Istanbul Technical University

Digital System Design Applications

Experiment VIII
Image Processing System

In this experiment, students will implement an image processing system. A zero padded input image will be loaded up to a RAM. Using a given kernel, 2-D convolution will be applied on the input image. Output image will be recorded to a text file, then will be examined in MATLAB.

Objectives

- Creating an image processing system from scratch.
- Testing the created system by using a sample grayscale image.
- Creating testbenches to generate output text files.
- Verifying the design by examining the output image.

Requirements

Students are expected to know;

- The general concept of 2-D convolution,
- How to read state diagrams and express them in Verilog.
- How to set and instantiate a Xilinx Block RAM IP.
- How to record circuit outputs to a text file using a testbench.

Experiment Report Checklist

1. Codes for newly created modules (do not include the modules written in previous experiments).
2. Clear explanations for designing the processes of submodules.
3. Show post-sytnhesis FPGA resource utilization of the full design (number of LUTs, flip flops and I/O).
4. Determine the maximum clock frequency for the full system.
5. Testbench code, if edited or re-written.
6. MATLAB screenshots of input and output images.

- **Projects and reports are to be done INDIVIDUALLY. High amounts of points will be deducted from similar works.**
- **Reports must be written in a proper manner. Divide your text to sections and subsections if needed, label your figures and connect your sections with proper explanations of your works. Reports filled with imprecisely placed tables and figures, with no verbal explanations in workflow, will not fare well.**

Implementation of an Image Processing System

The image processing system with the top level schematic given at **Figure 1** will be realized in the scope of this experiment. Each submodule will be designed separately and connected together using the **Block Design**.

Convolution Unit

1. Add the previously designed **Multiply and Accumulate (MAC)**, **Behavioral Multiplier (MULTB)** and **Behavioral Adder** modules to the project.

2. Create a new module called **MAC_Normalize**. This module should have 20-bit input **data**, and 8-bit output **result**. **MAC_Normalize** unit will normalize the output of the MAC unit. 20 bits output data of the MAC unit will be mapped to the 8 bits data since the range of the grayscale images is **[0,255]**. Map the values exceeding this range to boundary values 0 or 255. For example: -100 to 0, 300 to 255. Obtain this behavior using an always block.

3. Create an another module called **CONV**. This module should include a **MAC** instance and a **MAC_Normalize** instance connected one after the other. This module should have 1-bit inputs **clk**, **reset**; 24-bit inputs **data**, **weight** and 8-bit output **result**.

4. Create a new module called **CONV128**. This module will have 128 parallel **CONV** units that perform multiply and accumulation operation of an entire row of the 130x130 image in one clock cycle. This module should have 1-bit inputs **clk**, **reset**; 1040-bit (130*8) input **data**, 24-bit input **weight** and 1024-bit (128*8) output **result**. Use a **generate block** to obtain 128 parallel CONV units.

5. 1040-bit (130×8) input **data** should be provided by Block RAM1 and 24-bit input **weight** should be provided by Input Controller. 1024-bit (128*8) output **result** will be the concatenation of the results of the 128 CONV units.

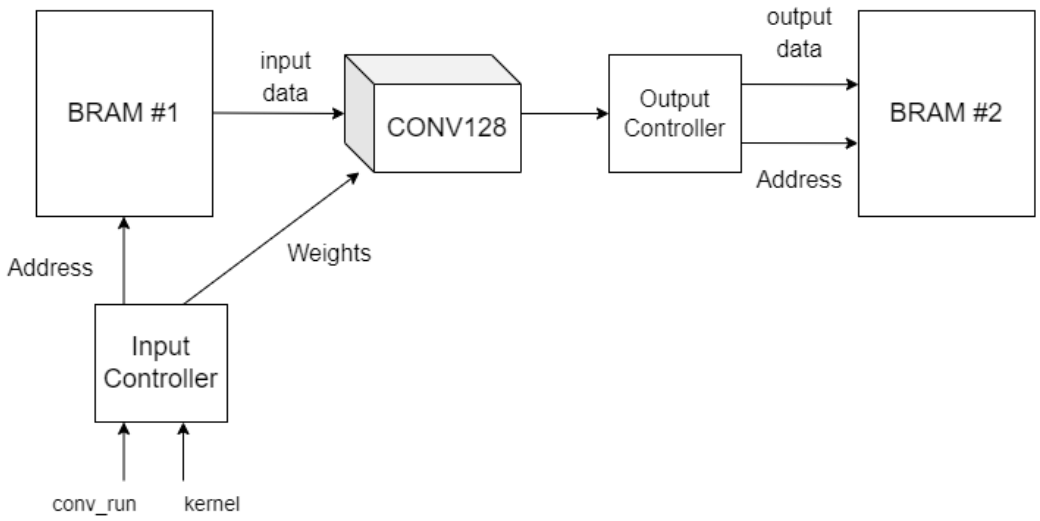

Figure 1: Top level schematic of the image processing system.

Block RAM #1

1. **Block RAM1** will contain the **input image**, which is given as a .coe file (see Ninova Course Files, under /Experiment_Files/Experiment_8/image.coe).

2. Create a block BRAM IP using the settings given at **Figure 2**. Leave unspecified settings as default.

3. Load up the **"image.coe"** as your BRAM's initialization file. This file has 1040-bit values in 130 addresses. Each row includes 130 columns and each element is 8-bit pixels (130×8). Considering that there are 130 different rows (addresses), image size is 130×130 bytes.

4. The original image size is actually 128×128. It's edges were padded with zeros to obtain the output as the same size with the original input. Thus the new input image size is 130×130 and output will be 128×128.

5. Block RAM1 will only be used for reading data.

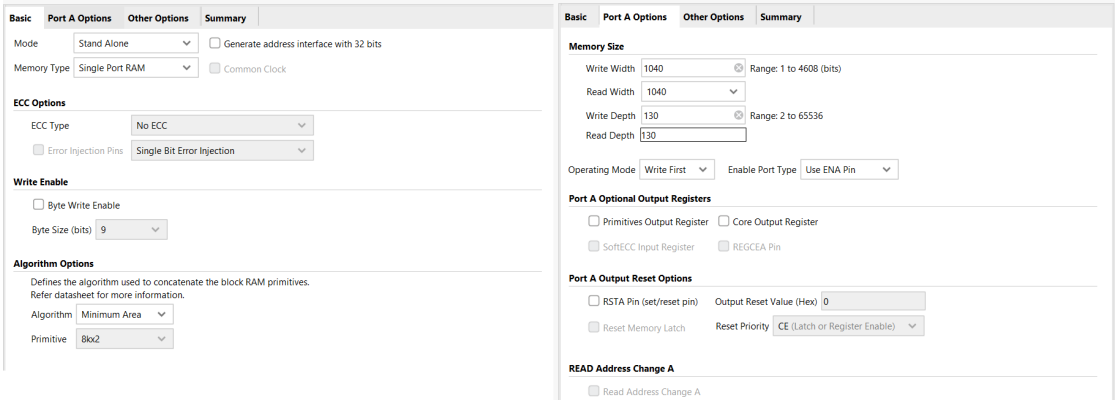

Figure 2: Settings for Block RAM IP.

Input Controller

1. Create a new Verilog module named **control_input**. This module should have 1-bit input **clk**, 1-bit input **reset**, 1-bit input **conv_run**, 72-bit input **kernel**; 1-bit output **enable_ram**, 1-bit output, 8-bit output **address_ram** and 24-bit output **weight** .

2. This module should provide the **ram addresses** for Block RAM1 and **input weights** for CONV128 block. When **conv_run** signal is set, controller starts the convolution operation and give the appropriate addresses and weights. Define a 2-bit register **counter** to count the number of accumulations of the parallel MAC units. For each count, Block RAM1 address should be incremented by 1 and 24-bit weights should be shifted. Parallel convolution operation can be shown in **Figure 3**.

3. Input **kernel** will be 3×3 **Laplacian Filter**.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

4. If **reset** signal is 1, **ram_address** should be 0x00, **enable_ram** should be 0 and also **CONV** units should be reset.

5. When the **conv_run** signal is high, convolution operation should start or continue. If it is low convolution operation should be paused.

6. After 3 clock cycle, convolution of the one row is completed, CONV units are reset and the next row of the input image is loaded to CONV units.

7. Coding can be done in any desired style.

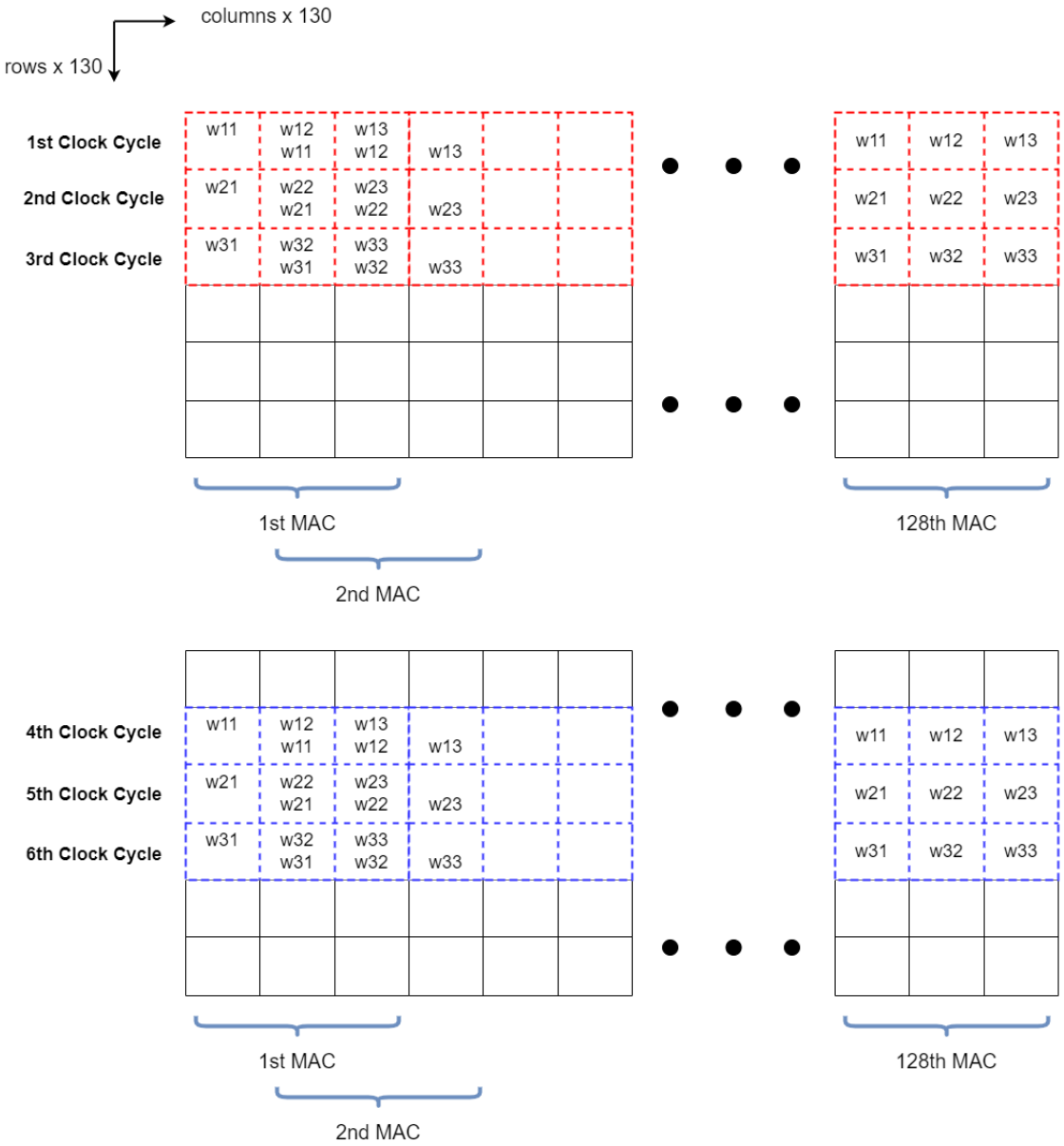

Figure 3: Parallel Convolution

Block RAM #2 and Output Controller

1. **Block RAM2** will contain the **output image**.
2. Create a block BRAM IP using the same settings except the Memory size given at **Figure 2**. The Memory size will be 1024×128 since the output image size will be 128×8 (1024) bits and the number of rows will be 128.
3. Create another module named **output_control**. This module should have 1-bit input **clk**, 1-bit input **reset**, 1024-bit input **data**, 1-bit output **conv_done**, 7-bit output **ram_address** and 1024-bit output **data_out**.
4. This module write the each row of the output image after 3 cycles is completed. Define a 2-bit register **counter** to count the number of cycles.
5. When the 130th row is calculated, **conv_done** signal should set to 1.

Forming the Top Module

- When all submodule designs are completed, the system must be gathered up in a top level module as seen at Figure 1. Create a new Verilog source file named "TOP.v" and instantiate all the blocks seen at Figure 1.
- Make all module connections using wires.
- Figure 4 shows the application that is going to be performed on this top module, and it specifies the kernel for convolution to be used.

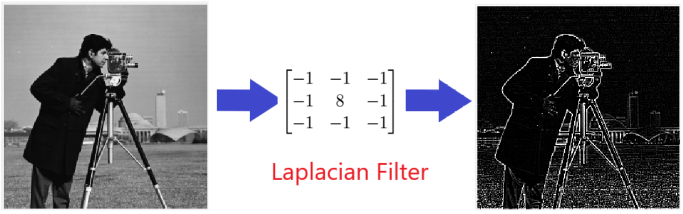

Figure 4: Edge Detection using Laplacian Filter

Testing the Image Processing System

- Write a new testbench that reads the Block RAM2 and generate a text file containing pixel values of your output image.
- Download the sample MATLAB script, "test_image.m" for viewing your output text file as an image by using Matlab.
- View the input and output images by using the corresponding text files.
- View the input and output images by using MATLAB scripts.
- MATLAB script must be located in the same folder with input and output text images.
- Once the script is ran, you should be seeing the detected edges of the input image, just like at Figure 4.

References:

1. http://www.songho.ca/dsp/convolution/convolution.html
2. Xilinx ISim User Guide (UG660)
3. Xilinx Vivado Design Suite Tutorial: Using Constraints (UG945)
4. Vivado Design Suite Properties Reference Guide (UG912)