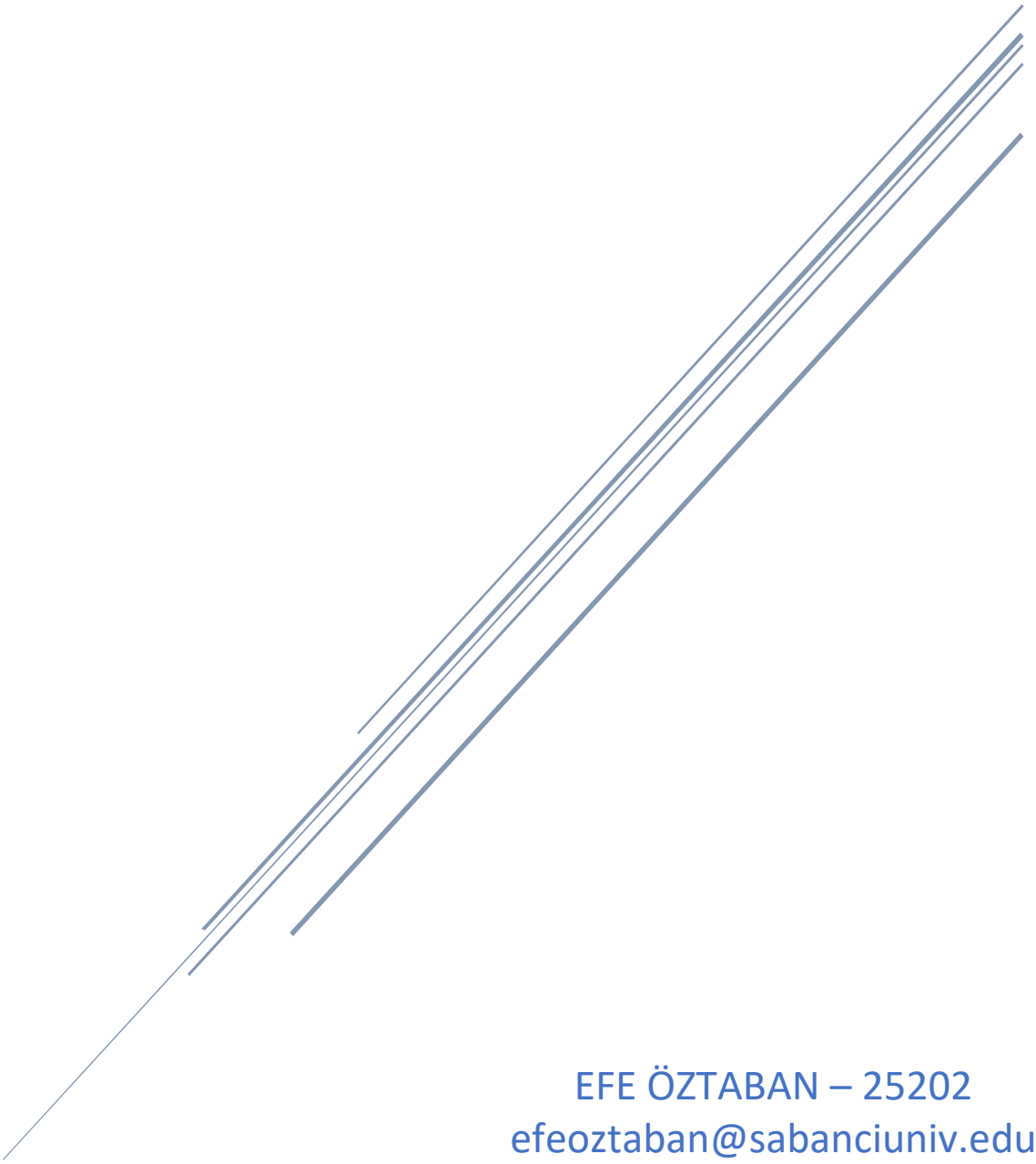
EE 401 VLSI SYSTEM DESIGN I

Lab Report for Lab #1



EFE ÖZTABAN – 25202
efeoztaban@sabanciuniv.edu

KAYRA BİLGİN – 25117
kayrabilgin@sabanciuniv.edu

1. Ripple Carry Adder

a. Implemented Code

This is the Verilog implementation of the Ripple Carry Adder. To implement the adder, firstly, Full Adder is implemented in the gate level:

```
21
22  module FullAdder(A,B,Cin,Cout,C);
23
24     input A,B,Cin;
25
26     output Cout,C;
27
28     wire w1,w2,w3;
29
30     assign w1 = B ^ A;
31     assign w2 = B & A;
32     assign w3 = Cin & w1;
33     assign C = Cin ^ w1;
34     assign Cout = w3 | w2;
35
36  endmodule
```

Then using the Full Adder, full implementation of the 12-bit Ripple Carry Adder is done as shown below:

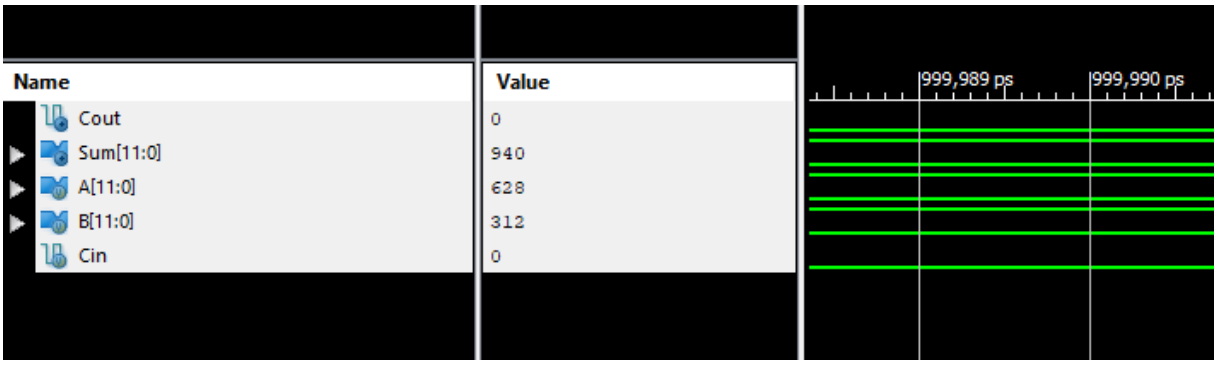
```
39
40  module ripple_carry_12bit(A,B,Cin,Cout,Sum);
41
42     input [11:0] A,B;
43     input Cin;
44
45     output [11:0] Sum;
46     output Cout;
47
48     wire [11:0] w;
49
50     FullAdder FA0 (A[0],B[0],Cin,w[0],Sum[0]);
51     FullAdder FA1 (A[1],B[1],w[0],w[1],Sum[1]);
52     FullAdder FA2 (A[2],B[2],w[1],w[2],Sum[2]);
53     FullAdder FA3 (A[3],B[3],w[2],w[3],Sum[3]);
54     FullAdder FA4 (A[4],B[4],w[3],w[4],Sum[4]);
55     FullAdder FA5 (A[5],B[5],w[4],w[5],Sum[5]);
56     FullAdder FA6 (A[6],B[6],w[5],w[6],Sum[6]);
57     FullAdder FA7 (A[7],B[7],w[6],w[7],Sum[7]);
58     FullAdder FA8 (A[8],B[8],w[7],w[8],Sum[8]);
59     FullAdder FA9 (A[9],B[9],w[8],w[9],Sum[9]);
60     FullAdder FA10 (A[10],B[10],w[9],w[10],Sum[10]);
61     FullAdder FA11 (A[11],B[11],w[10],Cout,Sum[11]);
62
63
64  endmodule
```

Also, the full implementation of the 24-bit Ripple Carry Adder is done as shown below:

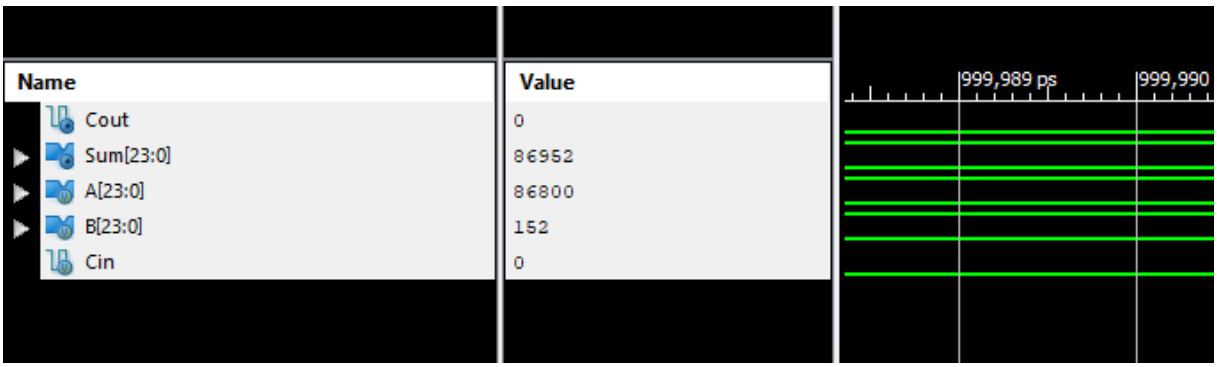
```
40  module ripple_carry_24bit(A,B,Cin,Cout,Sum);
41
42     input [23:0] A,B;
43     input Cin;
44
45     output [23:0] Sum;
46     output Cout;
47
48     wire [23:0] w;
49
50     FullAdder FA0 (A[0],B[0],Cin,w[0],Sum[0]);
51     FullAdder FA1 (A[1],B[1],w[0],w[1],Sum[1]);
52     FullAdder FA2 (A[2],B[2],w[1],w[2],Sum[2]);
53     FullAdder FA3 (A[3],B[3],w[2],w[3],Sum[3]);
54     FullAdder FA4 (A[4],B[4],w[3],w[4],Sum[4]);
55     FullAdder FA5 (A[5],B[5],w[4],w[5],Sum[5]);
56     FullAdder FA6 (A[6],B[6],w[5],w[6],Sum[6]);
57     FullAdder FA7 (A[7],B[7],w[6],w[7],Sum[7]);
58     FullAdder FA8 (A[8],B[8],w[7],w[8],Sum[8]);
59     FullAdder FA9 (A[9],B[9],w[8],w[9],Sum[9]);
60     FullAdder FA10 (A[10],B[10],w[9],w[10],Sum[10]);
61     FullAdder FA11 (A[11],B[11],w[10],w[11],Sum[11]);
62
63     FullAdder FA12 (A[12],B[12],w[11],w[12],Sum[12]);
64     FullAdder FA13 (A[13],B[13],w[12],w[13],Sum[13]);
65     FullAdder FA14 (A[14],B[14],w[13],w[14],Sum[14]);
66     FullAdder FA15 (A[15],B[15],w[14],w[15],Sum[15]);
67     FullAdder FA16 (A[16],B[16],w[15],w[16],Sum[16]);
68     FullAdder FA17 (A[17],B[17],w[16],w[17],Sum[17]);
69     FullAdder FA18 (A[18],B[18],w[17],w[18],Sum[18]);
70     FullAdder FA19 (A[19],B[19],w[18],w[19],Sum[19]);
71     FullAdder FA20 (A[20],B[20],w[19],w[20],Sum[20]);
72     FullAdder FA21 (A[21],B[21],w[20],w[21],Sum[21]);
73     FullAdder FA22 (A[22],B[22],w[21],w[22],Sum[22]);
74     FullAdder FA23 (A[23],B[23],w[22],Cout,Sum[23]);
75
76  endmodule
```

b. Testbench Results

To test the circuit 628+312 = 940 is simulated. This is the testbench result of the 12-bit Ripple Carry Adder:



To test the circuit 86800+152 = 86952 is simulated. This is the testbench result of the 24-bit Ripple Carry Adder:



c. Timing Report

i. For 12-bit implementation

For the synthesis, firstly time = 5 is synthesised:

```
data required time                                 4.80
data arrival time                                 -5.80
-----------------------------------------------------------
slack (VIOLATED)                                  -1.00
```

Because it is violated, new values are tried, and best time is found as 5.84:

```
-------------------------------------------------------------
data required time                                     5.64
data arrival time                                     -5.64
-------------------------------------------------------------
slack (MET)                                            0.00
```

ii. For 24-bit implementation

For the synthesis, firstly time = 10 is synthesised:

```
data required time                                     9.80
data arrival time                                    -11.20
-------------------------------------------------------------
slack (VIOLATED)                                      -1.40
```

Because, it is violated new values are tried, and best time is found as 13.5:

```
-------------------------------------------------------------
data required time                                    13.30
data arrival time                                    -13.30
-------------------------------------------------------------
slack (MET)                                            0.00
```

d. Area Report

i. For 12-bit implementation

```
Total cell area:          315.192001
Total area:               487.455143
```

ii. For 24-bit implementation

```
Total cell area:          620.244000
Total area:               963.414701
```

2. Linear Carry Select Adder

a. Implemented Code

This is the Verilog implementation of the Linear Carry Select Adder. To implement the adder, firstly, Full Adder is implemented in the gate level:

```
module FullAdder(A,B,Cin,Cout,C);

   input A,B,Cin;

   output Cout,C;

   wire w1,w2,w3;

   assign w1 = B ^ A;
   assign w2 = B & A;
   assign w3 = Cin & w1;
   assign C = Cin ^ w1;
   assign Cout = w3 | w2;

endmodule
```

Then 4-bit Ripple Carry Adder is done in the gate level:

```
module RippleCarryAdder4bit(A,B,Cin,Cout,Sum);

   input [3:0] A,B;
   input Cin;

   output [3:0] Sum;
   output Cout;

   wire [2:0] w;

   FullAdder FA0 (A[0],B[0],Cin,w[0],Sum[0]);
   FullAdder FA1 (A[1],B[1],w[0],w[1],Sum[1]);
   FullAdder FA2 (A[2],B[2],w[1],w[2],Sum[2]);
   FullAdder FA3 (A[3],B[3],w[2],Cout,Sum[3]);


endmodule
```

Also 1-bit and 4-bit 2to1 Mux is implemented in the gate level:

```
module Mux2to14bit(In1,In2,s,Out);

    input [3:0] In1,In2;
    input s;

    output [3:0] Out;

    wire s_not;
    wire [3:0] w0,w1;



    assign s_not = ~s;

    assign w0[0] = In1[0] & s_not;
    assign w0[1] = In1[1] & s_not;
    assign w0[2] = In1[2] & s_not;
    assign w0[3] = In1[3] & s_not;

    assign w1[0] = In2[0] & s;
    assign w1[1] = In2[1] & s;
    assign w1[2] = In2[2] & s;
    assign w1[3] = In2[3] & s;

    assign Out[0] = w0[0] | w1[0];
    assign Out[1] = w0[1] | w1[1];
    assign Out[2] = w0[2] | w1[2];
    assign Out[3] = w0[3] | w1[3];


endmodule
```

```
module Mux2to11bit(In1,In2,s,Out);

    input In1,In2;
    input s;

    output Out;

    wire s_not;
    wire w0,w1;



    assign s_not = ~s;

    assign w0 = In1 & s_not;

    assign w1 = In2 & s;

    assign Out = w0 | w1;


endmodule
```

Then using the implemented building blocks, full implementation of the 12-bit Linear Carry Select Adder is done as shown below:

```
module linear_carry_select_12bit(A,B,Cin,Cout,Sum);


   input [11:0] A,B;
   input Cin;

   output [11:0] Sum;
   output Cout;

   wire [11:0] sum0,sum1;
   wire [2:0] cout0,cout1,cout;


   RippleCarryAdder4bit RP0_0 (A[3:0],B[3:0],0,cout0[0],sum0[3:0]);
   RippleCarryAdder4bit RP0_1 (A[3:0],B[3:0],1,cout1[0],sum1[3:0]);
   Mux2to14bit mux0_sum (sum0[3:0],sum1[3:0],Cin,Sum[3:0]);
   Mux2to11bit mux0_cout (cout0[0],cout1[0],Cin,cout[0]);

   RippleCarryAdder4bit RP1_0 (A[7:4],B[7:4],0,cout0[1],sum0[7:4]);
   RippleCarryAdder4bit RP1_1 (A[7:4],B[7:4],1,cout1[1],sum1[7:4]);
   Mux2to14bit mux1_sum (sum0[7:4],sum1[7:4],cout[0],Sum[7:4]);
   Mux2to11bit mux1_cout (cout0[1],cout1[1],cout[0],cout[1]);

   RippleCarryAdder4bit RP2_0 (A[11:8],B[11:8],0,cout0[2],sum0[11:8]);
   RippleCarryAdder4bit RP2_1 (A[11:8],B[11:8],1,cout1[2],sum1[11:8]);
   Mux2to14bit mux2_sum (sum0[11:8],sum1[11:8],cout[1],Sum[11:8]);
   Mux2to11bit mux2_cout (cout0[2],cout1[2],cout[1],Cout);



endmodule
```

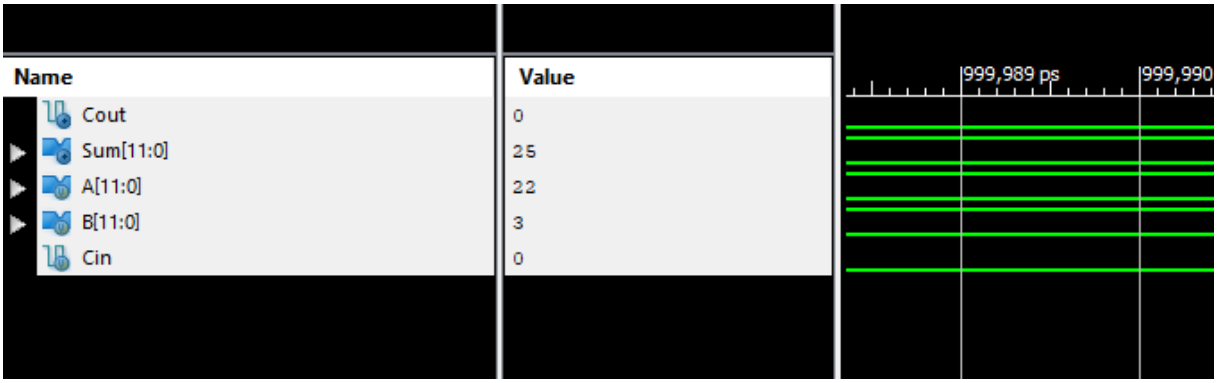
Also, the full implementation of the 24-bit Linear Carry Select Adder is done as shown below:

```
module linear_carry_select_24bit(A,B,Cin,Cout,Sum);


   input [23:0] A,B;
   input Cin;

   output [23:0] Sum;
   output Cout;

   wire [23:0] sum0,sum1;
   wire [5:0] cout0,cout1,cout;


   RippleCarryAdder4bit RP0_0 (A[3:0],B[3:0],0,cout0[0],sum0[3:0]);
   RippleCarryAdder4bit RP0_1 (A[3:0],B[3:0],1,cout1[0],sum1[3:0]);
   Mux2to14bit mux0_sum (sum0[3:0],sum1[3:0],Cin,Sum[3:0]);
   Mux2to11bit mux0_cout (cout0[0],cout1[0],Cin,cout[0]);
```

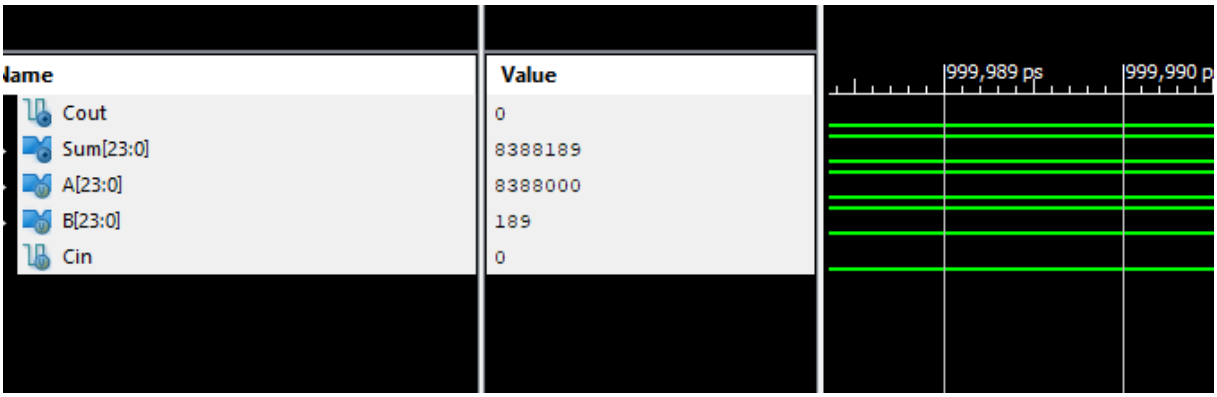
```
RippleCarryAdder4bit RP1_0 (A[7:4],B[7:4],0,cout0[1],sum0[7:4]);
RippleCarryAdder4bit RP1_1 (A[7:4],B[7:4],1,cout1[1],sum1[7:4]);
Mux2to14bit mux1_sum (sum0[7:4],sum1[7:4],cout[0],Sum[7:4]);
Mux2to11bit mux1_cout (cout0[1],cout1[1],cout[0],cout[1]);

RippleCarryAdder4bit RP2_0 (A[11:8],B[11:8],0,cout0[2],sum0[11:8]);
RippleCarryAdder4bit RP2_1 (A[11:8],B[11:8],1,cout1[2],sum1[11:8]);
Mux2to14bit mux2_sum (sum0[11:8],sum1[11:8],cout[1],Sum[11:8]);
Mux2to11bit mux2_cout (cout0[2],cout1[2],cout[1],cout[2]);


RippleCarryAdder4bit RP3_0 (A[15:12],B[15:12],0,cout0[3],sum0[15:12]);
RippleCarryAdder4bit RP3_1 (A[15:12],B[15:12],1,cout1[3],sum1[15:12]);
Mux2to14bit mux3_sum (sum0[15:12],sum1[15:12],cout[2],Sum[15:12]);
Mux2to11bit mux3_cout (cout0[3],cout1[3],cout[2],cout[3]);

RippleCarryAdder4bit RP4_0 (A[19:16],B[19:16],0,cout0[4],sum0[19:16]);
RippleCarryAdder4bit RP4_1 (A[19:16],B[19:16],1,cout1[4],sum1[19:16]);
Mux2to14bit mux4_sum (sum0[19:16],sum1[19:16],cout[3],Sum[19:16]);
Mux2to11bit mux4_cout (cout0[4],cout1[4],cout[3],cout[4]);

RippleCarryAdder4bit RP5_0 (A[23:20],B[23:20],0,cout0[5],sum0[23:20]);
RippleCarryAdder4bit RP5_1 (A[23:20],B[23:20],1,cout1[5],sum1[23:20]);
Mux2to14bit mux5_sum (sum0[23:20],sum1[23:20],cout[4],Sum[23:20]);
Mux2to11bit mux5_cout (cout0[5],cout1[5],cout[4],Cout);


endmodule
```

b. Testbench Results

To test the circuit 22+3 = 25 is simulated. This is the testbench result of the 12-bit Linear Carry Select Adder:



To test the circuit 8388000 + 189 = 8388189 is simulated. This is the testbench result of the 24-bit Linear Carry Select Adder:



c. Timing Report

i. For 12-bit implementation

Different values are tried, and best time is found as 3:

```
-----------------------------------------------------------
data required time                                    2.80
data arrival time                                    -2.80
-----------------------------------------------------------
slack (MET)                                           0.00
```

ii. For 24-bit implementation

Different values are tried, and best time is found as 4:

```
-----------------------------------------------------------
data required time                                    3.80
data arrival time                                    -3.80
-----------------------------------------------------------
slack (MET)                                           0.00
```

d. Area Report

i. For 12-bit implementation

```
Total cell area:                  903.953996
Total area:                       924.307999
1
```

ii. For 24-bit implementation

```
Total cell area:                 1514.139997
Total area:                      1548.271995
1
```

3. Brent-Kung Adder

a. Implemented Code

This is the Verilog implementation of the Brent-Kung Adder for 12-bit. The dot operations in the adder are implemented level by level. Implementation and levels can be seen in the code below:

```
module brent_kung_adder_12bit(a,b,Cin,Cout,Sum);


    input [11:0] a,b;
    input Cin;

    output [11:0] Sum;
    output Cout;

    wire [11:0] p,g,c,gc1,pc1,gc2,pc2,gc3,pc3,gc4,pc4,gc5,pc5;


    // first level

    assign p[0] = a[0] ^ b[0];
    assign g[0] = a[0] & b[0];
    assign pc1[0] = p[0];
    assign gc1[0] = g[0];

    assign p[1] = a[1] ^ b[1];
    assign g[1] = a[1] & b[1];
    assign gc1[1] = (p[1] & g[0])| g[1];
    assign pc1[1] = p[1] & p[0];

    assign p[2] = a[2] ^ b[2];
    assign g[2] = a[2] & b[2];
    assign pc1[2] = p[2];
    assign gc1[2] = g[2];

    assign p[3] = a[3] ^ b[3];
    assign g[3] = a[3] & b[3];
    assign gc1[3] = (p[3] & g[2]) | g[3];
    assign pc1[3] = p[3] & p[2];

    assign p[4] = a[4] ^ b[4];
    assign g[4] = a[4] & b[4];
    assign gc1[4] = g[4];
    assign pc1[4] = p[4];
```

```
    assign p[5] = a[5] ^ b[5];
    assign g[5] = a[5] & b[5];
    assign gc1[5] = (p[5] & g[4]) | g[5];
    assign pc1[5] = p[5] & p[4];

    assign p[6] = a[6] ^ b[6];
    assign g[6] = a[6] & b[6];
    assign gc1[6] = g[6];
    assign pc1[6] = p[6];

    assign p[7] = a[7] ^ b[7];
    assign g[7] = a[7] & b[7];
    assign gc1[7] = (p[7] & g[6]) | g[7];
    assign pc1[7] = p[7] & p[6];
```

```
assign p[8] = a[8] ^ b[8];
assign g[8] = a[8] & b[8];
assign gc1[8] = g[8];
assign pc1[8] = p[8];

assign p[9] = a[9] ^ b[9];
assign g[9] = a[9] & b[9];
assign gc1[9] = (p[9] & g[8]) | g[9];
assign pc1[9] = p[9] & p[8];

assign p[10] = a[10] ^ b[10];
assign g[10] = a[10] & b[10];
assign gc1[10] = g[10];
assign pc1[10] = p[10];

assign p[11] = a[11] ^ b[11];
assign g[11] = a[11] & b[11];
assign gc1[11] = g[11];
assign pc1[11] = p[11];

//second level

assign pc2[2] = pc1[2] & pc1[1];
assign gc2[2] = (pc1[2] & gc1[1]) | gc1[2];

assign gc2[3] = (pc1[3] & gc1[1]) | gc1[3];
assign pc2[3] = pc1[3] & pc1[1];

assign gc2[7] = (pc1[7] & gc1[5]) | gc1[7];
assign pc2[7] = pc1[7] & pc1[5];

assign gc2[11] = (pc1[11] & gc1[9]) | gc1[11];
assign pc2[11] = pc1[11] & pc1[9];
```

```
//third level

assign gc3[4] = (pc1[4] & gc2[3]) | gc1[4];
assign pc3[4] = pc1[4] & pc2[3];

assign gc3[5] = (pc1[5] & gc2[3]) | gc1[5];
assign pc3[5] = pc1[5] & pc2[3];

assign gc3[7] = (pc2[7] & gc2[3]) | gc2[7];
assign pc3[7] = pc2[7] & pc2[3];

//fourth level


assign gc4[6] = (pc1[6] & gc3[5]) | gc1[6];
assign pc4[6] = pc1[6] & pc3[5];

assign gc4[8] = (pc1[8] & gc3[7]) | gc1[8];
assign pc4[8] = pc1[8] & pc3[7];

assign gc4[9] = (pc1[9] & gc3[7]) | gc1[9];
assign pc4[9] = pc1[9] & pc3[7];

assign gc4[11] = (pc2[11] & gc3[7]) | gc2[11];
assign pc4[11] = pc2[11] & pc3[7];
```

```
    //fifth level

    assign gc5[10] = (pc1[10] & gc4[9]) | gc1[10];
    assign pc5[10] = pc1[10] & pc4[9];

    //c

    assign c[1] = g[0] | (p[0] & Cin);
    assign c[2] = gc1[1] | (pc1[1] & Cin);
    assign c[3] = gc2[2] | (pc2[2] & Cin);
    assign c[4] = gc2[3] | (pc2[3] & Cin);
    assign c[5] = gc3[4] | (pc3[4] & Cin);
    assign c[6] = gc3[5] | (pc3[5] & Cin);
    assign c[7] = gc4[6] | (pc4[6] & Cin);
    assign c[8] = gc3[7] | (pc3[7]  & Cin);
    assign c[9] = gc4[8] | (pc4[8] & Cin);
    assign c[10] = gc4[9] | (pc4[9] & Cin);
    assign c[11] = gc5[10] | (pc5[10] & Cin);

    assign Cout = gc4[11] | (pc4[11] & Cin);

    //sums

    assign Sum[0] = p[0] ^ Cin;
    assign Sum[1] = p[1] ^ c[1];
    assign Sum[2] = p[2] ^ c[2];
    assign Sum[3] = p[3] ^ c[3];
    assign Sum[4] = p[4] ^ c[4];
    assign Sum[5] = p[5] ^ c[5];
    assign Sum[6] = p[6] ^ c[6];
    assign Sum[7] = p[7] ^ c[7];
    assign Sum[8] = p[8] ^ c[8];
    assign Sum[9] = p[9] ^ c[9];
    assign Sum[10] = p[10] ^ c[10];
    assign Sum[11] = p[11] ^ c[11];


endmodule
```

Also, the full implementation of the 24-bit Brent-Kung Adder is done as shown below:

```
module brent_kung_adder_24bit(a,b,Cin,Cout,Sum);


  input [23:0] a,b;
  input Cin;

  output [23:0] Sum;
  output Cout;

  wire [23:0] p,g,c,gcl,pcl,gc2,pc2,gc3,pc3,gc4,pc4,gc5,pc5,gc6,pc6;


  // first level

  assign p[0] = a[0] ^ b[0];
  assign g[0] = a[0] & b[0];
  assign pcl[0] = p[0];
  assign gcl[0] = g[0];

  assign p[1] = a[1] ^ b[1];
  assign g[1] = a[1] & b[1];
  assign gcl[1] = (p[1] & g[0])| g[1];
  assign pcl[1] = p[1] & p[0];

  assign p[2] = a[2] ^ b[2];
  assign g[2] = a[2] & b[2];
  assign pcl[2] = p[2];
  assign gcl[2] = g[2];

  assign p[3] = a[3] ^ b[3];
  assign g[3] = a[3] & b[3];
  assign gcl[3] = (p[3] & g[2]) | g[3];
  assign pcl[3] = p[3] & p[2];

  assign p[4] = a[4] ^ b[4];
  assign g[4] = a[4] & b[4];
  assign gcl[4] = g[4];
  assign pcl[4] = p[4];
```

```
  assign p[5] = a[5] ^ b[5];
  assign g[5] = a[5] & b[5];
  assign gcl[5] = (p[5] & g[4]) | g[5];
  assign pcl[5] = p[5] & p[4];

  assign p[6] = a[6] ^ b[6];
  assign g[6] = a[6] & b[6];
  assign gcl[6] = g[6];
  assign pcl[6] = p[6];

  assign p[7] = a[7] ^ b[7];
  assign g[7] = a[7] & b[7];
  assign gcl[7] = (p[7] & g[6]) | g[7];
  assign pcl[7] = p[7] & p[6];
```

```
assign p[8] = a[8] ^ b[8];
assign g[8] = a[8] & b[8];
assign gcl[8] = g[8];
assign pcl[8] = p[8];

assign p[9] = a[9] ^ b[9];
assign g[9] = a[9] & b[9];
assign gcl[9] = (p[9] & g[8]) | g[9];
assign pcl[9] = p[9] & p[8];

assign p[10] = a[10] ^ b[10];
assign g[10] = a[10] & b[10];
assign gcl[10] = g[10];
assign pcl[10] = p[10];

assign p[11] = a[11] ^ b[11];
assign g[11] = a[11] & b[11];
assign gcl[11] = g[11];
assign pcl[11] = p[11];

assign p[12] = a[12] ^ b[12];
assign g[12] = a[12] & b[12];
assign pcl[12] = p[12];
assign gcl[12] = g[12];

assign p[13] = a[13] ^ b[13];
assign g[13] = a[13] & b[13];
assign gcl[13] = (p[13] & g[13])| g[13];
assign pcl[13] = p[13] & p[13];

assign p[14] = a[14] ^ b[14];
assign g[14] = a[14] & b[14];
assign pcl[14] = p[14];
assign gcl[14] = g[14];

assign p[15] = a[15] ^ b[15];
assign g[15] = a[15] & b[15];
assign gcl[15] = (p[15] & g[15]) | g[15];
assign pcl[15] = p[15] & p[15];


assign p[16] = a[16] ^ b[16];
assign g[16] = a[16] & b[16];
assign gcl[16] = g[16];
assign pcl[16] = p[16];

assign p[17] = a[17] ^ b[17];
assign g[17] = a[17] & b[17];
assign gcl[17] = (p[17] & g[17]) | g[17];
assign pcl[17] = p[17] & p[17];

assign p[18] = a[18] ^ b[18];
assign g[18] = a[18] & b[18];
assign gcl[18] = g[18];
assign pcl[18] = p[18];
```

```
assign p[19] = a[19] ^ b[19];
assign g[19] = a[19] & b[19];
assign gcl[19] = (p[19] & g[19]) | g[19];
assign pcl[19] = p[19] & p[19];

assign p[20] = a[20] ^ b[20];
assign g[20] = a[20] & b[20];
assign gcl[20] = g[20];
assign pcl[20] = p[20];

assign p[21] = a[21] ^ b[21];
assign g[21] = a[21] & b[21];
assign gcl[21] = (p[21] & g[21]) | g[21];
assign pcl[21] = p[21] & p[21];

assign p[22] = a[22] ^ b[22];
assign g[22] = a[22] & b[22];
assign gcl[22] = g[22];
assign pcl[22] = p[22];

assign p[23] = a[23] ^ b[23];
assign g[23] = a[23] & b[23];
assign gcl[23] = g[23];
assign pcl[23] = p[23];

//second level

assign pc2[2] = pcl[2] & pcl[1];
assign gc2[2] = (pcl[2] & gcl[1]) | gcl[2];

assign gc2[3] = (pcl[3] & gcl[1]) | gcl[3];
assign pc2[3] = pcl[3] & pcl[1];

assign gc2[7] = (pcl[7] & gcl[5]) | gcl[7];
assign pc2[7] = pcl[7] & pcl[5];

assign gc2[11] = (pcl[11] & gcl[9]) | gcl[11];
assign pc2[11] = pcl[11] & pcl[9];

assign pc2[15] = pcl[15] & pcl[13];
assign gc2[15] = (pcl[15] & gcl[13]) | gcl[15];



 assign gc2[18] = (pcl[18] & gcl[17]) | gcl[18];
 assign pc2[18] = pcl[18] & pcl[17];

 assign gc2[19] = (pcl[19] & gcl[17]) | gcl[19];
 assign pc2[19] = pcl[19] & pcl[17];

 assign gc2[23] = (pcl[23] & gcl[21]) | gcl[23];
 assign pc2[23] = pcl[23] & pcl[21];
```

```
//third level

assign gc3[4] = (pc1[4] & gc2[3]) | gc1[4];
assign pc3[4] = pc1[4] & pc2[3];

assign gc3[5] = (pc1[5] & gc2[3]) | gc1[5];
assign pc3[5] = pc1[5] & pc2[3];

assign gc3[7] = (pc2[7] & gc2[3]) | gc2[7];
assign pc3[7] = pc2[7] & pc2[3];

assign gc3[15] = (pc2[15] & gc2[11]) | gc2[15];
assign pc3[15] = pc2[15] & pc2[11];

assign gc3[20] = (pc1[20] & gc2[19]) | gc1[20];
assign pc3[20] = pc1[20] & pc2[19];

assign gc3[21] = (pc1[19] & gc2[19]) | gc1[19];
assign pc3[21] = pc1[19] & pc2[19];

assign gc3[23] = (pc2[23] & gc2[19]) | gc2[23];
assign pc3[23] = pc2[23] & pc2[19];

//fourth level


assign gc4[6] = (pc1[6] & gc3[5]) | gc1[6];
assign pc4[6] = pc1[6] & pc3[5];

assign gc4[8] = (pc1[8] & gc3[7]) | gc1[8];
assign pc4[8] = pc1[8] & pc3[7];

assign gc4[9] = (pc1[9] & gc3[7]) | gc1[9];
assign pc4[9] = pc1[9] & pc3[7];

assign gc4[11] = (pc2[11] & gc3[7]) | gc2[11];
assign pc4[11] = pc2[11] & pc3[7];

assign gc4[15] = (pc3[15] & gc3[7]) | gc3[15];
assign pc4[15] = pc3[15] & pc3[7];

assign gc4[22] = (pc1[22] & gc3[21]) | gc1[22];
assign pc4[22] = pc1[22] & pc3[21];
```

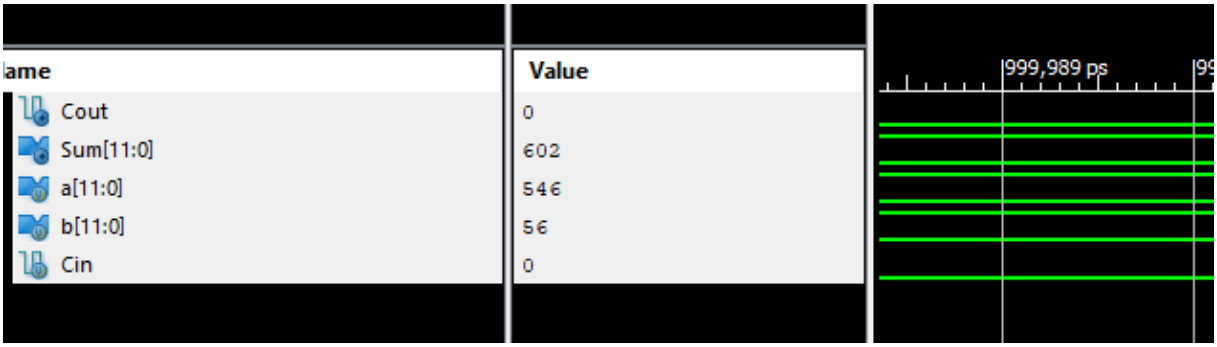
```
//fifth level

assign gc5[10] = (pc1[10] & gc4[9]) | gc1[10];
assign pc5[10] = pc1[10] & pc4[9];

assign gc5[12] = (pc1[12] & gc4[11]) | gc1[12];
assign pc5[12] = pc1[12] & pc4[11];

assign gc5[13] = (pc1[13] & gc4[11]) | gc1[13];
assign pc5[13] = pc1[13] & pc4[11];

//sixth level

assign gc6[14] = (pc1[14] & gc5[13]) | gc1[14];
assign pc6[14] = pc1[14] & pc5[13];
```

```
    //c

    assign c[1] = g[0] | (p[0] & Cin);
    assign c[2] = gc1[1] | (pc1[1] & Cin);
    assign c[3] = gc2[2] | (pc2[2] & Cin);
    assign c[4] = gc2[3] | (pc2[3] & Cin);
    assign c[5] = gc3[4] | (pc3[4] & Cin);
    assign c[6] = gc3[5] | (pc3[5] & Cin);
    assign c[7] = gc4[6] | (pc4[6] & Cin);
    assign c[8] = gc3[7] | (pc3[7]  & Cin);
    assign c[9] = gc4[8] | (pc4[8] & Cin);
    assign c[10] = gc4[9] | (pc4[9] & Cin);
    assign c[11] = gc5[10] | (pc5[10] & Cin);

    assign c[12] = gc4[11] | (pc4[11] & Cin);
    assign c[13] = gc5[12] | (pc5[12] & Cin);
    assign c[14] = gc5[13] | (pc5[13] & Cin);
    assign c[15] = gc6[14] | (pc6[14] & Cin);
    assign c[16] = gc4[15] | (pc4[15] & Cin);
    assign c[17] = gc1[16] | (pc1[16] & Cin);
    assign c[18] = gc1[17] | (pc1[17] & Cin);
    assign c[19] = gc2[18] | (pc2[18]  & Cin);
    assign c[20] = gc2[19] | (pc2[19] & Cin);
    assign c[21] = gc3[20] | (pc3[20] & Cin);
    assign c[22] = gc3[21] | (pc3[21] & Cin);
    assign c[23] = gc4[22] | (pc4[22] & Cin);

    assign Cout = gc3[23] | (pc3[23] & Cin);
```

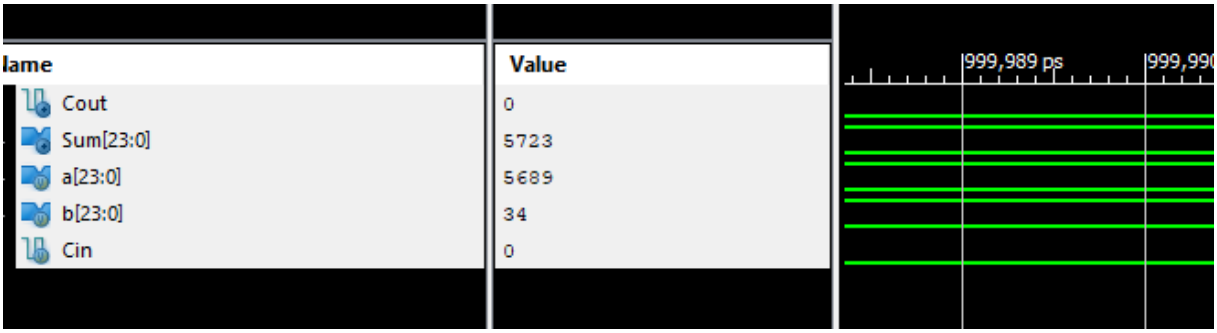
```
    //sums

    assign Sum[0] = p[0] ^ Cin;
    assign Sum[1] = p[1] ^ c[1];
    assign Sum[2] = p[2] ^ c[2];
    assign Sum[3] = p[3] ^ c[3];
    assign Sum[4] = p[4] ^ c[4];
    assign Sum[5] = p[5] ^ c[5];
    assign Sum[6] = p[6] ^ c[6];
    assign Sum[7] = p[7] ^ c[7];
    assign Sum[8] = p[8] ^ c[8];
    assign Sum[9] = p[9] ^ c[9];
    assign Sum[10] = p[10] ^ c[10];
    assign Sum[11] = p[11] ^ c[11];

    assign Sum[12] = p[12] ^ c[12];
    assign Sum[13] = p[13] ^ c[13];
    assign Sum[14] = p[14] ^ c[14];
    assign Sum[15] = p[15] ^ c[15];
    assign Sum[16] = p[16] ^ c[16];
    assign Sum[17] = p[17] ^ c[17];
    assign Sum[18] = p[18] ^ c[18];
    assign Sum[19] = p[19] ^ c[19];
    assign Sum[20] = p[20] ^ c[20];
    assign Sum[21] = p[21] ^ c[21];
    assign Sum[22] = p[22] ^ c[22];
    assign Sum[23] = p[23] ^ c[23];

endmodule
```

b. Testbench Results

To test the circuit 546+56 = 602 is simulated. This is the testbench result of the 12-bit Brent-Kung Adder:



To test the circuit 5689+34 = 5723 is simulated. This is the testbench result of the 24-bit Brent-Kung Adder:



c. Timing Report

i. For 12-bit implementation

Different values are tried, and best time is found as 1.9:

```
-------------------------------------------------------------
data required time                                    1.70
data arrival time                                    -1.70
-------------------------------------------------------------
slack (MET)                                           0.00
```

ii. For 24-bit implementation

Different values are tried, and best time is found as 2.5:

```
-----------------------------------------------------------
data required time                                  2.30
data arrival time                                  -2.30
-----------------------------------------------------------
slack (VIOLATED: increase significant digits)       0.00
```

d. Area Report

i. For 12-bit implementation

```
Total cell area:                2115.978017
Total area:                     2185.599389
```

ii. For 24-bit implementation

```
Total cell area:                2916.768025
Total area:                     3007.776662
```

4. Discussion about comparison of circuits about timing and area

Timing of;

- 12-bit Ripple Carry Adder: 5.84
- 24-bit Ripple Carry Adder: 13.5
- 12-bit Linear Carry Select Adder: 3
- 24-bit Linear Carry Select Adder: 4
- 12-bit Brent-Kung Adder: 1.9
- 24-bit Brent-Kung Adder: 2.5

Area of;

- 12-bit Ripple Carry Adder: 487
- 24-bit Ripple Carry Adder: 963
- 12-bit Linear Carry Select Adder: 924
- 24-bit Linear Carry Select Adder: 1548
- 12-bit Brent-Kung Adder: 2185
- 24-bit Brent-Kung Adder: 3007

- With the designs and the simulation results following discussions can be made:

    - As it is expected, when the adder works with bigger number of bits area and time is increasing for the same adder design.
    - Linear Carry Adder works faster than the Ripple Carry Adder because it is calculating both carry situations in spite of waiting. When the carry information arrives, it just selects the correct results with a multiplexer. However, its area is larger than the Ripple Carry Adder because of the extra circuit.
    - Brent-Kung Adder works faster than both of the adder designs. Because, it is calculating all the propagate, generate and carry information in parallel. Then calculates the sums. That's why it is the fastest design. However, because of these conditions, this design has the largest circuit area.