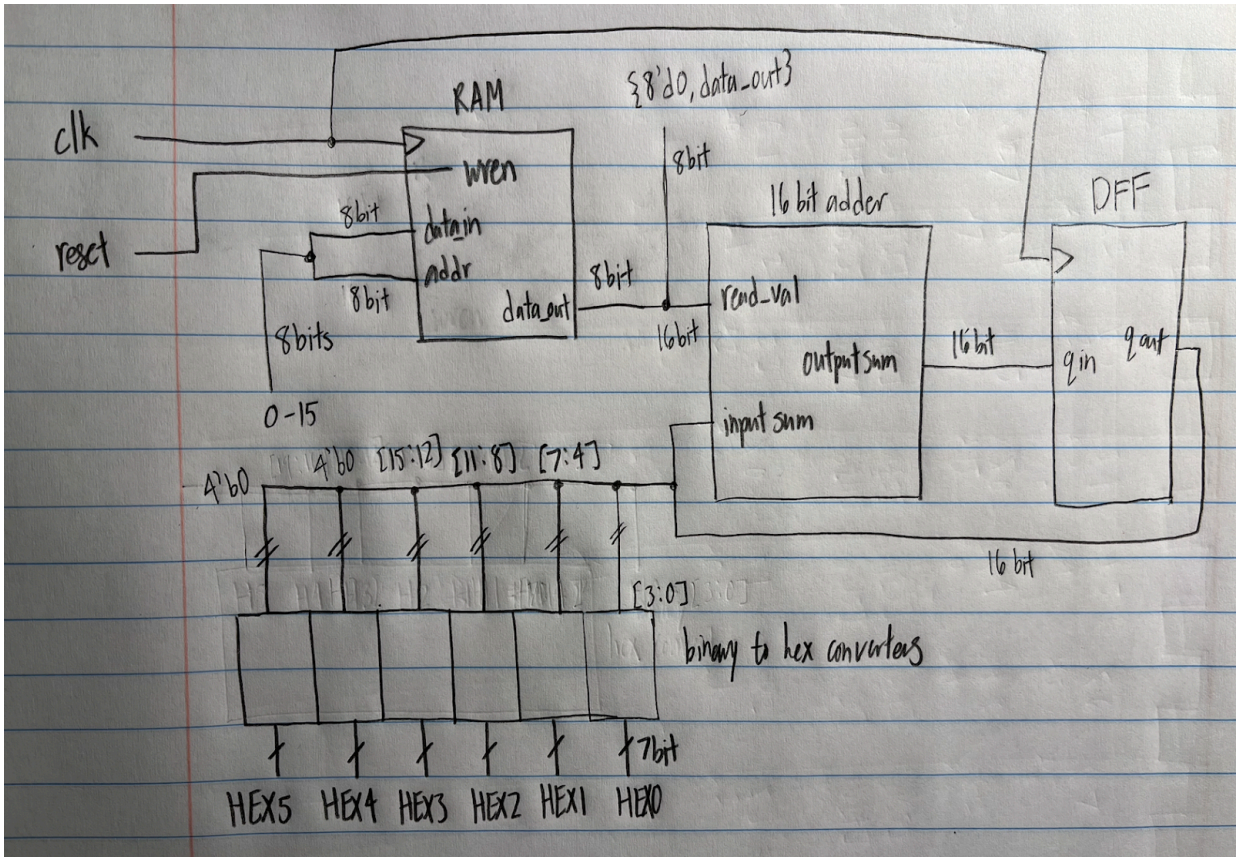
Lab 2

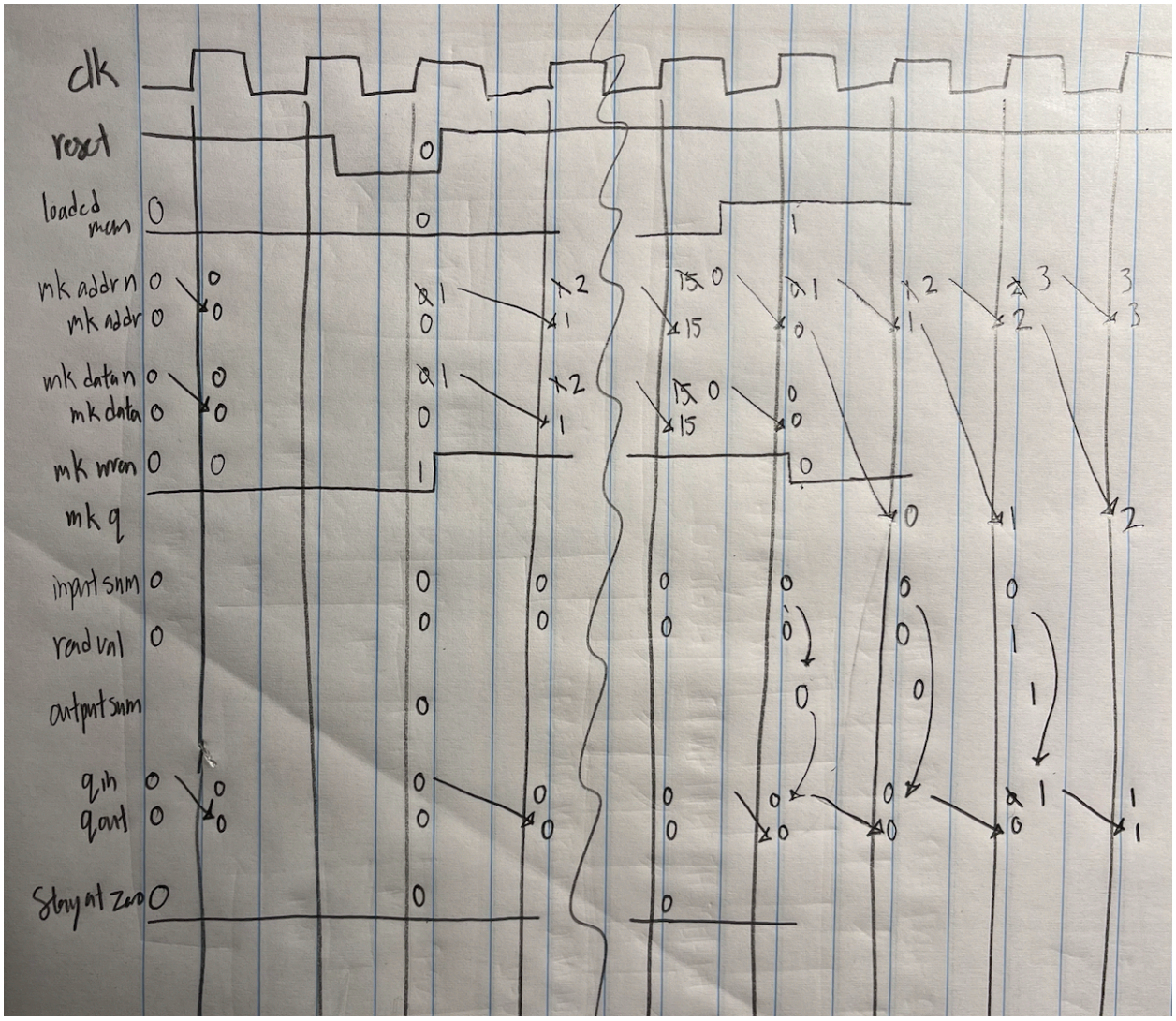
1. Prelab - no work to show

2. M10K Memory-Based Accumulator

Block diagram of my code:



Timing diagram



Notes on code:

- Mk_wren is a signal I defined to let me split the write and read parts of the process
    - It is set off initially by the input reset button, and flips once writing to the ram is done, and waits for the reset button again
    - I used this in combination with my loaded_mem signal to determine whether I am waiting for the first reset press, or if I am done loading values into the ram
- I used a separate always statement for the DFF input since I was not sure when the adder would finish
- Struggles:
    - I am not sure that I was supposed to hardcode 0-15 as ram inputs, and I used my adders from eec 180, so I am not sure if those are also done correctly
    - Generally not sure that I understood the question

3. Understanding Matrix Multiplication in Matlab

- I used for loops to more efficiently loop through the input matrices
- The bottom of the code shows my function for xoring the values in the final matrix

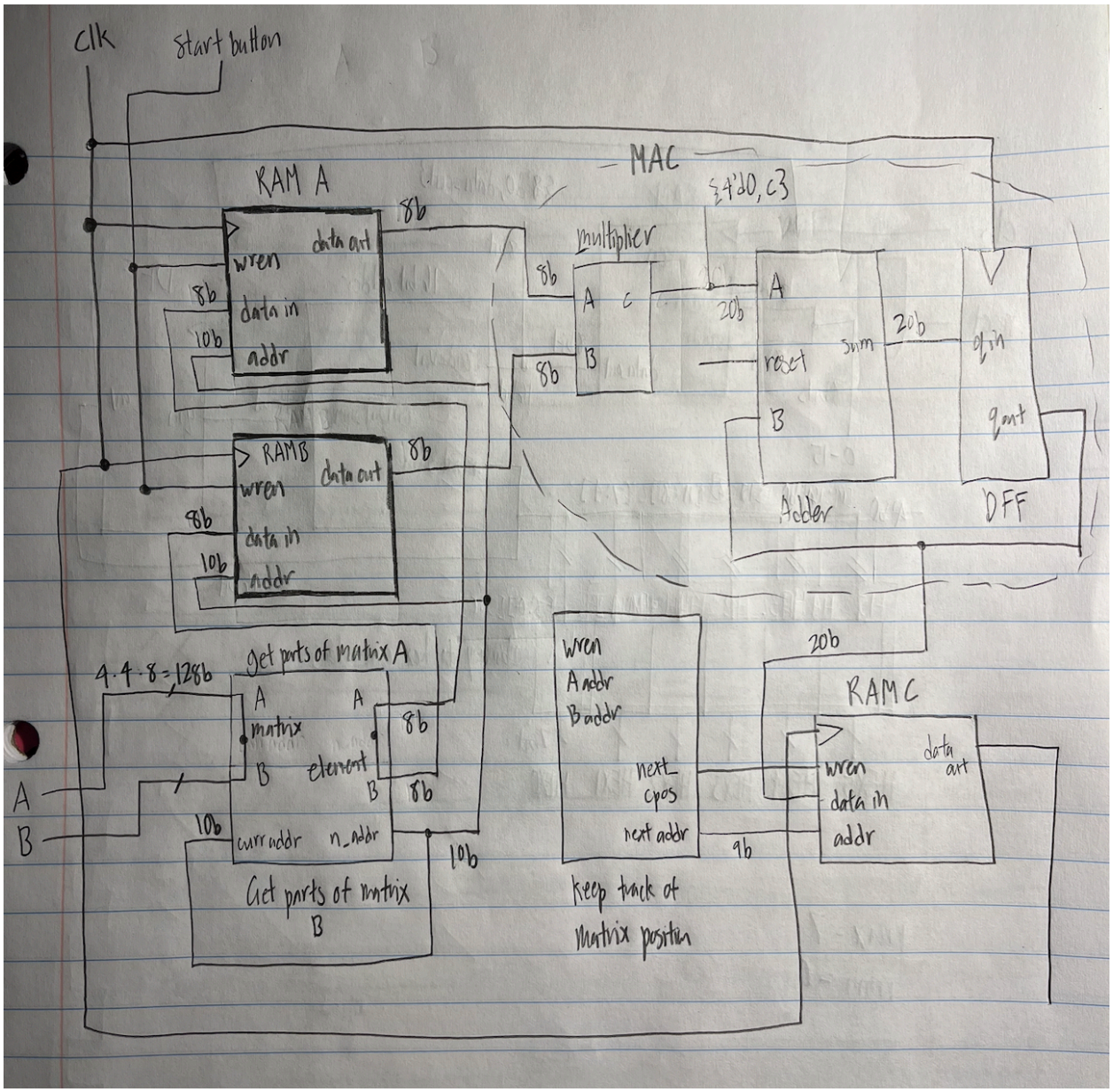
CODE:

```
%part 2 code provided in lab
a = [1 2 3 4 ; 5 6 7 8 ; 9 10 11 12 ; 13 14 15 16]
 b = [0 0 0 0 ; 1 1 1 1 ; 2 2 2 2 ; 3 3 3 3]
 c = [0 1 2 3 ; 0 1 2 3 ; 0 1 2 3 ; 0 1 2 3]

 d = round(rand(4) * 100)
 dd = round(rand(8) * 100)
 ddd = round(rand(8) * 100)

 a * b
 four_by_four = fxf(a, b)

 dd * ddd
 eight_by_eight = exe(dd, ddd)
 %a * c
 %a * d

 matrix_single_val = matrixXOR(four_by_four)

 %c=a*b,  multiply rows of a and cols of b and put into position of c where
 %row and col  intersect



function f  = fxf(x, y)
    f = zeros(4, 4);
    for i = 1:size(x,1)
        for j=1:size(x,1)
            f(i,j) = x(i,1)*y(1,j) + x(i,2)*y(2,j) + x(i,3)*y(3, j) + x(i,4)*y(4,j);
        end
    end
    return
end

function f  = exe(x, y)
    f = zeros(8, 8);
    for i = 1:size(x,1)
        for j=1:size(x,1)
```

```
        f(i,j) = x(i,1)*y(1,j) + x(i,2)*y(2,j) + x(i,3)*y(3, j) + x(i,4)*y(4,j)+ x(i,5)*y(5,j)+ x(i,6)*y(6,j)+
x(i,7)*y(7,j)+ x(i,8)*y(8,j);
      end
   end
   return
end



function tot_xor = matrixXOR(m)
% linnk  to xor calculator = https://xor.pw/#
   tot_xor = 0;         %xor(a, 0) = a
   for i = 1:size(m,1)
      for j=1:size(m,1)
         tot_xor = bitxor(m(i,j), tot_xor);
      end
   end

   return
end
```

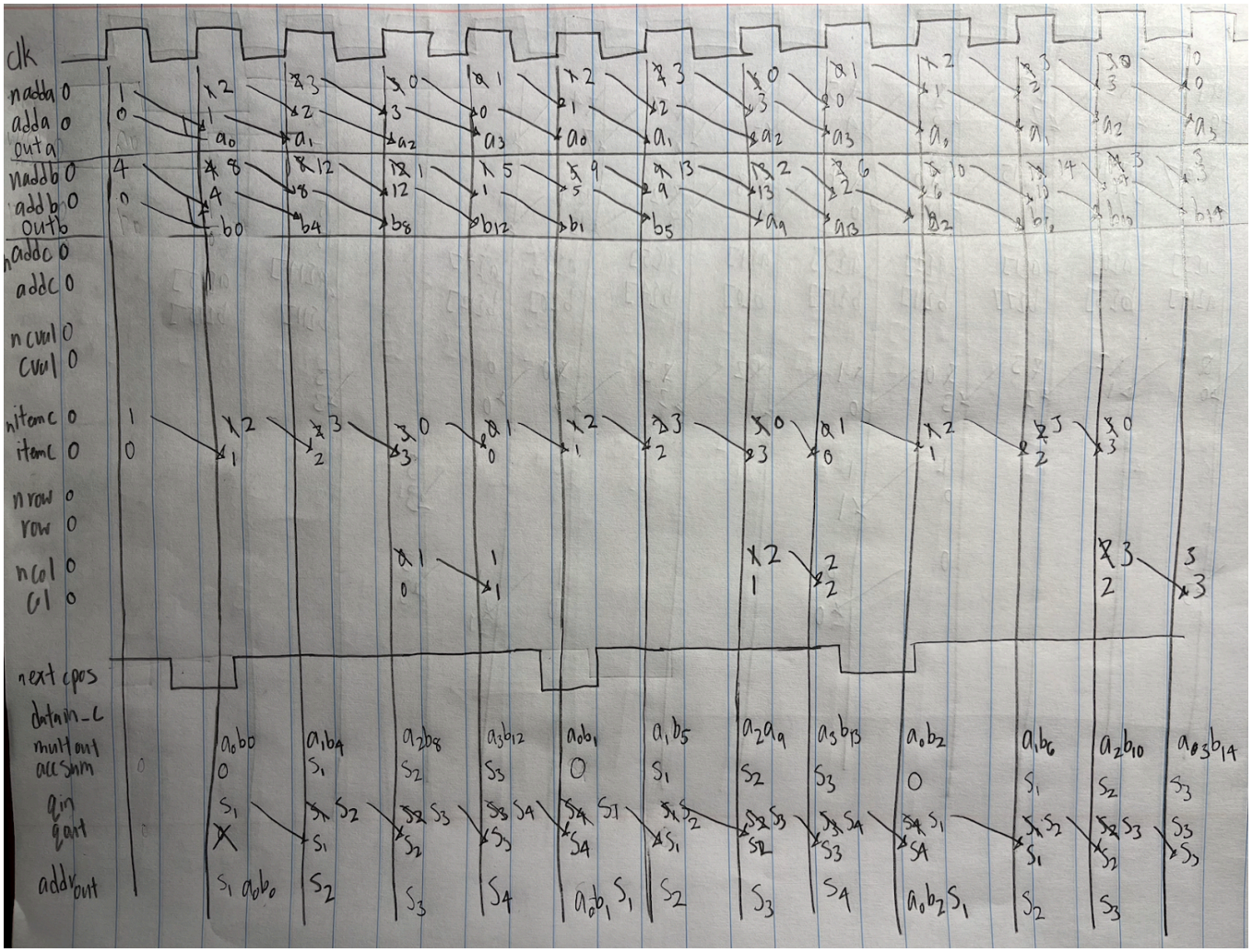
4. Matrix Multiplier Processor Using One MAC

Block diagram

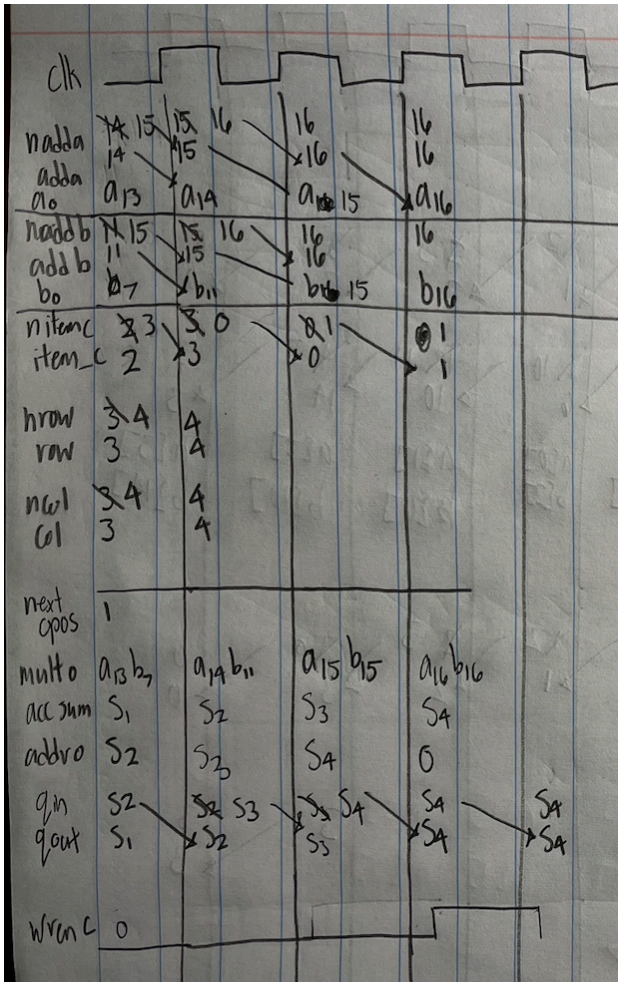


- Inputs for rams A and B are not directly connected to the module inputs
    - In my code I added another section that allows me to split the reading and writing for the rams into two separate sections
    - The right side of the diagram shows the multiplication process with the MAC

Timing diagram of the multiplying portion of the process:



- Shows the timing right after reading in A and B into rams is complete
- The bottom portion shows the MAC adding the multiplied values from above to a total sum
    - When the next_cpos signal goes from 1->0, the sum is reset; this is done when the ram reads for A and B show the correct final values to multiply for a given row/column



- This photo jumps to the end when we are multiplying the last matrix values A15 and B15
- Assuming each ram is structured in this way where the numbers represent indices of the ram storing the matrix

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- I designated index 16 for both rams to be 0 so that I can add 0 to the sum continuously and not mess with the sum while the ram reads for A and B are still processing and the MAC is waiting

Notes on code:

- I assumed the input matrices were flattened out to be consecutive elements based on rows and then columns

- I made the module twenty_bit_ram for output C that is the same as the basic_ram module taken from lecture notes, but with a larger bit width and sizing based on information from lecture
- I made a OneMac module to represent the multiply accumulator
- I split the process into writing A and B matrices into rams, and then reading information from them and putting it into the MAC
    - To read the right values from the matrices, I went through the final C matrix in row order; I multiplied every B column by a given A row before incrementing the A row
- Struggles:
    - I struggled with understanding the goal of this part of the project, and I ran out of time before I could implement attaching the output to the final C matrix
    - I drew out timing diagrams to show what I thought my code was doing, but I am not sure that they are accurate
    - I made a separate module to represent the MAC that was pictured in the lab manual, but I wasn't sure if I was supposed to use the accumulator we made in the first section of this lab since I implemented that to do a summation of 0-15

Verilog code

2.M10K Memory-Based Accumulator

Accumulator.v

```
module Accumulator (
        input clk, reset,
//      output we, loaded,
//      output [9:0] mkadd,
//      output [7:0] mkdata, mkq,
//      output [15:0] qout, outputsum,
        output [6:0] H0, H1, H2, H3, H4, H5
);


        //have 16bits read at a time = 2 addresses in M10k

        reg loaded_mem = 1'b0;

        reg [9:0] mk_addr = 10'd0;
        reg [9:0] mk_addr_n = 10'd0;
        reg [7:0] mk_data = 8'd0;
        reg [7:0] mk_data_n = 8'd0;
        reg mk_wren = 1'b0;
        wire [7:0] mk_q;                        //concatenate to add bitwidth for adder; read_val =
{8'd0,mk_q}

        reg [15:0] input_sum = 16'd0;
        reg [15:0] read_val = 16'd0;
        wire [15:0] output_sum;

//      reg carry_i= 1'd0;
//      wire carry_o;

        reg [15:0] q_in = 16'd0;
        reg [15:0] q_out = 16'd0;

        reg stay_at_zero = 1'd0;


        assign  we = mk_wren;
```

```
//		assign loaded = loaded_mem;
//		assign mkadd = mk_addr;
//		assign mkdata = mk_data;
//		assign mkq = mk_q;
//		assign qout = q_out;
//		assign  outputsum = output_sum;


//		MTenK mk(
//			//inputs
//			.address(mk_addr), .clock(clk), .data(mk_data),
//			.wren(mk_wren),
//			//output
//			.q(mk_q)
//		);

		basic_ram br(
			//input
			 .clk(clk), .wr_en(mk_wren),
			.addr(mk_addr), .data_in(mk_data),
			//output
			.q(mk_q)
		);


		FA_sixteen fasix(
			//input
			.A(input_sum), .B(read_val),
			//output
			.sum(output_sum)
		);

		HexDisplay hd0( .binary_in(q_out[3:0]), .h0(H0) );
		HexDisplay hd1( .binary_in(q_out[7:4]), .h0(H1) );
		HexDisplay hd2( .binary_in(q_out[11:8]), .h0(H2) );
		HexDisplay hd3( .binary_in(q_out[15:12]), .h0(H3) );
		HexDisplay hd4( .binary_in(4'b00), .h0(H4) );
		HexDisplay hd5( .binary_in(4'b00), .h0(H5) );

//////////////////////

		always @(posedge clk) begin
			if(~reset)  begin
			//if reset is pressed, write enable is activated for the m10k
```

```
                mk_wren <= 1'b1;
            end
            else if(loaded_mem) begin
                mk_wren <= 1'b0;
            end

            mk_addr <= mk_addr_n;
            mk_data <= mk_data_n;
            q_out <= q_in;
        end

////////////////////

        always @(mk_wren, mk_q, mk_addr) begin

            if(mk_wren) begin
                input_sum = 16'd0;
                read_val = 16'd0;
//              q_in = 16'd0;
                stay_at_zero = 1'b0;

                //when write  enable is activated, fill memory
                if(mk_addr < 8'd15) begin
                    mk_addr_n = mk_addr+10'd01;
                    mk_data_n = mk_data+8'd01;
                    loaded_mem = 1'b0;

                end else if (mk_addr == 8'd15) begin
                    mk_addr_n = 10'd0;
                    mk_data_n = 8'd0;
                    loaded_mem = 1'b1;
                end
            end

            else if(~mk_wren) begin
                if(loaded_mem == 1'b1) begin
                    //read in the value to add,  update next m10k addr
                    mk_data_n = 8'd0;

                    if(stay_at_zero) begin
                        mk_addr_n = 10'd0;
                        input_sum = q_out;
                        read_val = {8'd0, mk_q};
            //              q_in = output_sum;
```

```
                                stay_at_zero = 1'b1;

                        end
                        else if(mk_addr == 8'd0) begin
                                mk_addr_n = mk_addr+10'd01;
                                read_val = 16'd0;
                                input_sum = 16'd0;
                //                      q_in = q_out;
                                stay_at_zero = 1'b0;

                        end
                        else if(mk_addr < 8'd15) begin
                                mk_addr_n = mk_addr+10'd01;
                                read_val = {4'd0, mk_q};
                                input_sum = q_out;

                //                      q_in  =  output_sum;
                                stay_at_zero = 1'b0;

                        end
                        else if(mk_addr == 8'd15) begin
                                mk_addr_n = 10'd0;
                                read_val = {4'd0, mk_q};
                                input_sum = q_out;
                //                      q_in  =  output_sum;
                                stay_at_zero = 1'b1;
                        end
                end
        end
    end

    always @(output_sum) begin
        q_in = output_sum;
    end

endmodule
```

Basic_ram.v - copied from lecture

```
module basic_ram (
	input clk, wr_en,
	input [9:0] addr,
	input [7:0] data_in,
	output [7:0] q
);

	reg [7:0] mem [1023:0];
	reg [7:0] qout;
	assign q = qout;
	// To initialize the RAM, Quartus supports initialization
	// which normal RAMs and synthesis do not support.
	// initial begin // mem[0] = 24'h03B03F;
	// mem[1] = 24'h000FFF; // mem[2] = 24'hBEBEBE;
	// ... // mem[1023] = 24'h726384;
	// end

	always @(posedge clk) begin
		if (wr_en) begin
			mem[addr] <= #1 data_in; // write mem
		end
		else begin
			qout <= #1 mem[addr]; // read mem
		end
	end
endmodule
```

FullAdder - taken from eec 180 notes

```
module FullAdder(
	input a, b, c,
	output cout, sum
);
//borrowed from my code in 180

assign sum = a^b^c;
assign cout = (a&b)|(a&c)|(b&c);

endmodule
```

FA_sixteen - taken from eec 180 notes

```
module FA_sixteen(
	input [15:0] A, B,
```

```
//          input cin,
//          output cout,
            output [15:0] sum
);
//borrowed from my code in 180
//cout = overflow bit, unsigned adding

            wire [16:0] carries;
            assign carries[0] = 1'b0;


            genvar i;
            generate
                        for(i=0; i<16; i=i+1) begin: gen_name
                                    FullAdder fa(A[i], B[i], carries[i], carries[(i+1)], sum[i]);
                        end
            endgenerate


//          assign cout = carries[16];


endmodule
```

HexDisplay

```
module HexDisplay(
	input [3:0] binary_in,
	output [6:0] h0
);

		reg [6:0] display = 7'd0;
		assign h0 = display;

//0 = on
	always @(*) begin
		case (binary_in)
			4'd000: display = 7'b1000000;
			4'd001: display = 7'b1111001;
			4'd002: display = 7'b0100100;
			4'd003: display = 7'b0110000;
			4'd004: display = 7'b0011001;
			4'd005: display = 7'b0010010;
			4'd006: display = 7'b1111101;
			4'd007: display = 7'b1111000;
			4'd008: display = 7'b00;
			4'd009: display = 7'd0011000;
			4'd010: display = 7'd0001000;
			4'd011: display = 7'd0000011;
			4'd012: display = 7'd0111001;
			4'd013: display = 7'd0100001;
			4'd014: display = 7'd0000110;
			4'd015: display = 7'd0001110;
		endcase
	end


endmodule
```

Lab2_testbench - used to test accumulator

```
module lab2_testbench;
	reg clk;
	reg [0:10000] i;
	reg rst = 1'b1;
	wire write_en, loaded_m;
	wire [9:0] mk_add;
	wire [7:0] mk_data, mk_q;
	wire [15:0] qo, outsum;
	wire [6:0] h0, h1, h2, h3, h4, h5;

	initial begin
		clk=0;
		forever begin
			#10
			clk = ~clk;

		end
	end

	Accumulator ac(clk, rst, write_en, loaded_m, mk_add, mk_data, mk_q, qo, outsum, h0,
h1, h2, h3, h4, h5);


	initial begin
		for(i=0; i<10000; i = i+1) begin
			#5
			if(i==10) begin
				rst = 1'b0;
			end
			if(i==30) begin
				rst = 1'b1;
			end

		end


	end



endmodule
```

4. Matrix Multiplier Processor Using One MAC

FA_twenty

```
module FA_twenty(
	input [19:0] A, B,
//	input cin,
//	output cout,
	output [19:0] sum
);
//borrowed from my code in 180
//cout = overflow bit, unsigned adding

	wire [20:0] carries;
	assign carries[0] = 1'b0;


	genvar i;
	generate
		for(i=0; i<20; i=i+1) begin: gen_name2
			FullAdder fa(A[i], B[i], carries[i], carries[(i+1)], sum[i]);
		end
	endgenerate


//	assign cout = carries[16];


endmodule
```

Twenty_bit_ram

```
module twenty_bit_ram(

	input clk, wr_en,
	input [8:0] addr,			//log2(512) = 9bit addresses
	input [19:0] data_in,
	output [19:0] q
);

	reg [19:0] mem [512:0];

	reg [19:0] qout;
	assign q = qout;
	// To initialize the RAM, Quartus supports initialization
	// which normal RAMs and synthesis do not support.
```

```
	// initial begin // mem[0] = 24'h03B03F;
	// mem[1] = 24'h000FFF; // mem[2] = 24'hBEBEBE;
	// ... // mem[1023] = 24'h726384;
	// end

	always @(posedge clk) begin
		if (wr_en) begin
			mem[addr] <= #1 data_in; // write mem
		end
		else begin
			qout <= #1 mem[addr]; // read mem
		end
	end


endmodule
```

OneMac

```
module OneMac (
	input clock, reset,
	input [7:0] ain, bin,

	output [19:0] sum
);



//adder input and output are both 16 bits, doesn't account for overflow
//will keep multiplying and adding the inputs every clock cycle
		// if you want to stop it, need to set ain or bin to 0 !!!!!!!!
		//changing ain and bin between  clock cycles will make the result show  up
			//at the next  posedge

	reg [15:0] mult_out = 16'd0;
	wire [15:0] adder_out;
	reg [19:0] qin = 20'd0;
	reg [19:0] qout = 20'd0;
	reg [19:0] acc_sum = 20'd0;
	reg rst_sig = 1'b1;

	assign sum = qout;


	FA_twenty add(
```

```
        //input
        .A(({4'd0, mult_out})), .B(acc_sum),
        //output
        .sum(adder_out)
    );



    always @(posedge clock) begin
        if(~reset) begin
            rst_sig <= 1'b0;
        end
        else begin
            rst_sig <= 1'b1;
        end
        qout <= qin;
    end

    always @(ain, bin, qout) begin
        if(~rst_sig) begin
            acc_sum = 20'd0;
        end
        else begin
            acc_sum = qout;
        end
        mult_out = ain*bin;

    end
    always @(adder_out) begin
        qin = adder_out;
    end




endmodule
```

MatrixMultFour

```
module MatrixMultFour(
        input clock, start,
        //flatten array into rows followed by each other {row1, row2, row3, row 4}
                //a11 = [7:0], a12 = [15:8]
        input [127:0] A, B,       //128 bits = 8*(4*4)

        //testbbench
//      output [3:0] curr_st, nxt_st,
//      output wrenab, wrenc, mac_rst,
//      output [9:0] addra, addrb,
//      output [8:0] addrc,
//      output [7:0] outa, outb,
//      output [19:0] outc,
//      output [1:0] cols, rows, items,
//      output [19:0] cdatain,
        ////////////////////

//      output [319:0] matrix_C,                //20*(4*4) = 320
        output  [19:0] final_xor,
        output [9:0] MC
);

        localparam [3:0]  wait_for_start = 4'b0000;
        localparam [3:0]  store_ab = 4'b0001;
        localparam [3:0]  read_and_mult = 4'b0010;
        localparam [3:0]  read_c = 4'b0100;



        reg [3:0]  state = wait_for_start;
        reg [3:0] n_state = wait_for_start;


        reg [1:0] item_count =2'b00;
        reg [1:0] n_item_count =2'b00;
        reg [1:0] row_count = 2'b00;
        reg [1:0] n_row_count = 2'b00;
        reg [1:0] col_count= 2'b00;
        reg [1:0] n_col_count= 2'b00;



        reg wren_ab = 1'b0;
```

```
        reg [9:0] addr_a = 10'd0;
        reg [9:0] n_addr_a = 10'd0;
        reg [7:0] datain_a;
        wire [7:0] out_a;

//      reg ram_wren_b = 1'b0;
        reg [9:0] addr_b = 10'd0;
        reg [9:0] n_addr_b = 10'd0;
        reg [7:0] datain_b;
        wire [7:0] out_b;

        reg wren_c = 1'b0;
        reg [8:0] addr_c = 9'd0;
        reg [8:0] n_addr_c = 9'd0;
        wire [19:0] datain_c;
        wire [19:0] out_c;

        reg next_cpos = 1'b1;


//      reg [319:0] C = 320'd0;
//      assign matrix_C = C;
        reg [19:0] xor_tot  = 20'd0;
        reg [19:0] xor_tot_n  = 20'd0;

        assign final_xor = xor_tot;

        reg [9:0] mult_count = 10'd0;
                reg[9:0] m_c = 10'd1;
        assign MC = mult_count;

////////////TESTBENCH
//      assign curr_st = state;
//      assign nxt_st = n_state;
//      assign  wrenab = wren_ab;
//      assign wrenc = wren_c;
//      assign mac_rst = next_cpos;
//      assign addra = addr_a;
//      assign addrb = addr_b;
//      assign addrc = addr_c;
//      assign  outa = out_a;
//      assign  outb = out_b;
//      assign outc = out_c;
//      assign cols = col_count;
```

```
//          assign  rows = row_count;
//          assign items = item_count;
//          assign cdatain = datain_c;
////////////



        basic_ram ram_a(
                //input
                .clk(clock), .wr_en(wren_ab),
                .addr(addr_a),                  //[9:0] = 10bits
                .data_in(datain_a),             //[7:0] = 8bits
                //output
                .q(out_a)                                       //[7:0] = 8bits
        );

        basic_ram ram_b(
                //input
                .clk(clock), .wr_en(wren_ab),
                .addr(addr_b),                  //[9:0] = 10bits
                .data_in(datain_b),             //[7:0] = 8bits
                //output
                .q(out_b)                                       //[7:0] = 8bits
        );




//NNEEDS TO BE 20BITS WIDE FOR DATAIN
        twenty_bit_ram ram_c(
                //input
                .clk(clock), .wr_en(wren_c),
                .addr(addr_c),                  //[9:0] = 10bits
                .data_in(datain_c),             //[7:0] = 8bits
                //output
                .q(out_c)                                       //[7:0] = 8bits
        );
```

```
OneMac mac1(
        .clock(clock), .reset(next_cpos),
        .ain(out_a), .bin(out_b),
        //output
        .sum(datain_c)
);


////////////////////////////////////////////////////////
// 1. when start is pressed, write A and B into memory
// 2. process 1 MAC per cycle


always @(posedge clock) begin
        if(~start) begin
                state  <= store_ab;
                wren_ab <= 1'b1;
        end
        else if(n_state == store_ab) begin
                state <= n_state;
                wren_ab <= 1'b1;
        end
        else begin
                state <= n_state;
                wren_ab <= 1'b0;
        end

        mult_count <= m_c;
        item_count <= n_item_count;
        row_count <= n_row_count;
        col_count <= n_col_count;
        addr_a <= n_addr_a;
        addr_b <= n_addr_b;
        addr_c <= n_addr_c;

        xor_tot  <= xor_tot_n;
end




always @(state, wren_ab, item_count, row_count, addr_a, addr_b, addr_c) begin
        case(state)
```

```
//////////////////////////////////////////////////////
                wait_for_start: begin
                        //still in start state waiting for start to be pressed
                        datain_a = 8'd0;
                        datain_b = 8'd0;
                        n_addr_a = 10'd0;
                        n_addr_b = 10'd0;

                        n_item_count = 2'd0;
                        n_row_count = 2'd0;
                        n_col_count = 2'd0;
                        next_cpos = 1'b1;
                        wren_c = 1'b0;
                        n_addr_c = 9'd0;

                        n_state = wait_for_start;
                end

//////////////////////////////////////////////////////

                store_ab : begin
                        case(addr_a)
                                10'd0: begin
                                        datain_a = A[7:0];
                                        datain_b = B[7:0];
                                        n_addr_a = addr_a + 10'd01;
                                        n_addr_b = addr_b + 10'd01;
                                        n_state = store_ab;
                                end
                                10'd1: begin
                                        datain_a = A[15:8];
                                        datain_b = B[15:8];
                                        n_addr_a = addr_a + 10'd01;
                                        n_addr_b = addr_b + 10'd01;
                                        n_state = store_ab;
                                end
                                10'd2: begin
                                        datain_a = A[23:16];
                                        datain_b = B[23:16];
                                        n_addr_a = addr_a + 10'd01;
                                        n_addr_b = addr_b + 10'd01;
                                        n_state = store_ab;
                                end
                                10'd3: begin
```

```
            datain_a = A[31:24];
            datain_b = B[31:24];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
        end
        10'd4: begin
            datain_a = A[39:32];
            datain_b = B[39:32];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
        end
        10'd5: begin
            datain_a = A[47:40];
            datain_b = B[47:40];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
        end
        10'd6: begin
            datain_a = A[55:48];
            datain_b = B[55:48];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
        end
        10'd7: begin
            datain_a = A[63:56];
            datain_b = B[63:56];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
        end
        10'd8: begin
            datain_a = A[71:64];
            datain_b = B[71:64];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
        end
        10'd9: begin
            datain_a = A[79:72];
            datain_b = B[79:72];
```

```
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
    end
    10'd10: begin
            datain_a = A[87:80];
            datain_b = B[87:80];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
    end
    10'd11: begin
            datain_a = A[95:88];
            datain_b = B[95:88];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
    end
    10'd12: begin
            datain_a = A[103:96];
            datain_b = B[103:96];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
    end
    10'd13: begin
            datain_a = A[111:104];
            datain_b = B[111:104];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
    end
    10'd14: begin
            datain_a = A[119:112];
            datain_b = B[119:112];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
            n_state = store_ab;
    end
    10'd15: begin
            datain_a = A[127:120];
            datain_b = B[127:120];
            n_addr_a = addr_a + 10'd01;
            n_addr_b = addr_b + 10'd01;
```

```
                                        n_state = store_ab;
                                end
                                10'd16: begin
                                        datain_a = 8'd0;
                                        datain_b = 8'd0;
                                        n_addr_a = 10'd0;
                                        n_addr_b = 10'd0;
                                        n_state = read_and_mult;

                                end

                        endcase

                        n_item_count = 2'd0;
                        n_row_count = 2'd0;
                        n_col_count = 2'd0;
                        next_cpos = 1'b1;
                        wren_c = 1'b0;
                        n_addr_c = 9'd0;
                end
////////////////////////////////////////////////////////

                read_and_mult: begin
                        m_c = mult_count +  10'd1;
                        if(addr_a == 10'd16) begin
                                n_addr_a = 10'd16;
                                n_addr_b = 10'd16;

                                n_item_count = item_count+2'd1;
                                n_row_count = row_count;
                                n_col_count = col_count;

                                next_cpos = 1'b1;

                                if(item_count == 2'd1) begin
                                        wren_c = 1'b1;
                                        n_addr_c = addr_c + 9'd01;

                                        n_state = read_and_mult;
                                end
                                else begin
                                        wren_c = 1'b0;
                                        if (item_count > 2'd1) begin
                                                n_state = read_c;
```

```
                        n_addr_c = 9'd0;
                end
                else begin
                        n_state = read_and_mult;
                        n_addr_c = addr_c;
                end
        end

end


else if(item_count == 2'd0) begin
        n_addr_a = addr_a+10'd01;
        n_addr_b = addr_b + 10'd04;

        n_item_count = item_count+2'd1;
        n_row_count = row_count;
        n_col_count = col_count;

        //reset accumulator
        next_cpos = 1'b0;
        wren_c = 1'b0;
        n_addr_c = addr_c;

        n_state = read_and_mult;
end

else if(item_count == 2'd3) begin

        if((row_count == 2'd3) && (col_count == 2'd3)) begin
                //donewith the whole thing
                //address 16 in matrices a and b = 0 value

                n_addr_a  = addr_a + 10'd01;
                n_addr_b = addr_b + 10'd01;
                n_row_count = row_count+2'd1;       // becomes 4
                n_col_count = col_count+2'd1;

        end else if(col_count == 2'd03) begin
                n_addr_a = addr_a + 10'd01;
                n_addr_b = 10'd0;
                n_row_count = row_count+2'd1;
                n_col_count = 2'd0;
```

```
                    end else begin
                        n_addr_a = addr_a - 10'd03;
                        n_addr_b = {8'd00, (col_count + 2'd1)};
                        n_row_count = row_count;

                        n_col_count = col_count+2'd1;
                    end
                    n_item_count = 2'd00;

                    next_cpos = 1'b1;
                    wren_c = 1'b0;
                    n_addr_c = addr_c;

                    n_state = read_and_mult;
                end

                else begin
                    n_addr_a = addr_a+10'd01;
                    n_addr_b = addr_b+ 10'd04;

                    n_item_count= item_count+2'd1;
                    n_row_count = row_count;
                    n_col_count = col_count;

                    next_cpos = 1'b1;

                    n_state = read_and_mult;

                    if((item_count == 2'd1) &&  ((row_count> 2'd0) || (col_count
> 2'd0))) begin
                        //write to c ram
                        wren_c = 1'b1;
                        n_addr_c = addr_c + 9'd01;
                    end
                    else begin
                        wren_c = 1'b0;
                        n_addr_c = addr_c;
                    end
                end

            end

////////////////////////////////////////////////////////
```

```
                        read_c: begin
                                if(addr_c < 9'd16) begin
                                        n_addr_c = addr_c + 9'd01;
                                        n_state = read_c;
                                end else if(addr_c == 9'd16) begin
                                        n_addr_c = 9'd00;
                                        n_state = wait_for_start;
                                end

                                wren_c = 1'b0;
                                datain_a = 8'd0;
                                datain_b = 8'd0;
                                n_addr_a = 10'd0;
                                n_addr_b = 10'd0;

                                n_item_count = 2'd0;
                                n_row_count = 2'd0;
                                n_col_count = 2'd0;
                                next_cpos = 1'b1;

                        end

////////////////////////////////////////////////////
                endcase
        end


        always @(out_c) begin
//              case(addr_c)
//              //xor_tot_n = xor_tot ^  out_c;
//                      9'd1: C[19:0] = out_c;
//                      9'd2: C[39:20] = out_c; // 20 bits
//                      9'd3: C[59:40] = out_c;
//                      9'd4: C[79:60] = out_c;
//                      9'd5: C[99:80] = out_c;
//                      9'd6: C[119:100] = out_c;
//                      9'd7: C[139:120] = out_c;
//                      9'd8: C[159:140] = out_c;
//                      9'd9: C[179:160] = out_c;
//                      9'd10: C[199:180] = out_c;
//                      9'd11: C[219:200] = out_c;
//                      9'd12: C[239:220] = out_c;
//                      9'd13: C[259:240] = out_c;
//                      9'd14: C[279:260] = out_c;
```

```
//                      9'd15: C[299:280] = out_c;
//                      9'd16: C[319:300] = out_c;
//              endcase
                if((addr_c  >= 9'd1) && (addr_c <= 9'd16) && (state == read_c)) begin
                        xor_tot_n = xor_tot ^  out_c;
                end
                else begin
                        xor_tot_n =  xor_tot;
                end

        end









endmodule
```