

# 8x8 Dadda Multiplier (MAC)

by

**Suman Mahato**

August 18, 2022

## Dependencies :

compilation was done using icarus-iverilog, this can be installed in ubuntu 18 and above by `sudo apt install iverilog` for other operating systems please goto [http://iverilog.wikia.com/wiki/Installation Guide](http://iverilog.wikia.com/wiki/Installation_Guide) to analyze the waveforms, gtkwave was used can be installed in ubuntu by `sudo apt install gtkwave` A small java program, GenLoops.java was used to generate test data and certain recurring assign statements. This doesn't pose any requirement on java runtime if you are using the txt files includes for the test benches. Verilog code uses constructs in verilog like generate loops , and parameters, you can refer to IEEE standard for verilog for details.

## Top level module :

An 8x8 dadda multiplier was designed and verified using verilog. The details of the top level module are as given below. A,B - 8 bit multiplicands M - previous result existing in Accumulator(16bits) RES - 17 bit output. submodules: gen part products - generate the partial products of the multiplication, A\*B as an 8x8 array. processing block - takes the partial products,M and does the dadda reduction adder 16 - 16 bit Carry select adder , test data was randomly generated using a small program and verified the circuit using the test bench - top level tb(file:top level tb.v). Test data was loaded from files ain.txt,bin.txt,m.txt and res.txt

- run.sh will compile all modules

// Codes for the top level module is given below :

```
module top_level(
    input [7:0] A,
    input [7:0] B,
    input [15:0] M,
    output [16:0] RES);

    genvar i;
    wire [7:0] P [7:0] ;
    wire [1:0] PRE[15:0];
    gen_part_products U1(A,B,P);

    processing_block U2(P,M,PRE);

































































    adder16 U3(PRE[1],PRE[0],1'b0,RES);
```

endmodule

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
p0								•	•	•	•	•	•	•	•	•
p1								•	•	•	•	•	•	•	•	
p2							•	•	•	•	•	•	•			
p3						•	•	•	•	•	•	•				
p4					•	•	•	•	•	•	•					
p5				•	•	•	•	•	•	•						
p6			•	•	•	•	•	•	•	•						
p7		•	•	•	•	•	•	•	•							

























































Capacity – 9

---


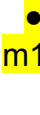

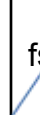
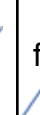

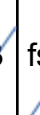
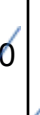







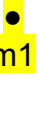

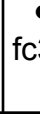
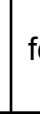
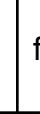
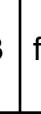
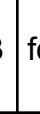
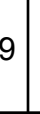
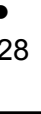
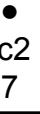
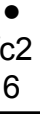
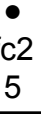
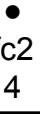
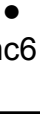

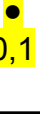
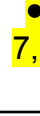
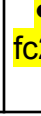
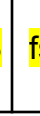

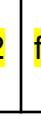


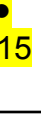



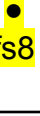

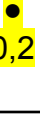
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
 m1 5	 m1 4	 m1 3	 m1 2	 m1 1	 m1 0	 fs5	 fs3	 fs1	 fs0	 hs0	 m4	 m3	 m2	 m1	 m0
	 7,6	 6,7	 5,7	 4,7	 fc5	 fc3	 fc1	 fc0	 hc0	 0,5	 0,4	 0,3	 0,2	 0,1	 0,0
		 7,6	 6,6	 5,6	 3,7	 fs6	 fs4	 fs2	 hs1	 1,4	 1,3	 1,2	 1,1	 1,0	
			 7,5	 6,5	 fc6	 fc4	 fc2	 hc1	 m6	 2,3	 2,2	 2,1	 2,0		
				 7,4	 fs7	 m9	 hs3	 hs2	 0,6	 3,2	 3,1	 3,0			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				• fc7	• 4,6	• hc3	• hc2	• m7	• 1,5	• m5	• 4,0				

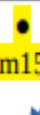

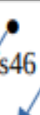
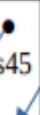

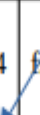
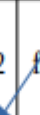



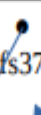

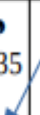



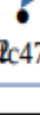
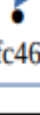
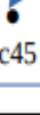
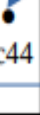
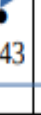
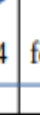

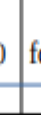
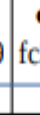
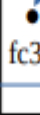
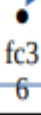
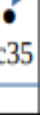
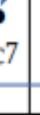



Cap -4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
 m15	 m1 4	 m13	 fs2 3	 fs2 1	 fs1 9	 fs17	 fs15	 fs1 3	 fs1 1	 fs9	 fs8	 hs4	 m2	 m1	 m0
	 7,6	 fc23	 fc2 1	 fc1 9	 fc1 7	 fc15	 fc13	 fc1 1	 fs9	 fc8	 hc4	 m3	 0,2	 0,1	 0,0
		 6,7	 m1 2	 fs2 2	 fs2 0	 fs18	 fs16	 fs1 4	 fs1 2	 fs1 0	 hs5	 0,3	 1,1	 1,0	
		 7,6	 fc2 2	 fc2 0	 fc1 8	 fc16	 fc14	 fc1 2	 fc1 0	 hc5	 m4	 1,2	 2,0		

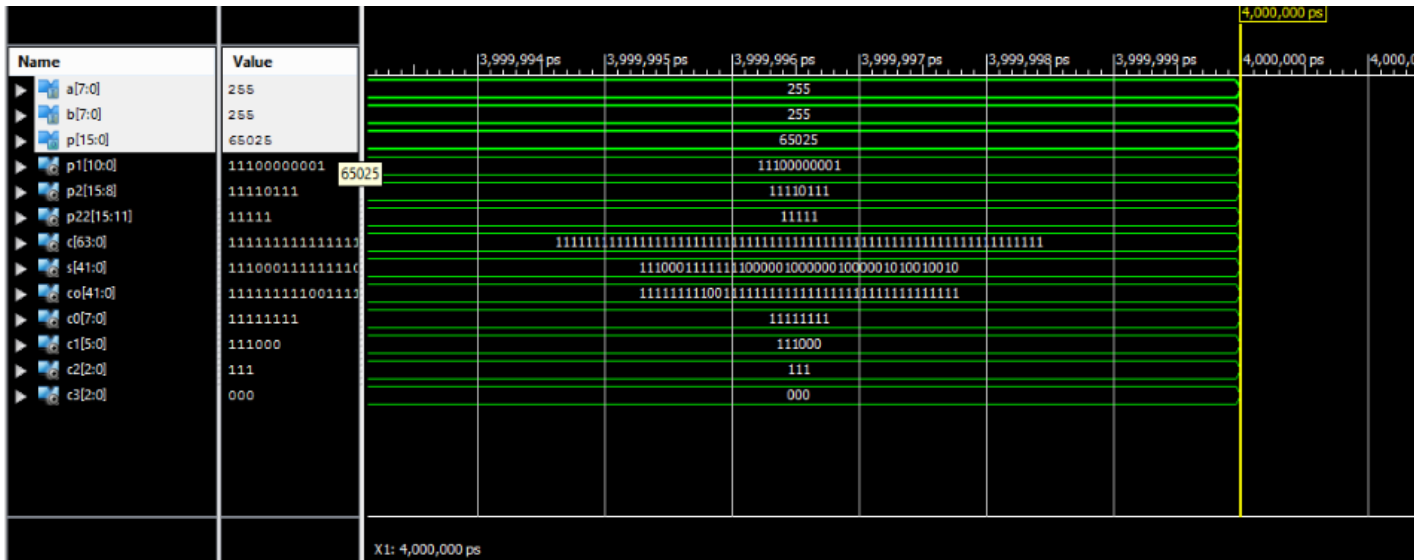
Cap -3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
 m1 5	 m14	 fs34	 fs3 3	 fs3 2	 fs3 1	 fs30	 fs29	 fs2 8	 fs2 7	 fs2 6	 fs2 5	 fs2 4	 hs6	 m1	 m0
	 fc34	 fc33	 fc3 2	 fc3 1	 fc3 0	 fc29	 fc28	 fc2 7	 fc2 6	 fc2 5	 fc2 4	 hc6	 m2	 0,1	 0,0
	 7,6	 fc23	 fs2 2	 fs2 1	 fs1 9	 fs17	 fs15	 fs1 3	 fs1 1	 fs9	 fs8	 hs4	 0,2	 1,0	

Cap -2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
 m15	 fs47	 fs46	 fs45	 fs44	 fs4 3	 fs42	 fs41	 fs40	 fs39	 fs38	 fs37	 fs36	 fs35	 hs7	 m0
 fc47	 fc46	 fc45	 fc44	 fc43	 fc4 2	 fc41	 fc40	 fc39	 fc38	 fc37	 fc3 6	 fc35	 hc7	 m1	 0,0

### Simulation Results for 8-bit Dadda Multiplier :



### Codes :

- Testbench :

```

`timescale 1ps/100fs
module top_level_tb(); //testbench doesnt have any inputs or outputs
    reg [7:0] A;        //inputs are taken as registers ( they need to hold the value)
    reg [7:0] B;
    reg [15:0] M;
    wire [16:0] RES; //outputs are taken as wires in tb .
    reg [7:0] ain_array[0:250];
    reg [7:0] bin_array[0:250];
    reg [16:0] res_array[0:250];
    reg [15:0] M_array[0:250];
    top_level dut( A,B, M,RES); //since all the inputs to the dut are the wires of same name
    integer i;
    initial begin
        $dumpfile("top_level_tb.vcd");
        $dumpvars(0,top_level_tb); //first argument is the level of debugging
                                     //level 0 will log all the variable even in

        $monitor(A,B,M,RES);

        $readmemb("ain.txt",ain_array);
        $readmemb("bin.txt",bin_array);
        $readmemb("res.txt",res_array);
        $readmemb("m.txt",M_array);
    end
endmodule

```



```

//sub modules
//whereas level 1 will log only the ones in the top module
A = 8'h0;
B = 8'h0;
M = 16'h0;
#2000;
A = 8'hff;
B = 8'haa;
#2000;
B = 8'hff;
#200;
#1000;

M = 16'h02;
$display("starting....");

for(i = 0;i<250;i = i+1)begin
    A = ain_array[i];
    B = bin_array[i];
    M = M_array[i];
    #1000;
    $display("A = %h, B = %h, M = %h , RES = %h",A,B,M,RES);
    if(RES != res_array[i])
        $display("error");
    else
        $display("test passed");

end

end
endmodule

```

- **Generate partial products :**

```

`timescale 1ps/100fs
module gen_part_products(
    input [7:0] A,
    input [7:0] B,
    output[7:0] P[7:0]);    //portlist can be 2D array in verilog
    genvar i;
    generate
        for(i = 0; i < 8; i = i +1) begin:part_product
            assign P[i][0] = A[0] & B[i] ;
            assign P[i][1] = A[1] & B[i] ;
            assign P[i][2] = A[2] & B[i] ;
            assign P[i][3] = A[3] & B[i] ;
            assign P[i][4] = A[4] & B[i] ;
            assign P[i][5] = A[5] & B[i] ;
            assign P[i][6] = A[6] & B[i] ;
            assign P[i][7] = A[7] & B[i] ;

        end
    endgenerate
endmodule

```

- **Carry Out Select :**

```

`timescale 1ps/100fs
module CSA_block #(parameter width = 4)(
    input [width-1:0]A,
    input [width-1:0]B,
    output[1:0][width-1:0]SUM,
    output[1:0]c_out);

    genvar i;
    wire [1:0][width-1:0]carry_out;
    full_adder fa0(A[0],B[0],1'b0,SUM[0][0],carry_out[0][0]);
    full_adder fa1(A[0],B[0],1'b1,SUM[1][0],carry_out[1][0]);
    generate
        for (i = 1; i < width-1 ; i = i + 1) begin:gen_CSA_block
            full_adder fa_carry_out_zero(A[i],B[i],carry_out[0][i-1],SUM[0][i],carry_out[0][i]);
            full_adder fa_carry_out_one(A[i],B[i],carry_out[1][i-1],SUM[1][i],carry_out[1][i]);
        end
    endgenerate

    full_adder fa_carry_out_zero_last(A[width-1],B[width-1],carry_out[0][width-2],SUM[0][width-1],c_out[0]);
    full_adder fa_carry_out_one_last(A[width-1],B[width-1],carry_out[1][width-2],SUM[1][width-1],c_out[1]);
endmodule

module selector #(parameter width = 4)(
    input [1:0][width-1:0]SUM,
    input [1:0]c_out,
    input c_in,
    output [width-1:0]SUM_OUT,
    output CARRY_OUT);

    genvar i;

    assign SUM_OUT = c_in?SUM[1]:SUM[0];
    assign CARRY_OUT = c_in?c_out[1]:c_out[0];
endmodule

module adder16(
    input [15:0]A,
    input [15:0]B,
    input c_in,
    output [16:0]SUM);

    wire [2:0][1:0] block_carry_out;
    wire [1:0][3:0] sum_block1;
    wire [1:0][4:0] sum_block2;
    wire [1:0][5:0] sum_block3;
    wire [3:0]cout_inter;
    wire cout_0;
    //carry select adder is implemented in stages of 1 , 4 , 5 , 6

```

```
full_adder fa0_in16(A[0],B[0],c_in,SUM[0],cout_0);
CSA_block #(.width(4)) block_1(A[4:1],B[4:1],sum_block1,block_carry_out[0]);
CSA_block #(.width(5)) block_2(A[9:5],B[9:5],sum_block2,block_carry_out[1]);
CSA_block #(.width(6)) block_3(A[15:10],B[15:10],sum_block3,block_carry_out[2]);

selector #(.width(4)) sel0(sum_block1,block_carry_out[0],cout_0,SUM[4:1],cout_inter[0]);
selector #(.width(5)) sel1(sum_block2,block_carry_out[1],cout_inter[0],SUM[9:5],cout_inter[1]);
selector #(.width(6)) sel2(sum_block3,block_carry_out[2],cout_inter[1],SUM[15:10],SUM[16]);
```

```
endmodule
```