click on the module in the chip display and then press *Shift-G* you will go down one level in the hierarchy. You can now select modules in the design browser to highlight them. Figure 12 shows the register file in red (the large shaded region at the bottom of the chip). As expected it accounts for over half of the chip area.

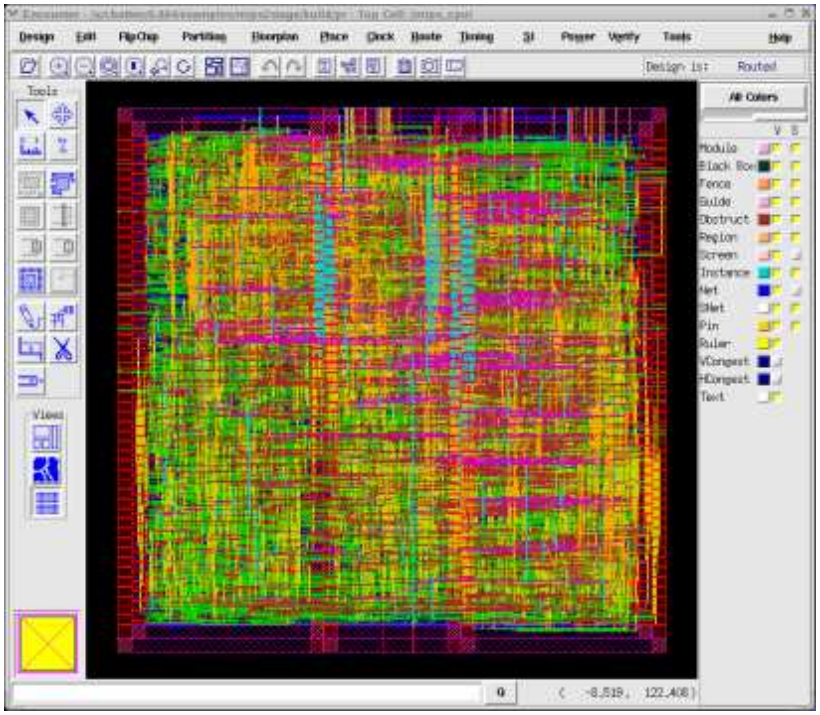

Figure 10: Screen shot of final chip in Encounter GUI

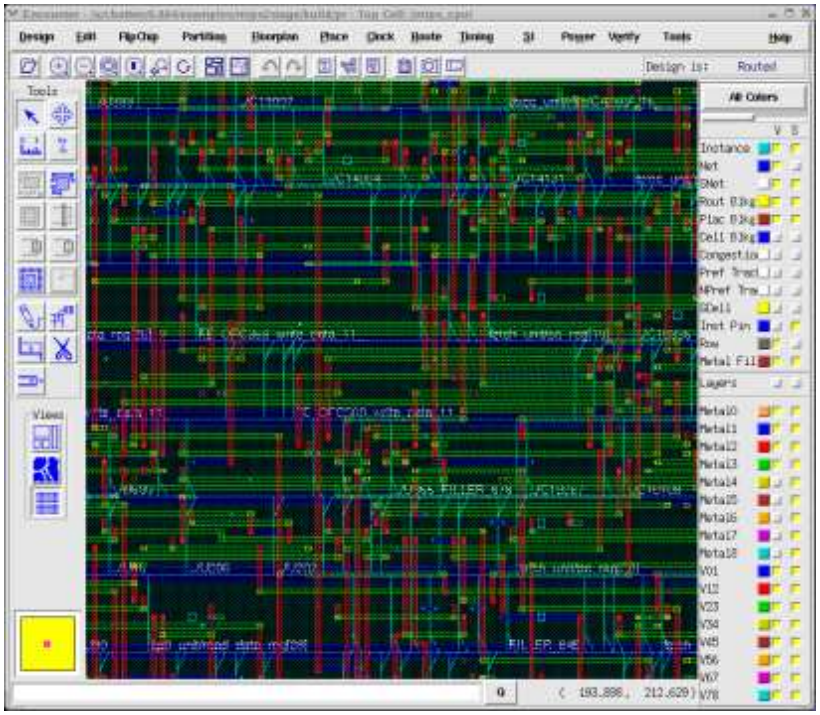

Figure 11: Closeup of standard cells and routing

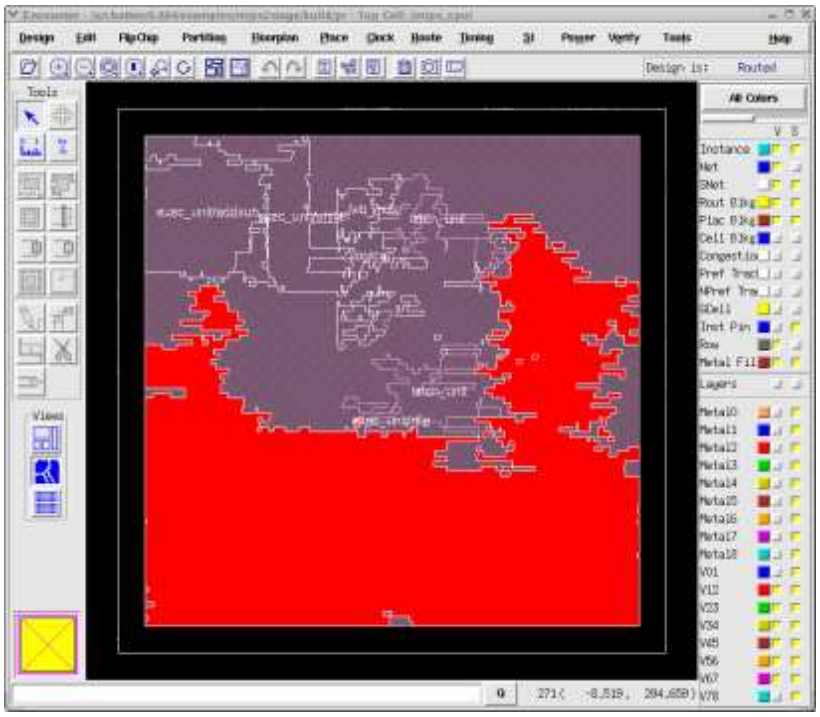

Figure 12: Register file module is highlighted in red

Encounter is a very sophisticated tool and as with Design Compiler, the only way to become competent with using it is to spend some time looking over the relevant documentation. The following list identifies what documentation is available in the course locker.

- `encounter-user-guide.pdf` - Encounter User Guide
- `encounter-command-line-guide.pdf` - Encounter Text Command Reference
- `encounter-menu-ref.pdf` - Encounter GUI Reference

You are now done with the first part of the lab assignment. You should prepare a short write up (half page) as a text document in your project directory. This writeup should discuss any changes you had to make to your original Verilog to get it to synthesize, plus any insights you have into your initial results. Create a new subdirectory called `mips2stage/lab2writeup` and place your writeup in this directory as a text document called `writeup.txt`. Also copy the following files into this directory and rename them as indicated. Be sure to use CVS to add and commit the new directory and its contents. The `make save-results` command can be a useful way to quickly save some results after working on a design.

- `synth_area.rpt` → `initial_synth_area.rpt`
- `synth_timing.rpt` → `initial_synth_timing.rpt`
- `dc.log` → `initial_dc.log`
- `pr_area.rpt` → `initial_pr_area.rpt`
- `pr_timing_slacks.rpt` → `initial_pr_timing_slacks.rpt`

- `pr_timing_violations.rpt` → `initial_pr_timing_violations.rpt`

We suggest that you create a CVS tag for your initial synthesizable design now that you have pushed it completely through the toolflow for the first time. You should only create tags on *clean checkouts*. A clean checkout is when you use `cvs checkout` in an empty temporary directory. When you tag a file it leaves a sticky bit on that file which can complicate later modifications. So the best approach to tagging your project is as follows:

1. Make sure you add all the appropriate files and commit your changes
   ```
   % pwd
   .../2005-spring/<username>/mips2stage
   % cvs update
   % cvs commit
   ```
2. Create a new junk directory somewhere
   ```
   % cd
   % mkdir junk
   ```
3. Checkout your project in the junk directory
   ```
   % cd junk
   % cvs checkout 2005-spring/<username>/mips2stage
   ```
4. Verify that your project builds correctly
   ```
   % mkdir build
   % cd build
   % ../configure.pl ../config/mips2stage.mk
   % make run-tests
   % make pr
   ```
5. Add the CVS tag
   ```
   % cd ..
   % pwd
   .../junk/2005-spring/<username>/mips2stage
   % cvs tag lab2-initial .
   ```
6. Delete the junk directory
   ```
   % cd
   % rm -rf junk
   ```

Now you can always recreate this initial synthesis and place+route using the following CVS commands in a fresh working directory.

```
% mkdir temp
% cvs checkout -r lab2-initial 2005-spring/<username>/mips2stage
% cd 2005-spring/<username>/mips2stage
% mkdir build
% cd build
% ../configure.pl ../config/mips2stage.mk
% make pr
```

4 Design Space Exploration

In the second part of the lab you will try and increase the performance or decrease the area of your initial design. There are many techniques you can use to do this and a few examples will be discussed in this section. You should feel free to try both Verilog level optimizations as well as using the tools more effectively.

4.1 Control Logic Optimization

As a first optimization we will add don't cares to our control logic to enable the synthesizer to perform more aggressive optimizations. The results reported in the first part of this document used zeros instead of don't cares. After changing the Verilog, we rerun our tests to verify that our design is still functionally correct. This is an essential step after *any* optimization; after all, an optimized but incorrect design is of no use. We can rerun the tests quite simply with the `make run-tests` command. We then rerun Design Compiler to see the impact on performance and area. We observe that the area of the control module has been reduced to 617 for an area savings of 33%. Furthermore, we now note that the control logic is no longer on the critical path, and instead the critical path goes from the register file, through the alu input mux, through the `addsub` module, through the writeback mux, and back to the register file. Notice that we evaluated this optimization before place+route - this is a standard approach since place+route is relatively time consuming. Just remember to occasionally push the design all the way to layout to verify that you are actually making progress. For the control logic optimization, after place+route we see a comparable decrease in area, the control logic is no longer on the critical path, and the clock period is reduced from 7.096 ns to 7.041 ns.

4.2 Multiplexer Optimization

It is advisable to examine the `synth_area.rpt` report to see how the synthesizer is actually implementing you design. It can sometimes reveal situations where the synthesizer is unable to infer what you intended. As an example let's look at the alu input muxes and the write back mux. The original Verilog and a fragment from the resulting `synth_area.rpt` is shown in Figure 13 (the eight input multiplexer shows similar results). If you consult the standard cell databook, you will see that Design Compiler is not using the MUX standard cells and is instead synthesizing a mux from random logic. It may or may not be doing this intentionally, but it is usually a good idea to use the MUX cells for your multiplexers. If we take a look at the HDL Verilog Compiler Reference (`presto-HDL-compiler.pdf`) we can learn a little more about how Design Compiler infers muxes. Based on this information we recode the muxes as shown in Figure 14. We also rerun our tests to make sure the design is still functionally correct. The alternative design is actually slightly larger. The muxes seem to be slightly faster since the effective clock period has decreased to 7.006 ns.

```
module mux4 #( parameter width = 0 )
(
  input  [width-1:0] in0, in1, in2, in3,
  input  [1:0] sel,
  output [width-1:0] out
);

  assign out
    = ( sel == 2'd0 ) ? in0 :
      ( sel == 2'd1 ) ? in1 :
      ( sel == 2'd2 ) ? in2 :
      ( sel == 2'd3 ) ? in3 : {width{1'bx}};

endmodule
```

```
 Reference      Area
--------------------
 BUFFD16      162.95
 CKBD16        40.73
 CKND3         11.88
 CKND4         15.27
 CKND6         44.13
 IND2D4        56.01
 INVD1          6.78
 INVD2        208.78
 ND2D2         33.94
 NR2XD1       195.20
 NR2XD4        96.75
 NR2XD8        61.10
 OAI22D0       16.97
 OAI22D1       20.36
--------------------
 Total        970.91
```

Figure 13: Verilog and fragment from the `synth_area.rpt` for 4 input mux

```
module mux4 #( parameter width = 0 )
(
  input  [width-1:0] in0, in1, in2, in3,
  input  [1:0] sel,
  output [width-1:0] out
);

  reg [width-1:0] out;
  always @(*)
  begin
    case ( sel ) // synopsys infer_mux
      2'd0 : out <= in0;
      2'd1 : out <= in1;
      2'd2 : out <= in2;
      2'd3 : out <= in3;
      default : out <= {width{1'bx}};
    endcase
  end

endmodule
```

```
 Reference     Area
-------------------
 BUFFD16      81.47
 CKND0         3.39
 CKND2D0      10.18
 INR3D0       61.10
 MUX2ND0      11.88
 MUX4D1      806.26
-------------------
 Total       974.30
```

Figure 14: Verilog and fragment from the `synth_area.rpt` for alternative 4 input mux

4.3 Adder/Subtracter Optimization

Figure 15 shows a common idiom used in the first laboratory assignment for implementing the various functional units. The problem with this is that the synthesizer might infer a separate adder, subtracter, and comparator and then connect them with a mux. We can examine the `synth_area.rpt` report (shown in Figure 16) to see exactly what was synthesized. The report indicates that synthesizer did indeed infer a separate adder, subtracter, and two comparators. We might consider recoding our Verilog to be more explicit with what we actually want in terms of hardware. Figure 17 shows one possibility where we use a single adder. For subtraction we simple invert and add one to the operand and use the same adder. After making this change (and rerunning our tests), Design Compiler now synthesizes an adder and an incrementer. We also avoid using the comparison operators eliminating the inferred comparators. The alternative implementation is 20% smaller and although the `addsub` module is still on the critical path the effective clock period is now 6.957 ns. Can you think of same way to eliminate the incrementer?

```
module alu_addsub
(
  input  [1:0]  addsub_fn, // 00 = add, 01 = sub, 10 = slt, 11 = sltu
  input  [31:0] alu_a,     // A operand
  input  [31:0] alu_b,     // B operand
  output [31:0] result     // result
);

  assign result
    = ( addsub_fn == 2'b00 ) ? ( alu_a + alu_b ) :
      ( addsub_fn == 2'b01 ) ? ( alu_a - alu_b ) :
      ( addsub_fn == 2'b10 ) ? ( { 31'b0, ($signed(alu_a) < $signed(alu_b)) } ) :
      ( addsub_fn == 2'b11 ) ? ( { 31'b0, (alu_a < alu_b) } ) :
                               ( 32'bx );
endmodule
```

Figure 15: Initial Verilog for adder/subtracter functional unit

```
Reference          Library       Unit Area   Count    Total Area   Attributes
--------------------------------------------------------------------------------
BUFFD16            tcb013ghpwc    40.737598       1    40.737598
...
alu_addsub_DW01_add_1          5177.075195       1   5177.075195  h
alu_addsub_DW01_cmp2_32_0      1402.053589       1   1402.053589  h
alu_addsub_DW01_cmp2_32_1      1403.751343       1   1403.751343  h
alu_addsub_DW01_sub_1          5195.747559       1   5195.747559  h
--------------------------------------------------------------------------------
Total 19 references                                 14219.134766
```

Figure 16: Area results initial for adder/subtracter functional unit

4.4 Design Ware Optimizations

Based on the previous section you might assume that performing such low-level Verilog transformations is the best way to improve your design. While this is sometimes true, it is also very possible that such approaches will make your design *worse*. Low-level Verilog can get in the way of the synthesis tool and complicate its job. Synopsys provides a library of commonly used arithmetic components as highly optimized building blocks. This library is called DesignWare and the toolflow is already setup to try and use DesignWare components where it can. For example, the adder, subtracter, and comparators listed in Figure 16 are DesignWare blocks as indicated by the DW in their name. We can try to optimize our design by making sure that the tool is inferring the DesignWare components we desire.

For example, we might want to try to have Design Compiler use a unified adder/subtracter DesignWare component instead of complementing and incrementing `alu_b` ourselves. We first consult the DesignWare quick reference `/mit/6.884/doc/design-ware-quickref.pdf` to see if the DesignWare libraries include an adder/subtracter. It does indeed, so we then lookup the datasheet for the `DW01_addsub` component (`/mit/6.884/doc/design-ware-datasheets/dw01_addsub.pdf`). The data sheet recommends that we write our Verilog code as show in Figure 18. After making this change we re-synthesize our design only to find that Design Compiler has chosen to use an adder and a subtracter instead of the `DW01_addsub` component. Nothing is really wrong here - the tool has used various cost functions and ultimately decided that using a separate adder and subtracter was better than using the `DW01_addsub` component. It might choose to use the `DW01_addsub` component if we adjusted the clock period constraint. Can you use a DesignWare component in your alu shifter?

If we *really* want to try and force Design Compiler to use the `DW01_addsub` module, we can do so by simply instantiating the DesignWare component directly as shown below.

```
DW01_addsub#(32) dw_addsub( .A(alu_a), .B(alu_b), .CI(1'b0), .ADD_SUB(~ctl),
                            .SUM(sum), .CO(cout) );
```

Note that since we still need to simulate our processor with VCS, we need to somehow point to a valid functional implementation of the adder/subtracter. Synopsys provides these functional models for all of the DesignWare components and to use them you must add the following line to your `vcs_extra_options` variable in `mips2stage.mk`.

```
-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 DesignWare libraries, and it limits the options available to Design Compiler during synthesis.

Documentation on the DesignWare libraries can be found in the locker. This documentation discusses the best way to encourage the tools to infer the proper DesignWare block.

- `design-ware-quickref.pdf` - DesignWare quick reference
- `design-ware-user-guide.pdf` - DesignWare User Guide
- `design-ware-datasheets` - Directory containing datasheets on each component

```
module alu_addsub
(
  input  [ 1:0] addsub_fn, // 00 = add, 01 = sub, 10 = slt, 11 = sltu
  input  [31:0] alu_a,     // A operand
  input  [31:0] alu_b,     // B operand
  output [31:0] result     // result
);

  wire [31:0] xB = ( addsub_fn != 2'b00 ) ? ( ~alu_b + 1 ) : alu_b;
  wire [31:0] sum = alu_a + xB;

  wire diffSigns = alu_a[31] ^ alu_b[31];

  reg [31:0] result;
  always @(*)
  begin

    if (( addsub_fn == 2'b00 ) || ( addsub_fn == 2'b01 ))
      result = sum;

    else if ( addsub_fn == 2'b10 )
      result = ( diffSigns ) ? { 31'b0, ~alu_b[31] } : { 31'b0, sum[31] };

    else if ( addsub_fn == 2'b11 )
      result = ( diffSigns ) ? { 31'b0,  alu_b[31] } : { 31'b0, sum[31] };

    else
      result = 32'bx;

  end

endmodule
```

Figure 17: Alternative Verilog for adder/subtracter functional unit (see Verilog source code in `examples/mips2stage` for more details on how the set-less-than logic works)

```
register [31:0] sum;
always @( alu_a or alu_b or ctl )
begin
  if ( ctl == 1 )
    sum = alu_a + alu_b;
  else
    sum = alu_a - alu_b;
end
```

Figure 18: Verilog always block for adder/subtracter functional unit in an attempt to have Design Compiler infer a `DW01_addsub` DesignWare component

5 Deliverables

For this lab assignment you are to synthesize your initial design as well as an optimized design. You are free to optimize your design however you wish. You can try to focus on decreasing area, on increasing performance, or both. You can use Verilog modifications, tool script changes, or DesignWare components. It is not enough to just follow the suggestions made in Sections 4.1-4.3; you need to try something on your own. Note that throughout this handout we have been using a clock period constraint of 7 ns, but you should try several different clock period constraints. If you are trying to decrease area you will probably have a longer clock period, while if you are trying to increase performance you will obviously be trying for a shorter clock period.

You should submit your final optimized design using CVS. If you would like you can create two separate makefile fragments in the `config` directory along with possibly multiple versions of the `template*` scripts so that you can build both the original design as well as the optimized design. However, this is not necessary. As discussed in Section 3 you should create a `lab2-initial` CVS tag so you can always recreate your initial synthesis and place+route.

In addition to the optimized design, you must also submit a short (one page) lab report. This report should be a text document and it should be placed in the following directory `mips2stage/lab2writeup`. The writeup should discuss any modifications you made to your original Verilog to get it to synthesize, and it should also describe what optimizations you made to increase performance or decrease area. You should also clearly indicate the optimized total area, total area minus the register file, and final effective clock period. Please comment on what the final critical path is in your design. You should also include the reports for your initial and optimized design. We will put together an (anonymous) scatter plot of each student's area vs. clock cycle results so you can compare your design to everyone else's design. The following list outlines exactly what should be in your `mips2stage/lab2writeup` directory.

- `writeup.txt`
- `initial_synth_area.rpt`
- `initial_synth_timing.rpt`
- `initial_dc.log`
- `initial_pr_area.rpt`
- `initial_pr_timing_slacks.rpt`
- `initial_pr_timing_violations.rpt`
- `final_synth_area.rpt`
- `final_synth_timing.rpt`
- `final_dc.log`
- `final_pr_area.rpt`
- `final_pr_timing_slacks.rpt`
- `final_pr_timing_violations.rpt`

It would be useful if you use a CVS tag once you have committed all of your changes and finished your writeup. This will make it easier for you to go back and reexamine your design. As before,

you should always create CVS tags on clean checkouts. See Section 2 for a list of commands to use when tagging your project. Please make sure that all the necessary files are checked into the `lab2writeup` directory.