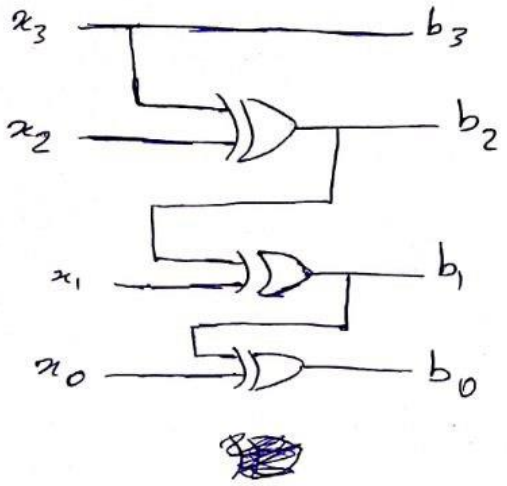
2 روز گریس

محمد حسین جوادی

۸۱۰۱۰۳۵۶۷

Converter : (gray code to 2's complement)

number: $x_3x_2x_1x_0$ x_3 msb*



تأخیرها:

b_3 : 0ns

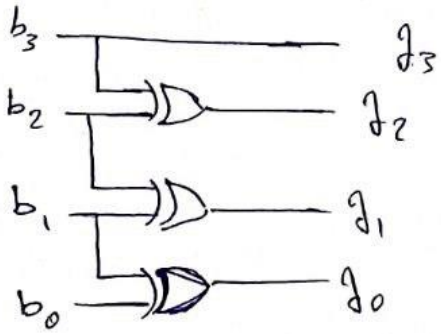
b_2 : 0 + 3ns = 3ns

b_1 : 0 + 3 + 3 = 6ns

b_0 : 0 + 3 + 3 + 3 = 9ns

reverse converter(2's complement to gray code)

number: $b_3 b_2 b_1 b_0$

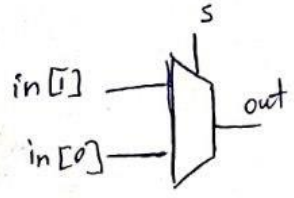


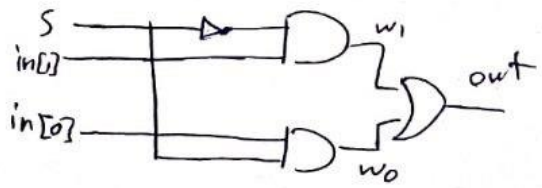
تأخیرها:

g_3 = 0ns

$g_2 = g_1 = g_0$ = 3ns

2 to 1 mux: $out = \overline{S} \cdot in[1] + S \cdot in[0]$





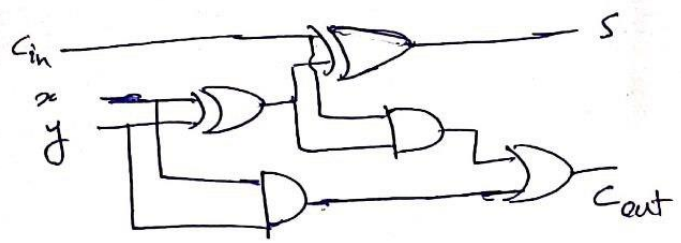
تأخیرها:

w_1 : 3ns

w_0 : 2ns

out : 5ns

Full adder:

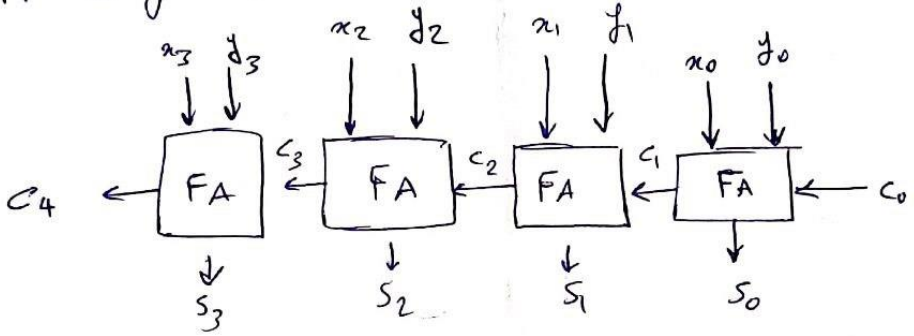


Critical path: 3+2+2=7ns

C_{out}: 7ns (max)

S: 6ns (max)

4bit ripple carry adder:



[illegible]

C_1: 7ns

C_2: 11ns

C_3: 15ns

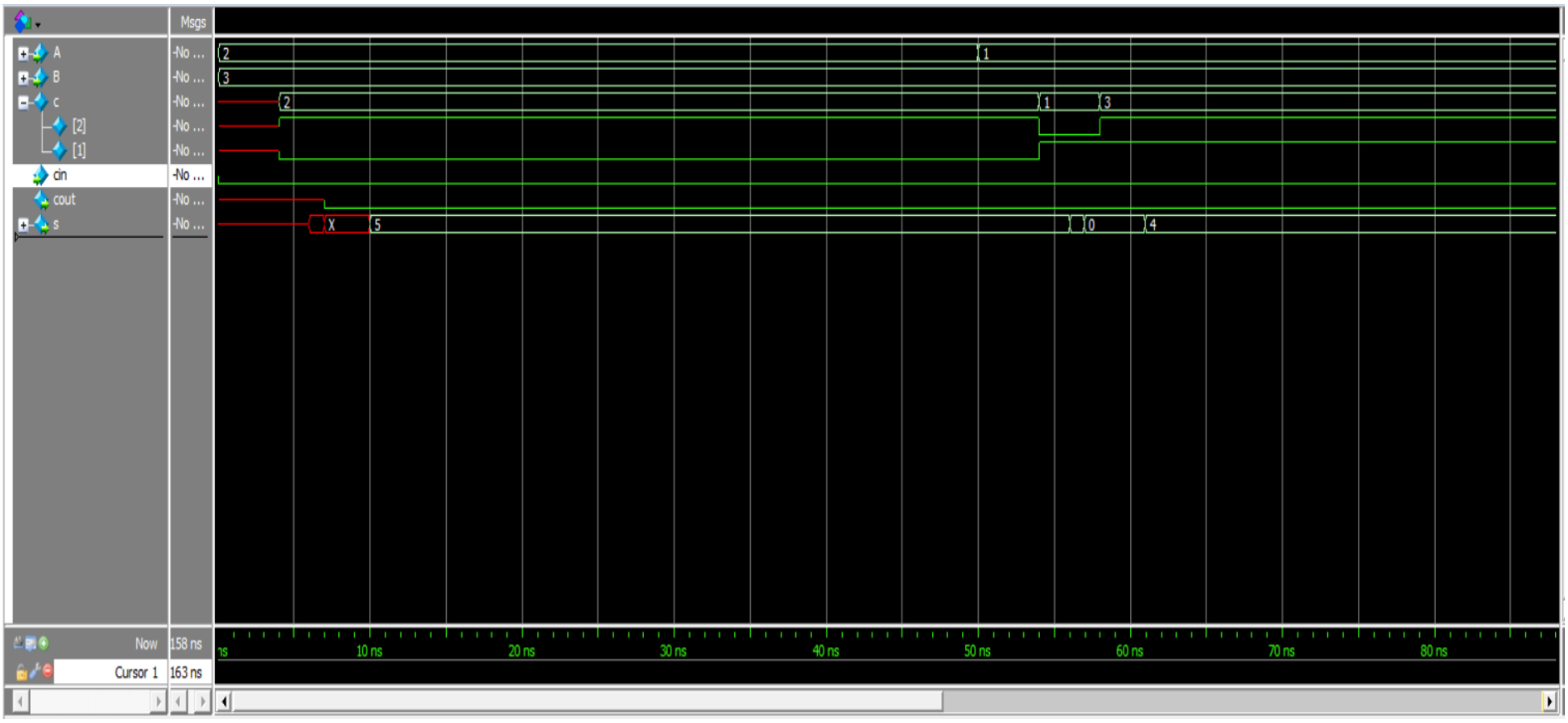
C_4: 19ns

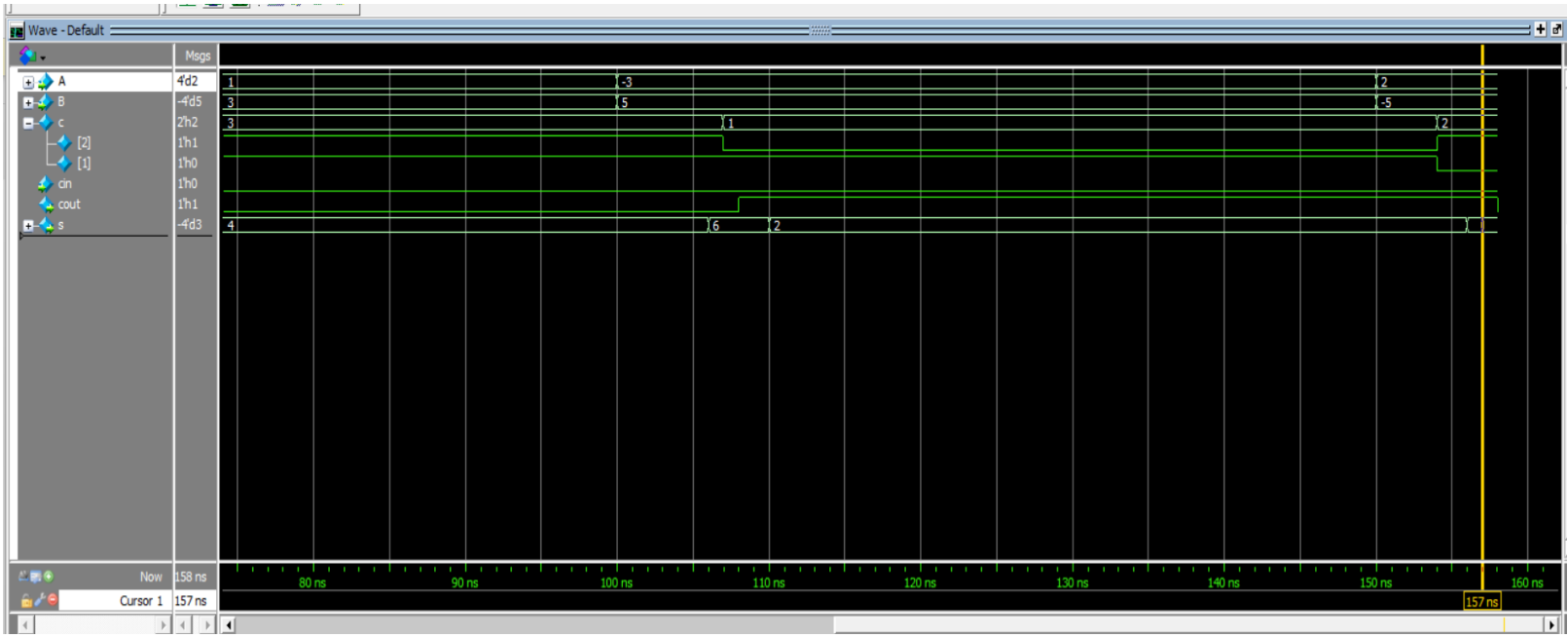
S_0: 6ns

S_1: ~~10~~ 10ns

S_2: 14ns

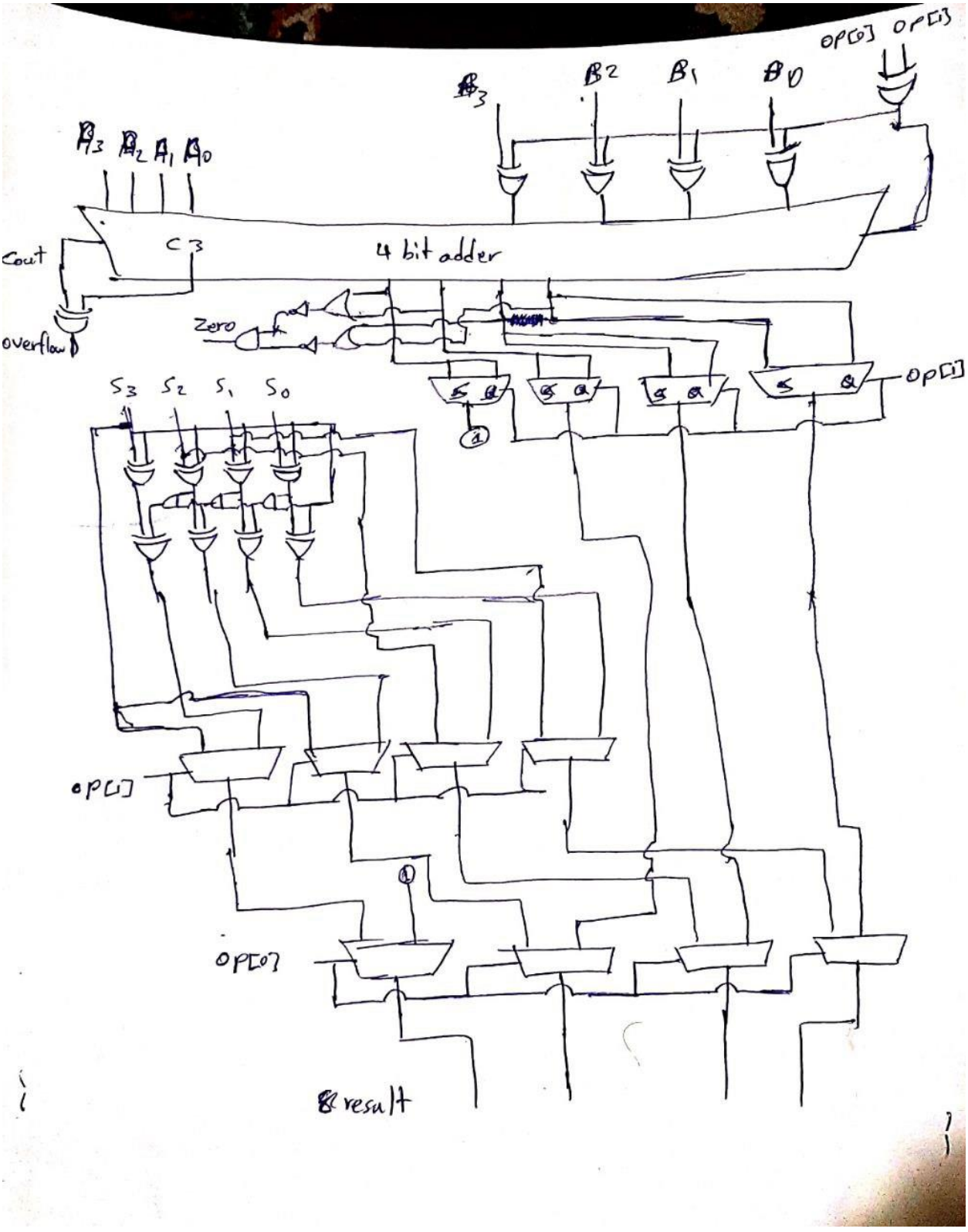
S_3: 18ns


Msgs
A
B
c
[2]
[1]
cin
cout
s
-No ...
2
3
1
X
5
0
4
Now
158 ns
Cursor 1
163 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns



جمع ۲و۳ در ۱۰ نانوثانیه برابر ۵ شده است در صورتی که حداکثر تاخیر ۱۸ نانو ثانیه است. علت ان است که قبل از امدن هرکری مرحله قبلی ai and bi بر روی گیت ار رفته اند که اگر یک باشند خروجی یک میشود و این در صورتی است که ما منتظر کری مرحله قبل هستیم.

اما همیشه اینگونه نیست به عنوان مثال جمع ۱و۳ پس از ۱۱نانوثانیه خروجی ۴ را داده است و قبل از ان ۲ بار خروجی تغییر کرده است.


op[0] op[1]
B3 B2 B1 B0
A3 A2 A1 A0
Cout
C3
4 bit adder
overflow
zero
S3 S2 S1 S0
op[1]
op[1]
op[0]
result

Add:6 + 18 + 5 +5 = 34ns

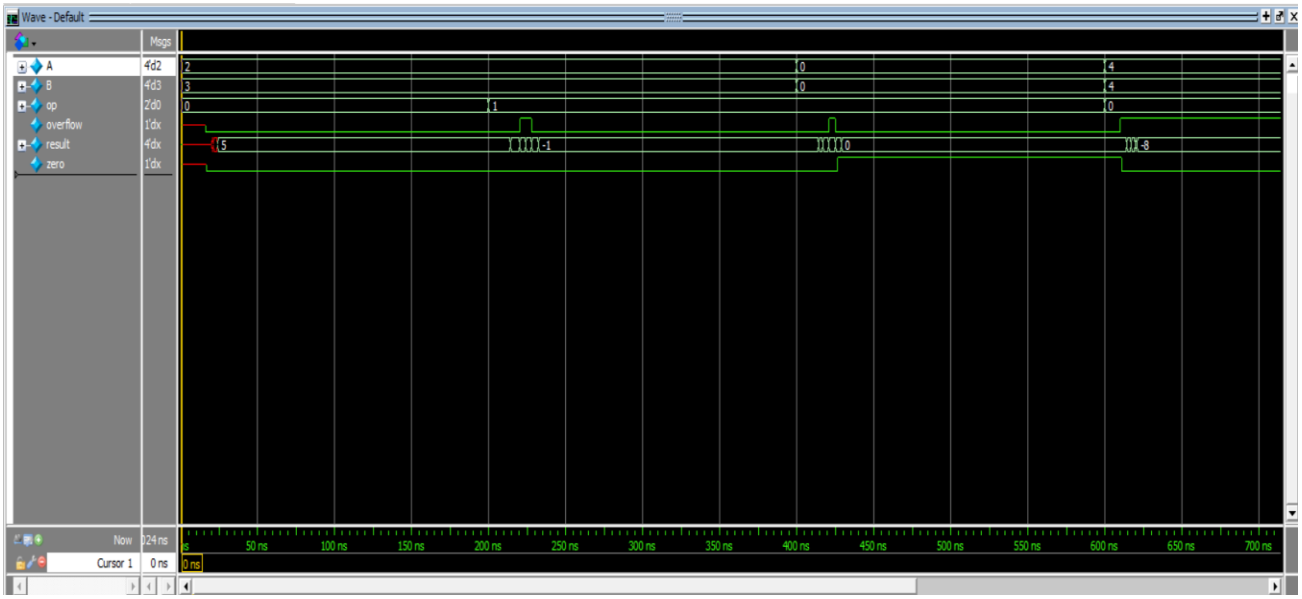
Sub: 6 +18 +5 + 5 = 34ns

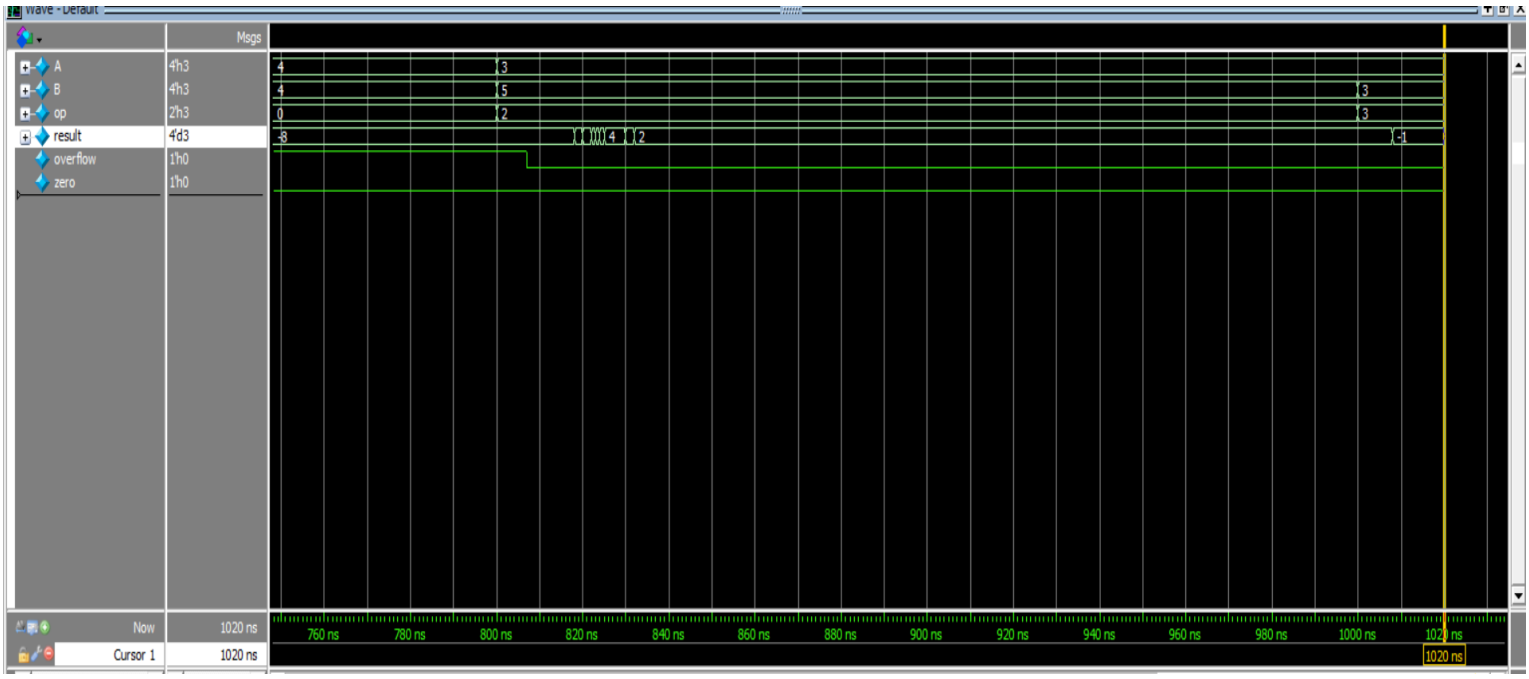
Overflow:6 + 19+ 3= 29ns

Zero:6+15+3 = 24ns

Abs:6+18+3+6+3+5+5=43ns

Shift:6+18+5+5=34ns

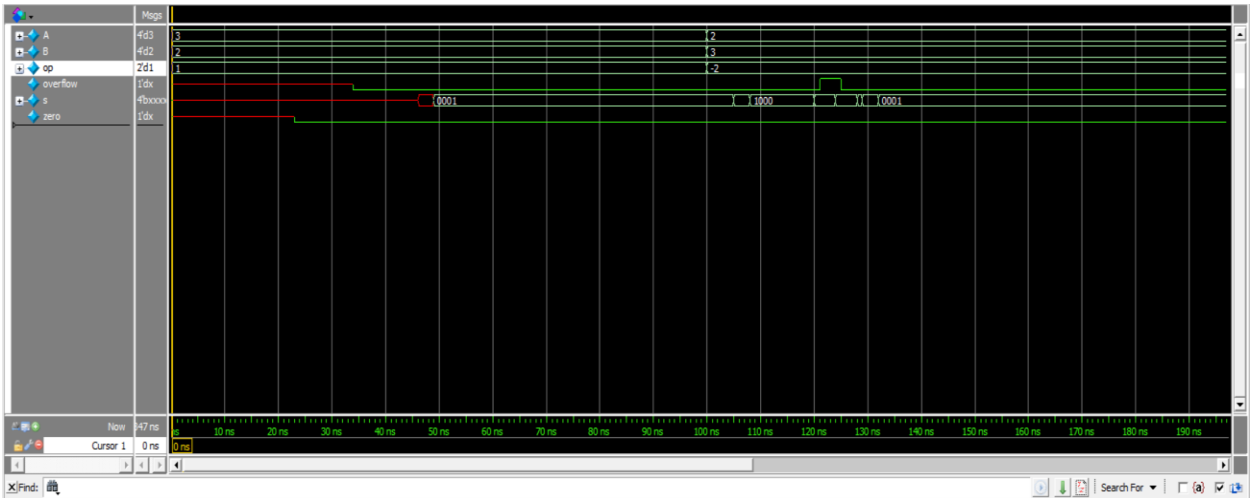


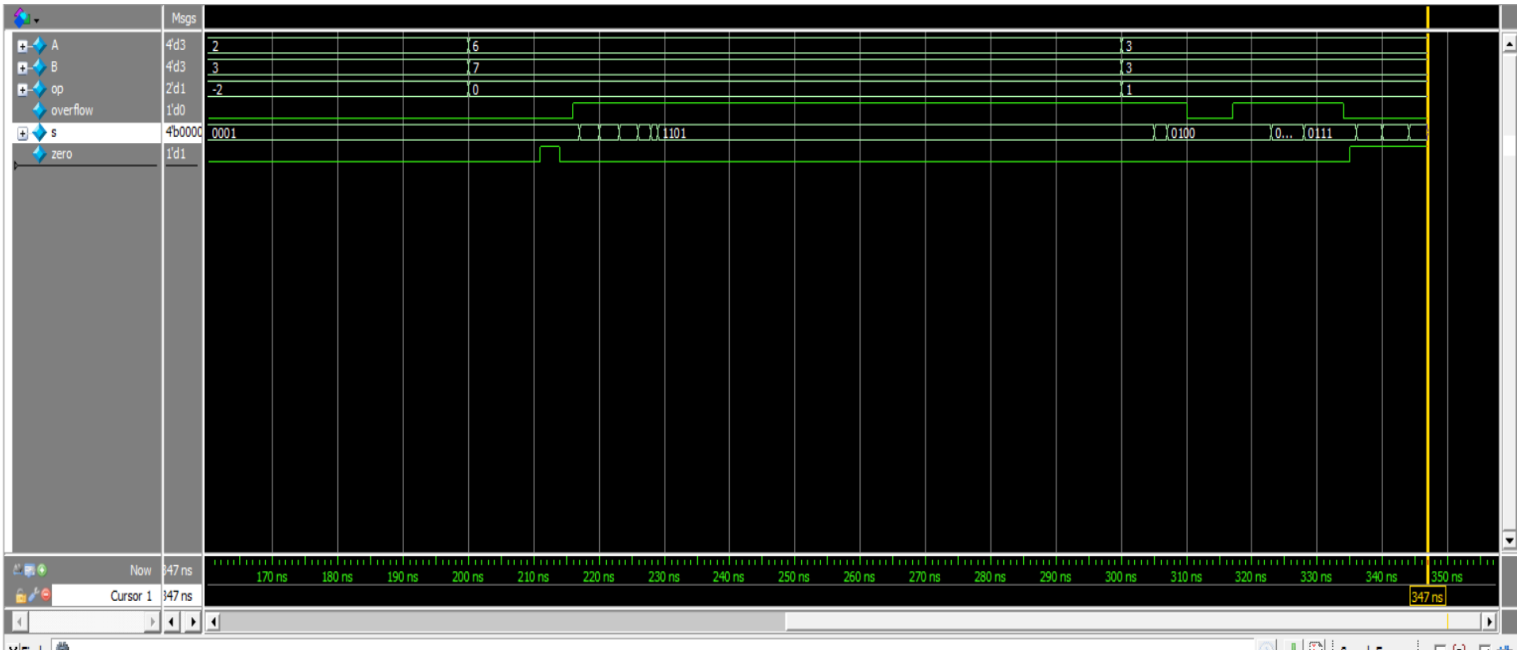


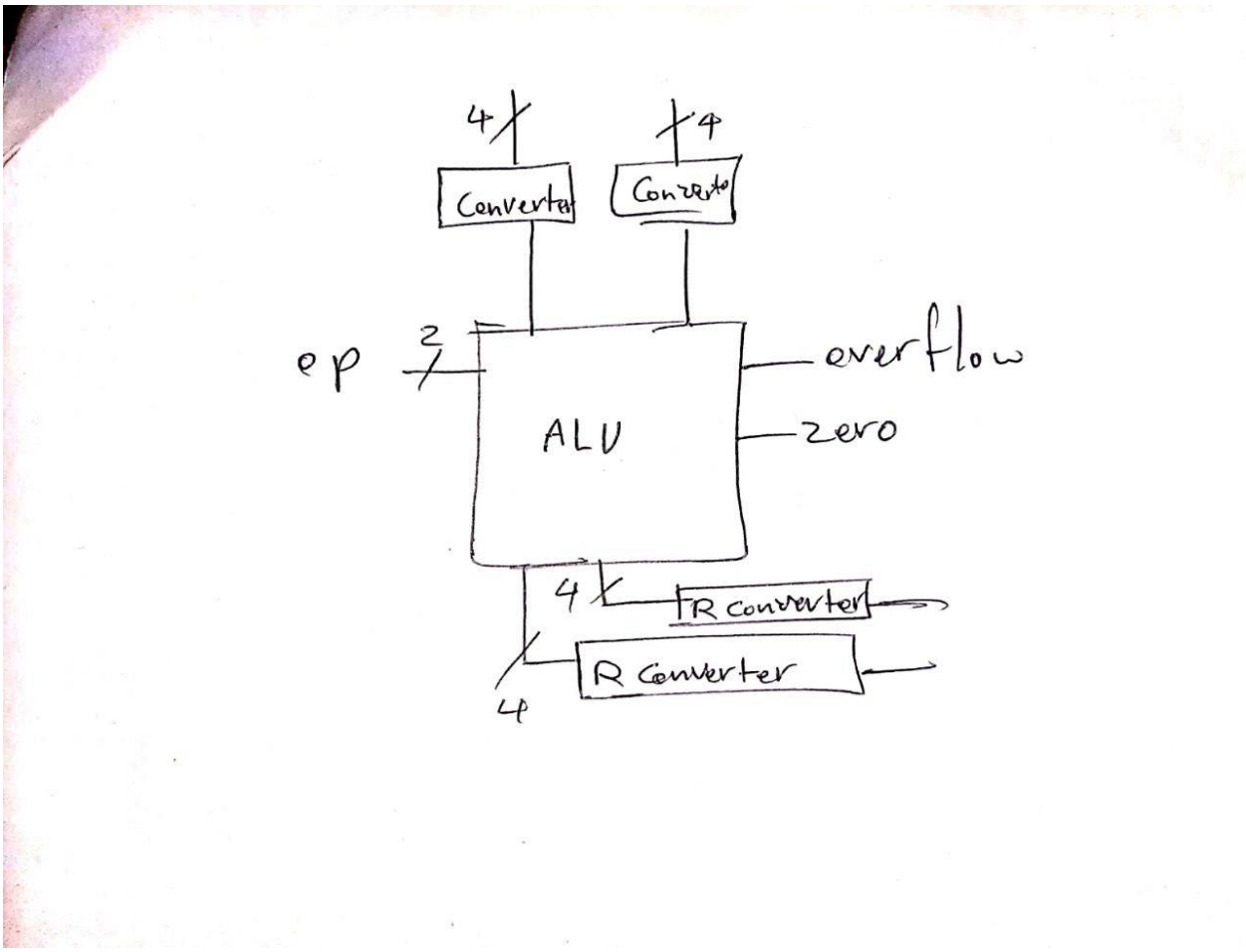
در جمع ۳ و ۲ نتیجه پس از ۲۴ نانوثانیه امده است چرا که ار هایی که برای کری هستند زودتر یکی از مقادیرشان ۱ شده است و زودتر کری منتقل شده به بعدی 0011 کری دوم در ۴ نانوثانیه منتقل شده چرا که بیت دوم یک است و ار ها را یک کرده است.

اما همیشه برقرار نیست این قضیه به عنوان مثال در تفریق ۲ و ۳ جواب پس از ۳۲ نانوثانیه امده است.

در جمع ۴ و ۴ سیگنال اورفلو ثانیه ۱۰ اماده است و حاصل -۸ درزمان ۳۲نانوثانیه امده است.




Msgs
A
B
op
overflow
s
zero
4'd3
4'd3
2'd1
1'd0
4'b0000
1'd1
2
6
3
3
7
3
-2
0
1
0001
1101
0100
0...
0111
Now
347 ns
Cursor 1
347 ns
170 ns
180 ns
190 ns
200 ns
210 ns
220 ns
230 ns
240 ns
250 ns
260 ns
270 ns
280 ns
290 ns
300 ns
310 ns
320 ns
330 ns
340 ns
350 ns
347 ns



Add:6 + 18 + 5 +5 + 12 = 46ns

Sub: 6 +18 +5 + 5 +12= 46ns

Overflow:6 + 19+ 3 + 9 = 37ns

Zero:6+18 +5+ 9 = 38ns

Abs:6+18+3+6+3+5+5 + 12=55ns

Shift:6+18+5+5 + 12=46ns

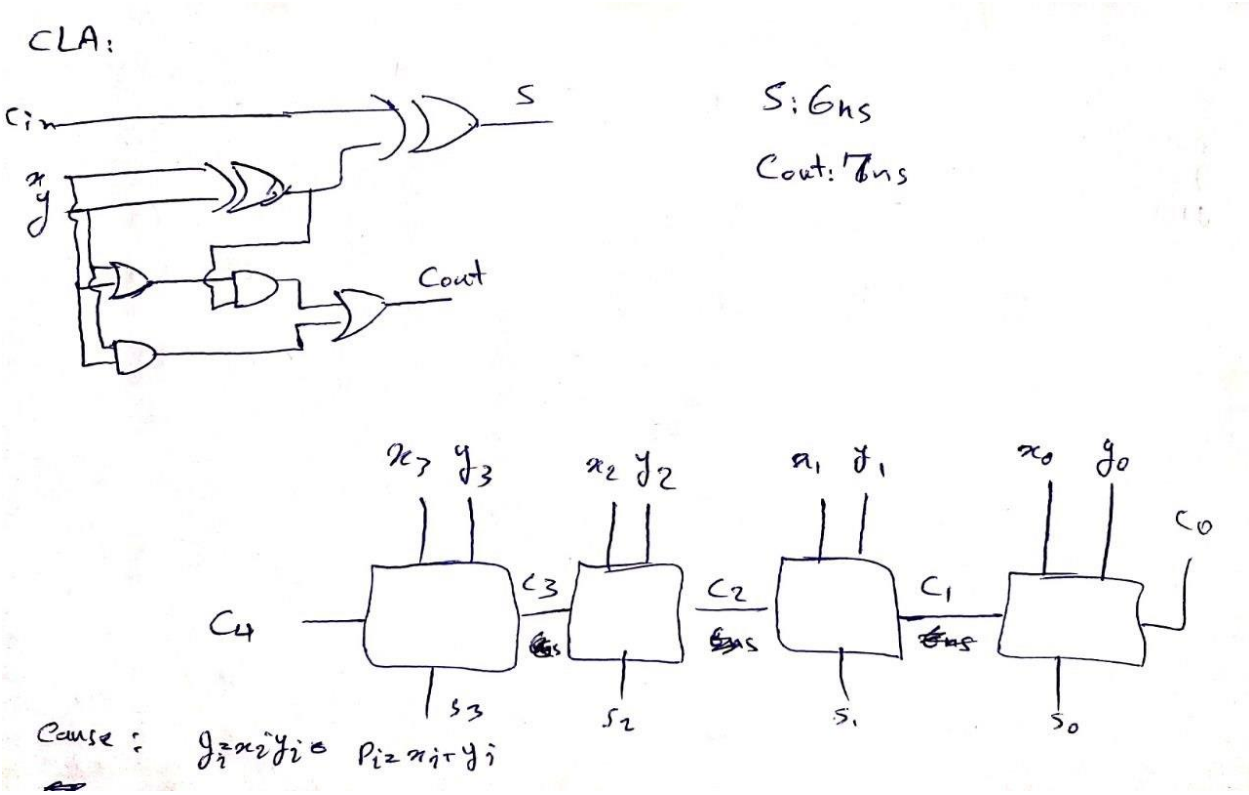
خروجی اول(جمع ۲ و ۳) برابر ۵ شده و ۴۶ نانوثانیه طول کشیده یعنیحداکثر تاخیر

دوم قدر مطلق تفریق ۳ از ۲ که ۳۴ نانوثانیه طول کشیده.

سوم جمع ۴ و ۵ که اورفلو داده پس از ۱۶ نانوثانیه

چهارم تفریق ۲ و ۲ که سیگنال صفر را ۳۳ نانوثانیه بعد فعال کرده

پنجم شیفت جمع ۲ و ۲ که پس از ۲۷ نانوثانیه ایجاد شده.



$$c_4 = g_3 + P_3 c_3 = g_3 + P_3(g_2 + P_2 c_2) = g_3 + P_3(g_2 + P_2(g_1 + P_1 c_1)) =$$

$$g_3 + P_3(g_2 + P_2(g_1 + P_1(g_0 + P_0 c_0))) = g_3 + P_3(g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 c_0)$$

$$= g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0 + P_3 P_2 P_1 P_0 c_0 \;*$$

$c_3 = g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 c_0$ $\quad$ $c_2 = g_1 + P_1 g_0 + P_1 P_0 c_0$ $\quad$ $c_1 = g_0 + P_0 c_0$

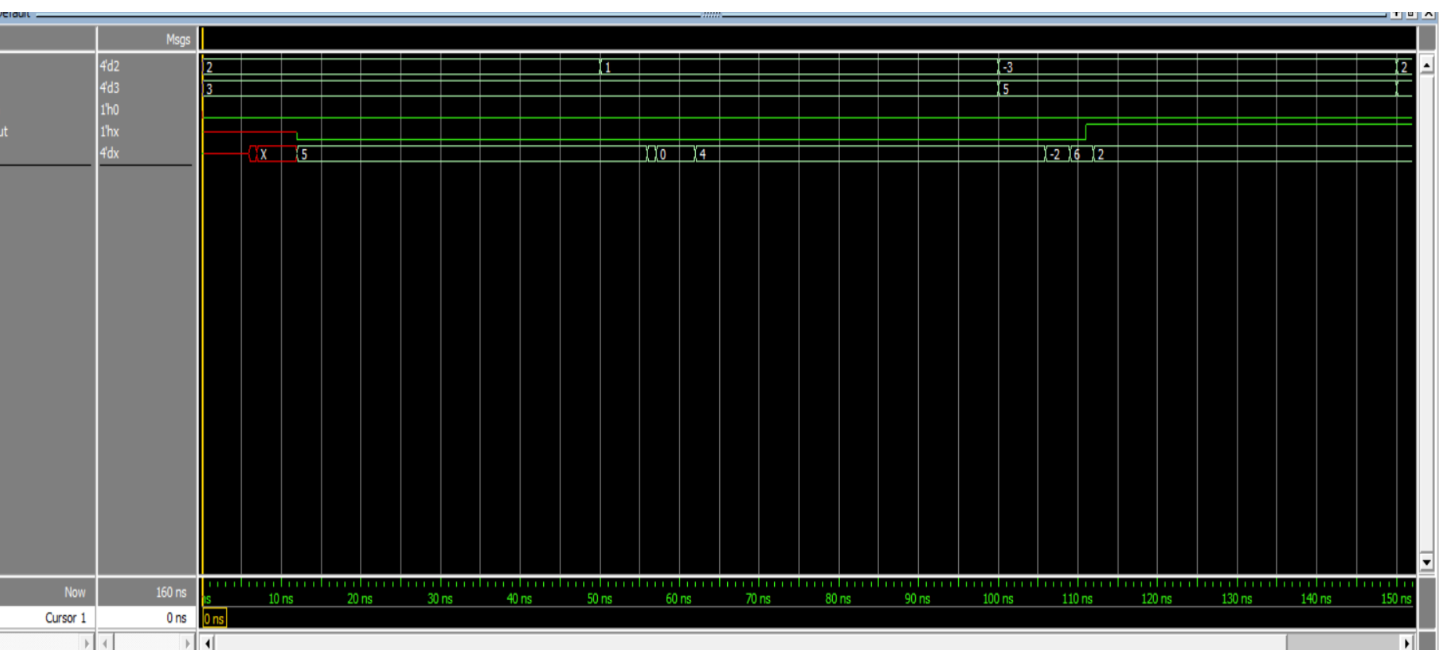
* تأخیر کمتر برای c_i ها داریم ولی مدار پیچیده تر شد.

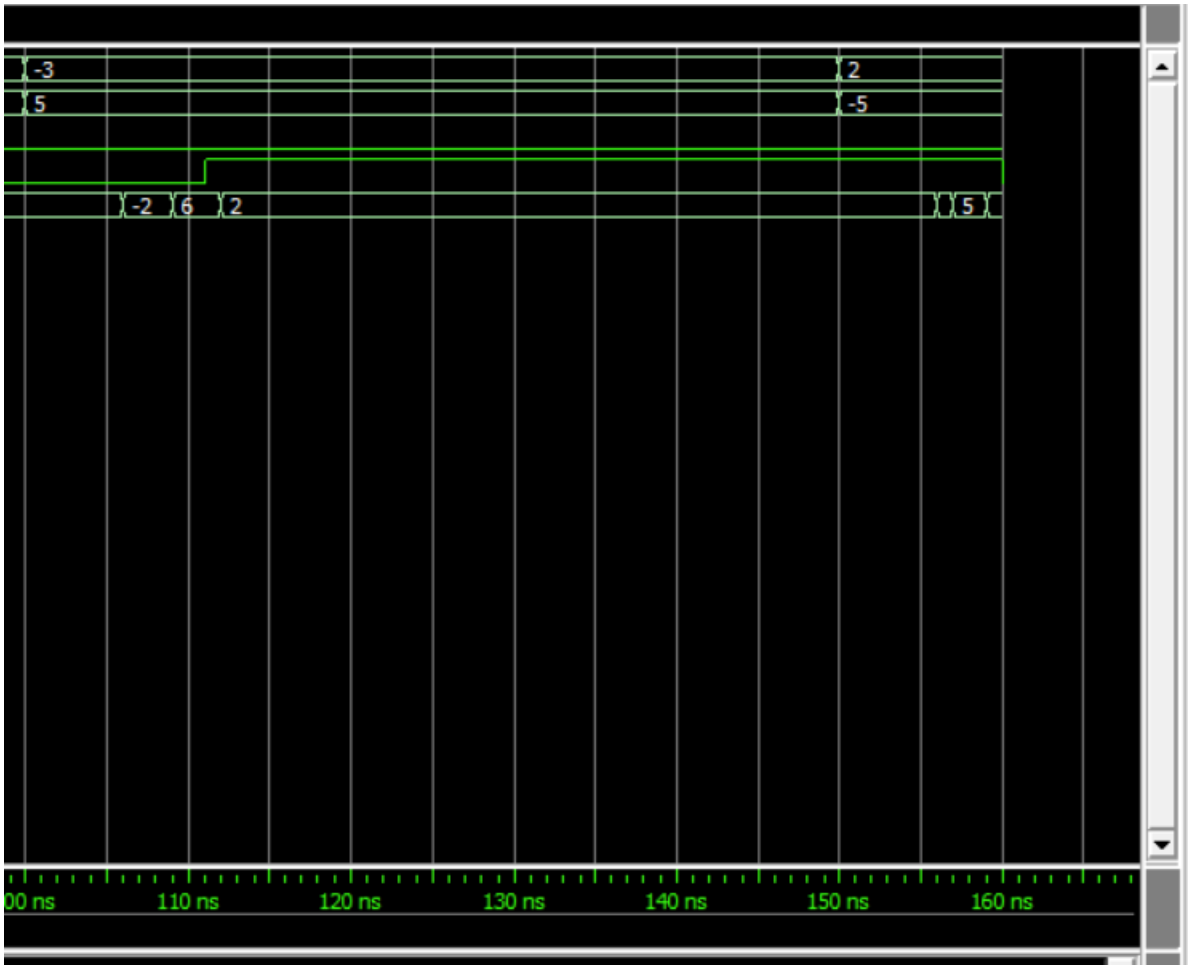
S_0: 6ns $\quad$ ~~...~~ S_3 $\quad$ c_1 = 7ns ~~...~~ $\quad$ c_2: 9ns $\quad$ c_3: 11ns

S_1: 10ns $\quad$ c_4: 13ns

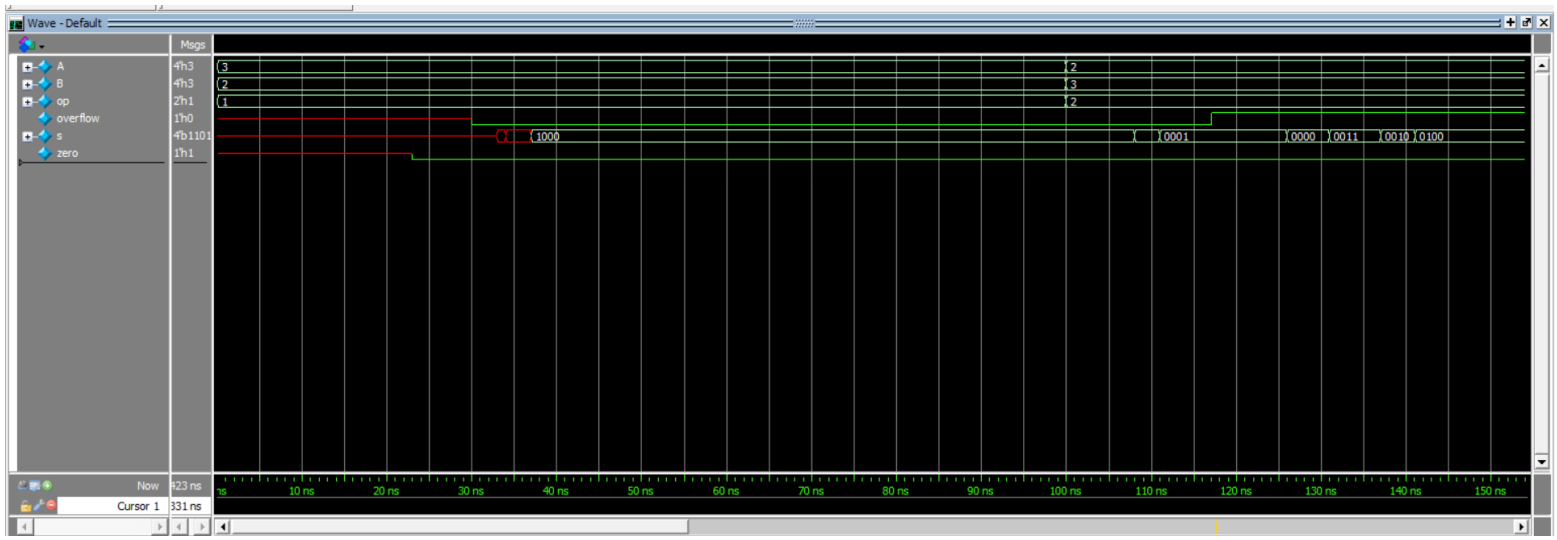
S_2: 12ns

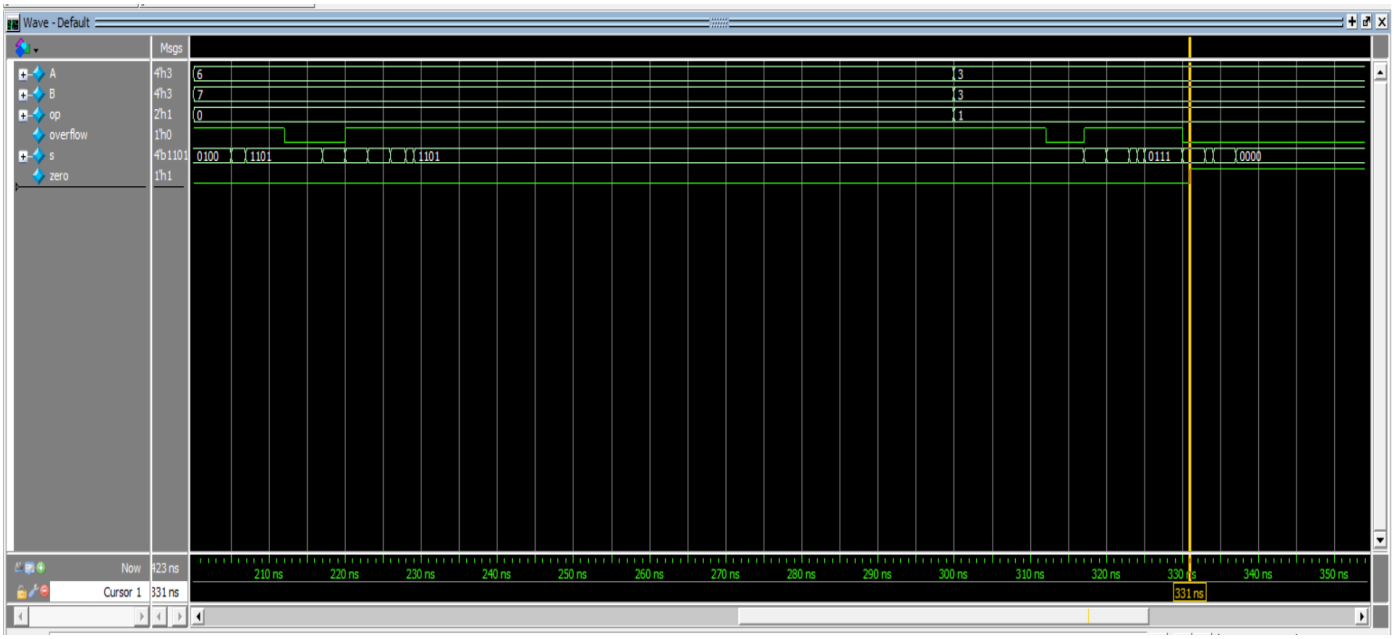
S_3: 14ns

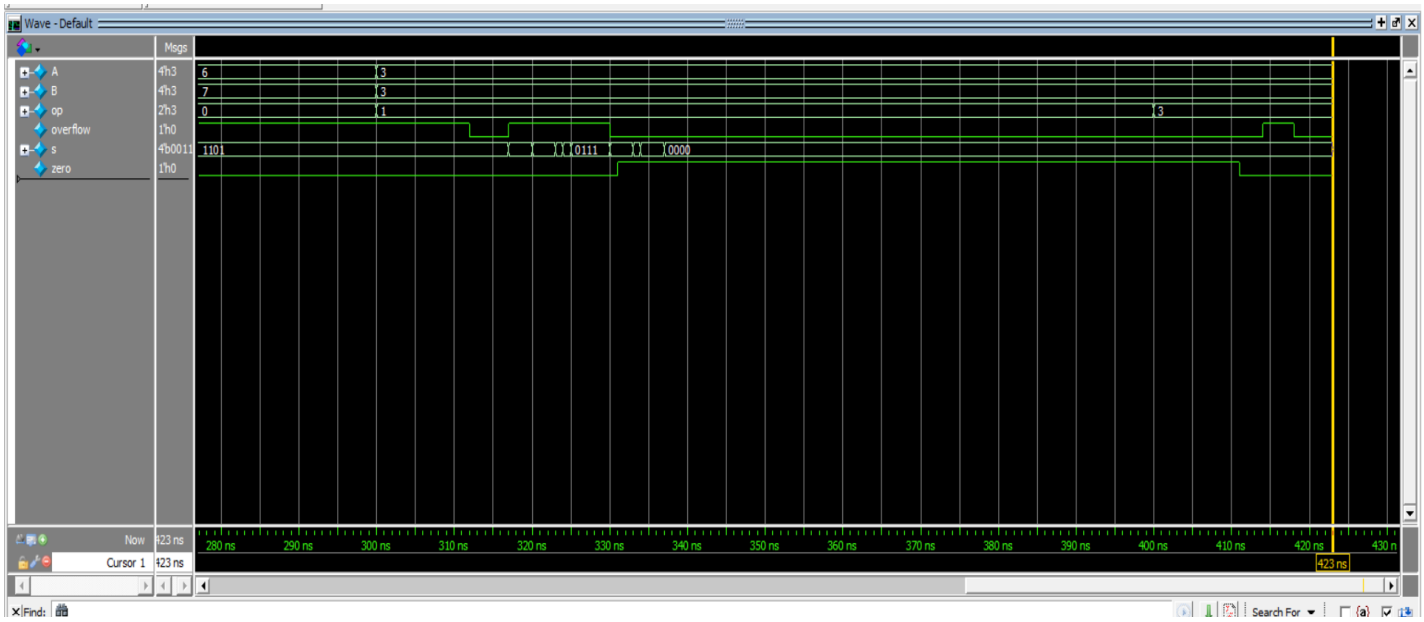




خروجی اول و دوم پس از ۱۲ نانوثانیه به نمایش درامدند اما خروجی سوم پس از ۱۳ نانوثانیه و خروجی اخر پس از ۱۰ نانو ثانیه


Wave - Default
Msgs
A
B
op
overflow
s
zero
4'h3
4'h3
2'h1
1'h0
4'b1101
1'h1
3
2
1
2
3
2
1000
0001
0000
0011
0010
0100
Now
423 ns
Cursor 1
331 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
130 ns
140 ns
150 ns


Wave - Default
Msgs
A
B
op
overflow
s
zero
4'h3
4'h3
2'h1
1'h0
4'b1101
1'h1
6
7
0
3
3
1
0100
1101
1101
0111
0000
Now
423 ns
Cursor 1
331 ns
210 ns
220 ns
230 ns
240 ns
250 ns
260 ns
270 ns
280 ns
290 ns
300 ns
310 ns
320 ns
330 ns
340 ns
350 ns
331 ns



Add:6 + 14 + 5 +5 + 12 = 42ns

Sub: 6 +14 +5 + 5 +12= 42ns

Overflow:6 + 13+ 3 + 9 = 32ns

Zero:6+14+3+5 + 9 = 33ns

Abs:6+14+3+6+3+5+5 + 12=51ns

Shift:6+14+5+5 + 12=42ns

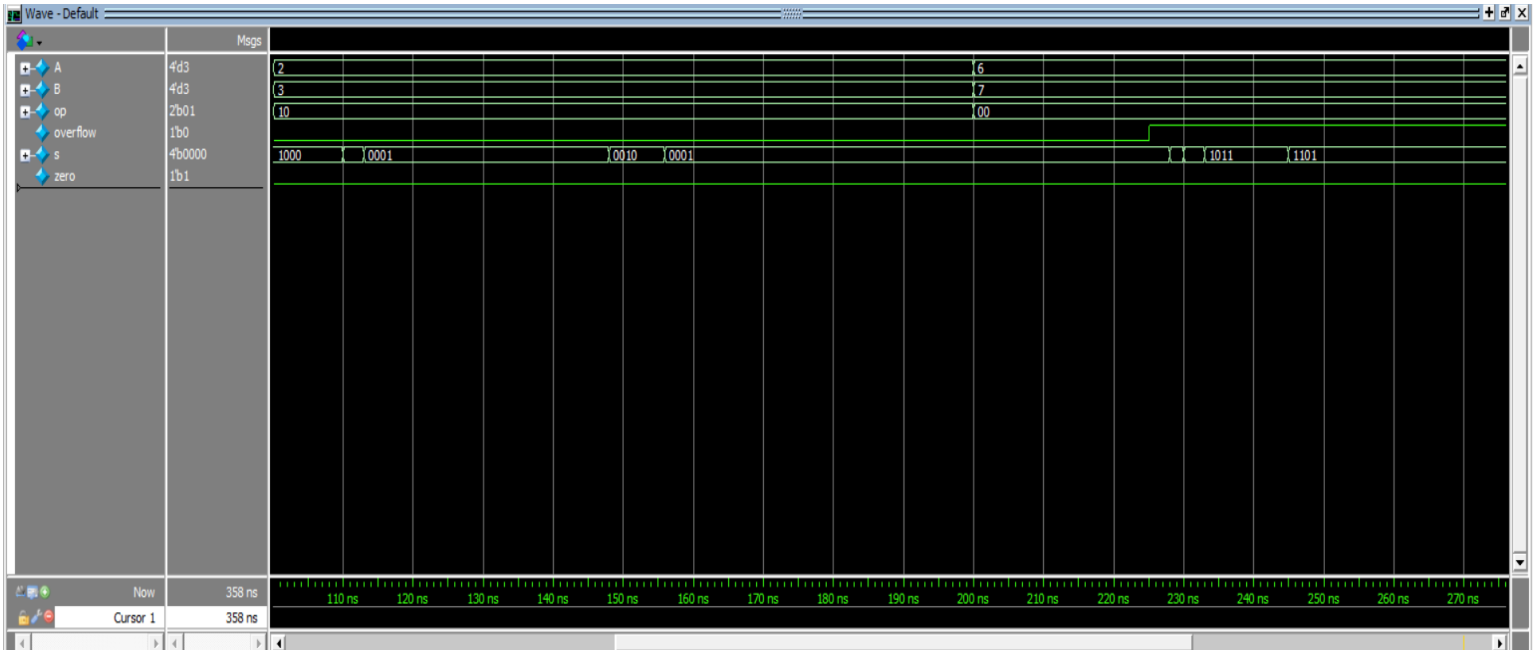
خروجی اول در ۳۷ نانو ثانیه امده که نسبت به RCA ۹ نانوثانیه کمتر است به این دلیل که هم خروجی جمع سریع تر است و هم کری.

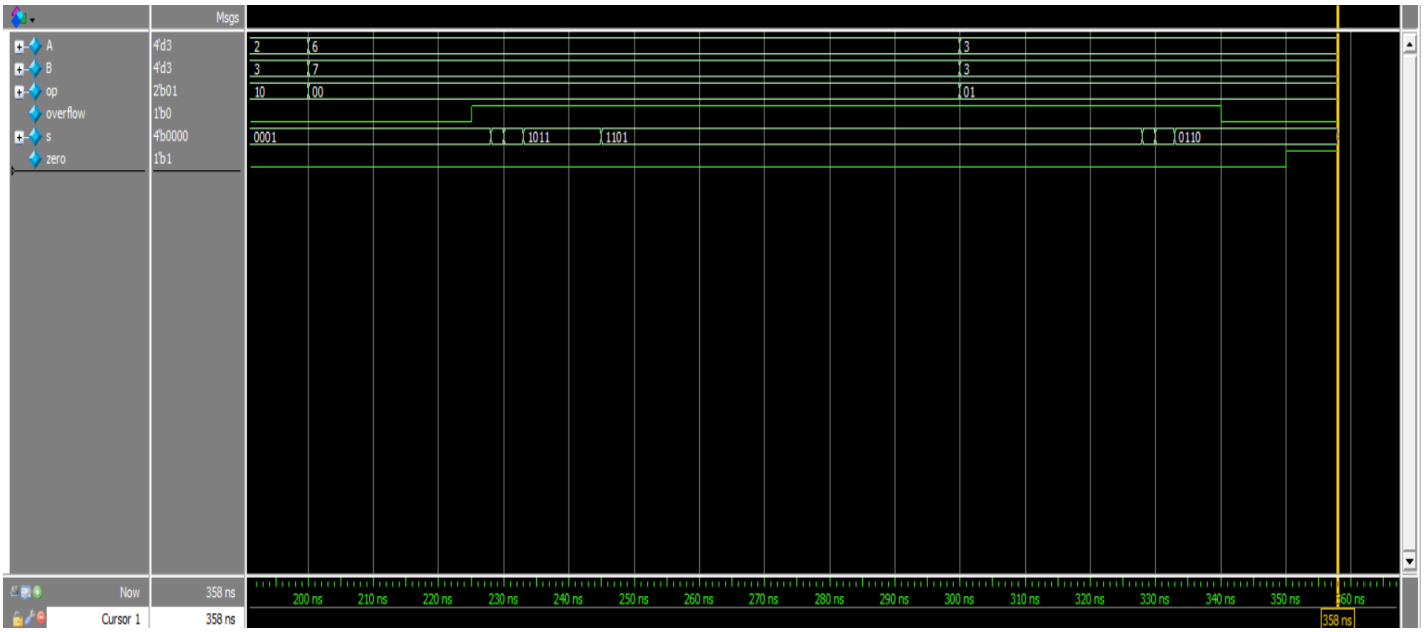
خروجی ۴ در ۳۱ نانو ثانیه سیگنال زیرو داریم که ۲ نانو ثانیه کمتر شده است.

خروجی اخر مقدار شیفت در ۲۳ نانوثانیه امده است که از قبلی ۵ نانوثانیه کمتر است.

و در نهایت سیگنال اورفلو نیز برای جمع ۴ و ۵ در ۲۰ نانو ثانیه فعال شده است.


Msgs
A
B
op
overflow
s
zero
4'd3
4'd3
2'b01
1'b0
4'b0000
1'b1
3
2
01
2
3
10
x...
1000
0001
0010
0001
Now
358 ns
Cursor 1
358 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
130 ns
140 ns
150 ns
160 ns
170 ns


Wave - Default
Msgs
A
B
op
overflow
s
zero
4'd3
4'd3
2'b01
1'b0
4'b0000
1'b1
2
3
10
6
7
00
1000
0001
0010
0001
1011
1101
Now
358 ns
Cursor 1
358 ns
110 ns
120 ns
130 ns
140 ns
150 ns
160 ns
170 ns
180 ns
190 ns
200 ns
210 ns
220 ns
230 ns
240 ns
250 ns
260 ns
270 ns




```
1.      `timescale 1ns/1ns
2.
3. module ALUCoreBehavioral (input [3:0]g1,g2,input [1:0]op,output Zero,Overflow,output [3:0]R);
4.
5.     //Converting from gray to binary
6.     wire [3:0]b1;
7.     wire [3:0]b2;
8.
9.     assign b1[3] = g1[3];
10.    assign #(3) b1[2] = b1[3] ^ g1[2];
11.    assign #(6) b1[1] = b1[2] ^ g1[1];
12.    assign #(9) b1[0] = b1[1] ^ g1[0];
13.
14.    assign b2[3] = g2[3];
15.    assign #(3) b2[2] = b2[3] ^ g2[2];
16.    assign #(6) b2[1] = b2[2] ^ g2[1];
17.    assign #(9) b2[0] = b2[1] ^ g2[0];
18.
19.    // selecting carry
20.    wire cin;
21.    wire [3:0]sb;
22.
23.    assign #(3) cin = op[0] ^ op[1];
24.
25.    assign #(3) sb[0] = cin ^ b2[0];
26.    assign #(3) sb[1] = cin ^ b2[1];
27.    assign #(3) sb[2] = cin ^ b2[2];
28.    assign #(3) sb[3] = cin ^ b2[3];
29.
30.    //adding
31.    wire cout;
32.    wire [3:1]c;
33.    wire [3:0]s;
```

```
34.     wire [3:0]go;
35.     wire [3:0]po;
36.
37.     assign #(2) go[0] = b1[0] & sb[0];
38.     assign #(2) go[1] = b1[1] & sb[1];
39.     assign #(2) go[2] = b1[2] & sb[2];
40.     assign #(2) go[3] = b1[3] & sb[3];
41.
42.     assign #(3) po[0] = b1[0] ^ sb[0];
43.     assign #(3) po[1] = b1[1] ^ sb[1];
44.     assign #(3) po[2] = b1[2] ^ sb[2];
45.     assign #(3) po[3] = b1[3] ^ sb[3];
46.
47.     assign #(7) c[1] = go[0] | (po[0] & cin);
48.     assign #(9) c[2] = go[1] | (po[1] & go[0]) | (po[1] & po[0] & cin);
49.     assign #(11) c[3] = go[2] | (po[2] & go[1]) | (po[2] & po[1] & go[0]) | (po[2] & po[1] &
po[0] & cin);
50.     assign #(13) cout = go[3] | (po[3] & go[2]) | (po[3] & po[2] & go[1]) | (po[3] & po[2] &
po[1] & go[0]) | (po[3] & po[2] & po[1] & po[0] & cin);
51.
52.     assign #(6) s[0] = b1[0] ^ sb[0] ^ cin;
53.     assign #(10) s[1] = b1[1] ^ sb[1] ^ c[1];
54.     assign #(12) s[2] = b1[2] ^ sb[2] ^ c[2];
55.     assign #(14) s[3] = b1[3] ^ sb[3] ^ c[3];
56.
57.     //overflow
58.
59.     assign #(3) Overflow = cout ^ c[3];
60.
61.     //zero
62.
63.     assign #(5) Zero = ~(s[0] | s[1]) & ~(s[2] | s[3]);
64.
65.     //abs
66.     wire [3:0] abs;
67.     wire [3:0] first_layer;
68.     wire [2:0] second_layer;
69.
70.     assign #(3) first_layer[0] = s[3] ^ s[0];
71.     assign #(3) first_layer[1] = s[3] ^ s[1];
72.     assign #(3) first_layer[2] = s[3] ^ s[2];
73.     assign #(3) first_layer[3] = s[3] ^ s[3];
74.
75.     assign #(3) abs[0] = s[3] ^ first_layer[0];
76.     assign #(2) second_layer[0] = s[3] & first_layer[0];
77.
78.     assign #(3) abs[1] = second_layer[0] ^ first_layer[1];
79.     assign #(2) second_layer[1] = second_layer[0] & first_layer[1];
80.
81.     assign #(3) abs[2] = second_layer[1] ^ first_layer[2];
82.     assign #(2) second_layer[2] = second_layer[1] & first_layer[2];
83.
84.     assign #(3) abs[3] = second_layer[2] ^ first_layer[3];
85.
86.     //selecting
87.
88.     assign #(5) mux1 = (~op[0] & s[0]) | (op[0] & s[0]);
89.     assign #(5) mux2 = (~op[0] & s[1]) | (op[0] & s[1]);
90.     assign #(5) mux3 = (~op[0] & s[2]) | (op[0] & s[2]);
91.     assign #(5) mux4 = (~op[0] & s[3]) | (op[0] & s[3]);
92.
93.     assign #(5) mux5 = (op[0] & s[1]) | (~op[0] & abs[0]);
94.     assign #(5) mux6 = (op[0] & s[2]) | (~op[0] & abs[1]);
95.     assign #(5) mux7 = (op[0] & s[3]) | (~op[0] & abs[2]);
96.     assign #(5) mux8 = (op[0] & s[3]) | (~op[0] & abs[3]);
```

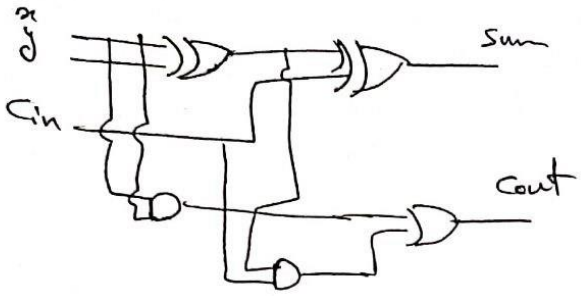
```
 97.
 98.     assign #(5) mux9 = (~op[1] & mux1) | (op[1] & mux5);
 99.     assign #(5) mux10 = (~op[1] & mux2) | (op[1] & mux6);
100.     assign #(5) mux11 = (~op[1] & mux3) | (op[1] & mux7);
101.     assign #(5) mux12 = (~op[1] & mux4) | (op[1] & mux8);
102.
103.     assign #(3) R[0] = mux9 ^ mux10;
104.     assign #(3) R[1] = mux11 ^ mux10;
105.     assign #(3) R[2] = mux12 ^ mux11;
106.     assign R[3] = mux12;
107.
108. endmodule
109.
```

با توجه به تست بنچ ها تاخیر ها بسیار افزایش یافتند حتی با استفاده از CLA مثلا در خروجی اول میبینیم تاخیر ۶۲ نانوثانیه ای داریم در صورتی که قبلا تاخیر نهایتا ۴۶ نانوثانیه بود و دلیل این است که زمان ها فیکس شدند و برابر با ماکسیمم تاخیر هستند.

رفتاری: کد راحت تر تاخیر بیشتر و فاقد هزارد.

ساختاری: کد بیشتر و دقیق تر و تاخیر های دقیق برای هر گیت ممکن است هزارد داشته باشد.

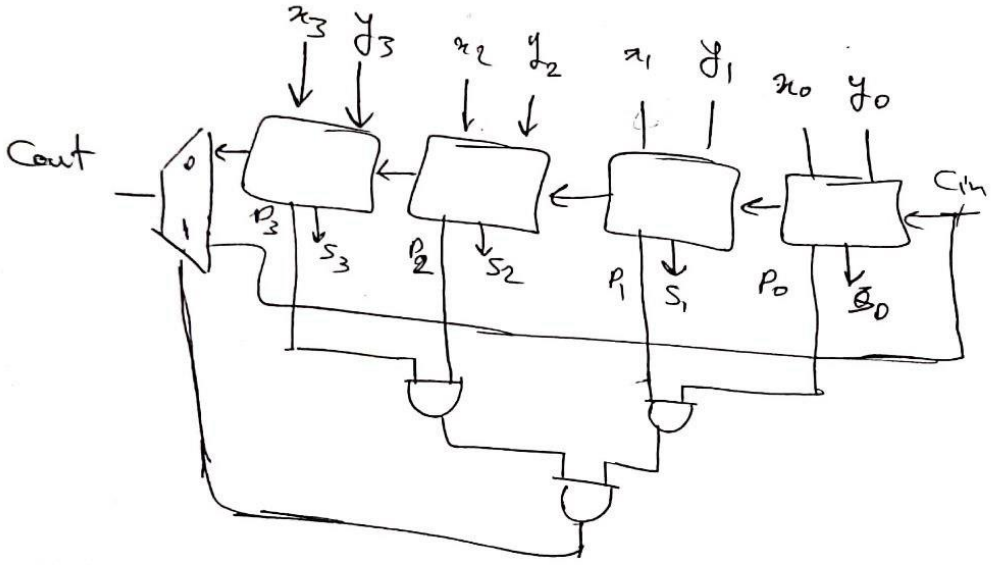
CSk:



$C_{out} = xy + (x \oplus y) C_{in}$

g p

↳ propagation



S_3: 18ns S_2: 14ns S_1: 10ns S_0: 6ns

Best case: $3 + 2 + 2 + 2 + 2 = 11ns = C_{out}$

Worst case: $C_4 = 19ns \Rightarrow C_{out} = 23ns$

Gates: RCA: 20 CLA: 35 CSK: 27 gates

CLA > CSK > RCA