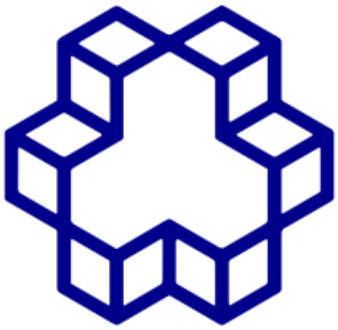
به نام خدا



دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی برق و کامپیوتر

ترم ۴۰۱۲

درس: معماری کامپیوتر

گزارش پروژه دوم

مجتبی هادئی

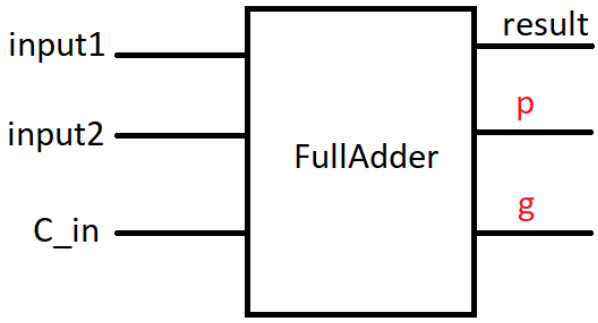
شماره دانشجویی: 40010303

سوالات

1. Carry Lookahead Adder 64 bit

پاسخ:

در ابتدا یک تمام جمع کننده 1 بیتی ساخته شده است. که ورودی های مورد نیاز ماژول Lookahead (که در ادامه توضیح داده خواهد شد) که همان p , g هستند را هم میسازد.



همانطور که در شکل بالا مشاهده می شود ورودی های تمام جمع کننده دو بیت داده ی input1 و input2 و یک c_in می باشد که برای دریافت بیت نقلی به کار می رود. سپس این سه بیت ورودی باهم جمع می شوند و خروجی این مدار result ، p و g می باشد که با توجه به جدول درستی زیر مشخص می شوند

Input1	Input2	C_in	result	p	g
0	0	0	0	0	**0**
0	0	1	1	0	**0**
0	1	0	1	1	**0**
0	1	1	0	1	**0**
1	0	0	1	1	**0**
1	0	1	0	1	**0**
1	1	0	0	0	**1**
1	1	1	1	0	**1**

g = and(input1, input2) , p = xor(input1, input2).

با توجه به توضیحات داده شده پیاده سازی در نرم افزار به صورت زیر خواهد بود :

نام فایل: FullAdder.v

```
Ln#
 1  `timescale 1ns/1ns
 2  module FullAdder (input1, input2, result, cin, p, g);
 3      input input1;
 4      input input2;
 5      output reg result;
 6      input cin;
 7      output reg p;
 8      output reg g;
 9      always @(*) begin
10          p <= input1^input2;
11          g <= input1 && input2;
12          result <= input1^input2^cin;
13      end
14  endmodule
```

برای پیاده سازی CLA_64_bit ما از چهار CLA_16_bit و یک CarryLookahead استفاده کرده ایم که خود CLA_16_bit از چهار CLA_4_bit و یک CarryLookahead ایجاد شده و هر کدام از CLA_4_bit ها خود از 4 FullAdder و یک CarryLookahead ساخته شده است.

پیاده سازی CarryLookahead با نام فایل CarryLookahead.v به شرح زیر است:

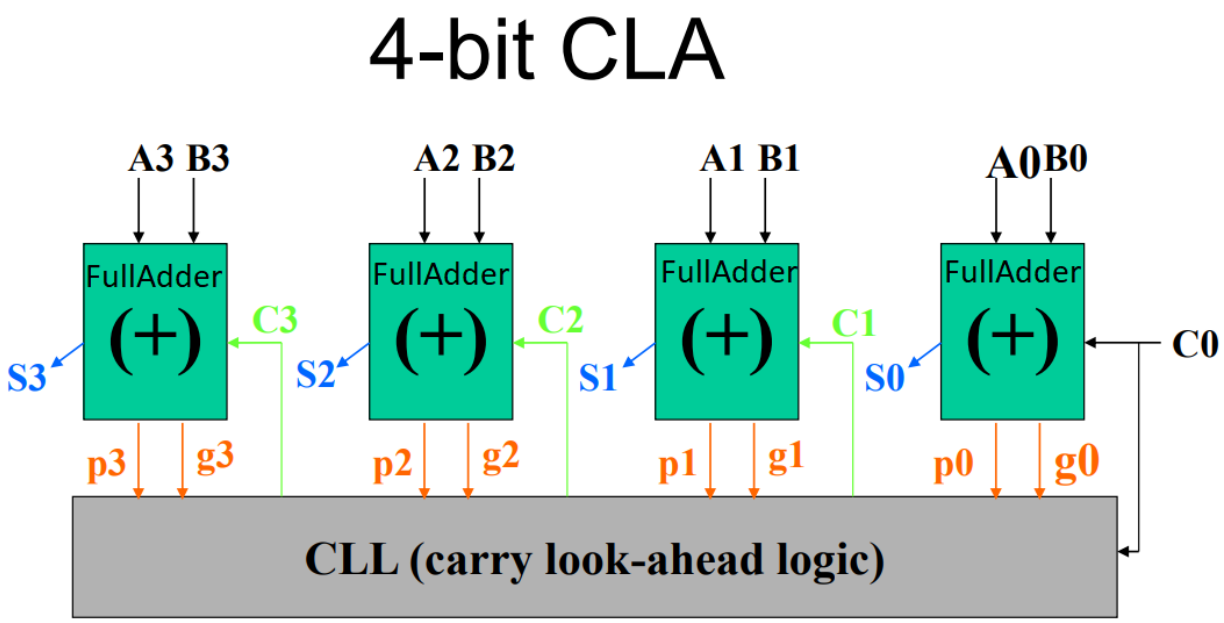
```
Ln#
 1  `timescale 1ns/1ps
 2  module CarryLookahead (p, g, cin, c, c4, pout, gout);
 3      input [3:0] p;
 4      input [3:0] g;
 5      input cin;
 6      output reg [3:1] c;
 7      output reg c4;
 8      output reg pout;
 9      output reg gout;
10      always @(*) begin
11              c[1] <= g[0] | (p[0]&cin);
12              c[2] <= g[1] | (p[1]&g[0]) | (p[1]&p[0]&&cin);
13              c[3] <= g[2] | (p[2]&&g[1]) | (p[2]&p[1]&g[0]) | (p[2]&p[1]&p[0]&cin);
14              c4 <= g[3] | (p[3]&g[2]) || (p[3]&p[2]&g[1]) | (p[3]&p[2]&p[1]&g[0]) | (p[3]&p[2]&p[1]&p[0]&cin);
15              pout <= p[0]&p[1]&p[2]&p[3];
16              gout <= g[3] | (g[2]&p[3]) || (g[1]&p[3]&p[2]) | (g[0]&p[3]&p[2]&p[1]);
17      end
18  endmodule
```

این ماژول p ، g و Cin که همان رقم نقلی ورودی به مجموعه CLA هست را میگیرد و با توجه به روابط زیر carray های ورودی به مجموعه های ریزتر مجموعه را تولید میکند.

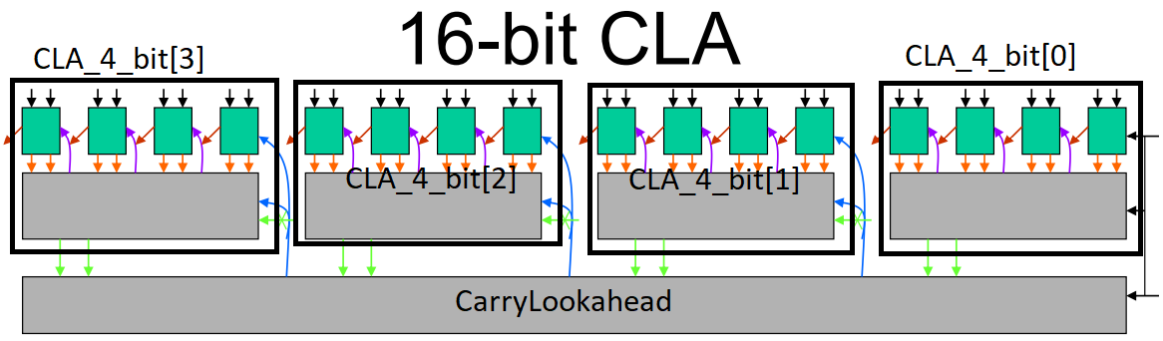
- $C1=g_0+p_0C_0$,
- $C2=g_1+p_1g_0+p_1p_0C_0$,
- $C3=g_2+p_2g_1+p_2p_1g_0+p_2p_1p_0c_0$,
- $C4=g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0+p_3p_2p_1p_0c_0$

در نهایت pout وgout خروجی هایی هستند که در صورت وجود سطوح بالاترCLA برای CarryLookahead آنها تولید میشوند و به آن منتقل میشوند.

در زیر شمای کلیCarry-Lookahead-Adder به نمایش گذاشته میشود.



وCLA-16-bit در زیر:



شکل کلی CLA_64_bit هم مانند شکل CLA_16_bit است با این تفاوت که به جای CLA_4_bit از CLA_16_bit استفاده میشود.

در زیر کد های مربوط به CLA ها را میبینیم.

CLA_4_bit.v :

```
`timescale 1ns/1ps
module CLA_4_bit(input1, input2, cin, result, cout, pout, gout);
    parameter N = 4;
    input [N-1:0] input1;
    input [N-1:0] input2;
    input cin;
    output [N-1:0] result;
    output cout;
    output pout;
    output gout;
    wire [N-1:0] p;
    wire [N-1:0] g;
    wire [N-1:1] c;
    CarryLookahead carrylookahead (
                .cin(cin),
                .p(p),
                .g(g),
                .c(c),
                .c4(cout),
                .pout(pout),
                .gout(gout)
        );
        FullAdder uut0 (
                .input1(input1[0]),
                .input2(input2[0]),
                .result(result[0]),
                .cin(cin),
                .p(p[0]),
                .g(g[0])
        );
        genvar i;
        for(i = 1; i < N; i = i + 1) begin
                FullAdder uut1 (
                        .input1(input1[i]),
                        .input2(input2[i]),
                        .result(result[i]),
                        .cin(c[i]),
                        .p(p[i]),
                        .g(g[i])
                );
        end
endmodule
```

CLA_4_bit شامل 2 ورودی 4 بیتی input1 و input2 و یک ورودی یک بیتی cin هست.

همچنین شامل 4 خروجی result (که نتیجه جمع را در خود نگه میدارد)، cout که رقم نقلی جمع را نگه میدارد، pout و gout که در lookahead برای سطوح بالاتر (در صورت وجود) تولید میشوند و امکان توسعه سخت افزار را ایجاد میکنند.

همچنین سیم های درونی p ,g, c برای انتقال p, g های تولید شده توسط FullAdder ها به carrylookahead و انتقال carry های تولید شده توسط carrylookahead به FullAdder ها استفاده میشود.

CLA_16_bit.v :

```
Ln#
 3  module CLA_16_bit(input1, input2, cin, result, cout, pout, gout);
 4          parameter N = 16;
 5          input [N-1:0] input1;
 6          input [N-1:0] input2;
 7          output [N-1:0] result;
 8          input cin;
 9          output cout;
10          output pout;
11          output gout;
12          wire [3:0] p;
13          wire [3:0] g;
14          wire [3:1] c;
15          CarryLookahead carrylookahead2 (
16                  .cin(cin),
17                  .p(p),
18                  .g(g),
19                  .c(c),
20                  .c4(cout),
21                  .pout(pout),
22                  .gout(gout)
23          );
24          CLA_4_bit f0(
25                  .input1(input1[(N/4)-1:0]),
26                  .input2(input2[(N/4)-1:0]),
27                  .result(result[(N/4)-1:0]),
28                  .cin(cin),
29                  .cout(),
30                  .pout(p[0]),
31                  .gout(g[0])
32          );
33          genvar i;
34          generate
35            for(i=1; i<4; i=i+1) begin
36                  CLA_4_bit f1(
37                          .input1(input1[(N/4)*(i+1)-1:(N/4)*i]),
38                          .input2(input2[(N/4)*(i+1)-1:(N/4)*i]),
39                          .result(result[(N/4)*(i+1)-1:(N/4)*i]),
40                          .cin(c[i]),
41                          .cout(),
42                          .pout(p[i]),
43                          .gout(g[i])
44                  );
45            end
46          endgenerate
47  endmodule
```

CLA_64_bit.v :

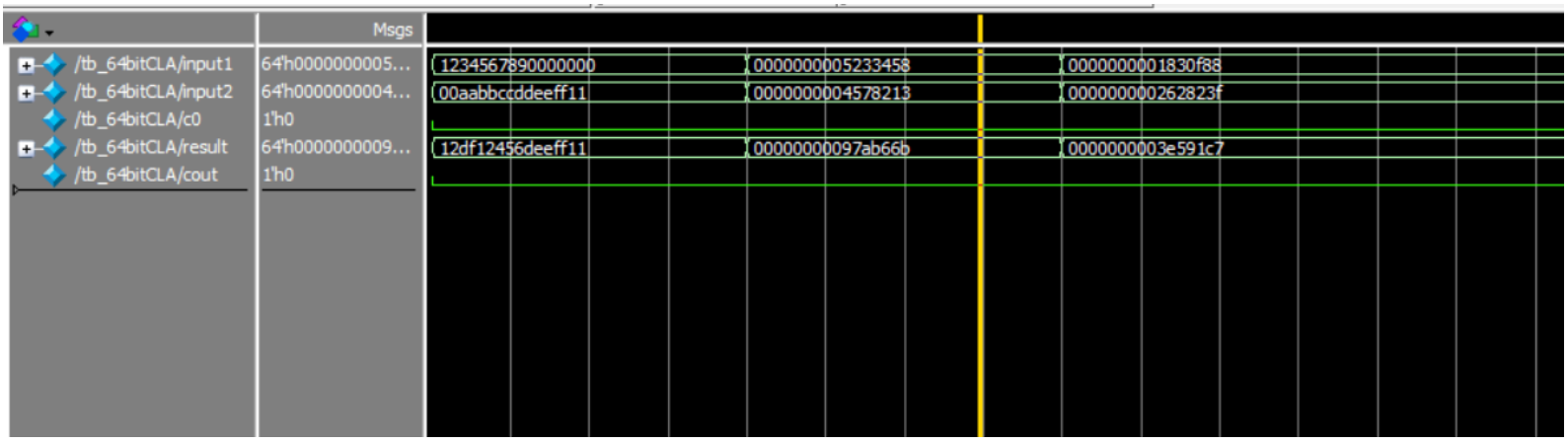
```
Ln#
 3  module CLA_64_bit(input1, input2, cin, result, cout, pout, gout);
 4          parameter N = 64;
 5          input [N-1:0] input1;
 6          input [N-1:0] input2;
 7          output [N-1:0] result;
 8          input cin;
 9          output cout;
10          output pout;
11          output gout;
12          wire [3:0] p;
13          wire [3:0] g;
14          wire [3:1] c;
15          CarryLookahead carrylookahead3 (
16                  .cin(cin),
17                  .p(p),
18                  .g(g),
19                  .c(c),
20                  .c4(cout),
21                  .pout(pout),
22                  .gout(gout)
23          );
24          CLA_16_bit f0(
25                  .input1(input1[(N/4)-1:0]),
26                  .input2(input2[(N/4)-1:0]),
27                  .result(result[(N/4)-1:0]),
28                  .cin(cin),
29                  .cout(),
30                  .pout(p[0]),
31                  .gout(g[0])
32          );
33          genvar i;
34          generate
35            for(i=1; i<4; i=i+1) begin
36                  CLA_16_bit f1(
37                          .input1(input1[(N/4)*(i+1)-1:(N/4)*i]),
38                          .input2(input2[(N/4)*(i+1)-1:(N/4)*i]),
39                          .result(result[(N/4)*(i+1)-1:(N/4)*i]),
40                          .cin(c[i]),
41                          .cout(),
42                          .pout(p[i]),
43                          .gout(g[i])
44                  );
45            end
46          endgenerate
```

از آنجایی که ساختار CLA_16_bit و CLA_64_bit شبیه به ساختار CLA_4_bit است با تفاوت های جزئی پس به توضیحات داده شده درباره مسئله اول این پروژه اکتفا میکنیم و به تکرار مکررات نمیپردازیم.

در ادامه به testbench نوشته شده میپردازیم و با آن کارکرد ماژول ساخته شده را بررسی میکنیم.

```
Ln#
 1  `timescale 1ns/1ps
 2  module tb_64bitCLA;
 3          //module CLA_64_bit(input1, input2, cin, result, cout, pout, gout);
 4          reg [63:0] input1;
 5          reg [63:0] input2;
 6          reg c0;
 7          wire [63:0] result;
 8          wire cout;
 9          CLA_64_bit uut(
10                  .input1(input1),
11                  .input2(input2),
12                  .cin(c0),
13                  .result(result),
14                  .cout(cout),
15                  .pout(),
16                  .gout()
17          );
18          initial begin
19                  input1 = 64'h1234567890000000;
20                  input2 = 64'h00aabbccddeeff11;
21                  c0 = 1'b0;
22                  #20;
23                  input1 = 64'h5233458;
24                  input2 = 64'h4578213;
25                  c0 = 1'b0;
26                  #20;
27                  input1 = 64'd0025366408;
28                  input2 = 64'd40010303;
29                  c0 = 1'b0;
30                  #20;
31          end
32
33
34  endmodule
35
```

در اینجا 3 بار ورودی های مختلف را با فاصله زمانی 20ns به یک CLA_64_bit تولید شده میدهیم و خروجی متناسب با آن ها را مشاهده میکنیم.



همانطور که مشاهده میشود با دادن ورودی های مختلف خروجی های درست را گرفته ایم.

Carry Select Adder 32 bit .2

پاسخ:

در این سوال هم مانند سوال قبل به یک تمام جمع کننده احتیاج داریم با تفاوت های کوچک. مثلا در سوال قبل تمام جمع کننده ما cout تولید نمیکرد ولی اینجا نیاز به تولید cout داریم. زیرا میخواهیم 4 جمع کننده ی 8 بیتی موازی بسازیم که با کمک آنها CarrySelectAdder خود را تکمیل کنیم.

به روند تولید FullAdderForCSA.v و N_bit_RippleCarryAdder.v نمیپردازیم چون به طور مفصل در پروژه قبل به آنها پرداخته شده است و فقط کد مربوط به آنها را در ذیل میاوریم.

```
Ln#
 1  `timescale 1ns/1ps
 2  module FullAdderForCSA (input1, input2, result, cin, cout);
 3      input input1;
 4      input input2;
 5      output reg result;
 6      input cin;
 7      output reg cout;
 8      always @(*) begin
 9                  cout <= (input1 & input2) | ((input1 ^ input2) & cin);
10                  result <= input1^input2^cin;
11      end
12  endmodule
```

```
Ln#
 1  `timescale 1ns/1ps
 2  module N_bit_RippleCarryAdder(input1, input2, result, cout, cin);
 3   parameter N = 8;
 4   input [N-1:0] input1;
 5   input [N-1:0] input2;
 6   input cin;
 7   output [N-1:0] result;
 8   output cout;
 9   wire [N:0] carry;
10   assign cout = carry[N];
11   assign carry[0] = cin;
12   // instantiating N 1-bit full adders in Verilog
13   genvar i;
14   generate
15      for(i = 0; i < N; i = i + 1) begin
16      FullAdderForCSA f (
17           .input1(input1[i]),
18               .input2(input2[i]),
19               .result(result[i]),
20               .cin(carry[i]),
21               .cout(carry[i+1])
22           );
23           end
24   endgenerate
25  endmodule
```

در ادامه ما به اصل ماجرا که همان CarrySelectAdder_32_bit.v است میپردازیم.

منتهی قبل از آن به توضیح Mux_2_In_1.v که همان مالتی پلکسر با دو ورودی و یک selector است میپردازیم.

```
Ln#
1   `timescale 1ns / 1ps
2
3   module Mux_2_In_1 #(parameter BITS = 8) (Input0, cin0, Input1, cin1, Sel, Data_out, cout);
4       // constant declaration
5       parameter S0 = 1'b0;
6       parameter S1 = 1'b1;
7       //inputs
8       input [BITS-1:0] Input0;
9       input [BITS-1:0] Input1;
10      input cin0;
11      input cin1;
12      input Sel;
13      //outputs
14      output reg [BITS-1:0] Data_out;
15      output reg cout;
16      always @ (Sel or Input0 or Input1)
17      begin
18          case(Sel)
19              S0: begin
20                  Data_out <= Input0;
21                  cout <= cin0;
22              end
23              S1: begin
24                  Data_out <= Input1;
25                  cout <= cin1;
26              end
27          endcase
28      end
29  endmodule
```

این مالتی پلکسر 2 ورودی با تعداد بیت های BITS تایی که یک پارامتر قابل تغییر است میگیرد و carry های متناسب با هر input را هم از ورودی میگیرند. این مالتی پلکسر هیچ کاری با carry ها ندارد و فقط آنها را از خود عبور میدهند.

درون always block ورودی select چک میشود اگر برابر با s0 = 1’b0 بود input0 و cin0 را به خروجی میدهد و اگر متناسب با s1 = 1’b1 بود input1 و cin1 را به خروجی میدهند.

میتوانیم هنگام instantiate کردن مالتی پلکسر پارامتر BITS را تغییر بدهیم و مالتی پلکسر با ورودی های بزرگتر (مثلا 16 بیتی) بسازیم.

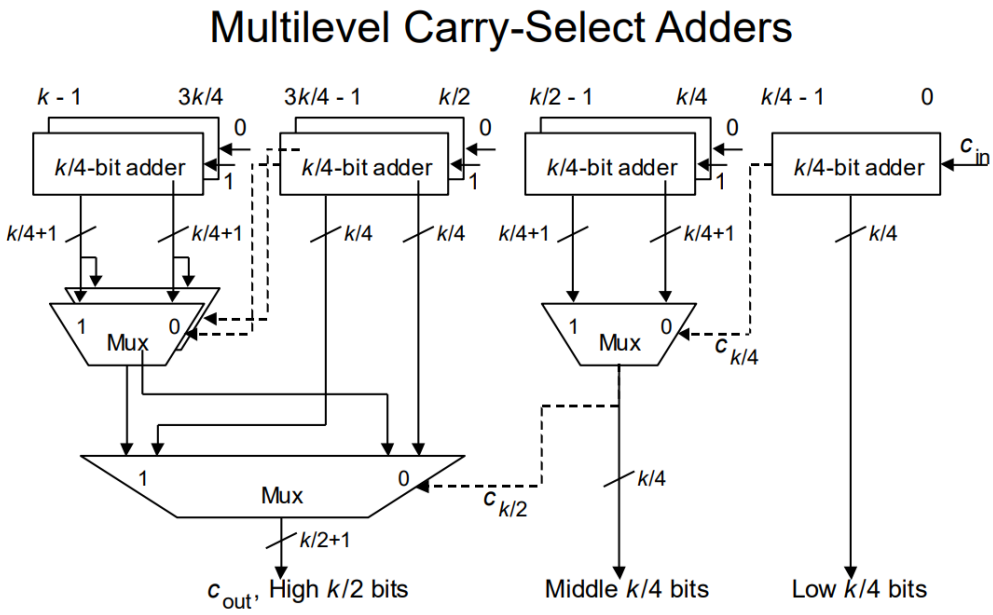
در ادامه به بررسی CarrySelectAdder_32_bit.v میپردازیم.

CarrySelectAdder_32_bit.v :

```
1   `timescale 1ns/1ps
2   module CarrySelectAdder_32_bit (input1, input2, cin, result, cout);
3           input [31:0] input1;
4           input [31:0] input2;
5           input cin;
6           output [31:0] result;
7           output cout;
8           //Inner Connections
9           wire [7:0] bus [0:2][0:1];
10          wire [3:0] muxSelect;
11          wire rippleAddersCout[0:2][0:1];
12          wire [7:0] littleMuxResults [0:2];
13          wire muxCouts [0:3];
14          wire [15:0] bigMUXResult; //change output to wire and delete from ()
15          //digit as RCAs Cins
16          reg digit[0:1];
17          initial begin
18                  digit[1] = 1'b1;
19                  digit[0] = 1'b0;
20          end
21          //instantiate necessary modules
22          N_bit_RippleCarryAdder rca0 (
23                  .input1(input1[7:0]),
24                  .input2(input2[7:0]),
25                  .result(result[7:0]),
26                  .cin(cin),
27                  .cout(muxSelect[0])
28          );
29          genvar i,j;
30          generate
31                  for(j = 0; j < 3; j = j + 1) begin
32                          for(i = 0; i < 2; i = i + 1) begin
33                                  N_bit_RippleCarryAdder rcas (
34                                          .input1(input1[(j+2)*8-1:(j+1)*8]),
35                                          .input2(input2[(j+2)*8-1:(j+1)*8]),
36                                          .result(bus[j][i]),
37                                          .cin(digit[i]),
38                                          .cout(rippleAddersCout[j][i])
39                                  );
40                          end
41                  end
42          endgenerate
```

```
43          genvar k;
44          generate
45                  for(k = 0; k < 3; k = k + 1) begin
46                          Mux_2_In_1 littleMUXs (
47                                  .Input0(bus[k][0]),
48                                  .Input1(bus[k][1]),
49                                  .cin0(rippleAddersCout[k][0]),
50                                  .cin1(rippleAddersCout[k][1]),
51                                  .Sel(muxSelect[k]),
52                                  .Data_out(littleMuxResults[k]),
53                                  .cout(muxCouts[k])
54                          );
55                  end
56          endgenerate
57          assign muxSelect[1] = rippleAddersCout[1][0];
58          assign muxSelect[2] = rippleAddersCout[1][1];
59          assign muxSelect[3] = muxCouts[0];
60          Mux_2_In_1 #(16) bigMUX (
61                  .Input0({littleMuxResults[1],bus[1][0]}),
62                  .Input1({littleMuxResults[2],bus[1][1]}),
63                  .cin0(muxCouts[1]),
64                  .cin1(muxCouts[2]),
65                  .Sel(muxSelect[3]),
66                  .Data_out(bigMUXResult),
67                  .cout(cout)
68          );
69          assign result[31:8] = {bigMUXResult, littleMuxResults[0]};
70  endmodule
```

در اینجا ما ساختار زیر را که در جزوه آمده است پیاده کردیم:



Two-level carry-select adder built of *k*/4-bit adders.

دو ورودی 32 بیتی input1, input2 از ورودی گرفته میشوند و به صورت بسته های 8 تایی به 4 سری RCA (شامل 7 RCA) داده میشوند و حاصل جمع آنها درون bus ذخیره میشوند. البته جمع RCA اول به صورت مستقیم درون 7 بیت کم ارزش نتیجه نهایی ذخیره

میشوند. مالتی پلکسر ها و RCA ها همانند شکل بالا instantiate شده و با اتصالات متناسبی که در شکل هم نشان داده شده اند و با اسم مناسب درون کد تعریف شده اند و نیازی به تعریف بیشتر نیست فقط باید اشاره کنم که bigMUX همان مالتی پلکسر 16 بیتی پایین است و littleMUX ها هم مالتی پلکسر های 8 بیتی هستند.

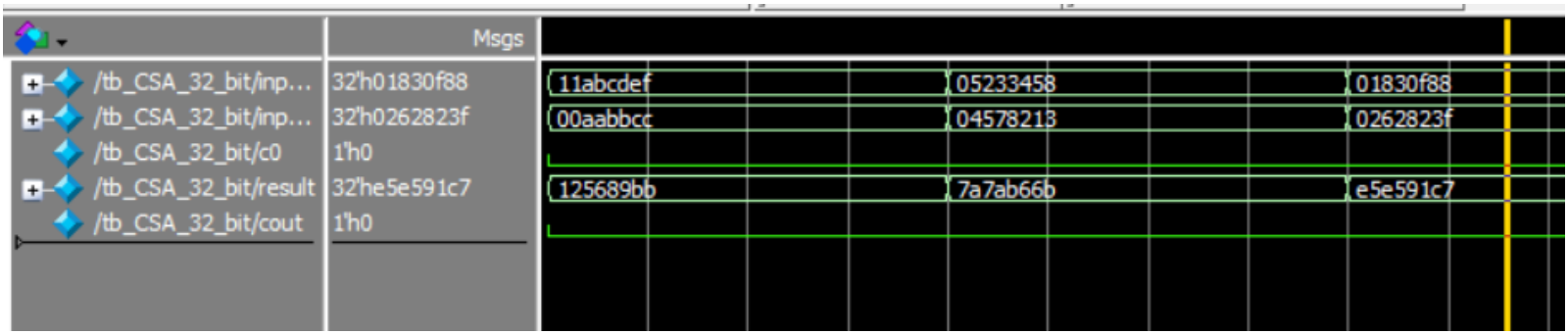
حال به testbench طراحی شده برای این مدار میپردازیم و با دادن ورودی های مختلف نتایج را بررسی میکنیم:

tb_CSA_32_bit.v :

```
Ln#
 1  `timescale 1ns/1ps
 2  module tb_CSA_32_bit;
 3          //module CarrySelectAdder_32_bit (input1, input2, cin, result, cout);
 4          reg [31:0] input1;
 5          reg [31:0] input2;
 6          reg c0;
 7          wire [31:0] result;
 8          wire cout;
 9          CarrySelectAdder_32_bit uut(
10                  .input1(input1),
11                  .input2(input2),
12                  .cin(c0),
13                  .result(result),
14                  .cout(cout)
15          );
16          initial begin
17                  input1 = 32'h11abcdef;
18                  input2 = 32'h00aabbcc;
19                  c0 = 1'b0;
20                  #20;
21                  input1 = 32'h5233458;
22                  input2 = 32'h4578213;
23                  c0 = 1'b0;
24                  #20;
25                  input1 = 32'd0025366408;
26                  input2 = 32'd40010303;
27                  c
28                  #
29          end
30  endmodule
31
32
```

/tb_CSA_32_bit/input2
0262823f

در اینجا چند ورودی مختلف به CarrySelectAdder_32_bit داده شده و انتظار داریم خروجی مناسب بدهد. در ادامه خروجی ها را بررسی میکنیم:



میبینیم که با ورودی های مختلفی که با فاصله زمانی 20ns به قطعه مد نظر دادیم خروجی های صحیح و متناسبی دریافت کردیم و این ماژول درست کار میکند.