# SHA3: Pipelining and interfaces with Exploration

CS250 Laboratory 2 (Version 091014)
Written by Colin Scmidt
Portions based on previous work by Yunsup Lee
Updated by Brian Zimmer, Rimas Avizienis, Ben Keller

## Overview

For this lab you will be buliding on our previous implementation of Sha3 and tackling a few new important topics for accelerators. The first change you will make is to add a parameterized number of pipeline stages to the previous datapath. You will also flesh out the interface the accelerator has with other components in the system including the Rocket processor driving it and the memory system. The interface with the Rocket processor using the RoCC interface will be given to you, but you will be responsible for updating the memory interface. The given memory interface represents a low performance implementation and can be improved drastically.

In addition, to these updates to the Chisel RTL you will also be using another CAD tool, Design Compiler to synthesize your RTL into a gate-level netlist. In order to accomplish this synthesis across the parameterized design you have created we will introduce another Berkeley tool, Jackhammer, which will help us manage a design space exploration of our accelerator.

### Deliverables

This lab is due **Thursday, September 18 at 9:30AM**. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (c) build results and reports generated by Chisel C++, VCS, and DC checked into your git repo (results and reports only! No binaries!)
- (d) written answers to the questions given at the end of this document checked into your git repository as `writeup/report.pdf` or `writeup/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.

### VLSI Synthesis Introduction

Figure 1 illustrates the toolflow you will be using for the first lab. You will use Chisel to generate both Verilog and C++ versions of your design. The C++ versions will be used by Chisel to create an *emulator* of your circuit. Using the *tests* you write you can verify the functionality of your RTL without the use of any CAD tools. Once you are satisfied with the quality of you design you can then also use Chisel to generate a verilog implmentation of it. You will use Synopsys VCS (`vcs`) to *simulate* and *debug* your verilog RTL design. Both tools are capable of producing a more detailed debugging aid a *vpd* file. This extra detail comes at a slow down in simulation and is a less productive but sometimes necessary method. Another CAD tool Discovery Visualization Environment (`dve`) can read and display a waveform view of the circuits operation. After you

get your design right, you will use Synopsys Design Compiler (`dc_shell-xg-t`) to *synthesize* the design. Synthesis is the process of transforming RTL into a gate-level netlist.

The diagram below illustrates how the tools work together.

## Getting Started

The setup procedure mirrors that given in the first lab. Once you have pulled the latest template navigate to the `lab2` directory. This directory contains the following subdirectories: `src` contains your source Chisel; `build` contains several nested subdirectories as well. The `build/emulator` contains the generated files for simulating the Chisel code with the emulator. The `build/vlsi` contains a few directories to simulate your generated verilog(`vlsi/generate-src`), synthesize this verilog into a gate-level design(`vlsi/dc-syn`) and then simulate that gate level design(`vlsi/vcs-sim-gl-syn`).

### Baseline Design Overview

The baseline design for this lab has changed in several ways from the solution to Lab 1. The most imporant revolves around the new interface in the top level module `Sha3Accel`. This module now implements the RoCC interface. This means that the module is notified that it has a new hash two complete via two instructions set over the RoCC interface. The first instruction specifies the memory addresses for the message to be read from and the hash to be written too. The design waits until it has received both of these instructions at which point it becomes busy and begins reading from memory and hashing the message.

The second important change is the use of memory to obtain the message to be hashed. The baseline design has a simple memory system that sends out one request and waits for its reply before continuing.

Finally, a smaller change is that the accelerator now supports variable length messages and you are provided with the control to manage this as well as the control to padd the messages appropriately.

### SHA3 Datapath: Pipelining

The first part of this lab involves pipelining your design in a parameterized fashion. The idea is to decrease the critical path in order to increase performance. As we have seen in class this increase in performance can also be traded for a decrease in energy if we are also able to scale the voltage down.

The baseline design in this lab includes a new parameter `S` that corresponds to the number of pipeline stages to complete one round of the Sha3 permutation. This baseline design also includes an implementation of this parameter for `S=1` which was the design implemented for the previous lab. The baseline design also has a combined `rhopi` and so is designed to support `S={1,2,4}`. If you are starting from the given design these are the only values your finished implementation need support. If you are using your previous design with separate `rho` and `pi` modules then you should choose what set of values for `S` you should support but must support at least three distinct values.

The pipelining you will be implementing is slightly different than traditional pipelining because we are not adding additional state to keep track of multiple in-flight hashes at once. Rather your

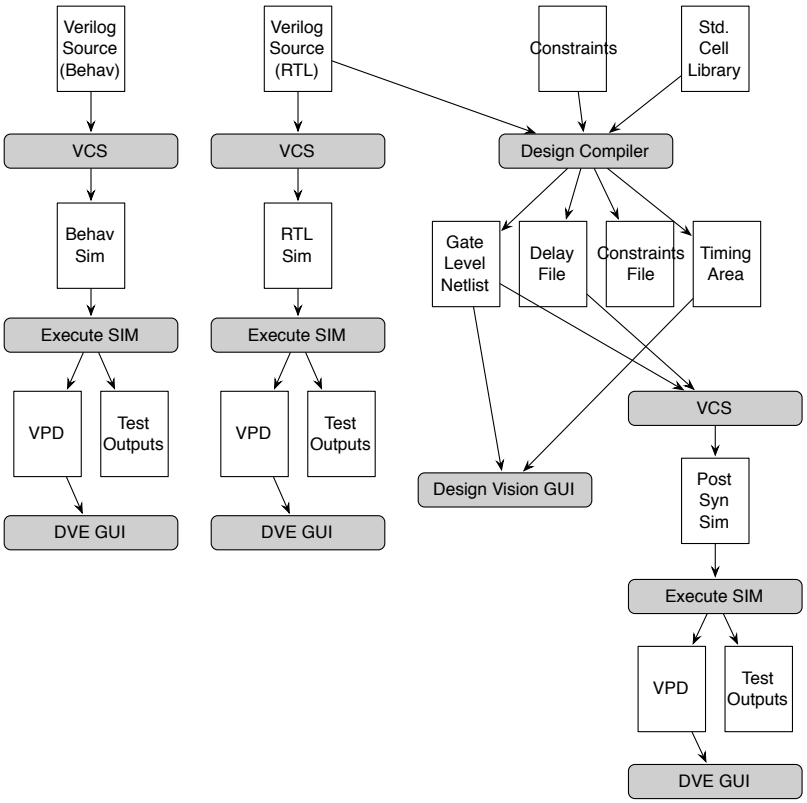


Figure 1: CS250 Toolflow for Lab 2

alteration will be more similar to a multi-cycle implementation, in which the larger operation has been broken up into smaller segements that still end in the same piece of state at the end of each cycle. The reasons for this are two fold. First, not having multiple in flight hashes significantly reduces the complexity of implementing this lab. And second, the additional state required to have multiple hashes in flight is significantly larger than the extra state in a processor for additional instructions in flight. Both of these reasons don't eliminate the potential usefullness of such a design and in fact in a later lab we will implement just such a modification to gauge its effectiveness.

**SHA3 Control: Memory Interface and RoCC Interfaces**

The second part of the lab mentioned before is to improve the baseline designs memory interface. The idea is that in order to improve performance for the entire accelerator all parts must be accelerated and the simple memory interface given in the baseline can be improved significantly.

Because this baseline design has added the ability to fetch the message from memory, this has become another portion of the design in need of optimization. In addition, the memory system we use can support multiple in-flight requests and so there is the potential to improve the baseline design by allowing it to send multiple requests before waiting for their response, hopefully increasing the bandwidth obtained from the memory system.

Your task in this lab is to do just that. You should update the memory portion of the accelerator to send as many reads from one message chunk as it can before waiting for any responses. Keep in mind that we are not able to put back-pressure on the memory system so your accelerator must always be able to handle the memory responses when they arrive.

**Testing**

The baseline design includes two tests. One mirrors the test given for Lab 1, which is intended to help validate that your design is functionally correct for a simple case. The second test is more of a benchmark in that it is designed to take longer and give you an idea the performance of your design on a longer hash.

The tests are still implemented in scala and run from scala but some infrastructure has been added to make it easier to develop tests for a RoCC accelerator that accesses memory. You can see this extra infrastructure in `test_infrastructure.scala`, and the given tests provide an example of how to use it. The infrastructure provides a model of memory as well as a simpler method of modeling the RoCC interface, other than poking all the individual bits. This encapsulation allows for a similar level of simplicity in our tests as we had in the first lab.

**SHA3 Chisel Testing**

In addition too the two methods of testing used in the previous lab, C++ and RTL level VCS, a new option has been added this lab which is post synthesis level verilog simulation. This last test is designed to be run after your design has been synthesised using Design Compiler (see below).

First, the Chisel compiler can produce C++ code which implements a cycle-accurate simulation of your design. To generate the C++ code, compile the simulator, and run the testbench, run the following commands:

```
% cd $LABROOT
% make emulator
% make run-emulator
```

In addition to a C++ description of a simulator, the Chisel compiler can also generate Verilog code that can be used as input to an ASIC flow.

```
% cd $LABROOT
% make vlsi
% make run-vlsi
```

The new tester for post synthesis gate level simulation is run from the Makefile with the following commands:

```
% cd $LABROOT
% make vlsi
% make run-vlsi-syn
```

**Exploring Design Space and Generating Results**

This section contains details on how to test different parameterization of your design, in an automated fashion, as well as how to collect results from synthesis and simulations automatically.

The tool we will be using to explore the implications have our pipelining and memory interface changes is a new tool developed here at Berkeley called Jackhammer. There is an introduction to the parameter system of JackHammer available on the course website, and the remaining features are still being documented but this lab should give you enough background to use the tool for the lab.

The first step to using Jackhammer is to parameterize your design using the params object, and give them top level definitions This is already done for you in the given code, see the top of sha3.scala and topDefinitions at the bottom. You should note that the parameters we intend to explore across must be defined as Knobs. In order for Jackhammer to know what values parameters can take we also need to provide it with constraints on the parameters. This can be seen in the example code at the end of sha3.scala in the topConstraints list. You can see that we have constrained stages to be between 1 and 4, and divisible by 2 or equal to 1. Jackhammer will use a constraint solver to determine what values it can fill in for the parameters.

Once we have defined our paramters we can use Jackhammer to generate a few files to show its understanding of the parameter space.

```
% cd $LABROOT/jackhammer
% make
```

This generates a directory inside of the main scala directory called `config`. This directory contains three files, a `.cst` file listing the constraints that jackhammer has discovered from our definitions. The `.knb` file contains the names of the knobs in our design. Finally, the most important file for us is the `.scala` file which contains the names of each of the valid configurations. In our case, this file only contains three points, which is what we expected, each one setting stages to a different directory.

With this file we can now setup what we want Jackhammer to do for us. If you open up `jackhammer/IclusterSettings.scala` you can see a bunch of boilerplate settings. The most important things to note about this file are the scripts variables, the qors list and the designs list. Since we have configured Jackhammer to use this settings file it will use these variables to determine what it does. Explicitly Jackhammer will, for each design configuration in the designs list, execute the execute script and then the parse script for each of the listed qors. Jackhammer is setup on the Icluster to be able to submit these runs in parallel on any of the icluster machines our class is able to use. It will use the machine you start the run from as the base machine where it will collate the results. It will create a folder in the scratch directory `/scratch/cs250-aa/hammer` which it will use to stage the different run and collect the results to. You should read over the scripts in the `scripts` directory to see what they are doing, in future labs you may be asked to modify them.

Now that you understand what Jackhammer will do it is time to have it start. Before you begin there is one more icluster setup step to do. You should create a file in your home directory with a list of all the icluster servers we are using.

```
% cat ~/.rhosts
icluster12.EECS.Berkeley.EDU
icluster13.EECS.Berkeley.EDU
icluster14.EECS.Berkeley.EDU
icluster15.EECS.Berkeley.EDU
icluster16.EECS.Berkeley.EDU
```

This ensures Jackhammer can communicate with all of the icluster servers.

Finally to run jackhammer you can execute the following commands:

```
% cd $LABROOT/jackhammer
% make parent
```

After running this command Jackhammer will submit the jobs to the servers, which you can monitor as follows.

```
% qstat -u cs250-aa
```

Once the jobs are done you will see several files in the `jackhammer` directory that show the `stderr` and `stdout` of each job. You can also investigate the results of the execute and parse scripts in the `/scratch/cs250-aa/hammer/sha3/{outputs,results}` directories. You should copy the results files of your last run into the `lab2/writeup` directory and commit them for grading.

**Synopsys Design Compiler: RTL to Gate-Level Netlist**

You can choose to experiment with Design Compiler before, during, or after you implement the RTL changes. Either way the steps we take will be the same. The instructions below are designed to show you the steps the makefile must execute in order to synthesize your design, and are useful to gain a better understand of how the tools work.

This section will also be updated later with more specific information with regards to Sha3 rather than GCD.

Design Compiler performs hardware synthesis. A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as an output. The resulting gate-level netlist is a completely structural description of the design, with only standard cells (and later on, possibly SRAM macros) as leaves in the design hierarchy. To cut and past commands from this lab into your Design Compiler shell and make sure Design Compiler ignores the `dc_shell-topo>` string, we will use an alias to "undefine" the `dc_shell-topo>` string.

```
% cd $LABROOT/build/vlsi/dc-syn
% dc_shell-xg-t -64bit -topographical_mode
...
Initializing...
alias "dc_shell-topo>" ""
```

You will now execute some commands to setup your environment.

```
dc_shell-topo> set ucb_vlsi_home [getenv UCB_VLSI_HOME]
dc_shell-topo> set stdcells_home \
  $ucb_vlsi_home/stdcells/synopsys-32nm/typical_rvt
dc_shell-topo> set_app_var search_path \
  "$stdcells_home/db $ucb_vlsi_home/install/vclib ../generated-src"
dc_shell-topo> set_app_var target_library "cells.db"
dc_shell-topo> set_app_var synthetic_library "dw_foundation.sldb"
dc_shell-topo> set_app_var link_library "* $target_library $synthetic_library"
dc_shell-topo> set_app_var alib_library_analysis_path "alib"
dc_shell-topo> set_app_var mw_logic1_net "VDD"
dc_shell-topo> set_app_var mw_logic0_net "VSS"
dc_shell-topo> create_mw_lib -technology $stdcells_home/techfile/techfile.tf \
  -mw_reference_library $stdcells_home/mw/cells.mw "Sha3Accel_LIB"
dc_shell-topo> open_mw_lib "Sha3Accel_LIB"
dc_shell-topo> check_library
dc_shell-topo> set_tlu_plus_files \
  -max_tluplus $stdcells_home/tluplus/max.tluplus \
  -min_tluplus $stdcells_home/tluplus/min.tluplus \
  -tech2itf_map $stdcells_home/techfile/tech2itf.map
dc_shell-topo> check_tlu_plus_files
dc_shell-topo> define_design_lib WORK -path "./work"
dc_shell-topo> set_svf "Sha3Accel.svf"
```

These commands point to your Verilog source directory, create a Synopsys work directory, and point to the standard libraries you will be using for the class. The `set_svf` command is used to set

up a guidance file which is used by Synopsys Formality. Now you can load your Verilog design in to Design Compiler with the `analyze`, `elaborate`, `link`, and `check_design` commands.

```
dc_shell-topo> analyze -format verilog \
  "Sha3Accel.v"
dc_shell-topo> elaborate "Sha3Accel"
dc_shell-topo> link
dc_shell-topo> check_design
```

Before you can synthesize your design, you must specify some constraints; most importantly you must tell the tool your target clock period. The following command tells the Design Compiler that the pin named `clk` is the clock and that your target clock period is 1 nanosecond.

```
dc_shell-topo> create_clock clk -name ideal_clock1 -period 1
```

Now you are ready to use the `compile_ultra` command to actually synthesize your design into a gate-level netlist. `-no_autoungroup` is specified in order to preserve the hierarchy during syntehsis.

```
dc_shell-topo> compile_ultra -gate_clock -no_autoungroup
 ...
Beginning Delay Optimization
 ----------------------------
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
   0:01:42   57293.5      0.00       0.0       0.0
 ...
```

The `compile_ultra` command will report how the design is being optimized. You should see Design Compiler performing technology mapping, delay optimization, and area reduction. The fragment from `compile_ultra` above shows the worst negative slack which indicates how much room there is between the critical path in your design and your specified clock constraint. Larger negative slack values are worse since this means that your design is missing the desired clock frequency by a great amount. Total negative slack is the sum of all negative slack summed over all the endpoints (register inputs or top level input/output ports) in the design.

Now you can generate the guidance information required by the formal verification tool, and produce the synthesized gate-level netlist and derived constraints files.

```
dc_shell-topo> set_svf -off
dc_shell-topo> change_names -rules verilog -hierarchy
dc_shell-topo> write -format ddc -hierarchy -output Sha3Accel.mapped.ddc
dc_shell-topo> write -f verilog -hierarchy -output Sha3Accel.mapped.v
dc_shell-topo> write_sdf Sha3Accel.mapped.sdf
dc_shell-topo> write_sdc -nosplit Sha3Accel.mapped.sdc
```

```
dc_shell-topo> write_milkyway -overwrite -output "Sha3Accel_DCT"
dc_shell-topo> source ./dc_scripts/find_regs.tcl
dc_shell-topo> find_regs test/Sha3Accel
```

Take a look at various reports describing the synthesis results.

```
dc_shell-topo> report_timing -transition_time -nets -attributes -nosplit
...
Point                                          Fanout     Trans      Incr       Path      Attri
-------------------------------------------------------------------------------------------------
clock ideal_clock1 (rise edge)                                       0.00       0.00
clock network delay (ideal)                                          0.00       0.00
ctrl/hashed_reg_11_/CLK (DFFX2_RVT)                        0.00      0.00       0.00 r
ctrl/hashed_reg_11_/Q (DFFX2_RVT)                          0.03      0.10       0.10 f
ctrl/T115[11] (net)                              6                   0.00       0.10 f
ctrl/U104/Y (INVX2_RVT)                                    0.02      0.01 *     0.11 r
ctrl/n12 (net)                                   3                   0.00       0.11 r
ctrl/U12/Y (NAND2X0_RVT)                                   0.02      0.02 *     0.13 f
ctrl/n8 (net)                                    1                   0.00       0.13 f
ctrl/U13/Y (NAND2X0_RVT)                                   0.02      0.02 *     0.16 r
ctrl/n9 (net)                                    1                   0.00       0.16 r
ctrl/U14/Y (OA22X1_RVT)                                    0.02      0.04 *     0.20 r
ctrl/n15 (net)                                   2                   0.00       0.20 r
ctrl/U19/Y (AO22X1_RVT)                                    0.02      0.05 *     0.24 r
ctrl/n20 (net)                                   1                   0.00       0.24 r
ctrl/U70/Y (OA221X1_RVT)                                   0.02      0.06 *     0.30 r
ctrl/n53 (net)                                   2                   0.00       0.30 r
ctrl/U83/Y (AO221X1_RVT)                                   0.04      0.07 *     0.37 r
ctrl/n106 (net)                                  1                   0.00       0.37 r
ctrl/U85/Y (OA221X1_RVT)                                   0.03      0.06 *     0.43 r
ctrl/n107 (net)                                  1                   0.00       0.43 r
ctrl/U79/Y (OR3X1_RVT)                                     0.02      0.04 *     0.47 r
ctrl/T62 (net)                                   1                   0.00       0.47 r
ctrl/U1253/Y (AND2X1_RVT)                                  0.02      0.03 *     0.50 r
ctrl/n1324 (net)                                 2                   0.00       0.50 r
ctrl/U535/Y (NAND2X4_RVT)                                  0.02      0.05 *     0.55 f
ctrl/io_absorb_BAR (net)                        13                   0.00       0.55 f
ctrl/U3947/Y (AND2X2_RVT)                                  0.03      0.05 *     0.61 f
ctrl/io_round[4] (net)                           6                   0.00       0.61 f
ctrl/io_round[4] (CtrlModule)                                        0.00       0.61 f
ctrl_io_round[4] (net)                                               0.00       0.61 f
dpath/io_round[4] (DpathModule)                                      0.00       0.61 f

dpath/iota/io_round[4] (IotaModule)                                  0.00       0.61 f
dpath/iota/io_round[4] (net)                                         0.00       0.61 f
dpath/iota/U5/Y (INVX4_RVT)                                0.02      0.02 *     0.62 r
dpath/iota/n29 (net)                             8                   0.00       0.62 r
dpath/iota/U23/Y (AND2X1_RVT)                              0.02      0.03 *     0.65 r
```

```
dpath/iota/n13 (net)                          2                0.00       0.65 r
dpath/iota/U25/Y (AO22X1_RVT)                           0.02     0.04 *     0.70 r
dpath/iota/n10 (net)                          1                0.00       0.70 r
dpath/iota/U26/Y (AO22X1_RVT)                           0.02     0.05 *     0.75 r
dpath/iota/n11 (net)                          1                0.00       0.75 r
dpath/iota/U28/Y (NAND3X4_RVT)                          0.03     0.07 *     0.82 f
dpath/iota/n12 (net)                          1                0.00       0.82 f
dpath/iota/U29/SO (HADDX1_RVT)                          0.02     0.07 *     0.88 r
dpath/iota/io_state_o_0[15] (net)             1                0.00       0.88 r
dpath/iota/io_state_o_0[15] (IotaModule)                         0.00       0.88 r
dpath/iota_io_state_o_0[15] (net)                                0.00       0.88 r
dpath/U2349/Y (AO22X1_RVT)                              0.02     0.05 *     0.93 r
dpath/N41 (net)                               1                0.00       0.93 r
dpath/state_0_reg_15_/D (DFFX2_RVT)                     0.02     0.00 *     0.93 r
data arrival time                                                           0.93

clock ideal_clock1 (rise edge)                                   1.00       1.00
clock network delay (ideal)                                      0.00       1.00
dpath/state_0_reg_15_/CLK (DFFX2_RVT)                            0.00       1.00 r
library setup time                                              -0.03       0.97
data required time                                                          0.97
-------------------------------------------------------------------------------------
data required time                                                          0.97
data arrival time                                                          -0.93
-------------------------------------------------------------------------------------
slack (MET)                                                                 0.04

...
```

This report shows the *critical path* of the design. The critical path has the longest propagation delay between any two registers in the design and therefore sets an upper bound on the design's operating frequency. In this report, we see that the critical path begins at the output of bit 2 of the operand A register in the datapath, goes several subtractors in the datapath, and ends at output register 15. The critical path takes a total of 0.91ns which is less than the 1ns clock period constraint. This is reflected by the final line declaring that the positive slack has met timing.

```
dc_shell-topo> report_area -nosplit -hierarchy
...
                                Global cell area           Local cell area
                                -------------------  -------------------------------
Hierarchical cell               Absolute    Percent  Combi-      Noncombi-   Black
                                Total       Total    national    national    boxes   Desig
------------------------------  ----------  -------  ----------  ----------  ------  -----
Sha3Accel                       59792.2057    100.0    507.7797      0.0000  0.0000  Sha3A
ctrl                            19785.8732     33.1   9841.7265   9780.4779  0.0000  CtrlM
...
dpath                           39498.5528     66.1   8319.4041  10953.0984  0.0000  Dpath
...
```

```
--------------------------------  ----------  -------  ----------  ----------  ------  ------
Total                                                  38883.2700  20908.9357  0.0000
...
```

This report tells you about the post synthesis area results. The units are $um^2$. You can see that the datapath accounts for 91.8% of the total chip area.

```
dc_shell-topo> report_power -nosplit -hier
...
--------------------------------------------------------------------------------
                                    Switch   Int      Leak     Total
Hierarchy                           Power    Power    Power    Power    %
--------------------------------------------------------------------------------
Sha3Accel                           2.48e+03 1.09e+04 8.11e+09 2.14e+04 100.0
  dpath (DpathModule)               1.36e+03 5.25e+03 5.47e+09 1.21e+04  56.3
    iota (IotaModule)                  1.246    1.510 1.11e+07   13.851   0.1
    ChiModule (ChiModule)              0.000    0.000 1.05e+09 1.05e+03   4.9
    RhoPiModule (RhoPiModule)          0.000    0.000    0.000    0.000   0.0
    ThetaModule (ThetaModule)          0.000    0.000 1.52e+09 1.52e+03   7.1
  ctrl (CtrlModule)                 1.13e+03 5.60e+03 2.60e+09 9.33e+03  43.5
...
```

This report contains post synthesis power estimates. The dynamic power units are $uW$ while the leakage power units are $pW$.

```
dc_shell-topo> report_reference -nosplit -hierarchy
...
****************************************
Design: DpathModule
****************************************
Reference          Library       Unit Area   Count    Total Area   Attributes
--------------------------------------------------------------------------------
AND2X1_RVT         saed32rvt_tt1p05v25c     2.033152     512  1040.973877
AND3X4_RVT         saed32rvt_tt1p05v25c     3.049728       1     3.049728
AO22X1_RVT         saed32rvt_tt1p05v25c     2.541440    1088  2765.086731
ChiModule                      9895.605033       1   9895.605033  h
DFFX1_RVT          saed32rvt_tt1p05v25c     6.607744     851  5623.190329 n
DFFX2_RVT          saed32rvt_tt1p05v25c     7.116032     749  5329.908061 n
HADDX1_RVT         saed32rvt_tt1p05v25c     3.303872    1082  3574.789621 r
HADDX2_RVT         saed32rvt_tt1p05v25c     3.812160       6    22.872960 r
INVX2_RVT          saed32rvt_tt1p05v25c     1.524864       1     1.524864
IotaModule                      112.839937       1    112.839937  h
NAND2X0_RVT        saed32rvt_tt1p05v25c     1.524864       3     4.574592
NBUFFX2_RVT        saed32rvt_tt1p05v25c     2.033152     306   622.144544
NBUFFX4_RVT        saed32rvt_tt1p05v25c     2.541440      50   127.072001
NBUFFX8_RVT        saed32rvt_tt1p05v25c     3.812160      35   133.425601
NBUFFX16_RVT       saed32rvt_tt1p05v25c     6.099456       3    18.298368
NOR2X0_RVT         saed32rvt_tt1p05v25c     2.541440       1     2.541440
NOR4X1_RVT         saed32rvt_tt1p05v25c     3.049728       1     3.049728
```

```
RhoPiModule                          0.000000       1      0.000000  h
SNPS_CLOCK_GATE_HIGH_DpathModule_0     5.845312       1     5.845312 b, h
SNPS_CLOCK_GATE_HIGH_DpathModule_1     5.845312       1     5.845312 b, h
ThetaModule                     10205.914785       1  10205.914785  h
--------------------------------------------------------------------------------
Total 21 references                                   39498.552822
...
```

This report lists the standard cells used to implement each module. The `gcdGCDUnitDpath` module consists of 13 `AO22X1` cells, 32 `DFFARX1` cells, and so on. The total area consumed by each type of cell is also reported.

```
dc_shell-topo> report_resources -nosplit -hierarchy
...
Resource Report for this hierarchy in file ../generated-src/Sha3Accel.v
===============================================================================
| Cell           | Module         | Parameters | Contained Operations       |
===============================================================================
| add_x_1        | DW01_inc       | width=5    | add_1133                   |
| lte_x_2        | DW_cmp         | width=64   | lte_1170                   |
| ash_3          | DW_leftsh      | A_width=4  | sll_1194                   |
|                |                | SH_width=2 |                            |
| sub_x_4        | DW01_dec       | width=2    | sub_1197                   |
| lte_x_5        | DW_cmp         | width=5    | lte_1200                   |
...
| lte_x_49       | DW_cmp         | width=5    | lte_1954                   |
| add_x_50       | DW01_inc       | width=3    | add_2049                   |
| add_x_56       | DW01_add       | width=64   | add_2080                   |
| add_x_59       | DW01_add       | width=5    | add_2091                   |
| lt_x_338       | DW_cmp         | width=64   | lt_1241 lt_1280            |
| lt_x_339       | DW_cmp         | width=64   | lt_1294 lt_1341 lt_2098    |
| lt_x_340       | DW_cmp         | width=5    | lt_1257 lt_2097            |
| lte_x_341      | DW_cmp         | width=5    | lte_1282 lte_1288          |
| lte_x_342      | DW_cmp         | width=5    | lte_1267 lte_1306 lte_1307 |
| add_x_7        | DW01_add       | width=32   | add_1248 add_1251 add_1302 |
| add_x_8        | DW01_inc       | width=5    | add_1263 add_1264 add_1355 |
| DP_OP_467J2_124_1151           |            |                            |
|                | DP_OP_467J2_124_1151 |      |                            |
===============================================================================

...

Implementation Report
===============================================================================
|                    |                  | Current            | Set            |
| Cell               | Module           | Implementation     | Implementation |
===============================================================================
| add_x_1            | DW01_inc         | apparch (area)     |                |
```

```
| lte_x_2            | DW_cmp           | apparch (area)     |                |
| ash_3              | DW_leftsh        | astr (area)        |                |
| sub_x_4            | DW01_dec         | apparch (area)     |                |
| lte_x_5            | DW_cmp           | apparch (area)     |                |
| add_x_6            | DW01_add         | pparch (area,speed)                 |
| add_x_9            | DW01_inc         | apparch (area)     |                |
| lt_x_10            | DW_cmp           | apparch (area)     |                |
| lt_x_11            | DW_cmp           | pparch (speed)     |                |
...
| lt_x_340           | DW_cmp           | apparch (area)     |                |
| lte_x_341          | DW_cmp           | apparch (area)     |                |
| lte_x_342          | DW_cmp           | apparch (area)     |                |
| add_x_7            | DW01_add         | pparch (speed)     |                |
| add_x_8            | DW01_inc         | apparch (area)     |                |
| DP_OP_467J2_124_1151                  |                    |                |
|                    | DP_OP_467J2_124_1151 | str (area)     |                |
================================================================================

...
```

Synopsys provides a library of commonly used arithmetic components as highly optimized building blocks. This library is called Design Ware and Design Compiler will automatically use Design Ware components when it can. This report can help you determine when Design Compiler is using Design Ware components. The `DW01_sub` in the module name indicates that this is a Design Ware subtractor. This report also gives you what type of architecture it used.

You can use makefiles and scripts to help automate the process of synthesizing your design. Type `exit` to leave the DC shell. Then type:

```
% cd $LABROOT/build/dc-syn
% make
```

Go ahead and take a look what the automated build system produced.

```
% cd $LABROOT/build/dc-syn
% ls -l
total 20
drwx------ 3 cs250 cs250 4096 Sep 12 06:13 alib
drwxr-xr-x 7 cs250 cs250 4096 Sep 16 14:18 build-dc-01.250ns-2014-09-16_14-13
lrwxrwxrwx 1 cs250 cs250   34 Sep 16 14:13 current-dc -> build-dc-01.250ns-2014-09-16_14-13
drwx------ 2 cs250 cs250 4096 Sep 12 06:13 dc_scripts
-rw------- 1 cs250 cs250 1062 Sep 12 06:13 Makefile
-rw------- 1 cs250 cs250 3963 Sep 12 06:13 Makefrag
lrwxrwxrwx 1 cs250 cs250    8 Sep 12 06:13 setup -> ../setup
% cd current-dc
% ls -l
total 856
-rw-r--r-- 1 cs250 cs250     37 Sep 16 14:18 access.tab
-rw-r--r-- 1 cs250 cs250 370697 Sep 16 14:18 command.log
```

```
-rw------- 1 cs250 cs250   5139 Sep 16 14:13 common_setup.tcl
-rw------- 1 cs250 cs250   3632 Sep 16 14:13 constraints.tcl
-rw-r--r-- 1 cs250 cs250     29 Sep 16 14:18 dc
-rw------- 1 cs250 cs250   4625 Sep 16 14:13 dc_setup_filenames.tcl
-rw------- 1 cs250 cs250   4914 Sep 16 14:13 dc_setup.tcl
-rw------- 1 cs250 cs250   4612 Sep 16 14:13 dc.tcl
-rw------- 1 cs250 cs250   2720 Sep 16 14:13 find_regs.tcl
-rw-r--r-- 1 cs250 cs250 420655 Sep 16 14:18 force_regs.ucli
drwxr-xr-x 2 cs250 cs250   4096 Sep 16 14:13 log
-rw------- 1 cs250 cs250   3686 Sep 16 14:13 Makefrag
-rw-r--r-- 1 cs250 cs250   1387 Sep 16 14:13 make_generated_vars.tcl
drwxr-xr-x 2 cs250 cs250   4096 Sep 16 14:18 reports
drwxr-xr-x 2 cs250 cs250   4096 Sep 16 14:18 results
drwxr-xr-x 2 cs250 cs250   4096 Sep 16 14:13 Sha3Accel_LIB
-rw-r--r-- 1 cs250 cs250     29 Sep 16 14:13 timestamp
drwxr-xr-x 2 cs250 cs250   4096 Sep 16 14:13 WORK
```

Notice that the Makefile does not overwrite build directories. It creates a new build directory every time you run make. This makes it easy to change your synthesis scripts or source Verilog, resynthesize your design, and compare your results to those from an earlier design. You can use symlinks to keep track of various build directories. Inside the `current-dc` directory, you can see all the tcl scripts as well as the directories named `results` and `reports`. `results` contains your synthesized gate-level netlist, and `reports` contains various post synthesis reports.

Synopsys provides a GUI front-end for Design Compiler called Design Vision which you will use to analyze the synthesis results. You should avoid using the GUI to actually perform synthesis since scripting the process is more efficient. Start Design Vision, and open the .ddc file to load your synthesized design. (.ddc is a proprietary binary format used by Synopsys to encapsulate all post-synthesis data.)

```
% cd $LABROOT/build/dc-syn/current-dc
% design_vision-xg -64bit
...
Initializing...
design_vision> alias "design_vision>" ""
design_vision> source dc_setup.tcl
design_vision> read_file -format ddc "results/Sha3Accel.mapped.ddc"
```

You can browse your design with the hierarchical view (see Figure 2). If you right click on a module and select the *Schematic View* option, the tool will display a schematic view of the standard cells used to implement that module.

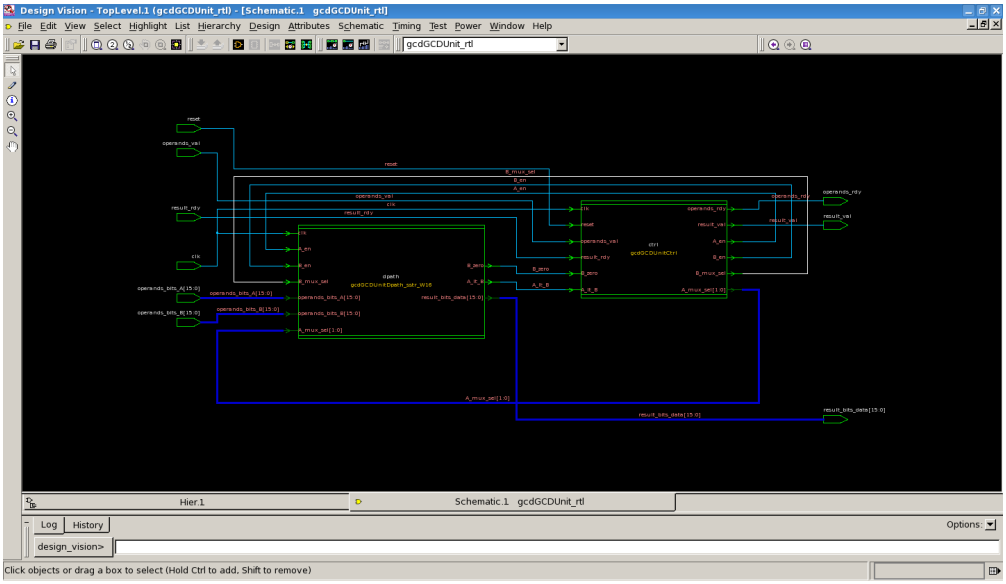


Figure 2: Design Vision Hierarchical View

## Questions

Your writeup should not exceed one page in length. Make your writing as crisp as you can! These questions are intended to be thought provoking rather than work intensive.

### Q1. Chisel and Jackhammer

- Do you feel the combination of Chisel and Jackhammer helped you for this lab?
- Are there features you would like to see in Chisel and/or Jackhammer that could improve your experience?
- Other than the obvious bugs noted on piazza are there any issues you encountered.

### Q2. Pipelined exploration

- Which of the design you synthesized has the best performance in terms of Gigabits hashed per second?
- Which of the design you synthesized has the best performance/energy in terms of Gigabits hashed per second per joule?
- What do you believe to be the bottlenecks for each of your design points?
- What would you recommend to alleviate these bottlenecks?

## Read me before you commit!

This section will be updated later with submission instructions

- Commiting is not enough for us to grade this lab. You will also need to push your changes to github with the following command: `git push origin master`

- 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.
- To summarize, your Git tree for lab2 should look like the following (use the Github web browser to check that everything is there):

```
/cs250-ab
  /lab2
    /src: COMMIT CHISEL CODE
    /jackhammer: original files only
    /chisel: original files only
    /build:
      /vlsi: original files only
        /generated-src: original files only
      /emulator: original files only
        /generated-src: original files only
  /writeup: COMMIT REPORT
    /results: COMMIT Jackhammer results for S={1,2,4} and fast and slow memory unit
```

## Acknowledgements

Many people have contributed to versions of this lab over the years. The lab was originally developed for CS250 VLSI Systems Design course at University of California at Berkeley by Yunsup Lee. Original contributors include: Krste Asanović, Christopher Batten, John Lazzaro, and John Wawrzynek. Versions of this lab have been used in the following courses:

- CS250 VLSI Systems Design (2009-2013) - University of California at Berkeley
- CSE291 Manycore System Design (2009) - University of California at San Diego