ASIC Implementation of a Three-Stage SMIPSv2 Processor

6.375 Laboratory 3
Part I
March 6, 2007

In the first lab assignment, you built and tested an RTL model of a two-stage pipelined SMIPSv2 processor while in the second lab, you built a Bluespec model of a 3 stage pipeline. In the third lab assignment, you will be using various commercial EDA tools to synthesize, place, and route your Bluespec design. In addition, you will attempt to optimize your design to increase performance and/or decrease area. This lab has two objectives: the first is to introduce you to the tools you will be using in your final projects and give you some intuition into how high-level hardware descriptions are transformed into layout. The second objective is to deepen your understanding of Bluespec to the point where you can achieve full parallelism in your Bluespec designs. This will require you to fully grok what rule scheduling is all about, what makes rules conflict, and what semantic ugliness we use to bypass the problem: RWIRES and EHR registers. CircBSV.tar, which is posted on the resources page of the course web-site, has some good examples, though you may want to use the EHR elements from the course bluespec library (/mit/6.375/install/bsvclib). Remember: RWire and its relatives polute the otherwise clean semantics of bluespec and while a solution to this problem of sequential rule composition has been proposed, it is not yet in the Bluespec compiler.

The deliverables for this lab are (a) your optimized Bluespec 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 16. As usual, you must submit the written answers by hand in class.

Before starting this lab, it is recommended that you revisit the Bluespec model you wrote in the second 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. Thus you might want to consider breaking your design into smaller pieces, allowing you to reason about performance of isolated components. By default, the Bluespec compiler flattens all module definitions into one large module. Use of the `synthesize` annotation before a module implementation will direct BSC to generate modular verilog. Please refer to the language reference guide (/mit/6.375/doc/bsc-reference-guide.pdf) for more information on the use of this directive. 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 remove the (*synthesize*) directive from your bluespec code, reverting to the Bluespec compiler's default behavior of generating flattened Verilog. (additionally, you can instruct the CAD tools to flatten the design before optimization)

Figure 1 illustrates the 6.375 ASIC toolflow we will be using for the third lab. You should already be familiar with the simulation and compilation paths from the first and second labs. The verilog used by the CAD tools will be generated by the Bluespec compiler and all the coding you do for this assignment should be in BSV. 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 the 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

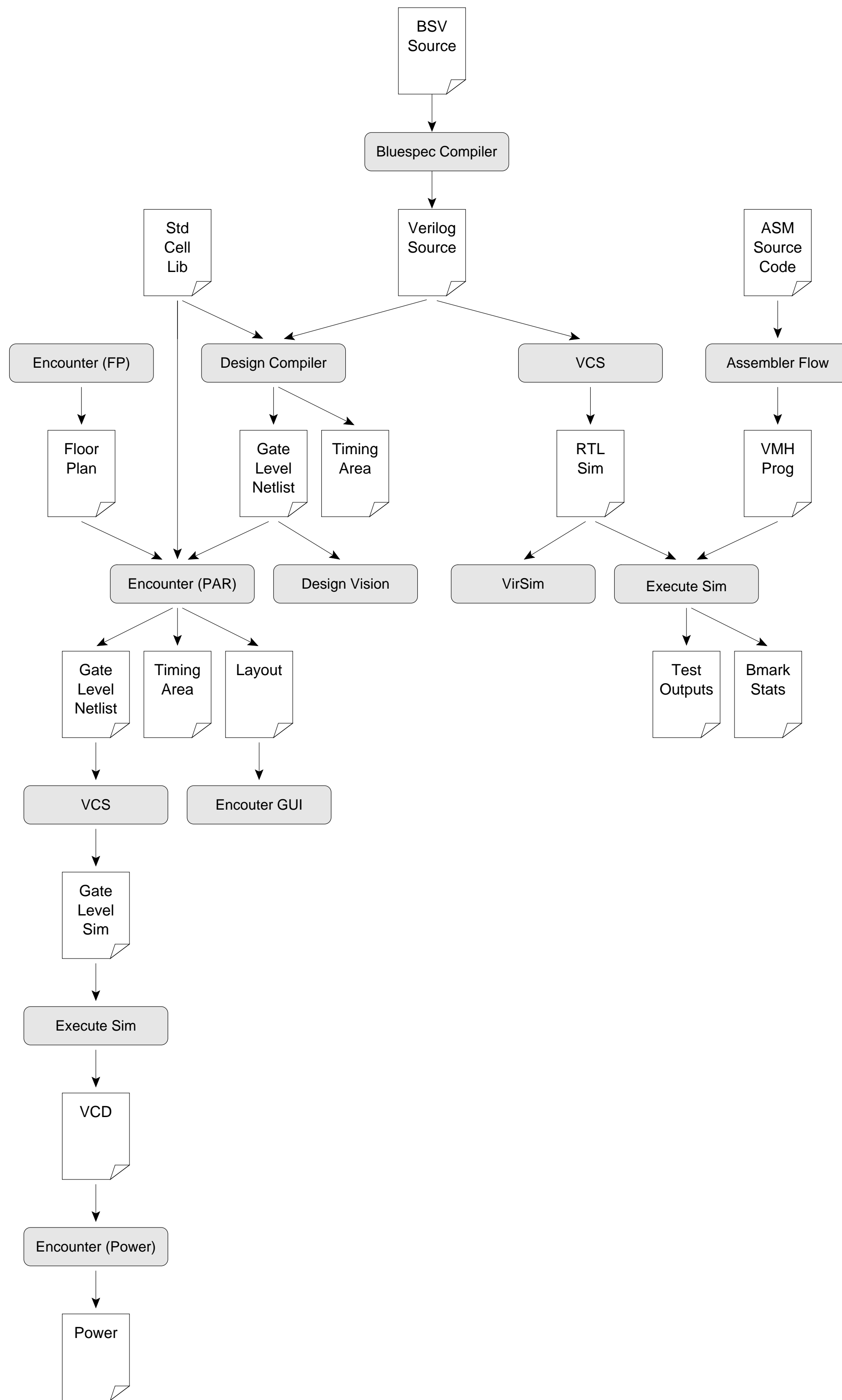

Figure 1: 6.375 Toolflow for Lab 2

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. Note that these tutorials are written for projects written explicitly in Verilog, but for this lab we will be coding in Bluespec and generating Verilog code using the Bluespec compiler.

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

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 2007s/students/cbatten
```

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 other infrastructure required to complete the lab. In order to get started, unpack the test harness, copy the src directory from your lab2 project into the lab3 project directory, make sure you can still pass the benchmarks and assembly tests, and check everything into your CVS directory as lab3. Assuming your lab2 code is located on 2007s/cbatten/lab2, the following commands extract the lab harness into your CVS directory, copy over the relevant files from from lab2, and adds the new project to CVS:

```
% cd 2007s/students/cbatten
```

```
% tar -xzvf /mit/6.375/lab-harnesses/lab3-harness.tgz
% cp lab2/src/*.* lab3/src/
<copy over any local tests if they exist>
<make sure you can still compile and run all the assembly tests>
% find lab3 | xargs cvs add
% cvs commit -m "Initial checkin"
```

The resulting `lab3` project directory closely resembles the structure you used in lab3. 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.

- `bsc-compile` - Bluespec compilation generates Verilog
- `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

Each subdirectory includes its own makefile and additional script files. **You may 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
2007s/students/cbatten/lab3/build
% make enc-par
```

Automatic placement and routing is a very computationally intensive task. On your simple three-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 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 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

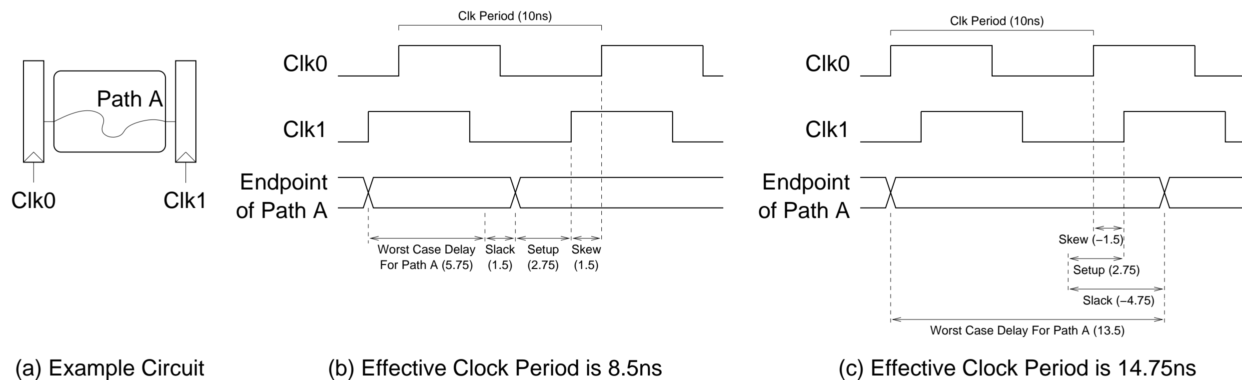

Figure 2: Determining your hardware's effective clock period

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.

Critical Thinking Questions

Question 1: Evaluate Your Baseline Processor

Push your baseline processor (no branch predictor, RWIREs, or EHR registers) 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 memory, and any other modules you wish.

- Post-synthesis total area of the processor in "abstract units" from `synth_area.rpt`. (make sure that the Memory is excluded).

- Post-synthesis critical path and corresponding effective clock period in nanoseconds from `synth_timing.rpt`
- Post-place+route total area of the processor in square micron from `postroute_area.rpt` (no floorplanning)
- Post-place+route total area of the processor in square micron from the Encounter GUI (no floorplanning)
- Post-place+route 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? Try running both the synthesis and place+route tools, successively tightening the clock period constraints. What effect does this have on the area of your design? Look at the sizes of the various modules you use in your microarchitecture. Which modules contribute most to the size of your design?

Using the IPC's calculated running the benchmarks and the estimated clock period, calculate the speed of your processor in instructions per second.

Question 2: Evaluate a Simple Branch-Predictor

If didn't do so in Lab2, implement the simple branch predictor described in the lab2 handout, and verify its correctness by running the assembly tests and benchmarks. Although predictors generally increase the IPC, they may negatively impact the area or delay of your processor. In this question we want to evaluate whether or not the branch predictor is a beneficial addition to the processor. Report the change in post-place+route area (use `postroute_area.rpt`) and timing 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? Given that our baseline processor has poor rule concurrency (you probably won't have gotten pcgen and exec to fire in the same cycle), the increase in IPC is most likley negligible. What does this say about the benefits of action level speculation in the absence of action level concurrency?

Question 3: Refining using Ephemeral History Registers

Identify the souce of conflict between the rules implementing the three pipeline stages of your processor. Through a combination of rule decomposition (you may want to think about having a seperate exec rule for branch and jump instructions) and EHR's, you should be able to increase your IPC to something close to 1.

In the BSVCLib there's an package called `EHR2`. In this package you'll find `mkEHReg2` an implementation of an Ephemeral History Register. The interface is similar to a normal register except with "subscripts" added:

```
interface EHReg2#(type valType);
      method valType read_0();
      method valType read_1();
      method Action write_0(valType val);
      method Action write_1(valType val);
endinterface
```

Recall that the schedule for the EHR is as follows:

```
read_0 < write_0 < read_1 < write_1
```

This package also contains `mkEHRFile2`, a register file which has the Ephemeral History structure. It has the following interface:

```
interface EHRFile2#(type addrType, type valType);
  method Action   wr_0(addrType rindx, valType data);
  method Action   wr_1(addrType rindx, valType data);
  method valType rd1_0(addrType rindx);
  method valType rd1_1(addrType rindx);
  method valType rd2_0(addrType rindx);
  method valType rd2_1(addrType rindx);
endinterface
```

The schedule for this register file is as follows;

```
rd1_0, rd2_0 < wr_0 < rd1_1, rd2_1 < wr_1
```

Finally, the package provides `mkEHRFIFO2` and `mkEHRSFIFO2`. These have the following interface:

```
interface EHRFIFO2#(type valType);
  method Action enq_0(valType d);
  method Action enq_1(valType d);
  method Action deq_0();
  method Action deq_1();
  method valType first_0();
  method valType first_1();
  method Action clear_0();
  method Action clear_1();
endinterface
```

These have the following schedule:

```
first_0 < find_0 < deq_0 < enq_0 < clear_0 < first_1 < find_1 < deq_1 < enq_1
                                     < clear_1
```

Note that there is no bypassing version of these FIFOs. With EHRs this same bypassing effect can be achieved by using `enq_0` and `first_1` with .

Using these constructs, create a version of your design which has the following scheuling property:

```
writeback < execute < pcGen
```

To do this you will need to mechanically transform your design as described in class. After identifying the state elements responsible for the rule conflicts, change them to the appropriate EHR type. Then label all method calls in `writeback` with subsript 0, and proceed from there.

Toggle your branch-predictor and report the IPC numbers with and without branch prediction. With the addition of action level concurrency (more rules firing in parallel), how does the branch predictor (action level speculation) effect performance? What throughput does this version of your design acheive? Synthesize this version of the design as you did for Question 1. How have the results changed and what was the cause of the biggest change? In your opinion is the difference worth it? How does this method compare to experimentally sizing your FIFOs as in Question 1 of lab2?

Question 4: 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`

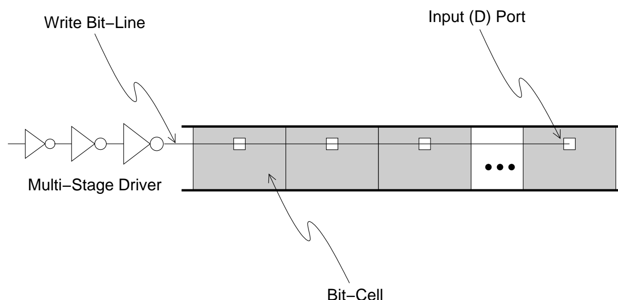

Figure 4: Multi-Stage Driver for a Register File Write Bit-Line

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

Parameters for Metal 2 Wire	**Value**
Wire resistance per unit length	$0.4\,\Omega/\mu\text{m}$
Wire capacitance per unit length	$0.2\,\text{fF}/\mu\text{m}$

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 *custom* inverter (i.e. NMOS width is 0.36 μm and PMOS width is 0.72 μm). 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 5: Use Logical Effort to Design an Optimal Branch Comparator (Extra Credit)

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. You should assume that the XOR/XNOR gates are implemented with static CMOS logic, not transmission gates. The logical effort of XOR and XNOR gates is the same. A two input XOR gate and a two input XNOR gate both have a logical effort of four.

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> 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
2007s/students/cbatten/lab2/build
% cd dc-synth/current
% design_vision-xg
design_vision-xg> source libs.tcl
design_vision-xg> read_file -format verilog ../../enc-par/current/par.v
design_vision-xg> 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?