# Building your First Image Processing ASIC

CS250 Laboratory 2 (Version 092312)
Written by Rimas Avizienis (2012)

## Overview

The goal of this assignment is to give you some experience implementing an image processing algorithm for an ASIC using Chisel, and to teach you more about the ASIC tool flow. The image processing algorithm you will be implementing is a median filter. A median filter is a non-linear filter in which the value of each output pixel is computed by taking the median of the pixel values in a "window" surrounding the corresponding pixel in the input image.

The input to your filter module is a sequence of numbers representing the intensity of each pixel in an image, starting from the upper left corner of the image and scanning across the image row by row, from left to right and top to bottom. Since a median filter requires access to pixel data from multiple rows of the input image, your filter module will need to buffer up several rows worth of pixel data. For the first part of this assignment, you will use registers (flip-flops) to implement the storage for these "row 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 **Tuesday, September 25th at 11 AM**. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (b) Reports generated by DC Compiler, IC Compiler, and PrimeTime PX 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/report.pdf` or `writeup/report.txt`

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

### Median Filter Implementation

For this assignment, you will be implementing a median filter that operates on a a $3 \times 3$ pixel window. Figure 1 illustrates how such a filter computes the value of one output pixel by taking the median value of the nine pixels in the window. In this lab, pixel intensity values will be represented by 8 bit unsigned integers. A value of 0 encodes black and 255 encodes white. You will be working with images that are $128 \times 128$ pixels in size, so buffering two rows of pixel data will require $128 \times 2 \times 8 = 2048$ bits of storage. That's a lot of flip flops!

We have provided you with a test harness and the skeleton of a Chisel implementation of a median filter module. Your task is to fill in the Chisel code for the sub-modules in the design. The top level

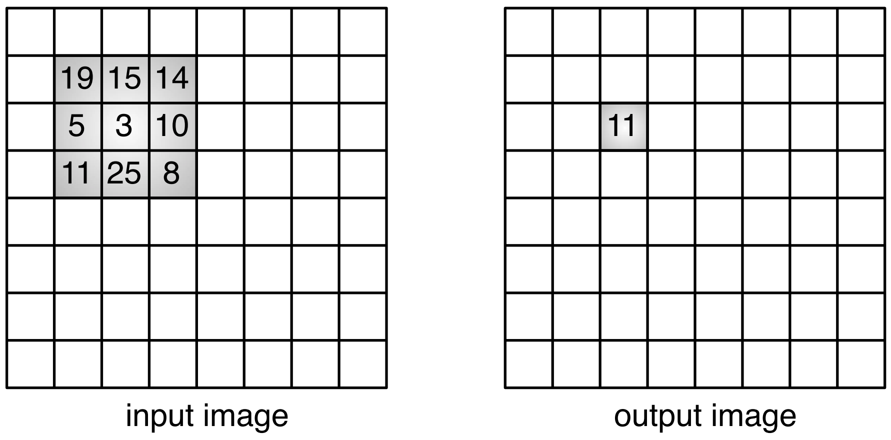

Figure 1: 3 x 3 Median Filter Example

module of the design (shown in Figure 2 ) has two inputs (not including clock and reset): a frame synchronization (`frame_sync_in`) signal and an 8-bit wide data bus (`data_in`). The `frame_sync_in` signal goes high for one cycle at the start of each frame. During that same cycle, the data for the pixel at row 0, column 0 is available on the `data_in` bus. On the next cycle, the data for row 0, column 1 will be available on the `data_in` bus, and so on. The top level module has two outputs, a `frame_sync_out` signal and a `data_out` bus. Your design should assert the `frame_sync_out` signal during the same cycle as data for the pixel at row 0, column 0 is presented on the `data_out` bus. Note that the output of the median filter is not defined for pixels around the perimeter of the image, since part of the window would extend beyond the edge of the image. For the pixels around the perimeter, your filter should pass through the original (un-filtered) input image pixel values.

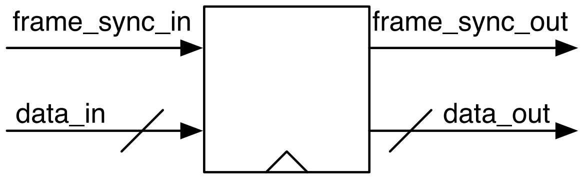

Figure 2: Top Level Filter Module

Figure 2 shows a block diagram of the internal structure of the top level filter module. The Chisel code for this module has been provided to you in the `filter.scala` file. There are three sub-blocks that you need to implement: a window buffer (in `window.scala`), a median computation block (`median.scala`), and a control module (`control.scala`). The `data_in` input bus is connected to the input of the window buffer. The window buffer has 9 outputs, which are the pixel values covered by a $3 \times 3$ window in the input image. The median computation block is purely combinational (it contains no state elements) and its output is the median value of its 9 inputs. The control module generates the `frame_sync_out` signal and a control signal for a multiplexer which selects between the output of the median computation block and the center pixel output of the window buffer. When the center pixel of the window is over one of the edge pixels in the input image, this multiplexer should pass through the unaltered input pixel data.

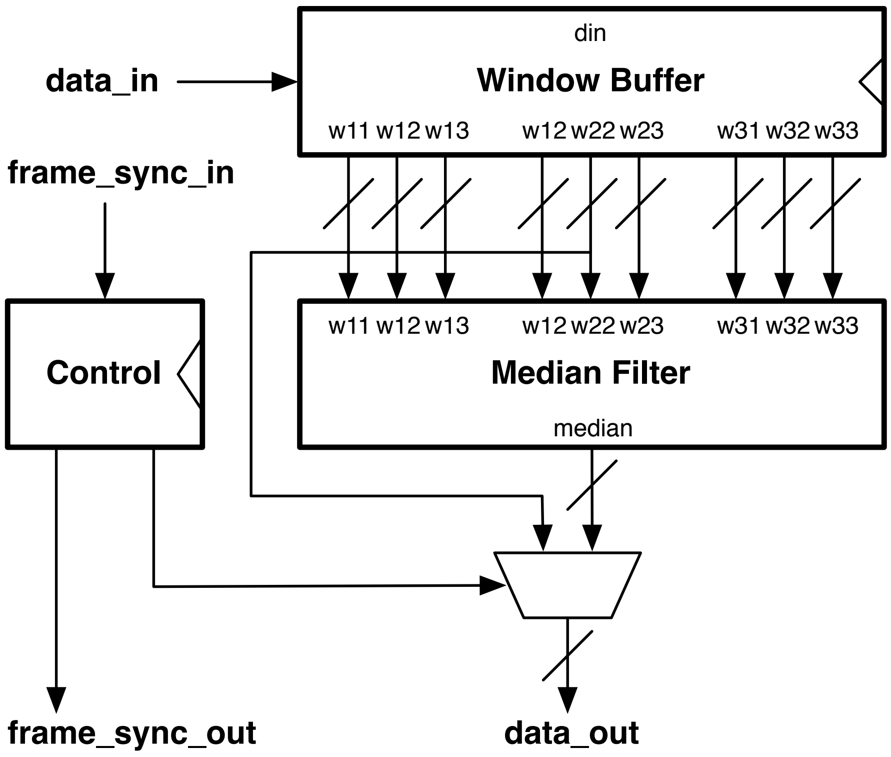

Figure 3: Top Level Block Diagram

A block diagram of the window buffer module is shown in Figure 4. It consists of 9 registers and two row buffers. The output of each register is connected to one of the module outputs. You should implement a row buffer as a separate component, and instantiate two of them in the window buffer component. A row buffer is just a FIFO queue that reads and writes a value during every cycle. It needs to be able to read and write a new value during every cycle, and its output should be a delayed version of its input. The number of cycles of delay is constant and fixed, and determined by the row width. Make the number of delay cycles and the data width parameters to your row buffer component. Use Chisel's `Mem` class to implement storage for your row buffer.

Your median computation block should use a sorting network to determine which of its nine inputs is the median. Figure 5 shows one way to build a sorting network that computes the median of nine input values by using three layers of three input sorting modules. A three element sorting module can be constructed from three two input sorting modules, and a two element sorting module can be implemented using a comparator and a multiplexer. You can implement your median computation block using a different approach, as long as it isn't doesn't use significantly more area than the one described above. You can estimate the relative size of two sorting networks by counting the number of multiplexers and comparators in each.

Your control module should keep track of the current location (x and y coordinates) of the center pixel of the window, and use this information to generate the `frame_sync_out` signal and the control signal for the output select multiplexer.

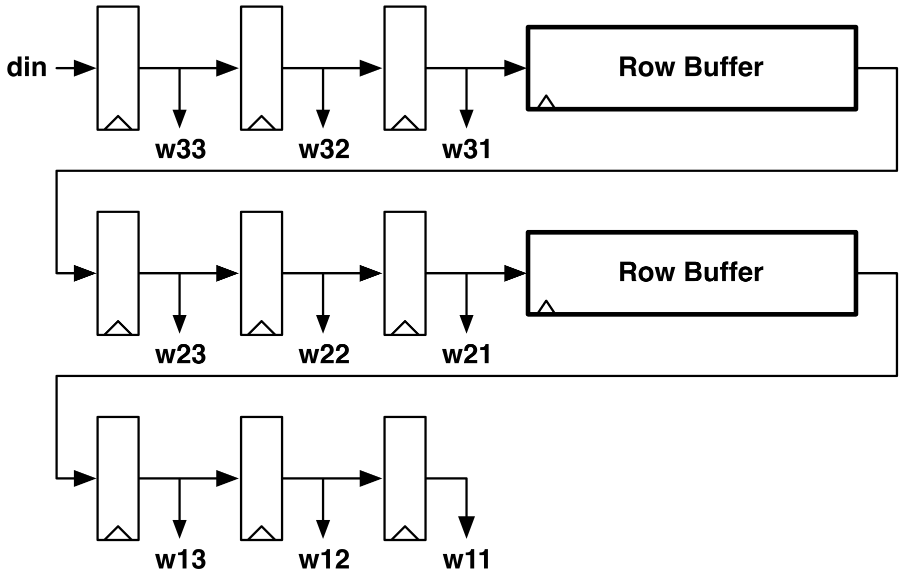

Figure 4: Window Buffer Block Diagram

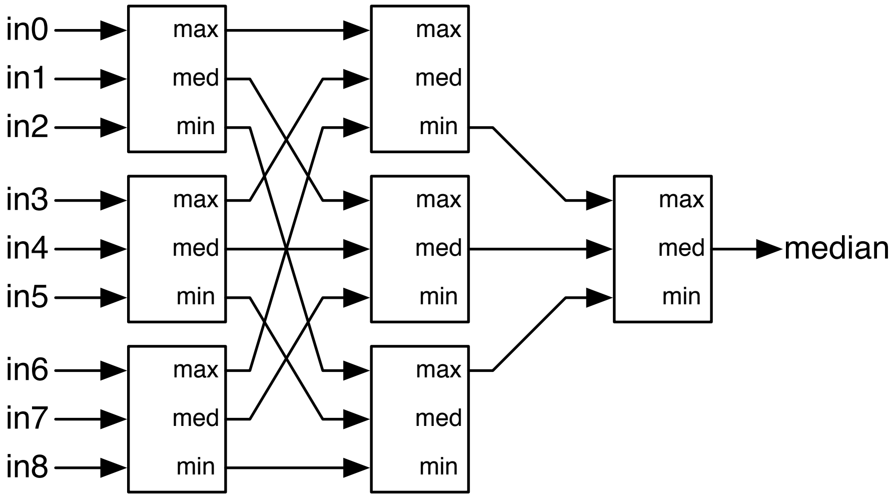

Figure 5: A nine element sorting network used to find a median value

## Getting Started

To get the files for Lab 2, go to the working directory you used for Lab 1 (or create a new one, following the instructions in the Lab 1 handout) and run `git pull template master`. That should create a `lab2` directory: this is the top-level directory for your Lab 2 design files. The skeleton Chisel code we provide will compile without errors, but will fail verification.

One caveat to be aware of: if you modify the Chisel code in such a way that one of the input or output ports of the top level module gets optimized away, the C++ emulator test harness will not compile and you'll get an error message like:

```
  error: class 'medianFilter_t' has no member named 'medianFilter__io_data_in'
```

During the compilation process, the Chisel compiler traverses the graph of nodes that constitute your design, starting at the outputs of the top level module and working its way back towards the inputs. If a node (component or top level port) is not touched during the graph traversal, it does not effect the module's outputs and hence will not appear in the C++ or Verilog produced by the compiler.

## Chisel, Mem, and ASIC SRAMs

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 and put a register at the output of the SRAM. Below is an example of how to do this for a one read, one write memory (the kind you should use in your design). You should instantiate the memory for your row buffer module in a similar fashion. For more about the `Mem` class, see Chapter 9 of the Chisel manual.

```
val ram1r1w = Mem(1024, seqRead = true) { Bits(width = 32) }
val dout = Reg() { Bits() }
when (wen) { ram1r1w(waddr) := wdata }
when (ren) { dout := ram1r1w(raddr) }
```

While you could use two separate SRAM macros to implement the storage for each row buffer, a single SRAM with twice the capacity takes up significantly less area. Assuming you implemented the window buffer following our recommendations, you will need to change it slightly. Instead of instantiating two separate row buffer modules, instantiate a single row buffer that is twice as wide (16 bits instead of 8). Use the top half of the data input and output ports for one row, and the lower half for the other. This is illustrated in Figure 6.

Note: you must follow this suggestion, as we have only given you one SRAM macro to use in this design. The macro is called `SRAM2RW128x16`, it dual-ported, 16 bits wide and 128 entries deep. Its library files and Verilog model (used for simulation) are located in the `ref/` directory. Make sure you the memory you instantiate using `Mem` has the same depth and width at the SRAM macro. Assuming your row buffer module takes the number of delay cycles as a parameter, you will need to round that value up to the nearest power of 2 (128 in this case) when instantiating the memory. You can do this rounding in Scala as follows, where n is the value you want to round up:

```
val rounded = scala.math.pow(2,log2Up(n)).toInt
```

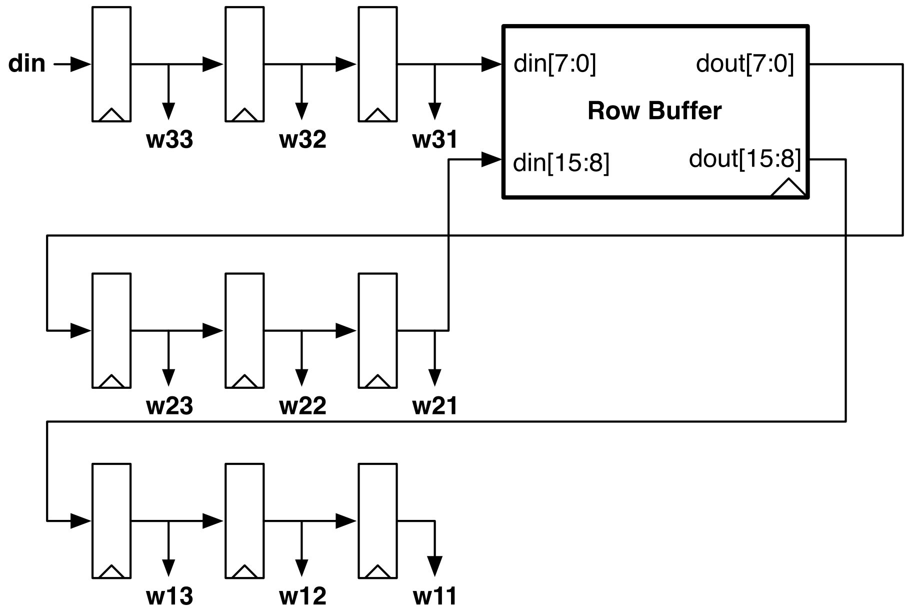

Figure 6: Window buffer implemented using a single double-width row buffer

## Build Infrastructure

The build system for Lab 2 is slightly different than the one you used for Lab 1. There is no longer a `Makefile` in the top level directory, only in the `emulator` and `vlsi/build*` directories. These `Makefiles` will invoke the Chisel compiler to regenerate the C++ or Verilog output for your design as necessary. There are three separate build directory trees in the `vlsi` directory that you will use to implement different versions of your design.

## Test Harnesses

We have provided you with Verilog and C++ test harnesses for the top level median filter module. The test harnesses generate random input images, filter them to produce the expected output images, and verify that the output of your filter module matches the expected output. We recommend that you first debug your design using the C++ emulator and test harness before simulating the generated Verilog with VCS.

The same C++ code (in the `common/` directory) is used to generate the input and expected output images for both test harnesses. The linkage between the Verilog test harness and the C++ code is done using the `DirectC` API supported by VCS. DirectC enables you to call arbitrary C code from within a Verilog simulation. For more details about `DirectC`, see Chapter 19 in the VCS User Guide. You will most likely want to use `DirectC` when building a Verilog test harness for your class project.

**C++ Emulator Test Harness**

The C++ test harness has some additional features (as compared to the version from Lab 1) that are configured through command line options. They are listed below:

```
-h <number> : specify image height (defaults to 128)
-l <number> : specify cycle limit (simulator will timeout after specified number of cycles)
-n <number> : specify number of images to simulate (defaults to 2)
-t    : enable cycle by cycle trace output (text output to console)
-v    : enable VCD (trace) file generation
-h <number> : specify image width (defaults to 128)
```

To open a VCD trace file in DVE, you first need to convert it to a VPD file. This can be done using the `vcd2vpd` command, for example:

```
vcd2vpd -q trace.vcd trace.vpd
```

Running `make run-debug` will invoke the C++ simulator with verbose (text) debugging output enabled. `make run-trace` will run the C++ simulator to produce a VCD file and then convert it to a VPD file. We have provided a TCL script for DVE to help you get started debugging. To open a trace file in DVE and run the provided script, type the following:

```
dve -full64 -vpd trace.vpd -script debug.tcl
```

That should open up a waveform window showing the inputs and outputs to your top level module, as well as the expected output and a `mismatch` signal that goes high when the simulator output doesn't match the expected output.

**Verilog Test Harness**

To simulate the Verilog version of your Chisel design, go to the `vlsi/build-rvt/vcs-sim-rtl` directory and type `make run`.

The `simv` simulator binary produced by VCS takes the following command line options:

```
+num-images=<num> : specify number of images to simulate (defaults to 2)
+max-cycles=<num> : specify number of cycles to simulate before timing out
+vcdpluson=1      : enable VPD trace file generation (off by default)
```

Once the Verilog version of your design passes functional verification, you are ready to push it through the ASIC flow. To do this, you can either manually run "make" in the directory for each step of the flow, or you can run "make" in the top level build directory to execute the steps of the flow in sequence.

**Scala Test Harness**

We have also provided a separate test harness for the median computation module to demonstrate one of Chisel's newer features: support for writing test harnesses in Scala. The test harness is included at the end of the `median.scala` file. The test vectors (inputs and expected outputs) are defined in the `median9Test` class. The code we have provided generates random numbers as inputs for the `median9` component, computes their median, and checks it against the component's output. Writing unit tests for submodules of your design in Scala can be quicker and easier than trying to do the same thing in Verilog or C++. Another benefit to writing test harnesses in Scala is that the Chisel compiler can automatically invoke the C++ compiler (or VCS) to produce a simulator binary, execute it, run the test harness and report the results.

To build and run this test harness, type `make median-test` in the `emulator` directory. To create a new Scala test harness, you need to implement a subclass of `Tester` for the component that you want to test (as in `median.scala`). Then, you need to add a few lines to `src/work.scala` and `Makefile` - use the existing lines for the `median-test` target as a template.

### ASIC Flow

Once you've simulated your design and verified that it works correctly, you're ready to push it through the ASIC flow. You will do this three times, using different tool settings each time but without modifying your Chisel code. First, implement your design using only a regular threshold voltage (RVT) standard cell library. Next, implement your design using three standard cell libraries (built with low, regular, and high VT transistors) with the tools set to optimize for leakage. Whenever possible (without violating timing contraints) the tools will use high VT cells to reduce leakage. Finally, implement your design using an SRAM macro as storage for the row buffers.

Use the `build-rvt` tree to build the first version of your design (regular VT cells only), and then use the `build-multi-vt` tree to build a version using low, regular, and high VT cells. Finally, use the `build-sram` tree to build a version that uses an SRAM macro. Most of the build scripts and Makefiles used by all three build trees are the same (symlinks that point to the same file instead of redundant copies). The differences are in the `Makefrag` file located at the top level of each build tree. The `Makefrag` file defines common parameters used by multiple steps in the flow, such as timing constraints and target libraries. Take a look at all three `Makefrag` files and make sure you understand what's going on. Running `make` in the top level of each build directory tree will push your design all the way through the flow using the settings defined in the local `Makefrag` file.

## Submission and Writeup

Along with your Chisel source code, push the reports produced by Design Compiler, IC Compiler, and PrimeTime for each of the three implementations of your median filter design to your private repository at Github. To include just the reports, add only the `reports` subdirectory of the build directory inside of dc-syn, icc-par and pt-pwr for each version of your design. Try to avoid pushing any of the other files produced by the tools, as they are rather large.

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

| Area | | RVT | Multi-VT | SRAM |
|---|---|---|---|---|
| Window Buffer | $um^2$ | | | |
| Median Sorter | $um^2$ | | | |
| Total | $um^2$ | | | |
| Cell Count | SVT | | | |
| Cell Count | RVT | | | |
| Cell Count | HVT | | | |

| Power | | RVT | Multi-VT | SRAM |
|---|---|---|---|---|
| Entire Design | | | | |
| Leakage | $uW$ | | | |
| Total | $uW$ | | | |
| Window Buffer | | | | |
| Leakage | $uW$ | | | |
| Total | $uW$ | | | |
| Median Sorter | | | | |
| 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 agree with 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 or low VT?

5. Where in your design do they appear and why?

**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 0.595 $um^2$, what is the efficiency (area used for bitcells over total area) of this SRAM macro?

4. Would you expect this SRAM to use 6T or 8T bitcells, and why?

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

**Questions: Power and Performance**

1. Assuming a clock frequency of 250 MHz, calculate the amount of energy (in Joules) used to compute a single output pixel and an entire image for all three versions of your design.

2. How would pipelining your median sorting module change the timing (critical path) and area of your design? How can pipelining improve performance, and how does it effect latency and throughput? What effect would you expect pipelining to have on energy efficiency?