LOGIC DESIGN
FINAL PROJECT

18.4.24

PELEG SEGAL & NADAV HUGI

Main Goal:

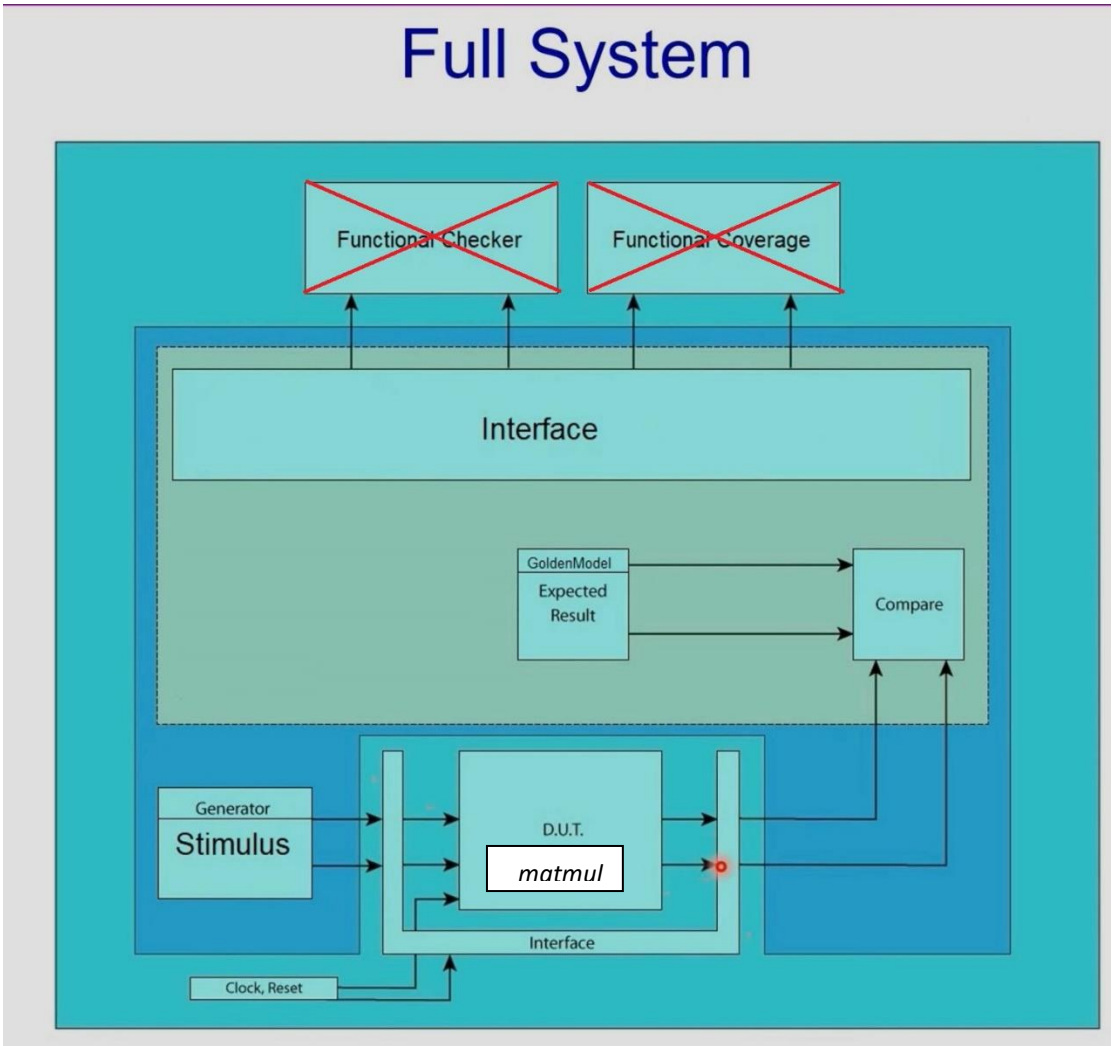
The following documentation file explains about our Logic Design Course project.

The main task is implementing a systolic array multiplication by using APB master-slave protocol.

1st part of the project – implementing the Verilog files that handle the task of systolic arrays multiplication.

It's important to note that our program supports general sizes of parameters to the choice of the user (BW,DW,ADDW etc).

2nd part: verification of the 1st part – the following diagram emphasizes the verification modules that we implemented using system Verilog:



VERIFICATION STRATEGY

Our verification strategy is designed to formally evaluate the matrix multiplication functions of our systolic array processor. Using a complex verification environment composed of several Verilog modules - including Stimulus and Golden Model - we aim to fully validate our design across various conditions.

1. Objectives of Verification Tests

The main goal of our testing approach is to validate the precise functionality of our systolic array processor in executing matrix multiplication. Our test bench emulates diverse data inputs and operational conditions to thoroughly assess the design. By automating the input of test vectors and the analysis of outputs, we strive to encompass all conceivable operational scenarios, ensuring the design's dependability and efficacy.

2. Overview of Test Bench Design and Architecture

Our verification framework revolves around the `tb_overall ` main unit, which integrates modules such `GoldModel`, and the Design Under Test (DUT) named `matmul`. Within the DUT, the `matmul` consists of sub-modules like `apbslave`, `mat_pad`. with the `apbslave` module hosting the Operand and scratchpad functionalities, and the `matmul_calc` incorporating a sequence of Processing Elements.

The `stimulus` module is key in our setup as it fetches data (random inputs) from text files which are generated by the 'generator', simulating an APB master that channels inputs into our `apbslave` under the `matmul.v` module. Outputs from the DUT are compared against expected results derived from our Python-based generator to verify the accuracy of computations (in the `GoldModel`) .

This architecture is selected to test our design's reaction to varied input scenarios and confirm that each component operates correctly within the whole system. Separating concerns into distinct modules increases the transparency and manageability of our testing environment.

Golden Model (GoldModel)

In our verification part of the project we use Golden Model which is a component of hardware design and testing. the Golden Model functions as a benchmark or reference point that we can compare our DUT to.

It's also known as the "ideal" model. This model plays a critical role in verifying and validating the design under test (DUT), which is the actual hardware design being evaluated. The primary function of the Golden Model is to compare the expected results, which we calculated beforehand by our generator (will be explained later on) with the actual outputs from the DUT.

The Golden Model's main task is to ensure that the output from the DUT matches the expected output. If we have mismatches - meaning the actual output differs from what was expected, the Golden Model "points" on these differences and provide detailed diagnostic information. This capability is crucial as it helps us to point on specific issues within the DUT, which helps the debugging process.

In our case, the golden model is set up to validate the results of a matrix multiplication

If no errors are found during a test, a message indicating success is displayed. And If mismatches are detected, the model logs the number of errors and sets an error flag.

After processing all tests, the model summarizes the results, providing a tally of how many tests passed versus the total number conducted.

Interface (interface_m)

In our project, due to the number of components we need to work with, managing the flow of information between them can be quite complex. This is where something like an interface in SystemVerilog can help us. An interface is basically a collection of wires or signals bundled together to make connections between different parts of the project more manageable. In our project the interface named interface_m can help us to understand how it connects between different modules and simplifies the design process.

The interface_m is defined in SV and includes a variety of signals that are crucial for the operation of digital circuits. This interface is particularly designed to handle interactions within a system that deals with data processing, in our case matrix operations.

Our interface receives clk and rst_ni (negative edge reset input) because the clk and rst signals are universals for the entire system.

Besides these signals, interface_m "packs" several specialized signals that help us controlling data movements. These include psel_i, penable_i, and pwrite_i, which are used in the APB protocol and use to enable the operation, and dictate whether the operation is a read or write. There's also pstrb_i, which stands for strobe signals that enable specific parts of the data bus, useful in operations where not all data bits are needed. The pwdata_i and prdata_o signals represent data being written to and read from a device.

In addition, interface defines address signals through paddr_i, allowing specific locations within a memory or peripheral to be accessed, which is crucial for reading from or writing to correct locations. The pready_o and pslverr_o are feedback signals indicating that the data is ready to be processed or if there's an error in the transaction, ensuring reliable data handling. Another signal, busy_o, informs the system whether a particular component is busy and cannot take new instructions. And as we said before, because we have components that share functionality they can use the same signals.

One of the important features of interface is its use of `modport`s, which specify different roles for the signals depending on the module they're interacting with. There are three modports in interface_m: DUT, STIMULUS, and GOLDEN. The DUT or modport mainly deals with inputs from the system and outputs for status and error flags, configuring how the device being tested receives and sends signals. The STIMULUS modport, on the other hand, is configured to output control signals and inputs status and data, effectively driving the tests by providing necessary signals to the DUT. Lastly, the GOLDEN modport focuses on checking the correctness of the outputs against expected results, essential for verifying that the DUT operates as intended.

Generator (python)

The generator is code we wrote in Python and it is a practical tool designed to create "artificial" data, in our case matrices, to help us in testing and verifying the performance of our design that's written in Verilog. This kind of testing is crucial and ensuring that it functions as intended.

Our Python script primarily focuses on generating matrices of various sizes and characteristics based on user inputs or randomly selected parameters. These matrices are then used to perform matrix multiplication. The results from these operations are vital as they are later compared against expected results compared by the Golden Model in System Verilog. The Golden Model as we explained serves as a benchmark or standard model that represents the correct behavior and outputs of the system under test.

When you run the script, it first asks the user how many sets of data, or "generations," they want to create. Each generation corresponds to a set of matrices along with their multiplication results. The user can input a specific number or just press Enter to default to creating just one generation.

The script allows the user to define or let the program randomly select several parameters that influence how the matrices are generated (BW,DW,AW,SPN).

These parameters determine the characteristics of the matrices that are generated. For example, the maximum dimensions of the matrices are calculated based on the ratio of BW to DW. This ensures that the matrices are sized appropriately for the bit and data widths specified, which is critical for ensuring the operations performed on them are valid and meaningful in the context of the design being tested.

Once the user has specified the number of generations and the parameters for each, the script proceeds to create the necessary directories to store each generation's data. This organization into folders makes it easier to manage and reference the data later, especially when dealing with multiple generations of test data.

For each generation, the script then generates the matrices A, B, and C, and an identity matrix I. The matrices A and B are multiplied together, and depending on whether a ModBit is true, matrix C is added to the result. This resulting matrix and all the original matrices are saved in text files within their respective generation's folder. Alongside these matrices, a parameters file is also saved, which documents the used parameters and the specifics of the matrix generation for that run.

This generated data is essential for testing because it is used by the Golden Model to verify the correctness of the design under test (DUT). The Golden Model takes the generated matrices and the results of their operations and compares them to what the DUT produces. Any mismatches can then be analyzed to point on issues in the design.

Stimulus

Our `stimulus` module in Verilog is designed to simulate a test environment that sets up and verifies digital designs, particularly focusing on the functionality of the APB (Advanced Peripheral Bus. This module is critical in the process of checking the design under test (DUT) by providing it with realistic scenarios that might be encountered in actual operations.

The stimulus module acts like the brain of the test operation. It starts by reading from several predefined text files, each containing essential data and parameters necessary for the tests. These files include matrices and operational flags that tell the DUT how to behave under certain conditions. By doing so, the stimulus module ensures that the DUT not only performs its tasks correctly under ideal conditions but also can handle real-world inputs and situations.

The key function of this module is to manage and configure the data transfer over the APB bus, operating by the protocol requirements. The stimulus module configures each test case by setting the matrix dimensions and operational flags, then sends these configurations along with the matrix data to the DUT.

The stimulus module is equipped with various tools and mechanisms. It uses file descriptors to manage file operations, ensuring that all data needed for the tests is accessible and correctly read..

Main objectives of the stimulus:

- Flat the input matrices from the generator.(in order to work with the hardware properly.
- Read the parameters from the generator such as DW, BW…
- Read the input matrices from the generator according to its sizes(N, K, M)
- Acts as the APB master simulator as following:
  In order to assign the specific matrix (A, B OR C) into our hardware memory we simply operated a mult operation. We assigned matrix A into operand A address, considering the address_offset (the sub address in order to hit a specific line in the matrix), and the same we did to matrix B into operand B address. Then, by assigning specific values to the control register we outcomes the mult operation into our hardware.
  In matrix C case, we also read an I matrix so we can mult it with matrix C in order to assign it into hardware memory.(we did it by the same operation as discussed with matrices A&B).
  Then, according to which place in mem (by the SPN value) we populate the resault matrix from hardware into mat_res_by_hw.

Furthermore, the stimulus module is designed with flexibility and robustness. It includes error handling routines that can detect and respond to file access issues or data mismatches during the testing. These routines are important for maintaining the correcteness of the test process, ensuring that any potential issues are addressed promptly to avoid impacting the overall testing outcomes.

TestBench (tb_overall)

The `tb_overall` module designed to validate the performance and correctness of our DUT. This test bench is essential for ensuring that it operates accurately under various scenarios, and effectively our TOP Entity for the project .

Key Components of the Test Bench:

1. Clock and Reset Signals: it generates a stable clock signal and manages a reset sequence to initialize all components.

2. Interface Management: It use interface_m to communicate between the test bench components. This interface handles all necessary signals such as data, control, and status indicators, ensuring smooth coordination across the test environment.

3 .Modules within the Test Bench (Holds instantiations of the GoldModel, Stimulus, portmap into the 'top' of part 1 – the matmul:

- Stimulus Module: This component drives the test by providing the DUT with required data and control signals, simulating external inputs that the DUT would encounter in actual deployment.

- Golden Model: It acts as a reference to ensure the DUT's output is correct. By processing the same inputs as the DUT and comparing outputs, it helps identify any discrepancies or errors in the DUT's processing.

- DUT: The central component under test, configured with parameters that define its operational characteristics such as data width and address width, receives inputs via the interface and processes them according to its logic.

**Operation and Verification: **

The test bench synchronizes interactions among these components monitoring the DUT's response to inputs and comparing them against the golden model. This method verifies the DUT's ability to handle various data types and operations, ensuring its readiness for real-world functioning.

PART 1:

Functional descriptions & Block diagrams:

1. Matmul (Top)

Description

This module represents the top-level. It integrates various sub-modules to facilitate communication with an APB bus, perform matrix padding, and execute the actual multiplication operation. The module acts as the central point for the matrix multiplication system, orchestrating interactions between its constituent parts.

It provides an interface to the APB bus through the integrated apbslave module, enabling data exchange and configuration.

Inputs and Outputs:

Inputs

- clk_i: Clock signal.
- reset_ni: Active low reset signal.
- psel_i: APB select signal.
- penable_i: APB enable signal.
- pwrite_i: APB write enable signal.
- pstrb_i: APB write strobe signal.
- pwdata_i: APB write data signal.
- paddr_i: APB address signal.

Outputs:

- pready_o: APB ready signal.
- pslverr_o: APB slave error signal.
- prdata_o: APB read data signal.
- busy_o: Busy signal, indicating ongoing calculations within the system.

The module incorporates three main sub-modules(portmaps):

1. apbslave: This sub-module serves as the APB slave interface, handling communication with the APB bus. It allows data exchange for configuration, operand transfer, and result retrieval.

2. mat_pad: This sub-module handles matrix padding. It takes matrices A and B from the APB slave and prepares them for the multiplication operation by adding necessary padding elements.

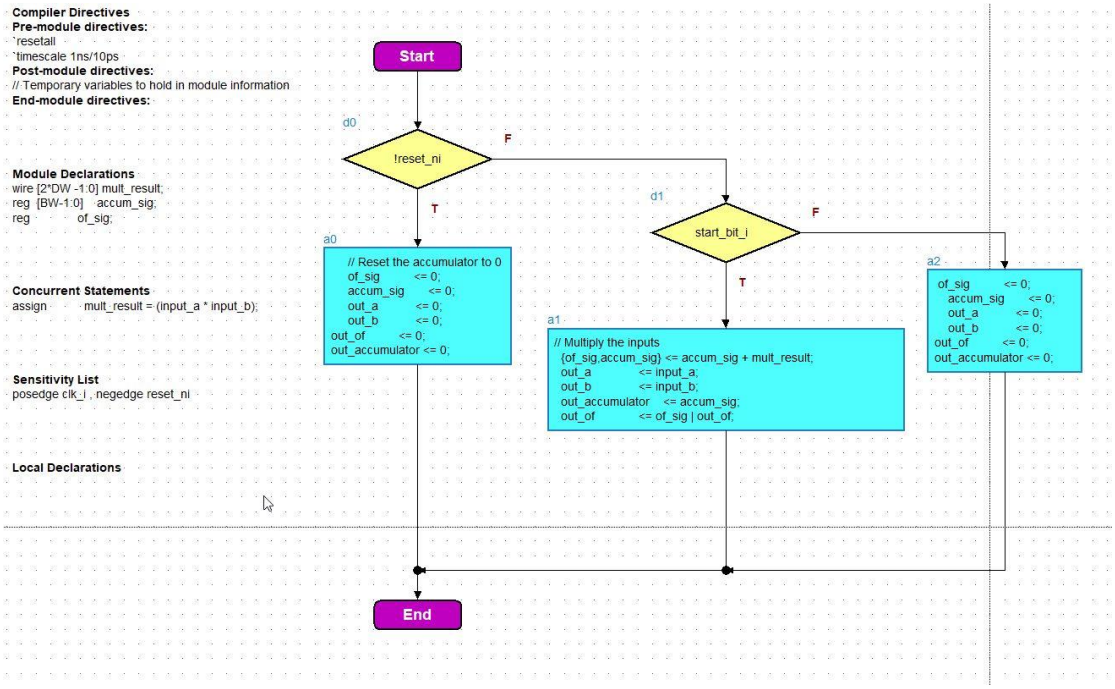
3. matmul_calc: This sub-module performs the core matrix multiplication calculations. It receives the padded matrices and computes the resulting product matrix.

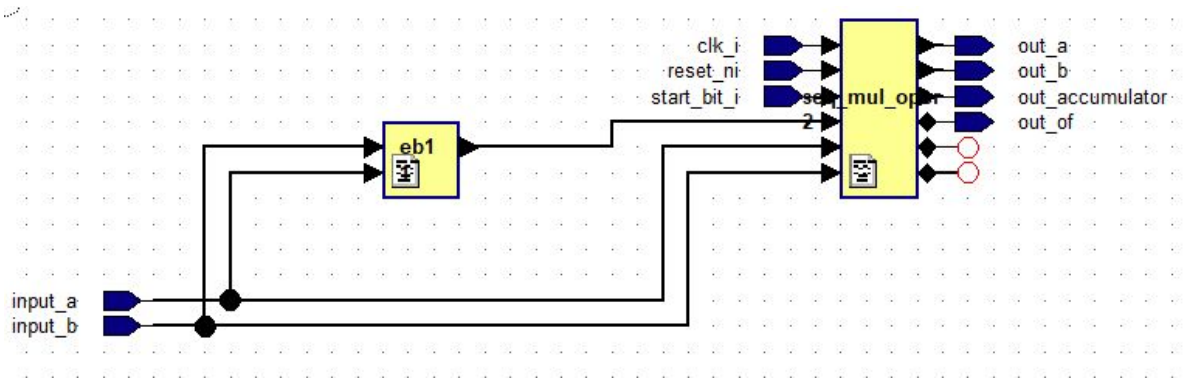
**The specific functionalities of the sub-modules (apbslave, mat_pad, and matmul_calc) are likely described in their respective documentation.

In summary, the matmul_top module acts as the conductor for the entire matrix multiplication system, enabling communication, data management, and execution of the core mathematical operation.

2. Precess element (pe)

a. Block diagram&Flow Chart





b. Description

This PE takes two input operands (input_a and input_b), a start signal (start_bit_i), and performs the following operations:

DW: The data width (number of bits) determined.

BW: The bit width of the accumulator to accommodate potential overflow.

1. Multiplication: When the start_bit_i is high, the PE multiplies input_a and input_b and stores the result in a temporary variable (mult_result).

2. Accumulation: The mult_result is added to a register (accum_sig) that acts as an accumulator, storing the partial sum of the multiplication process. Overflow is also detected during addition. (you can see we are using only signed)

3. Output: The PE outputs various signals:

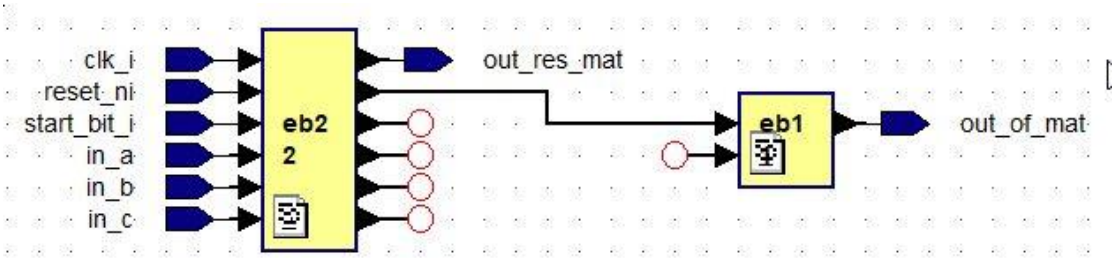
- out_a and out_b: The original input operands.
- out_accumulator: The current value of the accumulator, representing the partial sum.
- out_of: Overflow flag, indicating if an overflow occurred during addition.

4. Reset: When the reset signal (reset_ni) is low, the PE resets its internal state, including the accumulator, overflow flag, and output registers.

In our implementation, input_a (b) is a single element from the matrix A(B), which the pe is getting from matmul_calc (after a few conditions in that module which we will discuss later).

3. Matmul_calc

a. Block diagram



b. Description

Using as TOP module to generate the pe's.

This module takes three input matrices (in_a, in_b, and in_c) represented as one-dimensional arrays and performs the following:

Addition: Adds the resulting product matrix to the third input matrix (in_c).

Overflow Detection: Identifies and flags any overflows that occur during the multiplication and addition operations.

Inputs:

- clk_i: Clock signal .reset_ni: Reset signal, active low.
- start_bit_i: Start signal to initiate the matrix multiplication process.
- in_a: One-dimensional vector representing matrix A (elements in row-major order).
- in_b: One-dimensional vector representing matrix B (elements in row-major order).
- in_c: One-dimensional vector representing an additional matrix to be added to the result.

Outputs:

- out_of_mat: One-dimensional vector of overflow flags, where each bit indicates overflow for the corresponding element in the result matrix.
- out_res_mat: One-dimensional vector representing the resulting product matrix after adding in_c.

The module utilizes several internal signals to connect and manage the PEs:

cell_accumulator: An array to store the partial sums computed by each PE.

cell_a and cell_b: Arrays to store the current values of elements from matrices A and B used by each PE and are generated from the PE. The meaning is that those are the wires connecting each PE to another.
By the conditions in the module (which we mentioned we will discuss in the PE description), our implementation decides which element it should connect to each PE(rather its from the in_a (b) vec or the cell_a(b) wire.

of_pes: An array of flags indicating overflows from individual PEs.

of_mat_c: An array of flags indicating overflows during the addition of in_c.

The pe_input_a and pe_input_b arrays are calculated to provide appropriate data for each PE based on its position in the systolic array.

A loop using generate statements instantiates the required number of PEs (MAX_DIM * MAX_DIM) based on the defined parameters.

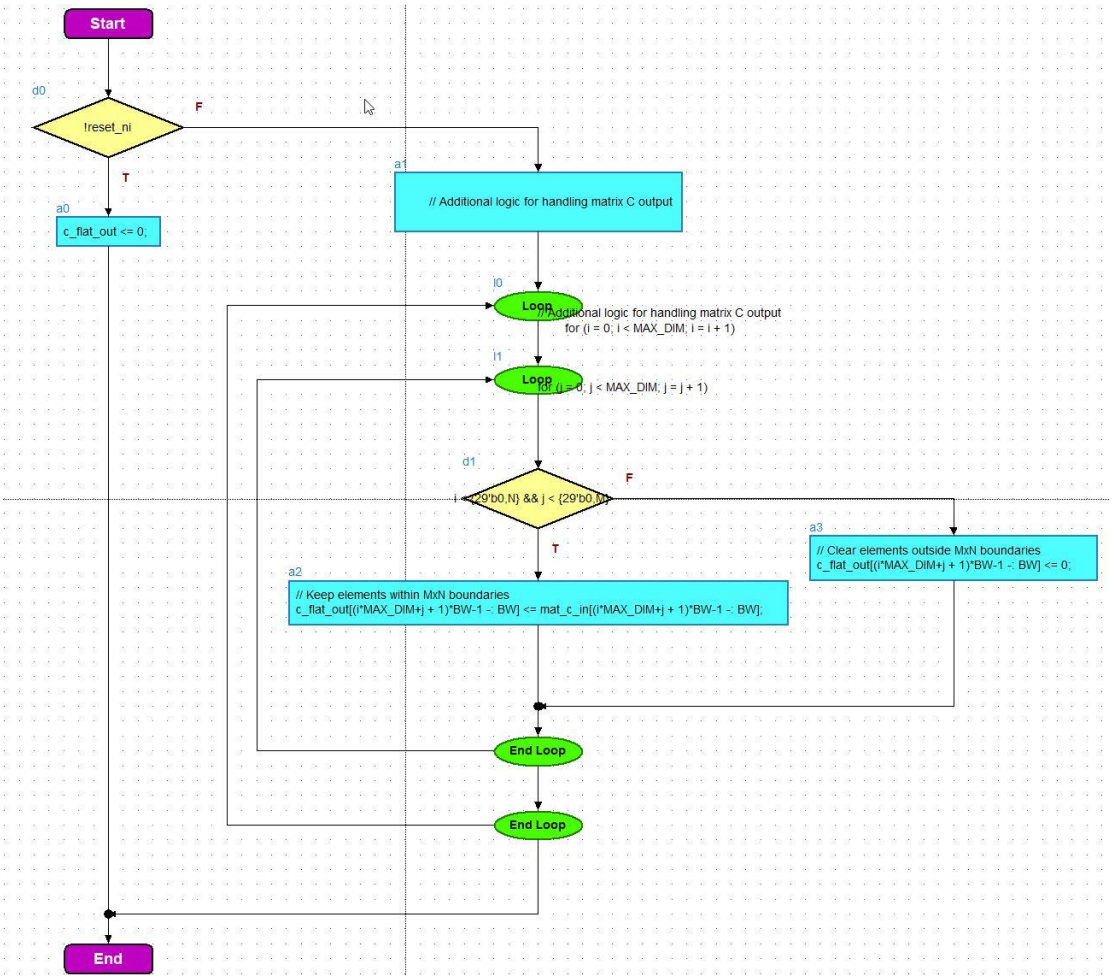
Each PE instance performs multiplication, accumulates partial sums, and detects overflows.

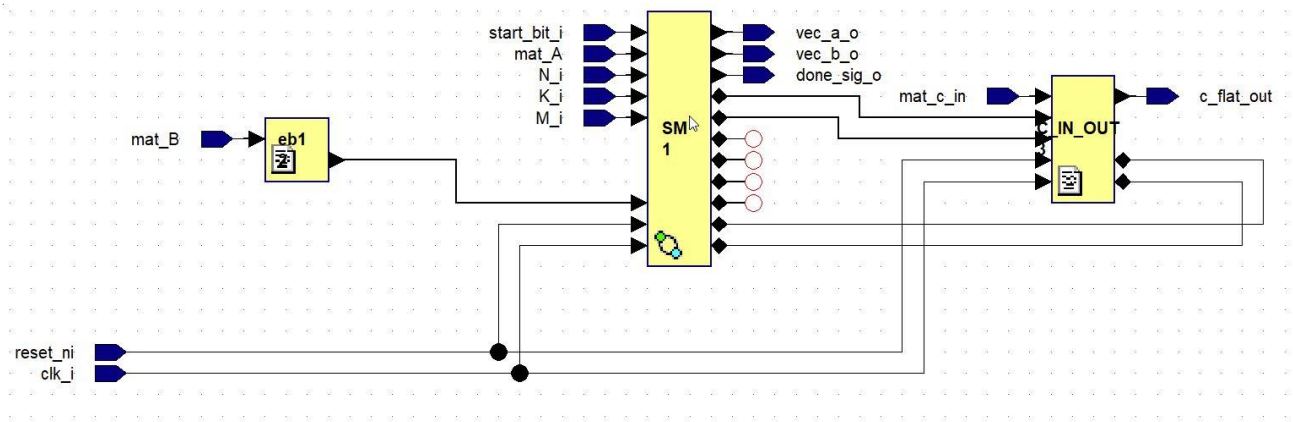
The final result is obtained by adding the corresponding elements from cell_accumulator and in_c, while also capturing any overflows that occur during this addition, which in the end is connected to out_res_mat.

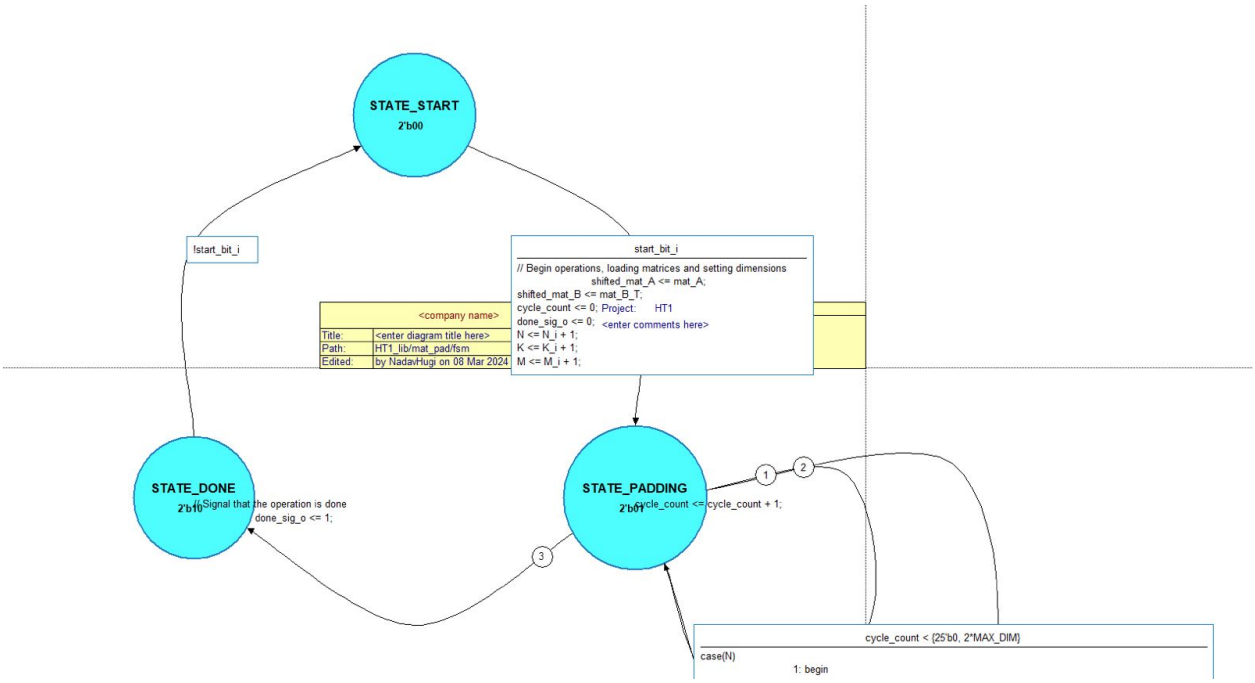
4. matpad

a. Block diagram &Flow Chart& State machine

b. diagram







c. Description

Using as TOP module to matmul_calc.

Its primary function is to prepare the input matrices for the matrix multiplication process by handling potential size mismatches and adding necessary padding.

The mat_pad module takes matrices A, B, and C (along with their dimensions) as input and prepares them for the matmul_calc module, which performs the actual multiplication.

After processing, the mat_pad module outputs the padded matrices (A and B) and the original matrix C to the matmul_calc module.

The mat_pad module operates in three main states:

Start: Upon receiving a start signal, it loads the input matrices and dimensions and transitions to the padding state.

Padding: In this state, the module iterates through a specific logic based on cycle count and matrix dimensions to pad the input matrices A and B with zeros as necessary.

This process is being handled by splitting into 2 cases (N&M) which by the input dimensions of the matrices decides which elements vector(vec_a_o(b)) to stream out onto matmul_calc. This process is shifting in each iteration the signal shifted_mat_A(B) and due to the conditions the relevant elements are being assigned.

**Note that in our implementation the matmul_calc(using the pe's) always generate MAX_DIM*MAX_DIM pe's and by the logic dynamically padding operation (corresponding to N&K&M as mentioned earlier) selects relevant elements or zeros.

Done: When the padding process is complete, the module enters the done state and signals completion.

The output matrix C (c_flat_out) is left unmodified, but elements outside the valid MxN boundaries are cleared to zeros.

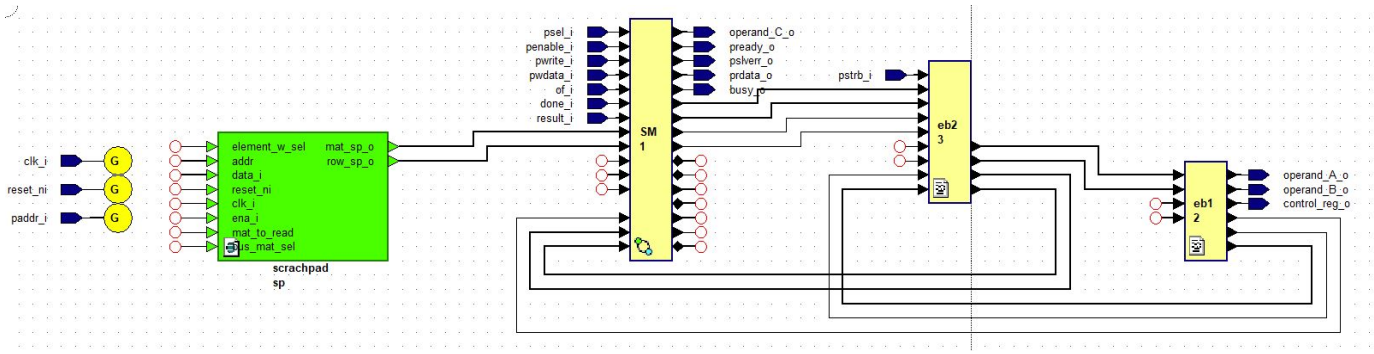
The module utilizes various internal signals to manage the padding process:

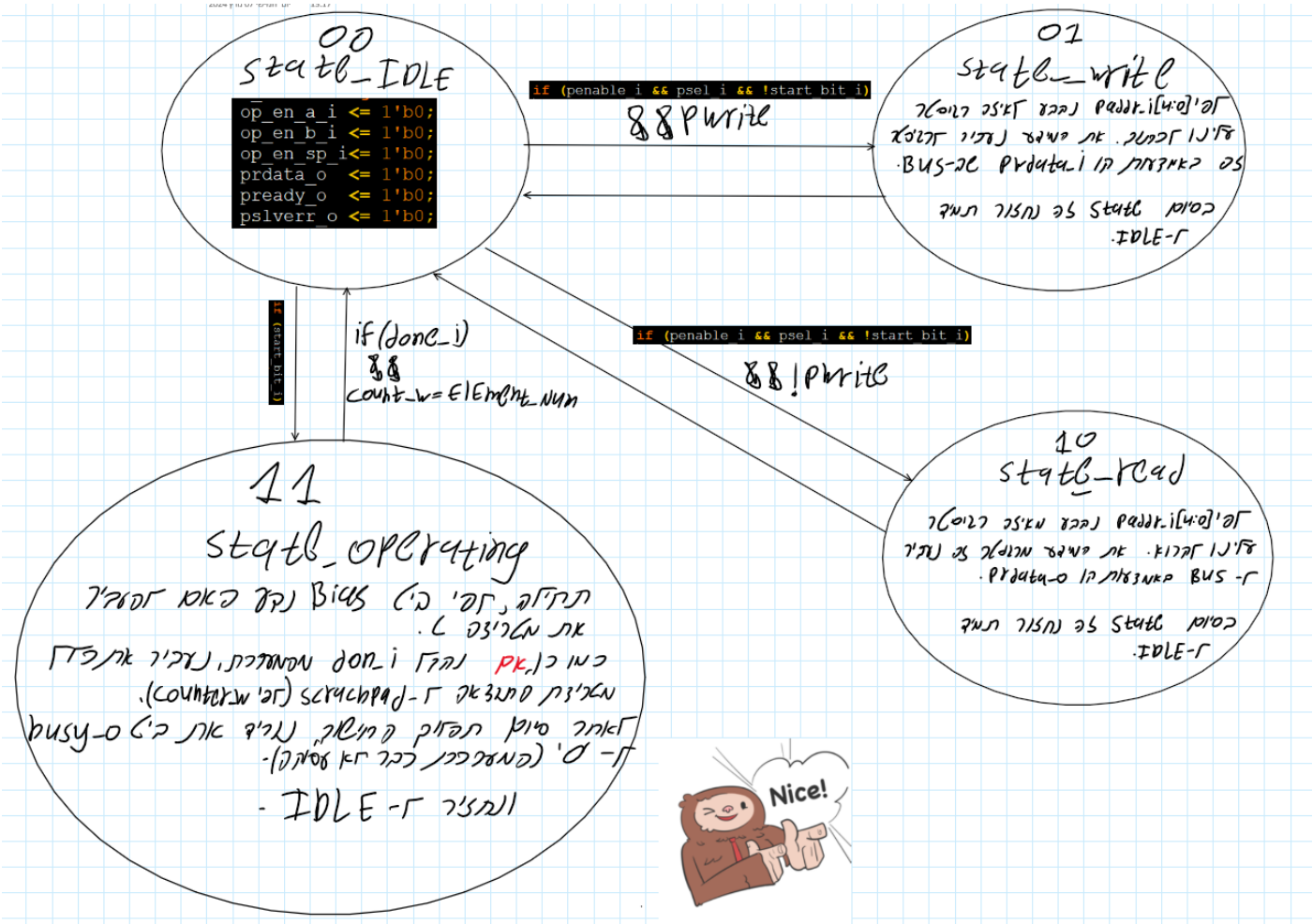
- shifted_mat_A and shifted_mat_B: Shifted versions of the input matrices used for padding.
- mat_B_T: Transposed version of matrix B (pre-computed for efficiency).
- cycle_count: Keeps track of the current cycle during the padding process.(the main condition of padding process - cycle_count < 2*MAX_DIM – is to make sure the multiplication is being set as needed.
- N, K, and M: Decoded dimensions of the input matrices.

Overall, the mat_pad module plays a crucial role in ensuring compatibility between matrices of different sizes before feeding them into the matmul_calc module for efficient matrix multiplication.

5. Apbslave (bus)

a. Block diagram& State machine diagram





b. Description

This module acts as an APB slave peripheral within a larger design. It facilitates communication and data exchange between the design and an external APB bus.

This module allows reading and writing data to various internal registers and memory elements through the APB bus. It supports operations on matrices, including operand storage and results retrieval.

It uses as TOP entity to scrachpad & reg_file_op (A&B instances).

Inputs and Outputs:

Inputs:

- clk_i: Clock signal.
- reset_ni: Active low reset signal.
- psel_i: APB select signal.
- penable_i: APB enable signal.
- pwrite_i: APB write enable signal.
- pstrb_i: APB write strobe signal.
- pwdata_i: APB write data signal.
- paddr_i: APB address signal.
- of_i: Overflow bits.
- done_i: Done signal indicating completion of an operation.
- result_i: Result of the operation performed within the design.

Outputs:

- operand_A_o: Output for operand A matrix data.
- operand_B_o: Output for operand B matrix data.
- operand_C_o: Output for operand C matrix data (read from scratchpad).
- control_reg_o: Output of the control register, containing various control bits.
- pready_o: APB ready signal, indicating the module's readiness for communication.
- pslverr_o: APB slave error signal, indicating any errors encountered during communication.
- prdata_o: APB read data signal, providing data from internal registers or memory based on the address.
- busy_o: Busy signal, indicating when the module is performing an operation and unavailable for communication.

The module utilizes a state machine (current_state) to manage different operational stages:

Idle: Waiting for APB communication signals.

Write: Handles data writing to specific registers based on paddr_i[4:0](the 5 bits we defines as which register we are working with at the moment). This can involve updating the control register, operand registers, or triggering the start of an operation. At the end of this state we wil go straight back to state IDLE.

Read: Provides data from specific registers or memory locations based on paddr_i[4:0](the 5 bits we defines as which register we are working with at the moment). This could be the content of the control register, operand registers, or specific sections of the scratchpad memory. At the end of this state we wil go straight back to state IDLE.

Operating: Performs the designated operation on the operands and stores the result.

It employs various registers and memories:

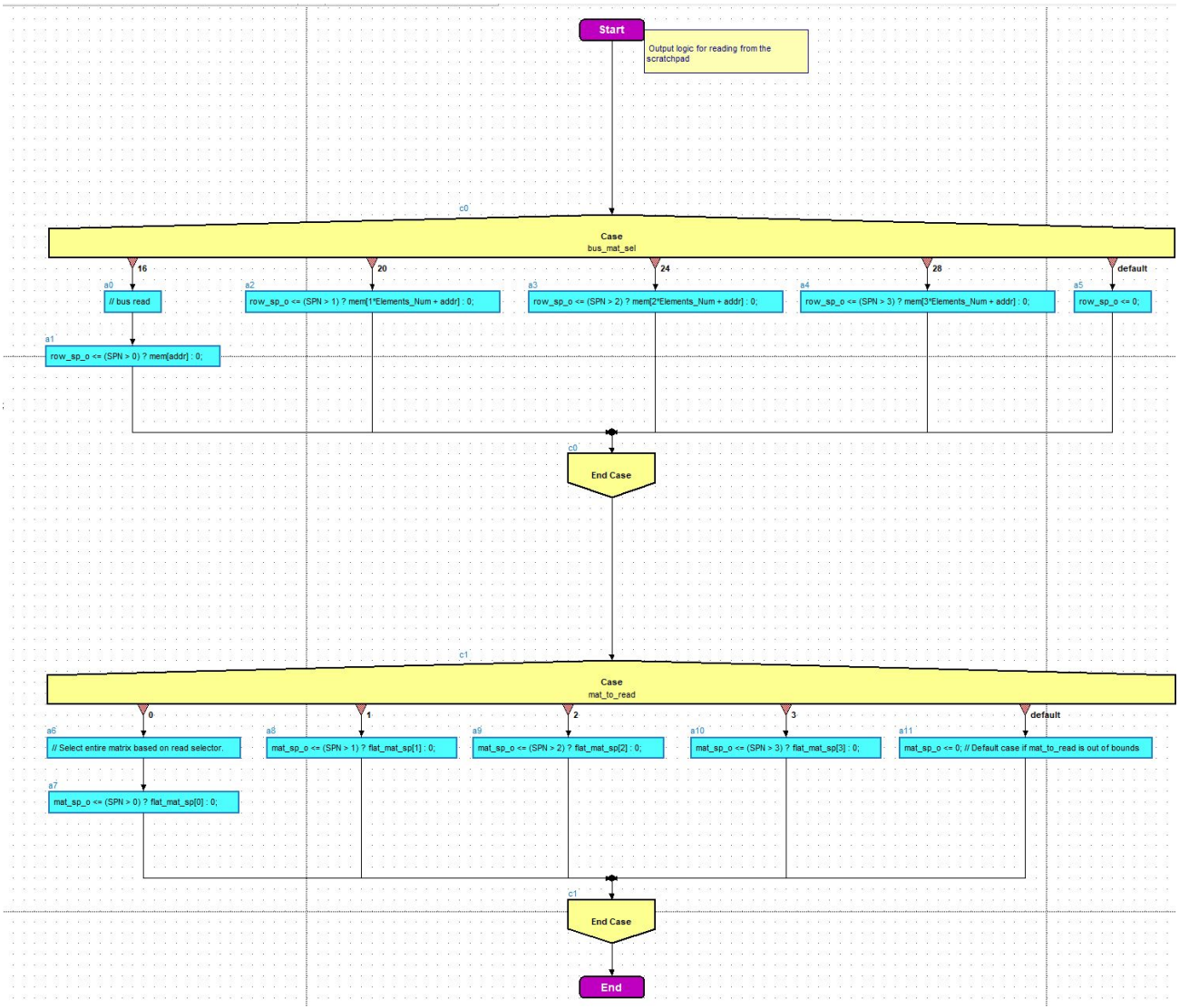
- control_reg: Stores control bits that govern the module's behavior.
- flags_reg: Stores flag bits related to the operation's status.
- result_reg: Temporarily stores the result before transferring it to the scratchpad.
- sp: Scratchpad memory (implemented as a separate module) for storing intermediate results.
- Additional internal signals facilitate data flow and communication within the module.

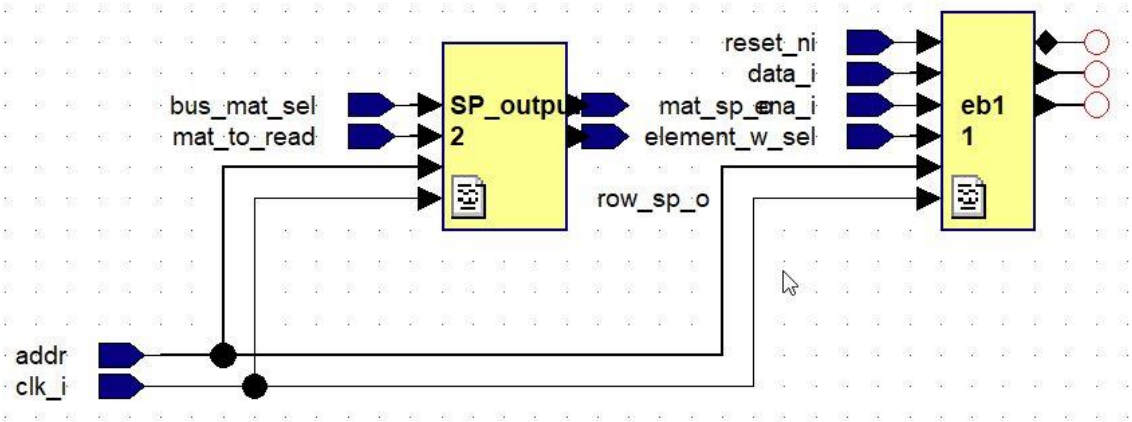
Matrix operations: Upon receiving a start signal from the control register, the module performs the designated operation on the operands stored in the operand registers. It leverages the scratchpad memory (sp) to store and retrieve intermediate results efficiently.

This process is being made if done_i is high (finishd the multiplication) and by (count_w < Elements_Num) we know we finishe writind the resault mat to the SP. Moreover, as we will discuss in the description of scrachpad mosule, count_w is connected to addr in scrachpad which indicates the specific row of the matrix.

6. Scrachpad (memory)

a. Block diagram & Flow Chart





b. Description

This module acts as a dedicated memory unit specifically designed for storing and retrieving intermediate results within the design.

This module provides multiple sections of scratchpad memory, each capable of storing a complete matrix. It allows reading and writing data to specific sections and addresses within the memory.

Inputs and Outputs:

Inputs:

- clk_i: Clock signal.
- reset_ni: Active low reset signal.
- addr: Address for reading/writing data within the selected scratchpad section.
- bus_mat_sel: Selector signal for read operations based on the APB bus.
- data_i: Data input for writing to the selected address within the scratchpad.
- ena_i: Enable signal for write operations.
- mat_to_read: Selector signal for reading an entire scratchpad section.
- element_w_sel: Selector signal for write operations, indicating the specific scratchpad section to write to.

Outputs:

- mat_sp_o: Output of the entire selected scratchpad section as a flat vector.
- row_sp_o: Output of a specific row based on the address and selected section.

The module utilizes an internal memory array (mem) to store the data. this memory is organized into multiple sections (SPN), with each section capable of holding an entire matrix(MAX_DIM*MAX_DIM= Elements_Num.

Additionally, separate flattened output registers (flat_mat_sp) are generated for each section to efficiently provide the entire section's data as a single vector.

Reading:

The row_sp_o output provides the contents of a specific row based on the addr (which the module gets as input from apbslave and indicates which row to select in the specific matrix by the SPN) and the selected section using a case statement.

**Further more, note that this addr gets its value from the signal op_addr in the apbslave, which is 1 or 2 buts depands on the MAX_DIM value. If its only 1 but (due to MAX_DIM = 2) so we connected 1 bit unused to addr because we need only 1 bit (only 2 rows).

The mat_sp_o output provides the entire contents of the selected section as a flat vector based on the mat_to_read signal and a case statement.

Writing:

When the ena_i signal is high, the module enables writing operations .The data_i input provides the new data to be written to the selected address within the section specified by element_w_sel.

Only the memory location corresponding to the provided address and selected section is updated.

A generate loop (gen_i and gen_j) efficiently packs the individual memory elements (mem) of each section into separate flattened output registers (flat_mat_sp) for faster access in order we can work with it properly.

The SP_output always block handles the read operations based on different input signals:

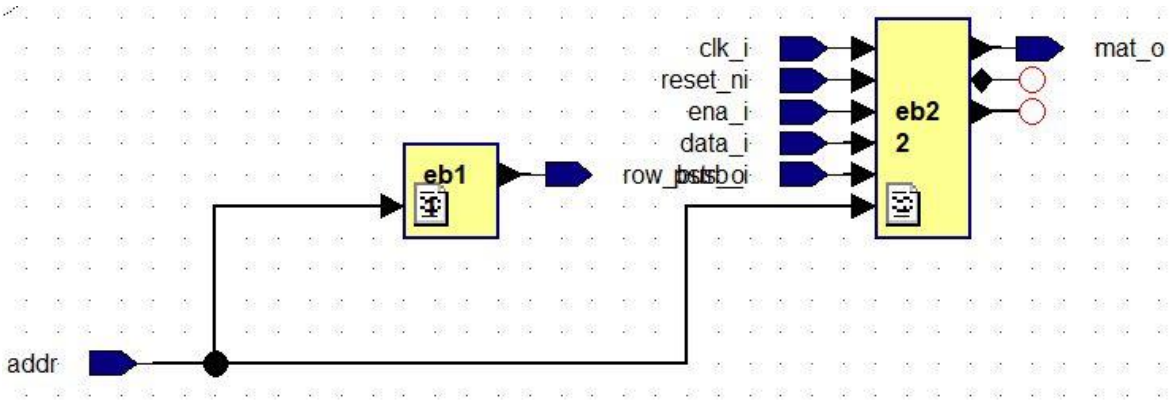
bus_mat_sel is used to select a specific section for reading a single row using a case statement. Its connected to paddr_i[4:0] which in our implementation we defined in apbslave as 5 bits to decide which location in PS we need to use at the moment.

mat_to_read is used to select an entire section for reading its complete contents as a flat vector using another case statement.

Overall, the scrachpad module provides a flexible and efficient memory solution for storing and retrieving intermediate results within the design, facilitating data management and potentially improving performance.

7. Reg_file_op (register file)

a. Block diagram



b. Description

This module serves as a memory unit specifically designed to store and manipulate matrices and acts as a register file capable of storing and managing a matrix with configurable dimensions.

It provides functionalities for reading and writing elements to specific rows within the matrix. This memory is organized as rows, with each row having a width of DW * MAX_DIM bits.

Inputs and Outputs:

Inputs:

- clk_i: Clock signal.
- reset_ni: Active low reset signal.
- ena_i: Enable signal for writing operations.
- data_i: Input data (one row of the matrix) to be written.
- addr: Address to select a specific row for read or write operations.
- pstrb_i: Byte enable signals to control the update of individual bytes within a row during write operations.

Outputs:

- mat_o: Output of the entire matrix as a single flat vector.
- row_bus_o: Output of the selected row based on the provided address.

Reading: The row_bus_o output provides the contents of the selected row based on the addr input.(which is the same addr we described previously in SP module explanation)

Writing: When the ena_i signal is high, the module enables writing operations.

data_i input provides the new data to be written to the selected row.

pstrb_i byte enable signals control which bytes within the selected row are updated(which element in the specific row). **Only** the byte positions corresponding to high bits in pstrb_i are updated with the corresponding bytes from data_i.

A generate loop (put_mat_lines_to_out) efficiently packs the individual rows stored in the memory array (mem) into a single flat output vector (mat_o).

Another generate loop (choos_exact_element_in_row) with a nested always block handles the write operations. During a rising edge of the clock (posedge clk_i), the block checks the enable signal (ena_i) .If enabled, the block selectively updates specific bytes within the addressed row (by the same addr signal) and based on the value of pstrb_i decides the exact element in the row to write\not write.
Only elements corresponding to high bits in pstrb_i are replaced with the corresponding elements from data_i.

Rules:

Score/Total Possible Score: 597/656 Excludes 8 Disabled Rules

RuleSet Hierarchy Report:

RuleSet	Score	%	Error	Warning	Note	Rule	Guideline	Total	Disabled
My_Essentials_Policy	597/656	91%	0	11	5	0	16	168	8
Essentials	179/185	97%	0	3	0	0	0	47	2
Coding Practices	81/83	98%	0	1	0	0	0	19	0
Downstream Checks	92/94	98%	0	1	0	0	0	22	2
Code Reuse	6/8	75%	0	1	0	0	0	6	0
DO-254	184/205	90%	0	8	5	0	0	54	4
Coding Practices	82/88	93%	0	3	0	0	0	18	1
Safe Synthesis	88/92	96%	0	2	0	0	0	25	3
Design Reviews	14/25	56%	0	3	5	0	0	11	0
RMM	234/266	88%	0	0	0	0	16	67	2
2 - Basic	132/144	92%	0	0	0	0	6	25	1
2.01 - General Naming Conventions	16/20	80%	0	0	0	0	2	6	0
2.02 - VITAL Naming Conventions	40/40	100%	0	0	0	0	0	4	0
2.10 - Port Ordering	4/6	67%	0	0	0	0	1	3	0
2.15 - Use Meaningful Labels	20/24	83%	0	0	0	0	2	4	0
3 - Portability	32/38	84%	0	0	0	0	3	16	1
3.01 - Use IEEE Types	16/16	100%	0	0	0	0	0	4	0
3.07 - Coding for Translation	12/14	86%	0	0	0	0	1	7	0
4 - Clocks & Resets	22/26	85%	0	0	0	0	2	9	0
4.01 - Avoid Mixed Clock Edges	12/12	100%	0	0	0	0	0	2	0
4.05 - Gated Clocks	4/4	100%	0	0	0	0	0	2	0
5 - Synthesis Coding	46/50	92%	0	0	0	0	2	13	0
5.09 - Coding Sequential Logic	4/6	67%	0	0	0	0	1	3	0
6 - Synthesis Partitioning	2/8	25%	0	0	0	0	3	4	0

1. **Avoid Asynchronous Reset Release:**
   The task instructs to use asynchronous reset.
2. **Use a separate line for each statement:**
   We were instructed to ignore this.
3. **Avoid Unresettable Register:**
   We don't aim to reset memory upon reset.
4. **Ensure Register Controllability:**
   We had problems implementing N,K,M as registers so we implemented them as integers.
5. **Avoid Shared Clock and Reset Signal:**
   Kjdfngjkdhgnksjhfds
6. **Ensure Consistent FSM State Encoding Style:**
   We were told in class that defining FSM states as arameters is also acceptable.