Adding SRAMs to Your Accelerator

CS250 Laboratory 3 (Version 022016)
Written by Colin Schmidt
Modified by Christopher Yarp
Adapted from Ben Keller

Overview

In this lab, you will use the CAD tools and jackhammer to explore tradeoffs in the different implementation of your Sha3 Accelerator from Lab 2. For the first part of this assignment, you will use registers (flip-flops) to implement the storage for these buffers. Later, you will replace the flip-flop arrays with SRAM macros to produce a smaller and more energy efficient implementation.

Deliverables

This lab is due **Thursday, March 3rd at 11:59PM**. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (b) Reports (only!) generated by DC Compiler, IC Compiler, checked into your git repo for the three implementations of your design
- (c) written answers to the questions given at the end of this document checked into your git repository as `writeup/lab3-report.pdf` or `writeup/lab3-report.txt`

You are encouraged to discuss your design with others in the class, but you must write your own code and turn in your own work.

Getting the Lab Template

To get the lab template, run the following commands:

```
git pull template master
git submodule update --init --recursive
```

There are some changes to the system environment for this lab. For these changes to take effect, you will need to completely log off and log back into the hpse or icluster machines.

Compute Resource Changes

In lab 2, you used jackhammer to run jobs serially on one system. While this worked for running several designs through synthesis, it does not scale well as the complexity and length of the job increases. To address this, you will be using a compute cluster (icluster) with Torque PBS installed. This will require some additional setup on your part and is detailed later in this document in the *Using Jackhammer and Torque PBS* section. While jackhammer and the icluster are good resources, they are also limited. Currently, users are limited to 2 simultaneously running jobs on the icluster and need to share the 3 icluster compute nodes. Also, because of the amount of system the resources required for icc, few jobs are run simultaneously on each machine. This is because oversubscribing the machines can result in thrashing which degrades overall performance for all users. As a result, you may may submit jobs but will need to wait for other jobs to finish before yours running. This is a common experience with batch systems like the icluster with torque but may be unexpected if you are used to running commands interactively.

To maximize your productivity, it is recommended that you do most of your development work on *other* systems (like the hpse machines) and move to the icluster once you are ready to run jackhammer. Ideally, you should verify that your SRAM modification (discussed below) works as expected before spending the time and compute resources to sweep the design space. You can do this by interactively running the make commands as usual.

The VLSI Flow

In this lab you will be experimenting with SRAMs in your Sha3 design and seeing the results when pushed through all of the CAD tools. The full toolflow can be seen in figure Figure 3. In addition to using SRAMs we will be adding two new libraries of standard cells for the tools to use. The new tool we will be using is ICC. You can find a good reference on how these tool work and what steps the makefiles go through, below.

ICC

ICC (The IC Compiler) takes a synthesized netlist from DC and performs place-and-route (PAR) operations. ICC goes through several different stages including forming an initial floorplan, performing initial placement, building clock networks, routing between cells, optimizing the design, and fixing design rule violations. The scripts in the `build/vlsi/icc-par/icc_scripts` and `build/vlsi/setup/icc_setup.tcl` detail some of the settings and tasks used by icc.

You can run icc using the make target

```
make run-vlsi-par
```

Once icc finishes, you can view its reports in `build/vlsi/icc-par/current-icc/reports`.

You can also view the resulting design using the icc gui:

1. Run the commands:
   ```
   ssh -X localhost #this is a workaround to a display bug
   cd build/vlsi/icc-par/current-icc
   icc_shell -64bit -gui
   ```
2. Type the following into the tcl console in the gui
   `source icc_setup.tcl`
3. Navigate to File ⇒ Open Design
4. Click on Folder Next to Library Name
5. Select Sha3Accel_LIB
6. Select "Open library as read-only"
7. Select change_names_icc

ICC should open the layout window which should look something like Figure 1.

Multi-V_t Flow

Thus far in CS250, you have used standard cells from a single library to build your design. Now you will have an opportunity to take advantage a modern VLSI power-saving techinque: multi-V_t design. The Synopsys educational libraries provide different "flavors" of standard cells. In this lab, you'll be using both regular-V_t, high-V_t, and low-V_t standard cells. Recall that transistors with a higher threshold voltage are slower but leak less. If provided with multiple standard-cell libraries, the tools will use the faster low-V_t devices only on the critical path, and will swap in slower devices elsewhere to save power.

Take a look at the top-level `Makefrag` in the `build/vlsi`. By linking to each library and providing a few key commands, Design Compiler can take advantage of the different types of standard cells to better optimize the design. The `cells_*` variables determine which libraries the CAD tools are able to use. The given Makefrag shows how to use all three libraries at once but the evaluation requests that you limit the tools for one set of runs to see the advantages of a mult-vt setup.

Unfortunately, jackhammer cannot easily modify the Makefiles so you will need to do this modification manually. You can move to a single Vt, run jackhammer, and copy the reports to a working directory. After copying the results, you can restore the multi-Vt cells and run jackhammer again.

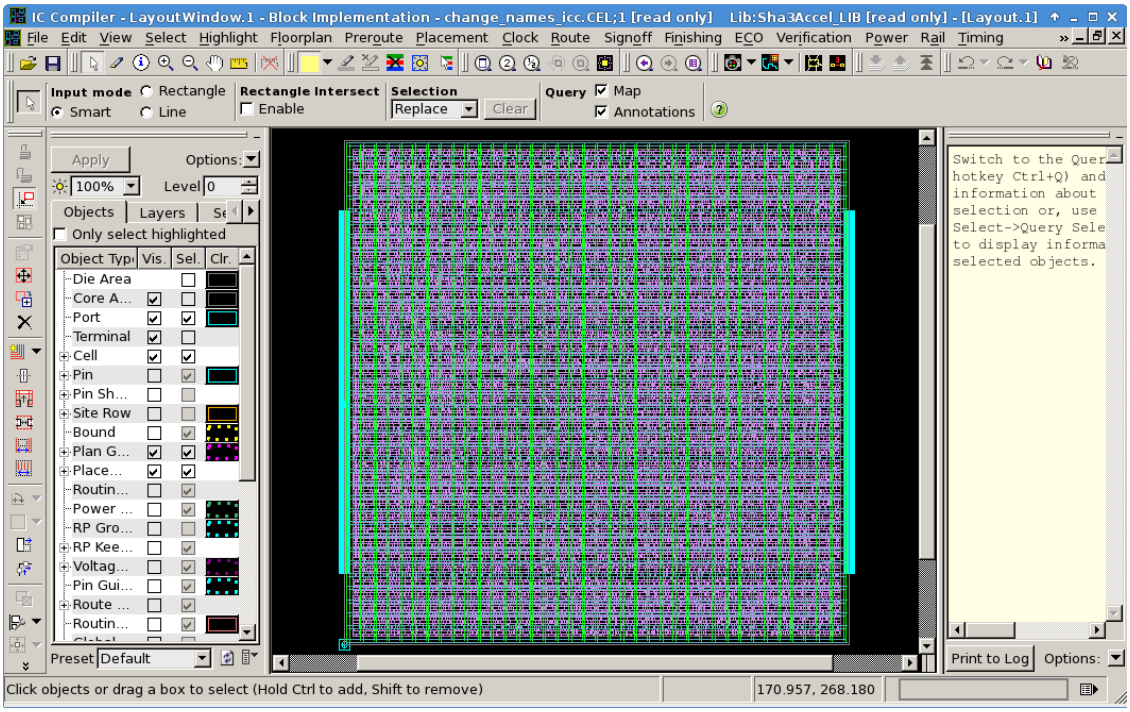

Figure 1: ICC Layout Window

SRAM

This section of the lab describes the general flow for having Chisel include SRAMs in your design. In this lab you are responsible for converting the absorb buffer from flip-flops to an SRAM macro.

Implementing your buffer as flip-flops introduces considerable overhead in area and energy. With some simple modifications, you can instantiate an SRAM instead.

Chisel's `Mem` class can be used to implement memories with arbitrary numbers of read and write ports. Writes always happen on the positive clock edge, but the timing of reads can be either combinational (result available during the same cycle as address is provided) or sequential (address is registered on positive clock edge, result available during the next cycle). However, the ASIC SRAMs we will be using in this course all use sequential reads. Therefore, to enable the Chisel compiler to map a `Mem`-generated memory to an ASIC SRAM macro (as opposed to an array of flip-flops), you must specify sequential read timing along with a register at the read address port. This is not exactly the style shown in the Chisel manual which specifies putting the register at the output. However, a couple experiments with this lab reveled that placing the register at the read port results in the proper inference of the SRAM. From your perspective, this should not matter much as the read latency is still one cycle.

Below is an example of how to instantiate a one read, one write memory (the kind you should use in your design). You should instantiate the memory for your absorb buffer in a similar fashion. You may want to make use of the buffer_sram parameter in ctrl.scala to allow jackhammer to produce designs that either have a register array or SRAM. For more about the `Mem` class, see Chapter 7 of the Chisel getting started guide or Chapter 9 of the Chisel manual.

```
val buffer_mem = Mem(UInt(width = W), round_size_words, seqRead = true)

val buffer_raddr = Reg(UInt(width = log2Up(round_size_words)))
val buffer_wen = Bool(); buffer_wen := Bool(false) //Default value
val buffer_waddr = UInt(width = W); buffer_waddr := UInt(0)
val buffer_wdata = UInt(width = W); buffer_wdata := UInt(0)
val buffer_rdata = Bits(width = W);
if(buffer_sram){
  when(buffer_wen) { buffer_mem.write(buffer_waddr, buffer_wdata) }
  buffer_rdata := buffer_mem(buffer_raddr)
}
```

You must instantiate 17-entry SRAM as sized for a `width` of 64, as we have only given you one SRAM macro to use in this design. The macro is called `sram8t17x64`; it is dual-ported, 64 bits wide, and 17 entries deep. Its library files and Verilog model (used for simulation) are located in the `generated-rams` directory. This macro was generated using Cacti, an SRAM modeling suite. Make sure you the memory you instantiate using `Mem` has the same depth and width at the SRAM macro.

As hint, the locations in `ctrl.scala` that need to be modified are locations with an empty `if(buffer_sram)` block. The modification to break absorption into multiple cycles has been done for you. Also, note that aindex (which is used as the index for absorbing into the state array) is buffered twice: once in `aindex` and once at `io.aindex`. You will need to select the correct version to feed into `buffer_raddr`. Also, during the padding operation you will need to read a value indexed with `pindex`. This means that you will need to change the index fed into `buffer_raddr` when in the `m_pad` stage. Also, padding unfortunately requires that a value be read from the SRAM, modified, and written back at the same location. The SRAM has a one cycle read latency so you will need to compensate for this. You will need to read the value (without writing) in once cycle, then perform the operations on the data and write it back in the next cycle. This means that you will need to increment `pindex` every other cycle instead of every cycle. You will also have to set `buffer_wen` to true only after the value was read from the SRAM. Introducing a new boolean register to keep track of when a read is occurring would probably be a good idea. This register can be updated in the same block that increments `pindex`. Finally, you need to keep in mind that, as a true dual ported SRAM, there are separate read and write address lines.

Checking the SRAM is Instantiated

If you use the `Mem` class in a way that does not imply a sequential memory, Chisel will just instantiate an array of flip-flops instead. To check that Chisel inferred the SRAM properly, run `make vlsi` and check the generated Verilog files. In `/build/vlsi/generated-src/Sha3Accel.DefaultConfig.v` you should see a section that looks like this:

```
sram8t17x64 sram(
  .CE1(CLK),
  .CSB1(~R1E),
  .OEB1(1'b0),
  .A1(R1A),
  .O1({R1O[63:0]}),
```

```
    .CE2(CLK),
    .CSB2(~W0E),
    .WEB2(1'b0),
    .A2(W0A),
    .I2({W0I[63:0]})
  );
```

If you do not see the SRAM, make sure that `case "buffer_sram" => true` in the knobValues block of `sha3.scala`.

Once you have successfully instantiated an SRAM, build your design with the provided flow.

After running dc, you can verify that the SRAM was used by looking at the `Sha3Accel.mapped.reference.rpt` report. You should see a reference to `sram8t17x64`.

You can also verify that the SRAM was placed in the design by looking at the icc generated layout. As Figure 2 shows, a rectangular SRAM block is placed in the design instead of registers.

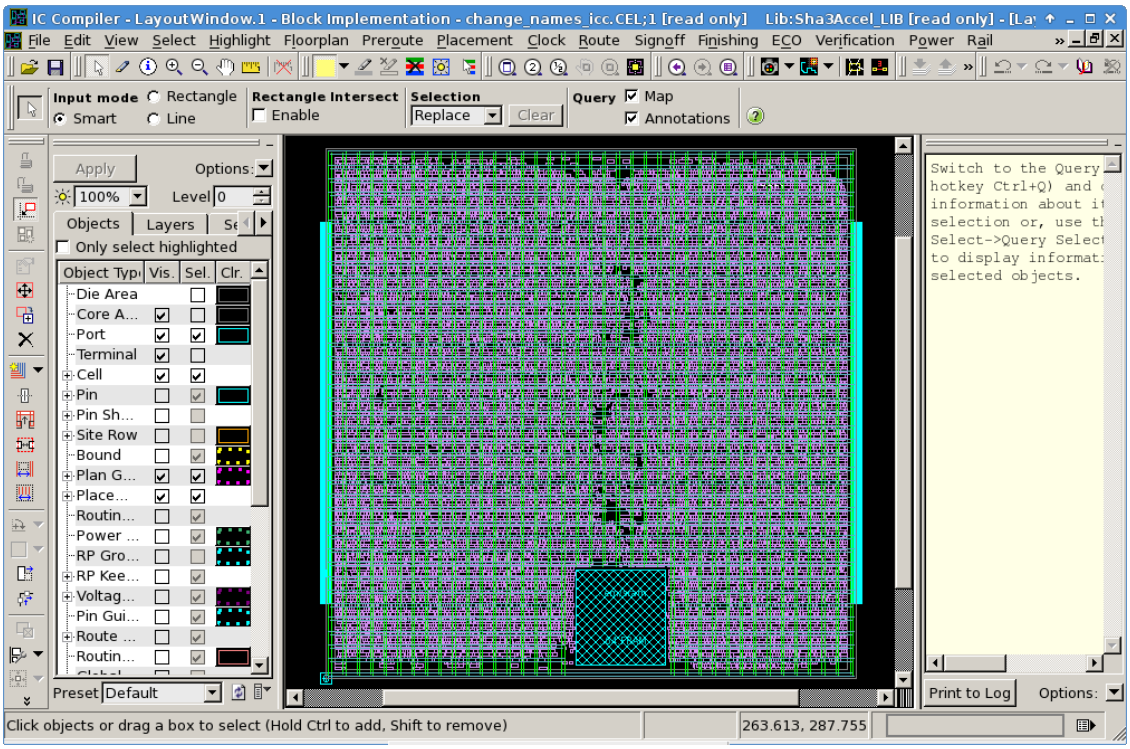

Figure 2: ICC Layout Window - With SRAM

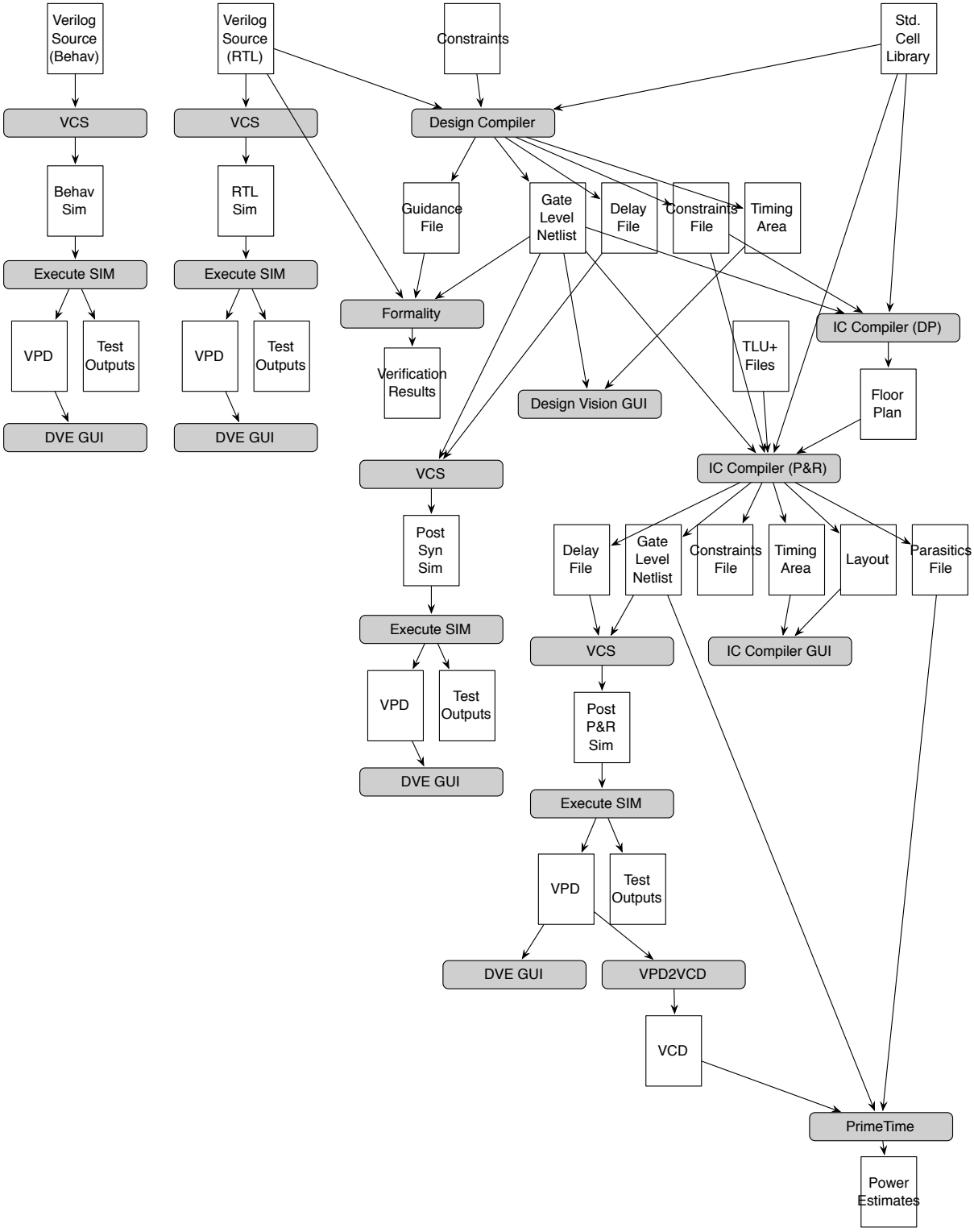

Figure 3: CS250 Toolflow for Lab 2

Using Jackhammer and Torque PBS

Torque is a job manager and scheduler that distributes jobs across multiple machines in a cluster. Instead of running commands interactivly from a terminal, jobs are submitted into a queue. The behavior for jobs is often contained within scripts that are provided with the submission The scheduler will automatically start a job when a machine becomes available. To help distribute resources on the cluster, users are limited to 2 simultaneously running jobs. Jackhammer has been modified for this lab so that each configuration is submitted as its own job. In order to use this new system, there are a couple steps you need to take.

Setup

The icluster machines currently only support x2go and not NX. To setup a connection to icluster1, follow these steps:

1. Download the x2go client if you do not already have it
2. Create a new session with icluster1.eecs.berkeley.edu as the hostname
3. Enter your cs250-## account name as the user account
4. Under session type, select "XFCE" (xfce is the only desktop manager that is working with x2go at the moment)
5. Click OK and connect (it may take a couple tries the first time)

The default configuration for xfce does not allow tab-complete. To fix this, follow these steps:

1. Navigate to Application Manu ⇒ Settings ⇒ Window Manager
2. Select the "Keyboard" tab
3. Select "Switch window for same application" - it should have 'tab' as the shortcut.
4. Press the "Clear" button.
5. Close the window

To allow the automated tools to easily login to other icluster machines, we need to create a ssh key with no password. However, you already have an ssh key for github which may have a password. To save your old key and generate a new one, follow these steps:

1. Run these commands to save your github key
   ```
   mv ~/.ssh/id_rsa ~/.ssh/github_rsa
   mv ~/.ssh/id_rsa.pub ~/.ssh/github_rsa.pub
   ```
2. Paste the following text into `~/.ssh/config` (including the tab on the second line) to continue using your old key for github
   ```
   Host github.com
       IdentityFile ~/.ssh/github_rsa
   ```

3. Run the command:
   `ssh-keygen -t rsa`
4. Select the default file (id_rsa)
5. When prompted for a password, just press enter (no password)
6. Run this command to place the new key into a list of authorized keys for login
   `cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys`

We need to setup the known hosts table with the thumbprints of each machine in the icluster. To do this, run `lab3/tools/prepareTorque.sh`.

Torque Allocation

icluster1 is serves as the head of the cluster. It runs the job scheduler that distributes jobs to the other icluster machines. Torque is not currently configured to allow submissions from other machines, only from icluster1. Since icluster1 is used as the user's gateway to the cluster, we sometimes refer to it as the "login" node. Since jobs must be submitted using icluster1, many people will be logged-in at once. As a result, you should refrain from running workload intensive tasks on it. Likewise, the other icluster machines are sometimes called "compute" nodes. While you can ssh into the other icluster machines, you should not run jobs on them interactively. This will upset Torque's resource allocation and will result in a poor experience for you and other users. Run all jobs on icluster through the Torque system.

Torque Commands

Jackhammer will automaticlaly submit jobs to the torque queue if it is run from icluster1. By default, jackhammer will evaluate all possible configurations that match the Config class in `sha3.scala`. Feel free to constrain knobs or remove them entirly if you would like to reduce the number of jobs jackhammer submits.

Running the commands from `lab3/jackhammer` will automatically submit jobs to torque:

```
make
make paraent
```

The torque command jackhammer uses is `qsub`.

Once the jobs are submitted, you can monitor them using `qstat`. `qstat -a` gives an alternative view and `qstat -n` will show the node that the job is running on. Using the numeric jobID, you can use `qstat -f jobID` to find details about a job. You can use `qpeek jobID` to peek into stdout and stderr of a running job. It does not matter that it is running on another system, `qpeek` will ssh in for you. Some examples of these commands and their corresponding outputs are below. In the 'S' column: 'C' means 'complete', 'R' means 'running', and 'Q' means 'queued'.

If you need to stop a job that is queued to run or is already running, you can use the `qdel jobID` command.

```
icluster1 [1470] /scratch/cs250-ta/dryrun/lab-templates-sltn/lab3/jackhammer # qstat
Job ID                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
346.icluster1              ...efaultConfig0 cs250-ta        01:29:48 C batch
347.icluster1              ...efaultConfig1 cs250-ta        01:33:00 R batch
348.icluster1              ...efaultConfig2 cs250-ta        00:01:02 R batch
349.icluster1              ...efaultConfig3 cs250-ta               0 Q batch
350.icluster1              ...efaultConfig4 cs250-ta               0 Q batch
351.icluster1              ...efaultConfig5 cs250-ta               0 Q batch


icluster1 [1472] /scratch/cs250-ta/dryrun/lab-templates-sltn/lab3/jackhammer # qstat -a

icluster1.EECS.Berkeley.EDU:
                                                                                  Req'd    Req'd       Elap
Job ID                  Username    Queue    Jobname          SessID  NDS   TSK   Memory   Time    S   Time
----------------------- ----------- -------- ---------------- ------ ----- ------ ------ --------- - ---------
346.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo  12293     1      2    --   03:00:00 C       --
347.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo   7994     1      2    --   03:00:00 R  00:53:00
348.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo  15890     1      2    --   03:00:00 R  00:03:40
349.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo    --      1      2    --   03:00:00 Q       --
350.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo    --      1      2    --   03:00:00 Q       --
351.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo    --      1      2    --   03:00:00 Q       --


icluster1 [1471] /scratch/cs250-ta/dryrun/lab-templates-sltn/lab3/jackhammer # qstat -n

icluster1.EECS.Berkeley.EDU:
                                                                                  Req'd    Req'd       Elap
Job ID                  Username    Queue    Jobname          SessID  NDS   TSK   Memory   Time    S   Time
----------------------- ----------- -------- ---------------- ------ ----- ------ ------ --------- - ---------
346.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo  12293     1      2    --   03:00:00 C       --
   icluster2+icluster2
347.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo   7994     1      2    --   03:00:00 R  00:51:52
   icluster3+icluster3
348.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo  15890     1      2    --   03:00:00 R  00:02:32
   icluster2+icluster2
349.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo    --      1      2    --   03:00:00 Q       --
    --
350.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo    --      1      2    --   03:00:00 Q       --
    --
351.icluster1.EECS.Ber  cs250-ta    batch    hammer-DefaultCo    --      1      2    --   03:00:00 Q       --


icluster1 [1474] /scratch/cs250-ta/dryrun/lab-templates-sltn/lab3/jackhammer # qstat -f 348
Job Id: 348.icluster1.EECS.Berkeley.EDU
    Job_Name = hammer-DefaultConfig2
    Job_Owner = cs250-ta@icluster1.EECS.Berkeley.EDU
    resources_used.cput = 00:20:45
    resources_used.mem = 2165460kb
    resources_used.vmem = 9040256kb
    resources_used.walltime = 00:11:21
    job_state = R
    queue = batch
    server = icluster1.EECS.Berkeley.EDU
```

Suggested Workflow

Because you are sharing the icluster with other students and are limited to running 2 jobs at a time, it is probably a good idea to debug and validate your design before moving on to jackhammer. You should be able to use the hpse machines as before to do your initial work. Once you are satisfied with your implementation, you should commit it to your git repository and clone it into the scratch of icluster1. Once there, you can use jachammer. Jackhammer will automatically run the c++

emulator, pre-synthesis simulation, dc, post synthesis simulation, icc, and post place and route simulation. You can change which jobs jackhammer runs by modifying `jackhammer/Settings.scala`. Removing items from the `override val qors =` line will cause jackhammer to skip those tasks.

Submission and Writeup

Write a script to collect the data necessary to fill in the following tables. Use the reports generated by ICC for area, power, and timing information.

You should use Jackhammer to run the experiments to generate the data. The given Jackhammer configuration will run the design through four qors with all design points, however this may not be the best configuration to start with and you should feel free to modify Settings.scala to better suit your needs.

You should create both tables for all 6 points in our design space, and include the bits hashed per second for the final test with each set of tables.

Area		RVT	Multi-VT	Multi-VT + SRAM
Sha3 Dpath	um^2			
Total	um^2			
Cell Count	LVT			
Cell Count	RVT			
Cell Count	HVT			

Power		RVT	Multi-VT	Multi-VT + SRAM
Entire Design				
Leakage	uW			
Total	uW			
datapath				
Leakage	uW			
Total	uW			

Questions: Multiple VT Flow

1. What impact does switching from a single VT to a multi-VT flow have on area? Does this match your expectations?

2. What effect does switching to a multi-VT have on the power consumption of your design? How much does it reduce leakage power?

3. Can you think of a reason why you wouldn't want to use multiple VT cells to implement a design?

4. What portion of the cells used in the multi-VT version of your design are regular VT? Does this match your expectations?

5. Where in your design do regular VT cells appear? Why do you think this is the case?

Questions: SRAM

1. How much area do you save by using an SRAM instead of registers to implement storage for your window buffer?

2. What is the area of the SRAM macro?

3. Assuming that the individual bitcells have an area of 0.415 um^2, what is the efficiency (area used for bitcells over total area) of this SRAM macro?

4. What are the other components of an SRAM, besides the bitcells?

Submission

To complete this lab, you should commit the following files to your private Github repository:

- Your working Chisel code.
- The `reports` directories from DC, and ICC.
- Your script.
- Your answers to the questions above, in a file called `writeup.txt` or `writeup.pdf`.

Some general reminders about lab submission:

- Please note in your writeup if you discussed or received help with the lab from others in the course. This will not affect your grade, but is useful in the interest of full disclosure.
- Please note in your writeup (roughly) how many hours you spent on this lab in total.

Acknowledgements

Parts of this lab, particularly the section on SRAMs, were originally written by Rimas Avizienis for CS250 Fall 2012 Lab 2. The `AdvTester` Chisel class and much of the test infrastructure for this lab were written by Stephen Twigg.