CSE 240D Final Report

A Systolic Array-based Single Convolution Layer in SystemVerilog

William Bao
A10043115
wbao@ucsd.edu

Dylan Vizcarra
A13211917
dvizcarr@ucsd.edu

Table of Contents:

Background and Design Specification ………………………………………………………….. 2

Hardware Architecture ……………………………………………………………………......3

Overview ……………………………………………………………………………….... 3

Instruction FIFO ………………………………………………………………………..... 4

Data Memory ………………………………………………………………………………4

Controller …………………………………………………………………………….... 5

Systolic Array …………………………………………………………………………… 6

Pixel Requester ……………………………………………………………………………8

Scratchpad …………………………………………………………………………………9

Prefetching Data ………………………………………………………………………...10

Instruction Set ……………………………………………………………………………… 12

Load Filter (LF) ……………………………………………………………………..... 12

Load Size (LS) …………………………………………………………………….. 12

Load Image (LI) ………………………………………………………………………... 13

Do Convolution (DC) …………………………………………………………………....13

Simulation Framework ……………………………………………………………………. 14

Overview …………………………………………………………………………….... 14

Example Simulation Flow ………………………………………………………………15

Performance ………………………………………………………………………………...18

Appendix …………………………………………………………………………………… 19

Background and Design Specification

This design is a systolic array-based convolution layer accelerator coded using SystemVerilog that is fully synthesizable and functionally verified. It is intuitively structured to be easily improvable and thus can be used as the starting design for future improvements. The design performs convolution using a sliding-window method by sliding kernel windows over an image from the top left pixel of an image to the bottom right pixel of an image. It supports the following inputs and features:

- Support for up to 16x16 kernel size.
- Support for up to 16 kernels.
- Support for up to 4096x4096 monochrome images.
- 8-bit data input, 8-bit signed weights (3 bits of decimal).
- Autonomous operation; once convolution is executed, host can leave accelerator to perform convolution autonomously.
- Autonomous (DMA-like) data fetching.
- A local scratchpad for storing fetched data.
- Prefetching of future predicted data to minimize data-dependency stalls.
- A configurable instruction set for easy programming of operation.

When performing computations, accuracy is preserved until the final output (i.e while the computed output is 8 bits, the intermediate values are allowed to grow in bit size). Rounding and clamping is performed at the very end to get rid of decimal bits and clamp the produced output to an 8-bit unsigned pixel value.

This design suffers from the following omitted features. Some of these features were purposely omitted so that future designs could improve upon the base design; possible approaches to implement these features are discussed later.

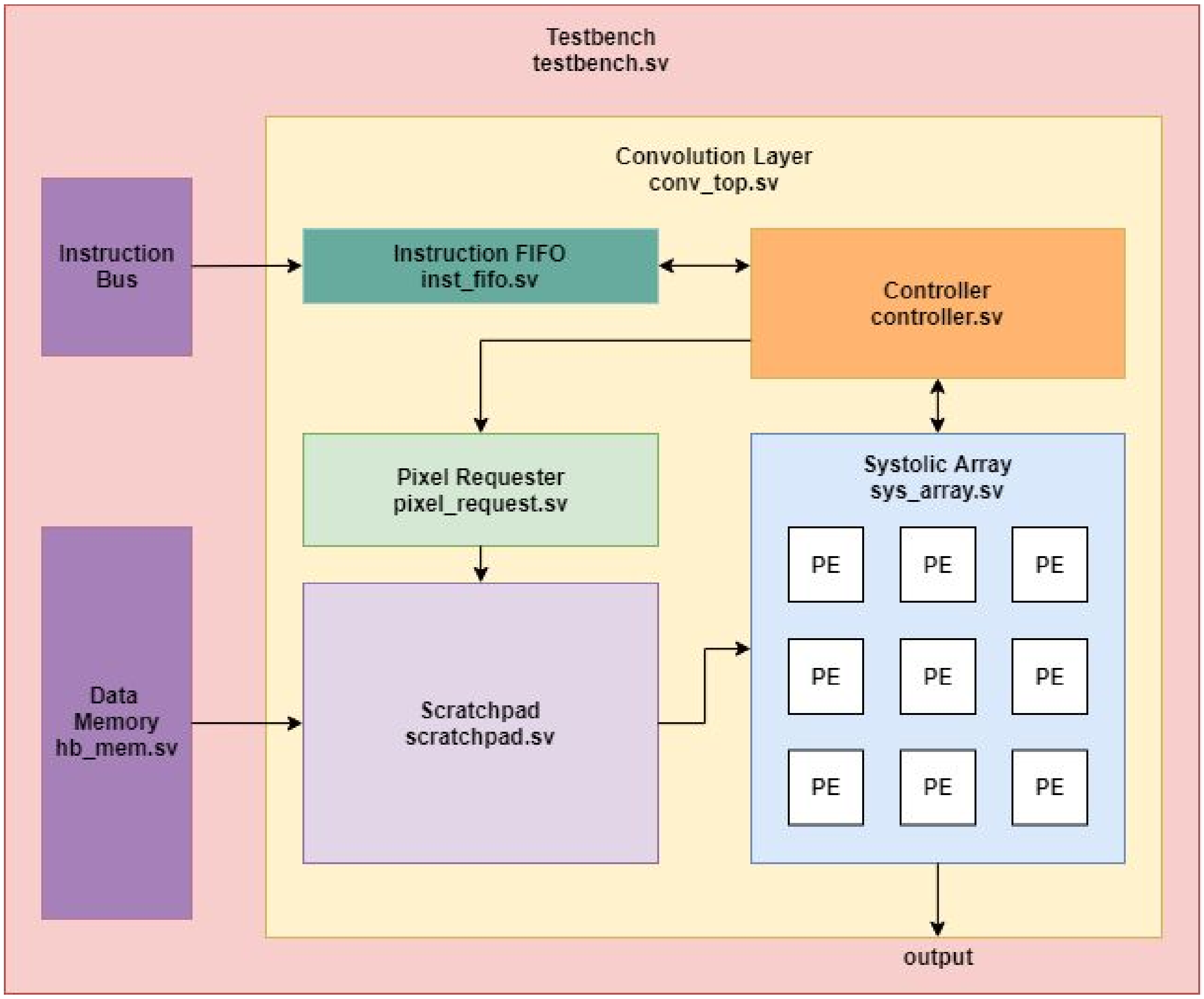
- No support for multi-channel input (e.g. RGB).
- No inherent support for zero-padding or other padding options; workaround supported.
- No support for variable (>1) step size.

Hardware Architecture:

Overview:

Below is an high-level overview of the overall architecture. Within the convolution layer, there exists five main parts:

- An instruction FIFO buffer that constantly fetches new instructions from the host instruction bus if a given instruction is a part of the convolution layer's ISA.
- A controller that contains the main state machine which executes a given convolution request from the host.
- A pixel requester that can request up to 16 pixel values from memory.
- A scratchpad which services and stores memory values that have been requested.
- A 2D systolic array that performs 2D convolution upon given pixels when given proper inputs.



Instruction FIFO:

The instruction FIFO is an 8-deep synchronous write, asynchronous read FIFO. On the write side, instructions are clocked in automatically every cycle if the instruction op code corresponds to an instruction that is part of the convolution layer's ISA. The read side is serviced by the controller and a FIFO empty signal indicates to the controller that there are no queued instructions to be serviced.

For more information on the ISA, please refer to the *ISA* section.

Data Memory:

The data memory is a high-bandwidth 256-bit memory. Since each pixel is a byte of data, each address corresponds to storage of 32 pixels. The memory exists as part of the host and is an input to our convolution layer; though it does not exist inside our convolution layer, it still vital for our layer's operation since it contains all image and filter data. Our testbench implementation does not have write capability; it is a read-only memory that is pre-loaded using SystemVerilog's *$readmemh* function. To adjust the default memory delay of the testbench, please change the value of *MEM_DELAY* within *conv_top.svh* in the *source* folder. *MEM_DELAY* is the number of delay cycles (in n-1 format) that it takes between sending a requested memory address and the data being available at the output of the memory address; clocking in the data and sending the address each take an additional cycle (e.g. if the value of *MEM_DELAY* is 7, there are 7 + 1 = 8 clock cycles between the send and receive cycles, meaning the total number of cycles for a memory access is 8 + 1 + 1 = 10 cycles).

Image and filter data is assumed to be stored byte-by-byte sequentially in data memory (with each address storing up to 32 sequential bytes) and can be stored at any starting address as long as the data does not overlap. If utilizing our provided simulation framework, the Python model will only accept an image that starts at an address that is after the filter's starting address; this is a constraint of the Python model, not a hardware constraint.

For examples on data memory format, please view *mem_doc.txt* within the *source* folder.

Controller:

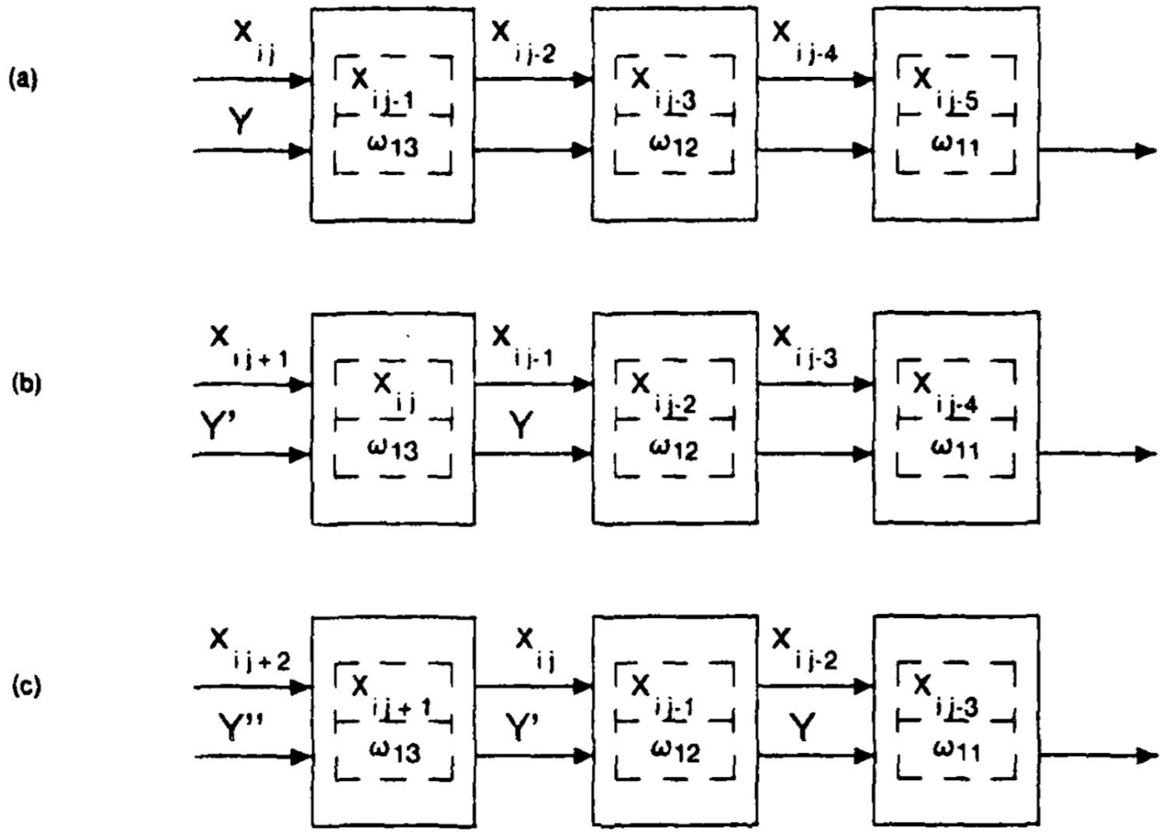
The controller's main function is to service instruction requests from the instruction queue/FIFO. It contains an instruction decoder that decodes incoming instructions from the FIFO, stores relevant information (e.g. image size, kernel size, starting addresses) and flags what instructions have been previously executed. The controller also contains the main state machine with the following states:

- IDLE state: In this state, no convolution is being done because the state machine has not received a DC (do convolution) op code. The instruction decoder will still store relevant information from other instructions but the state machine will never leave this state even if a DC op code is seen if the decoder's instruction flags do not indicate that all relevant information necessary for a convolution have been received.
- FILTER_LOAD state: In this state, a column of the filter is being loaded into the systolic array.
- FILTER_ZERO state: In this state, a column of zeros is being loaded into the systolic array. This state is necessary because data that is input into the systolic array travels across the processing elements at half the clock rate so it is necessary to alternate between this state and FILTER_ZERO when loading in a filter. Refer to the *Systolic Array* section for more information on why this is necessary.
- FILTER_SAVE state: In this state, the filter is saved into the systolic array and kept stationary at the corresponding processing elements.
- IMAGE_LOAD state: In this state, the actual convolution is done by shifting in image data. In this state, the controller will continuously request to load a new pixel of the image into the systolic array, starting from the top left of the pixel to the image to the bottom right pixel of the image.
- ITERATION_DONE state: In this state, all of the image has been loaded into the systolic array but we need to wait for the last pixel of the image to exit the pipeline in order to load the next filter or finish the convolution; thus, this is simply a wait state before starting the next iteration.

The full intricacies of the state machine's output control signals cannot be captured here; the state machine is too complex for an easy-to-digest state diagram. We recommend the reader to analyze the *controller.sv* source code under the *source* folder for a better understanding of the control signals generated by the controller.

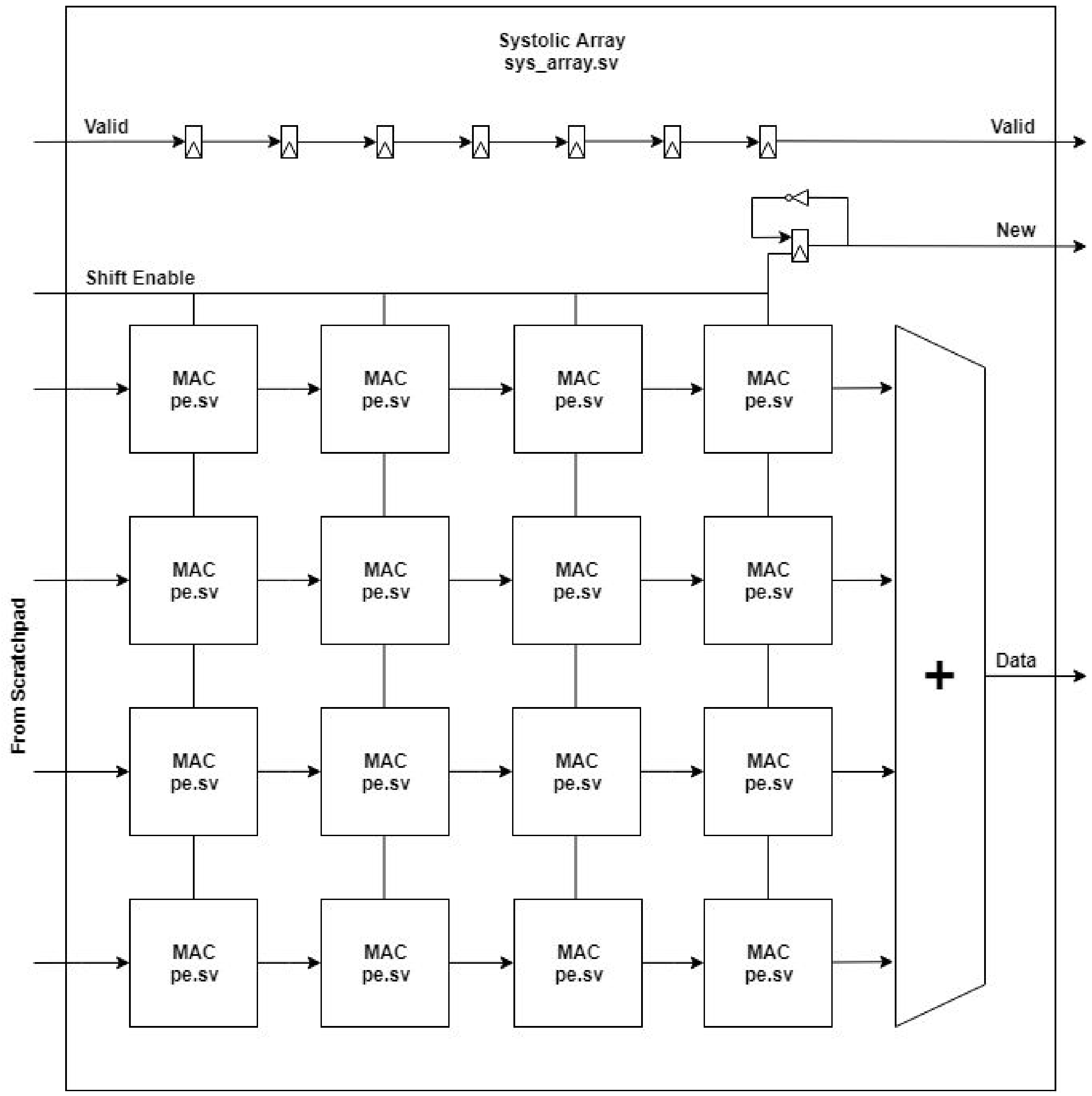
Systolic Array:

The architecture behind our systolic array is based off of the work of Kung and Song[1] in which a 1D convolution approach is adapted for 2D convolution by breaking apart a 2D convolution into rows and performing 1D convolution on each of these rows. Each of these 1D convolutions is done with a chain of processing units (PEs) that can perform multiply and accumulate (MAC) operations and are capable of storing a constant weight. In our approach, weights are initially shifted in and then kept stationary at the proper MAC; the input data stream is then is clocked in at half the rate of the accumulation stream in the same direction. This simple method produces one accumulated output at the end of the chain per clock cycle.



In the above example from Kung and Song, in time frames *a*, *b*, and *c*, weights *ω* are kept stationary while data inputs *x* are shifted in at half the rate of accumulation outputs *y*. This is accomplished in our design by placing additional D flip flops between PEs but only for the data stream and not the output stream.

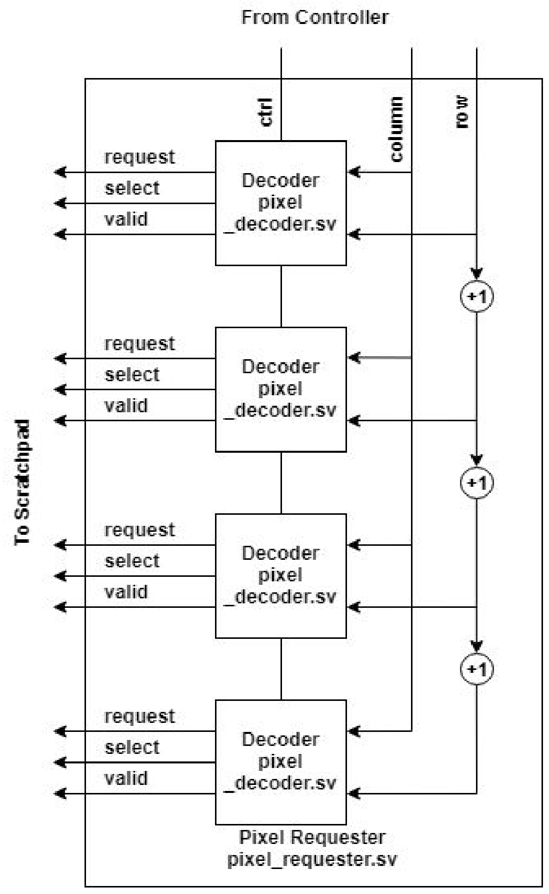
[1] H.T. Kung, S.W. Song: A Systolic 2-D Convolution Chip, in Multicomputers and Image Processing: Algorithms and Programs, ed. by K. Preston, Jr. and L. Uhr (Academic, New York 1982) pp.373-384



Here is a simplified view of our systolic array architecture. It contains rows of PE's that receive shifted inputs each time the scratchpad is ready to supply the proper pixel data at the input. Note that the above example is a 4x4 grid while our actual implementation is a 16x16 grid that supports up to 16x16 size kernels. During operation, as the systolic array travels from the right side of a row to the left side of the next row, this wrap-around computation produces invalid data; to solve this, a valid signal from the controller is also propagated in the systolic array, staying with the corresponding pixel until it reaches the output. Since the pipeline can stall for multiple cycles, in order for the output to know a new pixel has been produced, a new pixel signal at the output flips every time the pipeline is shifted.

Pixel Requester

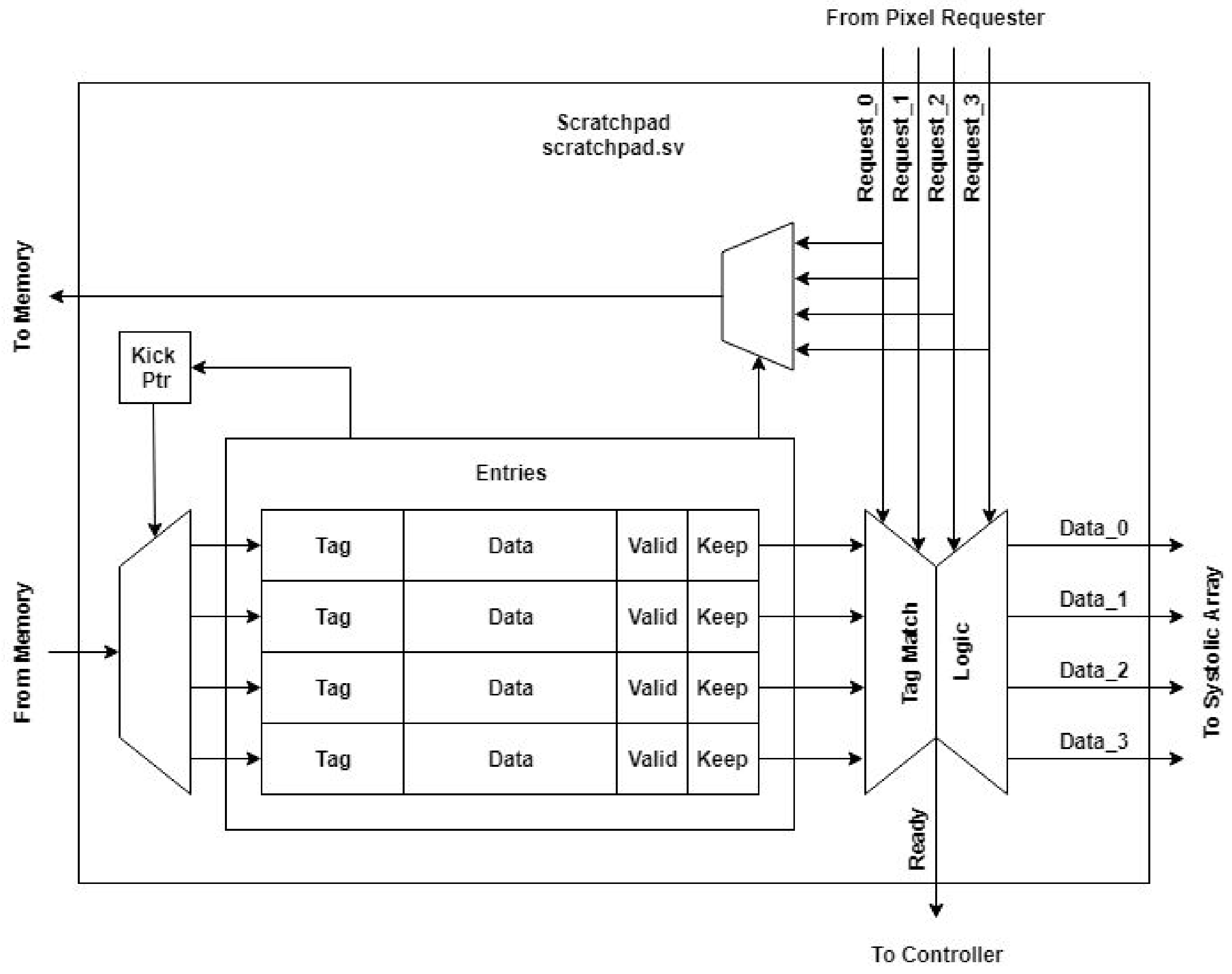
The pixel requester takes the row and column of a pixel as the input and generates up to 16 parallel address requests for up to 16 pixels; while the topmost request corresponds to the input pixel, the other requests correspond to pixels directly below the input pixel (e.g. if the input requests row 0 and column 0, the pixel requester will generate requests for row 1, row 2, row 3, etc. all in column 0). Each of these requests is computed by a decoder unit that outputs a request given an input pixel and dimensional information about the image and filter.



Above is a simplified view of a pixel requester that can generate up to 4 parallel requests. Generated requests have 3 components - a requested address, a requested data select, and a valid request signal. The requested address corresponds to the address in memory where the data for a requested pixel exists. Since the memory is a 256-bit memory, each address stores 32 pixels and thus a data select is necessary to select the corresponding byte for a pixel. Lastly, a valid signal lets the scratchpad know if a given request is valid. If the kernel size is less than the maximum 16x16, then requests past the dimensions of the kernel are not valid and should not be presented to the systolic array (e.g. if the kernel size is 3, only the top 3 requests are valid and the bottom 13 are invalid). If a request is invalid, the systolic array will present zeros to the systolic array for the corresponding request.

Scratchpad

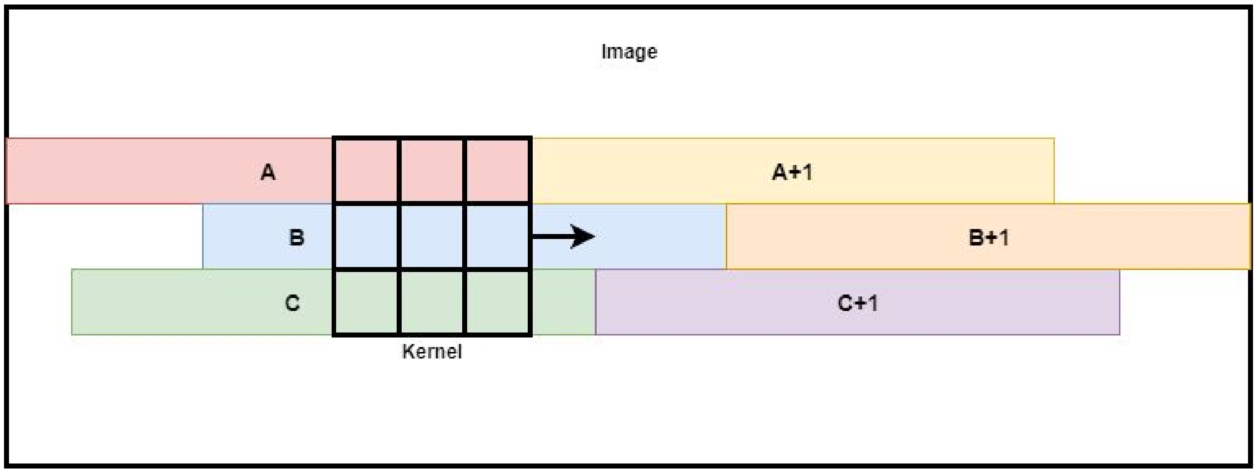
The scratchpad is a 16-entry fully associative cache with the capability to prefetch image data that it predicts the pixel requester will request.



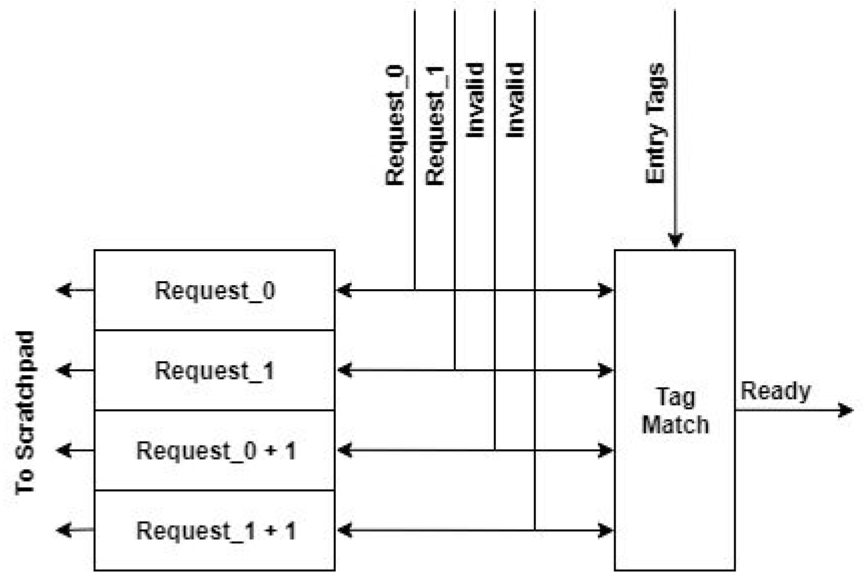
Above is a simplified view of the scratchpad assuming a 4 entry scratchpad. Each entry of the scratchpad has 4 components - a tag (20 bits), a valid bit, and a keep bit, and the actual data (256 bits). Since the scratchpad is fully associative, the tag is the full address of the associated data. The valid bit indicates whether the associated data is valid data, and the keep bit indicates whether an entry should be kept. This is determined by whether a current incoming request matches a tag that exists in the scratchpad; if there is a match, that means a current request needs the data and the data cannot be evicted from the scratchpad. A kick pointer is generated by picking the first entry that does not need to be kept; LRU or an advanced replacement policy is not enforced in our current implementation but is something to explore for future implementations. A request to memory is generated by observing which incoming requests have not been serviced by matching tags in the scratchpad. If all valid input requests have been serviced, then a ready signal is sent to the controller.

Prefetching Data:

Prefetching of image data is possible due to the predictable nature of the data being fetched; since all of the image exists sequentially in memory, sequential memory addresses also contain sequential pixel data. Consider the example below of a sliding window convolution with a 3x3 kernel across an image. Even though the kernel is currently requesting image data from address blocks A, B, and C, we can predict that the kernel will next request to utilize data from address blocks A+1, B+1, and C+1 as it moves to the right. Thus, if we can correctly prefetch address blocks A+1, B+1, C+1, we can avoid the majority of data dependency stalls.



To enable data prefetching in our design, we abuse empty request slots by placing prediction requests in them with guesses based off of addresses in slots that are currently in use. Since up to 16 parallel requests can be made, as long the kernel size is smaller than 16x16, then some of those request slots will be invalid requests. For example, if a kernel is 3x3, then no matter what the image size is, only 3 of the 16 requests will be valid since only 3 new pixels are needed to be fed in to continue performing a sliding window convolution. Thus, the scratchpad overwrites invalid requests with useful ones.



Consider the above example with a scratchpad that can service up to 4 parallel requests but only a 2x2 kernel currently being used. Therefore, two of the requests are invalid requests. Instead of not delivering these requests to the scratchpad, we instead intercept them and override them with useful requests. For address *Request_0*, we predict that as the convolution window is shifted, *Request_0 + 1* will eventually be used, which is why we override an invalid request slot with *Request_0 + 1*; similarly, we also present *Request_1 + 1* to the scratchpad as a valid request. Even though these prediction requests are seen as valid to the scratchpad and are serviced as such, the systolic array should not see them as valid since they are predictions and waiting for predictions to be serviced would cause stalls. Therefore, outputs such as data outputs and the data ready signal (which indicates that output data is ready to be processed by the systolic array) do not depend on prediction requests.

For more on prefetching performance, please refer to the *Performance* section.

Instruction Set

Our convolution uses a 32-bit instruction width with four supported instructions shown below. Image and filter/kernel information must first be loaded using LF, LS, and LI before the DC instruction can be called, otherwise a DC instruction will be ignored by the layer. From Load Filter, we obtain the kernel size, the number of filters that will be passed through our systolic array along with the starting address of the filter data as its stored in memory. From Load Size, we obtain the height and width of the image up to 4096x4096. Load Image gives the starting address of the image data. Finally, Do Convolution will execute an autonomous convolution across the image data for each of the input filters. Our assembler supports comma-delimited and space-delimited assembly code input.

Load Filter (LF):

Assembly format:

```
LF $kernel_size, $#_of_filters, $starting_address
```

Example:

```
LF 3, 3, 5 //3x3 kernel, 3 total kernels, address offset of 5
```

Machine code:

[31:28]	[27:24]	[23:20]	[19:0]
4 bits	4 bits	4 bits	20 bits
Opcode	Kernel Size	# of Filters	Starting Address

Load Size (LS):

Assembly format:

```
LS $image_height, $image_width
```

Example:

```
LS 500, 1000 //an image with a height of 500 and width of 1000
```

Machine code:

[31:28]	[27:24]	[23:12]	[11:0]
4 bits	4 bits	12 bits	12 bits
Opcode	NOP	Image Height	Image Width

Load Image (LI):

Assembly format:

```
LI $starting_address
```

Example:

```
LI 7 //address offset of 7
```

Machine code:

[31:28]	[27:12]	[19:0]
4 bits	8 bits	20 bits
Opcode	NOP	Starting Address

Do Convolution (DC):

Assembly format:

```
DC
```

Example:

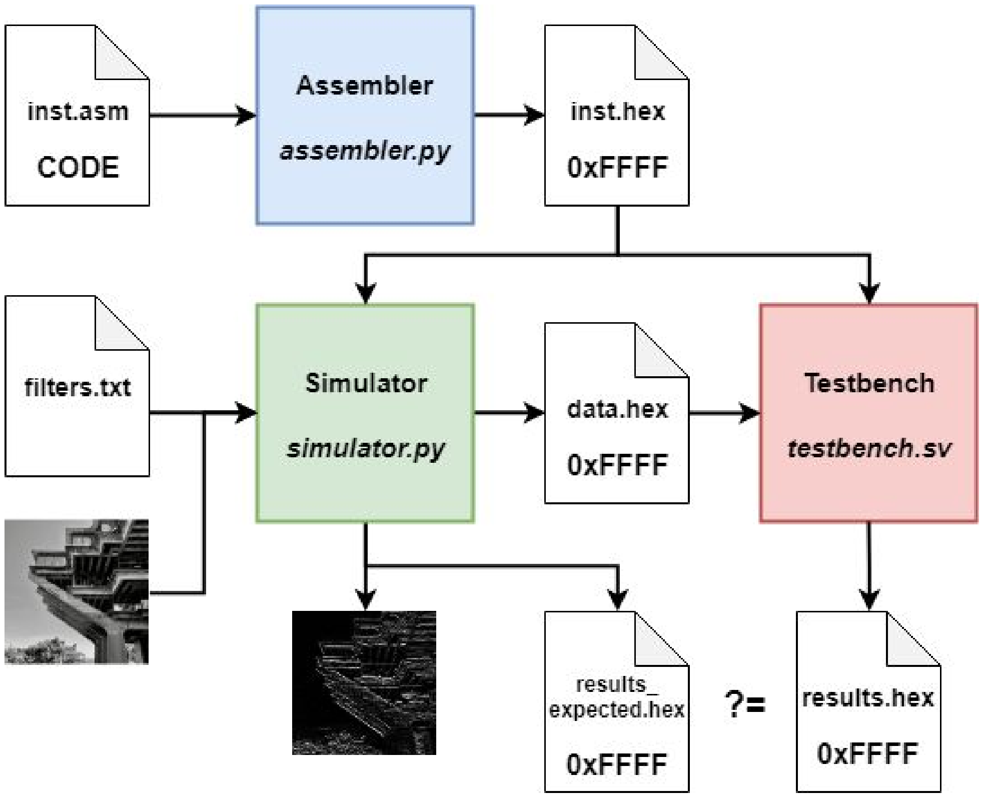
```
DC //executes convolution
```

Machine code:

[31:28]	[27:0]
4 bits	28 bits
Opcode	NOP

Simulation Framework

Overview:



The above diagram shows an overview of our simulation framework. The assembler performs the basic function of converting Assembly code to hex machine code that can be fed to our verilog testbench and simulated python model. In addition, it performs minor syntactic checks on the input file. The expected input and outputs are as follows:

Input:	inst.asm:	Assembly file formatted as defined by our ISA
Output:	inst.hex	Hex machine code

The Python model simulator performs a convolution on an input image and filters and generates outputs that can be used to verify the hardware model. It has the following inputs and outputs:

Input:	filters.txt	Contains filter data stored in a matrix format
	inst.hex	Hex machine code
	IMAGE	Input Image as png, jpeg, etc
Output:	data.hex	Data memory file containing filter and image data
	results_expected.hex	Expected convolution result
	IMAGE	Expected output images

The testbench reads and feeds the input and outputs generated by the assembler and simulator into our SystemVerilog hardware model and verifies that our hardware implementation correctly follows the Python model. It contains the following inputs and outputs:

Input:	inst.hex	Hex machine code
	data.hex	Data memory file containing filter and image data
Output:	results.hex	Convolution result to be compared against Python model

Example Simulation Flow:

IMPORTANT: For detailed usage instructions for *assembler.py* and *simulator.py*, please refer to *README.txt* under the *model* folder. This file details custom arguments and inputs when executing *assembler.py* and *simulator.py*.

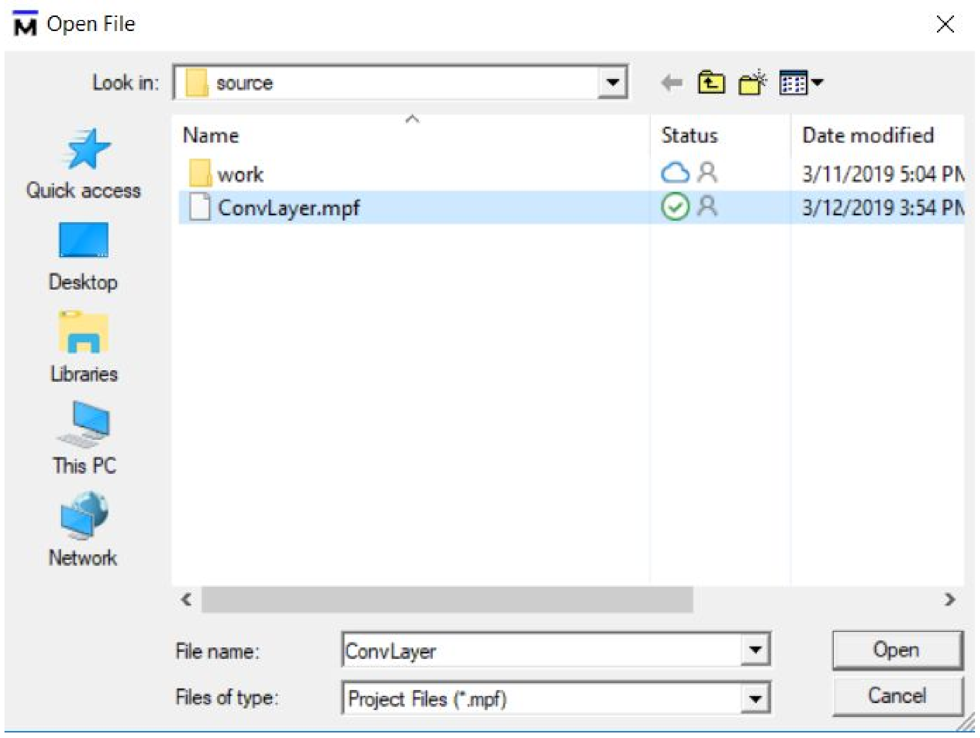
We provide an example simulation flow that can be executed using the following steps. The Python model has some non-standard Python library dependencies such as *imageio* that may require installation. Additionally, a SystemVerilog simulator is necessary such ModelSim or VCS to simulate the testbench; we use *ModelSim PE Student Edition* from Mentor Graphics, which is free for students.

1) Run *example.bat* under the *model* folder. This runs *assembler.py* and *simulator.py* with *./images/example.jpg, ./filters/example.txt,* and *./examples.asm* as inputs. This will automatically generate *./hex/data.hex* and *./hex/inst.hex* as inputs to our convolution layer. Additionally, this also generates *./hex/results_expected.hex* which the testbench will use to verify that our convolution layer is performing convolution as expected. Assuming no library dependencies are missing, you should see the follow console output:

```
> .\example.bat
> python ./assembler.py example.asm
['LF', '3', '2', '1']
12100001
['LS', '512', '512']
201FF1FF
['LI', '5']
30000005
['DC']
40000000
> python ./simulator.py example.jpg example.txt
Kernel Size : 3
Kernel Num  : 2
Kernel Addr : 1
Image Height: 512
Image Width : 512
Image Addr  : 5
Do convolution
Wait...

> PAUSE
Press any key to continue . . .
```

2) Run ModelSim, navigate to *source* folder and load *ConvLayer.mpf*. You may need to select *Project File (*.mpf)* as the file type:



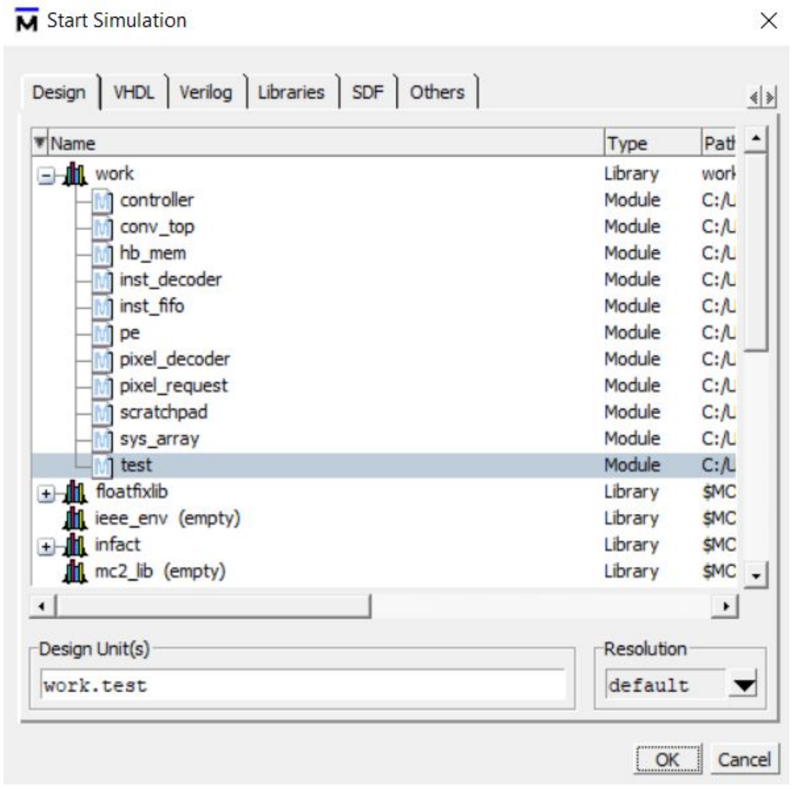
Name	Date modified
work	3/11/2019 5:04 PN
ConvLayer.mpf	3/12/2019 3:54 PN

File name: ConvLayer

Files of type: Project Files (*.mpf)

3) Compile design with *Compile* > *Compile All* if not already precompiled (all files should have a check mark next to them if they are compiled).

4) Load Simulation with *Simulation* > *Start Simulation*. Select *test* under *work* to load the testbench:



Start Simulation

Design | VHDL | Verilog | Libraries | SDF | Others

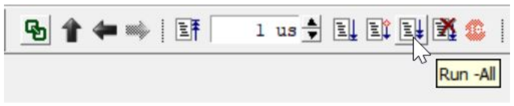
Name	Type	Path
work	Library	work
controller	Module	C:/L
conv_top	Module	C:/L
hb_mem	Module	C:/L
inst_decoder	Module	C:/L
inst_fifo	Module	C:/L
pe	Module	C:/L
pixel_decoder	Module	C:/L
pixel_request	Module	C:/L
scratchpad	Module	C:/L
sys_array	Module	C:/L
test	Module	C:/L
floatfixlib	Library	$MC
ieee_env (empty)	Library	$MC
infact	Library	$MC
mc2_lib (empty)	Library	$MC

Design Unit(s): work.test

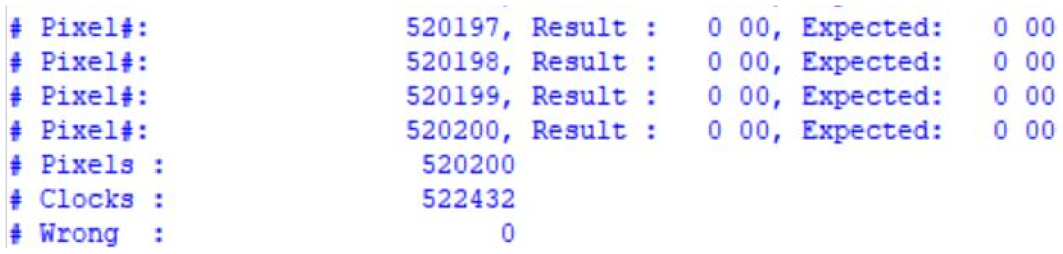
Resolution: default

OK Cancel

5) Add any waveforms you wish to observe (or none if verifying correctness only), and run the simulation with the Run -All button:



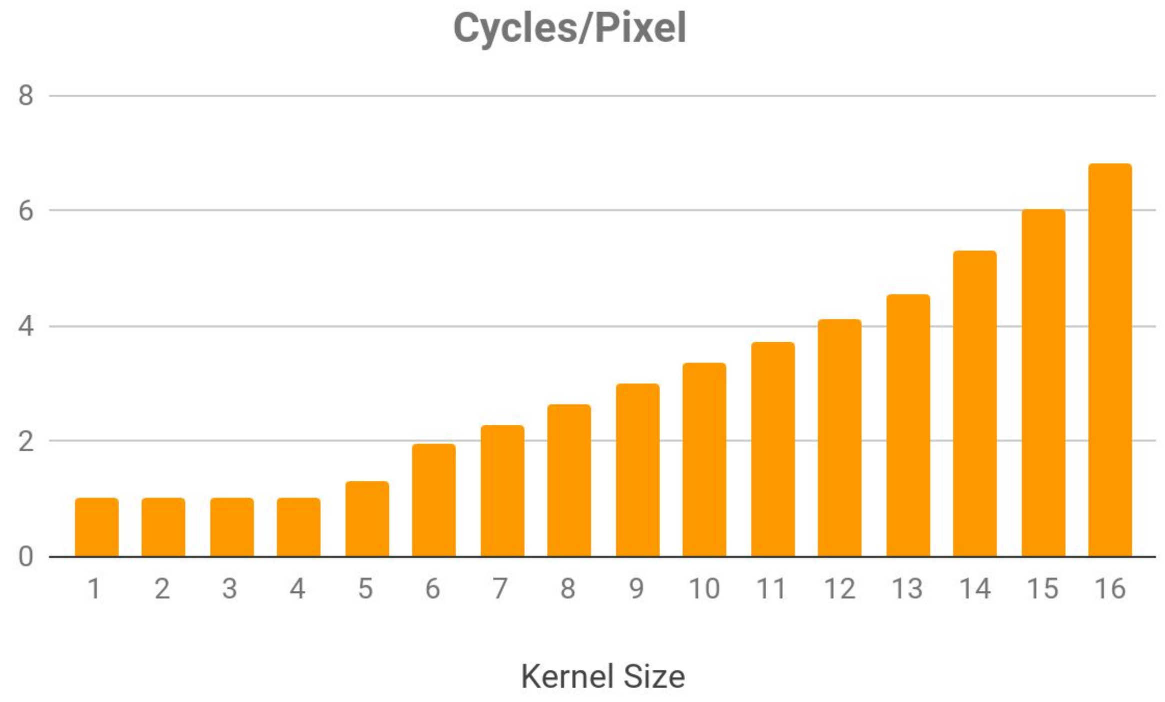
6) Simulation will run till completion if there are no differences between the expected result from the Python model and the generated result from the hardware simulation. The final generated transcript should indicate how many output pixels were processed, how many clock cycles were required, and whether any pixels do not match the Python model:



```
# Pixel#:            520197, Result :  0 00, Expected:  0 00
# Pixel#:            520198, Result :  0 00, Expected:  0 00
# Pixel#:            520199, Result :  0 00, Expected:  0 00
# Pixel#:            520200, Result :  0 00, Expected:  0 00
# Pixels :            520200
# Clocks :            522432
# Wrong  :                 0
```

Performance

Because of our hardware implementation of prefetching is limited by the amount of available empty/invalid request resources that we can override with prediction requests, prefetching performance gets progressively worse as there are less available request resources allocated for prefetching; as kernel size increases, more pixels are being requested simultaneously from memory, decreasing the amount of empty request slots available for prefetching and thus degrading performance.



We observe that with the default memory access time (10 total cycles), prefetching allows our design to produce almos 1 pixel per cycle with kernel sizes of 4x4 or smaller. Cycles/pixel becomes linearly progressively worse from kernel sizes of 5x5 to 13x13, and then become exponentially worse from 14x14 to 16x16; this behavior is expected because at a kernel size of 16x16, no prefetching is possible at all and the systolic array must stall and wait for a memory access whenever a new address is requested.

Please refer to *Appendix* for raw data.

Appendix

Below is the raw data prefetching performance of using the same image with different kernel sizes. Note that because a lack of padding support, larger kernels will produce less pixel output than smaller kernels; the cycles/pixel is still relevant despite this since it represents equilibrium throughput with a large image size.

Filter Size	**Pixels**	**Cycles**	**Cycles per pixel**
16x16	38306	260823	6.81
15x15	38988	235777	6.05
14x14	39675	210241	5.30
13x13	40368	184227	4.56
12x12	41067	168937	4.11
11x11	41771	156206	3.73
10x10	42482	143224	3.37
9x9	43200	129956	3.01
8x8	43923	116516	2.65
7x7	44652	102826	2.30
6x6	45387	88892	1.96
5x5	46128	59950	1.30
4x4	46875	48400	1.03
3x3	47628	48690	1.02
2x2	48387	49056	1.01
1x1	49152	49410	1.01