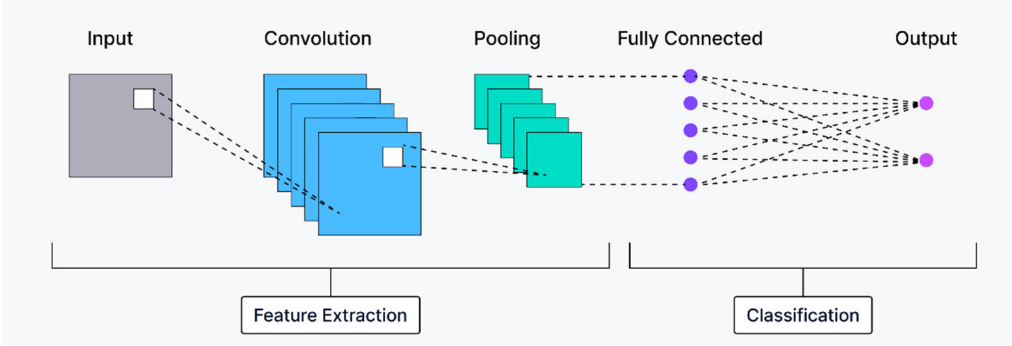
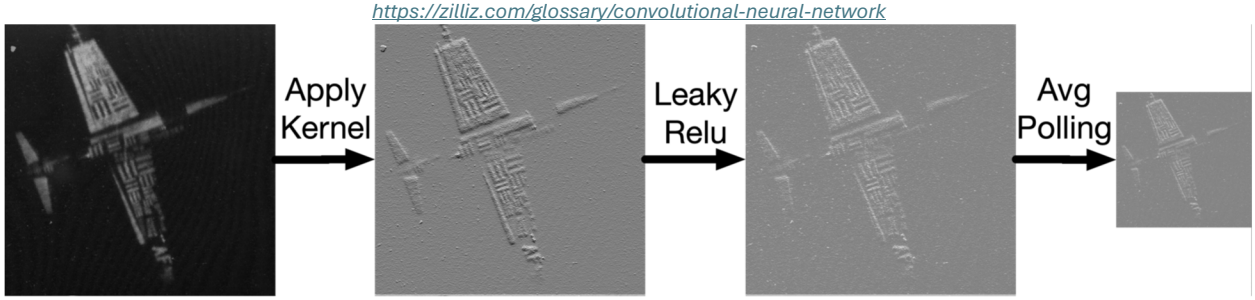
ECE 464/564 Fall 2025 Project (V1.0):

Convolutional Neural Network Pipeline on a 1024x1024 Input with a 4x4 Kernel

The major steps in a CNN for classification are given below. We are implementing the first 3 steps.



https://zilliz.com/glossary/convolutional-neural-network



Objective:

Implement a simplified CNN pipeline that performs the following operations:

1. **Convolution** with a 4×4 kernel
2. **Leaky ReLU Activation**
3. **2×2 Average Pooling**

Step 1: Convolution with a 4×4 Kernel

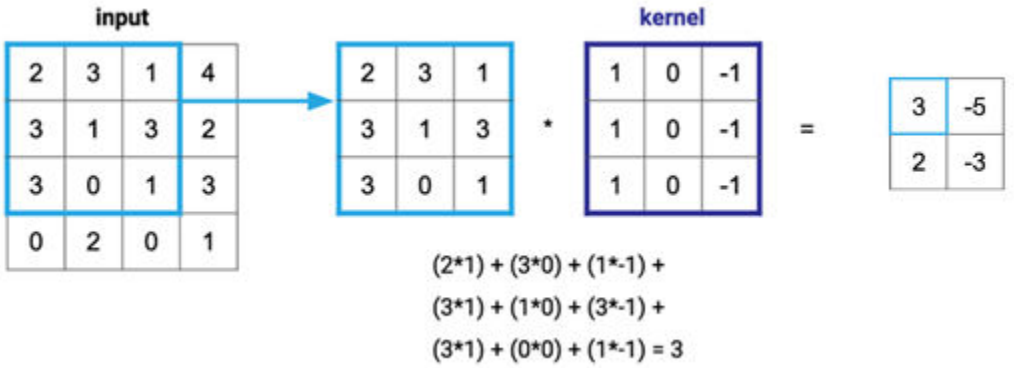
Operation: Slide the kernel across the input. At each position, compute:

$$Y(i,j) = \sum_{m=0}^{3} \sum_{n=0}^{3} X(i+m, j+n) \cdot K(m,n)$$

For this project, we will use the 4x4 kernel: i.e. The kernel is hard-wired into the logic.

$$K = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \end{bmatrix}$$

This is an illustration of how convolution works with a 3x3 kernel:



Adapted from: 4

And a 4x4 example:

Example convolution:

$$\begin{bmatrix} -3 & 2 & 1 & 1 & -3 & 2 & 1 & -1 \\ 1 & 2 & 1 & 1 & -1 & -3 & 0 & 1 \\ 2 & -2 & 0 & 1 & -2 & -1 & -2 & 2 \\ 2 & -2 & 2 & 0 & -2 & -2 & -3 & -3 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 1 & -2 & -3 & -1 & 1 & 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} -2 & -1 & 8 & 5 & -5 & -4 \\ 2 & -4 & 8 & 8 & 0 & -6 \\ -2 & -8 & 6 & 6 & 2 & 0 \\ -3 & -8 & 2 & 2 & 0 & 4 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 8 & -1 & -2 & -2 & -9 & -2 \end{bmatrix}$$

$$-3 * 1 + 2 * 0 + 1 * -1 + 1 * 1 + 1 * 0 + 2 * 0 + 1 * -1 + 1 * 0 + 2 * 1 + -2 * 0 + 0 * -1 + 1 * 0 + 2 * 1 + -2 * 0 + 2 * -1 + 0 * 0 = -2$$

Notes:

- The input is 1024x1024 matrix stored in row major order in the source DRAM. After the convolution, you are left with a 1021x1021 matrix.
- The convolution slides vertically and horizontally.
- The inputs are 8-bit signed variables. For the MAC unit we suggest using 16 bits (though not all of these are needed). You then clamp (saturate) the values for signed 8-bits, i.e. -128 to +127.

-

Step 2: Leaky ReLU Activation

ReLU (Rectified Linear Unit):

$$\text{LeakyReLU}(x) = \begin{cases} x, & x > 0, \\ 0, & -3 \le x \le 0, \\ \left\lfloor \frac{x}{4} \right\rfloor, & x \le -4. \end{cases}$$

- If convolution output is **positive**: keep it.
- If **negative**: Divide value by 4. We will implement this with an arithmetic right-shift. Any final value between -1 and 0 will be held at 0.
- The inputs are signed variables as above (8-bits)
- The outputs are unsigned variables of the same size (8-bits)

Continuing from Convolution Output:

$$\boldsymbol{ReLU}\begin{bmatrix} -2 & -1 & 8 & 5 & -5 & -4 \\ 2 & -4 & 8 & 8 & 0 & -6 \\ -2 & -8 & 6 & 6 & 2 & 0 \\ -3 & -8 & 2 & 2 & 0 & 4 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 8 & -1 & -2 & -2 & -9 & -2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 8 & 5 & -1 & -1 \\ 2 & -1 & 8 & 8 & 0 & -1 \\ 0 & -2 & 6 & 6 & 2 & 0 \\ 0 & -2 & 2 & 2 & 0 & 4 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 8 & 0 & 0 & 0 & -2 & 0 \end{bmatrix}$$

Step 3: 2x2 Average Pooling

The output of ReLU creates a matrix with an **odd** number of rows and columns (1021x1021). For average pooling with a stride of 2, the last row and column computed will need to be 0-padded to create effective dimensions of 1022x1022.

Average Pooling is a technique of **downsampling** the features extracted from the convolution and ReLU layers. With 2x2 max pooling, we operate on 2x2 blocks from the ReLU output and output the **average/mean** value from each block. With a stride of (2, 2) we do not overlap blocks.

Given a 2x2 matrix block: $\begin{bmatrix} 22 & 0 \\ 13 & 18 \end{bmatrix}$

Average pooling implements the function: $\text{avg}(22, 0, 13, 18) = 13.25$

We will be strictly handling 8-bit integer values in our system, so the decimal will be sheared. We suggest using 16 bits in the computation (though not all bits are needed) and then implement the /4 using a shift to get back to 8-bit unsigned integer.

This single block would then become a single output pixel with the value: 13

Continuing from ReLU Output Example:

$$\begin{bmatrix} 0 & 0 & 8 & 5 & 0 & 0 \\ 2 & 0 & 8 & 8 & 0 & 0 \\ 0 & 0 & 6 & 6 & 2 & 0 \\ 0 & 0 & 2 & 2 & 0 & 4 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 6 & 0 & 0 & 1 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{Avg\ Pool} \begin{bmatrix} 0 & 7 & 0 \\ 0 & 4 & 1 \\ \vdots & \ddots & \vdots \\ 3 & 0 & 0 \end{bmatrix}$$

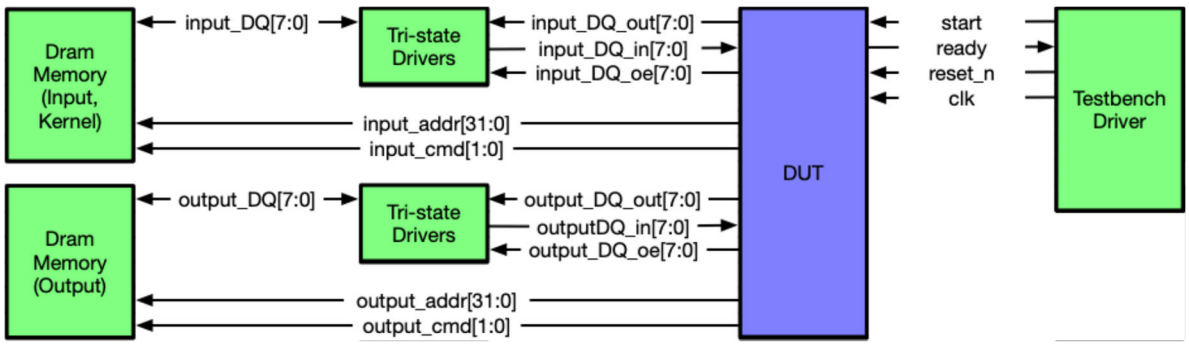
ECE464 and ECE564 EOL Project: Reduced project will require completion of only the convolutional layer. You then write the 1021x1021 8-bit values to the memory in row major order. Write this into the output DRAM starting at address 0. 0 pad the last write on the bottom row.

ECE 564 Project: Write out the 511 x 511 (x8bit) matrix in row major order. Write this into the output DRAM starting at address 0. 0 pad the last write on the bottom row.

Project Specifications:

Input data and kernel will be given in pre-loaded SDRAM memory modules. Implemented design will read these values from memory, complete the necessary computations, and write the outputs to memory.

Please refer to HW5 documentation for specific timing details of read/write operations regarding DRAM memory modules.



Test fixture (testbench) to DUT interface:

System signals

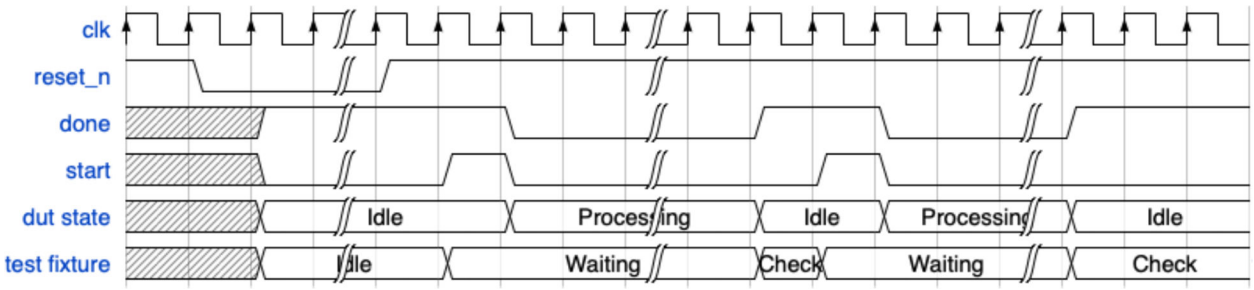
- **reset_n** — Active-low reset.
- **clk** — Clock for all sequential logic;

Control signals (handshake)

- **start** — Asserted by the test fixture when test starts.

Progress / completion

- **done** — Asserted whenever start is received. De-asserts only when final output is written to memory.
    - The test fixture treats the **falling edge** of done as the end-of-test indication.



DUT to SDRAM interface:

SDR bus signals (relative to the DUT)

- **CMD** — Command to the DRAM interface. Encodings: IDLE = 0x0, READ = 0x1, WRITE = 0x2. Must be IDLE when no transfer is requested.
- **Address** — Address for the current READ/WRITE; must be valid/stable whenever CMD is READ or WRITE.

DQ (tri-state, modeled with separate DUT signals)

- **DQ_oe** — Output-enable from the DUT. 1 → DUT is writing; 0 → DUT is reading (releases bus so DRAM drives it).
- **DQ_din** — Data from the DUT to the bus; used only when DQ_oe = 1 (WRITE beats).
- **DQ_dout** — Data from the bus to the DUT; sampled only when DQ_oe = 0 (READ beats).

- Reset/turnaround notes — Deassert DQ_oe except during active write beats. After the final write beat, deassert DQ_oe before any subsequent read to avoid contention.

DRAM contents:

DRAM stores each vector as a single 64-bit (8-byte) contiguous word (eight 8-bit signed elements, little-endian as described above). The first 16 entries will contain the 4x4 Kernel weights, and the proceeding addresses will contain the Input matrix, in row-major order.

Dram Address	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00
0x00000000	$K_{1,3}$	$K_{1,2}$	$K_{1,1}$	$K_{1,0}$	$K_{0,3}$	$K_{0,2}$	$K_{0,1}$	$K_{0,0}$
0x00000008	$K_{3,3}$	$K_{3,2}$	$K_{3,1}$	$K_{3,0}$	$K_{2,3}$	$K_{2,2}$	$K_{2,1}$	$K_{2,0}$
0x00000010	$I_{0,7}$	$I_{0,6}$	$I_{0,5}$	$I_{0,4}$	$I_{3,0}$	$I_{2,0}$	$I_{0,1}$	$I_{0,0}$
0x00000018	$I_{0,15}$	$I_{0,14}$	$I_{0,13}$	$I_{0,12}$	$I_{0,11}$	$I_{0,10}$	$I_{0,9}$	$I_{0,8}$
…	…							
0x00100008	$I_{1023,1023}$	$I_{1023,1022}$	$I_{1023,1021}$	$I_{1023,1020}$	$I_{1023,1019}$	$I_{1023,1018}$	$I_{1023,1017}$	$I_{1023,1016}$

Table 1: DRAM Memory Layout

The system is little-endian, so the least-significant byte (LSB) is at the lowest address of the vector and the most-significant byte (MSB) at the highest: $memory[base + 0] = v_0, memory[base + 7] = v_7$. If the vector's base address ends with 32'hXXXXXXX8, the eight bytes occupy 32'hXXXXXXX8 through 32'hXXXXXXXF. For example, bytes $[v_7..v_0] = [12,34,56,78,9A,BC,DE,F0]_{16}$ represent the 64-bit word 0x123456789ABCDEF0 and are stored at increasing addresses as F0 DE BC 9A 78 56 34 12.

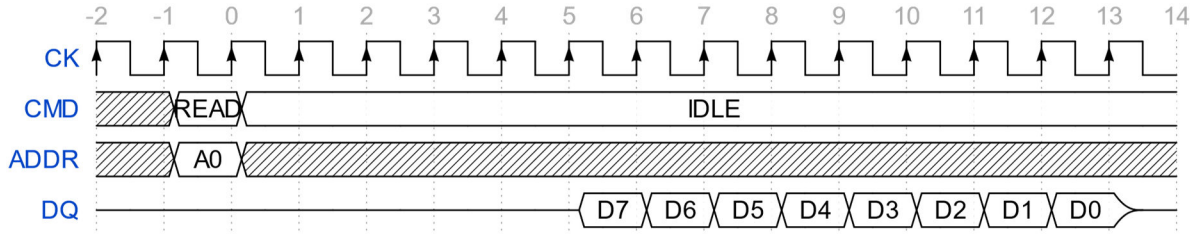
Output Memory Layout

Dram Address	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00
0x00000010	$O_{0,7}$	$O_{0,6}$	$O_{0,5}$	$O_{0,4}$	$O_{0,3}$	$O_{0,2}$	$O_{0,1}$	$O_{0,0}$
0x00000018	$O_{0,15}$	$O_{0,14}$	$O_{0,13}$	$O_{0,12}$	$O_{0,11}$	$O_{0,10}$	$O_{0,9}$	$O_{0,8}$
…	…							
0x000001f0	$O_{0,503}$	$O_{0,502}$	$O_{0,501}$	$O_{0,500}$	$O_{0,499}$	$O_{0,498}$	$O_{0,497}$	$O_{1,496}$
0x000001f8	$O_{0,510}$	$O_{0,509}$	$O_{0,509}$	$O_{0,508}$	$O_{0,507}$	$O_{0,506}$	$O_{0,505}$	$O_{1,0}$
0x00000200	$O_{1,8}$	$O_{1,7}$	$O_{1,6}$	$O_{1,5}$	$O_{1,4}$	$O_{1,3}$	$O_{1,2}$	$O_{1,1}$
…	…							
	$O_{510,509}$	$O_{510,508}$	$O_{510,507}$	$O_{510,506}$	$O_{510,505}$	$O_{510,504}$	$O_{510,503}$	$O_{510,502}$
0x0003fc00	$O_{510,510}$	…	…	…	…	…	…	…

DRAM Data Transaction

Data is transferred Most Significant Byte (MSB) first. Here's an example of a single memory transfer, showcasing the access of address A0 and the order of data D7-D0 that will appear on DQ.

Dram Address	**0x07**	**0x06**	**0x05**	**0x04**	**0x03**	**0x02**	**0x01**	**0x00**
…	**…**	**…**						
A0	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
…	**…**	**…**						



The following example demonstrates how a 4x4 kernel is mapped in memory and the relation between the data in the provided *.dat file.

Example 4x4 Kernel:

row/col	0	1	2	3
0	0x88	0x77	0x66	0x55
1	0x44	0x33	0x22	0x11
2	0x00	0xFF	0xEE	0xDD
3	0xCC	0xBB	0xAA	0x99

Memory File: *.dat

$$@0x00 \rightarrow 0x1122334455667788$$
$$@0x08 \rightarrow 0x99AABBCCDDEEFF00$$

Memory Map:

Dram Address	**0x07**	**0x06**	**0x05**	**0x04**	**0x03**	**0x02**	**0x01**	**0x00**
…	**…**	**…**						
A0	**0x11($K_{1,3}$)**	**0x22($K_{1,2}$)**	**0x33($K_{1,1}$)**	**0x44($K_{1,0}$)**	**0x55($K_{0,3}$)**	**0x66($K_{0,2}$)**	**0x77($K_{0,1}$)**	**0x88($K_{0,0}$)**
A1	**0x99($K_{3,3}$)**	**0xAA($K_{3,2}$)**	**0xBB($K_{3,1}$)**	**0xCC($K_{3,0}$)**	**0xDD($K_{2,3}$)**	**0xEE($K_{2,2}$)**	**0xFF($K_{2,1}$)**	**0x00($K_{2,0}$)**
…	**…**	**…**						

Final Project Guidelines

1. Please see the README.pdf for build and run instructions.
2. ./srcs/rtl/dut.sv is a stub for you to implement your RTL design.
3. Modify the setup to reflect the final clk that you synthesized to. Change line 21 and update the CLK_PER variable to the new clock.
4. Design should not have any major/minor synthesis errors pointed out in the Standard Class Tutorial (Appendix C). This includes but is not limited to latches, wired-OR, combination feedback, etc.
5. You are expected to provide a sketch of your design and the FSD down to the register, operator (e.g. "+"), and mux level.
6. If you use AI you are expected to provide your prompts.

Design, verify, synthesize a module that meets these specifications. **Use at least one coding feature unique to System Verilog.**

Submission Instructions:

- **Project Verilog and synthesis files.** Submitted electronically on the date indicated in the class schedule. Please turn in the following:
    - All Verilog files AS ONE FILE
    - There is one submit command, see README.pdf: the resulting command will produce a submission.unityID.tar.gz file: example submit.jdoe.tar.gz
    - Synopsys view_command.log file from complete synthesis run
- **Project Report.** Complete report to be turned in electronically with HW compressed file. It must follow the format attached. There is a 10% penalty for not following the format.
- **Logic Diagram.** You will be required to draw a detailed logic diagram of your implemented project design, down to the mux/flipflop/operator level. Must also be submitted with your project. This is a separate PDF to your project report.

Late Submissions:
Project will receive a 10% penalty per day late. Hard cutoff for late project submissions will be 1 week after the due date.

Due Date: November 12, 2025

Academic Integrity (AI) Policy:

This project is to be conducted individually. You can collaborate on the paper version of the design, including discussion of ideas, design approach, etc. However, you are forbidden to

share code or to reuse code of others. We will be running code comparison tools on your submitted code. **The usage of AI tools such as GPT is allowed but should be declared (delineated by comments) and the queries referenced and presented in your report.** Comments should indicate the start and end of any GPT derived code. The student is responsible for all that is submitted. No points will be awarded if the AI generated content was faulty.

Preliminary Report Grading Rubric

You are to submit a report reflecting your high level design. It should include most of the blocks that will make up a working design, including registers, multipliers, adders and other functional units along with a written description of operation. No controller is needed at this stage – datapath only. Full points will be given for a "reasonable" and "serious" attempt at capturing a datapath and major FSM steps that can execute this algorithm. The report should be neat but can be short.

Final Submission Grading Rubric

Requirement	**Standard**	**Points**
Simulates correctly	DUT produces correct results for first (public dataset) – 20 points DUT produces correct results for second (hidden) dataset – 10 points	30
Synthesizes correctly	Lose 5 points for each synthesis error including timing violations on synthesis with reported clock	30
Presents an accurate logic diagram and FSD	Accurate to register, mux, operator, truth table and FSD states. Lose 5 points for each missing element.	15
Pipelines design	Completes all computation in read time * 1.25, i.e. 1024*1024*1.25 cycles	15
Performance/Area	**Only for projects with 90/90 for above.** Report cell area, clock period, and #of cycles to complete. Will compete for performance/area.	10