Combinational Design Project – Pooling filter

Due: 11:59 PM PST, February 10

M16 – Winter 2021

The goal of this project is design a digital circuit that implements a common type of image filter known as a pooling filter (also known as a pooling layer). This type of filter is employed to reduce the size of an input image while keeping relevant features of the input and discarding irrelevant information, such as noise. Pooling layers are an essential component of convolutional neural networks as they allow models to learn relations between distant regions of the input image, which is necessary for most modern computer vision applications.

Pooling is performed by sliding a rectangular window across the input image and applying a reduction function for each position of the window, each time producing a single output pixel. For a 2×2 window and reduction function f, the pixels of the output image O are thus generated from an input image I as given by

$$O_{i,j} = f\left(I_{2i,2j}, I_{2i+1,2j}, I_{2i,2j+1}, I_{2i+1,2j+1}\right). \tag{1}$$

A common reduction function is the maximum function:

$$f_1(x_1, x_2, x_3, x_4) = \max\{x_1, x_2, x_3, x_4\}. \tag{2}$$

Because of its nature, it is effective at reducing the size of the manipulated data while keeping all useful features (i.e. peaks in the image). Another common pooling function is the mean function:

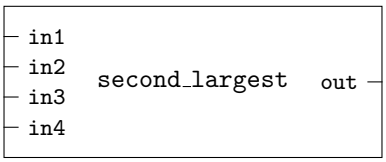
$$f_2(x_1, x_2, x_3, x_4) = \frac{x_1 + x_2 + x_3 + x_4}{4}. \tag{3}$$

When applied to an image, the mean pooling function has a smoothing effect.

In this project, you are asked to design a circuit that can compute the following pooling function:

$$f_3(x_1, x_2, x_3, x_4) = \text{second largest element of } \{x_1, x_2, x_3, x_4\}. \tag{4}$$

As it has four inputs and one output, your circuit will have the following interface:



This project assumes that the manipulated images have a *color depth* of 4 bits. As a consequence, the signals `in1`, `in2`, `in3`, `in4`, and `out` are all 4-bit unsigned integers, represented in Verilog using 4-bit buses. The circuit is combinational and therefore should not contain any memory elements (i.e. *latches* and *flip-flops*). You are expected to synthesize the `second_largest` module and check the

reported synthesis statistics to ensure that no memory element has been inferred.

To help you understand the behavior of the `second_largest` module, the following table presents examples of input combinations and the corresponding expected outputs.

in1	in2	in3	in4	out
0000	0001	0010	0011	0010
0001	0010	0001	0011	0010
0001	0001	0001	0011	0001
0001	0001	0001	0001	0001
1000	0100	0010	0001	0100
0111	1010	1011	0000	1010

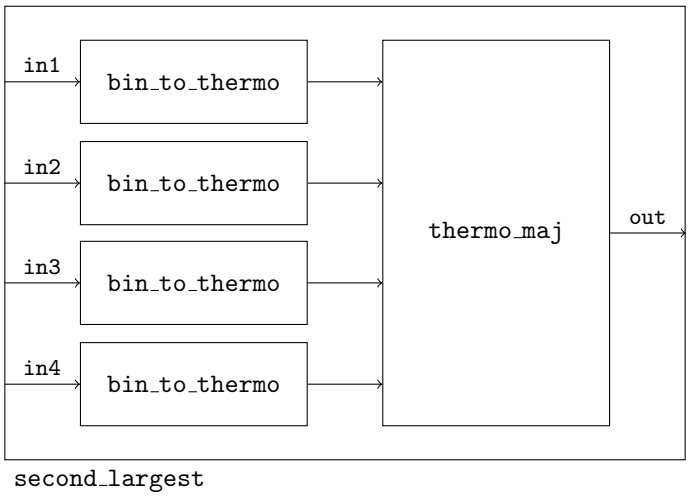
Implementation overview

To serve as examples of Verilog code, parts of the circuit have already been implemented. Your objective is to complete the implementation of the remaining submodules.

To find the second largest input, the strategy employed by the given incomplete circuit is to first convert the input binary values to *thermometer code*. The thermometer encoding of a 4-bit unsigned integer φ is the 15-bit sequence $a_{14}a_{13}\ldots a_1a_0$ defined as

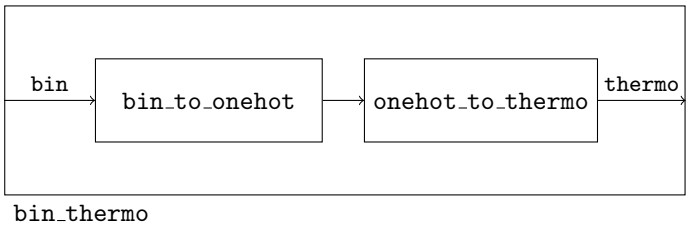
$$a_i = \begin{cases} 1 & \text{if } \varphi > i \\ 0 & \text{if } \varphi \leqslant i \end{cases} \tag{5}$$

After this conversion, the second largest element can be easily computed using majority gates. The `second_largest` module is thus implemented as shown here:

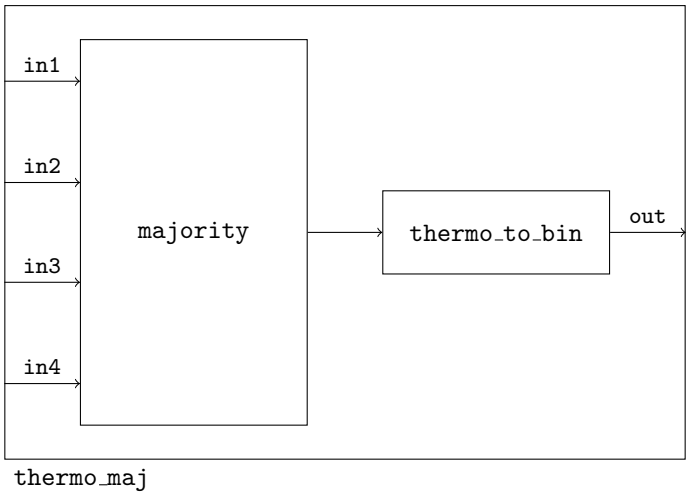


In the above diagram, `bin_to_termo` is a module that is itself divided into two submodules. As visible in the next diagram, conversion from binary to thermometer coding is performed by first converting the input to one-hot encoding. The one-hot encoding of a 4-bit unsigned integer φ is a 16-bit value $b_{15}b_{14}\ldots b_1b_0$ defined as

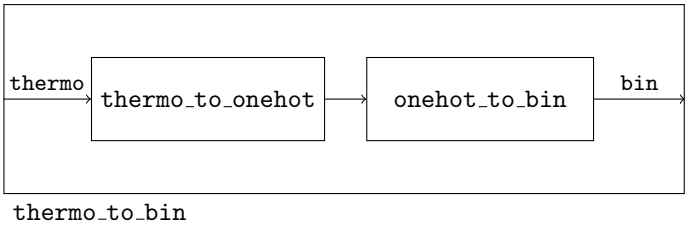
$$b_i = \begin{cases} 1 & \text{if } \varphi = i \\ 0 & \text{if } \varphi \neq i \end{cases} \tag{6}$$



The `thermo_maj` module, which effectively finds the second largest element, is divided into two submodules:



The `majority` module outputs the thermometer encoding of the second largest element, and the `thermo_to_bin` converts the output to a 4-bit integer. To find the second largest element, the `majority` module simply applies a 4-input majority gate to each of the 15 bit positions. This majority gate must output 1 whenever two or more of its inputs are 1. Just like `bin_to_thermo`, the `thermo_to_bin` module works by first converting its input to one-hot encoding, as shown in the following diagram.



List of modules

- `second_largest` (provided): top-level module, outputs the second largest of its four 4-bit inputs.
- `bin_to_thermo` (provided): converts its 4-bit integer input to a 15-bit thermometer code.
- `bin_to_onehot` (provided): converts its 4-bit integer input to a 16-bit one-hot encoded value.
- `onehot_to_thermo` (provided): converts its 16-bit one-hot encoded input to a 15-bit thermometer code.

- `thermo_maj` (provided): accepts as input four 15-bit thermometer codes and outputs the 4-bit binary value of the second largest.
- `majority` **(to do)**: accepts as input four 15-bit thermometer codes and outputs the second largest one.
- `thermo_to_bin` **(to do)**: converts its 15-bit thermometer input to the corresponding 4-bit output integer.
- `thermo_to_onehot` **(to do)**: converts its 15-bit thermometer input to the corresponding 16-bit one-hot value.
- `onehot_to_bin` **(to do)**: converts its 16-bit one-hot input to the corresponding 4-bit integer.

Testbenches

A testbench (`tb_pooling`) is provided to test the `second_largest` module. This testbench reads a 512×512 input image from a file, applies the implemented pooling function at each window position for each color channel, then writes the resulting 256×256 output image to a new file. This testbench uses a nonstandard raw image format for the input and output images. A properly encoded picture of UCLA's most recognizable building, Royce Hall, is provided for running your tests. To visualize this image and the filtered output image, an online converter is available at `https://ldelhez.github.io/image-converter`. Because of the nature of the applied pooling function, the extreme pixel values of the input image tend to be discarded by the filter. As a result, the output image should be overall less noisy than the original image.

In addition to the provided `tb_pooling` testbench, we require you to write your own testbench for the `thermo_maj` submodule. You are free to implement this testbench in whatever way you see fit. However, your testbench must at least satisfy the following requirements:

1. It must test at least 6 different combinations of inputs.
2. It must test some *edge cases* (e.g. multiple equal inputs, expected output is `0000` or `1111`).
3. It must display an error message if the output of the circuit is not correct.

Project deliverables

At the beginning of Week 5, by February 2, we require you to fill out the mid-project report available at `https://forms.gle/YfV1eRzMWymc8Nrf7`.

Please submit the following files on Gradescope by the deadline on February 10:

- `majority.v`: Verilog implementation of the `majority` module.
- `thermo_to_bin.v`: Verilog implementation of the `thermo_to_bin` module.
- `thermo_to_onehot.v`: Verilog implementation of the `thermo_to_onehot` module.
- `onehot_to_bin.v`: Verilog implementation of the `onehot_to_bin` module.
- `tb_thermo_maj.v`: Verilog testbench that checks the correctness of the `thermo_maj` module.