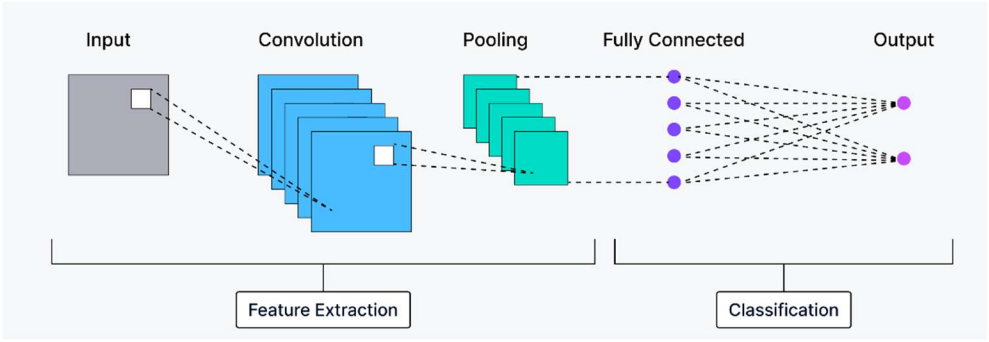
ECE 464/564 Fall 2025 Project (V3):

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



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

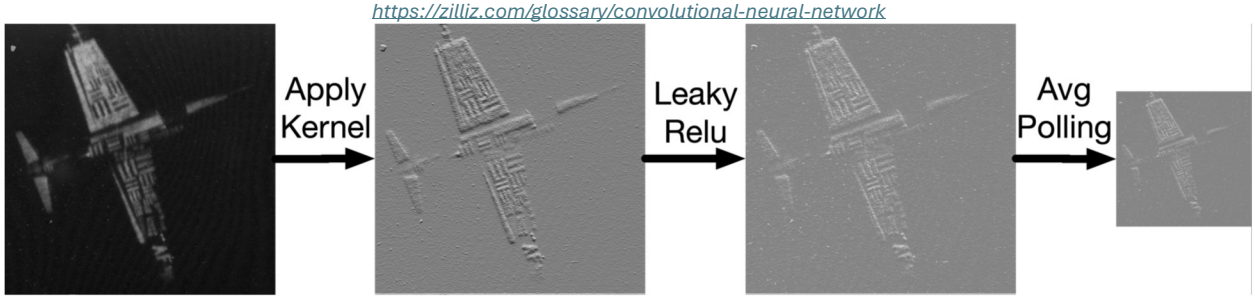


Figure 1: Visualization of Convolutional Processing on Input Image

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:

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

Example convolution:

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

$$= \begin{bmatrix} -2 & -3 & 12 & 7 & -4 \\ -2 & -7 & 8 & 10 & 1 \\ -1 & -11 & 4 & 4 & 0 \\ 0 & -6 & -2 & 2 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 8 & 4 & -1 & -3 & -14 \end{bmatrix}$$

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

Our input will be a 1024x1024 matrix. After the convolution we will have 1021x1021 matrix. The convolution output can use 20-bit values to deal with overflow, and we will truncate (clamp) the value back to an 8-bit signed value at the end of the project. *The 464 projects will truncate to signed 8-bit here for output to memory.*

Step 2: Leaky ReLU Activation

ReLU (Rectified Linear Unit):

$$LeakyReLU(x) = \begin{cases} x, & x > 0, \\ 0, & -4 < x \leq 0, \\ \left\lceil \frac{x}{4} \right\rceil, & x \leq -4 \end{cases}$$

- If convolution output is **positive**: keep it.
- If **value is between -4 and 0**, the ReLU function will **set this to 0.**
  - **Note that this behavior is different from an arithmetic shift**
- If **value is less than or equal to -4**: Divide value by 4. We will implement this with an arithmetic right-shift. Negative non-integer values are truncated (i.e. rounded up with ceiling function).

Continuing from Convolution Output:

$$ReLU \begin{bmatrix} -2 & -3 & 12 & 7 & -4 \\ -2 & -7 & 8 & 10 & 1 \\ -1 & -11 & 4 & 4 & 0 \\ 0 & -6 & -2 & 2 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 8 & 4 & -1 & -3 & -14 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 12 & 7 & -1 \\ 0 & -1 & 8 & 10 & 1 \\ 0 & -2 & 4 & 4 & 0 \\ 0 & -1 & 0 & 2 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 8 & 4 & 0 & 0 & -3 \end{bmatrix}$$

Step 3: 2x2 Average Pooling

The output of ReLU creates a matrix with an **odd** number of rows and columns (1024x1024 to 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. Figure 4 shows a visualization of this process for a smaller 32x32 example input.

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: $\text{avg}(22, 0, 13, 18) = 13.25$

With our system design of only signed integer values, calculating the mean takes on a slightly different behavior than dividing by 4:

$$Average(x) = \begin{cases} \left\lfloor \frac{x}{4} \right\rfloor, & x \geq 4, \\ 0, & -4 < x < 4, \\ \left\lceil \frac{x}{4} \right\rceil, & x \leq -4 \end{cases}$$

We will be strictly handling 8-bit integer values in our system, so the decimal will be sheared. Take notice that this shifting behavior results in 0 for values between -4 and 4, which deviates from the -1 that an arithmetic shift of negative numbers would result in.

This 2x2 block would then become a single output pixel with the value: 13

The output from average pooling will then be truncated to an 8-bit value for final output to the DRAM for the 564 project.

Average Pooling 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.

Project Specifications:

Input data and kernel will be given in pre-loaded DRAM 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.

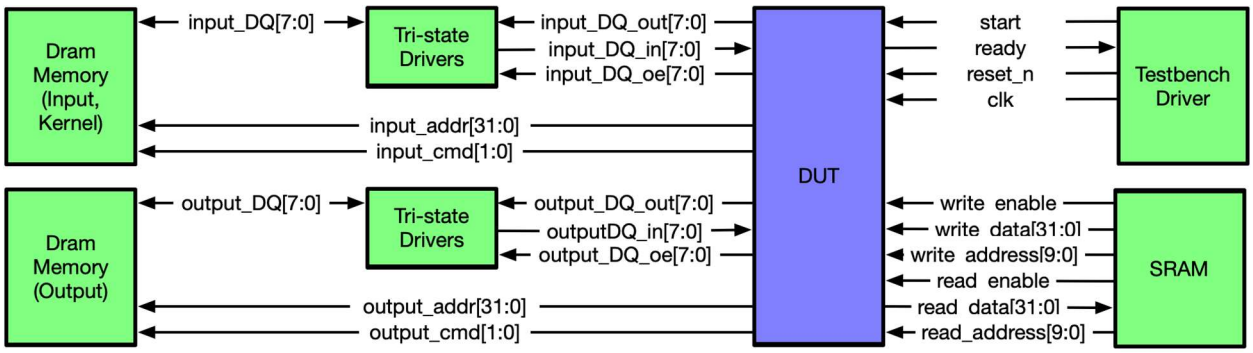


Figure 2: Test Fixture Block Diagram

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

- **ready** – Asserted when the design is idle. De-asserts upon detecting **start** and remains low during processing. **Re-asserts** only after the final output write is committed to memory.
  - The test fixture treats the synchronous **rising edge (low-to-high)** of ready as the end-of-test indication.

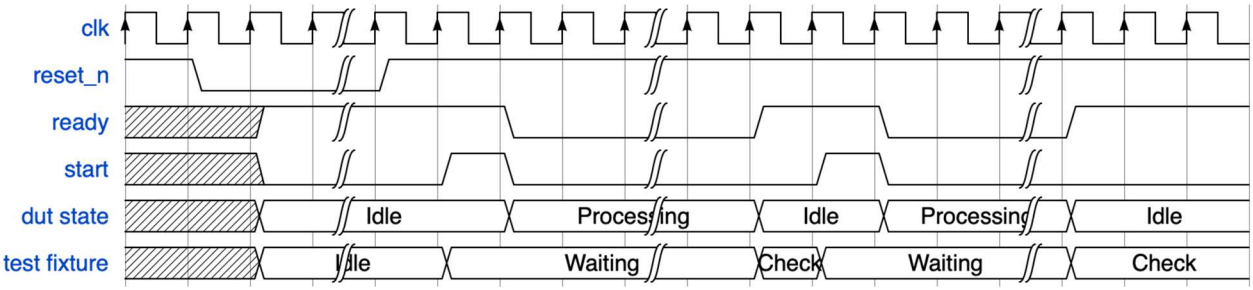


Figure 3: DUT to Testbench handshake Timing Diagram

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.

Final outputs will be written to the DRAM in row-major order. At the end of each row, (0x00) will be written to extend the input burst to a multiple of 8. This ensures the start of each row aligns with a modulo 8 address byte.

The following figure shows the 0 padding (blue) involved with the ReLU output, and with the final output for a 32x32 input size example. The same behavior can be extrapolated for the full-sized matrices used: 511 is "padded" to 512, and 1021 is "padded" to 1024, as shown in the output memory layouts (Figures 6 & 7).

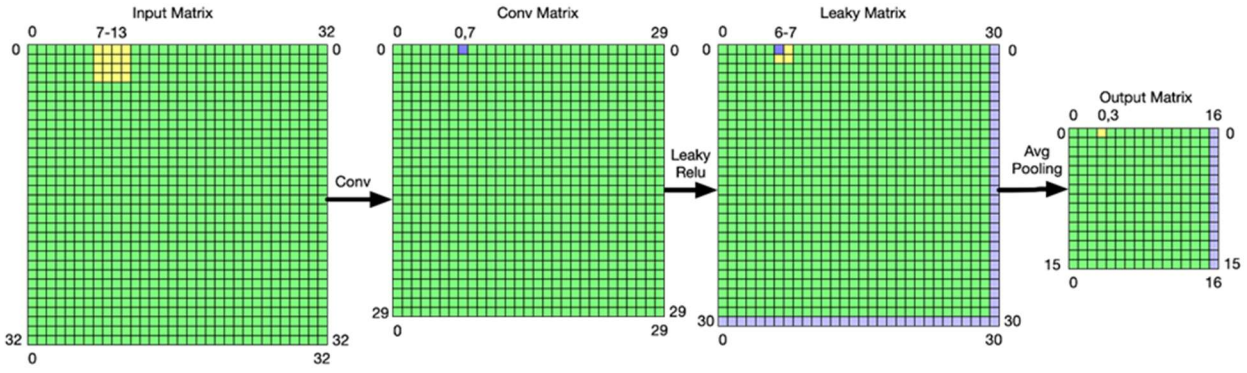


Figure 4: Example Matrix sizes/0 padding for 564 project.

For the 464 project, the same applies but only for the convolution output:

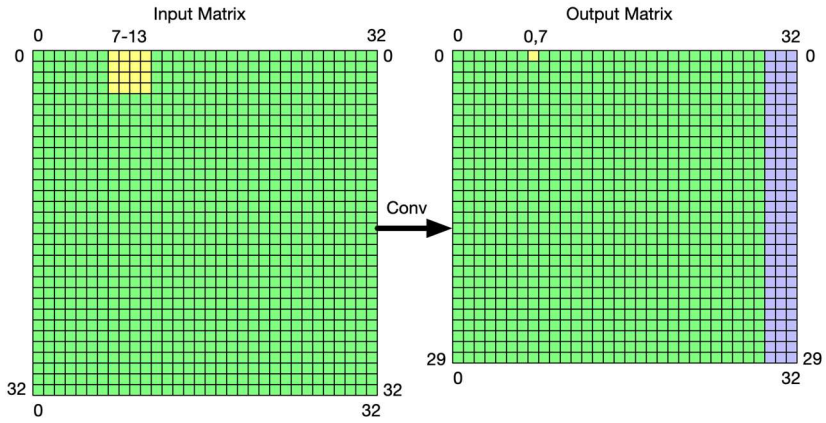


Figure 5: Example Matrix sizes/0 padding for 464 project

Output Memory: 564

Dram Address	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00
0x00000000	$O_{0,7}$	$O_{0,6}$	$O_{0,5}$	$O_{0,4}$	$O_{0,3}$	$O_{0,2}$	$O_{0,1}$	$O_{0,0}$
0x00000008	$O_{0,15}$	$O_{0,14}$	$O_{0,13}$	$O_{0,12}$	$O_{0,11}$	$O_{0,10}$	$O_{0,9}$	$O_{0,8}$
…	…							
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	(0x00)	$O_{0,510}$	$O_{0,509}$	$O_{0,508}$	$O_{0,507}$	$O_{0,506}$	$O_{0,505}$	$O_{0,504}$
0x00000200	$O_{1,7}$	$O_{1,6}$	$O_{1,5}$	$O_{1,4}$	$O_{1,3}$	$O_{1,2}$	$O_{1,1}$	$O_{1,0}$
…	…							
0x0003fdf8	(0x00)	$O_{510,510}$	$O_{510,509}$	$O_{510,508}$	$O_{510,507}$	$O_{510,506}$	$O_{510,505}$	$O_{510,504}$

Figure 6: Expected Output DRAM Layout for 564 project

Output Memory: 464

Dram Address	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00
0x00000000	$O_{0,7}$	$O_{0,6}$	$O_{0,5}$	$O_{0,4}$	$O_{0,3}$	$O_{0,2}$	$O_{0,1}$	$O_{0,0}$
0x00000008	$O_{0,15}$	$O_{0,14}$	$O_{0,13}$	$O_{0,12}$	$O_{0,11}$	$O_{0,10}$	$O_{0,9}$	$O_{0,8}$
…	…							
0x000003f0	$O_{0,1015}$	$O_{0,1014}$	$O_{0,1013}$	$O_{0,1012}$	$O_{0,1011}$	$O_{0,1010}$	$O_{0,1009}$	$O_{0,1008}$
0x000003f8	(0x00)	(0x00)	(0x00)	$O_{0,1020}$	$O_{0,1019}$	$O_{0,1018}$	$O_{0,1017}$	$O_{0,1016}$
0x00000400	$O_{1,7}$	$O_{1,6}$	$O_{1,5}$	$O_{1,4}$	$O_{1,3}$	$O_{1,2}$	$O_{1,1}$	$O_{1,0}$
…	…	…	…	…	…	…	…	…
0x000ff3f0	$O_{1020,1015}$	$O_{1020,1014}$	$O_{1020,1013}$	$O_{1020,1012}$	$O_{1020,1011}$	$O_{1020,1010}$	$O_{1020,1009}$	$O_{1020,1008}$
0x000ff3f8	(0x00)	(0x00)	(0x00)	$O_{1020,1020}$	$O_{1020,1019}$	$O_{1020,1018}$	$O_{1020,1017}$	$O_{1020,1016}$

Figure 7: Expected Output DRAM Layout for 464 project

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
…	…	…						

Figure 8: Example data access order as laid out in memory

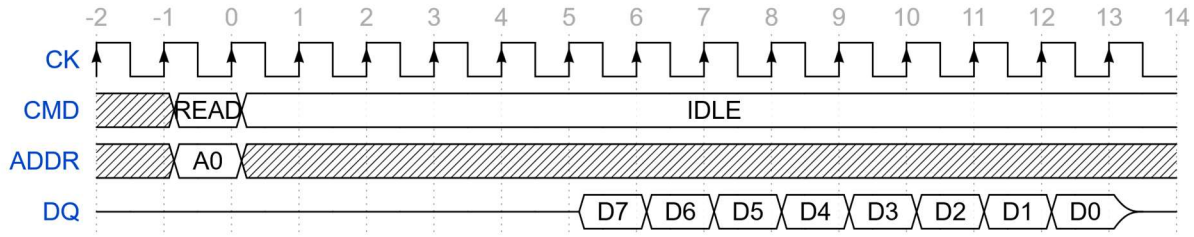


Figure 9: Memory Timing and little-endian behavior for a read burst access

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:

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($K_{2,0}$)
…	…	…						

Figure 10: Example Input Kernel mapping as laid out in DRAM

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

Figure 11: Example kernel laid out as original 4x4 array

SRAM:
An SRAM is provided with the project files in the */srcs/tb/* directory, to be used as an optional scratchpad memory. It is a dual-port memory, with separate read and write ports, and included for you in the testbench given, if you decide to use it in your design. The test fixture figure shows how this is instantiated by default in your testbench. Please refer to the SRAM documentation provided for more detailed instructions.

Those using the SRAM will be graded on a different performance/area curve.

Because the SRAM is instantiated by default, if you choose not to use the SRAM, you will get (OPT-1206) warnings like this:

> Information: The register 'write_address_reg[2]' is a constant and will be removed. (OPT-1206)

These specific ones will be OK to ignore.

Debug Inputs:
Provided in the project directory are a set of smaller "debug" inputs, sized 32x32, with their corresponding outputs, as shown in Figures 4 & 5. These are not part of the test suite, and included for your own convenience to test your designs on a smaller, more manageable scale first. Refer to the README for setting a "debug" preset target for simulation. Be sure to set this back to "run" target for the full test suite.

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. Make sure to change the values in the "*CMakePresets.json*" configuration to match your Unity ID, class (464/564), and best clock period.
    a. Double check that your submission tar runs successfully with the command sequence in the README using a fresh project directory
4. The final clock period you set for your submission will need to pass timing in a single shot, not one achievable only with incremental compiles.
5. 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, timing arcs, etc.
6. You are expected to provide a sketch of your design and the FSD down to the register, operator (e.g. "+"), and mux level.
7. 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 Submission.** 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
        - Make sure to change the values in the "CMakePresets.json" configuration to match your Unity ID, class (464/564), and clock period.
        - Double check that your submission tar runs successfully with the command sequence in the README using a fresh project directory
- **Project Report.** Complete report to be turned in electronically with compressed submission file. Included in the report should be performance metrics, high level project objectives, implementation strategies, and design explanations.
    - **Logic Diagram.** You will be required to draw a detailed logic diagram of your implemented project design, down to the mux/flip-flop level. Must be included with your report.
    - **Performance Characteristics.** Include clock period achieved, cycle count, cell/area report (cell_report_final), and setup and hold slack (max_slow_holdfixed, min_fast_holdcheck)
    - **AI Prompts and associated outputs used to implement the project should be documented in the appendix of your report.**
        - We may attempt to replicate your results following your documented processes.

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.

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/Project Plan 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. An outline of the controller is expected (but not required in 2025) – FSM and counter. 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	Grading Scheme	Grade
Simulates Correctly	DUT produces correct results for first (public dataset) – 20 points DUT produces correct results for second (hidden) dataset – 10 points	30
Synthesis	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	10
Pipelines design	Completes all computation in read time * 1.25 i.e. 1024*1024*1.25 cycles	10
Report Complete	Including all prompts if used AI. Report neat and well organized	10
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

Note, employers are often giving practical tests that AI cannot answer. If you over-rely on it you won't be able to do those questions. And can't get a job in this area.