Design Space Exploration: Implementing a Convolution Filter

CS250 Laboratory 3 (Version 101012)
Written by Rimas Avizienis (2012)

Overview

This goal of this assignment is to give you some experience doing a design space exploration, similar to what you'll need to do for your class project. First, you'll get some more experience coding in Chisel, this time with an emphasis on creating a parameterized design. Parameterizing your design will make it easy to automate the process of design space exploration through scripting. You will write a script which will vary the parameter values, invoke the Chisel compiler to generate a Verilog implementation of each design point, and push the resulting implementations through the toolflow to obtain area, power and performance numbers. In this assignment (as in your class projects), you will have a fixed performance requirement, and your design space exploration will investigate tradeoffs between area and energy efficiency. You will pipeline your design to see what effect that has on energy efficiency and area, and you will also scale down VDD to see if replicating functional units and running them at lower frequency makes sense for your design.

For this assignment, you will implementing a convolution filter module. Like the median filter you designed for Lab 2, a convolution filter is a "windowing" filter that uses multiple input pixel values to compute an output pixel value. Unlike the median filter, the convolution filter requires performing arithmetic (multiplications and additions) on its operands. In addition to a window of input pixel values, a convolution filter also has a filter "kernel" input. The filter kernel is the same size as the input window, and each element is a scale factor for the corresponding input pixel. The output of the convolution filter is thus the sum of the scaled values of its input pixels. The filter kernel coefficients are often fractional values, which we will encode using fixed-point number representations. Typically, programmers use floating point arithmetic in such situations, but using fixed point can lead to more optimal (smaller, more energy efficient) hardware implementations. Your filter module will take several parameters including the widths of the pixel and coefficient values and the number of pipeline stages.

Deliverables

This lab is due **Wednesday, October 17th at 11:59 PM**. 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, and PrimeTime PX checked into your git repo for each design point you push through the toolflow.
- (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.

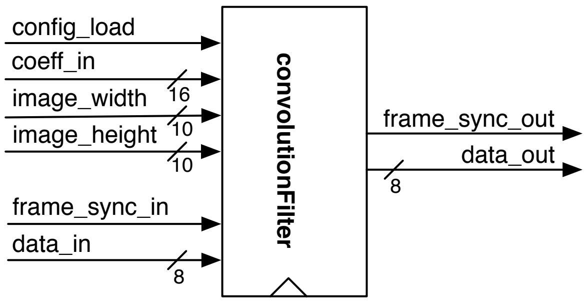

Figure 1: Top Level Module IO Ports

Design Specification

Your assignment is to implement a 5 × 5 convolution filter module that can process images that range in size between 8 × 8 and 1024 × 1024 pixels. Pixel values are represented as 8 bit unsigned numbers, and filter coefficients are represented as signed (two's complement) 16 bit fixed point numbers, with 12 bits to the right of the binary point. The input and output ports of the top level module are shown in Figure 1.

The `config_load`, `coeff_in`, `image_width` and `image_height` inputs are used to configure the filter module. One coefficient is loaded from the `coeff_in` bus during each cycle that `config_load` is asserted, so loading an entire filter kernel takes 25 cycles. The `image_width` and `image_height` inputs specify the dimensions of the input image, and their values should also be loaded into registers when `config_load` is asserted. The configuration parameters can't change mid-image, so the `config_load` signal can only legally be asserted while the filter is idle. The timing of the `frame_sync` and `data` input and output signals is the same as in Lab 2: the `frame_sync` signal goes high for one cycle, at the same time as the first pixel of an image is on the data bus.

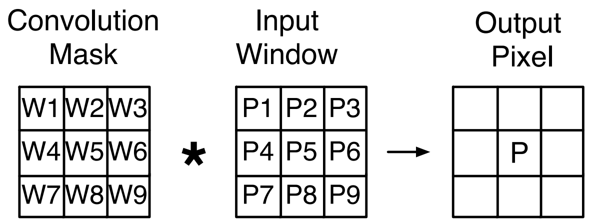

Figure 2: 3 x 3 Convolution Filter Example

The basic structure of the convolution filter module is shown in Figure 3. It consists of a window buffer, a convolution module, control unit, output select multiplexer, and registers for the outputs. The filter coefficients are stored in a bank of registers inside the control module.

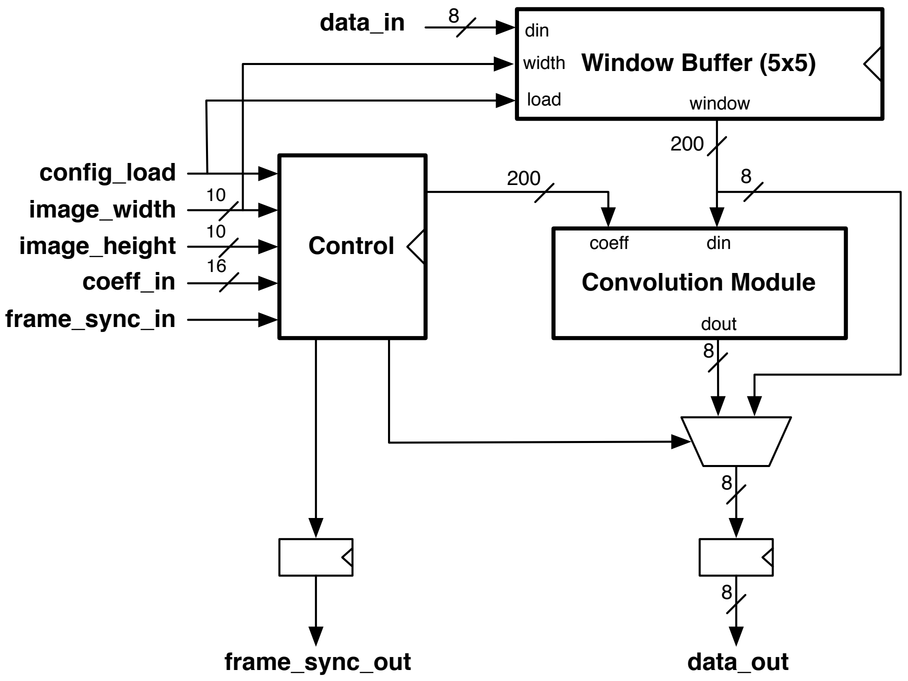

Figure 3: Convolution Filter Module Block Diagram

Convolution Filter Implementation

Figure 2 illustrates the computation performed by a 3 × 3 convolution filter, where pixels are represented as 8 bit unsigned integers. This convolution is described by the following equation:

$$P = \sum_{i=1}^{9} W_i * P_i \tag{1}$$

Each of the 9 input pixels is multiplied by a filter coefficient, and the results are summed to produce the filter's output. The output must fall inside the range of legal pixel values (0-255), so if the result of the summation is negative or greater than 255, the filter output should be set to 0 or 255, respectively. Pixels around the perimeter of the input image (where the filter window would extend beyond the border of the image) should pass through the filter unaltered. The convolution module you will implement in Chisel takes 4 parameters. They are listed below, along with their default values:

- `dataWidth`: The bit width of the pixel inputs (8)
- `coeffWidth`: The total bit width of the coefficient inputs (16)
- `coeffFract`: The bit width of the fractional part of the coefficient inputs (12)
- `pipeStages`: The number of pipeline stages (1)

The first step in any design space exploration is to choose a baseline or reference implementation. For this assignment, you will use a single cycle (purely combinational) implementation of the convolution function as your baseline. Figure **??** shows one way to implement a single-cycle 3×3 convolution operator, using nine multipliers and 16 adders arranged in a tree structure. The output of each multiplier is a 24 bit fixed-point value. Note that the output values computed by each level of the adder tree must be one bit wider than the inputs to avoid the possibility of overflow. Your convolution module will need to maintain full precision throughout the computation until the final step. At that point, you should round the output by adding 0.5 to the final result and truncating it to 8 bits.

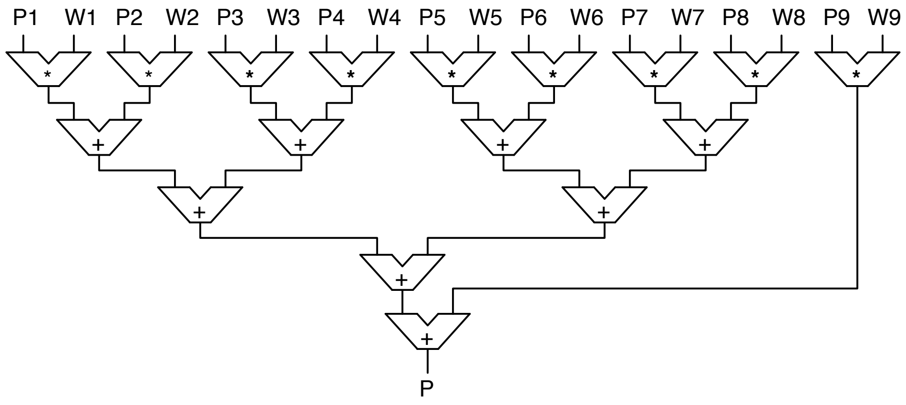

Figure 4: A single cycle implementation of a 3 x 3 convolution operator

This is just one (not necessarily optimal) way to implement this computation. Efficiently mapping a datapath like this one to a collection of gates is a challenging optimization problem in its own right: there are many ways to implement adders and multipliers, and the best choice for a particular situation is generally not obvious. Fortunately, this is the kind of thing that Design Compiler is really good at. At the level of RTL coding, expressing the desired computation at the highest level of abstraction possible (i.e. only using the $*$ and $+$ operators) gives the synthesis tool the freedom to consider the widest range of possible implementations. For more information about how Design Compiler synthesizes datapaths, see the `Coding Guidelines for Datapath Synthesis` document on the course handouts page.

NOTE: We recommend that you only use the UFix type in your convolution Chisel code even though you will be dealing with signed values (where you would ordinarily use Fix). At the moment, the Chisel generated Verilog doesn't correctly use the `signed` qualifier everywhere, causing DC to issue warnings that it may not be able to fully optimize the datapath. DC now generates a report about datapath extraction in the file `current-dc/reports/convolutionFilter.datapath.rpt`: you should read it and check for warnings. We also recommend that all signals that hold intermediate values in your design be explicitly set to the bit width you want for the final result. DC will optimize away any unnecessary extra bits.

After you have built your baseline design, the next step in your design space exploration will be to apply pipelining to your convolution module. Pipelining a functional unit requires making two decisions: how many pipeline stages to use, and where to place the pipeline registers. Figure 2 shows one way to pipeline the datapath from Figure **??**. The single cycle design has been partitioned into four pipeline stages by inserting three layers of pipeline registers. This placement of pipeline registers is probably not optimal, as the amount of logic (a multiplier, one or two adders) in each

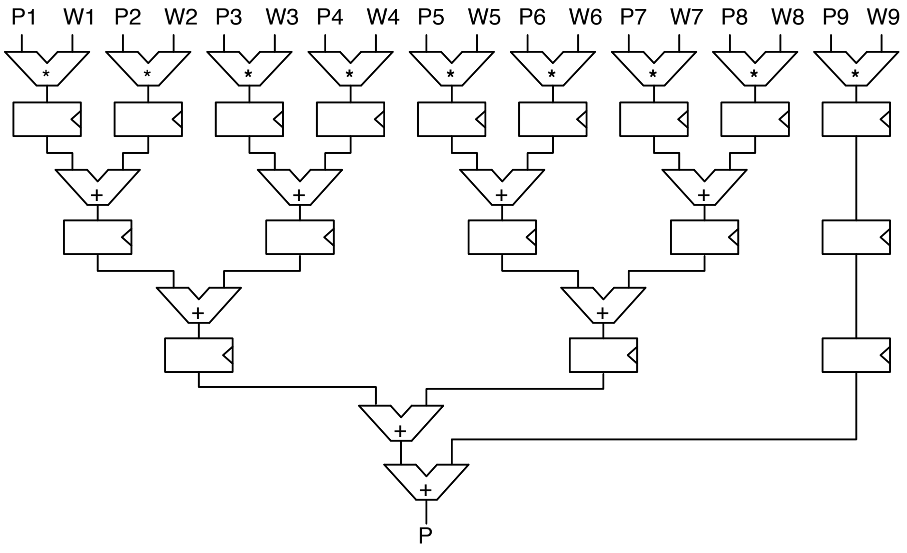

Figure 5: A pipelined version of a 3 x 3 convolution operator

stage is unequal. The longest path (in terms of delay) through any stage limits the performance of the entire pipeline, so balancing the delay across stages is important. Manually finding a good placement for pipeline registers can be challenging, however synthesis tools can aid in this process by `retiming` the pipeline registers. In register retiming, the locations of the flip-flops in a sequential design are automatically adjusted to equalize as nearly as possible the delays through each stage. A simple example of retiming is shown in Figure 6. For more information about how retiming is implemented in Design Compiler, see the `Design Compiler Register Retiming Reference Manual`.

For this lab, the Chisel code we provide includes a wrapper for your convolution module which handles the pipelining for you. It instantiates the number of pipeline registers specified by the `pipeStages` parameter at the output of your convolution module. Retiming the module in Design Compiler is a two step process: first the entire design is synthesized with a loosened timing constraint set on the convolution module. At this point, all the pipeline registers are bunched up at the end of the datapath. Next, the timing constraint is set to the target clock frequency and the module is retimed, distributing the pipeline registers through the datapath. The sequence of commands that implements this process is in the `dc-syn/dc_scripts/dc.tcl` file if you're curious. The DC log file (in `current-dc/log/dc.log`) will include lots of information about the retiming process. After synthesis, you can see where the pipeline registers have been placed in the pipeline by opening the synthesized design in Design Vision. To do this, type the following command in your `current-dc` directory:

```
design_vision-xg -64bit -f dc_setup.tcl -x "read_ddc results/convolutionFilter.mapped.ddc"
```

Select the convolver module from the Logical Hierarchy window, right click on it and select "Schematic View." To highlight the pipeline registers, click the "Select" pulldown menu and choose "By Name." Under "Match", make sure that "Full Hierarchical Names" is selected. In the "Search for:" field, enter `convolver/*reg*`, then click "Search" and "Select all." The registers should now be highlighted in the schematic view.

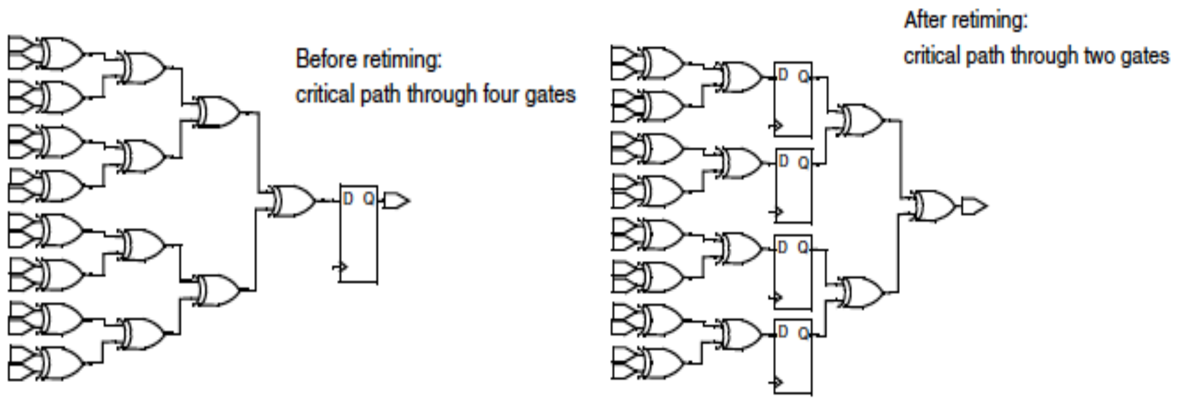
Before retiming:
critical path through four gates
After retiming:
critical path through two gates
D Q

Figure 6: An example of register retiming

Assignment Details

The files for Lab 3 are in the `lab3` directory of the lab-templates repository on github. We have provided you with incomplete Chisel code for the convolution filter module, along with C++ and Verilog test harnesses. The test harnesses behave the same way as those from Lab 2. We have provided an SRAM macro (`SRAM1R1W1024x32`: 1024 entries, 32 bits wide, one read + one write port) for you to use in your window buffer module. To make sure it gets used in the ASIC flow, follow the guidelines from Lab 2 about using `Mem` to infer SRAMs.

Chisel's `Vec` class

Chisel's `Vec` class creates an indexable vector of elements. You will see that `Vec` is used in the definitions of some of the IO interfaces in the skeleton design. For example, from `convolution.scala`:

```
val io = new Bundle {
  val din   = Vec(windowSize) { UFix(dir = INPUT, width = dataWidth) }
  val coeff = Vec(windowSize) { UFix(dir = INPUT, width = coeffWidth) }
  val dout = UFix(OUTPUT, dataWidth)
}
```

As you can see, `Vec` makes it easy to parameterize the number of input or output ports on a module. `Vec` can also be used to create arrays of wires or registers. Here is an example of how to create an array of wires:

```
val data = Vec(elements) { Fix(width = w) }
```

An example demonstrating how to use `Vec` and a `for` loop to create a shift register with a parameterized width (w) and depth (d) is shown below:

```
val shift_regs = Vec(d) { Reg() { Bits(width = w) } }
shift_regs(0) := io.din
for (i <- 1 until d) {
    shift_regs(i) := shift_regs(i-1)
}
io.dout := shift_regs(d-1)
```

You can also use `Vec` to create a two dimensional array of registers, for example:

```
val din_regs = Vec(x) { Vec(y) { Reg() { UFix(width = w) } } }
```

Build Infrastructure

The build infrastructure for Lab 3 is very similar to Lab 2. To partially automate the design space exploration process, the Makefiles (and Makefrag) have been modified so that design parameters can be specified as arguments to `make` and are propagated as necessary to both the Chisel compiler and the ASIC tools. There are 4 variables you can set from the command line:

- `PIPE_STAGES`: The number of pipeline stages in the design. This value is used as the `pipeStages` parameter to the top level Chisel module, and is also used by Design Compiler during the register retiming process. If not specified, it defaults to 1.
- `CLOCK_PERIOD`: The target clock period for the design. This value is passed to Design Compiler. If not specified, it defaults to 2 ns (500 MHz).
- `VOLTAGE`: The value of VDD used for the design. The three valid settings are: `STANDARD, MIDDLE, LOW`, corresponding to operating voltages of 1.05, 0.95 and 0.85 volts. DC uses this setting to choose which standard cells libraries to use. If not specified, it defaults to `STANDARD`.
- `PIPE_OVERHEAD`: The delay associated with the insertion of a pipeline register, which is determined by the setup time and clock to Q delay of the flip-flops used. It is used by DC during the retiming process. If not specified, it defaults to a value that depends on VDD (between 0.1-0.2 ns).

Here is an example of how you might invoke `make` from the `vlsi/build/dc-syn` directory to synthesize one design point:

```
make CLOCK_PERIOD=1.0 PIPE_STAGES=2 VOLTAGE=STANDARD
```

The Makefiles that build a C++ or VCS RTL simulation of your Chisel design only use the `PIPE_STAGES` variable. If you are performing post place-and-route simulation (in `vcs-sim-gl-par`) of a design that has a clock period other than 2 ns, you need to specify the `CLOCK_PERIOD` variable to `make` so that the simulation uses the proper clock frequency. Note that the Verilog file produced by Chisel (in `vlsi/generated-src`) is renamed to indicate the value of `pipeStages` used when building the design.

Design Space Exploration

Once you have verified the functional correctness of your Chisel design (both for the unpipelined and pipelined cases) through simulation, you are ready to push your design through the ASIC flow. Your initial performance target is 1000 MHz. Synthesize your design four times with 1–4 pipeline stages. Automate this process by writing a shell script that invokes `make` with different command line arguments. Write a python script to gather power, area and timing estimates from the reports generated by Design Compiler. Based on the post-synthesis results, pick the best design point (in terms of energy efficiency), and push it all the way through the flow.

Now, lets assume that what you actually want to do is filter four image streams running at 250 MPixels/second each. Your objective is to determine whether you could save energy by replicating your filter module and scaling down its operating voltage and frequency, and what the area cost would be. Synthesize your design at operating frequencies of 500 MHz and 250 MHz with 1–4 pipeline stages and middle and low settings for VDD. Based on the synthesis results, pick the most energy efficient design point for each target frequency and push it all the way through the toolflow to layout.

Writeup

Your writeup should include two tables: one with all your post-synthesis results and another with your post place-and-route results. Each table should include timing (critical pack slack - i.e. whether or not the design met timing), area and power numbers, as well as a breakdown of the types of cells (HVT, RVT, LVT) used. For the area and power numbers, report the total as well as the amount used by the window buffer and convolution modules. For your post-place and route results, include power estimates from PrimeTime and from IC Compiler.

Based on this data, write up a summary of your design space exploration and make sure to address the following questions:

Part 1

Compare the energy efficiency of the pipelined and un-pipelined versions of your design running at 1000 MHz.

1. Does the non-pipelined version of the design meet timing? Taking into account the non-pipelined version's clock period, what is the relative energy efficiency of the pipelined and non-pipelined versions of the filter module? Why do you think this is the case?

2. How many pipeline stages were used to obtain the most energy efficient implementation? Why would continuing to increase the number of pipeline stages beyond this point not yield further improvements in energy efficiency?

3. How do the individual components of power (switching, internal, leakage) differ between the versions?

4. How does pipelining effect the area of the resulting implementations? Does this agree with your expectations? If not, speculate as to what factors might account for the discrepancy.

5. Describe how the post-synthesis and post place-and-route area/power/timing numbers differ. To what do you attribute these variations?

Part 2

Compare the energy efficiency of the convolution module running at 250, 500 and 1000 MHz (only consider the most energy efficient design point for each, and use post place-and-route numbers).

1. How many pipeline stages were used and what was the operating voltage for each?

2. Disregarding the other components of the design (window buffer and control logic), compare the total area and energy efficiency of using 1 x 1000 MHz, 2 x 500 MHz, and 4 x 250 MHz filter modules in a module designed to process four 250 MHz image streams in parallel. Create a graph of this data with area on the X axis and energy efficiency on the Y axis. Which implementation would you pick if area were not a factor in your decision?