**Hardware IPs Implementation (Jingyi Liang)**

In order to fully utilize the benefit of the Pynq-Z1 board, different tasks are assigned to implementation in either Programmable logic (PL) or Processing system (PS) according to their properties. Since PL is good for intense data computation, we decided to implement the bottleneck of the application on hardware. The detailed design process of the hardware part is illustrated below.

To start with, we ran a profiler on the python application and identified the bottleneck of the application is the activation function Relu in the ESPCN network. For easy implementation in software and better acceleration, we decided to implement the first convolutional layer in hardware using Vivado 2022.2 instead of just the Relu function.

## All the hardware custom IPs are parameterized for easy fit into different situations, for example user may use different size of image or word length for the same network. The source code can be found in GitHub Super\_Resolution\_Hardware\_IPs directory.

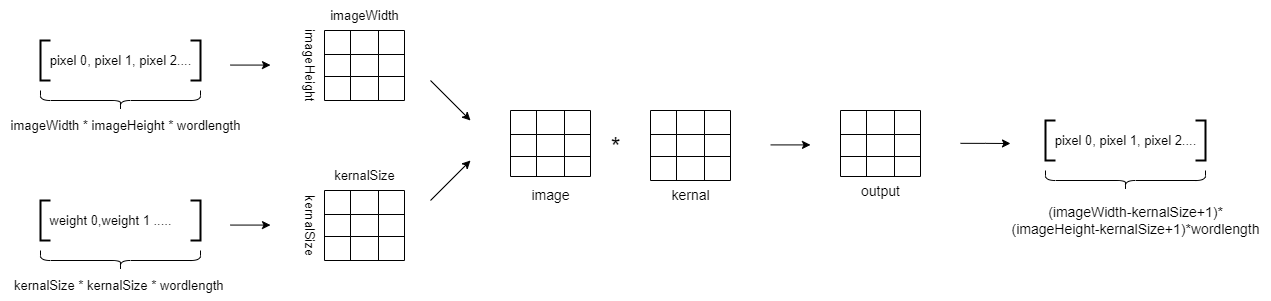


**2D Convolutional Layer**

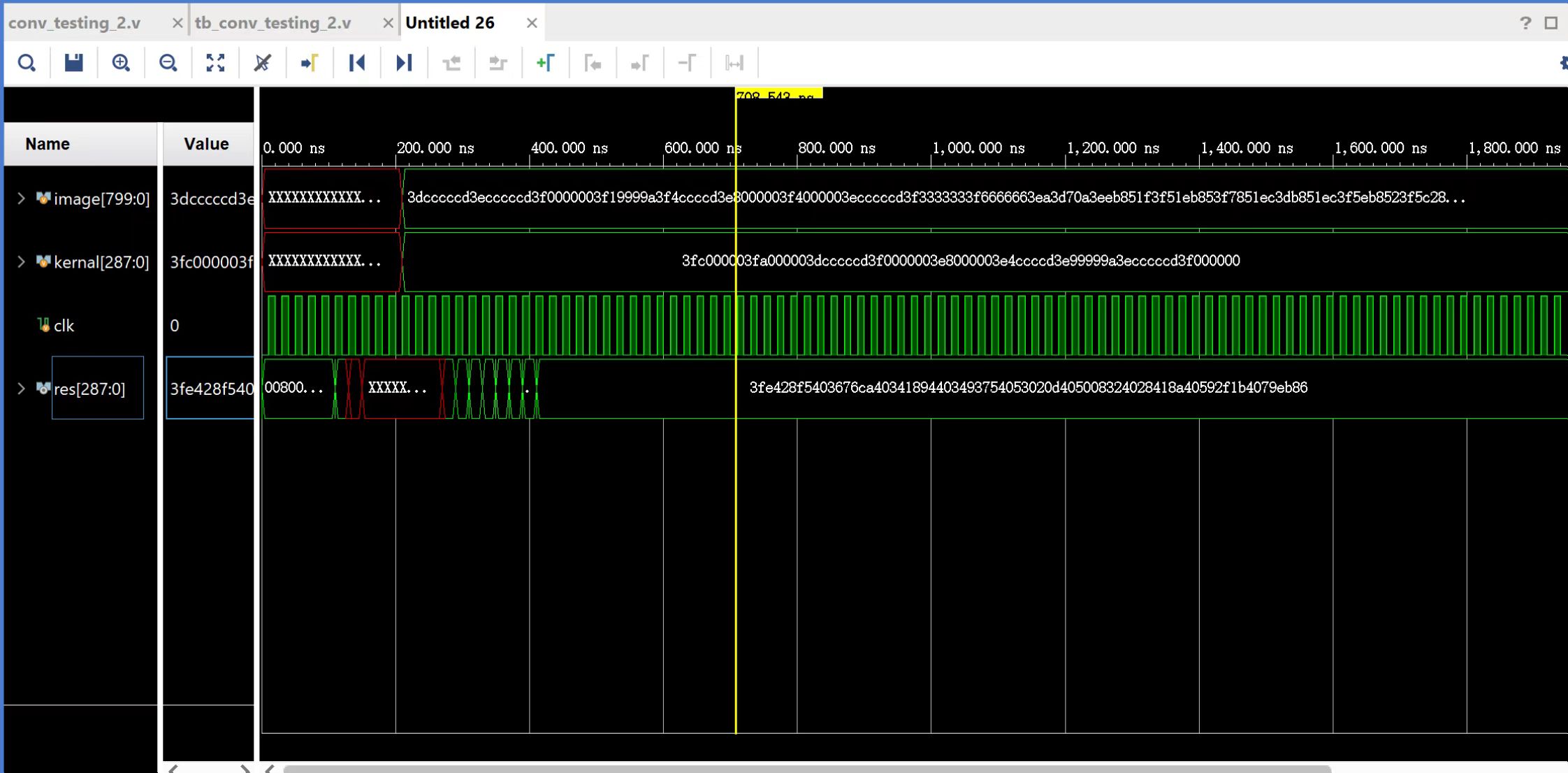
The idea of a 2D convolutional layer is that the convolution filters slide over the 2D input data and perform an elementwise multiplication, the results will then be summed up into a single output data.

*Figure 1* shows the steps of the Conv\_layer.v IP, and the explanation of each step is shown below.

* **Input:** The custom conv\_layer.v IP takes in a 1d array of size imageWidth \* imageHeight \* wordlength.
* **Convert:** It will be converted into a 2d array to be ready for convolution operation.
* **Convolution:** Then, floating-point multipliers are instantiated to perform an elementwise multiplication, and floating-point adders are used to sum up the results to give the 2D output.
* **Output:** Eventually, the 2d output array will be converted back to a 1d array since verilog does not support 2d array as ports of modules, the size of the output array will be (imageWidth – kernalSize + 1) \* (imageHeight – kernalSize + 1) \* wordlength.

 *Figure 1: 2D Convolutional Layer*

*Figure 2* shows the simulation result of the Conv\_layer.v IP. The test data are shown in the image, the input image has size 5\*5, and the kernel has size of 3\*3, the wordlength of both kernel and input image are 32-bit floating-point number. Therefore, the output is expected to be a 3\*3 image of wordlength 32-bit floating-point number. The simulation results matched the manual calculation result, so the operation of the custom IP is proved correct.



*Figure 2: Simulation result of 2D Convolutional Layer*

**Activation Function – ReLU**

After convolution, a Relu activation function is applied to the output data to introduce non-linear properties to the neural network. The Relu activation function is very simple, the curve of the function is shown below. While data is greater than zero, it is set to itself, else it is set to zero.

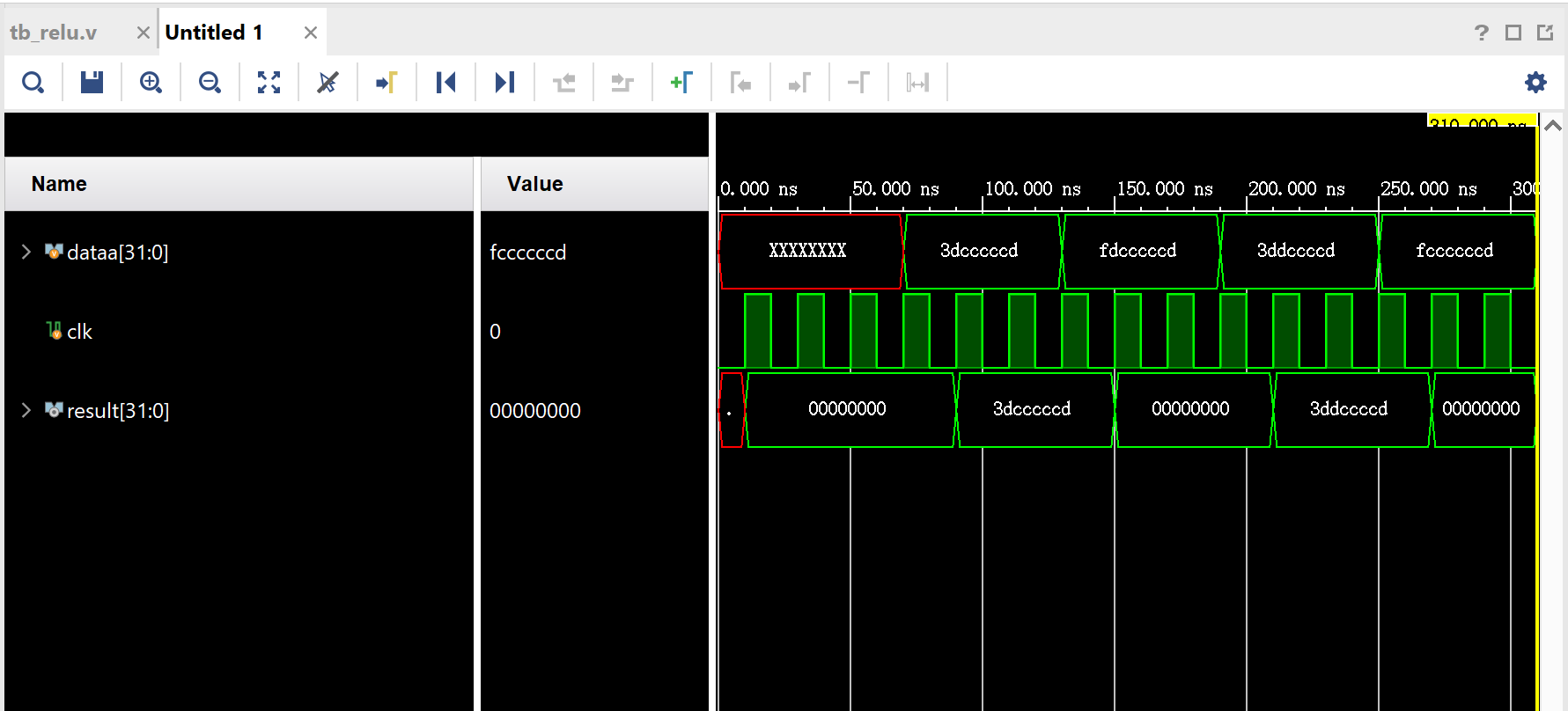
*Figure 3* shows the plot of the ReLU function, and the explanation of each step is shown below.

* Since pixels are signed 32-bit floating-point, the most significant bit represents sign.
* If the MSB is 0, it is set to itself.
* If the MSB is 1, it is set to zero.



*Figure 3: ReLU*

*Figure 4* shows the simulation result of the ReLU.v IP. The test data are shown in the image, the inputs are 32-bit floating-point numbers, as expected when MSB of the number is 0, the output is itself, otherwise, it is zero.



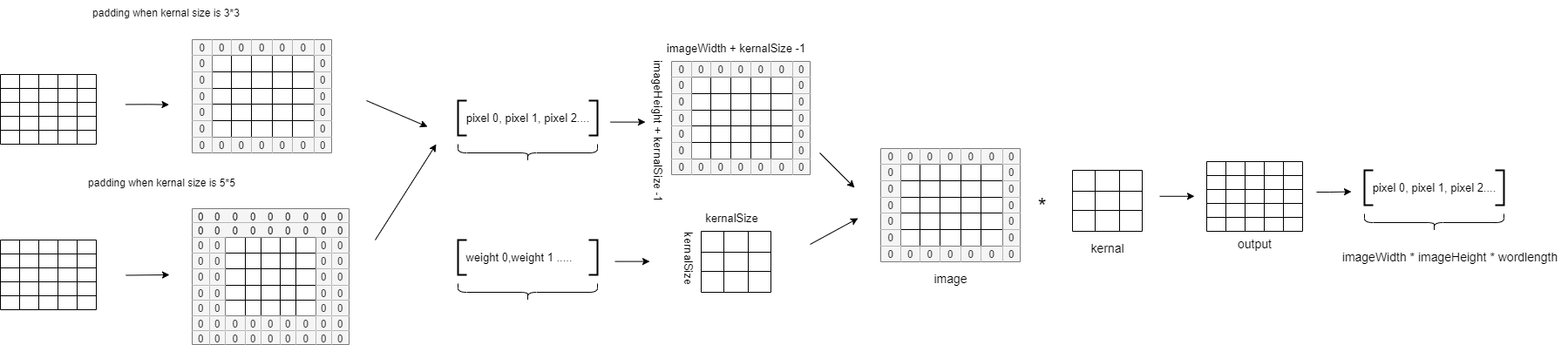
*Figure 4: Simulation result of ReLU activation function*

**Padding**

The ESPCN neural network has the "same" padding, and the stride is one, which means the output image size will be the same as the input image size. Since every time we do convolution, the size of the image will get smaller, an extra padding.v IP is implemented to preserve the original size of the image. When padding, a fixed number of extra pixels are added to the surrounding of the image before processing to convolution, the value of the extra pixels is zero. The number of extra pixels depends on the size of the input image and the size of the kernel.

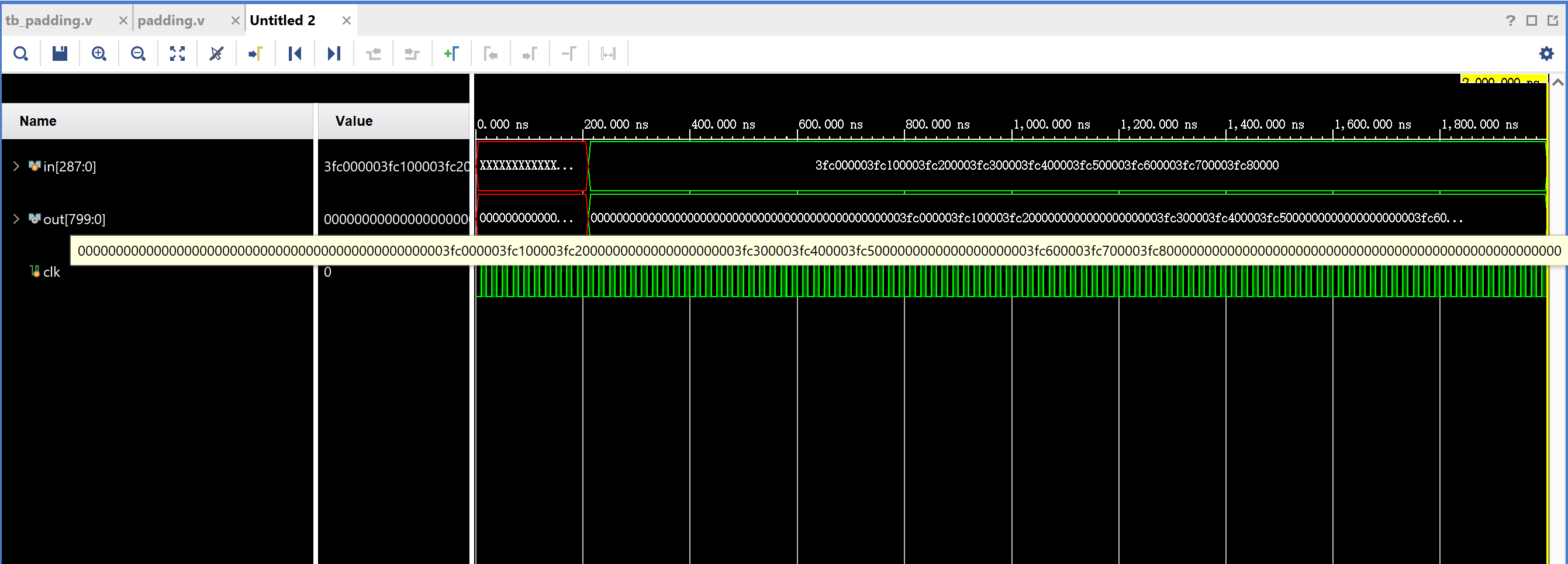
*Figure 5* shows the way padding.v IP and conv\_layer.v IP connected, and the explanation of the padding.v IP is shown below.

* The ESPCN neural network has kernel sizes of 3 or 5, the sizes of the kernels are odd number.
* For kernel size of 3, one layer of pixel of zeros are added to the surrounding of the image.
* For kernel size of 5, two layers of pixel of zeros are added to the surrounding of the image.



*Figure 5: 2D Convolutional Layer with Padding*

*Figure 6* shows the simulation result of the padding.v IP. The test data are shown in the image, the input image has size 3\*3, and kernel has size 3\*3, the wordlength of both kernel and input image are 32-bit floating-point number. Therefore, the output is expected to be a 5\*5 image of wordlength 32-bit floating-point number. As shown in the figure, the surrounding pixels of the image are zero which is what we expected, so the operation of the custom IP is proved correct.

 *Figure 6: Simulation result of Padding*

**Input and Output**

Pixel data are transferred between FPGA and CPU using AXI stream, it is always used for high-speed streaming data. Since there are limited amount of IOs available on PYNQ Z1 board, one pixel is read or write per one cycle. We can proceed to the next step until all the pixels in the image are read. The FSM\_wrapper.v IP takes care of the input data, and the splitting.v IP takes care of the output data.

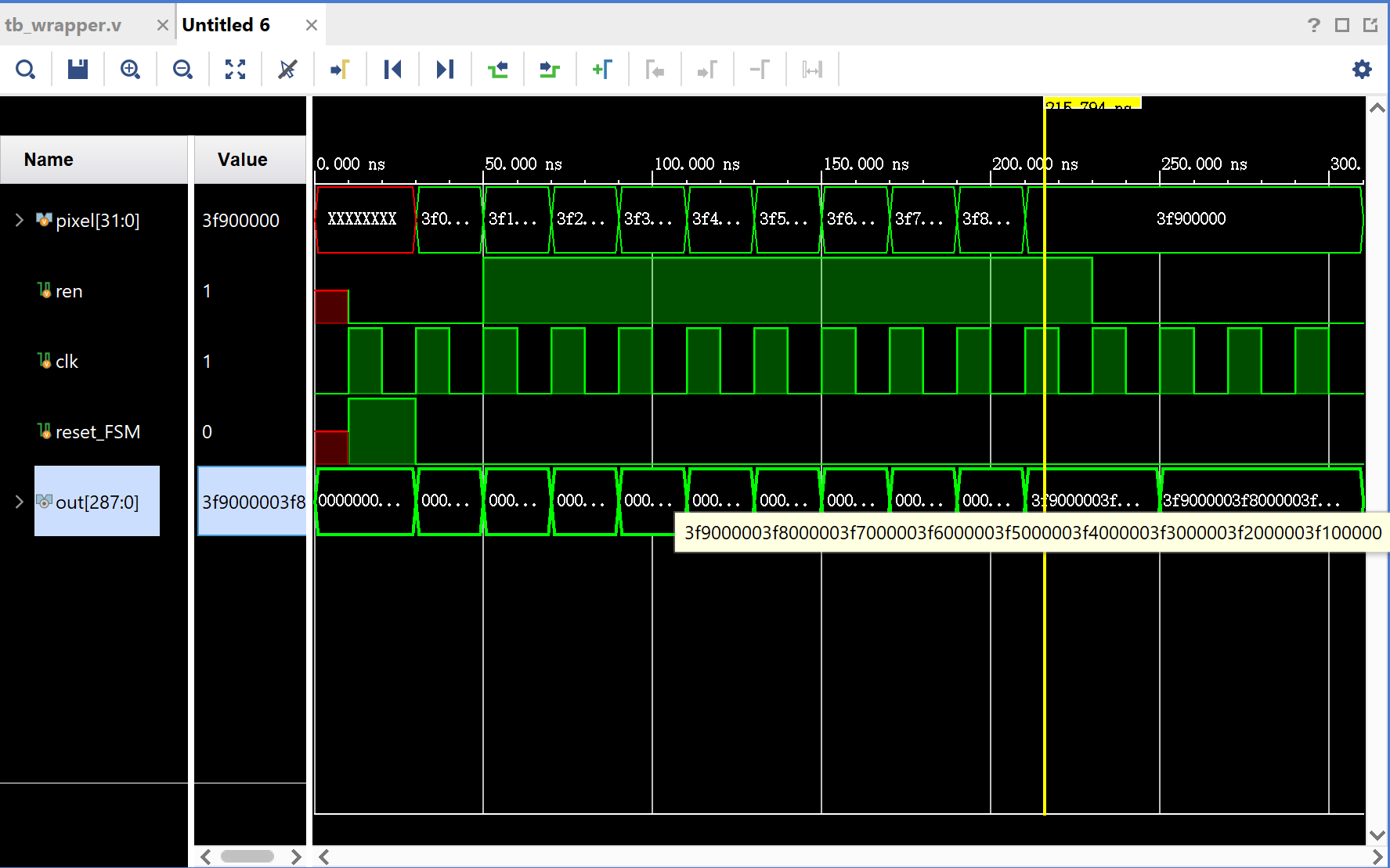
**FSM\_wrapper.v**

* The input stage is implemented using custom FSM\_wrapper.v IP, it consists of a counter and a state machine.
* The state machine has three states A, B and C, state A is the idle state, once the read enable signal is high, it goes to state B. Then if the done signal is low, it will go to state C. At state C, data are read into the storage register sequentially through AXI stream.
* The counter counts the number of pixels that have been read into the register. Until all the data are read, the done signal goes high, and the state machine goes back to state A.

**Splitting.v**

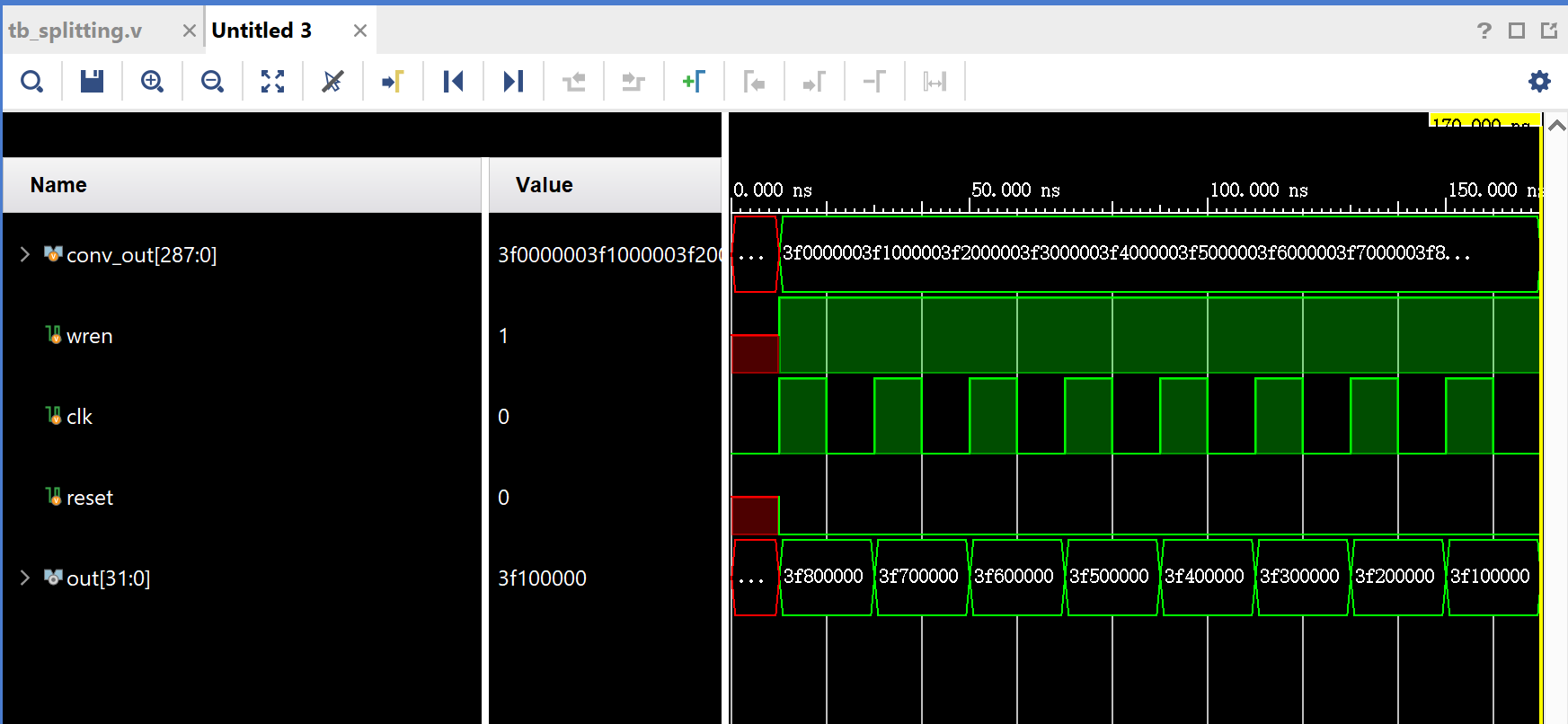
* The output stage is implemented using splitting.v IP, it consists of a counter.
* While write enable signal is high, data are read back to memory sequentially through AXI stream.
* Until all the output pixels are written to the memory, the write enable signal goes low, the memory stops receiving data.

*Figure 7* shows the simulation result of the FSM\_wrapper.v IP, the register value is initialized to 0, then 32-bit floating-point pixels are read one by one.



*Figure 7 ：Simulation result of input stage*

*Figure 8* shows the simulation result of the splitting.v IP, 32-bit floating-point pixels are read from the register and written to the memory one by one.

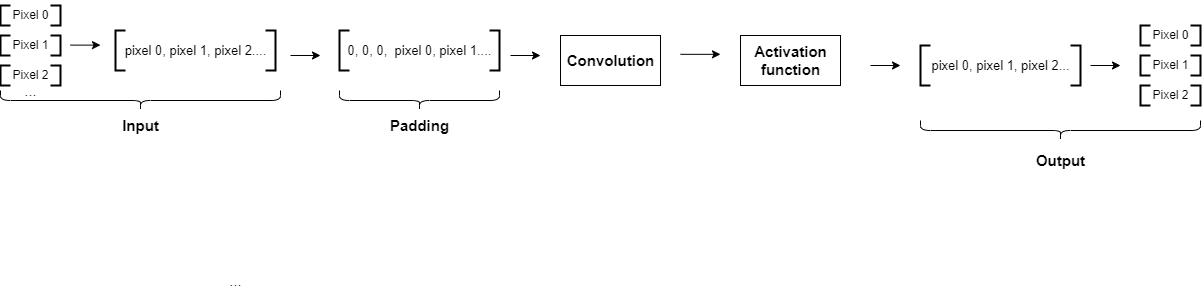


*Figure 8 ：Simulation result of output stage*

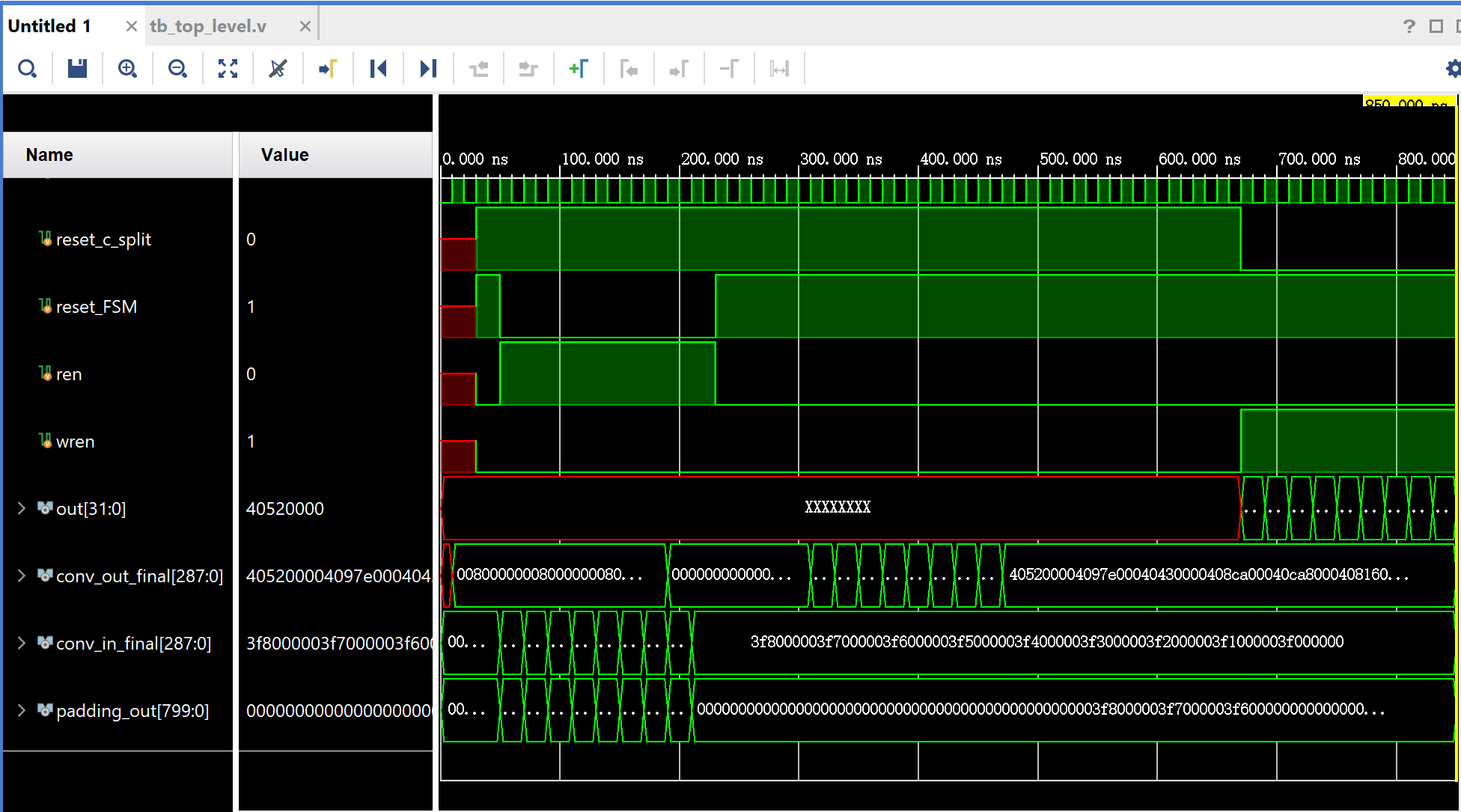
**Integrate all the custom IPs**

All the custom IPs are integrated into the top\_level.v, apart from these custom IPs, I also used Xilinx floating-point multiplier and floating-point adder. The architecture of the top\_level.v IP is shown in *Figure 9*.

It first takes in one pixel at a time, when all the pixels of the image are stored in the register, it processes to the padding stage. Pixels of zero value are added to the surrounding of the image, then the input image is convolved with the kernel. After convolution, the activation function ReLU is applied to the output data to introduce non-linear properties to the network. Eventually, the output pixels are written back to memory sequentially.

 *Figure 9 ：Integrated block of hardware design*

*Figure 10* shows the simulation result of the top\_level.v IP, the output pixel values matched the manually calculated values.



**Future work**

**Pipelining** - In order to optimize the accelerator, pipelining method can be used. The convolutional layer can be split into three stages, including reading pixels, processing pixels and writing pixels. Since reading and processing are using different parts of hardware, we can start taking in the pixels of the next image before the current instruction is completed.

**Different ways of implementing the 2D convolutional layer –** The current method is not perfect because we can only do convolution until reading in all the pixels value. According to Nelson Campos’s article, to perform a kernel convolution of 3x3 dimensions, the minimum number of pixels required is 2 lines of the image, by knowing the time delay of reading each pixel, the kernel can convolve with pixels in function of time.

**Reference**

# [1] “Convolutional Neural Network” by Arc. <https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05>

# [2] “An Overview of ESPCN: An Efficient Sub-pixel Convolutional Neural Network” by zhuo cen. <https://medium.com/@zhuocen93/an-overview-of-espcn-an-efficient-sub-pixel-convolutional-neural-network-b76d0a6c875e#:~:text=ESPCN%20can%20be%20seen%20as,the%20last%20pixel%20shuffle%20stage>.