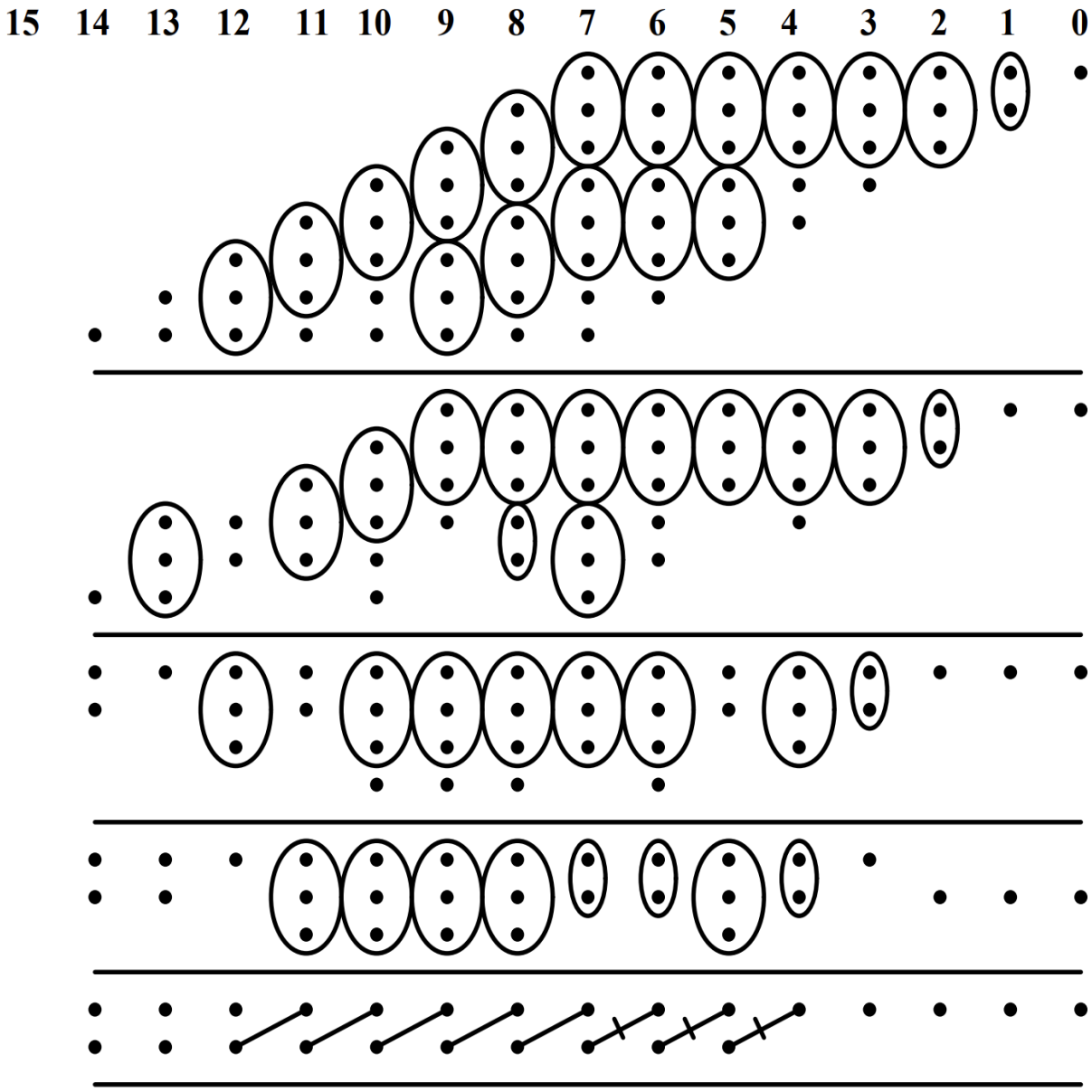
Problem Statement:

- Given Wallace multiplier algorithm implement it using Verilog.
- Implement the algorithm with 8bit x 8bit multiplication.
- Using structural blocks such as Half Adder and Full Adder.
- Generate RTL view.
- Write a Testbench for multiplying various two 8-bit numbers.
- Show two 8-bit multiplication in Modelsim.

Solution:

We have implemented a two 8-bit modified wallace multiplier which does not have any redundant Carry bit which it used to occur in normal wallace multiplier.

Given below is the modified Wallace Multiplier Algorithm:



Here in the first layer each dot represents a product between two 1-bit multiplication – for example→ for numbers A&B → a7a6a5a4a3a2a1a0 * b7b6b5b4b3b2b1b0→ one dot = b0a0, b1a1, etc.

For implementing this algorithm, we have used **39 Full Adders and 17 Half Adders**.

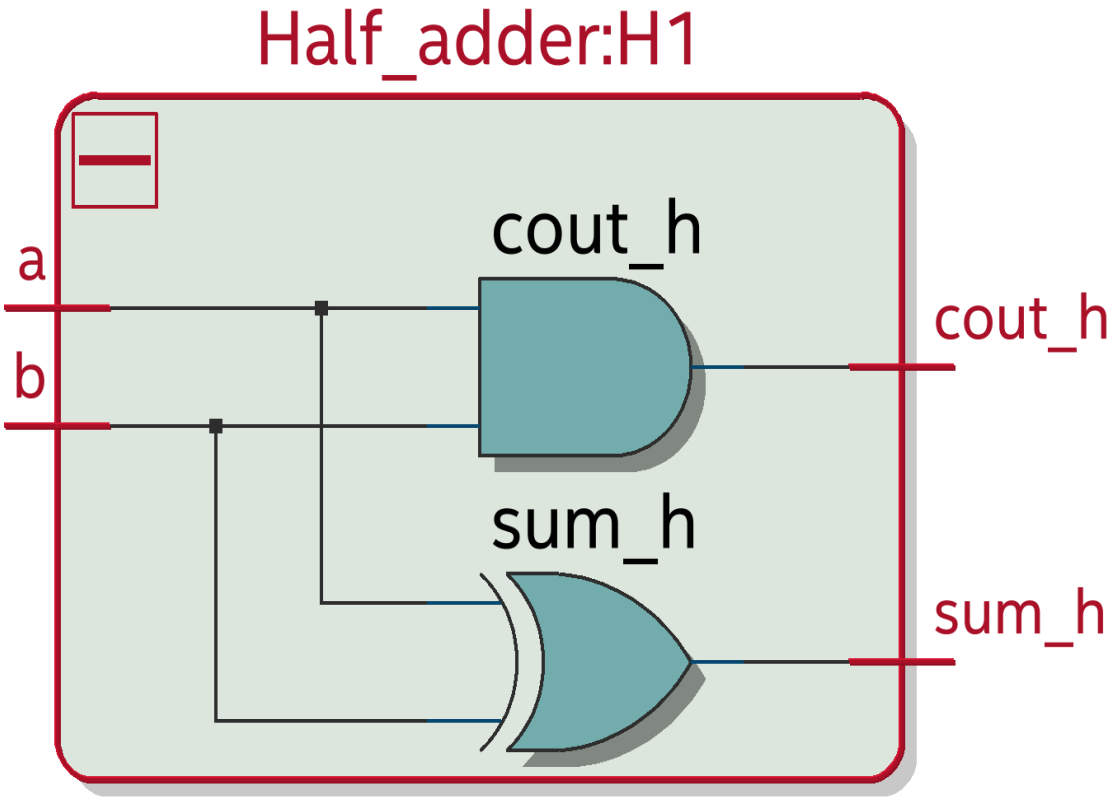
Below are code snippets and their respective RTL views of various structural blocks used for generating this algorithm.

For Half Adder:

CODE SNIPPET:

```
module Half_adder(input a,b,
output reg sum_h,cout_h);

always@(*)
begin
sum_h<= a^b;
cout_h<= a&b;
end
endmodule
```

RTL VIEW:

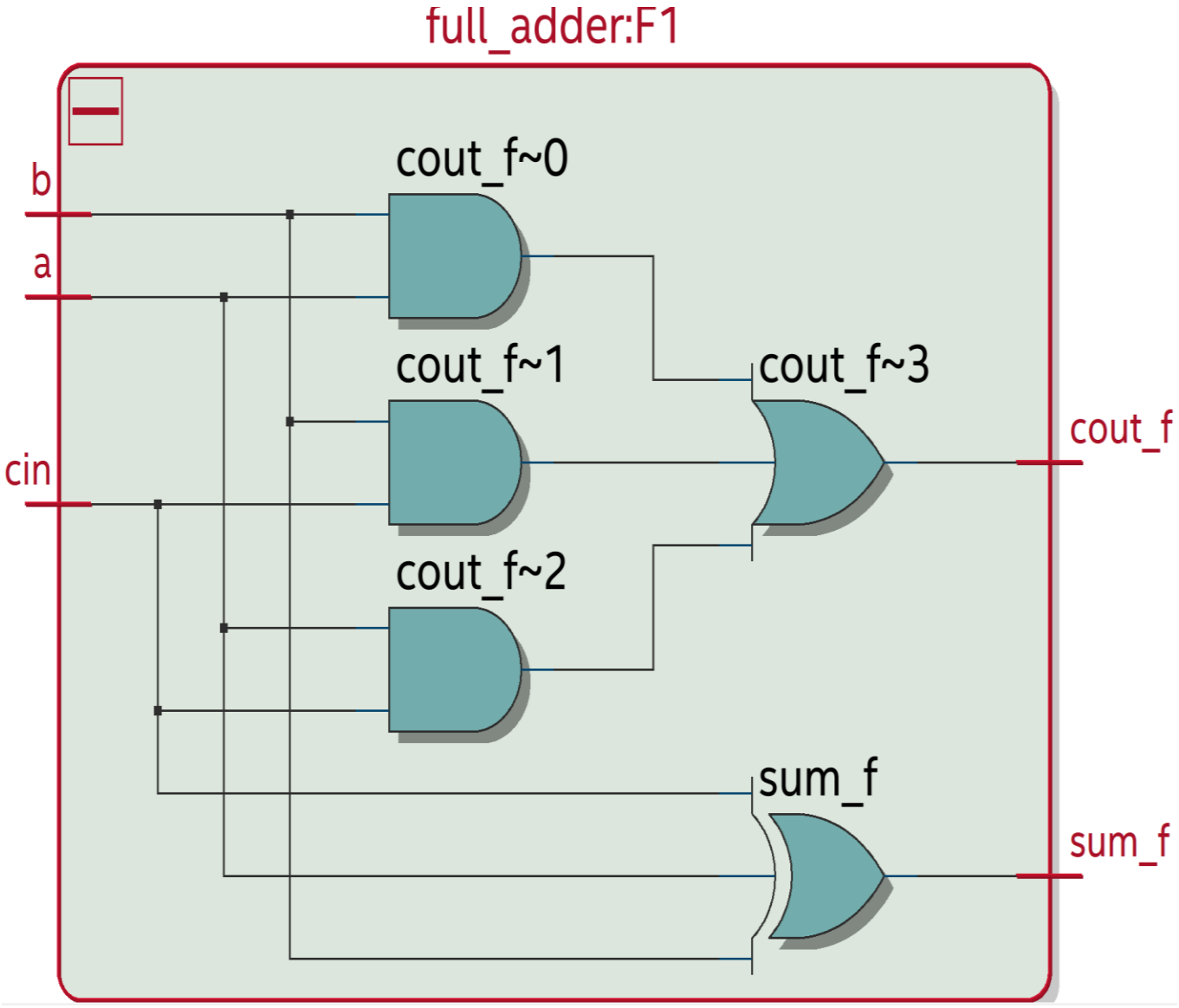


For Full Adder:

CODE SNIPPET:

```
1  module full_adder(input a,b,cin,
2  output reg sum_f,cout_f);
3   always@(*)
4  begin
5   sum_f <= a^b^cin;
6   cout_f <= ((a&b)|(b&cin)|(a&cin));
7  end
8   endmodule
9
```

RTL VIEW:



FOR FULL WALLACE MULTIPLIER:

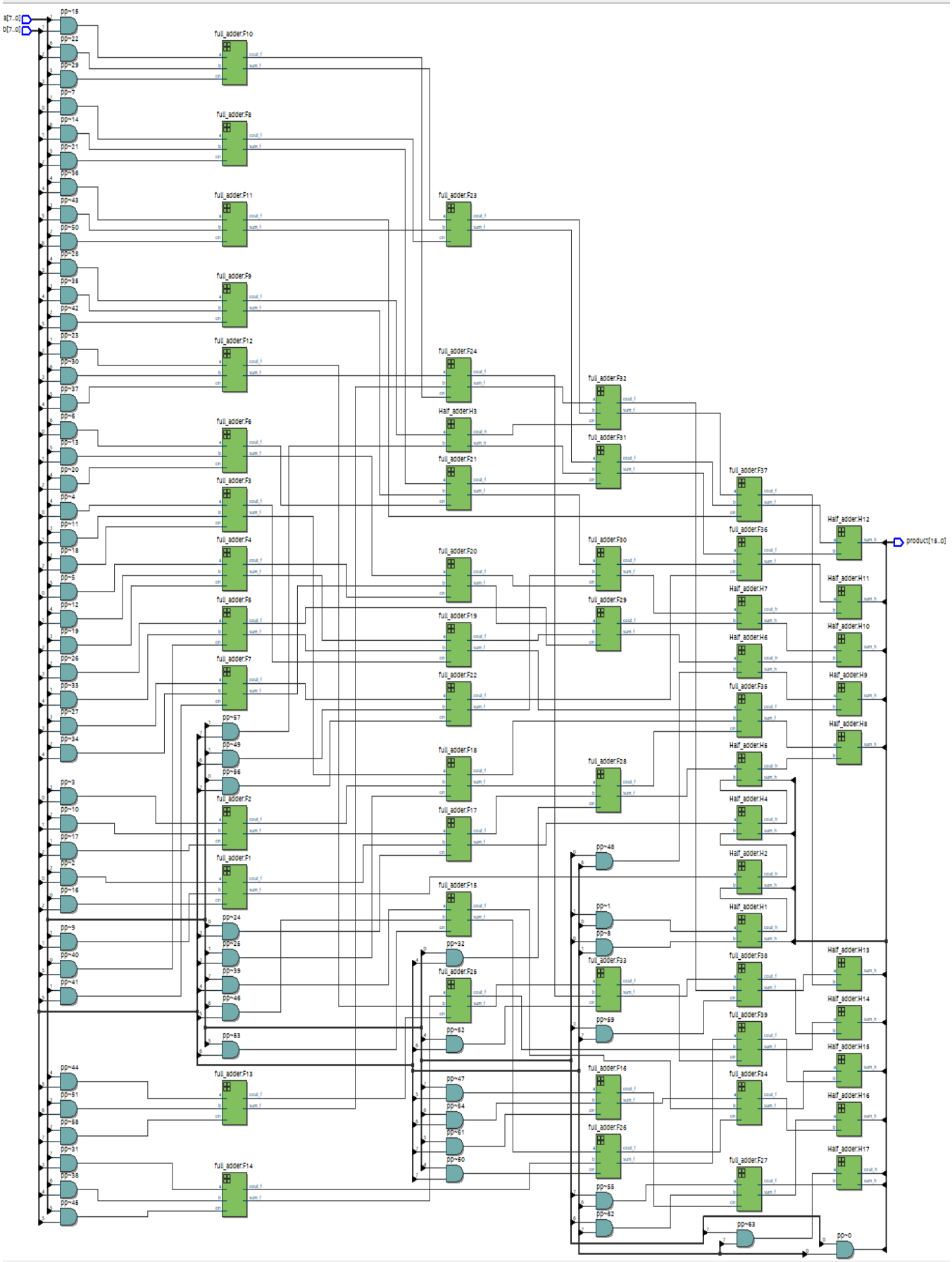
CODE SNIPPET:

```
module wallace_8(input [7:0]a,b,
output reg [15:0]product);

wire h[16:0],f[38:0],fc[38:0],hc[16:0];
reg pp[7:0][7:0];
integer i,j;

always@(*)

begin
for(i=0;i<=7;i=i+1)
  for (j=0;j<=7;j=j+1)
  pp[i][j]<=(a[j]&b[i]);
end

// first level

Half_adder H1(pp[0][1],pp[1][0],h[0],hc[0]);
full_adder F1(pp[0][2],pp[1][1],pp[2][0],f[0],fc[0]);
full_adder F2(pp[0][3],pp[1][2],pp[2][1],f[1],fc[1]);
full_adder F3(pp[0][4],pp[1][3],pp[2][2],f[2],fc[2]);
full_adder F4(pp[0][5],pp[1][4],pp[2][3],f[3],fc[3]);
full_adder F5(pp[3][2],pp[4][1],pp[5][0],f[4],fc[4]);
full_adder F6(pp[0][6],pp[1][5],pp[2][4],f[5],fc[5]);
full_adder F7(pp[3][3],pp[4][2],pp[5][1],f[6],fc[6]);
full_adder F8(pp[0][7],pp[1][6],pp[2][5],f[7],fc[7]);
full_adder F9(pp[3][4],pp[4][3],pp[5][2],f[8],fc[8]);
full_adder F10(pp[1][7],pp[2][6],pp[3][5],f[9],fc[9]);
full_adder F11(pp[4][4],pp[5][3],pp[6][2],f[10],fc[10]);
full_adder F12(pp[2][7],pp[3][6],pp[4][5],f[11],fc[11]);
full_adder F13(pp[5][4],pp[6][3],pp[7][2],f[12],fc[12]);
full_adder F14(pp[3][7],pp[4][6],pp[5][5],f[13],fc[13]);
full_adder F15(pp[4][7],pp[5][6],pp[6][5],f[14],fc[14]);
full_adder F16(pp[5][7],pp[6][6],pp[7][5],f[15],fc[15]);

//second level

Half_adder H2(f[0],hc[0],h[1],hc[1]);
full_adder F17(f[1],fc[0],pp[3][0],f[16],fc[16]);
full_adder F18(f[2],fc[1],pp[3][1],f[17],fc[17]);
full_adder F19(f[3],f[4],fc[2],f[18],fc[18]);
full_adder F20(f[5],f[6],fc[3],f[19],fc[19]);
full_adder F21(f[7],f[8],fc[5],f[20],fc[20]);
full_adder F22(fc[6],pp[6][1],pp[7][0],f[21],fc[21]);
```

```
full_adder F23(f[9],f[10],fc[7],f[22],fc[22]);
Half_adder H3(fc[8],pp[7][1],h[2],hc[2]);
full_adder F24(f[11],f[12],fc[9],f[23],fc[23]);
full_adder F25(f[13],fc[11],fc[12],f[24],fc[24]);
full_adder F26(f[14],fc[13],pp[7][4],f[25],fc[25]);
full_adder F27(pp[6][7],pp[7][6],fc[15],f[26],fc[26]);

//third level

Half_adder H4(f[16],hc[1],h[3],hc[3]);
full_adder F28(f[17],fc[16],pp[4][0],f[27],fc[27]);
full_adder F29(f[19],fc[18],fc[4],f[28],fc[28]);
full_adder F30(f[20],f[21],fc[19],f[29],fc[29]);
full_adder F31(f[22],h[2],fc[20],f[30],fc[30]);
full_adder F32(f[23],fc[22],hc[2],f[31],fc[31]);
full_adder F33(f[24],fc[23],pp[6][4],f[32],fc[32]);
full_adder F34(f[15],fc[14],fc[25],f[33],fc[33]);

//fourth level

Half_adder H5(f[27],hc[3],h[4],hc[4]);
full_adder F35(f[18],fc[17],fc[27],f[34],fc[34]);
Half_adder H6(f[28],pp[6][0],h[5],hc[5]);
Half_adder H7(f[29],fc[28],h[6],hc[6]);
full_adder F36(f[30],fc[29],fc[21],f[35],fc[35]);
full_adder F37(f[31],fc[30],fc[10],f[36],fc[36]);
full_adder F38(f[32],fc[31],pp[7][3],f[37],fc[37]);
full_adder F39(f[25],fc[24],fc[32],f[38],fc[38]);


// last level

Half_adder H8(f[34],hc[4],h[7],hc[7]);
Half_adder H9(h[5],fc[34],h[8],hc[8]);
Half_adder H10(h[6],hc[5],h[9],hc[9]);
Half_adder H11(f[35],hc[6],h[10],hc[10]);
Half_adder H12(f[36],fc[35],h[11],hc[11]);
Half_adder H13(f[37],fc[36],h[12],hc[12]);
Half_adder H14(f[38],fc[37],h[13],hc[13]);
Half_adder H15(f[33],fc[38],h[14],hc[14]);
Half_adder H16(f[26],fc[33],h[15],hc[15]);
Half_adder H17(pp[7][7],fc[26],h[16],hc[16]);
```

```
always@(*)
begin

product[0]<=pp[0][0];
product[1]<=h[0];
product[2]<=h[1];
product[3]<=h[3];
product[4]<=h[4];
product[5]<=h[7];
product[6]<=h[8];
product[7]<=h[9];
product[8]<=h[10];
product[9]<=h[11];
product[10]<=h[12];
product[11]<=h[13];
product[12]<=h[14];
product[13]<=h[15];
product[14]<=h[16];
product[15]<=hc[16];

end

endmodule
```

RTL VIEW:

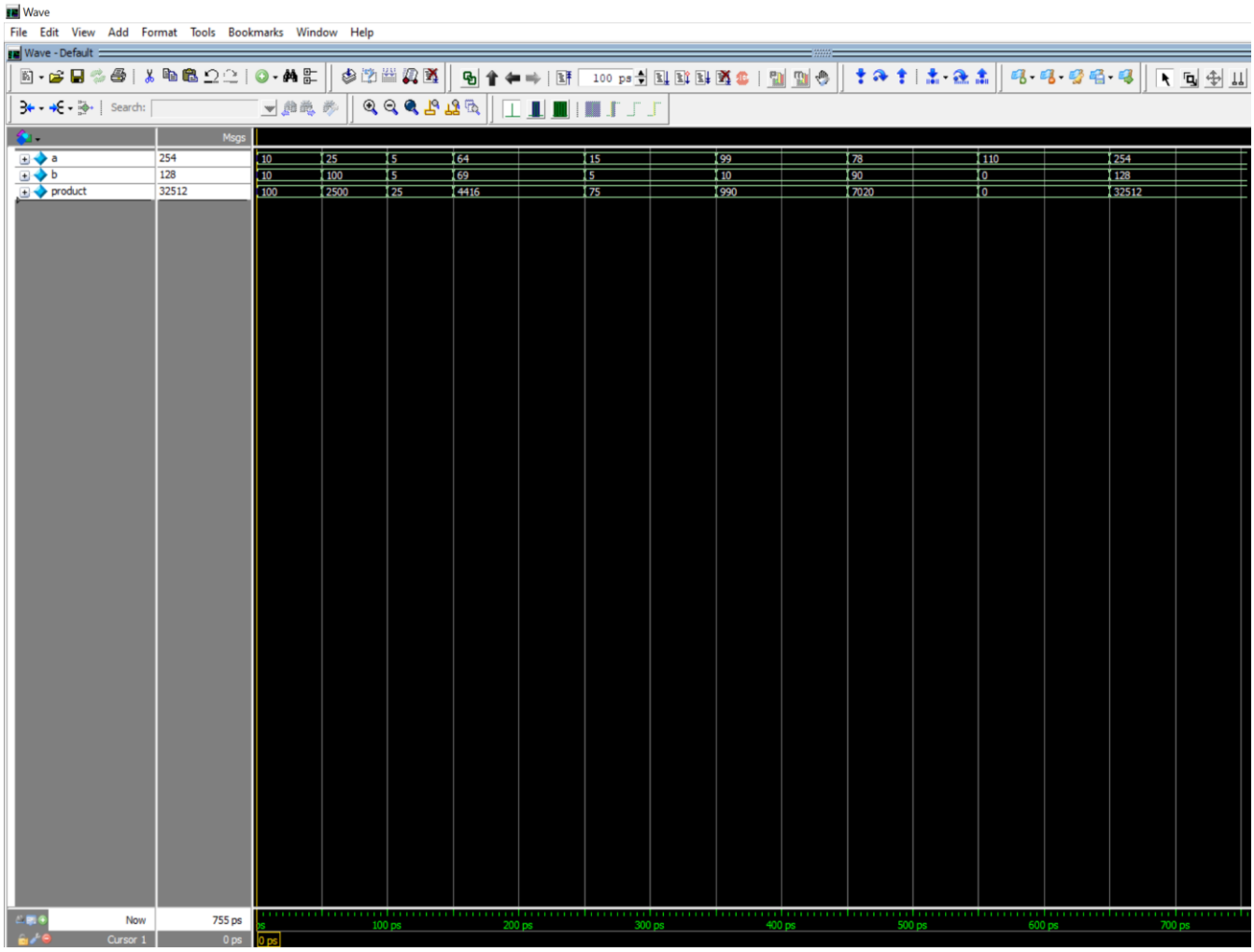


TESTBENCH:

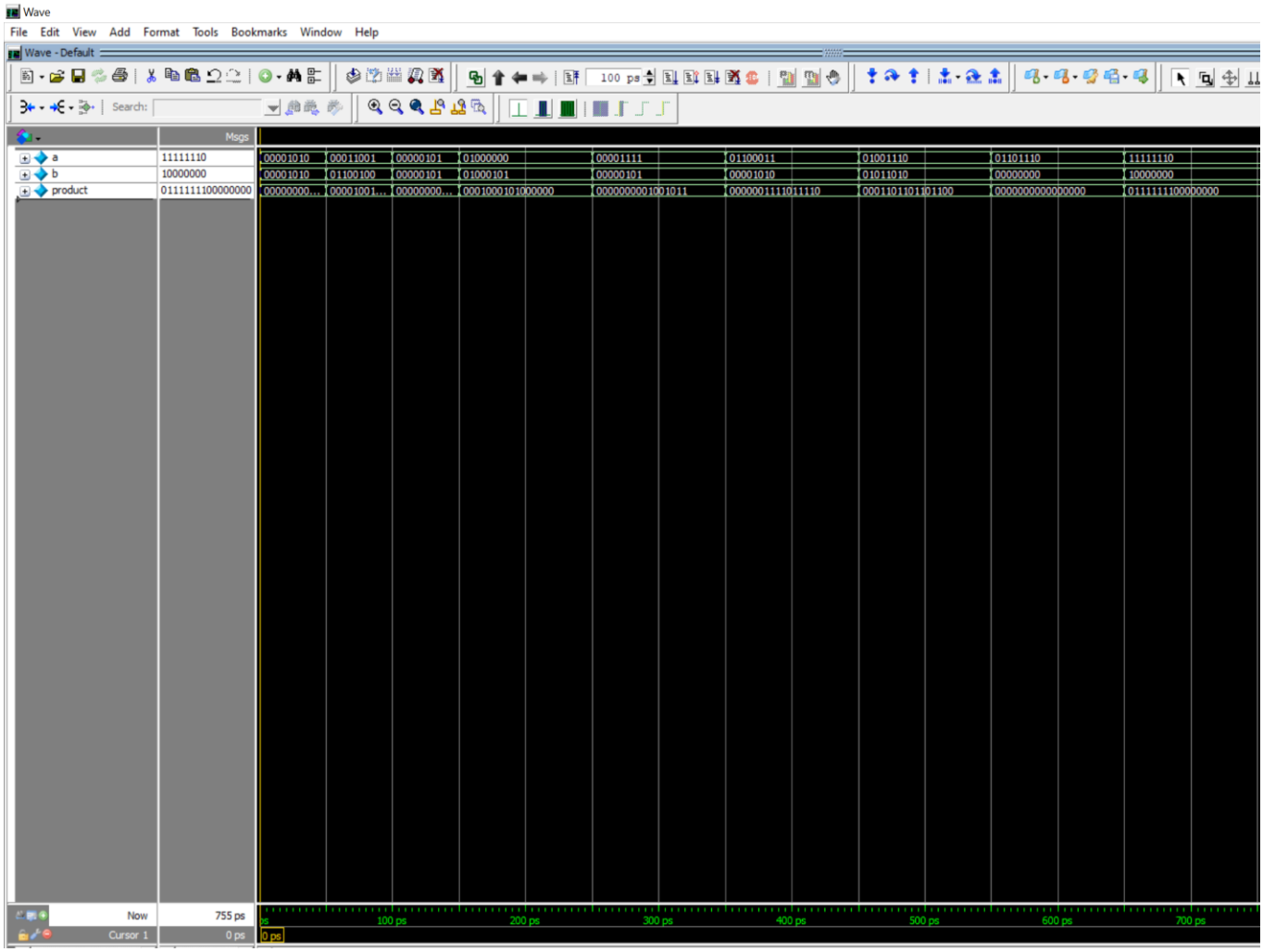
CODE SNIPPET:

```
module wallace_testbench;
reg [7:0]a;
reg [7:0]b;
wire [15:0]product;

wallace_8 DUT (.a(a),.b(b),.product(product));
initial
begin


 a = 8'd10; b = 8'd10; #50;
 a = 8'd25; b = 8'd100; #50;
 a = 8'd5; b = 8'd5; #50;
 a = 8'd64; b = 8'd69; #100;
 a = 8'd15; b = 8'd5; #100;
 a = 8'd99; b = 8'd10; #100;
 a = 8'd78; b = 8'd90; #100;
 a = 8'd110; b = 8'd0; #100;
 a = 8'b11111110; b = 8'b10000000; #100;
 #5 $finish;
end
endmodule
```

Modelsim output in Decimal:



Modelsim output in Binary:



Conclusion:

From the above two waves of modelsim we can see that for multiple values of two 8-bit numbers we are getting correct product results, for Example→ A = 78 & B = 90, Product = 7020. A = 110 & B = 0, Product = 0. A = 254 & B = 128, Product = 32512.

Moreover, we can see that for 8x8 bits multiplication we are getting only bits as output product and **no extra bit is being generated for carryout**, this is the special nature for modified Wallace multiplier.