ASIC Implementation of a RISC-V Core with On-Chip Caches

CS250 Laboratory 3 (Version 082511)
Written by Yunsup Lee (2010)
Updated by Brian Zimmer (2011)

Overview

In the second lab assignment, you wrote an RTL model of a two-stage pipelined RISC-V processor using Chisel and synthesized your RTL model. In the third lab assignment, you will substitute magic memory with on-chip caches, extend your design to a full five stage pipeline, and will use all the tools you have learned so far to simulate, synthesize, place and route your design and finally analyze energy consumption. After producing a preliminary ASIC implementation, you will attempt to optimize your design to make it more energy efficient while meeting your time constraint. The objective of this lab is to introduce memory blocks that you might use in your final project, as well as to give you some intuition into how high-level hardware descriptions are transformed into layout.

We have provided a 16 KB instruction cache and a 16 KB data cache to use. Both caches have a 32-bit wide CPU access port and a 128-bit wide memory refill port. These caches block on a cache miss. Cache lines are 64 bytes, and both caches are initially set to be a 8-way set associative cache. If you do the math, you should be able to figure out that there are 32 sets. These numbers can be changed, since the cache is parameterizable.

After producing a working RTL model for the core, you will attempt to optimize your design to increase energy efficiency.

Deliverables

This lab is due **Monday, October 24th at 1pm**. Deliverables for this lab are:

- (a) your optimized Chisel source and all of the scripts necessary to completely generate your ASIC implementation and analyze energy consumption checked into Git
- (b) an analytic energy model for your core implementation
- (c) written answers to the questions given at the end of this document checked into git as `writeup/report.pdf` or `writeup/report.txt`

Before starting this lab, it is recommended that you revisit the Chisel 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 power results for each module. It will be much more difficult to gain any intuition about the performance or power 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 VLSI tools will not be able to optimize across module boundaries. If you are concerned about this you can explicitly instruct the VLSI 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.

Block Diagram and Module Interfaces

Figure 1 shows the overall diagram. The core module contains the processor, instruction cache, data cache, and an arbiter. The processor is connected to the instruction cache and the data cache with a similar interface used in the second lab. These interfaces are 32-bits wide. Notice that there is a ready signal on the request side, and a valid signal on the response side. This is a more realistic memory interface, where you might not be able to issue a memory request every cycle because the cache is blocked on a miss (when the ready signal on the request side is deasserted), or not be able to read from memory every cycle because you missed in a cache (when the valid signal on the response side is deasserted). Also notice that the instruction memory refill port and the data memory refill port needs to arbitrate on the memory refill port going out of the core. There are several arbitration policies you can implement, but in this lab you will use a fixed priority arbitration scheme, where the instruction memory refill port always gets priority over the data memory refill port. The refill port is 128-bits wide. There will be four transactions on the memory refill interface when a cache line transfer occurs, since the cache line size is 64 bytes.

For this lab, the instruction and data caches operation is modeled by a C model. You are not responsible for implementing the caches in Cacti. Your Cacti code is responsible for describing the processor core and memory, and all signals to and from the caches are sent out of the top level.

For your emulator implementation, these cache signals are hooked up to a functional cache model from `emulator.cpp`.

For your VLSI implementation, there are additional models that describe the timing, area, and power of these caches as well as functionality. These are generated from a modeling program known as `Cacti`.

Both caches have a request/response port to/from the processor and a memory refill port. Notice that the instruction cache interface is simpler compared to the data cache interface since it is a read-only memory. You can take a look at `ICache_32x4096_BC.v` and `DCache_32x4096_BC.v` for more details.

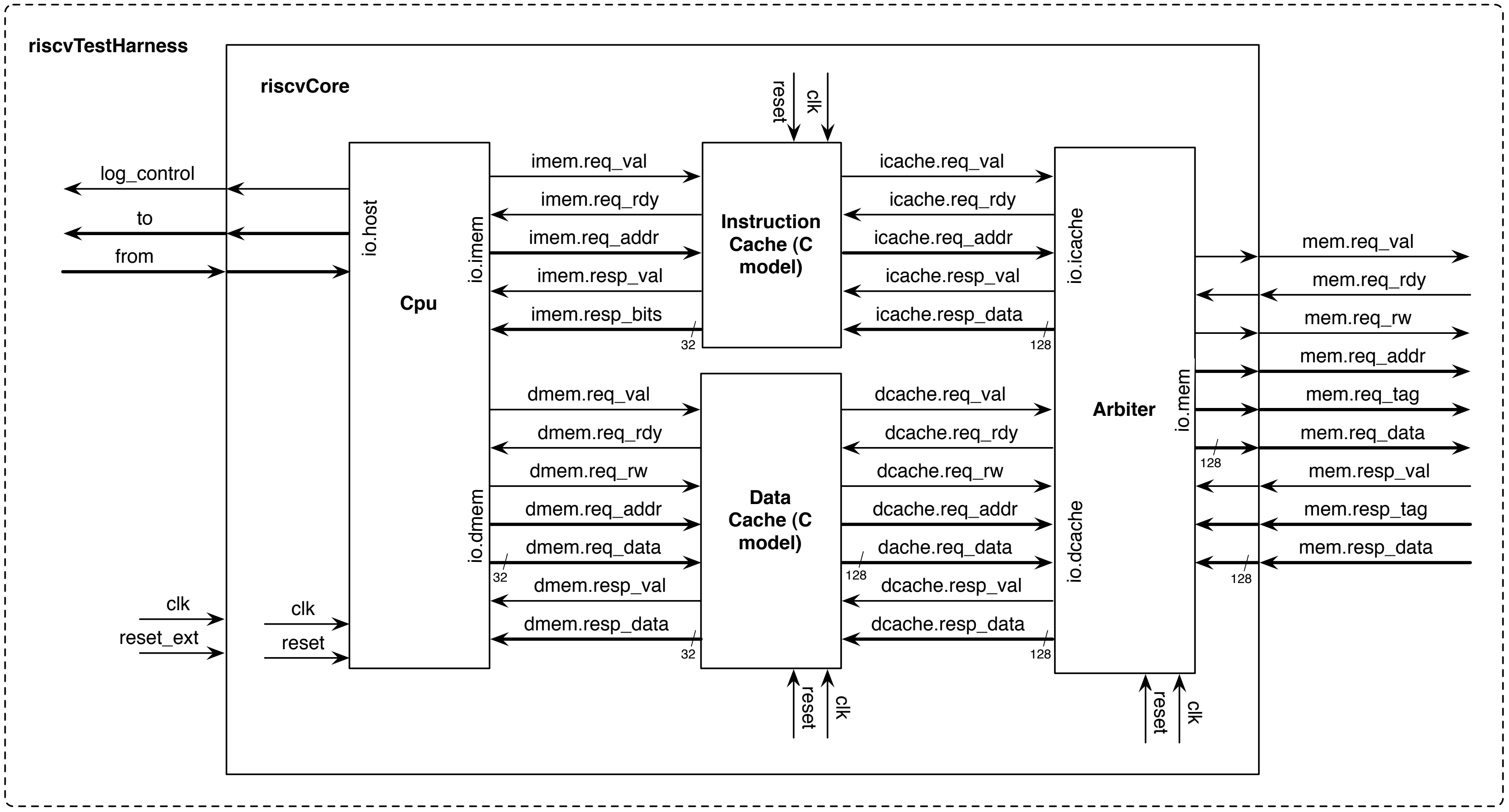

Figure 1: Block diagram for RISC-V v2 Core Test Harness

The arbiter sees both memory refill requests coming from the instruction cache and the data cache and puts its decision on the request ready signals. We have provided you with a simple arbiter.

Test Harness

We are providing a test harness to connect to your core model. The test harness loads a RISC-V binary into the memory. The provided makefile can load both assembly tests as well as C benchmarks to run on your core. The test harness will clock the simulation until it sees a non-zero value coming back on the `testrig_tohost` register, signifying that your core has completed a test program. The `testrig_tohost` port should be set to zero on reset. A very simple test program is shown in Figure 2.

It is worthwhile to mention that the harness differs for the emulator and VLSI implementations. For the VLSI implementation, only your arbiter and cpu are instantiated in a separate test harness. For the emulator, all connections to the caches are sent to the top level, then connected to C models of the caches.

```
# 0x00000000: Reset vector.
       addi $x1, $x0, 1    # Load constant 1 into register x1
       mtpcr $x1, $cr16     # Write x1 to tohost register
loop:  beq   $x0, $x0, loop # Loop forever
```

Figure 2: Simple test program

Getting Started

You can follow along through the lab yourself by typing in the commands marked with a '%' symbol at the shell prompt. To cut and paste commands from this lab into your bash shell (and make sure bash ignores the '%' character) just use an alias to "undefine" the '%' character like this:

```
% alias %=""
```

All of the CS250 laboratory assignments should be completed on an EECS Instructional machine and you have setup your account according to the setup instructions given on the class website. Also, this lab guide assumes that you have already completed lab 1. If you have not completed lab 1, run the following commands, assuming that your username is `yunsup`.

```
% cd /scratch/
% mkdir yunsup
% cd yunsup
% git init
% git remote add template git@github.com:ucberkeley-cs250/lab-templates.git
% git remote add origin git@github.com:ucberkeley-cs250/yunsup.git
```

You will be using Git to manage your CS250 laboratory assignments. Please see *Tutorial 1: Using Git to Manage Source RTL* for more information on how to use SVN. Every student has their own directory in the repository which is not accessible to other students.

To begin the lab you will need to make use of the lab harness located on Github. The lab harness provides makefiles, scripts, and the Verilog test harness required to complete the lab. Assuming your username is `yunsup`, follow these steps to download the materials for this lab:

```
% cd /scratch/yunsup
% git pull template master
% git push origin master
% cd lab3
% LABROOT=$PWD
```

The resulting `lab3/` project directory looks similar to lab 2's working directory structure: `src` contains your source Chisel; `build` contains automated makefiles and scripts for building your design; `vlsi/csrc/` contains the Direct C source files to simulate memory and caches, parse and load ELF files; `vlsi/caches/` contains the VLSI cache models; `vlsi/lib/` and /texttttvlsi/include/ contain files needed to disassemble instruciton codes for debugging purposes; `vlsi/testbench/` contains the Verilog testbench; `emulator/testbench/` contains the emulator testbench; `riscv-tests` contains local test assembly programs; and `riscv-bmarks` contains local C benchmark programs.

The `src` directory contains the Verilog test harness and other Verilog modules you will need in this lab assignment. The files marked with `(empty)` are the files you need to fill in. Copy over your RISC-V v2 implementation from lab 2.

- `arbiter.scala` - Allows both caches to share the main memory
- `consts.scala` - Bit enumerations for control signals
- `cpu.scala` - Wrapper that holds control and datapath
- `ctrl.scala` - Control part of the RISC-V processor
- `dpath.scala` - Datapath part of the RISC-V processor
- `instructions.scala` - RISC-V instruction definition
- `memories.scala` - "Magic" instruction and data memory
- `top.scala` - Used by Chisel to instantiate design

Feel free to add additional files (such as a seperate component for your register file, ALU, and BTB.

The `build` directory contains the following subdirectories which you will use when building your chip.

- `vcs-sim-rtl` - RTL simulation using Synopsys VCS
- `dc-syn` - Synthesis using Synopsys Design Compiler
- `vcs-sim-gl-syn` - Post synthesis gate-level simulation using Synopsys VCS
- `icc-par` - Automatic placement and routing using Synopsys IC Compiler
- `vcs-sim-gl-par` - Post place and route gate-level simulation using Synopsys VCS
- `pt-pwr` - Power analysis using Synopsys PrimeTime PX

Each subdirectory includes its own makefile and additional script files. **If you end up adding more files, you will have to make modification to these script files as you push your design through the 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, place and route, do post place and route gate-level netlist simulation, and power analysis.

```
% cd $LABROOT/vlsi/build
% make pt-pwr
```

Remember that before you do power analysis, you need to send switching activity to pt-pwr. You can do this by running `make convert` in `vcs-sim-gl-par`.

Now go ahead and copy your lab 2's RISC-V processor implementation to the `src` directory. Then change your core design to cope with a more realistic memory interface. Remember you also need to implement an arbiter.

To compile the emulator, run the following instructions until you see a [passed] message:

```
% cd $LABROOT
% make emulator
```

Once this works, you will attempt to run RISC-V assembly tests on your processor to check for correct operation. To do this, run the following instructions:

```
% cd $LABROOT
% cd emulator
% make run-asm-tests
% make run-bmarks-test
```

Once your emulator passes all tests, move on to creating a Verilog implementation and pushing it through the flow.

```
% cd $LABROOT
% make vlsi
% cd vlsi/build/vcs-sim-rtl
% make run-asm-tests
% make run-bmarks-test
```

Guidelines for Lab Grading

Your final lab submission should pass all of the assembly tests and also be able to successfully run the globally installed benchmarks on both RTL simulation, post synthesis and post place and route gate-level netlist simulation. The most important metric for lab grading will be energy efficiency (energy/instruction). You will also develop your own analytic energy model by measuring energy consumption of assembly tests and benchmarks, and investigating instruction mixes of assembly tests and benchmarks (see end of document for lab questions).

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.

Tip 2: Deal With Synchronous Memory Reads

You have used combinational memories in the previous lab, where reads are asynchronous and writes are synchronous. That's why you were able to connect the output of the PC register to the instruction memory, and the ALU output to the data memory. However, the memory interface used in this lab implements synchronous reads. The signals to the memory should be ready before the rising clock edge. You need to connect the output of the PC mux to the instruction cache. This is also one of the reasons to separate out a memory stage so you can get the memory address and the store data calculated before the rising edge.

Also think carefully about how you plan to come out of reset. For the first clock edge out of reset, your instruction cache should see an address of 0.

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.

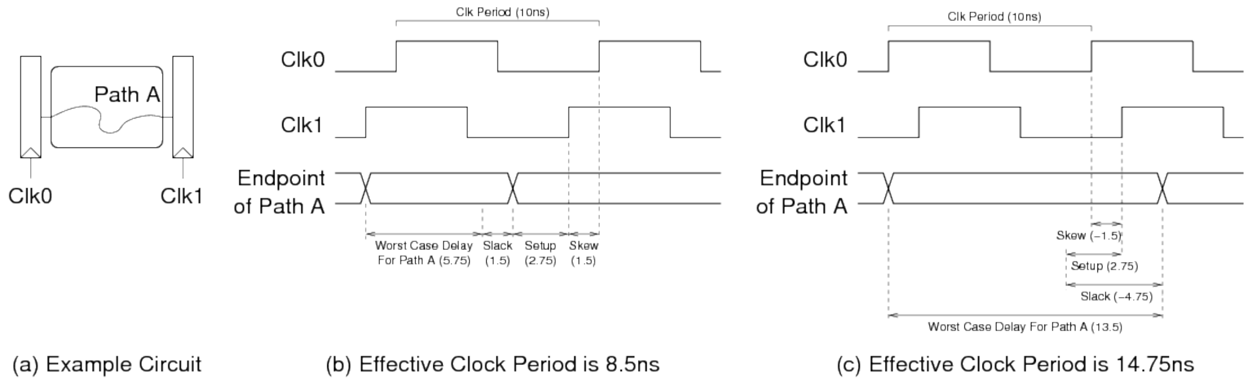

Figure 3: Determining your hardware's effective clock period

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 3 illustrates two examples: one with positive slack and one with negative slack. In this example, our clock period constraint is 10 ns. In Figure 3(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 3(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 3(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.

Tip 4: Useless Error Messages

Sometimes either the emulator or VLSI simulator will die horribly without an error message. The usual cause is an attempted memory access out of bounds. Check to make sure you addresses are being generated correctly and valid signals are only high when you intend.

Tip 5: Debugging

From Lab 2, you have correct program executation logs for all of the assembly tests. So when your tests are failing, you can compare steps of program executation until you spot a discrepency. The most useful lines to check are branches, writes to the register file (both address and value), and writes to the data cache (both address and value). If your program is incorrect, the cause will be most easily evident from these instructions and signals.

Make sure you also print out signals that tell you when pipeline stages are stalling or killed, and when branches are resolved taking. Also be careful to organize your debugging signals such that they are grouped by stage in the pipeline.

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 SVN. In addition, you should prepare written answers to the following questions and turn them in electronically.

Question 1: Draw a Block Diagram

Tell us how your `riscvCore` works in detail. How many pipeline stages did you use? Draw a block diagram and identify on the system diagram which components you placed in your datapath and which components you placed in your control logic.

Question 2: Take a Screenshot

Take a screenshot of your core and include it in your writeup. You should highlight the register file and the ALU with different colors. Point the caches from your design. Do you think pre-placement of the cache blocks would help minimize core area?

Question 3: Evaluate Your Baseline Core

Push your baseline processor through the physical toolflow and report the following numbers. (Summarize! don't just copy and paste.)

- Post-synthesis area of the register file, processor's datapath (excluding register file), processor's control unit, and cache blocks in um^2 from `*.mapped.area.rpt`
- Post-synthesis total area of core in um^2 from `*.mapped.area.rpt`
- Post-synthesis critical path and corresponding effective clock period in nanoseconds from `*.mapped.qor.rpt`
- Post-place+route area of the register file, processor's datapath (excluding register file), processor's control unit, and cache blocks in um^2 from IC Compiler `*.output.area.rpt`
- Post-place+route total area of the core in um^2 from IC Compiler `*.output.area.rpt`
- Post-place+route critical path and corresponding effective clock period in nanoseconds from IC Compiler `*.output.qor.rpt`

Your post-place+route numbers will probably be 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.

Question 4: Analytic Energy Model for your RISC-V v2 Core

For this question you are to calculate energy/instruction for assembly tests and benchmarks. Fill in the following table by counting individual instructions (instruction mix) and measuring power and performance. You should write a Python script which collects data from your build directories.

If you define the instruction mix as a matrix A, the energy/instruction numbers as x, and the consumed energy column as b then $Ax = b$ holds. Since you know A and b by filling up the table, you can guess what x should look like. Use MATLAB to run a regression to calculate x (energy/instruction). Ideally after this calculation you have your own power/energy model that you can plug into your simulator. Now you don't need to go through all the VLSI tools to get energy numbers out!

Provide answers to the following questions.

- List your energy/instruction numbers.
- Is there a natural grouping of energy/instruction numbers? Explain the grouping discpline. What instruction/group has the lowest energy/instruction? The highest energy/instruction? Why?
- How accurate is your model? Calculate the predicted energy consumption of benchmark programs. Compare this number to the measured energy consumption. What's the error?
- Identify the instruction mix of the benchmark program you wrote for lab 2. Run it through your energy model and compare the predicted energy consumption to the actual measured energy consumption. What's the error?
- Is the model accurate? How could you improve your energy model?

Program	Average Power	# of `addiw`	# of `addw`	... ...	# of `xor`	Target Time	Consumed Energy
`riscv-v1_simple.S`							
`riscv-v1_addiw.S`							
`riscv-v1_bne.S`							
`riscv-v1_lw.S`							
`riscv-v1_sw.S`							
`riscv-v2_addiw.S`							
`riscv-v2_addw.S`							
`riscv-v2_andi.S`							
`riscv-v2_and.S`							
`riscv-v2_beq.S`							
`riscv-v2_ble.S`							
`riscv-v2_bleu.S`							
`riscv-v2_blt.S`							
`riscv-v2_bltu.S`							
`riscv-v2_bne.S`							
`riscv-v2_j.S`							
`riscv-v2_jal.S`							
`riscv-v2_jalr.S`							
`riscv-v2_jalr_r.S`							
`riscv-v2_jalr_j.S`							
`riscv-v2_lui.S`							
`riscv-v2_lw.S`							
`riscv-v2_nor.S`							
`riscv-v2_ori.S`							
`riscv-v2_or.S`							
`riscv-v2_simple.S`							
`riscv-v2_slti.S`							
`riscv-v2_sltiu.S`							
`riscv-v2_slt.S`							
`riscv-v2_sltu.S`							
`riscv-v2_slliw.S`							
`riscv-v2_sllw.S`							
`riscv-v2_sraiw.S`							
`riscv-v2_sraw.S`							
`riscv-v2_srliw.S`							
`riscv-v2_srlw.S`							
`riscv-v2_subw.S`							
`riscv-v2_sw.S`							
`riscv-v2_xori.S`							
`riscv-v2_xor.S`							
`median`		N/A					
`qsort`		N/A					
`towers`		N/A					
`vvadd`		N/A					
`multiply`		N/A					

Question 4: Provide us Feedback

Please provide us feedback about the labs, tutorials, and experience using the VLSI tools.

Read me before you commit!

- For this lab, you don't need to commit any build results. We will build the design from your Chisel source files.
- We will checkout the stuff you committed to the `Github` and use that for lab grading. Feel free to take advantage of branches and tags. Use your late days wisely! If you have used your late days, please email TA.

Acknowledgements

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

- CS250 VLSI Systems Design (2009-2010) - University of California at Berkeley
- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CSE291 Manycore System Design (2009) - University of California at San Diego