University of Connecticut
ECE 3401: Digital Systems Design
Spring 2024

Programming Assignment 2: A Row-wise Matrix Multiplication Design

Due March 22, 2024 (Friday) @ 11:59 PM on HuskyCT

1 Introduction

In this assignment, we will perform the multiplication of two matrices A and B, and output the calculations to a third matrix, C. The goals of this assignment are to:

- Become familiarized with VHDL array data types.
- Implement a state machine in VHDL.
- Use test benches to verify the correctness of a design.

Section 2 reviews introductory-level linear algebra. Section 3 discusses the overall assignment. Section 3.1 discusses the state machine in depth. Section 4 discusses the test benches. Finally 5 summarizes the deliverable to submit.

You are given a file `matrix.vhd`, which contains the `matrix_2x4`, `matrix_4x3`, and `matrix_2x3` types for use in the source files. You do not need to modify this file. `pa2.vhd` contain skeleton code that you need to complete. The entirety of the state machine in contained within `pa2.vhd`. You are also provided with one test bench `smtest.vhd` which is partially complete. More details are given in 4.

2 Background

In this assignment, the first matrix A can be *dense* or *sparse*, meaning that the elements in the matrix can have mostly non-zero or zero elements respectively. A is fixed to a 2x4 (two rows and four columns) matrix. B is a *dense* matrix. B is fixed to a 4x3 matrix. The output matrix contains the same number of rows as A and the same number of columns as B; thus, the output matrix C is 2x3. The format of the two input matrices A and B, and the output matrix C is shown:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2.1 Row-Wise Multiplication

Row-wise multiplication uses a different approach than the classical approach [1]. In the row-wise multiplication, all non-zero scalar elements of each row of matrix A are multiplied with all elements of

[1] `https://en.wikipedia.org/wiki/Matrix_multiplication`

the corresponding rows of matrix B, where the row index of matrix B is determined by the column index of the non-zero value from matrix A. The results are accumulated in the corresponding elements of the row of output matrix C, as shown below:

$$C[i,:] = \sum_{j=1}^{n} A[i,j] \cdot B[j,:] \tag{1}$$

where $C[i,:]$ is the i^{th} row-vector of matrix C. The algorithm iterates over each non-zero element $A[i,j]$ of matrix A and performs a scalar-vector multiplication with $B[j,:]$, which is the j^{th} row-vector of matrix B. Note that all rows of the output matrix C need to be computed for this matrix multiplication. The detailed algorithmic pseudocode for row-wise multiplication is shown in Algorithm 1.

Algorithm 1 Row-wise Matrix Multiplication of A and B

```
procedure MatMult($A, B, C$)
    for $i = 1$ to 2 do                                  ▷ iterate through rows of $A$
        for $j = 1$ to 4 do                              ▷ iterate through columns of $A$
            if $A[i,j] \neq 0$ then        ▷ Skip matrix multiplication if the element of $A$ is non-zero
                for $k = 1$ to 3 do                      ▷ Iterate through the columns of $B$
                    prod_reg = $A[i,j] \cdot B[j,k]$   ▷ Update element-wise multiplication in a register
                    $C[i,k] = C[i,k] +$ prod_reg          ▷ Accumulate and update the output
                end for
            end if
        end for
    end for
    return $C$
end procedure
```

2.2 Row-Wise Multiplication Example

Given the following A and B input matrices:

$$A = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 0 & 7 & 0 \end{bmatrix} B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

The implementation of the algorithm must first initialize the output matrix C with zeros during the reset process. This is done to ensure that the accumulations can be performed directly in the C matrix:

$$C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The first non-zero of A is $a_{11} = 1$. It is in the **first row** and the **first column** of A. The row number corresponds to the row of C that will get updated. The column number corresponds to the row-vector of B that will get multiplied. So, the a_{11} scalar is multiplied by the **first row** row-vector of B, and this result is accumulated in the **first row** row-vector of C:

$$C = \begin{bmatrix} 0 + (1)(1) & 0 + (1)(2) & 0 + (1)(3) \\ 0 & 0 & 0 \end{bmatrix}$$

The a_{12} and a_{13} elements are zeros, so their computations are skipped. The next non-zero is $a_{14} = 4$ which is in the **first row** and the **fourth column** of A. So, the a_{14} scalar is multiplied by the **fourth row** row-vector of B, and this result is accumulated in the **first row** row-vector of C:

$$C = \begin{bmatrix} 1 + (4)(10) & 2 + (4)(11) & 3 + (4)(12) \\ 0 & 0 & 0 \end{bmatrix}$$

At this time, the computations for the first row of A have completed, the algorithm iterates to the second row of A for processing. In the second row of A, the only non-zero is $a_{23} = 7$ which is in the **second row** and the **third column** of A. So, the a_{23} scalar is multiplied by the **third row** row-vector of B, and this result is accumulated in the **second row** row-vector of C:

$$C = \begin{bmatrix} 41 & 46 & 51 \\ 0 + (7)(7) & 0 + (7)(8) & 0 + (7)(9) \end{bmatrix}$$

Since there are no remaining rows to be processed in A, the algorithm is done, and the final result of the multiplication is stored in matrix C.

$$C = \begin{bmatrix} 41 & 46 & 51 \\ 49 & 56 & 63 \end{bmatrix}$$

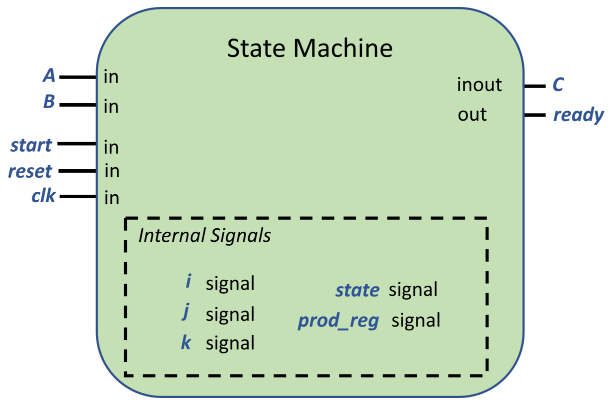
3 Design Overview

You will implement the 2x4 times 4x3 row-wise matrix multiplication Algorithm 1 in VHDL by using a state machine to sequence the operations. The vector multiplication / addition is performed one element at a time.

3.1 State Machine

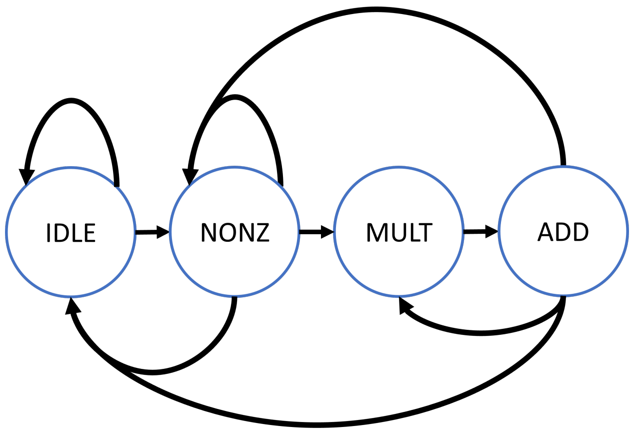
The state machine implementation is a Mealy state machine. The entirety of the state machine is defined in `pa2.vhd`, and each input / output signal to the module is mapped to the external signals to the test bench.

You are responsible for completing the transition and output logic of each state. You **are not allowed** to add additional signals to the architecture. The `reset` functionality is also provided in the `pa2.vhd` file and should not be further modified.



PA2 module FSM signal interface.

There are four states: IDLE, NONZ, MULT, and ADD. These states are used to implement Algorithm 1. Throughput this discussion, refer back to the algorithm for more details. On each rising edge of the `clk` signal, the state machine transition process should be invoked. The states of the SM are described below:



State machine states and incomplete transition edges.

- IDLE: This is the starting state of the SM. The SM remains in IDLE until the `start` signal is asserted. Once this happens, the iterators `i`, `j`, `k` are initialized, and the SM transitions to NONZ. When `start` is not asserted in IDLE, the SM remains in IDLE.
- NONZ: The current element of `A` is checked based on the corresponding `i` and `j` signals. If the element is zero, then multiply - add processing is skipped for it. The logic for `i`, `j`, and `k` must be set properly and the SM may transition to IDLE if the algorithm is done with its computations.

  If the element of `A` is non-zero, then the SM must proceed with the multiply - add sequence for the corresponding row-vector of `B`. The logic for `i`, `j`, and `k` must be set properly before transitioning to the next state, MULT.
- MULT: For the non-zero scalar `A` and an element of vector `B`, the multiplication output is stored in the `prod_reg` register. The SM then transitions to the ADD state. The logic for `i`, `j`, and `k` must be set properly before transitioning to the next state, ADD.

- ADD: The product stored in `prod_reg` is accumulated into the corresponding element of `C`. The state machine either (1) transitions back to IDLE if all computations for matrix `A` have completed, or (2) transitions back to MULT to operate on the next element of the row-vector `B` being processed, or (3) transitions back to NONZ to start operating on the next element of matrix `A`. The logic for `i`, `j`, and `k` must be set properly before transitioning to the next state.

When transitioning back to the IDLE state, the SM must assert the ready signal to indicate that the algorithm has completed and the results are ready in the matrix `C`.

4 Test Bench

The skeleton of the test bench `smtest.vhd` is given. Within `smtest.vhd`, constant `compC_1`, `compC_-2`, and `compC_3` matrices are defined with the expected output of various multiplications. Prior to beginning different matrix multiplications, the inputs `A` and `B` are modified to different values. You are required to fill in the testbench process to do the following:

After each start of the SM, using `for` loops, iterate through the completed output `C` and corresponding comparison matrices to confirm the correctness of your design. Use `assert` to verify that all the corresponding elements are equal (`compC_1`, `compC_2`, and `compC_3` are used for the first, second, and third multiplication, respectively). If any elements do not match, you should `report` an error message with a severity level of "FAILURE" (this will cause the simulation to stop at the problematic entry). In the event of a failure, you will need to debug your state machine. If the loops without any failed assertions, you should report an enthusiastic success message to the console.

5 Deliverables

Please submit a single PDF containing the following:

1. Your completed code of the following:
   - `pa2.vhd`
   - `smtest.vhd`
2. The following screenshots:
   - The output waveform from `smtest.vhd` from 0ns - 1050ns. Ensure that `A`, `B`, `C`, `start`, `clk`, `reset`, `ready`, `state`, `i`, `j`, and `k` are visible. You will need to add `state`, `i`, `j`, and `k` to the window from `pa2`, as it will not appear by default. Instructions are here. It is okay if the final matrix C / each state is not visible in this screenshot.
   - The output waveform windows of `smtest.vhd` from 0 - 250ns, 250 - 500ns, 500 - 750ns, and 750 - 1050ns. Show the same signals as in the previous one. **For these screenshots, ensure that every state in this range is clearly visible**.
   - A screenshot of the TCL console with your report messages.