

Self-Project

**Topic: Implementation of 8x8 bit Dadda
Multiplier**

Name: Gautam Sahay

Roll No: 213070084

Mtech 2nd year - EE7(SSD)

Dadda Multipliers

Dadda multipliers are very similar to Wallace multipliers and use the same 3 stages:

- 1) Generate all bits of the partial products in parallel.
- 2) Collect all partial products bits with the same place value in bunches of wires and reduce these in several layers of adders till each weight has no more than two wires.
- 3) For all bit positions which have two wires, take one wire at corresponding place values to form one number, and the other wire to form another number.

Add these two numbers using a fast adder of appropriate size.

The difference is in the reduction stage.

- ***Wallace multipliers reduce as soon as possible, while Dadda multipliers reduce as late as possible.***
- Dadda multipliers plan on reducing the final number of wires for any weight to 2 with as few and as small adders as possible.
- We determine the number of layers required first, beginning from the last layer, where no more than 2 wires should be left.
- The number of layers in Dadda multipliers is the same as in Wallace multipliers.

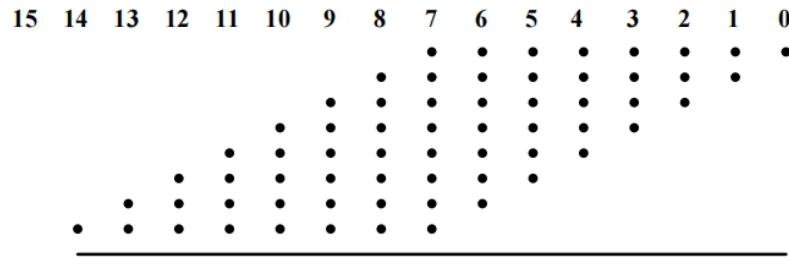
Dadda Multipliers: Number of layers

- We work back from the final adder to earlier layers till we find that we can manage all wires generated by the partial product generator.
- We know that the final adder can take no more than 2 wires for each weight.
- Let d_j represent the maximum number of wires for any weight in layer j , where $j = 1$ for the final adder. (Thus $d_1 = 2$). The maximum number of wires which can be handled in layer $j+1$ (from the end) is the integral part of $\frac{3}{2}d_j$.
- We go up in j , till we reach a number which is just greater than or equal to the largest bunch of wires in any weight.
- The number of reduction layers required is this $j_{final}-1$.

Wire Reduction in Dadda Multipliers

- At each layer we know the maximum number of wires which should be left for the next layer.
- For each weight, we place the **least number of smallest adders**, such that the wires going out to the next layer do not exceed the maximum number of wires it can handle.
- At each weight, we must consider all the sum and pass through wires at this weight, as well as the wires which will be transferred through carry of the less significant weights, to the next layer.
- That is why we must begin with the lowest weight and go towards higher weights in each layer.

Dadda 8X8 Multiplier: Dot diagrams



capacity: 6

capacity: 4

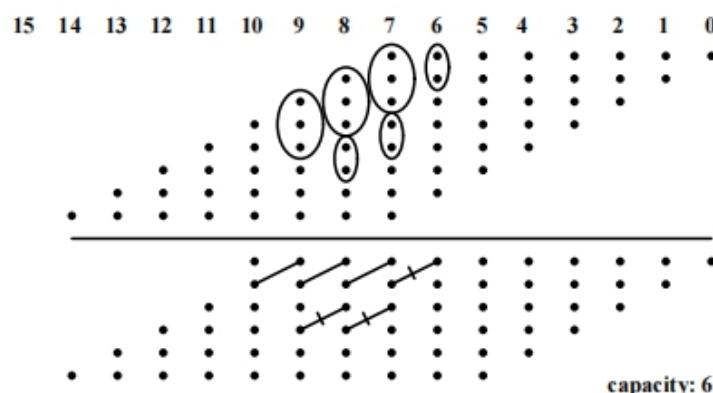
capacity: 3

capacity: 2

Reduction Stages:

Dadda 8X8 Multiplier: First reduction

- Capacity of next layer is 6. Since bits 0-5 have six or less wires, these are just passed through.
- bit 6 has 7 wires. To reduce to six, we place a half adder. (This gives a sum wire at bit 6 and a carry wire at bit 7).
- These outputs are shown by a dot each at bits 6 and 7, joined by a crossed line.
- Remaining 5 bits are passed through.



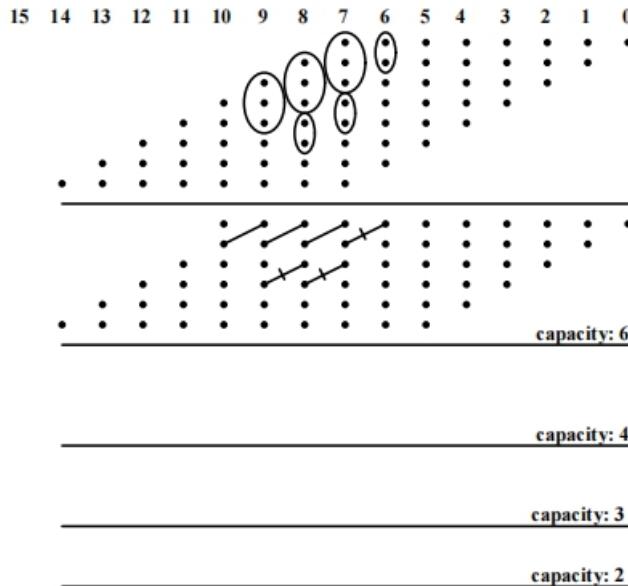
capacity: 6

capacity: 4

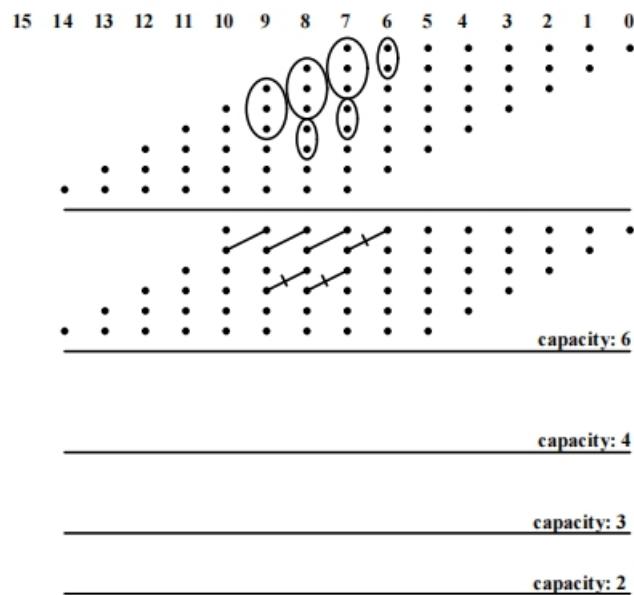
capacity: 3

capacity: 2

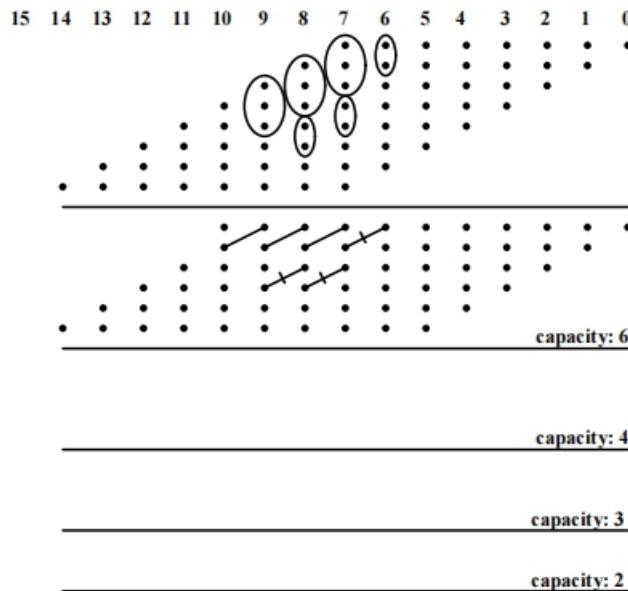
- Bit 7 has 8 wires. Of the six output places, one is already occupied by the carry of half adder at bit 6. So we should produce only 5 outputs at this bit – a reduction by 3.
- This can be done through a full and a half adder. Outputs of the full adder are shown as a dot at bit 7 (sum) and another at bit 8 (carry) joined by a line.
- Outputs of the half adder are also shown by two dots joined by a crossed line as before.



- Bit 8 has 7 wires. Of the 6 output places, 2 are occupied by carries of full and half adder.
- So we should produce only 4 outputs at this bit – again a reduction by 3.
- This can be done through a full and a half adder as before.
- Full and half adder take up 5 wires. The remaining 2 are passed through.

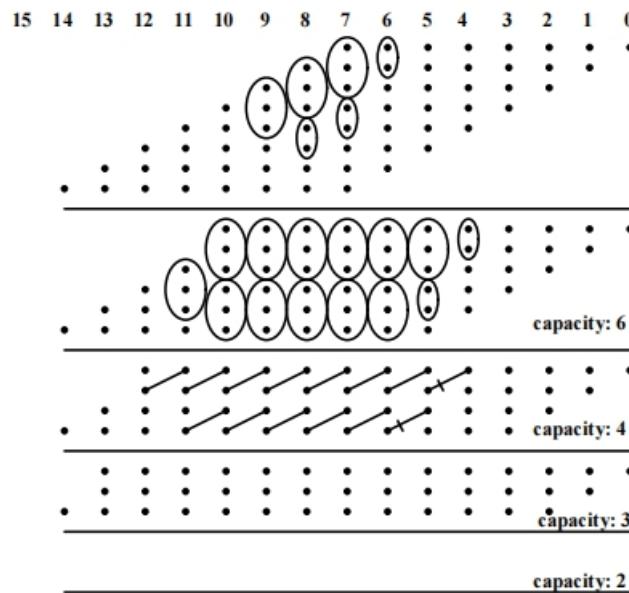


- Bit 9 has 6 wires, which should be reduced to 4 (since two places are taken up by carries of full and half adders).
- This can be done by a full adder whose outputs are shown by dots at bit 9 and 10 joined by a line.
- The remaining 3 wires are passed through.
- Wires of all the higher bits can be passed through without exceeding the limit of 6 outputs.

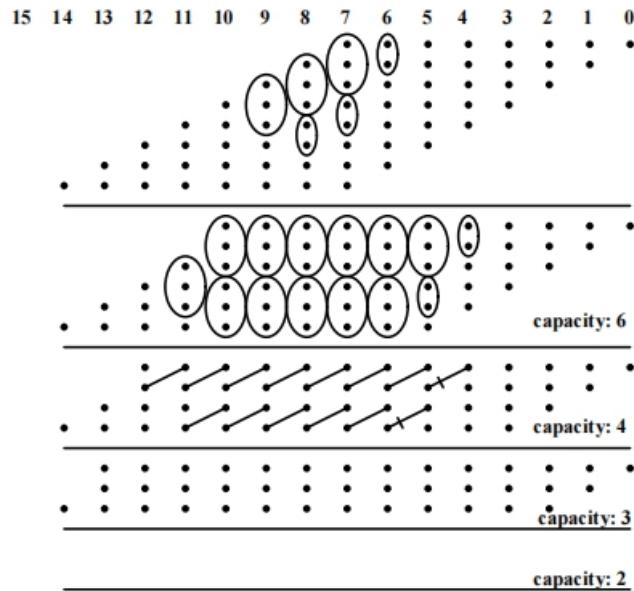


Dadda 8X8 Multiplier: Second reduction

- The output capacity of next layer is 4.
- Wires of bits 0-3 can just be passed through.
- For all bit position, we reduce the output places available by the incoming carries of previous bit.
- We place minimal number of full and half adders to reduce the total output wires to 4.
- Each full adder (FA) reduces wires by 2, half adder (HA) reduces by 1.

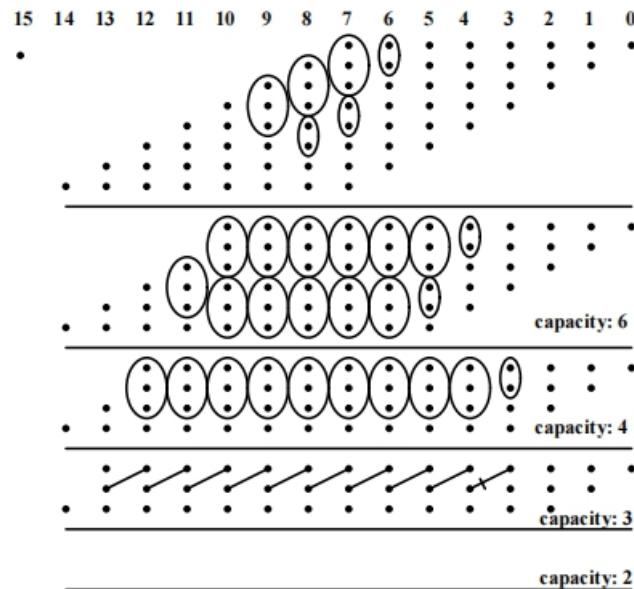


- Bit 4 has 5 wires. Reduced to 4 by HA.
- Bit 5 has 6 wires, reduced to (4-1) by FA+HA.
- Bit 6 has 6 wires, reduced to (4-2) by 2 FAs.
- This is repeated till bit 10.
- Bit 11 has 4 wires, reduced to (4-2) by a FA.
- All other wires can be passed through.



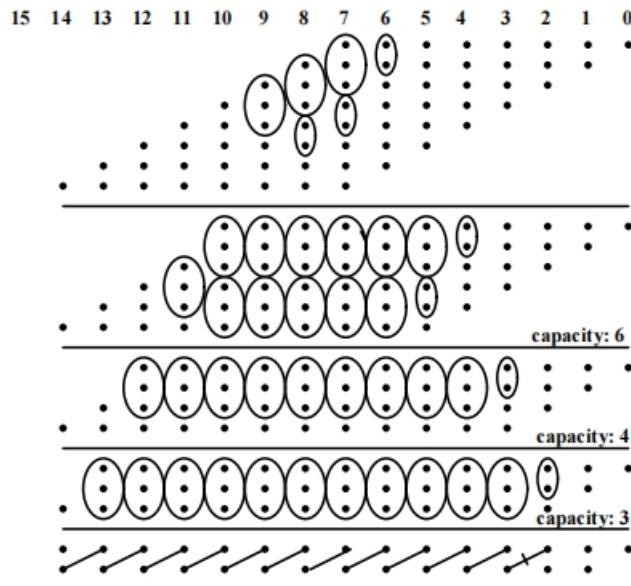
Dadda 8X8 Multiplier: Third reduction

- The reduction procedure can be repeated at each layer.
- If 2 wires (or multiples of 2) are to be reduced, we place FAs till 1 or 0 wires are left.
- If 1 wire remains, we place a Half adder.
- This layer requires a half adder at bit 4, FA + HA at bit 5, 2 FAs at bits 6-10 and a full adder at bit 11.
- Rest of the wires are just passed through.



Dadda 8X8 Multiplier: Final reduction

- Capacity of the final layer is 2.
- We continue with the same procedure, placing a half adder at bit 3 and full adders at bits 4-12.
- The remaining wires are passed through. Now we can make two words of 14 bit width and add these using a fast adder to get the final product.
- Notice there is no extra bit!



Comparison of Wallace and Dadda Multipliers

- Wallace and Dadda multipliers need the same number of layers.
- Dadda multiplier minimizes the number of adders, so it has the potential for lower complexity and power.
- Dadda multiplier uses more pass throughs and smaller adders which have lower delay. Thus it can also minimize the critical delay for reaching the final 2 wire stage.
- However, The final addition in Dadda multiplier needs wider carry propagating adders, which can slow it down.
- A careful evaluation inclusive of wiring and parasitic delays has to be made to determine which is the faster adder for a given configuration and process.

Verilog CODE for 8x8 Dadda Multiplier (Including TEST BENCH) :

```
1 // Designing in Half Adder
2 // Sum = a XOR b, Cout = a AND b
3
4
5 module HA(a, b, Sum, Cout);
6
7 input a, b; // a and b are inputs with size 1-bit
8 output Sum, Cout; // Sum and Cout are outputs with size 1-bit
9
10 assign Sum = a ^ b;
11 assign Cout = a & b;
12
13 endmodule
14
15
16 //carry save adder -- for implementing dadda multiplier
17 //csa for use of half adder and full adder.
18
19 module csa_dadda(A,B,Cin,Y,Cout);
20 input A,B,Cin;
21 output Y,Cout;
22
23 assign Y = A^B^Cin;
24 assign Cout = (A&B) | (A&Cin) | (B&Cin);
25
26 endmodule
27
28
29 // dadda multiplier
30 // A - 8 bits , B - 8bits, y(output) - 16bits
31
32 module dadda_8(A,B,y);
33
34     input [7:0] A;
35     input [7:0] B;
36     output wire [15:0] y;
37     wire gen_pp [0:7][7:0];
38 // stage-1 sum and carry
39     wire [0:5]s1,c1;
40 // stage-2 sum and carry
41     wire [0:13]s2,c2;
42 // stage-3 sum and carry
43     wire [0:9]s3,c3;
44 // stage-4 sum and carry
45     wire [0:11]s4,c4;
46 // stage-5 sum and carry
47     wire [0:13]s5,c5;
48
49
50
51
52 // generating partial products
53
54 /////////////////////////////////
55
56 //// j=0
57 assign gen_pp[0][0] = A[0]*B[0];
58 assign gen_pp[1][0] = A[0]*B[1];
59 assign gen_pp[2][0] = A[0]*B[2];
60 assign gen_pp[3][0] = A[0]*B[3];
61 assign gen_pp[4][0] = A[0]*B[4];
62 assign gen_pp[5][0] = A[0]*B[5];
63 assign gen_pp[6][0] = A[0]*B[6];
64 assign gen_pp[7][0] = A[0]*B[7];
65
66
67 //// j=1
68 assign gen_pp[0][1] = A[1]*B[0];
69 assign gen_pp[1][1] = A[1]*B[1];
70 assign gen_pp[2][1] = A[1]*B[2];
71 assign gen_pp[3][1] = A[1]*B[3];
72 assign gen_pp[4][1] = A[1]*B[4];
73 assign gen_pp[5][1] = A[1]*B[5];
```



```

147 //Reduction by stages.
148 // di_values = 2,3,4,6,8,13...
149
150
151 //Stage 1 - reducing from 8 to 6
152
153
154 HA h1(.a(gen_pp[6][0]),.b(gen_pp[5][1]),.Sum(s1[0]),.Cout(c1[0]));
155 HA h2(.a(gen_pp[4][3]),.b(gen_pp[3][4]),.Sum(s1[2]),.Cout(c1[2]));
156 HA h3(.a(gen_pp[4][4]),.b(gen_pp[3][5]),.Sum(s1[4]),.Cout(c1[4]));
157
158 csa_dadda c11(.A(gen_pp[7][0]),.B(gen_pp[6][1]),.Cin(gen_pp[5][2]),.Y(s1[1]),.
Cout(c1[1]));
159 csa_dadda c12(.A(gen_pp[7][1]),.B(gen_pp[6][2]),.Cin(gen_pp[5][3]),.Y(s1[3]),.
Cout(c1[3]));
160 csa_dadda c13(.A(gen_pp[7][2]),.B(gen_pp[6][3]),.Cin(gen_pp[5][4]),.Y(s1[5]),.
Cout(c1[5]));
161
162 //Stage 2 - reducing from 6 to 4
163
164 HA h4(.a(gen_pp[4][0]),.b(gen_pp[3][1]),.Sum(s2[0]),.Cout(c2[0]));
165 HA h5(.a(gen_pp[2][3]),.b(gen_pp[1][4]),.Sum(s2[2]),.Cout(c2[2]));
166
167
168 csa_dadda c21(.A(gen_pp[5][0]),.B(gen_pp[4][1]),.Cin(gen_pp[3][2]),.Y(s2[1]),.
Cout(c2[1]));
169 csa_dadda c22(.A(s1[0]),.B(gen_pp[4][2]),.Cin(gen_pp[3][3]),.Y(s2[3]),.Cout(c2[3]
));
170 csa_dadda c23(.A(gen_pp[2][4]),.B(gen_pp[1][5]),.Cin(gen_pp[0][6]),.Y(s2[4]),.
Cout(c2[4]));
171 csa_dadda c24(.A(s1[1]),.B(s1[2]),.Cin(c1[0]),.Y(s2[5]),.Cout(c2[5]));
172 csa_dadda c25(.A(gen_pp[2][5]),.B(gen_pp[1][6]),.Cin(gen_pp[0][7]),.Y(s2[6]),.
Cout(c2[6]));
173 csa_dadda c26(.A(s1[3]),.B(s1[4]),.Cin(c1[1]),.Y(s2[7]),.Cout(c2[7]));
174 csa_dadda c27(.A(c1[2]),.B(gen_pp[2][6]),.Cin(gen_pp[1][7]),.Y(s2[8]),.Cout(c2[8]
));
175 csa_dadda c28(.A(s1[5]),.B(c1[3]),.Cin(c1[4]),.Y(s2[9]),.Cout(c2[9]));
176 csa_dadda c29(.A(gen_pp[4][5]),.B(gen_pp[3][6]),.Cin(gen_pp[2][7]),.Y(s2[10]),.
Cout(c2[10]));
177 csa_dadda c210(.A(gen_pp[7][3]),.B(c1[5]),.Cin(gen_pp[6][4]),.Y(s2[11]),.Cout(c2[11]
));
178 csa_dadda c211(.A(gen_pp[5][5]),.B(gen_pp[4][6]),.Cin(gen_pp[3][7]),.Y(s2[12]),.
Cout(c2[12]));
179 csa_dadda c212(.A(gen_pp[7][4]),.B(gen_pp[6][5]),.Cin(gen_pp[5][6]),.Y(s2[13]),.
Cout(c2[13]));
180
181 //Stage 3 - reducing from 4 to 3
182
183 HA h6(.a(gen_pp[3][0]),.b(gen_pp[2][1]),.Sum(s3[0]),.Cout(c3[0]));
184
185 csa_dadda c31(.A(s2[0]),.B(gen_pp[2][2]),.Cin(gen_pp[1][3]),.Y(s3[1]),.Cout(c3[1]
));
186 csa_dadda c32(.A(s2[1]),.B(s2[2]),.Cin(c2[0]),.Y(s3[2]),.Cout(c3[2]));
187 csa_dadda c33(.A(c2[1]),.B(c2[2]),.Cin(s2[3]),.Y(s3[3]),.Cout(c3[3]));
188 csa_dadda c34(.A(c2[3]),.B(c2[4]),.Cin(s2[5]),.Y(s3[4]),.Cout(c3[4]));
189 csa_dadda c35(.A(c2[5]),.B(c2[6]),.Cin(s2[7]),.Y(s3[5]),.Cout(c3[5]));
190 csa_dadda c36(.A(c2[7]),.B(c2[8]),.Cin(s2[9]),.Y(s3[6]),.Cout(c3[6]));
191 csa_dadda c37(.A(c2[9]),.B(c2[10]),.Cin(s2[11]),.Y(s3[7]),.Cout(c3[7]));
192 csa_dadda c38(.A(c2[11]),.B(c2[12]),.Cin(s2[13]),.Y(s3[8]),.Cout(c3[8]));
193 csa_dadda c39(.A(gen_pp[7][5]),.B(gen_pp[6][6]),.Cin(gen_pp[5][7]),.Y(s3[9]),.
Cout(c3[9]));
194
195 //Stage 4 - reducing from 3 to 2
196
197 HA h7(.a(gen_pp[2][0]),.b(gen_pp[1][1]),.Sum(s4[0]),.Cout(c4[0]));
198
199
200 csa_dadda c41(.A(s3[0]),.B(gen_pp[1][2]),.Cin(gen_pp[0][3]),.Y(s4[1]),.Cout(c4[1]
));
201 csa_dadda c42(.A(c3[0]),.B(s3[1]),.Cin(gen_pp[0][4]),.Y(s4[2]),.Cout(c4[2]));
202 csa_dadda c43(.A(c3[1]),.B(s3[2]),.Cin(gen_pp[0][5]),.Y(s4[3]),.Cout(c4[3]));
203 csa_dadda c44(.A(c3[2]),.B(s3[3]),.Cin(s2[4]),.Y(s4[4]),.Cout(c4[4]));
204 csa_dadda c45(.A(c3[3]),.B(s3[4]),.Cin(s2[6]),.Y(s4[5]),.Cout(c4[5]));

```

```

205    csa_dadda c46(.A(c3[4]),.B(s3[5]),.Cin(s2[8]),.Y(s4[6]),.Cout(c4[6]));
206    csa_dadda c47(.A(c3[5]),.B(s3[6]),.Cin(s2[10]),.Y(s4[7]),.Cout(c4[7]));
207    csa_dadda c48(.A(c3[6]),.B(s3[7]),.Cin(s2[12]),.Y(s4[8]),.Cout(c4[8]));
208    csa_dadda c49(.A(c3[7]),.B(s3[8]),.Cin(gen_pp[4][7]),.Y(s4[9]),.Cout(c4[9]));
209    csa_dadda c410(.A(c3[8]),.B(s3[9]),.Cin(c2[13]),.Y(s4[10]),.Cout(c4[10]));
210    csa_dadda c411(.A(c3[9]),.B(gen_pp[7][6]),.Cin(gen_pp[6][7]),.Y(s4[11]),.Cout(c4[11]));
211
212 //Stage 5 - reducing from 2 to 1
213 // adding total sum and carry to get final output
214
215 HA h8(.a(gen_pp[1][0]),.b(gen_pp[0][1]),.Sum(y[1]),.Cout(c5[0]));
216
217
218
219    csa_dadda c51(.A(s4[0]),.B(gen_pp[0][2]),.Cin(c5[0]),.Y(y[2]),.Cout(c5[1]));
220    csa_dadda c52(.A(c4[0]),.B(s4[1]),.Cin(c5[1]),.Y(y[3]),.Cout(c5[2]));
221    csa_dadda c54(.A(c4[1]),.B(s4[2]),.Cin(c5[2]),.Y(y[4]),.Cout(c5[3]));
222    csa_dadda c55(.A(c4[2]),.B(s4[3]),.Cin(c5[3]),.Y(y[5]),.Cout(c5[4]));
223    csa_dadda c56(.A(c4[3]),.B(s4[4]),.Cin(c5[4]),.Y(y[6]),.Cout(c5[5]));
224    csa_dadda c57(.A(c4[4]),.B(s4[5]),.Cin(c5[5]),.Y(y[7]),.Cout(c5[6]));
225    csa_dadda c58(.A(c4[5]),.B(s4[6]),.Cin(c5[6]),.Y(y[8]),.Cout(c5[7]));
226    csa_dadda c59(.A(c4[6]),.B(s4[7]),.Cin(c5[7]),.Y(y[9]),.Cout(c5[8]));
227    csa_dadda c510(.A(c4[7]),.B(s4[8]),.Cin(c5[8]),.Y(y[10]),.Cout(c5[9]));
228    csa_dadda c511(.A(c4[8]),.B(s4[9]),.Cin(c5[9]),.Y(y[11]),.Cout(c5[10]));
229    csa_dadda c512(.A(c4[9]),.B(s4[10]),.Cin(c5[10]),.Y(y[12]),.Cout(c5[11]));
230    csa_dadda c513(.A(c4[10]),.B(s4[11]),.Cin(c5[11]),.Y(y[13]),.Cout(c5[12]));
231    csa_dadda c514(.A(c4[11]),.B(gen_pp[7][7]),.Cin(c5[12]),.Y(y[14]),.Cout(c5[13]));
232
233 assign y[0] = gen_pp[0][0];
234 assign y[15] = c5[13];
235
236
237 endmodule
238
239
240
241 //self checking test bench for 8*8 dadda
242 module dadda_STB();
243
244
245 parameter M=8,N=8;
246
247 reg [N-1:0]A;
248 reg [M-1:0]B;
249 wire [N+M-1:0]Y;
250
251
252
253 //---- Instantiation of main test module----
254 //Array_MUL_USign #(64,64) UUT(A,B,Y); //M=4,N=6
255 dadda_8 UUT(.A(A),.B(B),.y(Y));
256
257
258 // initializing the inputs to the test module
259 // initial block executes only once
260 initial
261 repeat(15)
262 begin
263 #10 A = $random; B = $random;
264 #100//give required simulation time to complete the operation one by one.
265 #100
266 #10
267 //-----VERIFICATION OF THE OBTAINED RESULT WITH EXISTING RESULT-----
268 $display(" A=%d,B=%d,AxB=%d", (A), (B), (Y));
269
270 if( (A)*(B) != (Y))
271 $display(" *ERROR* ");
272
273 end
274
275 endmodule
276

```

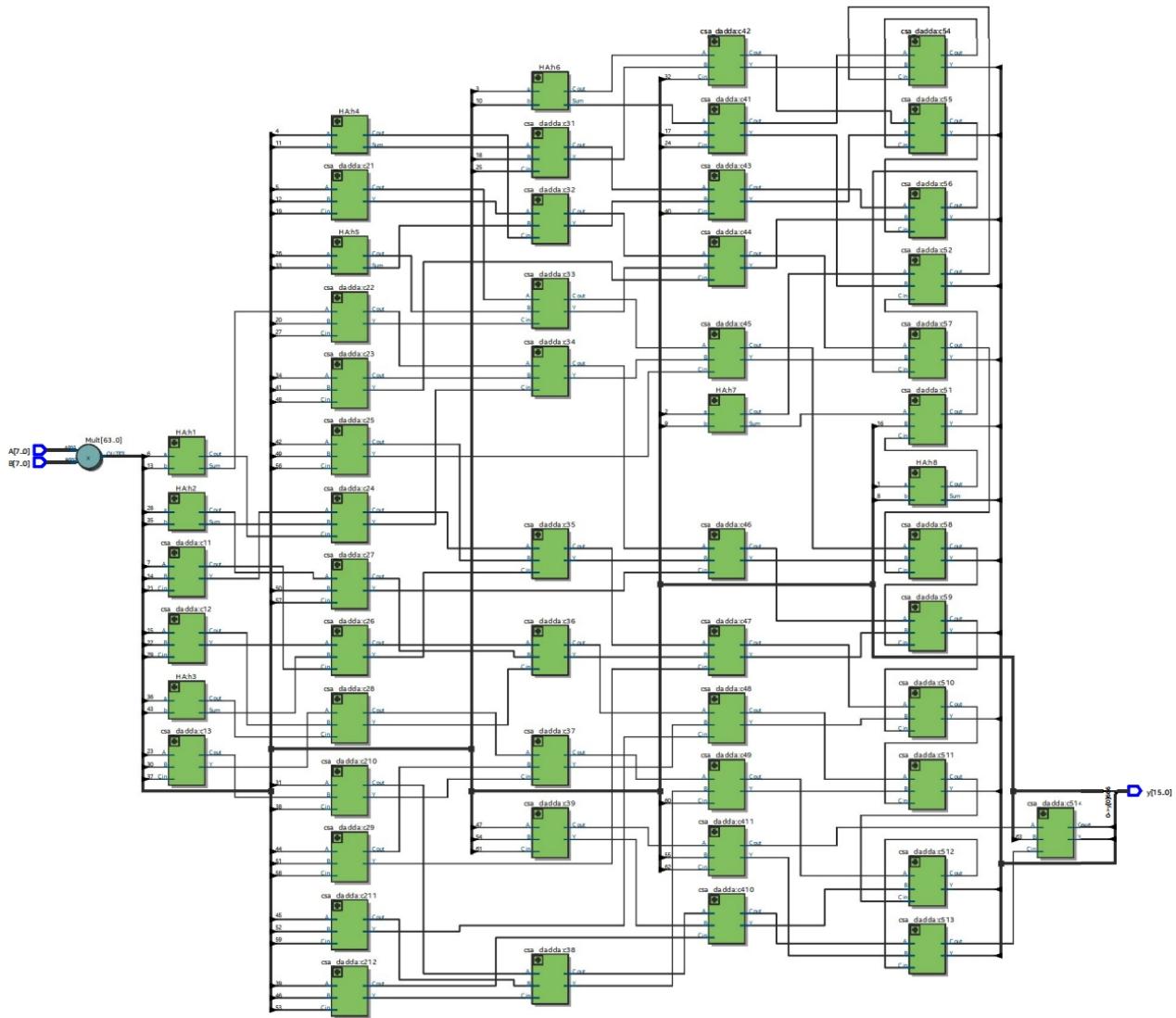
SYNTHESIS of 8x8 Dadda Multiplier using Quartus Prime

The screenshot shows the Quartus Prime Lite Edition interface with the following details:

- File Menu:** File, Edit, View, Project, Assignments, Processing, Tools, Window, Help.
- Project Navigator:** Shows EntityInstance Cyclone V: 5CGXFC7CF23CB and dadda_8.
- Hierarchy:** Shows the project structure.
- Tasks:** Task list including Compile Design, Analysis & Synthesis, Fitter (Place & Route), Assembler (Generate program), and Timing Analysis. Analysis & Synthesis is at 100% completion.
- Status:** Module, % Progress, Time. Analysis & Synthesis is at 100% 00:00:12.
- Code Editor:** Displays Verilog code for dadda_8.v:

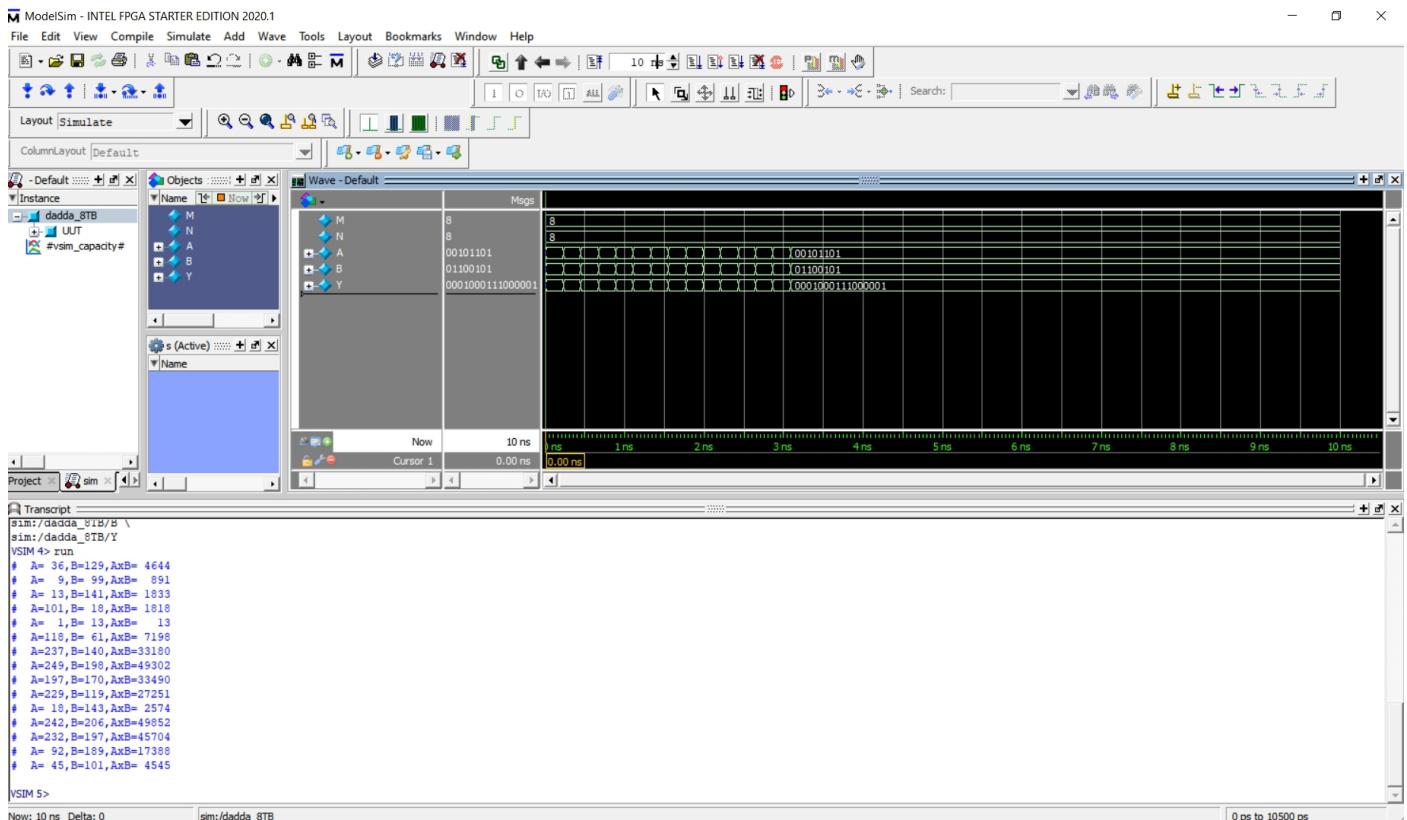
```
1 // Designing in Half Adder
2 // Sum = a XOR b, Cout = a AND b
3
4 module HA(a, b, Sum, Cout);
5
6 input a, b; // a and b are inputs with size 1-bit
7 output Sum, Cout; // Sum and Cout are outputs with size 1-bit
8
9 assign Sum = a ^ b;
10 assign Cout = a & b;
11
12 endmodule
13
14
15 //carry save adder -- for implementing dadda multiplier
16 //csa for use of half adder and full adder.
17
18 module csa_dadda(A,B,Cin,Y,Cout);
19 input A,B,Cin;
20 output Y,Cout;
21
22 assign Y = A&B&Cin;
23 assign Cout = (A&B)|(A&Cin)|(B&Cin);
24
25 endmodule
26
27
28
29 // dadda multiplier
30 // A - 8 bits , B - 8bits, y(output) - 16bits
31
32 module dadda_8(A,B,y);
33
34 input [7:0] A;
35 input [7:0] B;
36 output wire [15:0] y;
37 wire gen_pp [0:7][7:0];
38 // stage-1 sum and carry
```
- IP Catalog:** Shows Installed IP, Project Directory (No Selection Available), Library (Basic Functions, DSP, Interface Protocols, Memory Interfaces and Controllers, Processors and Peripherals, University Program), and Search for Partner IP.
- Messages:** Shows synthesis messages:
 - 16010 Generating hard_block partition "hard_block:auto_generated_inst"
 - 21057 Implemented 186 device resources after synthesis - the final resource count might be different
 - Quartus Prime Analysis & Synthesis was successful. 0 errors, 1 warning
- Bottom Status:** Opens an existing file, 100% 00:00:12.

RTL View of 8x8 Dadda Multiplier using Quartus Prime

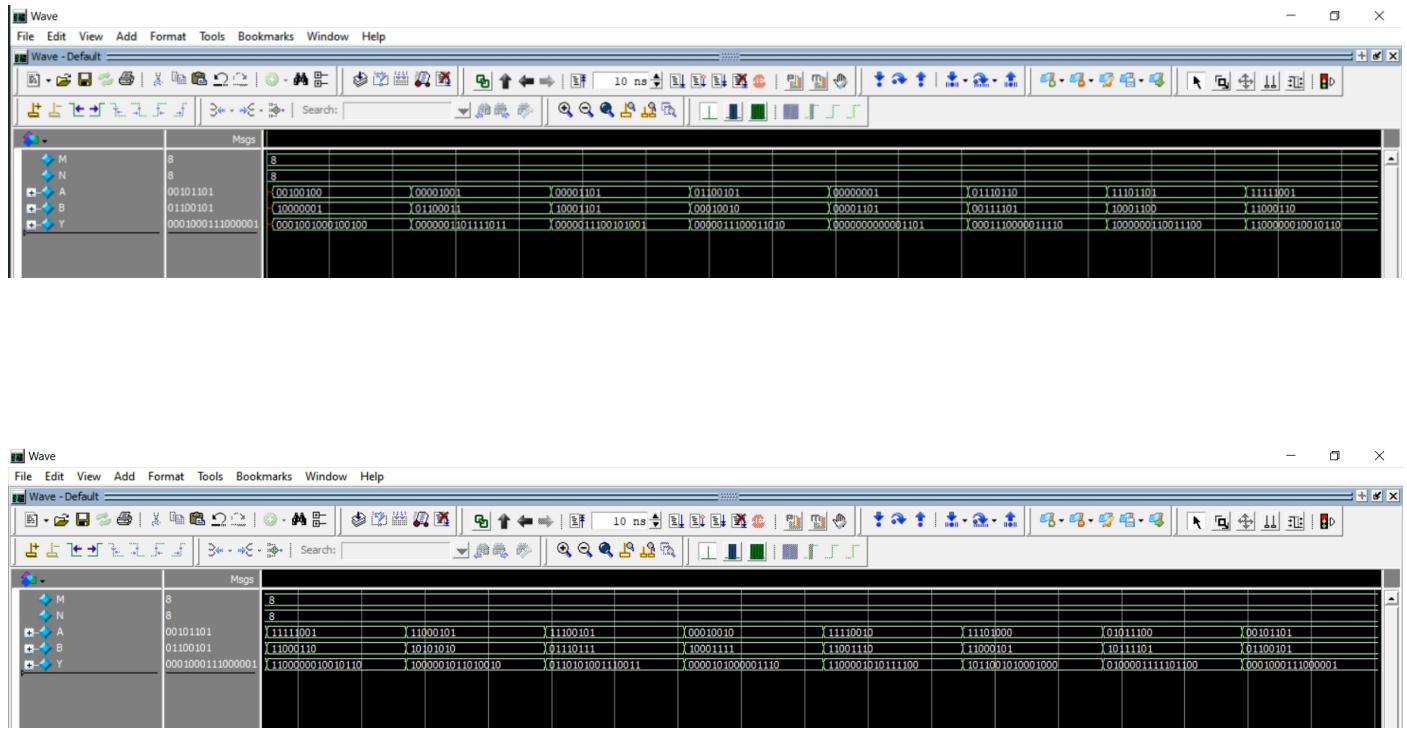


SIMULATION results using Modelsim :

Results in binary(base) representation



Full view of all the result values



Result in Decimal base

