VLSI Desing Project 1
Matrix ALU

Apostolos Kontarinis axk220238
Athanasia Karanika axk230133

Dallas 2023

Contents

Abstract

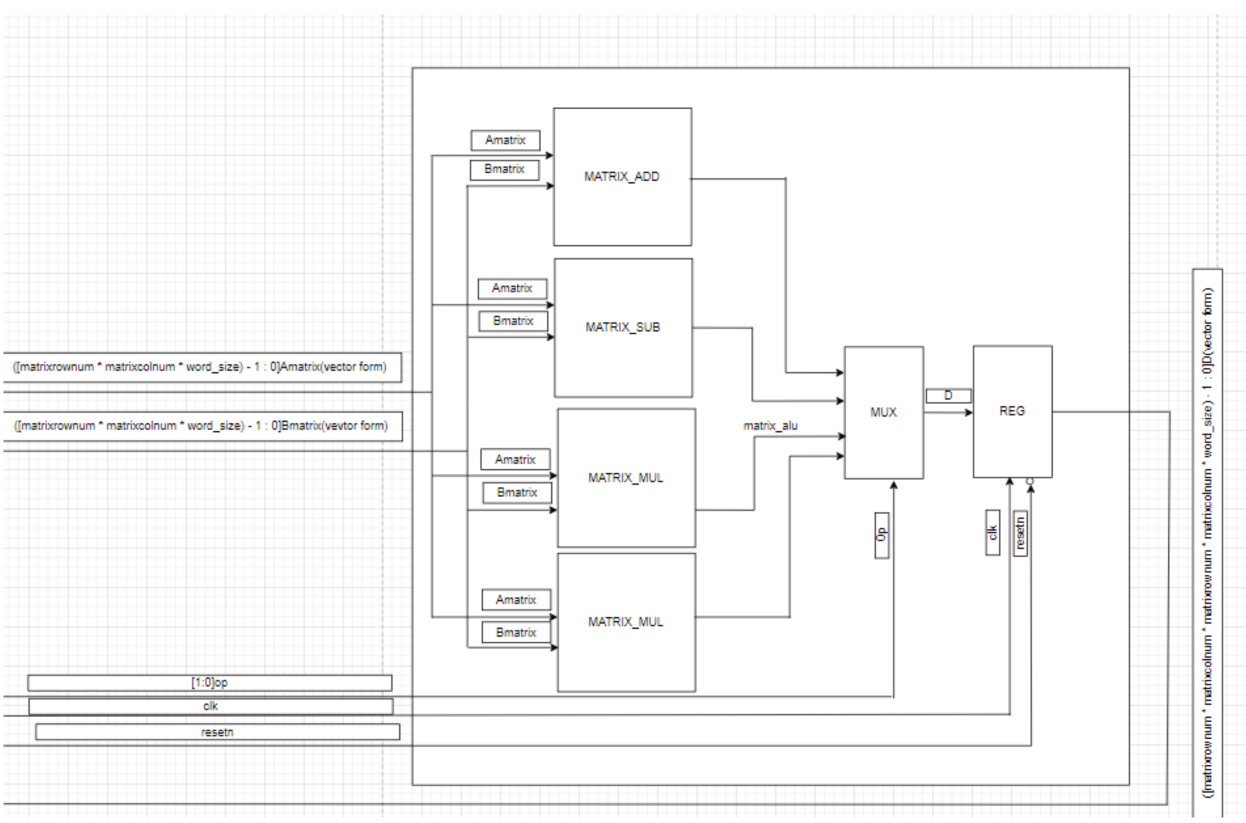
This report presents the design and implementation of a Verilog-based Matrix Arithmetic Logic Unit (Matrix ALU) capable of performing matrix addition, subtraction, multiplication, and Kronecker product operations on two square matrices of the same dimensions. The ALU takes a clock input (clk) and a reset input (resetn) which control the output register, an operation input (op) which is used to choose the operation that will appear at the output, 2 matrix inputs (A, B) and 1 matrix output (C) in vector form. E.g for matrix

$$A = \begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} \rightarrow A = [a_{00}a_{01}a_{10}a_{11}]$$

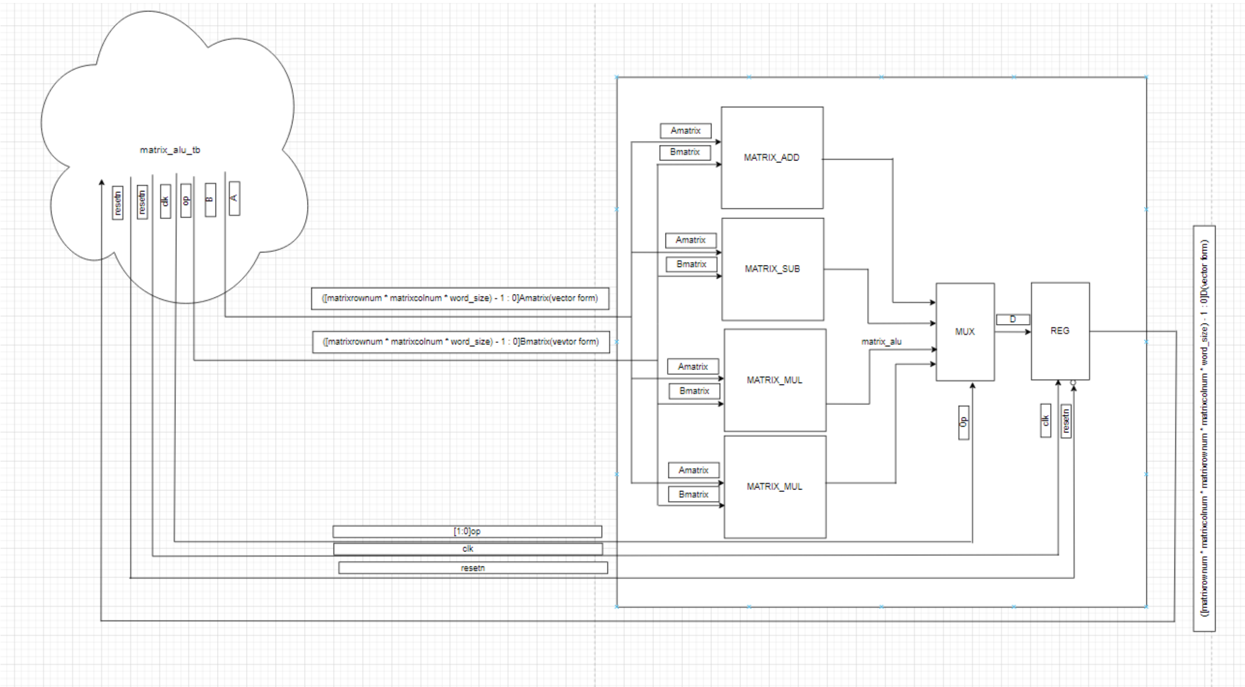
For the Kronecker product operation, the resulting output matrix has a size that depends on the dimensions of the input matrices. Unlike matrix addition, subtraction, and multiplication, which produce output matrices of the same size as the input, the Kronecker product generates an output matrix with dimensions determined by the Cartesian product of the dimensions of the input matrices. Specifically, the output matrix size is given by matrixrow * matrixcolumn * matrixrow * matrixcolumn.

In order to accommodate the potentially larger size of the Kronecker product output while maintaining a consistent format for all outputs, we have chosen to set the output matrix dimensions to be the largest possible size (i.e., matrixrow * matrixcolumn * matrixrow * matrixcolumn). In the cases of the other operations (Addition, Subtraction, Multiplication) which do not fully utilize this space, the remaining elements are set to a placeholder value, typically denoted as 'z' or zero.

Design's Block Diagram



Design's Block Diagram and connection with Testbench



Simulation Waveforms

Testbench changes op, the 2-bit control signal representing the matrix operation to be performed, at 100-time unit intervals, cycling through four different operation codes (2'b00, 2'b01, 2'b10, 2'b11) e.g addition, subtraction, multiplication and Kronecker product respectively. In each of the following 4 waveforms we examine the result of each operation. In the case of addition, subtraction and multiplication, the output signal size is determined by the product of the number of rows matrixrownum and columns matrixcolnum of the input matrices, multiplied by the word size. While the result may not occupy the entire signal width, the 'z' values simply indicate that the remaining bits are unused and do not affect the correctness of the operation and the result is stored in the last bits of output In this demo we have the following matrices:

$$A = \begin{vmatrix} 1 & 2 \\ 3 & 0 \end{vmatrix}, B = \begin{vmatrix} 5 & 6 \\ 7 & 8 \end{vmatrix}$$

Note that the result is shown in Testbench in hexadecimal

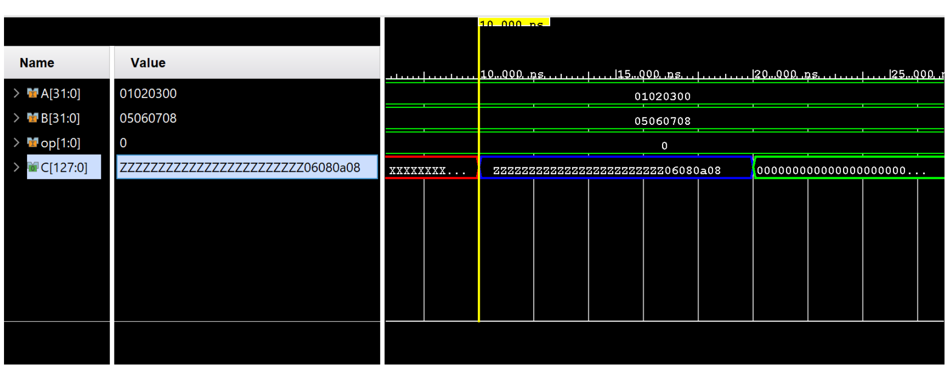


Figure 1: Matrix addition

As we observe in Fig.1, the result is not initialised until 10ns due to the fact that the clk becomes for first time 1 at 10 ns. The result is stored in the last matrixrow * matrixcol * wordsizebits and is ZZZZZZZZZZZZZZZZZZZZZZZZ06080a08 which means 06080a08 that is the 2's complement of:

$$C = \begin{vmatrix} 6 & 8 \\ 10 & 8 \end{vmatrix}$$

in decimal, which is correct.

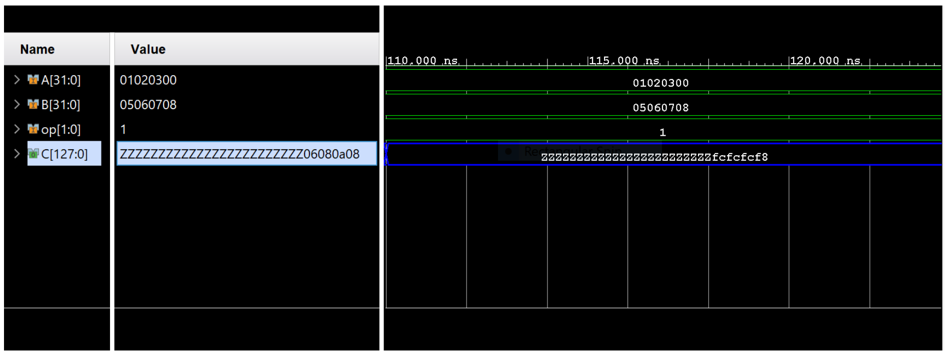


Figure 2: Matrix subtraction

As we observe in Fig.2, the result becomes ZZZZZZZZZZZZZZZZZZZZZZZZfcfcfcf8 which means fcfcfcf8 that is the 2's complement of:

$$C = \begin{vmatrix} -4 & -4 \\ -4 & -8 \end{vmatrix}$$

in decimal, which is correct.

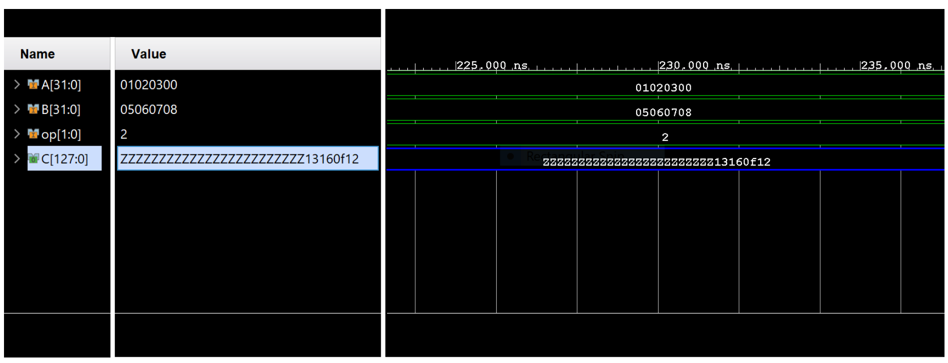


Figure 3: Matrix multiplication

As we observe in Fig.3, the result becomes ZZZZZZZZZZZZZZZZZZZZZZZZ13160f12 which means 13160f12 that is the 2's complement of:

$$C = \begin{vmatrix} 19 & 22 \\ 15 & 18 \end{vmatrix}$$

in decimal, which is also correct.

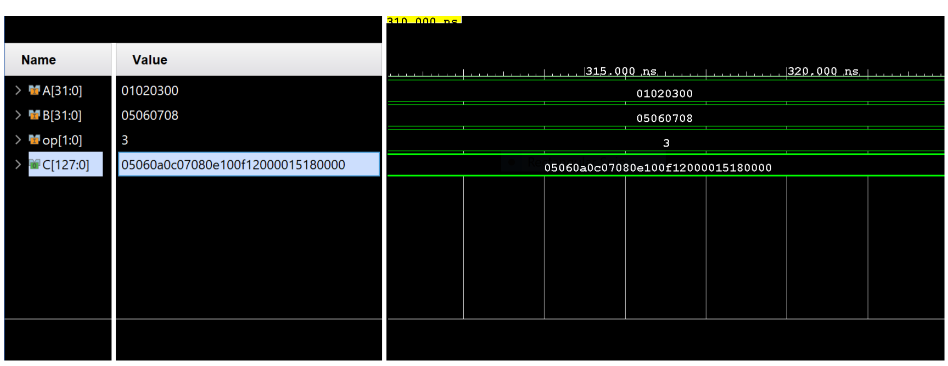


Figure 4: Matrix Kronecker product

As we observe in Fig.4, the result becomes 05060a0c07080e100f12000015180000 which means that is the 2's complement of:

$$C = \begin{vmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 0 & 0 \\ 21 & 24 & 0 & 0 \end{vmatrix}$$

in decimal, which is also correct.

Verilog code

Matrix Addition & Substraction

```
// Module Description:
   //
// The matrix_add_sub module performs matrix addition or
   subtraction operations          //
// based on the input control signal 'op' and produces the result
    in the signal 'ASP'.  //
// Note: Both Amatrixrownum and Bmatrixrownum and Amatrixcolnum
   and Bmatrixcolnum        //
// must be equal respectively, for valid operations.
                                          //

module matrix_add_sub #(parameter word_size = 32, // Word size
   for matrix elements //
                             parameter Amatrixrownum = 2, //
                                Number of rows in matrix A //
                             parameter Amatrixcolnum = 2, //
                                Number of columns in matrix A //
                             parameter Bmatrixrownum = 2, //
                                Number of rows in matrix B //
                             parameter Bmatrixcolnum = 2)( //
                                Number of columns in matrix B //
    input   wire        op,  // Control signal: 0 for addition, 1
        for subtraction //
    input   wire        [(Amatrixcolnum * Amatrixrownum) *
       word_size - 1 : 0]A, // Input matrix A //
    input   wire        [(Bmatrixcolnum * Bmatrixrownum) *
       word_size - 1 : 0]B, // Input matrix B //
    output  wire        [(Amatrixrownum *Bmatrixcolnum) *
       word_size - 1 : 0]ASP // Output matrix result //
);

// Declare wire signals to represent matrices A, B, and
   intermediate results //
    wire    [(word_size-1):0] Amatrix [0:Amatrixrownum - 1][0:
       Amatrixcolnum - 1];
    wire     [(word_size-1):0] Bmatrix [0:Bmatrixrownum - 1][0:
       Bmatrixcolnum - 1];
    wire    [(word_size-1):0] add_sub_wire[0:Amatrixrownum -
       1][0:Bmatrixcolnum - 1];
genvar x, y;
genvar i, j;
genvar Amatrixrowindex, Amatrixcolindex;
genvar Bmatrixrowindex, Bmatrixcolindex;
generate
    // Convert input vectors A and B to matrices Amatrix and
       Bmatrix //
    for (Amatrixrowindex = 0; Amatrixrowindex < Amatrixrownum;
       Amatrixrowindex = Amatrixrowindex + 1) begin
        for (Amatrixcolindex = 0; Amatrixcolindex < Amatrixcolnum
          ; Amatrixcolindex = Amatrixcolindex + 1) begin
               assign Amatrix[Amatrixrowindex][Amatrixcolindex]
                  = A[(Amatrixcolnum * Amatrixrownum) *
                  word_size - 1 - (Amatrixcolnum *
                  Amatrixrowindex + Amatrixcolindex) * word_size
                   : (Amatrixcolnum * Amatrixrownum) * word_size
                   -1 - (Amatrixcolnum * Amatrixrowindex +
                  Amatrixcolindex) * word_size  - (word_size-1)
                  ];
```

```
        end
    end
    for (Bmatrixrowindex = 0; Bmatrixrowindex < Bmatrixrownum;
       Bmatrixrowindex = Bmatrixrowindex + 1) begin
        for (Bmatrixcolindex = 0; Bmatrixcolindex < Bmatrixcolnum
           ; Bmatrixcolindex = Bmatrixcolindex + 1) begin
                assign Bmatrix[Bmatrixrowindex][Bmatrixcolindex]
                   = B[(Bmatrixcolnum * Bmatrixrownum) *
                   word_size - 1 - (Bmatrixcolnum *
                   Bmatrixrowindex + Bmatrixcolindex) * word_size
                    : (Bmatrixcolnum * Bmatrixrownum) * word_size
                    -1 - (Bmatrixcolnum * Bmatrixrowindex +
                   Bmatrixcolindex) * word_size  - (word_size-1)
                   ];
        end
    end
    // Perform matrix addition or subtraction on Amatrix and
       Bmatrix //
    for(x=0; x<Amatrixrownum; x=x+1) begin
        for(y=0; y<Amatrixcolnum; y=y+1) begin
            assign add_sub_wire[x][y] = (op == 1'b0) ? Amatrix[x
               ][y] + Bmatrix[x][y] : Amatrix[x][y] - Bmatrix[x][
               y];
        end
    end
     // Convert the addition or subtraction result to the output
        vector ASP //
    for(i=0; i<Amatrixrownum; i=i+1)begin
        for(j=0; j<Bmatrixcolnum; j=j+1)begin
                assign ASP[(Amatrixrownum*Bmatrixcolnum*word_size
                   -1)-(Bmatrixcolnum*i+j)*word_size:(
                   Amatrixrownum*Bmatrixcolnum*word_size-1)-(
                   Bmatrixcolnum*i+j) * word_size - (word_size-1)
                   ] = add_sub_wire[i][j];
        end
    end
endgenerate
endmodule
```

Matrix Multiplication

```
// Module Description:

   //
// The matrix_mul module performs matrix multiplication between
   two input matrices A and B  //
// and stores the result in the signal MP.
                                                             //
// Note: Both Amatrixrownum and Bmatrixcolnum must be equal, for
   valid operations.          //
                                              //

module matrix_mul #(parameter word_size = 32, // Word size for
   matrix elements //
                        parameter Amatrixrownum = 2, // Number of
                            rows in matrix A //
                        parameter Amatrixcolnum = 2, // Number of
                            columns in matrix A //
                        parameter Bmatrixrownum = 2, // Number of
                            rows in matrix B //
                        parameter Bmatrixcolnum = 1)( // Number
                           of columns in matrix B //
    input   wire        [(Amatrixcolnum * Amatrixrownum) *
       word_size - 1 : 0]A, // Input matrix A //
```

```
    input   wire        [(Bmatrixcolnum * Bmatrixrownum) *
       word_size - 1 : 0]B, // Input matrix B //
    output  wire        [(Amatrixrownum * Bmatrixcolnum) *
       word_size - 1 : 0]MP // Output matrix result //
);

// Declare wire signals to represent matrices A, B, and
   intermediate results //
    wire    [(word_size-1):0] Amatrix [0:Amatrixrownum - 1][0:
       Amatrixcolnum - 1];
    wire    [(word_size-1):0] Bmatrix [0:Bmatrixrownum - 1][0:
       Bmatrixcolnum - 1];
    wire    [(word_size-1):0] matrixproduct_wire[0:Amatrixrownum
       - 1][0:Bmatrixcolnum - 1];
genvar i, j;
genvar Amatrixrowindex, Amatrixcolindex;
genvar Bmatrixrowindex, Bmatrixcolindex;
genvar x, y, z, w;
generate
    // Convert input vectors A and B to matrices Amatrix and
       Bmatrix //
    for (Amatrixrowindex = 0; Amatrixrowindex < Amatrixrownum;
       Amatrixrowindex = Amatrixrowindex + 1) begin
        for (Amatrixcolindex = 0; Amatrixcolindex < Amatrixcolnum
           ; Amatrixcolindex = Amatrixcolindex + 1) begin
                assign Amatrix[Amatrixrowindex][Amatrixcolindex]
                   = A[(Amatrixcolnum * Amatrixrownum) *
                   word_size - 1 - (Amatrixcolnum *
                   Amatrixrowindex + Amatrixcolindex) * word_size
                    : (Amatrixcolnum * Amatrixrownum) * word_size
                    -1 - (Amatrixcolnum * Amatrixrowindex +
                   Amatrixcolindex) * word_size  - (word_size-1)
                   ];
        end
    end
    for (Bmatrixrowindex = 0; Bmatrixrowindex < Bmatrixrownum;
       Bmatrixrowindex = Bmatrixrowindex + 1) begin
        for (Bmatrixcolindex = 0; Bmatrixcolindex < Bmatrixcolnum
           ; Bmatrixcolindex = Bmatrixcolindex + 1) begin
                assign Bmatrix[Bmatrixrowindex][Bmatrixcolindex]
                   = B[(Bmatrixcolnum * Bmatrixrownum) *
                   word_size - 1 - (Bmatrixcolnum *
                   Bmatrixrowindex + Bmatrixcolindex) * word_size
                    : (Bmatrixcolnum * Bmatrixrownum) * word_size
                    -1 - (Bmatrixcolnum * Bmatrixrowindex +
                   Bmatrixcolindex) * word_size  - (word_size-1)
                   ];
        end
    end
     // Perform matrix multiplication //
    reg     [word_size-1 : 0] temp_res0 [Amatrixcolnum-1:0][
       Amatrixrownum*Bmatrixcolnum-1:0];
    wire    [word_size-1 : 0] temp_res1 [Amatrixcolnum-1:0][
       Amatrixrownum*Bmatrixcolnum-1:0];
    for(x=0; x<Amatrixrownum; x=x+1) begin
        for(w=0; w<Bmatrixcolnum; w=w+1)begin
            for(y=0; y<Amatrixcolnum; y=y+1) begin
                wire    [word_size-1 : 0] temp_res0_w;
                assign temp_res0_w = Amatrix[x][y] * Bmatrix[y][w
                   ];
                always@(temp_res0_w) begin
                    temp_res0[y][Bmatrixcolnum*x+w] = temp_res0_w
                       ;
                end
```

```
            end
            for(z=0; z<Amatrixcolnum-1; z=z+1)begin
                if (z == 0) begin
                    assign temp_res1[0][Bmatrixcolnum*x+w] =
                       temp_res0[0][Bmatrixcolnum*x+w] +
                       temp_res0[1][Bmatrixcolnum*x+w];
                end else begin
                    assign temp_res1[z][Bmatrixcolnum*x+w] =
                       temp_res1[z-1][Bmatrixcolnum*x+w] +
                       temp_res0[z+1][Bmatrixcolnum*x+w];
                end
            end
            assign matrixproduct_wire[x][w] = temp_res1[
               Amatrixcolnum-2][Bmatrixcolnum*x+w];
        end
    end
    // conversion of matrixproduct to output vector MP //
    for(i=0; i<Amatrixrownum; i=i+1)begin
        for(j=0; j<Bmatrixcolnum; j=j+1)begin
                assign MP[(Amatrixrownum*Bmatrixcolnum*word_size
                   -1)-(Bmatrixcolnum*i+j)*word_size:(
                   Amatrixrownum*Bmatrixcolnum*word_size-1)-(
                   Bmatrixcolnum*i+j)*word_size - (word_size-1)]
                   = matrixproduct_wire[i][j];
        end
    end
endgenerate
endmodule
```

Matrix Kronecker product

```
//  Module Description:

   //
// The matrix_kronecker module performs the Kronecker product
   between two input matrices A and B  //
// and stores the result in the signal TP.
                                                              //
// TP formula

   //
// A = | a00 a01 | B = | b00 b01 | TP = | a00 * b00 a00 * b01 a01
    * b00 a01 * b01 |               //
//     | a10 a11 |     | b10 b11 |      | a00 * b10 a00 * b11 a01
    * b10 a01 * b11 |               //
//                                      | a10 * b00 a10 * b01 a11
    * b00 a11 * b01 |               //
//                                      | a10 * b10 a10 * b11 a11
    * b10 a11 * b11 |               //

module matrix_kronecker #(parameter word_size = 32, // Word size
   for matrix elements //
                        parameter Amatrixrownum = 2, // Number of
                            rows in matrix A //
                        parameter Amatrixcolnum = 2, // Number of
                            columns in matrix A //
                        parameter Bmatrixrownum = 2,  // Number
                           of rows in matrix B //
                        parameter Bmatrixcolnum = 2)( // Number
                           of columns in matrix B //
    input   wire [(Amatrixcolnum * Amatrixrownum) * word_size - 1
        : 0]A, // Input matrix A //
    input   wire [(Bmatrixcolnum * Bmatrixrownum) * word_size - 1
        : 0]B, // Input matrix B //
```

```
    output  wire [(Amatrixcolnum * Amatrixrownum) * (
       Bmatrixcolnum * Bmatrixrownum) * word_size - 1 : 0]TP //
       Output matrix result //
);

// Declare wire signals to represent matrices A, B, and
   intermediate results //
    wire    [(word_size-1):0] Amatrix [0:Amatrixrownum - 1][0:
       Amatrixcolnum - 1];
    wire    [(word_size-1):0] Bmatrix [0:Bmatrixrownum - 1][0:
       Bmatrixcolnum - 1];
    wire    [(word_size-1):0] tensorproduct[Amatrixrownum *
       Bmatrixrownum - 1:0][ Bmatrixcolnum * Amatrixcolnum -
       1:0];

genvar x, y, z, w;
genvar i, j;
genvar Amatrixrowindex, Amatrixcolindex;
genvar Bmatrixrowindex, Bmatrixcolindex;
generate
    // Convert input vectors A and B to matrices Amatrix and
       Bmatrix //
    for (Amatrixrowindex = 0; Amatrixrowindex < Amatrixrownum;
       Amatrixrowindex = Amatrixrowindex + 1) begin
        for (Amatrixcolindex = 0; Amatrixcolindex < Amatrixcolnum
           ; Amatrixcolindex = Amatrixcolindex + 1) begin
                assign Amatrix[Amatrixrowindex][Amatrixcolindex]
                   = A[(Amatrixcolnum * Amatrixrownum) *
                   word_size - 1 - (Amatrixcolnum *
                   Amatrixrowindex + Amatrixcolindex) * word_size
                    : (Amatrixcolnum * Amatrixrownum) * word_size
                    -1 - (Amatrixcolnum * Amatrixrowindex +
                   Amatrixcolindex) * word_size  - (word_size-1)
                   ];
        end
    end
    for (Bmatrixrowindex = 0; Bmatrixrowindex < Bmatrixrownum;
       Bmatrixrowindex = Bmatrixrowindex + 1) begin
        for (Bmatrixcolindex = 0; Bmatrixcolindex < Bmatrixcolnum
           ; Bmatrixcolindex = Bmatrixcolindex + 1) begin
                assign Bmatrix[Bmatrixrowindex][Bmatrixcolindex]
                   = B[(Bmatrixcolnum * Bmatrixrownum) *
                   word_size - 1 - (Bmatrixcolnum *
                   Bmatrixrowindex + Bmatrixcolindex) * word_size
                    : (Bmatrixcolnum * Bmatrixrownum) * word_size
                    -1 - (Bmatrixcolnum * Bmatrixrowindex +
                   Bmatrixcolindex) * word_size  - (word_size-1)
                   ];
        end
    end
    // Calculate the Kronecker product //
    for(x=0; x<Amatrixrownum; x=x+1) begin
        for(y=0; y<Amatrixcolnum; y=y+1) begin
            for(z=0; z<Bmatrixrownum; z=z+1) begin
                    for(w=0; w<Bmatrixcolnum; w=w+1)begin
                    // A is an m x n matrix
                                                            //
                    // B is a p x q matrix
                                                             //
                    // A ? B is an (m * p) x (n * q) matrix
                                            //
                    // (A ? B)(i * p + k, j * q + l) = A(i, j) *
                       B(k, l) //
                    assign tensorproduct[x*Bmatrixrownum+z][y*
```

```
                        Bmatrixcolnum+w] = Amatrix[x][y] * Bmatrix
                        [z][w];
                    end
                end
            end
        end
        // Convert tensorproduct from a matrix to a vector //
        for(i=0; i<Amatrixrownum*Amatrixcolnum; i=i+1)begin
            for(j=0; j<Bmatrixrownum*Bmatrixcolnum; j=j+1)begin
                assign TP[(Amatrixcolnum * Bmatrixcolnum *
                   Amatrixcolnum * Bmatrixcolnum * word_size - 1) -
                   (((Amatrixcolnum * Bmatrixcolnum) * i) + j) *
                   word_size : (Amatrixcolnum * Bmatrixcolnum *
                   Amatrixcolnum * Bmatrixcolnum * word_size - 1) -
                   (((Amatrixcolnum * Bmatrixcolnum) * i) + j) *
                   word_size  - (word_size-1)] = tensorproduct[i][j];
            end
        end
endgenerate
endmodule
```

Matrix ALU

```
// Module Description:
                                                                //
// The Matrix ALU (Arithmetic Logic Unit) performs various matrix
     operations       //
// based on the specified op code and produces the result in the
   signal C.        //
// Note: Both Amatrixrownum and Bmatrixrownum and Amatrixcolnum
   and Bmatrixcolnum //
// must be equal respectively, for valid operations.
                                    //

module matrix_alu #(parameter word_size = 32, // Word size for
   matrix elements //
                    parameter Amatrixrownum = 2, // Number of
                       rows in matrix A //
                    parameter Amatrixcolnum = 2, // Number of
                       columns in matrix A //
                    parameter Bmatrixrownum = 2, // Number of
                       rows in matrix B //
                    parameter Bmatrixcolnum = 2)( // Number of
                       columns in matrix B //
input   wire        clk, // Clock input //
input   wire        resetn, // Reset signal (active low) //
input   wire        [(Amatrixcolnum * Amatrixrownum) * word_size
   - 1 : 0]A, // Input matrix A //
input   wire        [(Bmatrixcolnum * Bmatrixrownum) * word_size
   - 1 : 0]B, // Input matrix B //
input   wire        [1:0] op,  // Operation code //
output  wire        [(Amatrixcolnum * Amatrixrownum) * (
   Bmatrixcolnum * Bmatrixrownum) * word_size - 1 : 0] C //
   Output matrix result //
);
// Intermediate wire signals for different matrix operations //
wire    [(Amatrixrownum * Bmatrixcolnum) * word_size - 1 : 0]
   m_add, m_sub, m_mul;
wire    [(Amatrixcolnum * Amatrixrownum) * (Bmatrixcolnum *
   Bmatrixrownum) * word_size - 1 : 0] m_kro;

// Register for storing the final output //
reg     [(Amatrixcolnum * Amatrixrownum) * (Bmatrixcolnum *
   Bmatrixrownum) * word_size - 1 : 0] C_reg;
```

```
// Always block to handle output register assignment and reset //
always@(posedge clk or negedge resetn)begin
    if(!resetn)begin
        C_reg = 'd0;
    end else begin
        if(op==2'b00) begin
            C_reg = {{(((Amatrixcolnum * Amatrixrownum) * (
               Bmatrixcolnum * Bmatrixrownum) - 1)* word_size -
               1){1'bz}},m_add};
        end else if (op==2'b01) begin
            C_reg = {{(((Amatrixcolnum * Amatrixrownum) * (
               Bmatrixcolnum * Bmatrixrownum) - 1)* word_size -
               1){1'bz}},m_sub};
        end else if (op==2'b10) begin
            C_reg = {{(((Amatrixcolnum * Amatrixrownum) * (
               Bmatrixcolnum * Bmatrixrownum) - 1)* word_size -
               1){1'bz}},m_mul};
        end else if (op==2'b11) begin
            C_reg = m_kro;
        end
    end
end

// Assign the final output //
assign C = C_reg;

// Module instantiations for matrix operations (addition,
   subtraction, multiplication, and Kronecker product) //
// Instantiate these modules as needed to perform specific
   operations.                                        //

matrix_add_sub #(.word_size(word_size),
                 .Amatrixrownum(Amatrixrownum),
                 .Amatrixcolnum(Amatrixcolnum),
                 .Bmatrixrownum(Bmatrixrownum),
                 .Bmatrixcolnum(Bmatrixcolnum))mat_add(
.op(1'b0),
.A(A),
.B(B),
.ASP(m_add)
);
matrix_add_sub #(.word_size(word_size),
                 .Amatrixrownum(Amatrixrownum),
                 .Amatrixcolnum(Amatrixcolnum),
                 .Bmatrixrownum(Bmatrixrownum),
                 .Bmatrixcolnum(Bmatrixcolnum))mat_sub(
.op(1'b1),
.A(A),
.B(B),
.ASP(m_sub)
);
matrix_mul #(.word_size(word_size),
                 .Amatrixrownum(Amatrixrownum),
                 .Amatrixcolnum(Amatrixcolnum),
                 .Bmatrixrownum(Bmatrixrownum),
                 .Bmatrixcolnum(Bmatrixcolnum))mat_mul(
.A(A),
.B(B),
.MP(m_mul)
);
matrix_kronecker #(.word_size(word_size),
                 .Amatrixrownum(Amatrixrownum),
                 .Amatrixcolnum(Amatrixcolnum),
```

```
                  .Bmatrixrownum(Bmatrixrownum),
                  .Bmatrixcolnum(Bmatrixcolnum))mat_kron(
.A(A),
.B(B),
.TP(m_kro)
);
endmodule
```

Matrix ALU Testbench

```
// testbench (matrix_alu_tb) for testing a matrix arithmetic
   logic unit (Matrix ALU) //
// Initialize parameters, clock and reset signals, input matrices
    A and B, and the   //
// operation code (op), and then connects these signals to the
   Matrix ALU module.    //
// It toggles the clock signal, simulates a reset sequence, and
   then performs a      //
// sequence of operations on matrices A and B. The output result
   is captured in the  //
// signal C

   //

module matrix_alu_tb();

// Parameter Definitions //
parameter word_size = 8; // Word size for matrix elements //
parameter Amatrixrownum = 2;
parameter Amatrixcolnum = 2;
parameter Bmatrixrownum = 2;
parameter Bmatrixcolnum = 2;

// Signal Declarations //
reg     clk, resetn;
reg     [(Amatrixcolnum * Amatrixrownum) * word_size - 1 : 0] A,
   B;
reg     [1:0] op;
wire    [(Amatrixcolnum * Amatrixrownum) * (Bmatrixcolnum *
   Bmatrixrownum) * word_size - 1 : 0] C;

// Clock Generation //
always #10 clk = ~clk;
initial begin
clk = 1'b0;
resetn = 1'b1;
#20
resetn = 1'b0;
#20
resetn = 1'b1;
end

//Initial Block for Matrix and Operation Initialization //
initial begin
A={8'h01,8'h02,8'h03,8'h00};
B={8'h05,8'h06,8'h07,8'h08};

op = 2'b00;
#100
op = 2'b01;
#100
op = 2'b10;
#100
op = 2'b11;
```

```
end

// Module Instantiation //
matrix_alu #(.word_size(word_size),
             .Amatrixrownum(Amatrixrownum),
             .Amatrixcolnum(Amatrixcolnum),
             .Bmatrixrownum(Bmatrixrownum),
             .Bmatrixcolnum(Bmatrixcolnum)) alu (
.clk(clk),
.resetn(resetn),
.A(A),
.B(B),
.op(op),
.C(C)
);
endmodule
```