EE310 - Hardware Description Languages
Spring 2025
Lab Assignment 3
Pipelined Fully Connected Layer Operations

1 Objective

This lab assignment aims to design and implement a hardware accelerator for the MNIST handwritten digit classification task using Verilog. Students will develop a digital circuit optimized for inference and deploy it on an FPGA, gaining hands-on experience in hardware design, optimization, and real-world acceleration of machine learning models.

2 Introduction

Deep Neural Networks (DNNs) are a class of machine learning models inspired by the structure and function of the human brain. They consist of multiple layers of interconnected neurons that process input data through weighted connections and activation functions. DNNs are widely used in various applications, including image recognition, natural language processing, and autonomous systems, due to their ability to learn complex patterns from large datasets.

The MNIST (Modified National Institute of Standards and Technology) dataset is a widely used benchmark for handwritten digit classification. It consists of 70,000 grayscale images of digits (0-9), each with a size of 28×28 pixels. MNIST is a fundamental dataset for testing and evaluating machine learning models, particularly in image processing and neural network applications.

In this lab, we aim to implement a hardware accelerator for MNIST digit classification using Verilog and deploy it on an FPGA. However, we will not implement a large-scale DNN due to resource constraints on FPGAs. Instead, we will focus on a simplified model, such as a small, fully connected network or a feature extraction-based approach, to demonstrate the principles of hardware acceleration while keeping the design feasible for FPGA implementation.

3 Task Definition

Like Lab 2, our accelerator will be responsible for successive Fully Connected Layer (FCL) computations. However, given that the target network might have wider layers than our accelerator's input or output dimensions, each transition must be pipelined. If a transition from a 4-neuron layer to a 6-neuron layer is computed with an accelerator of dimensions 2x2, the operation should take at least 6 cycles. You are encouraged to experiment with further pipelining the accelerator, such that the said operation is, for example, completed in 12 cycles. This would allow you to increase the maximum frequency with which your design is compatible. However, while pushing the synthesis tool to meet your requested frequency, your design might utilize more and more resources and take longer to synthesize/implement/generate the bitstream. **As a benchmark for this competition, a network of a size 100-80-40-10 will be used.** However, your circuit should also be functionally correct for different network sizes. At the beginning of your testing phase, please refrain from creating considerably large networks with the provided DNN generator Python script. Large networks may take a couple of hours to simulate.

As in the previous lab, weights, biases, and initial input tensors will be read from BRAM to their respective registers. Computed and partial outputs are not required to be written back to the BRAM; however, you are free to do so when you see fit. When all the layers are propagated through, you should end up with 10 values. These values are our network's predictions about the handwritten digit classification, which was supplied to the network as a flattened input. The index of the maximum of these 10 values is the ultimate prediction of the network. If the first one is the largest, the network has guessed the input was a '0'. You should display your result on the FPGA's LEDs or similar peripherals.

For the format of our numbers, IEEE754 half-precision will be used. The necessary theoretical background about the subject can be observed from the uploaded slides. Additionally, certain helper functions written with Python are provided to help you check your results. For multiplier and adder circuitry, you are free to either write your own, find one from online resources (provide a reference for the source), or use the following examples:

- Adder: GitHub Repository
- Multiplier: GitHub Repository

The adder has a small problem; this bug is left for you to discover and fix. You will be awarded +5 points if the occurrence scenario and a possible fix are explained in your reports. This bug only occurs under specific input combinations and does not critically affect the overall process, thus it is possible to correctly complete everything without fixing the said bug.

4 Design Specifications

The designed accelerator should be parametrized. The following snippet is a reference for the definition of the top module.

```verilog
module small_fc_top #(
    parameter LAYER_0_NEURON_COUNT = 36,
    parameter LAYER_1_NEURON_COUNT = 20,
    parameter LAYER_2_NEURON_COUNT = 20,
    parameter LAYER_3_NEURON_COUNT = 10,
    parameter ACCEL_IN_DIM = 4,
    parameter ACCEL_OUT_DIM = 10
    )(
    input clk,
    input rst,
    input btns_debounced,
    output reg [9:0] prediction // one-hot encoding for fpga leds
);
```

You can add/remove peripherals to/from the provided reference. However, your design must be parametrized according to the neuron counts of the layers. Here are some important remarks about the testing process:

- There will always be 4 layers, with the last one having the size of 10 neurons.
- While modules are being tested for functional correctness, there will not be any test cases that neuron count of a layer is not divisible by its respective accelerator dimension, in other words, if layer structure is 25-10-10-10, there will not be a case that tests the network with a 3x5 accelerator.
- The First layer will always have a squared number as its neuron count.
- The neuron count of Layer_(i+1) can be lower than the neuron count of Layer_i.
- A layer's neuron count can be lower than 10 unless it is the final layer. The last layer always has 10 neurons.
- You can create multiple accelerators and chain them.
- Although your design must work with every network structure and accelerator size that abides by the above rules, you should mention a specific accelerator dimension tailored for the benchmark network of 100-80-40-10 in your report.

After the non-pipelined version is done, implement the pipelined version. Figure 1 can be used as a reference. However, other design opportunities are also possible. After demultiplexing the accelerator output to the correct output neuron's accumulator, the accumulated values are summed with bias and passed through ReLU6. If an output neuron is not targeted by the demultiplexer, then 0 is pushed to that line, such that accumulated sum stays the same.

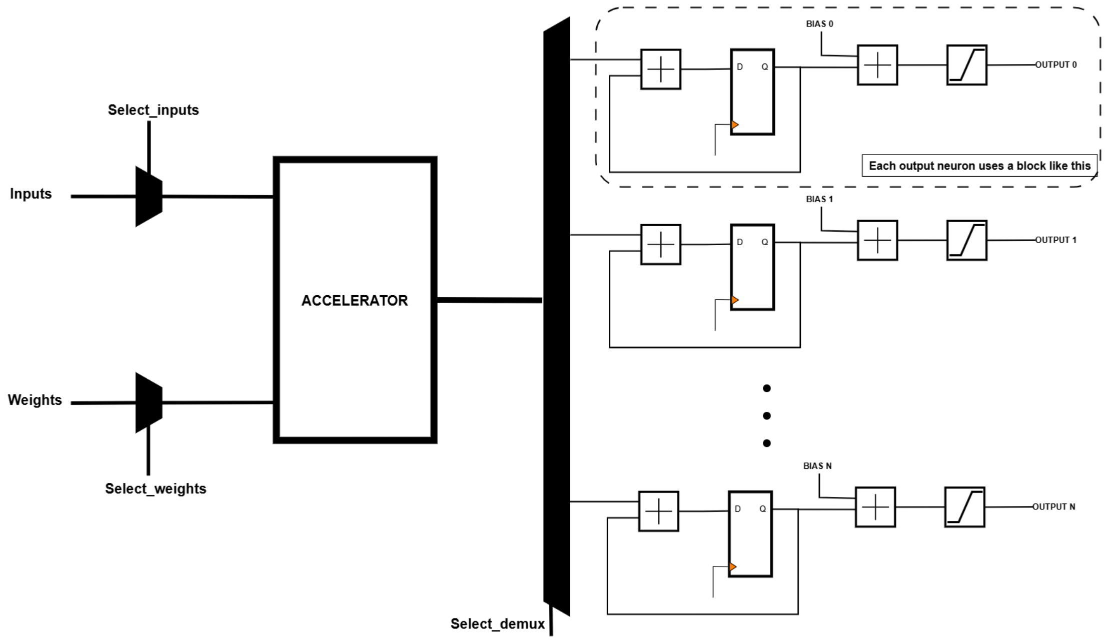


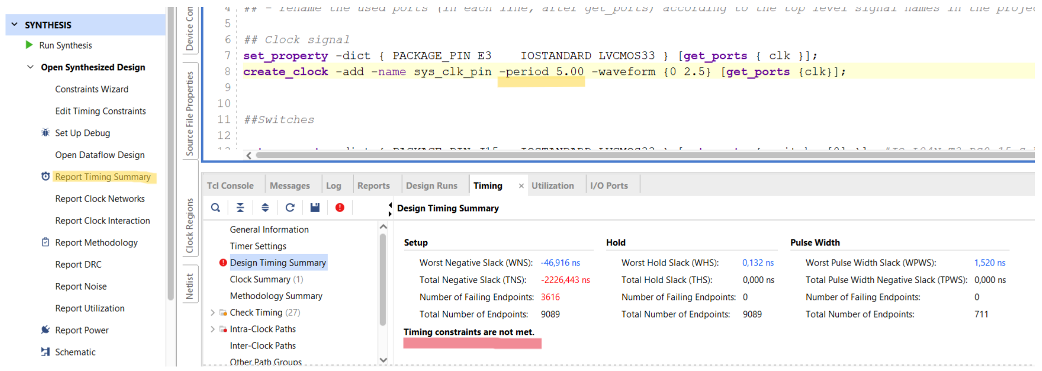
Figure 1: Reference Pipelined Accelerator

A testbench is also provided for you to test your design. However, this only targets your non-pipelined accelerator. Without the limitation of resource consumption, three accelerators are instantiated inside it. To use it,

1. Execute `pip install -r requirements.txt` console command on your cmd (This is only done once to install necessary dependencies; there is no need to do it every time).
2. Run the `mnist_lite_dnn.py` script. Input the required network sizes. When prompted which input class you would like to test, select a number between 0 and 9. The overall process should take a few minutes to complete on the CPU. There will be numerous files generated. You may use them to your liking.
3. Script will print Verilog-ready register values. Copy and paste them into the initial block inside the provided testbench. Arrange naming convention if necessary. Weights are provided in regular order and also in transpose order. You can use the one that is suitable for your implementation.

4. Adjust the neuron count values at the top of the testbench.
5. Simulate and observe the TCL Console output.
6. Copy the output for the last layer, or other layers, and paste it inside a string block as an argument to `parse_and_convert()` function found in `helper.py` file provided. An example is provided within the said Python file.
7. Observe the results and see if the index of the maximum value was indeed the class we selected in step 2. It may not be, since the network has a prediction accuracy < 1.0.

After achieving functional correctness with your pipelined accelerator, focus on optimizations for the benchmark case. Try to increase the clock frequency or your accelerator's size. However, your design must not violate timing constraints. After completing the synthesis, it is possible to check if the requested clock is problematic. The case in Figure 2 demonstrates **failed** timing constraints for a 200MHz clock.



```
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the projec

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name sys_clk_pin -period 5.00 -waveform {0 2.5} [get_ports {clk}];


##Switches
```

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-46,916 ns	Worst Hold Slack (WHS):	0,132 ns	Worst Pulse Width Slack (WPWS):	1,520 ns
Total Negative Slack (TNS):	-2226,443 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	3616	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	9089	Total Number of Endpoints:	9089	Total Number of Endpoints:	711

Timing constraints are not met.

Figure 2: 200 MHz clock violates critical path.

5 Example Workflow

1. Create and test IEEE754 Half-Precision Adder and Multiplier.
2. Interact with the BRAM and test your reading abilities with a basic FSM.
3. Using `generate` loops, instantiate input-weight multipliers and partial product adders. Test your circuit with small numbers and monitor every step. If the need arises, add the required signals to the waveform window and debug from end to start. After ensuring functionality, add a bias addition step alongside ReLU6 activation. (So far, it is quite similar to the second lab)
4. Try to run the provided `tb.v` with the explained Python scripts. If the results are incorrect, trace back to the source of the error with the printed operation flow details. You will be instantiating the accelerator directly; no need for BRAM or FSM on this step.
5. After the non-pipelined accelerator works correctly, devise a valid strategy to split the operation into cycles. This will create the base idea for the FSM.
6. Test any of your additional modules before putting everything together, such as the accumulator.
7. Integrate everything together without the BRAM and try to realize a small case. Check if everything works as intended. This step requires detailed analysis and is the most important step of the workflow.
8. Compare the outputs of your non-pipelined and pipelined accelerator in a testbench for a larger network. Ensure everything is accurate and as intended.
9. Try to maximize performance by pushing clock frequency to your circuit's limits. If possible, try to detect your critical path and pipeline it further. This task is quite complex and might take a long time to design.
10. Integrate BRAM and ensure correctness.
11. Program the FPGA to observe functional correctness. If you notice unusual behavior at this step, check Vivado's warnings for potential clues.

6 Deliverables

Students will work in groups of two, and only one submission per group is required. Late submissions will not be accepted.

6.1 Part 1 Deadline: Behavioral Simulation (30.03.2025)

Before submitting, please note the following expectations:

- Your design should pass the provided testbench and demonstrate successful operation across different DNN architectures, accommodating varying input sizes and neuron counts. (Exact output matching is not required due to bit representation differences, but the results should be similar.)

The required deliverables for Part 1 are:

- **Report:** A detailed explanation of your design, including:
  - The architecture and working principles of your hardware accelerator.
  - A description of the functional blocks and how they interact.
  - Justification of key design decisions.
- **Vivado Project (ZIP):** A compressed file containing all design files.

6.2 Part 2 Deadline: FPGA Implementation (09.04.2025)

Before submitting, please note the following expectations:

- Your design should generate a valid bitstream and run successfully on the FPGA.

The required deliverables for Part 2 are:

- **Updated Report:** An extension of the initial report, including:
  - Modifications and optimizations made for FPGA implementation.
  - Analysis of utilization and timing reports.
  - Performance improvements, with an explanation of latency and throughput optimizations.
- **Vivado Project (ZIP):** A compressed file containing the updated project files.

Note: Late submissions will not be accepted.

7 Grading Criteria

7.1 Part 1 (25% of Total Grade)

- Report: 12.5%
- Functional Correctness: 12.5%

7.2 Part 2 (75% of Total Grade)

- Report: 7.5%
- Functional Correctness: 17.5%
- FPGA Demo: 20%
- In-lab Explanation: 30%

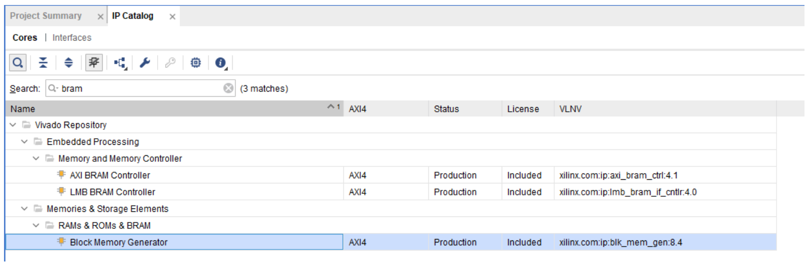
7.3 Bonus Points

- Exactly a 10% bonus will be added to the overall lab grade for the group whose implementation demonstrates the fastest performance, as measured by the product of the clock cycle count and the clock period, provided that the hardware produces correct results.

8 Generating BRAM with the IP Catalog

This section describes how to set up a Block RAM (BRAM) using Vivado's IP Catalog, configure its parameters, initialize its contents, and instantiate it into your design.

1. Locating the IP Catalog and the Block Memory Generator The IP Catalog is an integral part of the Vivado Design Suite and can be found in the Flow Navigator under the "Project Manager" section. Once you open your project in Vivado, click the **IP Catalog** in the left-hand pane. Here, you can browse through the available IP cores. Locate and select the **Block Memory Generator** IP, which is used to create BRAM instances for your design.



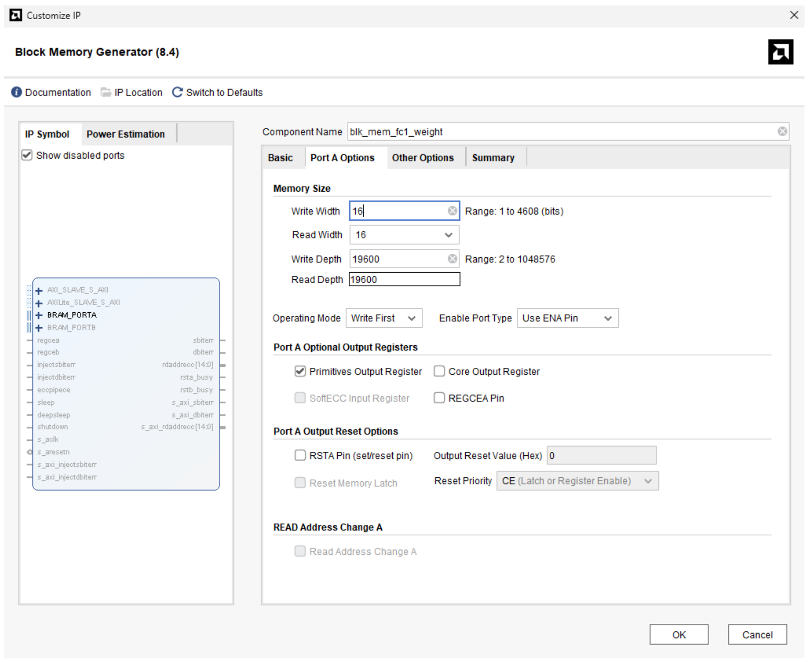
Search: bram (3 matches)

Name	AXI4	Status	License	VLNV
Vivado Repository				
Embedded Processing				
Memory and Memory Controller				
AXI BRAM Controller	AXI4	Production	Included	xilinx.com:ip:axi_bram_ctrl:4.1
LMB BRAM Controller	AXI4	Production	Included	xilinx.com:ip:lmb_bram_if_cntlr:4.0
Memories & Storage Elements				
RAMs & ROMs & BRAM				
Block Memory Generator	AXI4	Production	Included	xilinx.com:ip:blk_mem_gen:8.4

Figure 3: Navigating the IP Catalog to locate the Block Memory Generator.

2. Configuring the BRAM Parameters Double-click the **Block Memory Generator** to open its configuration window. Configure the memory parameters according to your design needs:

- **Write/Read Depth:** Set the depth based on the size of your memory array (i.e., the RAM size). This value determines the number of memory locations available.
- **Write/Read Width:** Set this to match the bit-width of a single data element.



[✓] Show disabled ports

Component Name: blk_mem_fc1_weight

Memory Size

Write Width: 16 — Range: 1 to 4608 (bits)
Read Width: 16
Write Depth: 19600 — Range: 2 to 1048576
Read Depth: 19600

Operating Mode: Write First — Enable Port Type: Use ENA Pin

Port A Optional Output Registers

[✓] Primitives Output Register — [ ] Core Output Register
[ ] SoftECC Input Register — [ ] REGCEA Pin

Port A Output Reset Options

[ ] RSTA Pin (set/reset pin) — Output Reset Value (Hex): 0
[ ] Reset Memory Latch — Reset Priority: CE (Latch or Register Enable)

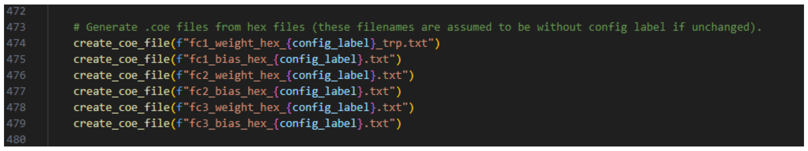
READ Address Change A

[ ] Read Address Change A

Figure 4: Setting the Write/Read Depth and Width based on your design requirements.

3. Initializing Memory Contents Enable the *Load Init File* option in the configuration window to load initial data into the BRAM. This option allows you to specify a `.coe` file for initializing the memory.

- The `.coe` file can be generated using the provided Python script. This script (see Figure 5) can be modified to output data in transposed or original format, depending on your design requirements.

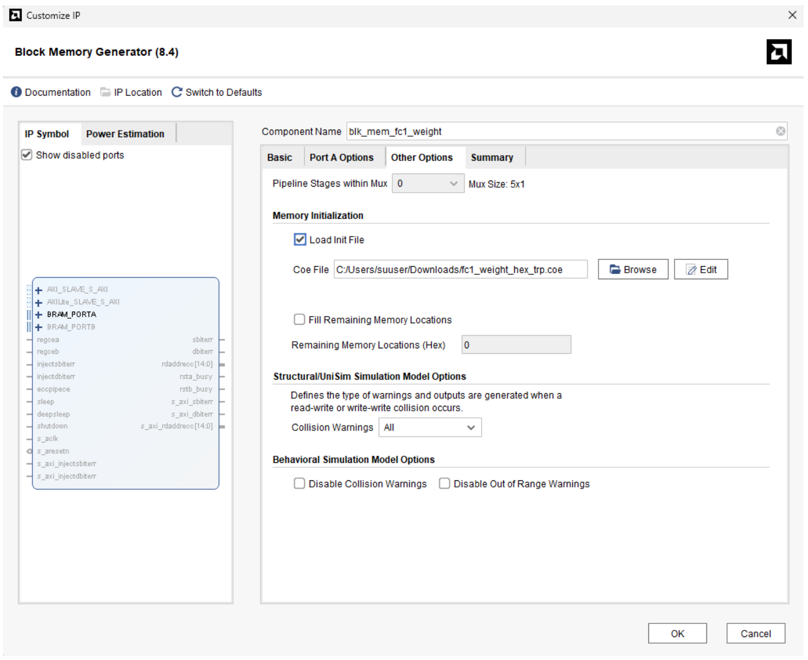


```python

    # Generate .coe files from hex files (these filenames are assumed to be without config label if unchanged).
    create_coe_file(f"fc1_weight_hex_{config_label}_trp.txt")
    create_coe_file(f"fc1_bias_hex_{config_label}.txt")
    create_coe_file(f"fc2_weight_hex_{config_label}.txt")
    create_coe_file(f"fc2_bias_hex_{config_label}.txt")
    create_coe_file(f"fc3_weight_hex_{config_label}.txt")
    create_coe_file(f"fc3_bias_hex_{config_label}.txt")

```

Figure 5: Python script used to generate the COE file. Modify this script to choose the desired data format.


Customize IP

Block Memory Generator (8.4)

Documentation | IP Location | Switch to Defaults

IP Symbol | Power Estimation

[✓] Show disabled ports

Component Name: blk_mem_fc1_weight

Basic | Port A Options | Other Options | Summary

Pipeline Stages within Mux: 0 — Mux Size: 5x1

Memory Initialization

[✓] Load Init File

Coe File: C:/Users/suuser/Downloads/fc1_weight_hex_trp.coe — Browse | Edit

[ ] Fill Remaining Memory Locations

Remaining Memory Locations (Hex): 0

Structural/UniSim Simulation Model Options

Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.

Collision Warnings: All

Behavioral Simulation Model Options

[ ] Disable Collision Warnings [ ] Disable Out of Range Warnings

OK | Cancel

Figure 6: Selecting and configuring the memory initialization using a COE file.

4. Generating and Instantiating the BRAM After configuring the BRAM parameters and setting up the initialization file, generate the IP. Once generated, the BRAM component can be instantiated in your design. For example, you can instantiate it in your Verilog code as follows:

```
blk_mem_fcl_weight mem1(
   .clka(clk),     // Clock signal
   .ena(ena),      // Enable signal
   .wea(wea),      // Write enable signal
   .addra(addra),  // Address input
   .dina(dina),    // Data input
   .douta(douta)   // Data output
);
```

Following these steps, you can successfully generate and configure a BRAM using Vivado's IP Catalog. The images provided are visual aids to help you navigate the process—from locating the IP Catalog to configuring memory parameters, initializing the memory contents, and integrating the BRAM into your design.