ASIC Implementation of a Two-Stage SMIPSv2 Processor

6.375 Laboratory 2
February 28, 2006

In the first lab assignment, you built and tested an RTL model of a two-stage pipelined SMIPSv2 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 checked into CVS, and (b) written answers to the critical questions given at the end of this document. The lab assignment is due at the start of class on Friday, March 6. You may submit your written answers by hand at the beginning of class or electronically by adding a directory titled `writeup` to your lab project directory. Electronic submissions must be in plain text or pdf format.

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 6.375 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 an RTL model into a gate-level model. For this lab assignment, Design Compiler will take your RTL model of the SMIPSv2 processor as input along with a description of the standard cell 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 SMIPSv2 processor will take on the order of 15 minutes, but for your projects it could take up to an hour. After place+route, we will generate and simulate a final gate-level netlist using VCS. We can use this gate-level simulation as a final test for correctness and to generate transition counts for every net in the design. Encounter can take these transition counts as input and correlate them with the capacitance values in the final layout to produce dynamic power measurements.

Each piece of the toolflow has its own build directory and its own makefile. Please consult the following tutorials for more information on using the various parts of the toolflow.

- Tutorial 4: RTL-to-Gates Synthesis using Synopsys Design Compiler
- Tutorial 5: Automatic Placement and Routing using Cadence Encounter
- Tutorial 6: Power Analysis using Synopsys VCS and Cadence Encounter

For this lab we will be using a dummy memory to model the combinational delay through the instruction and data memories. 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 should help you analyze more reasonable critical paths in your design. In the next lab we will start using memory generators which will create synchronous on-chip SRAMs for your processor. These SRAMs have synchronous read ports and thus they would force our two stage pipeline to be a four stage pipeline. For now we will limit our selves to a dummy memory and a simpler two stage pipeline.

The solution for the first lab is now available in CVS as `examples/smipsv2-2stage-v`. Feel free to reference the `smipsv2-2stage-v` code when completing this lab. You are particularly encouraged to copy the provided branch predictor and integrate it into your design.

Getting Started

All of the 6.375 laboratory assignments should be completed on an Athena/Linux workstation. Please see the course website for more information on the computing resources available for 6.375 students. Once you have logged into an Athena/Linux workstation you will need to setup the 6.375 toolflow with the following commands.

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

You will be using CVS to manage your 6.375 laboratory assignments. Please see *Tutorial 2: Using CVS to Manage Source Code* for more information on how to use CVS. Every student has their own directory in the repository which is not accessible to other students. Assuming your Athena username is `cbatten`, you can checkout your personal CVS directory using the following command.

```
% cvs checkout 2006s/students/cbatten
```

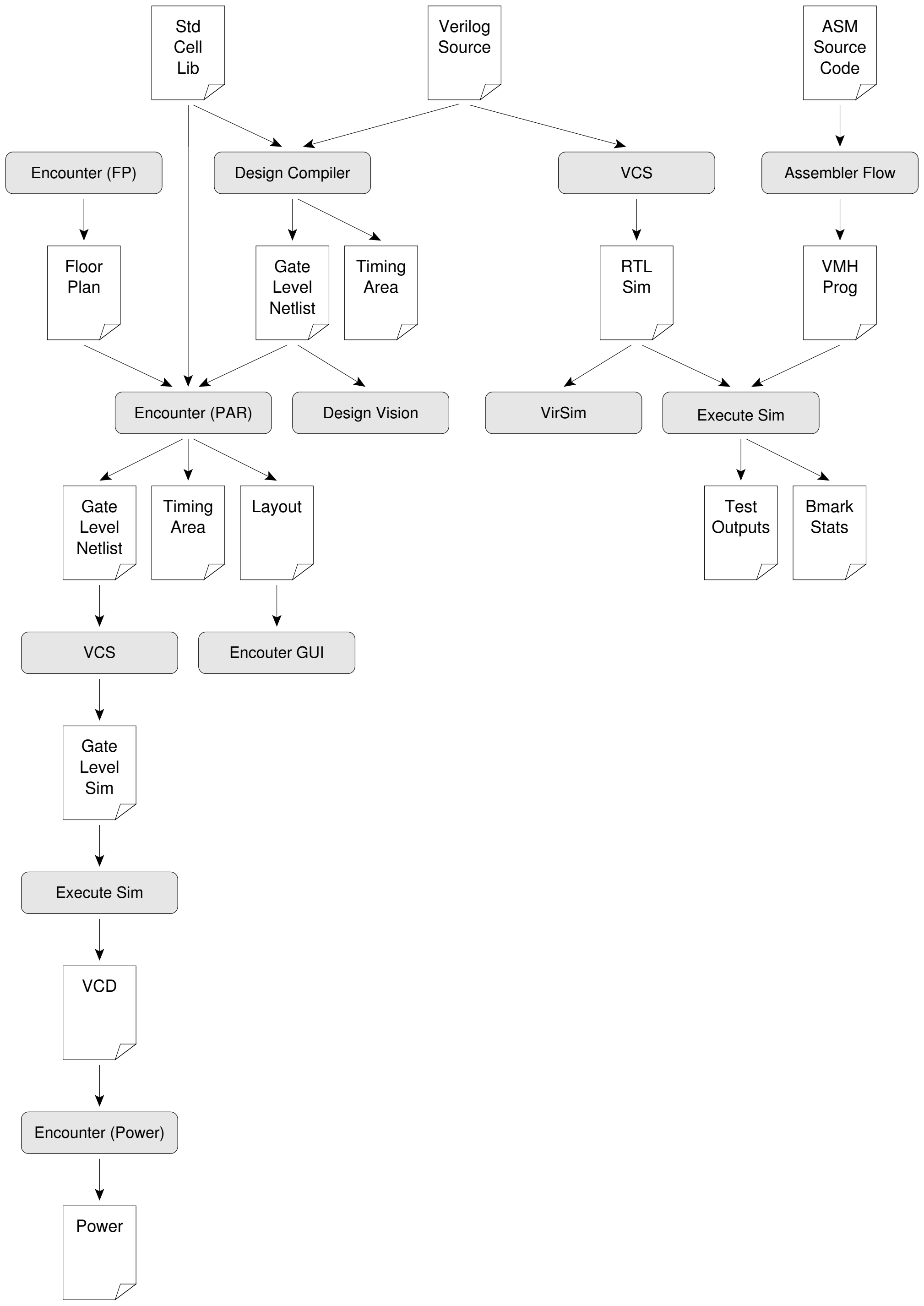


Figure 1: 6.375 Toolflow for Lab 2

To begin the lab you will need to make use of the lab harness located in `/mit/6.375/lab-harnesses`. The lab harness provides makefiles, scripts, and the Verilog test harness required to complete the lab. The following commands extract the lab harness into your CVS directory and adds the new project to CVS.

```
% cd 2006s/students/cbatten
% tar -xzvf /mit/6.375/lab-harnesses/lab2-harness.tgz
% find lab2 | xargs cvs add
% cvs commit -m "Initial checkin"
```

The resulting `lab2` project directory contains the following primary subdirectories: `src` contains your source Verilog; `tests` contains local assembly tests; `bmarks` contains local benchmarks; and `build` contains makefiles and scripts for building your processor. The `src` directory contains the Verilog test harness and other Verilog modules you will need in this lab assignment. Your first step is to copy over your source files from the first lab into the `lab2` project directory. Please be aware that the toplevel test harness structure has changed a bit. The `build` directory contains the following subdirectories which you will use when building your chip.

- `vcs-sim-rtl` - RTL simulation suing Synopsys VCS
- `dc-synth` - Synthesis using Synopsys Design Compiler
- `enc-fp` - Floorplanning using Cadence Encounter
- `enc-par` - Automatic placement and routing using Cadence Encounter
- `vcs-sim-gl` - Gate-level simulation using Synopsys VCS
- `enc-power` - Power analysis using Cadence Encounter

Each subdirectory includes its own makefile and additional script files. **You will have to make modifications to these script files as you push your design through the physical toolflow.** Once you have all the tools working you can use the toplevel makefile in the `build` directory to run multiple tools at once. For example, once all the scripts are properly setup you should be able to use the following command to synthesize, floorplan, and place+route your design.

```
% pwd
2006s/students/cbatten/lab2/build
% make enc-par
```

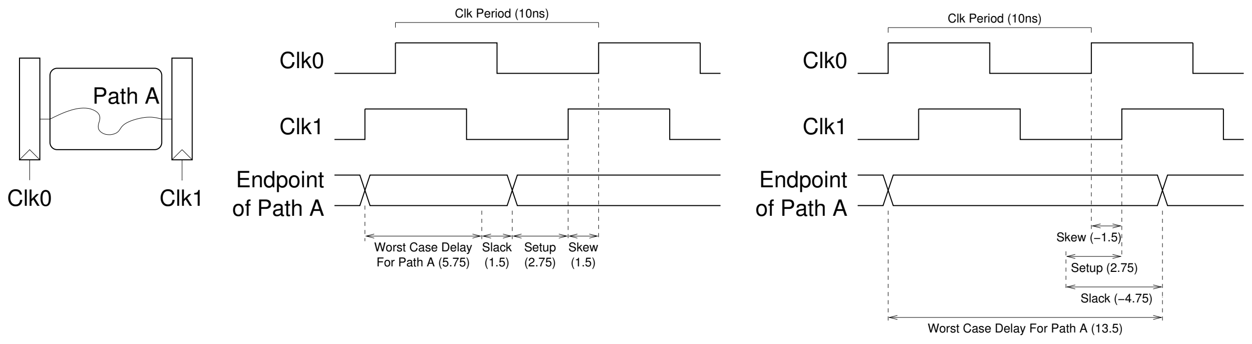
Automatic placement and routing is a very computationally intensive task. On your simple two-stage project place+route can take anywhere from 10 minutes to 20 minutes. Budget your time accordingly.

Lab Hints and Tips

This section contains several hints and tips which should help you in completing the lab assignment.

Tip 1: Always Test Your Processor After Making Modifications

When pushing your processor through the physical toolflow, it is common to make some changes to your RTL and then evaluate their impact on area, power, and performance. Always retest



(a) Example Circuit (b) Effective Clock Period is 8.5ns (c) Effective Clock Period is 14.75ns

Figure 2: Determining your hardware's effective clock period

your processor after making changes and before starting the physical toolflow. You can use the `run-asm-tests` make target to quickly verify that your processor is still functionally correct. Keep in mind, that a fast or small processor which is functionally incorrect is worse than a slow or large processor which works!

Tip 2: Replace Non-Synthesizable Verilog

Not all Verilog is synthesizable, so you may receive errors when you first push your processor through Design Compiler. For example, the `smipsInst.v` code which was supplied as part of the first lab included a useful `unpackInst` module. Unfortunately, hierarchical references are not supported by Design Compiler. Instead of using `unpackInst` you can use the `` `define`` macros provided in `smipsInst.v`. To retrieve the rs specifier from an 32 bit instruction wire named `inst` you can use ``inst[`INST_RS]``. Another common mistake is to use the Verilog reduction nor operator (`~|`) to implement a two operand nor. VCS will accept (`a ~| b`) but Design Compiler will not. You can change (`a ~| b`) into (`~(a|b)`). Don't forget to retest your design after making any changes!

Tip 3: Reporting Clock Period

As discussed in the tutorials, you need to specify a clock period constraint during both synthesis and place+route. The tools will try and meet this constraint the best they can. If your constraint is too aggressive, the tools will take a very long time to finish. They may not even be able to correctly place+route your design. If your constraint is too conservative, the resulting implementation will be suboptimal.

Even if your design does not meet the clock period constraint it is still a valid piece of hardware which will operate correctly at some clock period (it is just slower than the desired clock period). If your design does not meet timing the tools will report a *negative slack*. 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* ($T_{clk} - T_{slack}$). The synthesis and place+route timing reports are all 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 2 illustrates two examples: one with positive slack and one with negative slack. In this example, our clock period constraint is 10 ns. In Figure 2(b), the post-place+route reports indicate a positive slack value of 1.5 ns and thus the effective clock period is 8.5 ns. In Figure 2(c), the post-place+route reports indicate a negative slack value of 4.75 ns and thus the effective clock period is 14.75 ns. Notice that the effective clock period in Figure 2(c) is *not* equal to the worst case combinational critical path (i.e. 13.5 ns). This is because we must also factor in setup time and clock skew.

Determining the effective clock period when your design does not meet timing can be more tricky when we include latches (as opposed to flip-flops) in our design. Latches impose additional half-cycle constraints. For this lab it is adequate to ignore any half-cycle constraints and focus instead on the full cycle constraints.

Critical Thinking Questions

The primary deliverable for this lab assignment is your optimized Verilog source and all of the scripts necessary to completely generate your ASIC implementation checked into CVS. In addition, you should prepare written answers to the following questions and turn them in at the beginning of class on March 3.

Question 1: Evaluate Your Baseline Processor

Push your baseline processor through the physical toolflow and report the following numbers. Initially push your design through without any floorplanning and then try floorplanning the register file, the dummy memory, and any other modules you wish. For the remaining questions in this lab you can choose to either use floorplanning or not.

- Post-synthesis area of the register file, datapath (excluding register file), and control unit in "abstract units" from `synth_area.rpt`
- Post-synthesis total area of processor (excluding dummy memory) in "abstract units" from `synth_area.rpt`
- Post-synthesis critical path and corresponding effective clock period in nanoseconds from `synth_timing.rpt`
- Post-place+route area of the register file, datapath (excluding register file), and control unit in square micron from `postroute_area.rpt` (no floorplanning)
- Post-place+route total area of the processor (excluding dummy memory) in square micron from `postroute_area.rpt` (no floorplanning)
- Post-place+route total area of the processor (including dummy memory and power ring) in square micron from the Encounter GUI (no floorplanning)
- Post-place+route critical path (and just the critical path) and corresponding effective clock period in nanoseconds from `postroute_setup_timing.rpt` (no floorplanning)
- Post-place+route critical path and corresponding effective clock period from `postroute_setup_timing.rpt` (with floorplanning)

Your post-place+route numbers will probably be significantly worse than your post-synthesis numbers. Explain why the place+route tool is reporting more area and a longer clock period than the synthesis tool. In Question 2 of the first lab you made some predictions concerning the critical paths and the area of your design. Were these predictions correct? If they differ how has your intuition changed after pushing your processor through the physical toolflow?

Question 2: Use Logical Effort to Design an Optimal Branch Comparator

In this question, you will be using logical effort to implement the branch equality comparator in your SMIPSv2 processor. If you eliminated the branch comparator in your design, then use the `examples/smipsv1-1stage-v` project to complete this question. Before answering this question you should read the first chapter of *"Logical Effort: Designing Fast CMOS Circuits"* by Sutherland, Sproull, and Harris. It is available in the course locker at `/mit/6.375/doc/logical-effort-ch1.pdf`. You should make use of the tables contained in that chapter when answering this question.

For our process, the unit-less delay scaling factor (τ) is 10.5 ps and the unit-less parasitic delay of an inverter (P_{inv}) is approximately one. You may assume that we are using gates with balanced rise/fall times. Your branch equality comparator should take two 32-bit inputs and produce a 1-bit output. If the output is one then the two inputs are equal, and if the output is zero then the two inputs are not equal. The input capacitance of your comparator should be around 5 fF. The output must drive a capacitive load of 5 fF. You should limit yourself to the logical gates in the Tower 0.18 μm Standard Cell Library. The databook for the standard cell library is located in the course locker at `/mit/6.375/doc/tsl-180nm-sc-databook.pdf`. Although you should limit yourself to the standard cell *logical* gates (for example, a two-input XOR gate), do not limit yourself to the standard cell *physical* gates (for example, a two-input XOR gate with 2X drive). In other words, you can choose the size of each gate to be whatever you wish; you are not limited to the sizes in the standard cell library.

Your goal is to design the optimal gate topology for the branch comparator and to identify the optimal input capacitance for each gate in that topology. You do not need to convert the input capacitances into transistor widths. You should identify at least three likely candidate gate topologies (preferably with different numbers of stages) and calculate the path effort, optimal stage effort, and optimal path delay for each. How much difference is there between the optimal delay of the various topologies? For the topology with the minimum delay, work backwards from the output capacitance to calculate the optimal input capacitance *in femtofarads* for each stage.

Once you have completed your calculations, examine the topology chosen by Design Compiler and include it in your answer. You may need to isolate your branch comparator in its own module so that it is synthesized separately. What is the load capacitance that Design Compiler is using for the output of the branch comparator? You can determine this using the following command from the Design Vision prompt after synthesis. You will need to use the hierarchical net name which is appropriate for your design.

```
design_vision-xg-t> report_net proc/dpath/branch_cond_gen/out \
                               -significant_digits 5
```

Now examine the gate-level netlist for the branch comparator after place+route and include it in your answer. You can get a feel for this using the *Design Browser* window in the Encounter GUI. What cells did Encounter decide to use? Did it resize the gates? Did it add any new cells? Why do you think Encounter might change the design? What is the actual load capacitance for the output of the branch comparator according to Encounter? You can find this through the Encounter GUI. You will need to execute the `extractRC` command first, and then locate the branch comparator output net in the Encounter *Design Browser*. Choose the *Tool → Attribute Editor* menu option to display various properties (including the capacitance) of the net.

You can use Design Vision to visualize the final post-place+route netlist. After finishing place+route, move into your current `dc-synth` build directory and use the `read_file` command to read in the `par.v` file located in the place+route build directory. The following commands illustrate how to load the post-place+route netlist into Design Vision.

```
% pwd
2006s/students/cbatten/lab2/build
% cd dc-synth/current
% design_vision-xg-t
design_vision-xg-t> source libs.tcl
design_vision-xg-t> read_file -format verlog ../../enc-par/current/par.v
design_vision-xg-t> current_design smipsCore_synth
```

We need to use the `current_design` command, because otherwise Design Vision will set the current design to a module lower down in the design hierarchy. You can now use Design Vision to view a schematic of the post-place+route branch comparator.

Compare and contrast the three gate topologies: the topology you designed using logical effort, the topology synthesized by Design Compiler, and the final topology after Encounter finishes place+route. How do you think these results would change if the branch comparator output had to drive a load of 5000 fF instead of 5 fF?

Question 3: Use RC Modeling to Design a Register-File Write Bit-line Driver

In this question you will be designing a simple buffer for a write bit-line in a register file. The write bit-line (i.e. the write data) must drive the D input port of 32 flip-flops. The combined gate capacitance of these flip-flops can be a significant load on the write bit-line. The load on the write bit-line is further increased by wire capacitance, since flip-flops are usually large and thus often spread apart.

To witness the problem first hand, use Encounter to isolate the portion of your critical path which is contained in the register write bit-line. If the write bit-line is not on your critical path then browse through the paths in `postroute_setup_timing.rpt` until you find a path which includes the write bit-line. Figure 3 shows a fragment of an Encounter timing report. You can see the critical path going through the writeback mux and into the write data port of the register file. Notice the very large capacitive loads on the nets along this path. More specifically, the load on the final inverter is a portion of the capacitive load of the write bit-line (0.136 pF). You can see how the physical toolflow has inserted two buffers to help drive this large load. The `par.cap` file contains a breakdown between wire and gate capacitance for each net in the design. Searching for the `dpath/rfile/n255` net reveals that this final inverter is driving 12 other gates; so the tools have created a tree of inverters to help drive all 32-bits of the write-bit line. The wire capacitance is 96 fF and the gate capacitance is 40 fF for a total load of 136 fF. In your answer to this question, you should include a similar analysis of the impact the write bit-line has on your critical path.

```
Object name                        Delta r/f (ns) Sum r/f (ns)   Slew (ns)   Load (pF)
-------------------------------------------------------------------------------------
...
dpath/wb_mux/U212 I0->Z (mx02d1)  0.164r/0.208f  6.064r/5.624f  0.110r/0.102f  0.009
dpath/wb_mux/n148                  0.001r/0.001f  6.064r/5.625f
dpath/wb_mux/U24 I1->Z (mx02d4)   0.372r/0.419f  6.436r/6.043f  0.143r/0.122f  0.186
dpath/rf_wdata_Xnp[7]              0.128r/0.124f  6.564r/6.167f
dpath/rfile/U545 I->ZN (invbd2)   0.219f/0.164r  6.783f/6.332r  0.685r/0.447f  0.054
dpath/rfile/n254                   0.003f/0.003r  6.786f/6.335r
dpath/rfile/U198 I->ZN (inv0d7)   0.104r/0.065f  6.890r/6.399f  0.274f/0.227r  0.136
dpath/rfile/n255                   0.030r/0.028f  6.920r/6.427f
dpath/rfile/registers_reg[19][7] CP^^D (denrq4)
                                   0.189r/0.131f  7.109r/6.558f  0.227r/0.175f
```

Figure 3: Example fragment from the postroute_setup_timing.rpt

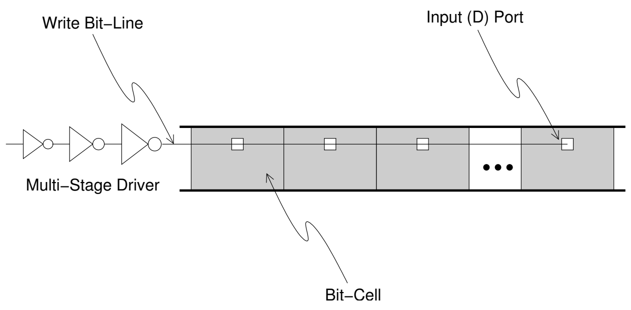


Figure 4: Multi-Stage Driver for a Register File Write Bit-Line

Transistor Process Parameters	**Value**
Desired ratio of PMOS/NMOS widths	2
PMOS gate capacitance per μm of transistor width	1.5 fF/μm
NMOS gate capacitance per μm of transistor width	1.5 fF/μm
PMOS drain capacitance per μm of transistor width	0.3 fF/μm
NMPS drain capacitance per μm of transistor width	0.3 fF/μm
PMOS effective on resistance	6.7 k$\Omega\mu$m
NMOS effective on resistance	3.3 k$\Omega\mu$m

Parameters for Metal 2 Wire	**Value**
Wire resistance per unit length	0.4 Ω/μm
Wire capacitance per unit length	0.2 fF/μm

Table 1: Process Parameters

We will now use a simple RC model to design a similar driver for a write bit-line. We will use process parameters from a 0.18 μm TSMC process, but they should be similar to the Tower 0.18 μm process. Table 1 shows the various parameters. Your first task is to model the write bit-line. Assume that the register file is using DENRQ1 standard cells for the register file bit-cells. Use the standard cell databook to determine the input capacitance for the D input in picofarads and to determine the size of the bit-cell. The databook is located in the course locker at `/mit/6.375/doc/tsl-180nm-sc-databook.pdf`. All of the Tower standard cells are 5.6 μm tall and the width of each standard cell is listed in "Gate Equivalents". One "Gate Equivalent" is 2.24 μm. Assume that the bit-cells are arranged as shown in Figure 4 and that the D input is located directly in the middle of each bit-cell. We will not be designing a tree; instead we will design a series of drivers which drive the design from one end of the bitline. The write bit-line will use metal 2 wire. Create a distributed RC model of the bit line which includes the bit-cell gate capacitance, the wire capacitance, and the wire resistance.

Now we will try and design a multi-stage driver suitable for driving the bit-line. You do not need to use standard cell inverters for the driver; assume we can use custom inverters that are whatever size we wish. Assume that it is okay to invert the signal (we can just add another inverter at the read port). The input capacitance of your driver should be a minimum sized inverter. Use a lumped model of the bit-line capacitance to estimate how many inverter stages we need. What should be the size of each stage? Use a simple RC model to estimate the worst case delay (in RC time constants) to drive the very last bit on the bit-line. You can use a π model for the bitline and assume that after one RC time constant the next inverter turns on.

Question 4: Optimize Your ALU

In lab one you used a semi-behavioral ALU. Although synthesizable, your ALU is probably much larger than what one would design by hand. Use the post-synthesis `synth_area.rpt` and `synth_resources.rpt` reports to identify how your ALU is being synthesized. How many Synopsys DesignWare components are being inferred? Ideally we should be able to implement the SMIPSv2 ALU with just an adder, a left shifter, a right shifter, and a logic unit. The logic unit would contain bit-wise and, or, nor, and xor. Optimize your ALU for area and push it through the physical toolflow. Report the change in post-place+route area (use `postroute_area.rpt`) and performance. Don't forget to retest your design with your optimized ALU! Feel free to use your optimized ALU in Questions 5 and 6.

Question 5: Evaluate a Latch Based Register-File

The register file is probably the largest component of your design. Identify what standard cell is used for the majority of the register file bit-cells. The lab harness includes a Verilog latch based register file (`smipsProcDpathRegfile_latch`). Push your design through the physical toolflow with the latch based register file. Report the change in post-place+route area (use `postroute_area.rpt`) and performance. Identify what standard cell is used for the majority of the latch based bit-cells. Give a high-level reasoning why latch based register files should be smaller than flip-flop based register files. (Hint: How do the master/slave parts of a standard flip-flop relate to the latches in a latch based register file?) Don't forget to retest your design with the latch based register file! Feel free to use the latch based register file in Questions 6.

This question assumes you used a flip-flop based register file. If you implemented a more optimized latch based register file in the first lab then use the `smipsProcDpathRegfile_flipflop` register file for this question. Evaluate the difference between your latch based register file and the provided flip-flop based register file.

Question 6: Evaluate a Simple Branch-Predictor

In lab one you implemented a simple branch predictor to help increase the IPC of your processor. Although the predictor increases the IPC, it may negatively impact the area, power, or performance of your processor. In this question we want to evaluate whether or not the branch predictor is a beneficial addition to the processor. If you did not have a chance to finish the predictor, feel free to copy the predictor from `examples/smipsv2-2stage-v` and integrate it into your processor. Make sure that your processor passes all of the assembly tests with the new predictor. If you have trouble with the integration you can also complete this question using the `examples/smipsv2-2stage-v` project instead of your own. Report the change in post-place+route area (use `postroute_area.rpt`) and performance after adding the branch predictor. Factoring in the advantages of the predictor (increased IPC) and the possible disadvantages (increased area and clock period), do you think adding the predictor is a good idea?