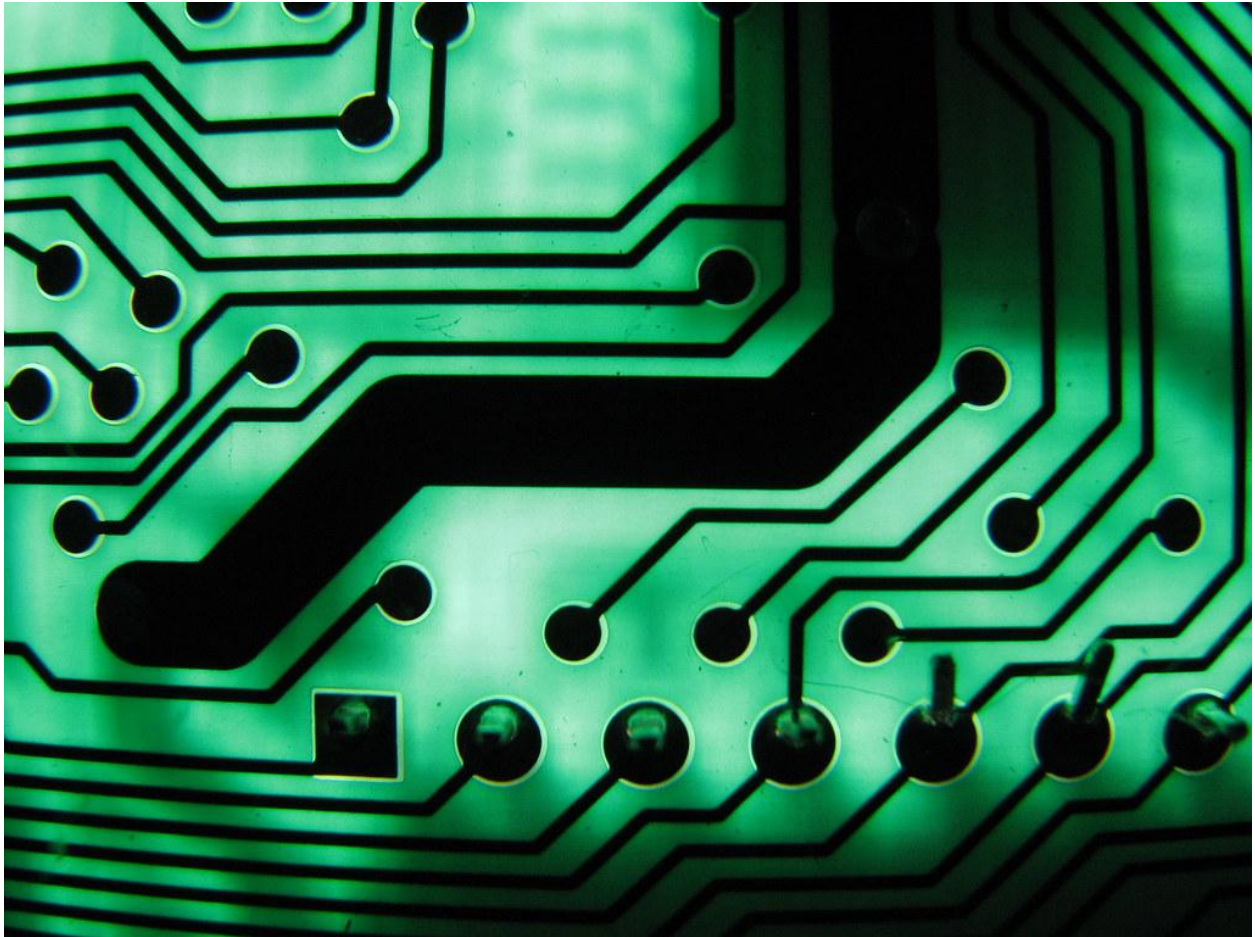


Lab #6

Matrix Multiplication in Hardware



Introduction

The purpose of this lab is to program an FPGA to make hardware matrix multiplications.

Results

1 Task 1 Implement a Multiply Accumulator (MACC)

Part 1.1 Write a synthesizable model for a MACC.

```
`timescale 1ns/10ps
module macc(
    input signed [7:0]inA,
    input signed [7:0]inB,
    input macc_clear,
    input clk,
    output reg signed [18:0]out);

    reg [18:0] s;

    always @(*) begin
        if (macc_clear == 1'b1) begin
            s = inA * inB;
        end
        else begin
            s = out + (inA * inB);
        end
    end
    always @(posedge clk) begin
        out <= #1 s;
    end
endmodule
```

Figure 1.1.1 MACC Model.

Part 1.2 Write a testbench to verify the operation of the MACC circuit

```
`timescale 1ns/10ps
module tb_macc();

    reg signed [7:0]X;

    reg signed [7:0]Y;

    reg macc_clear;

    reg clk;

    wire signed [18:0]result;
```

```

macc mc(X, Y, macc_clear, clk, result);
initial begin
    clk = 1'b1;
    repeat (400) begin
        #10 clk = ~clk;
    end
end

initial begin
    repeat (2) begin
        X = 127;
        Y = 127;
        macc_clear = 1'b1;
        #40
        $display("inputs: X = %d, Y = %d, macc_clear = %b, Output: %d",X,
        Y, macc_clear, result);
        #40
        X = -128;
        Y = -128;
        macc_clear = 1'b0;
        #40
        $display("inputs: X = %d, Y = %d, macc_clear = %b, Output: %d",X,
        Y, macc_clear, result);
        X = 0;
        Y = 127;
        macc_clear = 1'b0;
        #40
        $display("inputs: X = %d, Y = %d, macc_clear = %b, Output: %d",X,
        Y, macc_clear, result);
        X = 127;
        Y = 0;
        macc_clear = 1'b0;
        #40
        $display("inputs: X = %d, Y = %d, macc_clear = %b, Output: %d",X,
        Y, macc_clear, result);
        X = 8;
        Y = 8;
        macc_clear = 1'b0;
        #40
        $display("inputs: X = %d, Y = %d, macc_clear = %b, Output: %d",X,
        Y, macc_clear, result);
    end
end
endmodule

```

Figure 1.2.1 Testbench code to verify the operation of the MACC circuit.

Part 1.3 Functional Simulation of MACC

```
# inputs: X = 127, Y = 127, macc_clear = 1, Output: 16129
# inputs: X = -128, Y = -128, macc_clear = 0, Output: 32513
# inputs: X = 0, Y = 127, macc_clear = 0, Output: 48897
# inputs: X = 127, Y = 0, macc_clear = 0, Output: 48897
# inputs: X = 8, Y = 8, macc_clear = 0, Output: 48961
# inputs: X = 127, Y = 127, macc_clear = 1, Output: 16129
# inputs: X = -128, Y = -128, macc_clear = 0, Output: 32513
# inputs: X = 0, Y = 127, macc_clear = 0, Output: 48897
# inputs: X = 127, Y = 0, macc_clear = 0, Output: 48897
# inputs: X = 8, Y = 8, macc_clear = 0, Output: 48961
```

Figure 1.3.1 Functional simulation for the circuit.

Part 1.4 Estimate the resource usage – the number of flip-flops, logic elements, memory blocks, embedded multipliers, etc.

Analysis & Synthesis Resource Usage Summary		
<<Filter>>		
	Resource	Usage
1	Estimated Total logic elements	20
2		
3	Total combinational functions	20
4	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	0
2	-- 3 input functions	17
3	-- <=2 input functions	3
5		
6	▼ Logic elements by mode	
1	-- normal mode	2
2	-- arithmetic mode	18
7		
8	▼ Total registers	19
1	-- Dedicated logic registers	19
2	-- I/O registers	0
9		
10	I/O pins	149
11		
12	Embedded Multiplier 9-bit elements	1

Figure 1.4.1 Quartus Analysis & Synthesis Resource Usage Summary for MACC

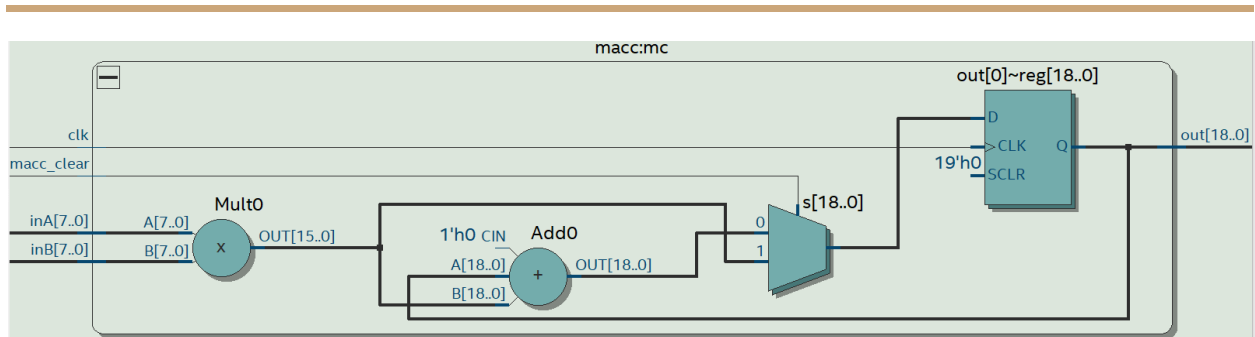


Figure 1.4.2 Quartus diagram for MACC for manual estimation of components

2 Task 2 Matrix Multiply with One MAC

Part 2.1 Matrix multiplication of two 8x8 matrices with a MACC.

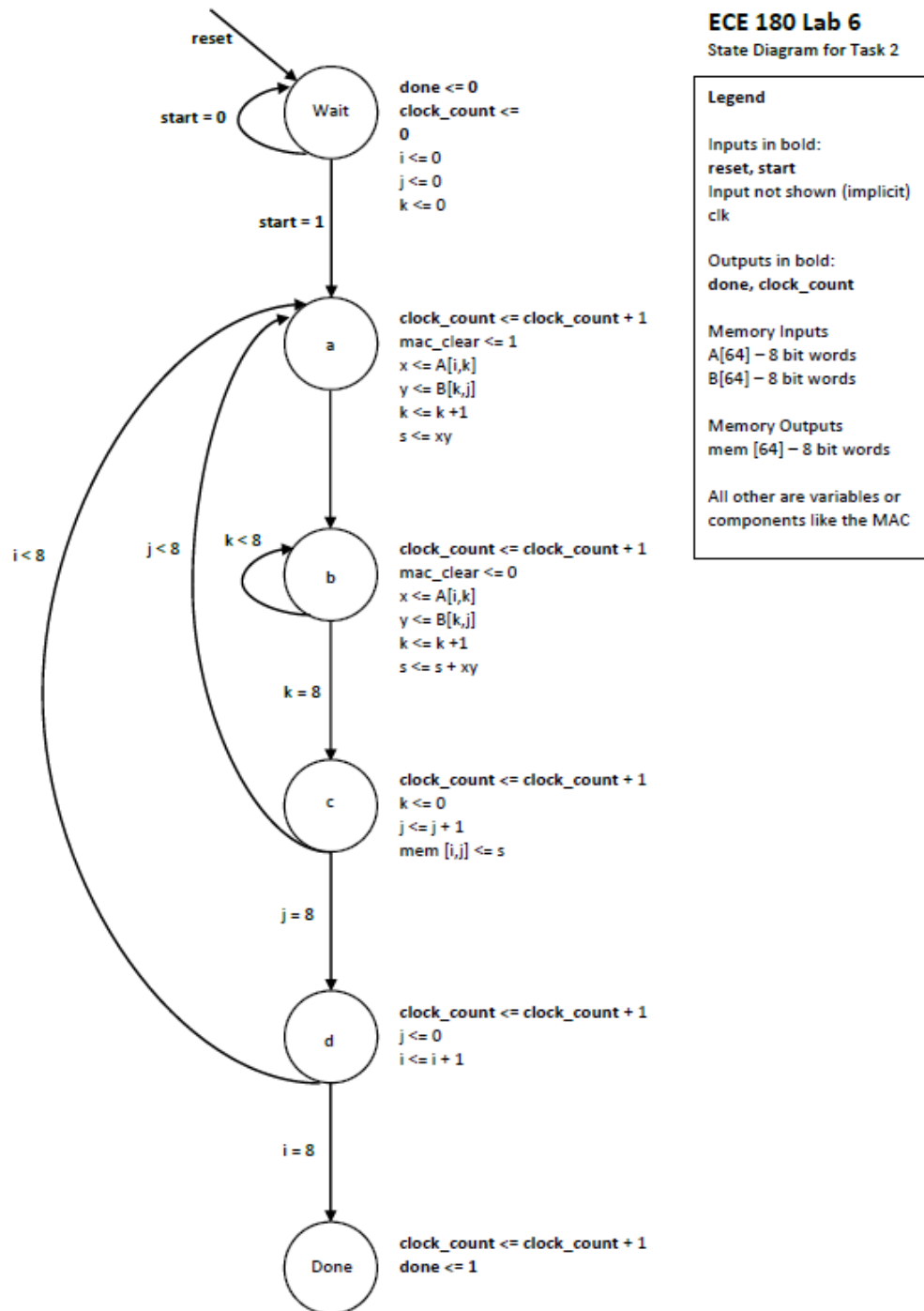


Figure 2.1.1 State diagram for design controller as presented in the pre-lab

```

`timescale 1ns/10ps
module matrix_mult_1mac
(
    input  clk,
    input  start,
    input  reset,
    output reg done,
    output reg [10:0]clock_count
);

    parameter [2:0]WAIT    = 3'b000;
parameter [2:0]A_STATE = 3'b001;
parameter [2:0]B_STATE = 3'b011;
parameter [2:0]C_STATE = 3'b111;
parameter [2:0]D_STATE = 3'b110;
parameter [2:0]DONE    = 3'b100;
parameter N = 8;

reg [2:0]state_c;
    reg [2:0]state;

integer i,j,k;
reg [3:0]i_c,j_c,k_c;

reg signed [7:0]X;
reg signed [7:0]Y;

reg macc_clear;
wire signed [18:0] result;

// Matrices
reg signed [7:0] inA [63:0];
reg signed [7:0] inB [63:0];
// provided testbench uses this for the result matrix
matrixC RAMOUTPUT();

macc mc(X, Y, macc_clear, clk, result);

// Initialize Matrices
initial begin
    $readmemb("ram_a_init.txt",inA);
    $readmemb("ram_b_init.txt",inB);

    // Print Input Matrices
    $display("\nMatrix A");
    for(i=0;i<8;i=i+1) begin

$display(inA[i],inA[i+8],inA[i+16],inA[i+24],inA[i+32],inA[i+40],inA[i+48],in

```

```

A[i+56]);
    end

    $display("\n Matrix B");
    for(i=0;i<8;i=i+1) begin

$display(inB[i],inB[i+8],inB[i+16],inB[i+24],inB[i+32],inB[i+40],inB[i+48],in
B[i+56]);
        end

end

always @(*) begin
    // default same state
    state_c = state;
    $display("State = %b" ,state);

    i_c = i;
    j_c = j;
    k_c = k;

    case (state)
        WAIT: begin
            // set vars
            clock_count = 1;
            macc_clear = 1'b0;
            i_c = 0;
            j_c = 0;
            k_c = 0;
            done = 0;
            begin
                if (start == 1'b1) begin
                    state_c = A_STATE;
                end
            end
        end
        A_STATE: begin
            // access arrays by column-major order
            X = inA[(k*N)+i];
            Y = inB[(j*N)+k];

            $display("\n X[%d,%d] = %d", i, j, X);
            $display("\n Y[%d,%d] = %d", i, j, Y);
            macc_clear = 1'b1; // we are starting a new cume
            clock_count = clock_count +1; // increment cycle count
            k_c = k + 1;

            state_c = B_STATE;

```

```

end
B_STATE: begin
    macc_clear = 1'b0; // accumulate
    X = inA[(k*N)+i];
    Y = inB[(j*N)+k];

    $display("\n X[%d,%d] = %d", i, k, X);
    $display("\n Y[%d,%d] = %d", k, j, Y);

    clock_count = clock_count + 1; // increment cycle count

    // verify where we go next
    if (k + 1 < N) begin
        k_c = k + 1;

        state_c = B_STATE; // keep adding and accumulating
    end
    else begin
        state_c = C_STATE;
    end
end
end
C_STATE: begin
    $display("\n Result[%d,%d] = %d", i, j, result);
    // save this result to memory
    RAMOUTPUT.mem[(j*N)+i] = result;

    // adjust vars
    clock_count = clock_count + 1; // increment cycle count

    // verify where we go next
    if (j + 1 < N)begin
        k_c = 0;
        j_c = j + 1;
        state_c = A_STATE;
    end
    else begin
        state_c = D_STATE;
    end
end
end
D_STATE: begin
    // adjust vars
    clock_count = clock_count + 1; // increment cycle count

    // verify where we go next
    if (i + 1 < N)begin
        i_c = i + 1;
        j_c = 0;

```

```

        k_c = 0;
        state_c = A_STATE;
    end
    else begin
        state_c = DONE;
    end
end
DONE: begin
    state_c = DONE;
    done = 0'b1;

    end
default: begin
    state_c = WAIT;
end
endcase
if (reset == 1'b1) begin
    state_c = WAIT;
end
end

always @(posedge clk or posedge start) begin
    state <= #1 state_c; // reset works even if disabled
    k <= k_c;
    j <= j_c;
    i <= i_c;
end

endmodule

```

Figure 2.1.2 Verilog implementation of the circuit using one MACC

Part 2.2 Testbench for matrix multiplication of two 8x8 matrices with a MACC.

```

`timescale 1ns/10ps
// Test bench module
module tb_lab6;

// Input Array
/////////////////////////////////////////////////////////////////
//                               Test Bench Signals                               //
/////////////////////////////////////////////////////////////////
reg clk;
integer i,j,k;

// Matrices

```

```

reg signed [7:0] matrixA [63:0];
reg signed [7:0] matrixB [63:0];
reg signed [18:0] matrixC [63:0];

// Comparison Flag
reg comparison;

////////////////////////////////////////
//                      I/O Declarations                      //
////////////////////////////////////////
// declare variables to hold signals going into submodule
reg start;
reg reset;

// Misc "wires"
wire done;
wire [10:0] clock_count;

////////////////////////////////////////
//                      Submodule Instantiation                //
////////////////////////////////////////

matrix_mult_1mac DUT
(
    .clk    (clk),
    .start  (start),
    .reset  (reset),
    .done   (done),
    .clock_count (clock_count)
);

initial begin
    //*****
    // CHANGE .TXT FILE NAMES TO MATCH THE ONES USED IN
    // YOUR MEMORY MODULES

    // Initialize Matrices
    $readmemb("ram_a_init.txt",matrixA);
    $readmemb("ram_b_init.txt",matrixB);

    //*****

    //////////////////////////////////////////
    //                      Perform Test                      //
    //////////////////////////////////////////
    reset <= 1'b1;
    start <= 1'b0;
    clk <= 1'b0;

```

```

repeat(2) @(posedge clk);
reset <= 1'b0;
repeat(2) @(posedge clk);
start <= 1'b1;
repeat(1) @(posedge clk);
start <= 1'b0;

// -----
// Wait for done or timeout
fork : wait_or_timeout
begin
    repeat(1000) @(posedge clk);
    disable wait_or_timeout;
end
begin
    @(posedge done);
    disable wait_or_timeout;
end
join
// End Timeout Routing
//-----

////////////////////////////////////
//          Verify Computation          //
////////////////////////////////////

// Print Input Matrices
$display("Matrix A");
for(i=0;i<8;i=i+1) begin

$display(matrixA[i],matrixA[i+8],matrixA[i+16],matrixA[i+24],matrixA[i+32],ma
trixA[i+40],matrixA[i+48],matrixA[i+56]);
end

$display("\n Matrix B");
for(i=0;i<8;i=i+1) begin

$display(matrixB[i],matrixB[i+8],matrixB[i+16],matrixB[i+24],matrixB[i+32],ma
trixB[i+40],matrixB[i+48],matrixB[i+56]);
end

// Generate Expected Result
for(i=0;i<8;i=i+1) begin
    for(j=0;j<8;j=j+1) begin
        matrixC[8*i+j] = 0;
        for(k=0;k<8;k=k+1) begin
            matrixC[8*i+j] = matrixC[8*i+j] + matrixA[j+8*k]*matrixB[k+8*i];
        end
    end
end

```

```

        end
    end

    // Display Expected Result
    $display("\nExpected Result");
    for(i=0;i<8;i=i+1) begin

$display(matrixC[i],matrixC[i+8],matrixC[i+16],matrixC[i+24],matrixC[i+32],ma
trixC[i+40],matrixC[i+48],matrixC[i+56]);
        end

    // Display Output Matrix
    $display("\nGenerated Result");
    for(i=0;i<8;i=i+1) begin

$display(DUT.RAMOUTPUT.mem[i],DUT.RAMOUTPUT.mem[i+8],DUT.RAMOUTPUT.mem[i+16],
DUT.RAMOUTPUT.mem[i+24],DUT.RAMOUTPUT.mem[i+32],DUT.RAMOUTPUT.mem[i+40],DUT.R
AMOUTPUT.mem[i+48],DUT.RAMOUTPUT.mem[i+56]);
        end

    // Test if the two matrices match
    comparison = 1'b0;
    for(i=0;i<8;i=i+1) begin
        for(j=0;j<8;j=j+1) begin
            if (matrixC[8*i+j] != DUT.RAMOUTPUT.mem[8*i+j]) begin
                $display("Mismatch at indices [%1.1d,%1.1d]",j,i);
                comparison = 1'b1;
            end
        end
    end
    if (comparison == 1'b0) begin
        $display("\nsuccess :)");
    end

    $display("Running Time = %d clock cycles",clock_count);

    $stop; // End Simulation
end

// Clock
always begin
    #10;           // wait for initial block to initialize clock
    clk = ~clk;
end
endmodule

```

Figure 2.2.1 Testbench for the circuit using one MACC

```

# Matrix A
# -12 -14 -71 63 122 53 44 -70
# -103-100 -82 46-116 -33 15-113
# 124 7 81 62 -20 -1-117 -87
# 59 8 69 105 -35-126-106-124
# 124-123 -16-118 96 95 -63 -14
# -66 25-107 32 85 127 61 114
# 111 10 61 -57 107 -98 28 54
# -106 63 91 -71-123 -9 0 48
#
# Matrix B
# -114 19 -62 80 -24 -28 108 -39
# -100 -35-121-104-118 103-114 13
# 114 -93 -17 2 65-106 -66 -4
# 81-110 110-101 8 5 -80-110
# 84 115-113 -21 75 -45 -3 124
# -44 16 30 -64 67 -59 40 -32
# 91 -76 -3-127 109 -50 18 39
# -62 -98-107 107 -69 73 -61 94
#
# Expected Result
# 16037 18329 5737 -25041 20156 -9192 6762 2208
# 16199 175 49104 -12098 7831 -326 8199 -25380
# -7469 2860 8798 9126 -6358 -10569 5509 -27078
# 9491 -2959 19512 2414 -11100 -2360 -6773 -30121
# -14199 39849 -11824 33005 17218 -22832 41142 11700
# -4547 301 -11770 -16788 5602 9879 -6507 19378
# 1183 5673 -36292 19966 450 -5320 4351 24399
# -2505 -23865 -1915 616 -12683 9026 -21875 1947
#
# Generated Result
# 16037 18329 5737 -25041 20156 -9192 6762 2208
# 16199 175 49104 -12098 7831 -326 8199 -25380
# -7469 2860 8798 9126 -6358 -10569 5509 -27078
# 9491 -2959 19512 2414 -11100 -2360 -6773 -30121
# -14199 39849 -11824 33005 17218 -22832 41142 11700
# -4547 301 -11770 -16788 5602 9879 -6507 19378
# 1183 5673 -36292 19966 450 -5320 4351 24399
# -2505 -23865 -1915 616 -12683 9026 -21875 1947
#
# success :)
# Running Time = 1088 clock cycles

```

Figure 2.2.2 Testbench results for the circuit using one MACC

Part 2.3 Estimate the resource usage


Analysis & Synthesis Resource Usage Summary		
 <<Filter>>		
	Resource	Usage
1	Estimated Total logic elements	73
2		
3	Total combinational functions	73
4	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	24
2	-- 3 input functions	23
3	-- <=2 input functions	26
5		
6	▼ Logic elements by mode	
1	-- normal mode	63
2	-- arithmetic mode	10
7		
8	▼ Total registers	0
1	-- Dedicated logic registers	0
2	-- I/O registers	0
9		
10	I/O pins	149
11		
12	Embedded Multiplier 9-bit elements	0

Figure 2.3.1 Estimated resource usage for the one MACC multiplier circuit.

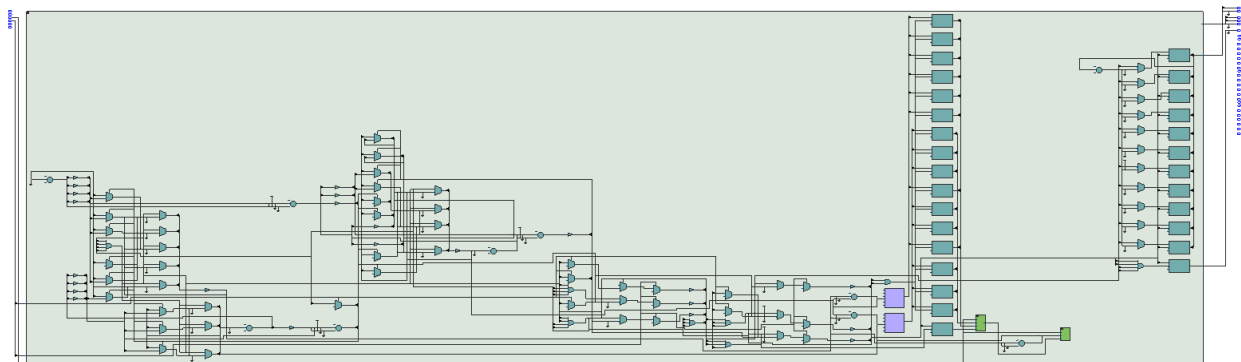


Figure 2.3.2 Diagram for estimating components manually.

Part 2.4 Performance estimation

Using the number of clock cycles we measured to calculate the matrix multiplication of 1088 clock cycles at a frequency of 50 MHz, the circuit does the calculation in 0.02176s. The bottlenecks that we have in this circuit are:

- a. Reading matrix elements from memory one by one.
- b. Being able to calculate one result at a time
- c. Writing one result at a time to the result matrix

3 Task 3 Matrix Multiply with two MACCs

Part 3.1 Matrix multiplication of two 8x8 matrices with two MACCs and matrix A read from dual-ported memory

```
`timescale 1ns/10ps
module matrix_mult_2mac
(
    input  clk,
    input  start,
    input  reset,
    output reg done,
    output reg [10:0]clock_count
);

    parameter [2:0]WAIT    = 3'b000;
    parameter [2:0]A_STATE = 3'b001;
    parameter [2:0]B_STATE = 3'b011;
    parameter [2:0]C_STATE = 3'b111;
    parameter [2:0]D_STATE = 3'b110;
    parameter [2:0]DONE    = 3'b100;
    parameter N = 8;

    reg [2:0]state_c;
    reg [2:0]state;

    integer i,j,k;
    reg [3:0]i_c,j_c,k_c;

    integer i1,j1,k1;
    integer i2,j2,k2;

    reg signed [7:0]X;
    reg signed [7:0]Y;
```

```

reg signed [18:0] buffer1;
reg signed [18:0] buffer2;

reg macc_clear;
wire signed [18:0] result;

reg signed [7:0]X2;
//reg signed [7:0]Y2;

reg macc_clear2;
wire signed [18:0] result2;

reg writebuffer1, writebuffer2;

// Matrices
reg signed [7:0] inA [63:0];
reg signed [7:0] inB [63:0];
// provided testbench uses this for the result matrix
matrixC RAMOUTPUT();

macc mc1(X, Y, macc_clear, clk, result);
macc mc2(X2, Y, macc_clear2, clk, result2);

// Initialize Matrices
initial begin
    $readmemb("ram_a_init.txt",inA);
    $readmemb("ram_b_init.txt",inB);

    // Print Input Matrices
    $display("\nMatrix A");
    for(i=0;i<8;i=i+1) begin

$display(inA[i],inA[i+8],inA[i+16],inA[i+24],inA[i+32],inA[i+40],inA[i+48],in
A[i+56]);
        end

    $display("\n Matrix B");
    for(i=0;i<8;i=i+1) begin

$display(inB[i],inB[i+8],inB[i+16],inB[i+24],inB[i+32],inB[i+40],inB[i+48],in
B[i+56]);
        end

    end

always @(start or state or reset or k or i or j) begin

```

```

// default same state
state_c = state;
$display("State = %b" ,state);

i_c = i;
j_c = j;
k_c = k;

case (state)
    WAIT: begin
        // set vars
        clock_count = 1;
        macc_clear = 1'b0;
        macc_clear2 = 1'b0;
        done = 0'b0;
        i_c = 0;
        j_c = 0;
        k_c = 0;
        begin
            if (start == 1'b1) begin
                state_c = A_STATE;
            end
        end
    end
    A_STATE: begin
        // access arrays by column-major order
        X = inA[(k*N)+i];
        Y = inB[(j*N)+k];

        $display("\n X[%d,%d] = %d", i, k, X);
        $display("\n Y[%d,%d] = %d", k, j, Y);
        macc_clear = 1'b1; // we are starting a new cume

        // access arrays by column-major order
        X2 = inA[(k*N)+(i+4)];

        $display("\n X[%d,%d] = %d", i+4, k, X2);
        $display("\n Y[%d,%d] = %d", k, j, Y);
        macc_clear2 = 1'b1; // we are starting a new cume
        clock_count = clock_count + 1; // increment cycle count
        k_c = k + 1;

        state_c = B_STATE;
    end
    B_STATE: begin
        macc_clear = 1'b0; // accumulate

```

```

X = inA[(k*N)+i];
Y = inB[(j*N)+k];

$display("\n X[%d,%d] = %d", i, k, X);
$display("\n Y[%d,%d] = %d", k, j, Y);

clock_count = clock_count +1; // increment cycle count

// access arrays by column-major order
X2 = inA[(k*N)+(i+4)];

$display("\n X[%d,%d] = %d", i+4, k, X2);
$display("\n Y[%d,%d] = %d", k, j, Y);
macc_clear2 = 1'b0; // we are starting a new cume

// verify where we go next
if (k + 1 < N) begin
    k_c = k + 1;

    state_c = B_STATE; // keep adding and accumulating
end
else begin
    state_c = C_STATE;
end
end
C_STATE: begin
    $display("\n Result[%d,%d] = %d", i, j, result);
    $display("\n Result[%d,%d] = %d", i+4, j, result2);
    // save this result to buffer
    //RAMOUTPUT.mem[(j*N)+i] = result;

    buffer1 = result;
    i1 <= i;
    j1 <= j;
    writebuffer1 = 1'b1;
    i2 <= i + 4;
    j2 <= j1;
    buffer2 = result2;
    writebuffer2 = 1'b1;

    // adjust vars
    clock_count = clock_count + 1; // increment cycle count

    // verify where we go next
    if (j + 1 < N) begin
        k_c = 0;
        j_c = j + 1;
        state_c = A_STATE;
    end
end

```

```

        end
        else begin
            state_c = D_STATE;
        end
    end
    D_STATE: begin
        // adjust vars
        clock_count = clock_count + 1; // increment cycle count

        // verify where we go next
        if (i + 1 < N/2)begin
            i_c = i + 1;
            j_c = 0;
            k_c = 0;
            state_c = A_STATE;
        end
        else begin
            state_c = DONE;
        end
    end
    DONE: begin
        state_c = DONE;
        // write for write to RAM
        if (writebuffer1 == 1'b0 && writebuffer2 == 1'b0) begin
            done = 0'b1;
        end
        else begin
            clock_count = clock_count + 1; // increment cycle
count
        end
    end
    default: begin
        state_c = WAIT;
    end
endcase
if (reset == 1'b1) begin
    state_c = WAIT;
end

end
always @(posedge clk or posedge start) begin
    state <= #1 state_c; // reset works even if disabled
    k <= k_c;
    j <= j_c;
    i <= i_c;
    if (writebuffer1 == 1'b1) begin
        RAMOUTPUT.mem[(j1*N)+i1] = buffer1;
        $display("\n Result1[%d,%d] = %d", i1, j1, buffer1);
    end
end

```

```

        writebuffer1 <= #1 0'b0;
    end
    else begin
        if (writebuffer2 == 1'b1) begin
            RAMOUTPUT.mem[(j2*N)+i2] = buffer2;
            $display("\n Result2[%d,%d] = %d", i2, j2, buffer2);
            writebuffer2 <= #1 0'b0;
        end
    end
end
endmodule

```

Figure 3.1.1 Verilog code for dual MACCs and two ports in matrix A.

Part 3.2 Testbench for matrix multiplication of two 8x8 matrices with two MACCs and matrix A read from dual-ported memory

```

`timescale 1ns/10ps

// Test bench module
module tb_lab6_partiii;

// Input Array
/////////////////////////////////////////////////////////////////
//                               Test Bench Signals                               //
/////////////////////////////////////////////////////////////////
reg clk;
integer i,j,k;

// Matrices
reg signed [7:0] matrixA [63:0];
reg signed [7:0] matrixB [63:0];
reg signed [18:0] matrixC [63:0];

// Comparison Flag
reg comparison;

/////////////////////////////////////////////////////////////////
//                               I/O Declarations                               //
/////////////////////////////////////////////////////////////////
// declare variables to hold signals going into submodule
reg start;
reg reset;

```

```

// Misc "wires"
wire done;
wire [10:0] clock_count;

////////////////////////////////////
//          Submodule Instantiation          //
////////////////////////////////////

/*****
|-----|    RENAME TO MATCH YOUR MODULE */
matrix_mult_2mac DUT
(
    .clk    (clk),
    .start  (start),
    .reset  (reset),
    .done   (done),
    .clock_count (clock_count)
);

initial begin

    /*****
    // CHANGE .TXT FILE NAMES TO MATCH THE ONES USED IN
    // YOUR MEMORY MODULES

    // Initialize Matrices
    $readmemb("ram_a_init.txt",matrixA);
    $readmemb("ram_b_init.txt",matrixB);

    /*****

    //////////////////////////////////////
    //          Perform Test          //
    //////////////////////////////////////
    reset <= 1'b1;
    start <= 1'b0;
    clk <= 1'b0;
    repeat(2) @(posedge clk);
    reset <= 1'b0;
    repeat(2) @(posedge clk);
    start <= 1'b1;
    repeat(1) @(posedge clk);
    start <= 1'b0;

    // -----
    // Wait for done or timeout
    fork : wait_or_timeout
    begin

```

```

        repeat(1000) @(posedge clk);
        disable wait_or_timeout;
    end
    begin
        @(posedge done);
        disable wait_or_timeout;
    end
    join
    // End Timeout Routing
    //-----

    //////////////////////////////////////
    //                               Verify Computation                               //
    //////////////////////////////////////
    // Print Input Matrices
    $display("Matrix A");
    for(i=0;i<8;i=i+1) begin

        $display(matrixA[i],matrixA[i+8],matrixA[i+16],matrixA[i+24],matrixA[i+32],ma
        trixA[i+40],matrixA[i+48],matrixA[i+56]);
        end

        $display("\n Matrix B");
        for(i=0;i<8;i=i+1) begin

            $display(matrixB[i],matrixB[i+8],matrixB[i+16],matrixB[i+24],matrixB[i+32],ma
            trixB[i+40],matrixB[i+48],matrixB[i+56]);
            end

        // Generate Expected Result
        for(i=0;i<8;i=i+1) begin
            for(j=0;j<8;j=j+1) begin
                matrixC[8*i+j] = 0;
                for(k=0;k<8;k=k+1) begin
                    matrixC[8*i+j] = matrixC[8*i+j] + matrixA[j+8*k]*matrixB[k+8*i];
                end
            end
        end

        // Display Expected Result
        $display("\nExpected Result");
        for(i=0;i<8;i=i+1) begin

            $display(matrixC[i],matrixC[i+8],matrixC[i+16],matrixC[i+24],matrixC[i+32],ma
            trixC[i+40],matrixC[i+48],matrixC[i+56]);
            end

        // Display Output Matrix

```

```

$display("\nGenerated Result");
for(i=0;i<8;i=i+1) begin

$display(DUT.RAMOUTPUT.mem[i],DUT.RAMOUTPUT.mem[i+8],DUT.RAMOUTPUT.mem[i+16],
DUT.RAMOUTPUT.mem[i+24],DUT.RAMOUTPUT.mem[i+32],DUT.RAMOUTPUT.mem[i+40],DUT.R
AMOUTPUT.mem[i+48],DUT.RAMOUTPUT.mem[i+56]);
end

// Test if the two matrices match
comparison = 1'b0;
for(i=0;i<8;i=i+1) begin
    for(j=0;j<8;j=j+1) begin
        if (matrixC[8*i+j] != DUT.RAMOUTPUT.mem[8*i+j]) begin
            $display("Mismatch at indices [%1.1d,%1.1d]",j,i);
            comparison = 1'b1;
        end
    end
end
if (comparison == 1'b0) begin
    $display("\nsuccess :)");
end

$display("Running Time = %d clock cycles",clock_count);

$stop; // End Simulation
end

// Clock
always begin
    #10;          // wait for initial block to initialize clock
    clk = ~clk;
end

endmodule

```

Figure 3.2.1 Testbench Verilog code for dual MACCs and two ports in matrix A.

```

# Matrix A
#  -12 -14 -71  63 122  53  44 -70
# -103-100 -82  46-116 -33  15-113
#  124   7  81  62 -20  -1-117 -87
#   59   8  69 105 -35-126-106-124
#  124-123 -16-118  96  95 -63 -14
#  -66  25-107  32  85 127  61 114
#  111  10  61 -57 107 -98  28  54
# -106  63  91 -71-123  -9   0  48
#

```

```

# Matrix B
# -114 19 -62 80 -24 -28 108 -39
# -100 -35-121-104-118 103-114 13
# 114 -93 -17 2 65-106 -66 -4
# 81-110 110-101 8 5 -80-110
# 84 115-113 -21 75 -45 -3 124
# -44 16 30 -64 67 -59 40 -32
# 91 -76 -3-127 109 -50 18 39
# -62 -98-107 107 -69 73 -61 94
#
# Expected Result
# 16037 18329 5737 -25041 20156 -9192 6762 2208
# 16199 175 49104 -12098 7831 -326 8199 -25380
# -7469 2860 8798 9126 -6358 -10569 5509 -27078
# 9491 -2959 19512 2414 -11100 -2360 -6773 -30121
# -14199 39849 -11824 33005 17218 -22832 41142 11700
# -4547 301 -11770 -16788 5602 9879 -6507 19378
# 1183 5673 -36292 19966 450 -5320 4351 24399
# -2505 -23865 -1915 616 -12683 9026 -21875 1947
#
# Generated Result
# 16037 18329 5737 -25041 20156 -9192 6762 2208
# 16199 175 49104 -12098 7831 -326 8199 -25380
# -7469 2860 8798 9126 -6358 -10569 5509 -27078
# 9491 -2959 19512 2414 -11100 -2360 -6773 -30121
# -14199 39849 -11824 33005 17218 -22832 41142 11700
# -4547 301 -11770 -16788 5602 9879 -6507 19378
# 1183 5673 -36292 19966 450 -5320 4351 24399
# -2505 -23865 -1915 616 -12683 9026 -21875 1947
#
# success :)
# Running Time = 356 clock cycles

```

Figure 3.2.2 Testbench results for dual MACCs and two ports in matrix A.

Part 3.3 Estimate the resource usage


Analysis & Synthesis Resource Usage Summary		
 <<Filter>>		
	Resource	Usage
1	Estimated Total logic elements	1,119
2		
3	Total combinational functions	1119
4	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	427
2	-- 3 input functions	656
3	-- <=2 input functions	36
5		
6	▼ Logic elements by mode	
1	-- normal mode	1109
2	-- arithmetic mode	10
7		
8	▼ Total registers	21
1	-- Dedicated logic registers	21
2	-- I/O registers	0
9		
10	I/O pins	149
11		
12	Embedded Multiplier 9-bit elements	0

Figure 3.3.1 Quartus Resource Usage report for dual MACCs and two ports in matrix A.

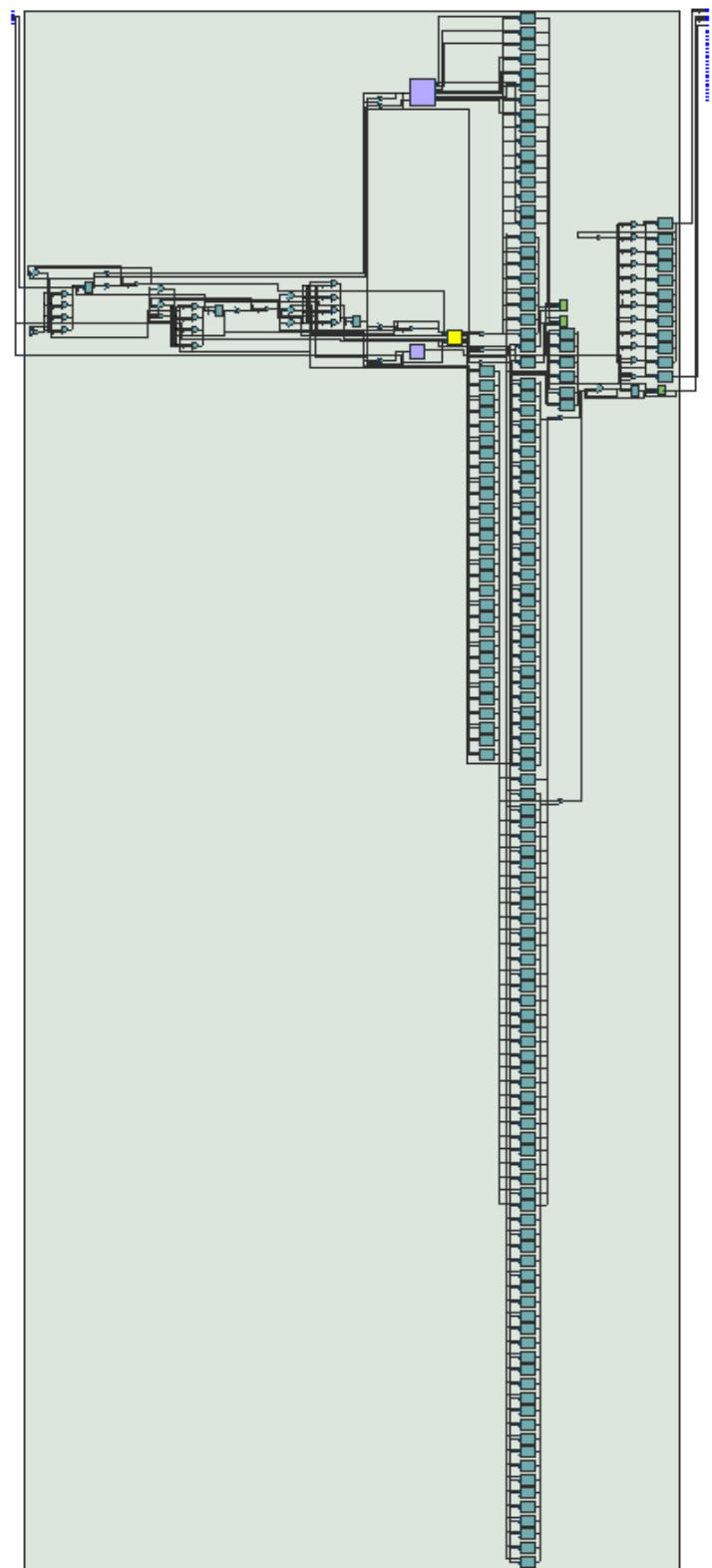


Figure 3.3.2 Quartus diagram for dual MACCs and two ports in matrix A

Part 3.4 Performance estimation

Using the number of clock cycles we measured to calculate the matrix multiplication of 356 clock cycles at a frequency of 50 MHz, the circuit does the calculation in 0.00712. Compared to the 0.02176s that it took in the previous circuit we got an improvement of approximately 3.06 times faster. The bottlenecks that we have in this circuit are:

- Reading matrix elements from matrix B one by one.
- Being able to calculate only two results at a time
- Writing one result at a time to the result matrix

4 Task 4 Matrix Multiply adopting parallelism in terms of MACC usage and pipelining the datapath

Part 4.1 Matrix multiplication of two 8x8 matrices with 4 MACCs and matrices A & B read from dual-ported memory

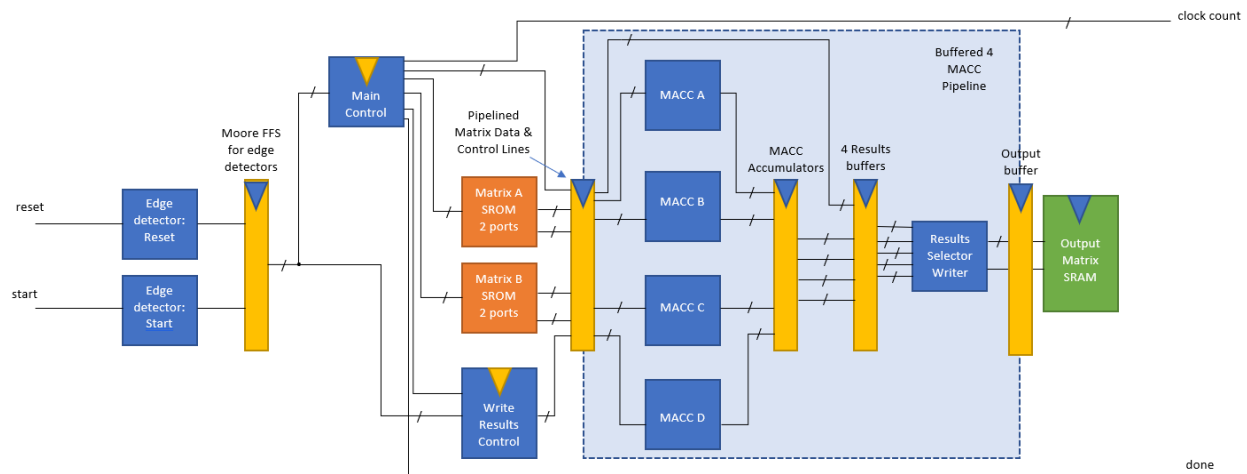


Figure 4.1.1 Block diagram for pipelined datapath with dual-ported reads on input matrices, 4 MACCs, and buffered write-to memory using an auxiliary write-to memory controller design.

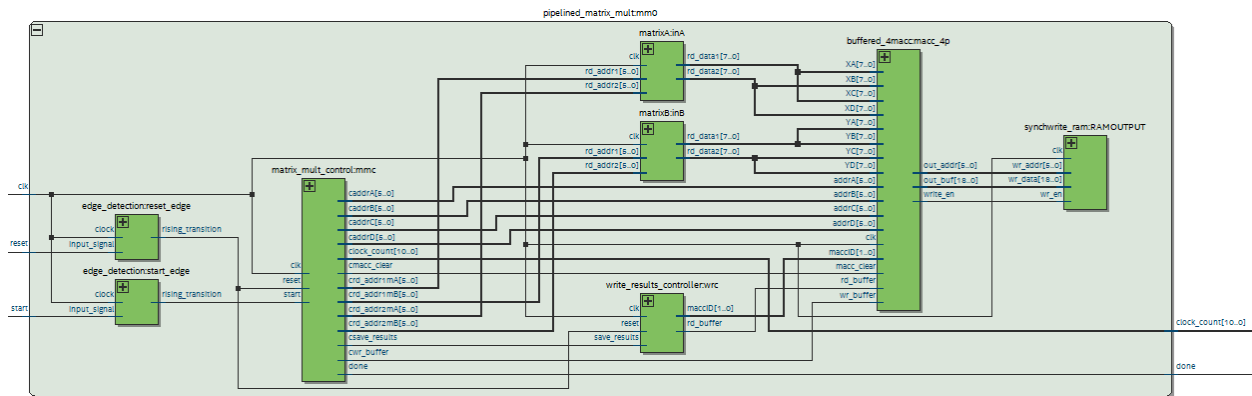


Figure 4.1.2 Quartus generated block diagram for pipelined datapath with dual-ported reads on input matrices, 4 MACCs, and buffered write-to memory using an auxiliary write-to memory controller design.

```
`timescale 1ns/10ps
module edge_detection(
input clock,
input input_signal,
output reg rising_transition
);
// declarations
reg n;
wire rising_transition_c;
// logic to detect 0 in previous cycle and 1 in current cycle
assign rising_transition_c= ~n & input_signal;
// flip-flop instantiations
always @(posedge clock) begin
    n <= #1 input_signal;
    rising_transition<= #1 rising_transition_c;
end
endmodule

`timescale 1ns/10ps
module matrixA(input clk,
                input [5:0]rd_addr1, output reg signed [7:0]rd_data1,    // read
port 1
                input [5:0]rd_addr2, output reg signed [7:0]rd_data2);

reg signed [7:0] mem [63:0];
reg [5:0]pipe_rd_addr1;
reg [5:0]pipe_rd_addr2;
```

initial begin

```
mem[    0] = -12;
mem[    1] = -103;
mem[    2] = 124;
mem[    3] = 59;
mem[    4] = 124;
mem[    5] = -66;
mem[    6] = 111;
mem[    7] = -106;
mem[    8] = -14;
mem[    9] = -100;
mem[   10] = 7;
mem[   11] = 8;
mem[   12] = -123;
mem[   13] = 25;
mem[   14] = 10;
mem[   15] = 63;
mem[   16] = -71;
mem[   17] = -82;
mem[   18] = 81;
mem[   19] = 69;
mem[   20] = -16;
mem[   21] = -107;
mem[   22] = 61;
mem[   23] = 91;
mem[   24] = 63;
mem[   25] = 46;
mem[   26] = 62;
mem[   27] = 105;
mem[   28] = -118;
mem[   29] = 32;
mem[   30] = -57;
mem[   31] = -71;
mem[   32] = 122;
mem[   33] = -116;
mem[   34] = -20;
mem[   35] = -35;
mem[   36] = 96;
mem[   37] = 85;
mem[   38] = 107;
mem[   39] = -123;
mem[   40] = 53;
mem[   41] = -33;
mem[   42] = -1;
mem[   43] = -126;
mem[   44] = 95;
mem[   45] = 127;
```

```

        mem[        46] = -98;
        mem[        47] = -9;
        mem[        48] = 44;
        mem[        49] = 15;
        mem[        50] = -117;
        mem[        51] = -106;
        mem[        52] = -63;
        mem[        53] = 61;
        mem[        54] = 28;
        mem[        55] = 0;
        mem[        56] = -70;
        mem[        57] = -113;
        mem[        58] = -87;
        mem[        59] = -124;
        mem[        60] = -14;
        mem[        61] = 114;
        mem[        62] = 54;
        mem[        63] = 48;
end

always @(posedge clk) begin

    rd_data1 <= #1 mem[rd_addr1];
    rd_data2 <= #1 mem[rd_addr2];

    pipe_rd_addr1 <= #1 rd_addr1;
    pipe_rd_addr2 <= #1 rd_addr2;

    // print pipeline stage 1
    $display("Pipeline Stage 0.5 A +++++");
    $display("Pipeline Stage 0.5: Next Read port 1 matrix A address[%d] =
%d", rd_addr1, mem[rd_addr1]);
    $display("Pipeline Stage 0.5: Next Read port 2 matrix A address[%d] =
%d", rd_addr2, mem[rd_addr2]);
    $display("Pipeline Stage 1 A +++++");
    $display("Pipeline Stage 1: Read port 1 matrix A[%d] = %d",
pipe_rd_addr1, rd_data1);
    $display("Pipeline Stage 1: Read port 2 matrix A[%d] = %d",
pipe_rd_addr2, rd_data2);
end

endmodule

`timescale 1ns/10ps
module matrixB(input clk,
               input [5:0]rd_addr1, output reg signed [7:0]rd_data1,    // read
port 1
               input [5:0]rd_addr2, output reg signed [7:0]rd_data2);

```

```
reg signed [7:0] mem [63:0];
reg [5:0] pipe_rd_addr1;
reg [5:0] pipe_rd_addr2;

initial begin
    mem[      0] = -114;
    mem[      1] = -100;
    mem[      2] =  114;
    mem[      3] =   81;
    mem[      4] =   84;
    mem[      5] =  -44;
    mem[      6] =   91;
    mem[      7] =  -62;
    mem[      8] =   19;
    mem[      9] =  -35;
    mem[     10] =  -93;
    mem[     11] = -110;
    mem[     12] =  115;
    mem[     13] =   16;
    mem[     14] =  -76;
    mem[     15] =  -98;
    mem[     16] =  -62;
    mem[     17] = -121;
    mem[     18] =  -17;
    mem[     19] =  110;
    mem[     20] = -113;
    mem[     21] =   30;
    mem[     22] =   -3;
    mem[     23] = -107;
    mem[     24] =   80;
    mem[     25] = -104;
    mem[     26] =    2;
    mem[     27] = -101;
    mem[     28] =  -21;
    mem[     29] =  -64;
    mem[     30] = -127;
    mem[     31] =  107;
    mem[     32] =  -24;
    mem[     33] = -118;
    mem[     34] =   65;
    mem[     35] =    8;
    mem[     36] =   75;
    mem[     37] =   67;
    mem[     38] =  109;
    mem[     39] =  -69;
    mem[     40] =  -28;
```

```

        mem[        41] = 103;
        mem[        42] = -106;
        mem[        43] = 5;
        mem[        44] = -45;
        mem[        45] = -59;
        mem[        46] = -50;
        mem[        47] = 73;
        mem[        48] = 108;
        mem[        49] = -114;
        mem[        50] = -66;
        mem[        51] = -80;
        mem[        52] = -3;
        mem[        53] = 40;
        mem[        54] = 18;
        mem[        55] = -61;
        mem[        56] = -39;
        mem[        57] = 13;
        mem[        58] = -4;
        mem[        59] = -110;
        mem[        60] = 124;
        mem[        61] = -32;
        mem[        62] = 39;
        mem[        63] = 94;
end

always @(posedge clk) begin

    rd_data1 <= #1 mem[rd_addr1];
    rd_data2 <= #1 mem[rd_addr2];

    pipe_rd_addr1 <= #1 rd_addr1;
    pipe_rd_addr2 <= #1 rd_addr2;

    // print pipeline stage 1
    $display("Pipeline Stage 0.5 B +++++");
    $display("Pipeline Stage 0.5: Next Read port 1 matrix B address[%d] =
%d", rd_addr1, mem[rd_addr1]);
    $display("Pipeline Stage 0.5: Next Read port 2 matrix B address[%d] =
%d", rd_addr2, mem[rd_addr2]);
    $display("Pipeline Stage 1 B +++++");
    $display("Pipeline Stage 1: Read port 1 matrix B[%d] = %d",
pipe_rd_addr1, rd_data1);
    $display("Pipeline Stage 1: Read port 2 matrix B[%d] = %d",
pipe_rd_addr2, rd_data2);

end

```

```

endmodule

`timescale 1ns/10ps
module buffered_4macc(input clk,
    input signed [7:0]XA, input signed [7:0]YA, input [5:0]addrA,
    input signed [7:0]XB, input signed [7:0]YB, input [5:0]addrB,
    input signed [7:0]XC, input signed [7:0]YC, input [5:0]addrC,
    input signed [7:0]XD, input signed [7:0]YD, input [5:0]addrD,
    input macc_clear, input rd_buffer, input wr_buffer, input
[1:0]maccID,
    output reg [5:0]out_addr, output reg signed [18:0]out_buf, output
reg write_en);

// local wiring / registers
wire [18:0]maccA_result;
wire [18:0]maccB_result;
wire [18:0]maccC_result;
wire [18:0]maccD_result;

// registers for pipeline inputs
//reg signed [7:0]iXA;
//reg signed [7:0]iYA;
//reg signed [7:0]iXB;
//reg signed [7:0]iYB;
//reg signed [7:0]iXC;
//reg signed [7:0]iYC;
//reg signed [7:0]iXD;
//reg signed [7:0]iYD;

reg [5:0]iaddrA;
reg [5:0]iaddrB;
reg [5:0]iaddrC;
reg [5:0]iaddrD;

reg [5:0]out_addrA;
reg [5:0]out_addrB;
reg [5:0]out_addrC;
reg [5:0]out_addrD;

reg signed [18:0]out_bufA;
reg signed [18:0]out_bufB;
reg signed [18:0]out_bufC;
reg signed [18:0]out_bufD;

reg imacc_clear, ird_buffer, iwr_buffer;
reg [1:0]imaccID;

// wire maccs, use pipelined inputs

```

```

macc maccA(XA, YA, imacc_clear, clk, maccA_result);
macc maccB(XB, YB, imacc_clear, clk, maccB_result);
macc maccC(XC, YC, imacc_clear, clk, maccC_result);
macc maccD(XD, YD, imacc_clear, clk, maccD_result);

// pipeline inputs
always @(posedge clk) begin
    imacc_clear <= #1 macc_clear;
    ird_buffer <= #1 rd_buffer;
    iwr_buffer <= #1 wr_buffer;
    imaccID <= #1 maccID;

    // input data is pipelined at memory, no need to do it here
    // for maccA
    //iXA <= #1 XA;
    //iYA <= #1 YA;
    iaddrA <= #1 addrA;

    // for maccB
    //iXB <= #1 XB;
    //iYB <= #1 YB;
    iaddrB <= #1 addrB;

    // for maccC
    //iXC <= #1 XC;
    //iYC <= #1 YC;
    iaddrC <= #1 addrC;

    // for maccD
    //iXD <= #1 XD;
    //iYD <= #1 YD;
    iaddrD <= #1 addrD;

    // print pipeline stage 2
    $display("Pipeline Stage 2 +++++");
    $display("Pipeline Stage 2: imacc_clear = %d", imacc_clear);
    $display("Pipeline Stage 2: ird_buffer = %d", ird_buffer);
    $display("Pipeline Stage 2: iwr_buffer = %d", iwr_buffer);
    $display("Pipeline Stage 2: imaccID = %d", imaccID);
    $display("Pipeline Stage 2: XA = %d", XA);
    $display("Pipeline Stage 2: YA = %d", YA);
    $display("Pipeline Stage 2: iaddrA = %d", iaddrA);
    $display("Pipeline Stage 2: XB = %d", XB);
    $display("Pipeline Stage 2: YB = %d", YB);
    $display("Pipeline Stage 2: addrB = %d", addrB);
    $display("Pipeline Stage 2: XC = %d", XC);
    $display("Pipeline Stage 2: YC = %d", YC);
    $display("Pipeline Stage 2: iaddrC = %d", iaddrC);

```

```

        $display("Pipeline Stage 2: XD = %d", XD);
        $display("Pipeline Stage 2: YD = %d", YD);
        $display("Pipeline Stage 2: iaddrD = %d", iaddrD);

end

// pipeline outputs - use pipelined inputs
always @(posedge clk) begin
    // print pipeline stage 3
    $display("Pipeline Stage 3 +++++");
    $display("Pipeline Stage 3: maccA_result[%d] = %d", iaddrA,
maccA_result);
    $display("Pipeline Stage 3: maccB_result[%d] = %d", iaddrB,
maccB_result);
    $display("Pipeline Stage 3: maccC_result[%d] = %d", iaddrC,
maccC_result);
    $display("Pipeline Stage 3: maccD_result[%d] = %d", iaddrD,
maccD_result);

    // print pipeline stage 4
    $display("Pipeline Stage 4 +++++");
    $display("Pipeline Stage 4: out_bufA[%d] = %d", out_addrA, out_bufA);
    $display("Pipeline Stage 4: out_bufB[%d] = %d", out_addrB, out_bufB);
    $display("Pipeline Stage 4: out_bufC[%d] = %d", out_addrC, out_bufC);
    $display("Pipeline Stage 4: out_bufD[%d] = %d", out_addrD, out_bufD);

    // write macc to buffers
    if (iwr_buffer == 1'b1) begin
        out_addrA <= #1 iaddrA;
        out_bufA <= #1 maccA_result;
        out_addrB <= #1 iaddrB;
        out_bufB <= #1 maccB_result;
        out_addrC <= #1 iaddrC;
        out_bufC <= #1 maccC_result;
        out_addrD <= #1 iaddrD;
        out_bufD <= #1 maccD_result;

    end

    // read one of the results
    if (ird_buffer == 1'b1) begin
        case (imaccID)
            2'b00: begin
                out_buf <= #1 out_bufA;
                out_addr <= #1 out_addrA;
            end
            2'b01: begin
                out_buf <= #1 out_bufB;

```

```

        out_addr <= #1 out_addrB;
    end
    2'b10: begin
        out_buf <= #1 out_bufC;
        out_addr <= #1 out_addrC;
    end
    2'b11: begin
        out_buf <= #1 out_bufD;
        out_addr <= #1 out_addrD;
    end
    default: begin
        imaccID <= imaccID;
    end
endcase
end
write_en <= #1 ird_buffer;

// print pipeline stage 5
$display("Pipeline Stage 5 +++++");
$display("Pipeline Stage 5: out_buf[%d] = %d", out_addr, out_buf);
$display("Pipeline Stage 5: write_en = %d", write_en);
end

endmodule

`timescale 1ns/10ps
module synchwrite_ram(input clk, input [5:0]wr_addr, input signed
[18:0]wr_data, input wr_en);

reg signed [18:0] mem[63:0];

always @(posedge clk) begin
    if (wr_en== 1'b1) begin
        mem[wr_addr] <= #1 wr_data;

        // last piepipeline stage
        $display("Pipeline Stage 6 +++++");
        $display("Pipeline Stage 6: matrix C [%d] = %d" , wr_addr,
mem[wr_addr]);
    end
end

end

endmodule

`timescale 1ns/10ps
module synchwrite_ram(input clk, input [5:0]wr_addr, input signed
[18:0]wr_data, input wr_en);

```

```

reg signed [18:0] mem[63:0];

always @(posedge clk) begin
    if (wr_en== 1'b1) begin
        mem[wr_addr] <= #1 wr_data;

        // last pipeline stage
        $display("Pipeline Stage 6 +++++");
        $display("Pipeline Stage 6: matrix C [%d] = %d" , wr_addr,
mem[wr_addr]);
        end
    end

end

endmodule

`timescale 1ns/10ps
modulematrix_mult_control    (input clk,
                             input start,
                             input reset,
                             output reg [5:0]caddrA,
                             output reg [5:0]caddrB,
                             output reg [5:0]caddrC,
                             output reg [5:0]caddrD,
                             output reg cmacc_clear,
                             output reg cwr_buffer,
                             output reg csave_results,
                             output reg [5:0]crd_addr1mA,
                             output reg [5:0]crd_addr2mA,
                             output reg [5:0]crd_addr1mB,
                             output reg [5:0]crd_addr2mB,
                             output reg done,
                             output reg [10:0] clock_count);
    parameter [2:0]WAIT      = 3'b000;
parameter [2:0]A_STATE = 3'b001;
parameter [2:0]B_STATE = 3'b011;
parameter [2:0]C_STATE = 3'b111;
parameter [2:0]D_STATE = 3'b110;
parameter [2:0]DONE      = 3'b100;
parameter N = 8;

parameter [1:0]MACCA = 2'b00;
parameter [1:0]MACCB = 2'b01;
parameter [1:0]MACCC = 2'b10;
parameter [1:0]MACCD = 2'b11;

reg [2:0]state_c;

```

```

        reg [2:0]state;

integer i,j,k;
reg [3:0]i_c,j_c,k_c;

reg [2:0] write_cycles;
reg [2:0] write_cyclesc;

reg state_control;
reg state_control_c;

always @(reset) begin
    state_control_c = 0;
end

always @(reset or start or state_control) begin
    state_c = state;
    state_control_c = ~state_control;
    $display("Main Controller =====");

    $display("State = %b" ,state);
    $display("state_control = %b" ,state_control);
    $display("clock_count = %d", clock_count);
    $display("k = %d", k);
    $display("j = %d", j);
    $display("i = %d", i);
    $display("csave_results = %d", csave_results);
    $display("done = %d", done);
    $display("Main Controller =====");

    i_c = i;
    j_c = j;
    k_c = k;

    case (state)
        WAIT: begin
            // set internal vars
            i_c = 0;
            j_c = 0;
            k_c = 0;

            // set matrix A read addresses
            crd_addr1mA = 0;
            crd_addr2mA = 0;

            // set matrix B read addresses
            crd_addr1mB = 0;
            crd_addr2mB = 0;

```

(default)

```
// set buffered_4macc vars
cmacc_clear = 1'b0;
cwr_buffer = 1'b0;
csave_results = 1'b0;
write_cyclesc = 0;

// set buffer addresses for matrix C (reults)
caddrA = 0;
caddrB = 0;
caddrC = 0;
caddrD = 0;

// set clock count
clock_count = 0;

// set done variable
done = 1'b0;

// determine if we get out of here, otherwise stay
if (start == 1'b1) begin
    state_c = A_STATE;
end
end
A_STATE: begin
    // access arrays by column-mayor order
    crd_addr1mA = (k*N)+i;
    crd_addr1mB = (j*N)+k;

    // access arrays by column-mayor order
    crd_addr2mA = (k*N)+(i+4);
    crd_addr2mB = ((j+4)*N)+k;

    // set buffered_4macc vars
    cmacc_clear = 1'b1; // we are starting a new cume
    cwr_buffer = 1'b0;
    csave_results = 1'b0;

    // set buffer addresses for matrix C (reults)
    caddrA = (j*N)+i;
    caddrB = (j*N)+(i+4);
    caddrC = ((j+4)*N)+i;
    caddrD = ((j+4)*N)+(i+4);

    // set clock count
    clock_count = clock_count +1; // increment cycle count
```

```

        // set done variable
        done = 1'b0;

        // increment row-column index
        k_c = k + 1;

        // we get out of here
        state_c = B_STATE;
    end
    B_STATE: begin
        // access arrays by column-major order
        crd_addr1mA = (k*N)+i;
        crd_addr1mB = (j*N)+k;

        // access arrays by column-major order
        crd_addr2mA = (k*N)+(i+4);
        crd_addr2mB = ((j+4)*N)+k;

        // set buffered_4macc vars
        cmacc_clear = 1'b0; // we are accumulating
        cwr_buffer = 1'b0;
        csave_results = 1'b0;

        // set buffer addresses for matrix C (results)
        caddrA = (j*N)+i;
        caddrB = (j*N)+(i+4);
        caddrC = ((j+4)*N)+i;
        caddrD = ((j+4)*N)+(i+4);

        // set clock count
        clock_count = clock_count + 1; // increment cycle count

        // set done variable
        done = 1'b0;

        // verify where we go next
        if (k + 1 < N) begin
            k_c = k + 1;

            state_c = B_STATE; // keep adding and accumulating
        end
        else begin
            state_c = C_STATE;
        end
    end
end
C_STATE: begin
    // set buffered_4macc vars

```

```

        cmacc_clear = 1'b0; // we are accumulating
        cwr_buffer = 1'b1; // write results to buffers
        csave_results = 1'b1; // start save results to memory on
next clock cycle

        // set buffer addresses for matrix C (results)
        caddrA = (j*N)+i;
        caddrB = (j*N)+(i+4);
        caddrC = ((j+4)*N)+i;
        caddrD = ((j+4)*N)+(i+4);

        // set clock count
        clock_count = clock_count +1; // increment cycle count

        // set done variable
        done = 1'b0;

        // verify where we go next
        if (j + 1 < N/2) begin
            k_c = 0;
            j_c = j + 1;
            state_c = A_STATE;
        end
        else begin
            state_c = D_STATE;
        end

    end
    D_STATE: begin
        // set buffered_4macc vars
        cmacc_clear = 1'b0; // we are accumulating
        cwr_buffer = 1'b0; // results are in buffers
        csave_results = 1'b1; // need to activate controller to
write the 4 buffered results

        // adjust vars
        clock_count = clock_count + 1; // increment cycle count

        // set done variable
        done = 1'b0;

        // verify where we go next
        if (i + 1 < N/2)begin
            i_c = i + 1;
            j_c = 0;
            k_c = 0;
            state_c = A_STATE;
        end
    end

```

```

        else begin
            state_c = DONE;
            write_cyclesc = 0;
        end

    end

    DONE: begin
        state_c = DONE;
        // set buffered_4maccc vars
        cmacc_clear = 1'b0;    // we are accumulating
        cwr_buffer = 1'b0;    // results are in buffers
        csave_results = 1'b0;  // need to activate controller to
write the 4 buffered results
        clock_count = clock_count + 1; // increment cycle count
        // wait four cycles
        if (write_cycles < 5) begin
            write_cyclesc = write_cycles + 1;
            done = 1'b0;
            $display("write_cycles in DONE = %d", write_cycles);
        end
        else begin
            //if (write_cycles < 7) begin
            //    write_cyclesc = write_cycles + 1;
            //    done = 1'b0;
            //    $display("write_cycles > 5 in DONE = %d",
write_cycles);
            //end
            //else begin
            state_control_c = state_control; // stay at
the same place, end
            $display("DONE *****");
            done = 1'b1;
            //end
        end

    end

    end

    default: begin
        state_c = WAIT;
    end

endcase
if (reset == 1'b1) begin
    state_c = WAIT;
    done = 1'b0;
    write_cyclesc = 0;
end

end

end

always @(posedge clk or posedge start) begin

```

```

        state_control <= #1 state_control_c;
        state <= #1 state_c; // reset works even if disabled
        k <= k_c;
        j <= j_c;
        i <= i_c;
        write_cycles <= #1 write_cyclesc;
end
endmodule

`timescale 1ns/10ps
module write_results_controller(input clk, input reset, input save_results,
output reg rd_buffer, output reg [1:0] maccID);

    parameter WAIT      = 1'b0;
    parameter A_STATE = 1'b1;
    parameter N = 8;

    parameter [1:0]MACCA = 2'b00;
    parameter [1:0]MACCB = 2'b01;
    parameter [1:0]MACCC = 2'b10;
    parameter [1:0]MACCD = 2'b11;

    reg state_c;
        reg state;

    integer i;
    reg [2:0]i_c;

    reg state_control;
    reg state_control_c;

    always @(reset) begin
        state_control_c = 0;
    end

    always @(save_results or reset or state_control) begin
        state_c = state;
        state_control_c = ~state_control;
        $display("Begin Write Results Controller
=====");
        $display("Write State = %b" ,state);
        $display("Write state_control = %b" ,state_control);
        $display("Write rd_buffer = %b" ,rd_buffer);
        $display("Write maccID = %b" ,maccID);
        $display("End Write Results Controller
=====");
        case (state)

```

```

        WAIT: begin
            i_c = 0;
            rd_buffer = 0;
            if (save_results == 1'b1) begin
                state_c = A_STATE;
            end
        end
    end
    A_STATE: begin
        rd_buffer = 1'b1;
        maccID = i;
        i_c = i + 1;
        if (i_c < 4) begin
            state_c = A_STATE;
        end
        else begin
            state_c = WAIT;
            i_c = 0;
        end
    end
    default: begin
        state_c = WAIT;
    end
endcase

if (reset == 1'b1) begin
    state_c = WAIT;
end

end

always @(posedge clk) begin
    state_control <= #1 state_control_c;
    state <= #1 state_c;
    i <= #1 i_c;
end

endmodule

`timescale 1ns/10ps
module pipelined_matrix_mult(input clk, input start, input reset, output
done, output [10:0] clock_count);

// 1st stage of pipeline, inputs are registered and edge detected
wire istart, ireset;      // pipeline inpts to module

// control lines for sram memory for input matrixes
wire [5:0]rd_addr1mA;

```

```

wire [7:0]rd_data1mA;      // read port 1, synch with pipeline registers
wire [5:0]rd_addr2mA;
wire [7:0]rd_data2mA;      // read port 2, synch with pipeline registers
wire [5:0]wr_addrmA;
wire [7:0]wr_datamA;
wire wr_enmA;              // write port, synch write with combinational logic

wire [5:0]rd_addr1mB;
wire [7:0]rd_data1mB;      // read port 1, synch with pipeline registers
wire [5:0]rd_addr2mB;
wire [7:0]rd_data2mB;      // read port 2, synch with pipeline registers
wire [5:0]wr_addrmB;
wire [7:0]wr_datamB;
wire wr_enmB;              // write port, synch write with combinational logic

// control lines for macc4 pipelined module
wire [7:0]XA;
wire [7:0]YA;
wire [5:0]addrA;           // for macc A
wire [7:0]XB;
wire [7:0]YB;
wire [5:0]addrB;           // for macc B
wire [7:0]XC;
wire [7:0]YC;
wire [5:0]addrC;           // for macc C
wire [7:0]XD;
wire [7:0]YD;
wire [5:0]addrD;           // for macc D
wire macc_clear;            // clea accumulators or auumulate
wire wr_buffer;             // write macc outputs to buffers
wire rd_buffer;
wire [1:0]maccID;           // read one of the four buffers with data an
address to pass to result memory

// these connec the macc4 pipelined module to the synch write ram
wire [5:0]out_addr;
wire [18:0]out_buf;         // address and data for the result memory
wire write_en;              // pipelined write to result
memory comand line

// synch write memory lines
wire [5:0]wr_addr;
wire [18:0]wr_data;         // address and data for the result memory
wire wr_en;                 // pipelined write to result memory
comand line

// control module inputs/outputs

```

```

wire [5:0]caddrA;                // address for result of macc A
wire [5:0]caddrB;                // address for result of macc B
wire [5:0]caddrC;                // address for result of macc C
wire [5:0]caddrD;                // address for result of macc D

wire cmacc_clear;                // clea accumulators or auumulate
wire cwr_buffer;                // write macc outputs to buffers
wire csave_results;              // activate write buffers
controller

wire crd_buffer;
wire [1:0]cmaccID;                // read one of the four buffers
with data an address to pass to result memory

wire [5:0]crd_addr1mA;           // address of matrix A port 1 to
read from
wire [5:0]crd_addr2mA;           // address of matrix A port 2 to
read from
wire [5:0]crd_addr1mB;           // address of matrix B port 1 to
read from
wire [5:0]crd_addr2mB;           // address of matrix B port 2 to
read from

// instantiate modules
// edge detection for reset
edge_detection reset_edge(
    clk,
    reset,
    ireset
);

// edge detection for start
edge_detection start_edge(
    clk,
    start,
    istart
);

// input matrixes
matrixA inA(clk, rd_addr1mA, rd_data1mA, // read port 1
            rd_addr2mA, rd_data2mA);    // read port 2

matrixB inB(clk, rd_addr1mB, rd_data1mB, // read port 1
            rd_addr2mB, rd_data2mB);    // read port 2

```

```

// buffered maccs
buffered_4macc macc_4p( clk,
                        XA, YA, addrA,
                        XB, YB, addrB,
                        XC, YC, addrC,
                        XD, YD, addrD,
                        macc_clear, rd_buffer, wr_buffer, maccID,
                        out_addr, out_buf, write_en);

// results memory
synchwrite_ram RAMOUTPUT(clk, wr_addr, wr_data, wr_en);

// control module
matrix_mult_control mmc(clk,
                        istart,
                        ireset,
                        caddrA,
                        caddrB,
                        caddrC,
                        caddrD,
                        cmacc_clear,
                        cwr_buffer,
                        csave_results,
                        crd_addr1mA,
                        crd_addr2mA,
                        crd_addr1mB,
                        crd_addr2mB,
                        done,
                        clock_count);

// write results control module
write_results_controller wrc(clk, ireset, csave_results, rd_buffer, maccID);

// connect modules

// connect control module to memory for matrix A & B
assign rd_addr1mA = crd_addr1mA;
assign rd_addr2mA = crd_addr2mA;
assign rd_addr1mB = crd_addr1mB;
assign rd_addr2mB = crd_addr2mB;

// connect input matrix A & B to maccs
assign XA = rd_data1mA;
assign YA = rd_data1mB;

assign XB = rd_data2mA;
assign YB = rd_data1mB;

```

```

assign XC = rd_data1mA;
assign YC = rd_data2mB;

assign XD = rd_data2mA;
assign YD = rd_data2mB;

// connect output addresses for accum into results
assign addrA = caddrA;           // address where to store
results for macc A
assign addrB = caddrB;           // address where to store
results for macc B
assign addrC = caddrC;           // address where to store
results for macc C
assign addrD = caddrD;           // address where to store
results for macc D

// connect command lines for maccs
assign macc_clear = cmacc_clear; // clear accumulators or
aumulate                                // write macc
assign wr_buffer = cwr_buffer ;        // write macc
outputs to buffers
assign rd_buffer = crd_buffer;         // read one of the
four buffers and then send to memory
assign maccID = cmaccID;               // which buffer to read to
send to memory

// connect maccs to results memory
assign wr_addr = out_addr;             // address to write
results to
assign wr_data = out_buf;              // data to write to
results
assign wr_en = write_en;               // write enable command,
from maccs module

// print pipeline stage 0
//   always @(posedge clk) begin
//       $display("Pipeline Stage 0 +++++");
//       $display("Pipeline Stage 0: istart = %d", istart);
//       $display("Pipeline Stage 0: ireset = %d", ireset);
//   end
endmodule

```

Figure 4.1.3 Verilog code for pipelined datapath with dual-ported reads on input matrices, 4 MACCs, and buffered write-to memory using an auxiliary write-to memory controller.

Part 4.2 Testbench for pipelined datapath with dual-ported reads on input matrices, 4 MACCs, and buffered write-to memory using an auxiliary write-to memory controller

```
`timescale 1ns/10ps

// Test bench module
module tb_lab6_partiv;

// Input Array
////////////////////////////////////////
//                               Test Bench Signals                               //
////////////////////////////////////////
reg clk;
integer i,j,k;

// Matrices
reg signed [7:0] matrixA [63:0];
reg signed [7:0] matrixB [63:0];
reg signed [18:0] matrixC [63:0];

// Comparison Flag
reg comparison;

////////////////////////////////////////
//                               I/O Declarations                               //
////////////////////////////////////////
// declare variables to hold signals going into submodule
reg start;
reg reset;

// Misc "wires"
wire done;
wire [10:0] clock_count;

////////////////////////////////////////
//                               Submodule Instantiation                               //
////////////////////////////////////////
pipelined_matrix_mult DUT
(
    .clk    (clk),
    .start  (start),
    .reset  (reset),
    .done    (done),
    .clock_count (clock_count)
);
```

initial begin

```
//*****
// CHANGE .TXT FILE NAMES TO MATCH THE ONES USED IN
// YOUR MEMORY MODULES

// Initialize Matrices
$readmemb("ram_a_init.txt",matrixA);
$readmemb("ram_b_init.txt",matrixB);

//*****

//////////////////////////////////////////
//                               Perform Test                               //
//////////////////////////////////////////
reset <= 1'b1;
start <= 1'b0;
clk <= 1'b0;
repeat(2) @(posedge clk);
reset <= 1'b0;
repeat(2) @(posedge clk);
start <= 1'b1;
repeat(1) @(posedge clk);
start <= 1'b0;

// -----
// Wait for done or timeout
fork : wait_or_timeout
begin
    repeat(250) @(posedge clk);
    disable wait_or_timeout;
end
begin
    @(posedge done);
    disable wait_or_timeout;
end
join
// End Timeout Routing
//-----

//////////////////////////////////////////
//                               Verify Computation                               //
//////////////////////////////////////////
// Print Input Matrices
$display("Matrix A");
for(i=0;i<8;i=i+1) begin

$display(matrixA[i],matrixA[i+8],matrixA[i+16],matrixA[i+24],matrixA[i+32],ma
```

```

trixA[i+40],matrixA[i+48],matrixA[i+56]));
end

$display("\n Matrix B");
for(i=0;i<8;i=i+1) begin

$display(matrixB[i],matrixB[i+8],matrixB[i+16],matrixB[i+24],matrixB[i+32],ma
trixB[i+40],matrixB[i+48],matrixB[i+56]));
end

// Generate Expected Result
for(i=0;i<8;i=i+1) begin
    for(j=0;j<8;j=j+1) begin
        matrixC[8*i+j] = 0;
        for(k=0;k<8;k=k+1) begin
            matrixC[8*i+j] = matrixC[8*i+j] + matrixA[j+8*k]*matrixB[k+8*i];
        end
    end
end
end

// Display Expected Result
$display("\nExpected Result");
for(i=0;i<8;i=i+1) begin

$display(matrixC[i],matrixC[i+8],matrixC[i+16],matrixC[i+24],matrixC[i+32],ma
trixC[i+40],matrixC[i+48],matrixC[i+56]));
end

// Display Output Matrix
$display("\nGenerated Result");
for(i=0;i<8;i=i+1) begin

$display(DUT.RAMOUTPUT.mem[i],DUT.RAMOUTPUT.mem[i+8],DUT.RAMOUTPUT.mem[i+16],
DUT.RAMOUTPUT.mem[i+24],DUT.RAMOUTPUT.mem[i+32],DUT.RAMOUTPUT.mem[i+40],DUT.R
AMOUTPUT.mem[i+48],DUT.RAMOUTPUT.mem[i+56]));
end

// Test if the two matrices match
comparison = 1'b0;
for(i=0;i<8;i=i+1) begin
    for(j=0;j<8;j=j+1) begin
        if (matrixC[8*i+j] != DUT.RAMOUTPUT.mem[8*i+j]) begin
            $display("Mismatch at indices [%1.1d,%1.1d]",j,i);
            comparison = 1'b1;
        end
    end
end
end
if (comparison == 1'b0) begin

```

```

        $display("\nsuccess :)");
    end

    $display("Running Time = %d clock cycles",clock_count);

    $stop; // End Simulation
end

// Clock
always begin
    #10;          // wait for initial block to initialize clock
    clk = ~clk;
end

endmodule

```

Figure 4.2.1 Verilog code for testbench for Part IV

```

# Matrix A
# -12 -14 -71 63 122 53 44 -70
# -103-100 -82 46-116 -33 15-113
# 124 7 81 62 -20 -1-117 -87
# 59 8 69 105 -35-126-106-124
# 124-123 -16-118 96 95 -63 -14
# -66 25-107 32 85 127 61 114
# 111 10 61 -57 107 -98 28 54
# -106 63 91 -71-123 -9 0 48
#
# Matrix B
# -114 19 -62 80 -24 -28 108 -39
# -100 -35-121-104-118 103-114 13
# 114 -93 -17 2 65-106 -66 -4
# 81-110 110-101 8 5 -80-110
# 84 115-113 -21 75 -45 -3 124
# -44 16 30 -64 67 -59 40 -32
# 91 -76 -3-127 109 -50 18 39
# -62 -98-107 107 -69 73 -61 94
#
# Expected Result
# 16037 18329 5737 -25041 20156 -9192 6762 2208
# 16199 175 49104 -12098 7831 -326 8199 -25380
# -7469 2860 8798 9126 -6358 -10569 5509 -27078
# 9491 -2959 19512 2414 -11100 -2360 -6773 -30121
# -14199 39849 -11824 33005 17218 -22832 41142 11700
# -4547 301 -11770 -16788 5602 9879 -6507 19378
# 1183 5673 -36292 19966 450 -5320 4351 24399
# -2505 -23865 -1915 616 -12683 9026 -21875 1947
#

```

```

# Generated Result
# 16037 18329 5737 -25041 20156 -9192 6762 2208
# 16199 175 49104 -12098 7831 -326 8199 -25380
# -7469 2860 8798 9126 -6358 -10569 5509 -27078
# 9491 -2959 19512 2414 -11100 -2360 -6773 -30121
# -14199 39849 -11824 33005 17218 -22832 41142 11700
# -4547 301 -11770 -16788 5602 9879 -6507 19378
# 1183 5673 -36292 19966 450 -5320 4351 24399
# -2505 -23865 -1915 616 -12683 9026 -21875 1947
#
# success :)
# Running Time = 154 clock cycles

```

Figure 4.2.2 Testbench results for Part IV

Part 4.3 Estimate the resource usage


Analysis & Synthesis Resource Usage Summary		
 <<Filter>>		
	Resource	Usage
1	Estimated Total logic elements	68
2		
3	Total combinational functions	66
4	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	25
2	-- 3 input functions	22
3	-- <=2 input functions	19
5		
6	▼ Logic elements by mode	
1	-- normal mode	56
2	-- arithmetic mode	10
7		
8	▼ Total registers	4
1	-- Dedicated logic registers	4
2	-- I/O registers	0
9		
10	I/O pins	149
11		
12	Embedded Multiplier 9-bit elements	0

Figure 4.3.1 Quartus Resource Usage Report

Part 4.4 Performance estimation

Using the number of clock cycles we measured to calculate the matrix multiplication of 154 clock cycles at a frequency of 50 MHz, the circuit does the calculation in 0.00308s. Compared to the 0.00712s it took on the previous circuit we got an improvement of approximately 2.3 times faster. Compared to the 0.02176s that it took in the original circuit we got an improvement of approximately 7.06 times faster. The bottlenecks that we have in this circuit are:

- a. Reading matrix elements from memory two by two only.
- b. Being able to calculate only four results at a time
- c. Writing one result at a time to the result matrix

5 Task 5 Extra Credit: Sparse matrix-vector Multiplication (SpMV)

Part 5.1 Implementation using CSR matrix representation with parallel data read on the indices and pipelining

```
`timescale 1ns/10ps
module edge_detection(
input clock,
input input_signal,
output reg rising_transition
);
// declarations
reg n;
wire rising_transition_c;
// logic to detect 0 in previous cycle and 1 in current cycle
assign rising_transition_c= ~n & input_signal;
// flip-flop instantiations
always @(posedge clock) begin
    n <= #1 input_signal;
    rising_transition<= #1 rising_transition_c;
end
endmodule

`timescale 1ns/10ps
module values(input clk,
               input [5:0]rd_addr1, output reg signed [7:0]rd_data1);

reg signed [7:0] mem [63:0];
```

```

initial begin
    $readmemb("ram_csr_values.txt",mem);
end

always @(posedge clk) begin

    rd_data1 <= #1 mem[rd_addr1];

    // print pipeline stage 1
    $display("Pipeline Stage 1 valuesx +++++");
    $display("Pipeline Stage 1: Next Read port 1 value address[%d] = %d",
rd_addr1, mem[rd_addr1]);
    $display("Pipeline Stage 1: Current Read port 1 value = %d", rd_data1);

end

endmodule

`timescale 1ns/10ps
module col_index(input clk,
    input [5:0]rd_addr1, output reg signed [3:0]rd_data1);

reg signed [3:0] mem [63:0];

initial begin
    $readmemb("ram_csr_cols.txt",mem);
end

always @(posedge clk) begin

    rd_data1 <= #1 mem[rd_addr1];

    // print pipeline stage 1
    $display("Pipeline Stage 1 column index +++++");
    $display("Pipeline Stage 1: Next Read port 1 column index address[%d] =
%d", rd_addr1, mem[rd_addr1]);
    $display("Pipeline Stage 1: Current Read port 1 column index = %d",
rd_data1);
end

endmodule

`timescale 1ns/10ps
module row_index(input clk,
    input [4:0]rd_addr1, output reg signed [7:0]rd_data1,
    input [4:0]rd_addr2, output reg signed [7:0]rd_data2);

```

```

reg signed [7:0] mem [4:0];

initial begin
    $readmemb("ram_csr_rows.txt",mem);
end

always @(posedge clk) begin

    rd_data1 <= #1 mem[rd_addr1];
    rd_data2 <= #1 mem[rd_addr2];

    // print pipeline stage 1
    $display("Pipeline Stage 1 row index +++++");
    $display("Pipeline Stage 1: Next Read port 1 row index address[%d] =
%d", rd_addr1, mem[rd_addr1]);
    $display("Pipeline Stage 1: Next Read port 2 row index address[%d] =
%d", rd_addr2, mem[rd_addr2]);

end

endmodule

`timescale 1ns/10ps
module buffered_macc(input clk,
    input signed [7:0]XA, input signed [7:0]YA, input [7:0]addrA,
    input macc_clear, input rd_buffer, input wr_buffer,
    output reg [7:0]out_addr, output reg signed [18:0]out_buf, output
reg write_en);

// local wiring / registers
wire [18:0]maccA_result;

reg [7:0]iaddrA;

reg [7:0]out_addrA;

reg signed [18:0]out_bufA;

reg imacc_clear, ird_buffer, iwr_buffer;

// wire maccs, use pipelined inputs
macc maccA(XA, YA, imacc_clear, clk, maccA_result);

// pipeline inputs
always @(posedge clk) begin

```

```

    imacc_clear <= #1 macc_clear;
    ird_buffer <= #1 rd_buffer;
    iwr_buffer <= #1 wr_buffer;

    // input data is pipelined at memory, no need to do it here
    // for maccA
    iaddrA <= #1 addrA;

    // print pipeline stage 2
    $display("Pipeline Stage 2 +++++");
    $display("Pipeline Stage 2: imacc_clear = %d", imacc_clear);
    $display("Pipeline Stage 2: ird_buffer = %d", ird_buffer);
    $display("Pipeline Stage 2: iwr_buffer = %d", iwr_buffer);
    $display("Pipeline Stage 2: XA = %d", XA);
    $display("Pipeline Stage 2: YA = %d", YA);
    $display("Pipeline Stage 2: iaddrA = %d", iaddrA);

end

// pipeline outputs - use pipelined inputs
always @(posedge clk) begin
    // print pipeline stage 3
    $display("Pipeline Stage 3 +++++");
    $display("Pipeline Stage 3: maccA_result[%d] = %d", iaddrA,
maccA_result);

    // print pipeline stage 4
    $display("Pipeline Stage 4 +++++");
    $display("Pipeline Stage 4: out_bufA[%d] = %d", out_addrA, out_bufA);

    // write macc to buffers
    if (iwr_buffer == 1'b1) begin
        out_addrA <= #1 iaddrA;
        out_bufA <= #1 maccA_result;
    end
    // read one of the results
    if (ird_buffer == 1'b1) begin
        out_buf <= #1 out_bufA;
        out_addr <= #1 out_addrA;
    end
    write_en <= #1 ird_buffer;

    // print pipeline stage 5
    $display("Pipeline Stage 5 +++++");
    $display("Pipeline Stage 5: out_buf[%d] = %d", out_addr, out_buf);
    $display("Pipeline Stage 5: write_en = %d", write_en);
end

```

```

endmodule

`timescale 1ns/10ps
module csr_synchwrite_ram(input clk, input [7:0]wr_addr, input signed
[18:0]wr_data, input wr_en);

reg signed [18:0] mem[255:0];
integer i;

initial begin
    for(i=0;i<256;i=i+1) begin
        mem[i] = 0;
    end
end

always @(posedge clk) begin
    if (wr_en== 1'b1) begin
        mem[wr_addr] <= #1 wr_data;

        // last piepipeline stage
        $display("Pipeline Stage 6 +++++");
        $display("Pipeline Stage 6: matrix C [%d] = %d" , wr_addr,
wr_data);
    end

end

endmodule

`timescale 1ns/10ps
module csr_smv_mult_control(input clk,
    input start,
    input reset,
    input [7:0]irow_i,
    input [7:0]irow_iplus1,
    input [3:0]icol_j,
    output reg [7:0]caddr_result,
    output reg cmacc_clear,
    output reg cwr_buffer,
    output reg csave_results,
    output reg [5:0]crd_addr_values,
    output reg [5:0]crd_addr_col_index,
    output reg [4:0]crd_addr_row_index1,
    output reg [4:0]crd_addr_row_index2,
    output reg [3:0]crd_addr_X,
    output reg done,

```

```

        output reg [10:0] clock_count);
    parameter [2:0]WAIT    = 3'b000;
parameter [2:0]A_STATE = 3'b001;
parameter [2:0]B_STATE = 3'b011;
parameter [2:0]C_STATE = 3'b111;
parameter [2:0]D_STATE = 3'b110;
parameter [2:0]DONE     = 3'b100;
parameter N = 16;

reg [2:0]state_c;
    reg [2:0]state;

integer j,i,total_j;
reg [3:0]j_c,i_c;

reg [2:0] write_cycles;
reg [2:0] write_cyclesc;

reg state_control;
reg state_control_c;

always @(reset) begin
    state_control_c = 0;
end

always @(reset or start or state_control) begin
    state_c = state;
    state_control_c = ~state_control;
    $display("Main Controller =====");

    $display("State = %b" ,state);
    $display("state_control = %b" ,state_control);
    $display("clock_count = %d", clock_count);
    $display("j = %d", j);
    $display("csave_results = %d", csave_results);
    $display("done = %d", done);
    $display("Main Controller =====");

    j_c = j;
    i_c = i;

    case (state)
        WAIT: begin
            // set internal vars
            j_c = 0;
            i_c = 0;

```

```

        // set matrix A read addresses
        crd_addr_values = 0;
        crd_addr_col_index = 0;
        crd_addr_row_index1 = 0;
        crd_addr_row_index2 = 0;

        // set vector read address
        crd_addr_X = 0;

        // set buffered_4macc vars
        cmacc_clear = 1'b0;
        cwr_buffer = 1'b0;
        csave_results = 1'b0;
        write_cyclesc = 0;

        // set buffer addresses for matrix C (reults)
        caddr_result = 0;

        // set clock count
        clock_count = 0;

        // set done variable
        done = 1'b0;

        // determine if we get out of here, otherwise stay
(default)    if (start == 1'b1) begin
                state_c = A_STATE;
            end

        end
A_STATE: begin
        // set matrix A read addresses
        crd_addr_values = i*4 + j;
        crd_addr_col_index = i*4 + j;
        crd_addr_row_index1 = i;    // read irow_i and irow_iplus1
to use with j    crd_addr_row_index2 = i + 1;

        // set vector read address
        crd_addr_X = icol_j;

        // set buffered_4macc vars
        cmacc_clear = 1'b1; // we are starting a new cume
        cwr_buffer = 1'b0;
        csave_results = 1'b0;

        // set buffer addresses for matrix C (reults)

```

```

        caddr_result = icol_j*16 + i;

        // set clock count
        clock_count = clock_count + 1 ; // increment cycle count

        // set done variable
        done = 1'b0;

        // we get out of here
        state_c = B_STATE;
end
B_STATE: begin
    // set row-column index
    total_j = 4; //irow_iplus1 - irow_i; // max j

    // set matrix A read addresses
    crd_addr_values = i*4 + j; // read value
    crd_addr_col_index = i*4 + j; // read column index
    crd_addr_row_index1 = i;
    crd_addr_row_index2 = i + 1;

    // set vector read address, now we can add in the MACC
    crd_addr_X = icol_j;

    // set buffered_4macc vars
    cmacc_clear = 1'b1; // we are starting a new cume
    cwr_buffer = 1'b0;
    csave_results = 1'b0;

    // set buffer addresses for matrix C (results)
    caddr_result = icol_j*16 + i;

    // set clock count
    clock_count = clock_count + 1; // increment cycle count

    // set done variable
    done = 1'b0;

    // we continue reading this row
    state_c = C_STATE;
end
C_STATE: begin
    // set matrix A read addresses
    crd_addr_values = i*4 + j; // read value
    crd_addr_col_index = i*4 + j; // read column index
    crd_addr_row_index1 = i;

```

```

        crd_addr_row_index2 = i + 1;

        // set vector read address, now we can add in the MACC
        crd_addr_X = icol_j;

        // set buffered_4macc vars
        cmacc_clear = 1'b1; // we are starting a new cume
        cwr_buffer = 1'b1;
        csave_results = 1'b1;

        // set buffer addresses for matrix C (results) (16x16)
        caddr_result = icol_j*16 + i;

        // set clock count
        clock_count = clock_count + 1; // increment cycle count

        // set done variable
        done = 1'b0;

        // verify where we go next
        if (j + 1 < total_j) begin
            j_c = j + 1;

            state_c = A_STATE; // keep adding and accumulating
        end
        else begin
            state_c = D_STATE;
        end
    end

    D_STATE: begin
        // set buffered_4macc vars
        cmacc_clear = 1'b1; // new cume
        cwr_buffer = 1'b0; // write results to buffer
        csave_results = 1'b0; // need to activate controller to
write the 4 buffered results

        // adjust vars
        clock_count = clock_count + 1; // increment cycle count

        // set done variable
        done = 1'b0;

        // verify where we go next
        if (i + 1 < N)begin
            i_c = i + 1;
            j_c = 0;
            state_c = A_STATE;

```

```

        end
        else begin
            state_c = DONE;
            write_cyclesc = 0;
        end

    end

    DONE: begin
        state_c = DONE;
        // set buffered_4macc vars
        cmacc_clear = 1'b0;    // we are accumulating
        cwr_buffer = 1'b0;    // results are in buffers
        csave_results = 1'b0;
        clock_count = clock_count + 1; // increment cycle count
        // wait four cycles
        if (write_cycles < 2) begin
            write_cyclesc = write_cycles + 1;
            done = 1'b0;
            $display("write_cycles in DONE = %d", write_cycles);
        end
        else begin
            //if (write_cycles < 7) begin
            //    write_cyclesc = write_cycles + 1;
            //    done = 1'b0;
            //    $display("write_cycles > 5 in DONE = %d",
write_cycles);
            //end
            //else begin
                state_control_c = state_control; // stay at
the same place, end
                $display("DONE *****");
                done = 1'b1;
            //end
        end

    end

    default: begin
        state_c = WAIT;
    end
endcase
if (reset == 1'b1) begin
    state_c = WAIT;
    done = 1'b0;
    write_cyclesc = 0;
end

end

always @(posedge clk or posedge start) begin

```

```

        state_control <= #1 state_control_c;
        state <= #1 state_c; // reset works even if disabled
        j <= j_c;
        i <= i_c;
        write_cycles <= #1 write_cyclesc;
end
endmodule

`timescale 1ns/10ps
module csr_smv_write_results_control(input clk, input reset, input
save_results, output reg rd_buffer);

    parameter WAIT    = 1'b0;
    parameter A_STATE = 1'b1;
    parameter N = 8;

    reg state_c;
    reg state;

    reg state_control;
    reg state_control_c;

    always @(reset) begin
        state_control_c = 0;
    end

    always @(save_results or reset or state_control) begin
        state_c = state;
        state_control_c = ~state_control;

        case (state)
            WAIT: begin
                rd_buffer = 0;
                if (save_results == 1'b1) begin
                    state_c = A_STATE;
                end
            end
            A_STATE: begin
                rd_buffer = 1'b1;
                state_c = WAIT;
            end
            default: begin
                state_c = WAIT;
            end
        endcase

        if (reset == 1'b1) begin
            state_c = WAIT;

```

```

        end
        $display("Begin Write Results Controller
=====");
        $display("Write State = %b" ,state);
        $display("Write state_control = %b" ,state_control);
        $display("Save Results = %b", save_results);
        $display("Write rd_buffer = %b" ,rd_buffer);
        $display("End Write Results Controller
=====");

end

always @(posedge clk) begin
    state_control <= #1 state_control_c;
    state <= #1state_c;
end

endmodule

`timescale 1ns/10ps
module csr_sparse_matrix_mult(input clk, input start, input reset, output
done, output [10:0] clock_count);

// 1st stage of pipeline, inputs are registered and edge detected
wire istart, ireset;      // pipeline inpts to module

// control lines for srom memory for input matrix
wire [5:0]rd_addr_values;
wire [7:0]rd_data_values;

wire [5:0]rd_addr_col_index;
wire [3:0]rd_data_col_index;

wire [4:0]rd_addr_row_index1;
wire [7:0]rd_data_row_index1;
wire [4:0]rd_addr_row_index2;
wire [7:0]rd_data_row_index2;

// control lines for srom vector input
wire [3:0]rd_addr_X;
wire [7:0]rd_data_X;

// control lines for macc4 pipelined module
wire [7:0]XA;
wire [7:0]YA;

```

```

wire [7:0]addrA;    // for macc A

wire macc_clear;           // clea accumulators or aumulate
wire wr_buffer;           // write macc outputs to buffers
wire rd_buffer;

// these connec the macc4 pipelined module to the synch write ram
wire [7:0]out_addr;
wire [18:0]out_buf;       // address and data for the
result memory
wire write_en;           // pipelined write to result
memory comand line

// synch write memory lines
wire [7:0]wr_addr;
wire [18:0]wr_data;       // address and data for the
result memory
wire wr_en;              // pipelined write to result memory
comand line

// control module inputs
wire [7:0]irow_i;
wire [7:0]irow_ipius1;
wire [3:0]icol_j;

// control module inputs/outputs
wire [7:0]caddr_result;   // address for result of macc A

wire cmacc_clear;         // clea accumulators or aumulate
wire cwr_buffer;         // write macc outputs to buffers
wire csave_results;       // activate write buffers
controller

wire crd_buffer;

wire [5:0]crd_addr_values; // address of value on aray
wire [5:0]crd_addr_col_index; // address of column index
wire [4:0]crd_addr_row_index1; // address of row index 1
wire [4:0]crd_addr_row_index2; // address of row index 2
wire [3:0]crd_addr_X;       // address of vector element

// instantiate modules
// edge detection for reset
edge_detection reset_edge(
    clk,
    reset,

```

```

        ireset
    );

    // edge detection for start
    edge_detection start_edge(
        clk,
        start,
        istart
    );

    // input matrix
    values v(clk, rd_addr_values, rd_data_values);        // read port 2

    row_index r_i(clk, rd_addr_row_index1, rd_data_row_index1,
rd_addr_row_index2, rd_data_row_index2);        // read port 2

    col_index c_i(clk, rd_addr_col_index, rd_data_col_index);

    // input vector
    vectorvalues x(clk, rd_addr_X, rd_data_X);        // read port 2

    // buffered maccs
    buffered_macc macc( clk,
        XA, YA, addrA,
        macc_clear, rd_buffer, wr_buffer,
        out_addr, out_buf, write_en);

    // results memory
    csr_synchwrite_ram RAMOUTPUT(clk, wr_addr, wr_data, wr_en);

    // control module
    csr_smv_mult_control mmc(clk,
        istart,
        ireset,
        irow_i,
        irow_iplus1,
        icol_j,
        caddr_result,
        cmacc_clear,
        cwr_buffer,
        csave_results,
        crd_addr_values,
        crd_addr_col_index,
        crd_addr_row_index1,
        crd_addr_row_index2,
        crd_addr_X,

```

```

        done,
        clock_count);

// write results control module
csr_smv_write_results_control wrc(clk, ireset, csave_results, crd_buffer);

// connect modules

// connect control module to memory for matrix and vector
assign rd_addr_values = crd_addr_values;
assign rd_addr_col_index = crd_addr_col_index;
assign rd_addr_row_index1 = crd_addr_row_index1;
assign rd_addr_row_index2 = crd_addr_row_index2;
assign rd_addr_X = crd_addr_X;

// connect row and column memory reads to control module
assign irow_i = rd_data_row_index1;
assign irow_iplus1 = rd_data_row_index2;
assign icol_j = rd_data_col_index;

// connect input matrix A & B to maccs
assign XA = rd_data_values;
assign YA = rd_data_X;

// connect output addresses for accum into results
assign addrA = caddr_result; // address where to
store results for macc A

// connect command lines for maccs
assign macc_clear = cmacc_clear; // clear accumulators or
aaccumulate
assign wr_buffer = cwr_buffer ; // write macc
outputs to buffers
assign rd_buffer = crd_buffer; // read one of the
four buffers and then send to memory

// connect maccs to results memory
assign wr_addr = out_addr; // address to write
results to
assign wr_data = out_buf; // data to write to
results
assign wr_en = write_en; // write enable command,
from maccs module

// print pipeline stage 0
// always @(posedge clk) begin
// $display("Pipeline Stage 0 +++++");

```

```
//          $display("Pipeline Stage 0: istart = %d", istart);
//          $display("Pipeline Stage 0: ireset = %d", ireset);
//      end
endmodule
```

Figure 5.1.1 Verilog code for implementation of Sparse matrix-vector Multiplication (SpMV)

Part 5.2 Testbench for Sparse matrix-vector Multiplication (SpMV)

```
`timescale 1ns/10ps
// Test bench module
module tb_lab6_extra;

// Input Array
////////////////////////////////////
//          Test Bench Signals          //
////////////////////////////////////
reg clk;
integer i,j,k;

// Matrices
reg signed [7:0] matrixA [255:0];
reg signed [7:0] vectorX [15:0];
reg signed [18:0] matrixC [255:0];

// Comparison Flag
reg comparison;

////////////////////////////////////
//          I/O Declarations          //
////////////////////////////////////
// declare variables to hold signals going into submodule
reg start;
reg reset;

// Misc "wires"
wire done;
wire [10:0] clock_count;

////////////////////////////////////
//          Submodule Instantiation          //
////////////////////////////////////
csr_sparse_matrix_mult DUT
(
    .clk    (clk),
    .start  (start),
```

```

        .reset (reset),
        .done      (done),
        .clock_count (clock_count)
    );

```

```
initial begin
```

```

//*****
// CHANGE .TXT FILE NAMES TO MATCH THE ONES USED IN
// YOUR MEMORY MODULES

// Initialize Matrices
$readmemb("ram_sparse_init.txt",matrixA);
$readmemb("ram_vector_init.txt",vectorX);

//*****

////////////////////////////////////
//                               Perform Test                               //
////////////////////////////////////
reset <= 1'b1;
start <= 1'b0;
clk <= 1'b0;
repeat(2) @(posedge clk);
reset <= 1'b0;
repeat(2) @(posedge clk);
start <= 1'b1;
repeat(1) @(posedge clk);
start <= 1'b0;

// -----
// Wait for done or timeout
fork : wait_or_timeout
begin
    repeat(250) @(posedge clk);
    disable wait_or_timeout;
end
begin
    @(posedge done);
    disable wait_or_timeout;
end
join
// End Timeout Routing
//-----

////////////////////////////////////
//                               Verify Computation                               //
////////////////////////////////////

```

```

// Print Input Matrices
$display("\n Matrix A");
for(i=0;i<16;i=i+1) begin

$display(matrixA[i],matrixA[i+16],matrixA[i+32],matrixA[i+48],matrixA[i+64],matrixA[i+80],matrixA[i+96],matrixA[i+112],matrixA[i+128],matrixA[i+144],matrixA[i+160],matrixA[i+176],matrixA[i+192],matrixA[i+208],matrixA[i+224],matrixA[i+240]);
end

$display("\n Vector X");

$display(vectorX[0],vectorX[1],vectorX[2],vectorX[3],vectorX[4],vectorX[5],vectorX[6],vectorX[7],vectorX[8],vectorX[9],vectorX[10],vectorX[11],vectorX[12],vectorX[13],vectorX[14],vectorX[15]);

// Generate Expected Result
for(i=0;i<16;i=i+1) begin
    for(j=0;j<16;j=j+1) begin
        matrixC[16*j+i] = matrixA[16*j + i]*vectorX[j];
    end
end

// Display Expected Result
$display("\nExpected Result");
for(i=0;i<16;i=i+1) begin

$display(matrixC[i],matrixC[i+16],matrixC[i+32],matrixC[i+48],matrixC[i+64],matrixC[i+80],matrixC[i+96],matrixC[i+112],matrixC[i+128],matrixC[i+144],matrixC[i+160],matrixC[i+176],matrixC[i+192],matrixC[i+208],matrixC[i+224],matrixC[i+240]);
end

// Display Output Matrix
$display("\nGenerated Result");
for(i=0;i<16;i=i+1) begin

$display(DUT.RAMOUTPUT.mem[i],DUT.RAMOUTPUT.mem[i+16],DUT.RAMOUTPUT.mem[i+32],DUT.RAMOUTPUT.mem[i+48],DUT.RAMOUTPUT.mem[i+64],DUT.RAMOUTPUT.mem[i+80],DUT.RAMOUTPUT.mem[i+96],DUT.RAMOUTPUT.mem[i+112],DUT.RAMOUTPUT.mem[i+128],DUT.RAMOUTPUT.mem[i+144],DUT.RAMOUTPUT.mem[i+160],DUT.RAMOUTPUT.mem[i+176],DUT.RAMOUTPUT.mem[i+192],DUT.RAMOUTPUT.mem[i+208],DUT.RAMOUTPUT.mem[i+224],DUT.RAMOUTPUT.mem[i+240]);
end

// Test if the two matrices match
comparison = 1'b0;
for(i=0;i<16;i=i+1) begin

```

```

        for(j=0;j<16;j=j+1) begin
            if (matrixC[16*i+j] != DUT.RAMOUTPUT.mem[16*i+j]) begin
                $display("Mismatch at indices [%1.1d,%1.1d]",j,i);
                comparison = 1'b1;
            end
        end
    end
end
if (comparison == 1'b0) begin
    $display("\nsuccess :");
end

$display("Running Time = %d clock cycles",clock_count);

$stop; // End Simulation
end

// Clock
always begin
    #10;          // wait for initial block to initialize clock
    clk = ~clk;
end

endmodule

```

Figure 5.2.1 Verilog code for testbench for Sparse matrix-vector Multiplication (SpMV)

```

# Matrix A
# 0-103 0 59 -12 0 0 0 0 124 0 0 0 0 0 0
# 0-106 111 0 0 -66 0 0 0 0 0 0 0 124 0 0
# 0 0 0 0 0 0-100 0 0 8 0 0 7 -14 0 0
# 0 0 0 0 0 25-123 63 0 0 10 0 0 0 0 0
# 0 0 -71 0 0 0 0 0 69 0 0 0 0 81 -82
# 0 0 0 91 0 -16 0 0 0 0 0-107 61 0 0
# 46 0 0 0 0 0 105 0 0 0 63 0 0 62 0 0
# 0 0 0-118 0 -71 0 0 0 0 0 -57 0 32 0 0
# 0 0 122 0 0 0 0 0 0 0 0 0 -20-116 -35
# 0 0-123 96 0 0 0 0 0 0 85 0 107 0 0 0
# 0 -33 0 0 0 0 0 0 -1-126 53 0 0 0 0 0
# 0 0 0 0 0 0 127 0 0 0 95 -9 0 -98 0
# 0-117 0 0 0-106 0 0 0 0 44 15 0 0 0 0
# 0 0 0 0 0 0 0 0 0 1 28 61 0 0 0 -63
# 0-113-124 0 0 -70 0 0 0 -87 0 0 0 0 0 0
# 0 0 0 0 0 0 0 48 54 0 0 0 -14 0 0 114
#
# Vector X
# -114-100 114 81 84 -44 91 -62 19 -35 -93-110 115 16 -76 -98
#

```

```

# Expected Result
# 0 10300 0 4779 -1008 0 0 0 0 -4340 0 0 0 0 0
# 0 10600 12654 0 0 2904 0 0 0 0 0 0 1984 0 0
# 0 0 0 0 0 0 -9100 0 0 -280 0 0 805 -224 0 0
# 0 0 0 0 0 -1100 -11193 -3906 0 0 -930 0 0 0 0 0
# 0 0 -8094 0 0 0 0 0 1311 0 0 0 0 -6156 8036
# 0 0 0 7371 0 704 0 0 0 0 0 0 -12305 976 0 0
# -5244 0 0 0 0 0 9555 0 0 0 -5859 0 0 992 0 0
# 0 0 0 -9558 0 3124 0 0 0 0 0 6270 0 512 0 0
# 0 0 13908 0 0 0 0 0 0 0 0 0 -320 8816 3430
# 0 0 -14022 7776 0 0 0 0 0 0 -7905 0 12305 0 0 0
# 0 3300 0 0 0 0 0 0 -19 4410 -4929 0 0 0 0 0
# 0 0 0 0 0 0 11557 0 0 0 0 -10450 -1035 0 7448 0
# 0 11700 0 0 0 4664 0 0 0 0 -4092 -1650 0 0 0 0
# 0 0 0 0 0 0 0 0 0 -35 -2604 -6710 0 0 0 6174
# 0 11300 -14136 0 0 3080 0 0 0 3045 0 0 0 0 0 0
# 0 0 0 0 0 0 0 -2976 1026 0 0 0 -1610 0 0 -11172
#
# Generated Result
# 0 10300 0 4779 -1008 0 0 0 0 -4340 0 0 0 0 0 0
# 0 10600 12654 0 0 2904 0 0 0 0 0 0 1984 0 0
# 0 0 0 0 0 0 -9100 0 0 -280 0 0 805 -224 0 0
# 0 0 0 0 0 -1100 -11193 -3906 0 0 -930 0 0 0 0 0
# 0 0 -8094 0 0 0 0 0 1311 0 0 0 0 -6156 8036
# 0 0 0 7371 0 704 0 0 0 0 0 0 -12305 976 0 0
# -5244 0 0 0 0 0 9555 0 0 0 -5859 0 0 992 0 0
# 0 0 0 -9558 0 3124 0 0 0 0 0 6270 0 512 0 0
# 0 0 13908 0 0 0 0 0 0 0 0 0 -320 8816 3430
# 0 0 -14022 7776 0 0 0 0 0 0 -7905 0 12305 0 0 0
# 0 3300 0 0 0 0 0 0 -19 4410 -4929 0 0 0 0 0
# 0 0 0 0 0 0 11557 0 0 0 0 -10450 -1035 0 7448 0
# 0 11700 0 0 0 4664 0 0 0 0 -4092 -1650 0 0 0 0
# 0 0 0 0 0 0 0 0 0 -35 -2604 -6710 0 0 0 6174
# 0 11300 -14136 0 0 3080 0 0 0 3045 0 0 0 0 0 0
# 0 0 0 0 0 0 0 -2976 1026 0 0 0 -1610 0 0 -11172
#
# success :)
# Running Time = 211 clock cycles

```

Figure 5.2.2 Results for testbench for Sparse matrix-vector Multiplication (SpMV)t

Part 5.3 Estimate the resource usage


Analysis & Synthesis Resource Usage Summary		
 <<Filter>>		
	Resource	Usage
1	Estimated Total logic elements	58
2		
3	Total combinational functions	56
4	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	21
2	-- 3 input functions	19
3	-- <=2 input functions	16
5		
6	▼ Logic elements by mode	
1	-- normal mode	46
2	-- arithmetic mode	10
7		
8	▼ Total registers	4
1	-- Dedicated logic registers	4
2	-- I/O registers	0
9		
10	I/O pins	149
11		
12	Embedded Multiplier 9-bit elements	0

Figure 5.3.1 Quartus Resource Usage Report

Part 5.4 Performance estimation

Using the number of clock cycles we measured to calculate the matrix multiplication of 211 clock cycles at a frequency of 50 MHz, the circuit does the calculation in 0.00422s.

Considering it was a 16x16 matrix, the sparsity factor of just 25% helped reduce the calculation using the Sparse matrix-vector Multiplication (SpMV). The bottlenecks that we have in this circuit are:

- Reading matrix elements from memory two by two only.
- Being able to calculate only one result at a time
- Writing one result at a time to the result matrix

Discussion

In Part I of this lab we constructed and tested a Multiplier Accumulator to be used as a basis for creating matrix multiplication circuits. Using a simple implementation we were able to get a relatively fast circuit with 4 elements: a multiplier, an adder, a 19-bit wide 2 to 1 multiplexer, and a register with 19 bits.

In Part II of this lab, we constructed a base circuit to multiply two 8x8 matrices using just one MACC unit with single read and single write memories. We established a base performance of 1088 clock cycles at 50 MHz to finish the computation. The testbench ran the circuit and validated the results with a slow gold standard implementation of the matrices multiplication.

In Part II we used a two-port reading memory to get the values from matrix A and a single-port memory to read the values from matrix B. This allowed us to include more parallelism by using two MACC units to calculate two results simultaneously as we had two inputs simultaneously read from matrix A. We had to add a two-results buffer before we could write to memory as the memory supported only one simultaneous write operation. All of that allowed us to increment the performance by a little over 3 times.

In Part 4, we used two-port reading memory for both input matrices but we were constrained to a single write output matrix. To take this disparity and put it to our advantage we pipelined the circuit into 5 stages and used 4 MACCs to calculate 4 results simultaneously. We had to include a 4 results buffer before storing the results in memory and we build an extra simple controller to take care of these operations, freeing the main controller from this task. The main controller takes care of retrieving the matrices data and making the calculations then passes control over to the write controller to store the data in the final memory destination from the buffers. All of this allowed us to double the performance of the previous circuit.

In the extra credit, we used the pipelined model from part 4 and adapted it to use the CSR formatted Sparse matrix. We included parallelism reading the matrix. With this configuration on a 16x16 matrix at 35% sparsity multiplied by a 16x1 vector, we got a run time of 211 clock cycles.

Conclusion

In this lab, we were able to test implementations that allowed us to improve performance using the concepts of parallelism and pipelining. From Part II to Part II we incremented performance by just adding some parallelism to the circuit improving performance by 3 times.

As we further added more parallelism and then pipelining to our design in Part IV, we were able to again double our performance over the circuit, and 7 times the performance of the original circuit. The first parallelism we introduces reduced the time taken to perform the calculations at a faster pace than we anticipated, as we were only expecting it to be half the time. The gains from more parallel elements were constrained by the one-write per cycle memory which hindered the final performance, although the pipelining and buffering helped with this as it allowed the first states of the pipeline to calculate as fast as possible while the last stages of the pipeline took care of the writing to the final memory.

Lastly in the extra credit part, we saw how the characteristics of the data, in this case, the sparsity of it, can be taken into account to establish data storing models and data processing models that can do efficient calculations. We included some parallelism and pipelining in this last part and we think there is room for improvement maybe using more parallel elements in the design architecture.