Assignment: Image Filtering

Name: Arnav Tuli

Entry Number: 2019CS10424

My Design: Description and Assumptions

I have designed an Image Filter that reads an image (pixel-by-pixel) from RAM, and performs filtering using filter coefficients which are read separately from ROM. Each pixel of the filtered image is formed by multiplying an image pixel with a filter coefficient and accumulating the sum over nine products (involving all the pixels in 3x3 grid around the corresponding image pixel). Images used in the filtering process are of **QQVGA** type and therefore, have a dimension of **120 x 160 (height x width)**. I am representing the image pixel array as a 2-D matrix **X (120 rows, 160 columns)**. Each pixel in itself is an **8-bit unsigned logical vector**, and therefore, can take any value from 0 to 255. Similarly, the filtered-image pixel array is represented as a 2-D matrix **Y (118 rows, 158 columns)**. Again, each pixel is an **8-bit unsigned logical vector** (range 0 to 255). Finally, the filter coefficients are represented by a 2-D matrix **C (3 rows, 3 columns)**, where each element in itself is a **9-bit signed logical vector** (range -256 to 255). These coefficients are actually scaled up by a factor of **2^7** so that fractions can be avoided. Therefore, in my circuit, I have made sure to **right-shift** the final filtered image pixel by **7-bits** (division by 2^7) before writing it down in the memory. This way, image filtering occurs as expected. Lastly, I have also assumed that all the matrices are stored in memory (RAM/ROM) in **row-major** format. This assumption is required in determining the **memory address** (RAM/ROM) from which the data is to be read. **(relevant calculations have been done to access a particular matrix entry)**

My Design: SPECIFICATIONS: INPUT/OUTPUT

My entity (**Filter**) has the following input/output ports:

- **clk**: *std_logic*: It is the input clock signal, which is used to trigger different processes and state transitions (synchronous ASM)
- **pb**: *bit*: It is the input push-button signal that is used to indicate start (**pb='1'**) of smoothening or sharpening operation.
- **switch**: *bit*: It is the input switch signal that is used to indicate whether the image is to be smoothened or sharpened. (**switch = '0'** for smoothening and **switch = '1'** for sharpening)
- **reset**: *bit*: It is the input reset that is used to initialize my entity. (start from **Idle** state)
- **done**: *bit*: It is the output signal that is used to indicate the end (**done = '1'**) of a filtering operation.

My Design: SPECIFICATION: COMPONENTs

My entity uses three pre-defined entities which are provided to us in the appendix of assignment description. These are listed below:

- **RAM64Kx8**: This component is basically a memory array of 64,000 8-bit logical vectors, and stores the image which is to be filtered **(row-major, 0 onwards)**. This image can be read from the memory by enabling its **read_enable** signal. Also, the filtered image is written into the memory via this component **(row major, 32768 onwards).** Write operation can be done by enabling **write_enable** signal.
- **ROM32x9**: This component is a memory array of 32 9-bit signed logical vectors, and stores the scaled-up filter coefficients **(scaling factor = 2^7)**. Smooth filter coefficients are stored from **address 0** onwards **(row-major)** and Sharp filter coefficients are stored from **address 16** onwards **(row-major)**. The coefficients can be read by enabling its **read_enable** signal. This component does not facilitate writing contents into the memory (Read-Only).

- **MAC**: This component is the multiplier-accumulator that is used to carry out the filtering operation by accumulating the **nine products**, one after another, at subsequent clock cycles. Hence, using MAC properly is crucial for my **Filter** entity.

Note: All the signals used in my entity have been **fully described** in the VHDL code by means of comments. I will be using these signals in my description below. Hence, kindly **refer** to the code for any clarification regarding signals used.

My Design: OVERVIEW: FILTER: Algorithmic State Machine

I have defined a new **state_type** type, where a signal of **state_type** can take the following **fourteen values** (corresponding to a different state) :- **Idle**, **Smooth**, **Sharp**, **Smooth_Active**, **Sharp_Active**, **Smooth_Mult**, **Sharp_Mult**, **Smooth_Addr**, **Sharp_Addr**, **Read_M**, **Load_M**, **Mult**, **Write_M** and **wait_state**. In my design there exists a symmetry between smooth and sharp states, i.e., for every smooth state there is a corresponding sharp state. Hence, I will only describe the common states, and every type of "smooth" state below: (corresponding "sharp" states work in a similar fashion)

Idle: This is the **default state** or the **initial state**. Choice between smoothening and sharpening is made from this state, therefore, it is a good practice to **reset** the filter after switching from smooth to sharp or vice-versa. This way it will be ensured that pressing the push-button **(pb signal)** results in the desired filtering operation.

Smooth: This is the first state that is reached from Idle state when the switch is in **0** position (smooth filtering). The machine waits for the push-button signal while in this state, and remains in this state as long as the button-signal is **0**. When the button-signal becomes high, the state of the machine changes from Smooth to **Smooth_Active**, thereby commencing the filtering operation. Signals **I** and **J** which correspond to matrix indices (**of Y**) are initialised with the value **1**.

Smooth_Active: In this state, the filtering operation has already started, and I am trying to find the pixel value corresponding to the entry **Y[I][J]**, (**1 <= I <= 118, 1 <= J <= 158**). Therefore, before accumulating products for **Y[I][J]**, I need to make sure that the indices are valid (in range). Signals **cJ** and **cI** make the corresponding checks **(J = 159 and I = 118)**. If both the conditions are **true**, it means that the filtering of the image is **complete** as in my design I am determining pixel values **row-wise**; hence, I no longer accumulate products, but rather move to the **wait_state**. However, if one of the conditions is **false**, it means that the filtering is still supposed to be carried out. Therefore, I appropriately change the values of **I** and **J**, and load the iterators, **i** and **j** (used in **multiplying** appropriate elements from **image** and **coefficient matrix**, initialised with **0**). Another signal, **control** is loaded and set to **0**. This signal is used in **initialization** of the **MAC** with the first product that it gets. After this, machine changes its state from Smooth_Active to **Smooth_Mult**.

Smooth_Mult: This state is quite similar to Smooth_Active, as we now check for sanity of the iterators **i** and **j**, just like we checked for sanity of **I** and **J** in Smooth_Active. Again, this step is required before we **read** any data from RAM/ROM as we do not want to read from a location where the data is **not present** in the first place **(or erroneous data)**. Since, **i** and **j** are used to iterate over a **3x3** grid, **0 <= i, j <= 2**. Hence, if **j = 3** and **i = 2**, it means that my iteration is **complete** as in my design I carry out the accumulation **row-wise**. In such a situation, I store the output of the MAC (**Y_mult**) into my signal **Y** (filtered pixel) after **right shifting** and suitable **resizing** (to 8 bits). If **Y_mult** is **negative**, **zero** is stored in **Y**, otherwise, it is stored as specified previously. Also, memory address where Y is supposed to be written is assigned to the **addr_I** signal (RAM input). Since, **Y** is stored in row-major format starting from the address **32768**, the address for **Y[I][J]** is given by,

$$\textbf{address} = 32768 + (I - 1) * 158 + (J - 1); \quad 1 <= I <= 118, 1 <= J <= 158$$

Also, in this case, the machine changes its state to **Write_M**. However, if one of the conditions (j = 3 or i = 2) is **false**, then appropriate changes are made in the values of **i** and **j**, and state is changed to **Smooth_Addr**.

Smooth_Addr: In this state, as the name suggests, I **set** the address from which the **image element** and **filter coefficient** has to be read from **RAM** and **ROM** respectively. This step can only be performed after making sure that the values **I, J, i** and **j** are all **sane**. Hence, this state is introduced after the states given above separately. Since, **X** is stored in row-major format starting from address **0** in **RAM**, its address is given by,

$$\textbf{address} = (I + i - 1) * 160 + (J + j - 1); \quad 1 <= I <= 118, \ 1 <= J <= 158, \ 0 <= i, j <= 2$$

Also, smooth filter coefficients **(C)** are stored in row-major format starting from address **0** in **ROM**, therefore, its address is given by,

address = i * 3 + j;   0 <= i, j <= 2

The only difference that **exists** between Smooth_Addr and Sharp_Addr is that sharp filter coefficients are stored in ROM from address **16 onwards** (in place of 0). So, address in that case becomes equal to,

address = 16 + i * 3 + j; 0 <= i, j <= 2

After this, state is changed to **Read_M**.

Read_M: This is a **common state** and, in this state, I enable the **read_enable** signals for both **RAM and ROM**. I read and store the image pixel and filter coefficient in **X** and **C** signals respectively. Now, before I carry out the actual multiplication, I need to **resize** this data, so that it meets the requirement of the **MAC** entity. Hence, state is changed to **Load_M**.

Load_M: This is also a **common state**, and in this state, the signals **X** and **C** are **appropriately resized** and stored in the signals **X_mult** and **C_mult** respectively, which are **18 bit signed-logical vectors**. These signals are also the **inputs** of the MAC, and will result in the accumulated output, **Y_mult** in the next clock-cycle.

Mult: In this state, the product **X_mult*C_mult** is accumulated into **Y_mult** through **MAC entity**. If the **control** signal is **0**, **Y_mult** is initialised with this product, otherwise the product is **added** to its **current value**. Note that the MAC is triggered at every clock cycle, hence to avoid **unnecessary accumulations** at subsequent clock cycles (before reading in next values of X_mult and C_mult), I make the value of **X_mult** and **C_mult '0'** by **resetting** them. This way, even though accumulation takes place at every clock cycle, the result (**Y_mult**) will not change as **zero is being added to it**. This also explains why it is crucial to set the values of **X_mult** and **C_mult** exactly **one clock cycle** before the actual multiplication (accumulation) can take place. As, loading them before will result in **multiple accumulations**, and loading them right next to multiplication will **skip** their product, as the two entities **work concurrently.** Apart from resetting **X_mult** and **C_mult**, I also set the value of **control** signal to **1**, so that **accumulation** can take place in subsequent cycles. (**0** indicates initialisation and is only needed for the **first time**). Iterator **j** is also incremented by **1**, and state changes to **Smooth_Mult** or **Sharp_Mult**, depending on the **switch** signal **('0' for smooth and '1' for sharp)**.

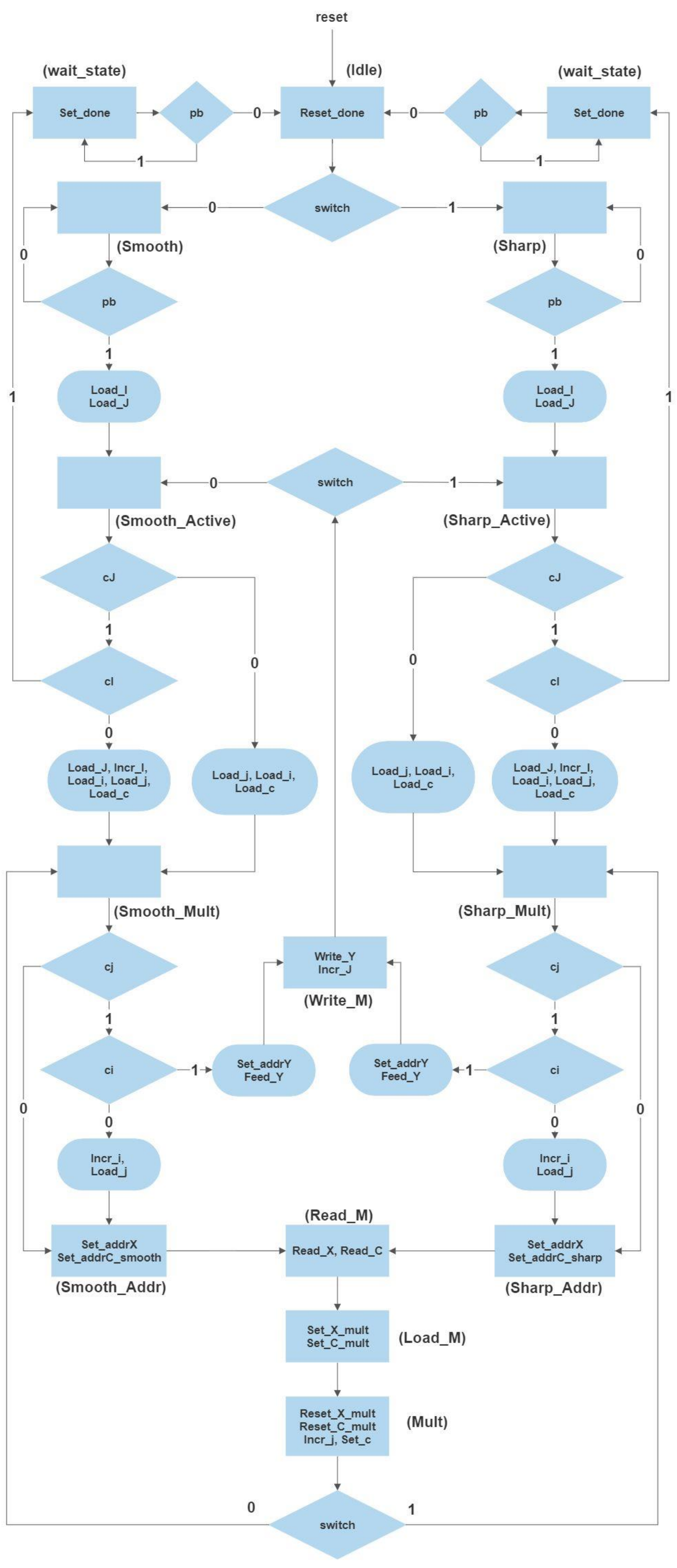
Write_M: The machine comes to this state after filtered value of a particular pixel is **determined** (i.e., **Y[I][J]**). This value is already stored in signal **Y**, which is then **written** into the RAM at the positive clock edge. Also, the value of index **J** is increased by **1**, and state is changed to **Smooth_Active** or **Sharp_Active** depending on the value of **switch** signal.

wait_state: The machine comes to this state after the filtering operation is complete (**all filtered pixels determined**), and it awaits the **release** of the push-button which was used to **start** the filtering. It sets the value of **done signal to 1** (indicating termination), and changes state to **Idle** as an when the button is released. **(synchronously, of course)**

The ASM Chart for my entity is given on next page.

ASM Chart (with control signals)
All signal names are the same as used in VHDL code.
The corresponding actions performed on data are mentioned on the next page.

Note: State names are written in () beside the rectangular boxes.



My Design: OVERVIEW: Data Management

Data is modified with change in the controller signals. I have listed down the action of all the controller signals on my data below:-

Reset_done: *Idle (Moore)*: assigns '0' to "done" signal.
Set_done: *wait_state (Moore)*: assigns '1' to "done" signal.
Load_I: *Smooth, Sharp (Mealey)*: initializes "I" signal by assigning "00000001" (1) to it.
Load_J*: Smooth, Sharp, Smooth_Active, Sharp_Active (Mealey)*: Initializes "J" signal by assigning "00000001" (1) to it.
Load_i: *Smooth_Active, Sharp_Active (Mealey)*: initializes "I" signal by assigning "00000000" (0) to it.
Load_j: *Smooth_Active, Sharp_Active, Smooth_Mult, Sharp_Mult (Mealey)*: initializes "j" signal by assigning "00000000" (0) to it.
Load_c: *Smooth_Active, Sharp_Active (Mealey)*: initializes "control" signal by assigning '0' to it.
Incr_I: *Smooth_Active, Sharp_Active (Mealey)*: increments the count/value of "I" signal by 1.
Incr_i: *Smooth_Mult, Sharp_Mult (Mealey)*: increments the count/value of "I" signal by 1.
Set_addrY: *Smooth_Mult, Sharp_Mult (Mealey)*: sets the address into which filtered pixel "Y[I][J]" is to be written.
Feed_Y: *Smooth_Mult, Sharp_Mult (Mealey)*: assigns the final output of MAC ("Y_mult") to "Y" after appropriate resizing/right-shifting and modifications (if "Y_mult" < 0, "Y" is assigned 0).
Write_Y: *Write_M (Moore)*: writes down the filtered pixel into RAM at designated memory location.
Incr_J: *Write_M (Moore)*: Increments the count/value of "J" signal by 1.
Set_addrX: *Smooth_Addr, Sharp_Addr (Moore)*: sets the address from which the image pixel "X[I + i – 1][J + j – 1]" is to be read (from RAM).
Set_addrC_smooth: *Smooth_Addr (Moore)*: sets the address from which the smooth filter coefficient "C[i][j]" is to be read (from ROM).
Set_addrC_sharp: *Sharp_Addr (Moore)*: sets the address from which the sharp filter coefficient C[i][j] is to be read (from ROM).
Read_X, Read_C: *Read_M (Moore)*: reads the image pixel and filter coefficient from the designated address in RAM and ROM respectively.
Set_X_mult, Set_C_mult: *Load_M (Moore)*: assigns the value of image pixel and filter coefficient to "X_mult" and "C_mult" respectively after proper resizing.
Reset_X_mult, Reset_C_mult: *Mult (Moore)*: assigns a "0" to both "X_mult" and "C_mult" so that they do not accumulate into "Y_mult" unnecessarily. (till next image pixel and filter coefficient are read)
Incr_j*: Mult (Moore)*: increments the count/value of "j" signal by 1.
Set_c: *Mult (Moore):* sets the "control" signal to '1', so that "Y_mult" gets accumulated with products from next clock cycle onwards. ("control" = '0' is used to initialise "Y_mult" with input signal product)