توضیحات پروژه سری دوم معماری کامپیوتر:

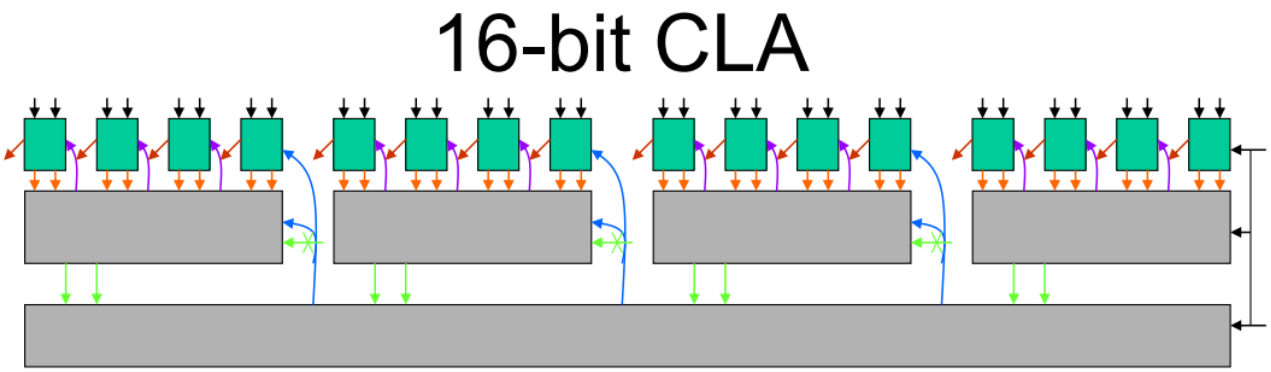
علیرضا سعیدنیا – ۴۰۰۱۰۸۳۳

توجه : پسوند فایلاها .v است .من از با اجازه استاد از سیستم وریلاگ و لاجیک استفاده کرده ام . برای استفاده از سیستم وریلاگ داخل مدلسیم روی فایلها کلیک راست کرده و properties را بزنید و type فایلها را به system Verilog تغییر دهید . فایل های پروژه هم قرار داده شده است.

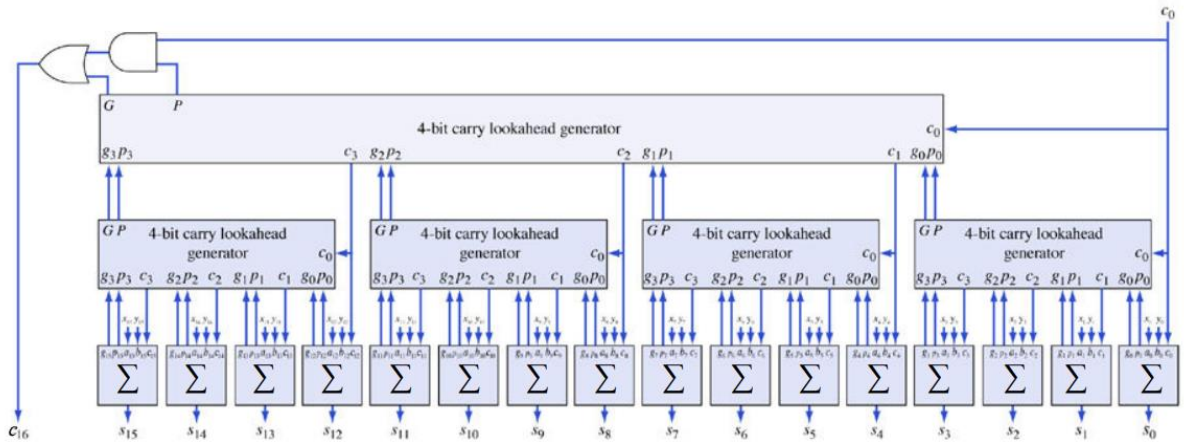
توضیحات مربوط به CLA ۶۴ بیتی :

منطق :

طبق گفته استاد این جمع کننده باید به صورت سلسله مراتبی (های آرکی و مولتی لول) پیاده سازی شود و دارای بلوک های جزیتور ۴ بیتی است (در داخل جزوه بلوک های جزیتور ۴ بیتی است)

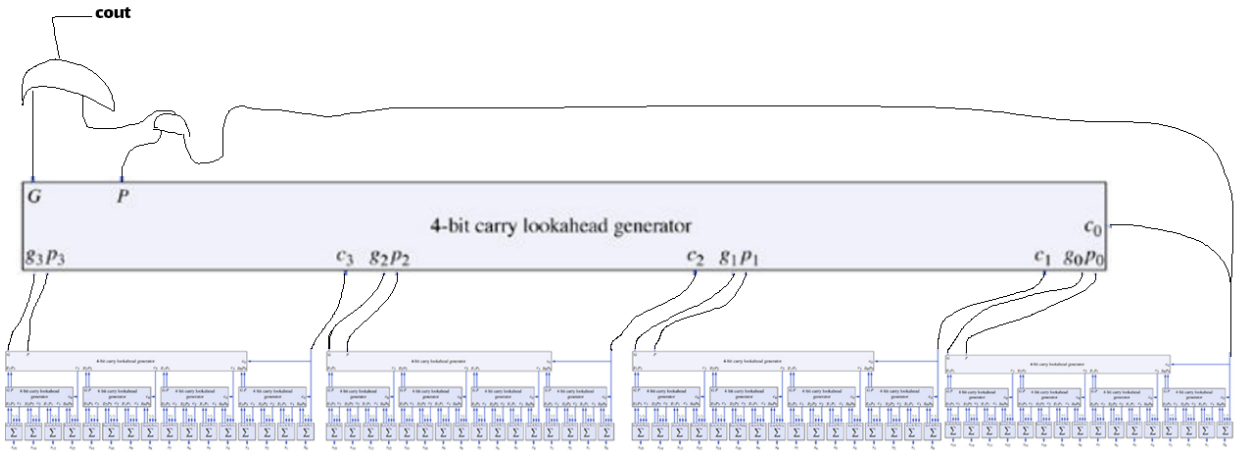


که شکل دقیق تر آن به صورت زیر است :



باید ۴ تا از این کری لوک هد ادر های ۱۶ بیتی به هم وصل شوند تا یک کری لوک هد ادر ۶۴ بیتی پدید آید.

که در نهایت باید چنین چیزی داشته باشیم:




```
module adder(x,y,p,g,cin);
input x,y,cin;
output p,g;
assign p=(~x&&y)|(~y&&x);
and (g,x,y);


endmodule;
```

کد:

توضیح متغیرهای تعریف شده در ماژول whole package :

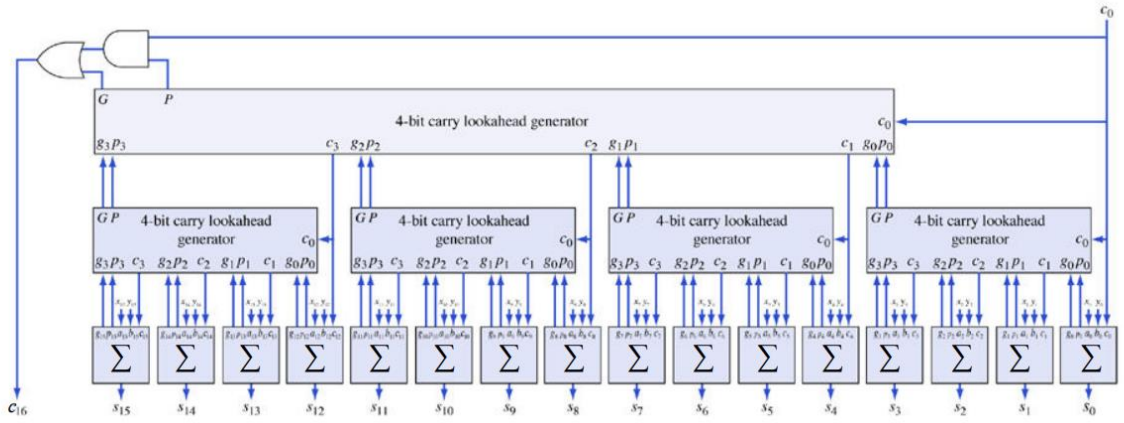
سام ۱۶ بیتی میخواهیم. ورودی و کری این ۱۶ بیتی میخواهیم. سه تا کری توسط طبقه سوم تولید میشود که باید یکجا ذخیره شود. ۱۶ تا پی و جی داریم ، ۴ تا هم پی و جی که از طبقه دوم به سوم میروند.

```
module wholepackage(x,y,firstcin,sum,PF,GF);
input logic [15:0] x,y,firstcin;
output logic [15:0] sum;
output logic PF,GF;
genvar j;
 logic [2:0] cother;
 logic [15:0] p,g;
 logic [3:0] P,G;
 logic [15:0] cin;
```

در ماژول ادر یک ادر معمولی و تک بیتی را پیاده سازی کرده ایم. وظیفه این ادر تولید پی و جی است ، در یک واحد زمانی کل پی و جی های ما ساخته میشوند (۱۶ تا)

```
for(j=0;j<16;j=j+1)
begin
adder fl(x[j],y[j],p[j],g[j],firstcin[0]);
end
```

شانزده بار پی و جی را میسازیم با حلقه فور تصویر بالا.



حالکه پی و جی کوچک را داریم نوبت به ساخت پی و جی بزرگ است که توسط چهارتا جنریتور وسط ساخته میشوند.

```
        module calculatePG(p,g,P,G);
input logic [3:0] p,g;
output logic P,G;
assign P=p[0]&&p[1]&&p[2]&&p[3];
assign G=(g[0]&&p[1]&&p[2]&&p[3])|(g[1]&&p[3]&&p[2])|(g[2]&&p[3])|g[3];
endmodule;
```

ماژول secondfloor در تصویر بالا وظیفه ساخت پی و جی بزرگ را دارد که از فرمول مربوطه بدست میاید.

چهارتا جنریتور داریم پس چهارتاP وG داریم پس طبق تصویر زیر میسازیمش

```
calculatePG k1(p[3:0],g[3:0],P[0],G[0]);
calculatePG k2(p[7:4],g[7:4],P[1],G[1]);
calculatePG k3(p[11:8],g[11:8],P[2],G[2]);
calculatePG k4(p[15:12],g[15:12],P[3],G[3]);
```

حال در طبقه سوم هایارکی نوبت به ساخت کریها هست .

همانطور که در عکس آبی رنگ قابل مشاهده است ، به جز سی این (که کری ۰ است) سه تا کری داریم که از جنریتور طبقه سوم خارج میشوند . ماژول محاسبه این کریهارا توسط طبقه سوم thirdfloor مینامیم.

```
        module thirdfloor(p,g,cin,cout);
input logic [3:0] p,g;
input logic cin;
output logic  [2:0]cout;
assign  cout[0]=(cin&&p[0])|g[0];
assign  cout[1]=(cin&&p[0]&&p[1])|(g[0]&&p[1])|g[1];
assign  cout[2]=(cin&&p[0]&&p[1]&&p[2])|(g[0]&&p[1]&&p[2])|(g[1]&&p[2])|g[2];

endmodule;
```

کریهای ۱ تا ۳ عکس آبی(۰ تا ۲ عکس بالا) با فرمول مزبور پیاده سازی میشوند.

بسیار خب حالا برمیگردیم طبقه دوم ، در طبقه دوم ما باید بیاییم به جز این ۳ تا کری طبقه سوم و یک کری این کل برنامه که میشود ۴ تا کری ، ۱۲ تا کری دیگر بسازیم ، خب پس هر جنریتور طبقه دوم وظیفه

اش ساختن ۳ تا کری هست. پس یک آرایه ۱۶ تایی تعریف میکنیم به نام cin . که بیت صفرم کری این ، بیت چهارم کری اول تولید شده در طبقه سوم ، بیت هشتم کری دوم تولید شده در طبقه سوم ، و بیت دوازدهم کری سوم تولید شده در طبقه سوم هست. همین حرف را کد میکنیم.

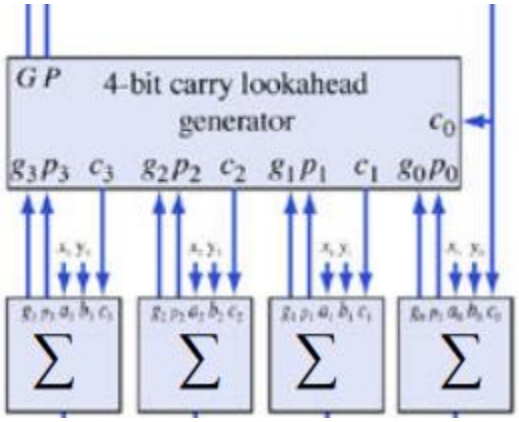
```
assign cin[0]=firstcin[0];
assign cin[4]=cother[0];
assign cin[8]=cother[1];
assign cin[12]=cother[2];
```

قبل از اینکه برگردیم طبقه اول . به طبقه سوم میرویم و پی و جی نهایی را میسازیم ، که با این فرمول محاسبه میشود :

```
assign PF=P[0]&&P[1]&&P[2]&&P[3];
assign GF=(G[0]&&P[1]&&P[2]&&P[3])|(G[1]&&P[2]&&P[3])|(G[2]&&P[3])|G[3];
```

حال برمیگردیم به طبقه دوم ، نوبت ساختن کری برای جمع کننده و زیگماهای ماست. باز هم با فرمول طبق عکس زیر میاییم و با استفاده از پی و جی هایی که از طبقه اول در مرحله اول بدست اوردیم و کری های ۴ و ۸ و ۱۲ و ۰ ، برای هر جنریتور سه تا سه تا کری میسازیم که در عکس زیر نشان داده شده است.

مثلا در w1 ، سه تا پی اول و جی اول را با سی این صفر گرفته ام و سه تا کری خروجی داده ام .



در w2، سه تا پی دوم و جی دوم را با سی این ۴ گرفته ام و سه تا کری دیگر خروجی داده ام.

و به همین صورت... – کد در عکس زیر:

```
secondcarry w1(p[3:0],g[3:0],cin[0],cin[3:1]);
secondcarry w2(p[7:4],g[7:4],cin[4],cin[7:5]);
secondcarry w3(p[11:8],g[11:8],cin[8],cin[11:9]);
secondcarry w4(p[15:12],g[15:12],cin[12],cin[15:13]);
```

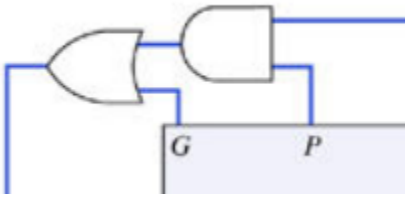
در اخر نوبت به سام میرسد که یکبار دیگر ماژول ادر دیگری میسازیم اما اینبار برای پی و جی نیست بلکه برای اسم است. باید طبیعتا در طبقه اول باید انجام شود. فرمولش به صورت عکس مزبور است که یک فور ۱۶ تایی (به دلیل وجود شانزده تا جمع کننده) نیاز داریم

```
for(j=0;j<16;j=j+1)
begin
adder2 hh(p[j],cin[j],sum[j]);
end
```

خب این کری لوک هد ادر ۱۶ بیتی ما بود ، ۴ تا ازینها را باید به هم وصل کنیم که شکلی که در نهایت باید به ان میرسیدیم را در پیجهای بالا نشان دادم. همین سیم بندی را تبدیل به کد کرده ام. در کلاس تست بنچ این وصل کردن هارا انجام میدهم. چون سه تا کری خروجی داریم + سی اوت یک ارایه سه تایی به نام فاینال سی اوت ساختم که کری هارا در ان ذخیره کنم.

```
 wholepackage me9(x[15:0],y[15:0],firstcin[15:0],sum[15:0],PF[0],GF[0]);
assign  finalcarry[0]=(firstcin[0]&&PF[0])|GF[0];
 wholepackage me4(x[31:16],y[31:16],finalcarry[0],sum[31:16],PF[1],GF[1]);
assign  finalcarry[1]=(finalcarry[0]&&PF[0]&&PF[1])|(GF[0]&&PF[1])|GF[1];
 wholepackage me5(x[47:32],y[47:32],finalcarry[1],sum[47:32],PF[2],GF[2]);
assign  finalcarry[2]=(finalcarry[1]&&PF[0]&&PF[1]&&PF[2])|(GF[0]&&PF[1]&&PF[2])|(GF[1]&&PF[2])|GF[3];
 wholepackage me6(x[63:48],y[63:48],finalcarry[2],sum[63:48],PF[3],GF[3]);
calculatePG finally(PF,GF,sixfbitP,sixfbitG);
```

خط اول ۱۶ بیت اول را به لوکهد ادر اول میدهیم با سی این. و ۱۶ بیت اول سام را برای ما با پی و جی میفرستد .که میدانیم برای بدست اوردن کری برای ادر باید طبق عکس زیر عمل کنیم:



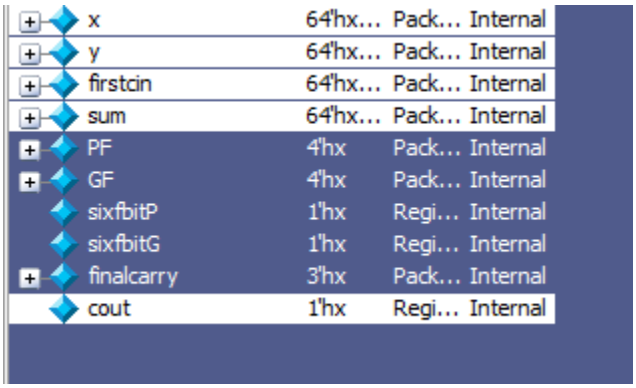
این کری اول است که در فاینال کری ۰ ذخیره میشود . همین فاینال کری ۰ میشود ورودی کری لوک هد ادر ۱۶ تایی بعدی و حالا به همین صورت فاینال کری ۱ ساخته میشود و میشود ورودی ۱۶ بیت بعدی و به همین صورت . تا اینکه کلا سه تا پی و جی داشته باشیم و به پی و جی کل ۶۴ بیت نیاز داشته باشیم .

که انهارا sixfbitp,sixfbitgl نامیده ام.

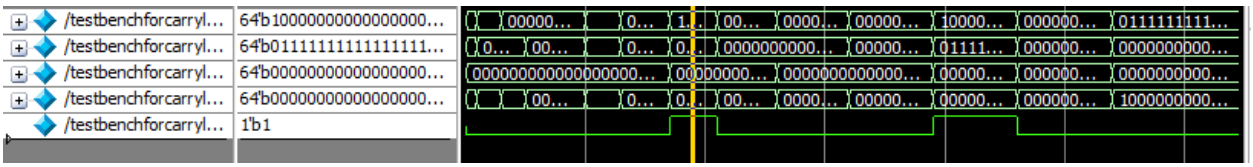
```
calculatePG finally(PF,GF,sixfbitP,sixfbitG);

assign cout=(firstcin[0]&&sixfbitP)|sixfbitG;
```

باز هم در گام نهایی کری اخر حساب میشود و این کری اخر و پایانی برنامه ماست و کری لوکهد ادر سلسله مراتبی ۶۴ بیتی ساخته شد.

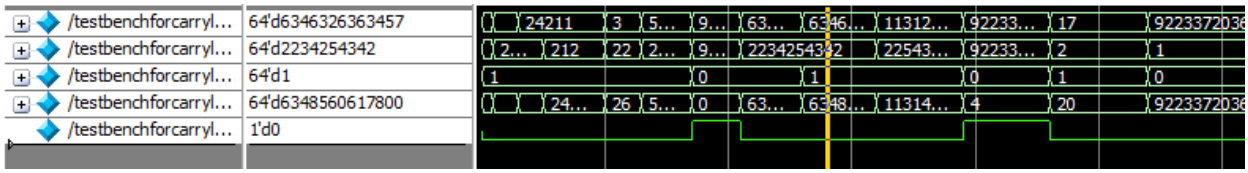
به سراغ ران گرفتن از برنامه میرویم:



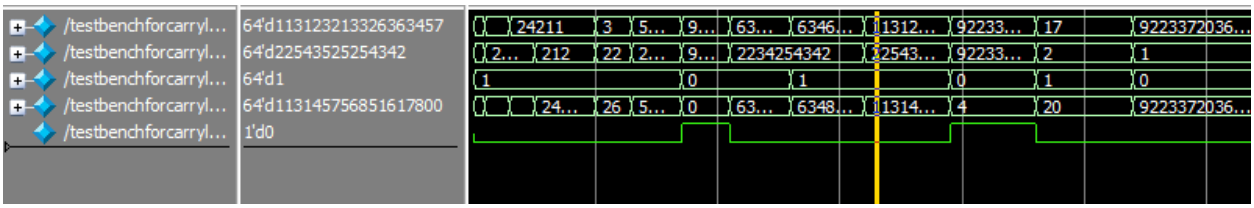
اینهارا برای ویو انتخاب میکنیم و اد ویو میزنیم



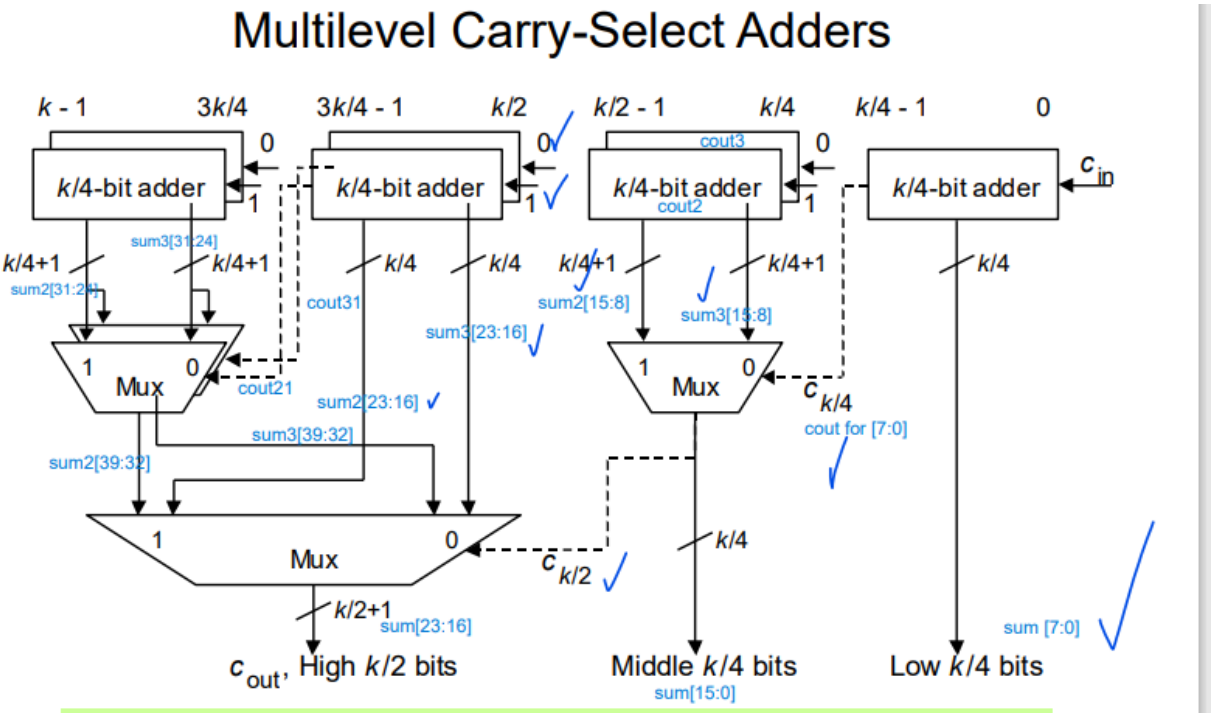
در حالت اورفلو ، سی اوت یک است.



مثالی از جمع ها بدون اورفلو و سی اوت ۰.



کری سلکت ادر ۳۲ بیتی با بلوک های ۸ تایی (پس ۴ تا بلوک میخواهیم):طبق گفته استاد باید مانند جزوه و به صورت بلوک های ۸ تایی زده شود پس همینکار را میکنیم.



کری سلکت ادر به مالتی پلکسر دو به یک نیاز دارد ، پس میاییم و ان را میسازیم.

```
module mult218bits(i0,il,select,out);
input logic [7:0] i0, il;
input logic select;
output logic[7:0] out;
assign out=(select)?i0:il;
endmodule
```

هشت بیت ورودی ایصفر و ای یک دارد و یک سلکت تک بیتی با خروجی ۸ بیتی .

```
module mult21forcarry(i0,il,select,out);
input logic i0, il;
input logic select;
output logic out;
assign out=(select)?i0:il;
endmodule
```

برای کری هم همین ماکس را

داریم ولی خب چون کری تک بیتی است ، ماکس ان هم تک بیتی است . ورودی ایصفر و اییک تک بیتی و خروجی کری مقبول تک بیتی داریم.

طبق گفته استاد ادرهای کری سلکت ادر میتوانند ریپل باشند پس یک ادر میسازیم قبل از همه.

```
module fulladder(x,y,cin,cout,sum);
  input logic x,y,cin;
  output logic cout,sum;
  assign sum=x^y^cin;
  assign cout=(x&y)|cin&(x^y);


endmodule;
```

که فرمول ان به صورت عکس مزبور است.

ادر ما باید ۸ بیتی باشد پس یعنی ۸ بار اینستنس گرفتن از این ادر تک بیتی که ما ساختیم. همین را ماژول میکنیم.

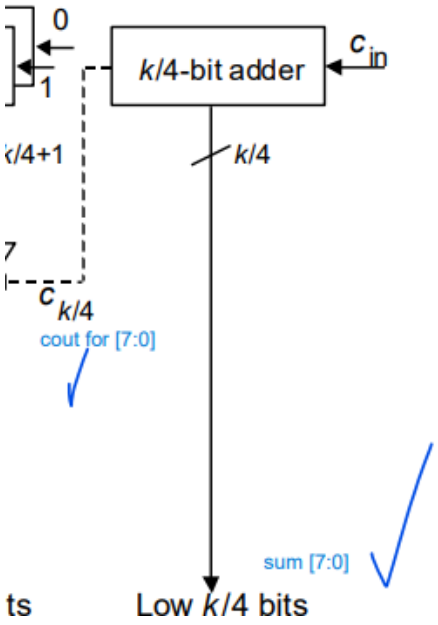
```
module eightbitadder(x,y,cin,cout,sum);
genvar j;
input logic [7:0]x,y;
output logic [7:0] sum;
input logic cin;
logic [6:0] c;
output logic cout;
fulladder me (x[0],y[0],cin,c[0],sum[0]);
for(j=1;j<7;j=j+1)
begin
fulladder me2 (x[j],y[j],c[j-1],c[j],sum[j]);
end

fulladder me3 (x[7],y[7],c[6],cout,sum[7]);

endmodule;
```

ادر هشت بیتی است پس ایکس و ایگرگ ورودی هشت بیتی داریم و سام هشت بیتی . ورودی سی این داریم و خروجی سی اوت . فور میزنیم و با استفاده از منطق ریپل ادر یک ارایه ۷ تایی از سی میسازیم که کری خروجی اولی کری ورودی ادر بعدی باشد پس هشت منهای یک یعنی هفتتا از این سی ها نیاز است.

در ماژول تست بنچ عدد ۳۲ بیتی و سی این خود را وارد میکنیم. کل کار اصلی ما در ماژول whole operation انجام میگردد پس ارگومان های ان را پاس میدهیم. در ورودی همانطور که گفته شد عدد سی و دو بیتی ورودی و سی این و خروجی سی و دو بیتی داریم. با یک سی اوت.

```
logic cout1;
logic cout2;
logic cout3;
logic coutforlower16bit;


logic cout21,cout22;
logic cout31,cout32;

logic [39:0] sum2;
logic [39:0] sum3;
```

متغیرهایی که در بالا تعریف شده اند متغیرهای کمکی ای هستند که هر کری ای که از مالتی پلکسر و هر سامی که خارج میشود را هدایت کنند.

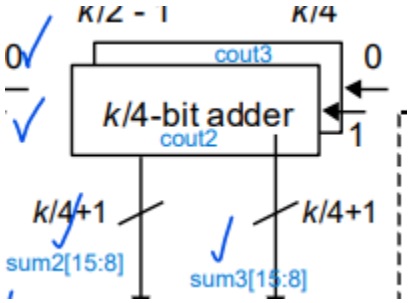


سام صفر تا ۷ باید ابتدا ساخته شود ، ts

```
eightbitadder me(x[7:0],y[7:0],cin,cout,sum[7:0]); //first adder
```

ان را میسازیم.

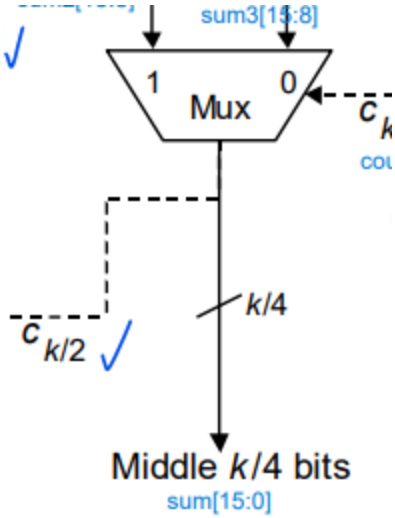
نوبت به پیاده سازی عکس زیر میرسد. ان را پیاده سازی میکنیم.




```
eightbitadder me1(x[15:8],y[15:8],1'b1,cout2,sum2[15:8]); //laye payini
eightbitadder me2(x[15:8],y[15:8],1'b0,cout3,sum3[15:8]); //laye balayi
```

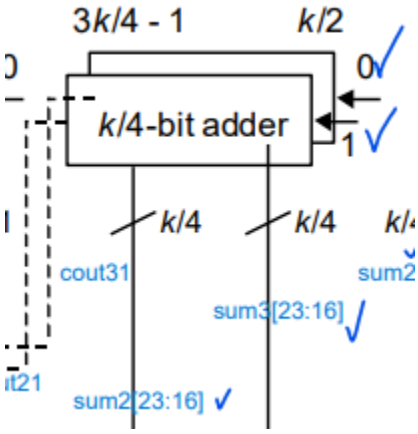
ورودی صفر و یک به ما سام ۲ و سام ۳ میدهند که هرکدام طبیعتا ۸ بیت اند چون بلوک های ما ۸ بیتی است. با مالتی پلکسر سام را هدایت میکنیم.

```
mult218bits hey2(sum2[15:8],sum3[15:8],cout,sum[15:8]); //making sum[15:0]
```



که به ما ۱۶ تا سام و سی اوت اولیه را میدهد.

```
eightbitadder mel2(x[23:16],y[23:16],1'b1,cout21,sum2[23:16]); //laye payini 8 bit seri dovom
eightbitadder mel3(x[23:16],y[23:16],1'b0,cout31,sum3[23:16]); //laye balayi 8 bit seri dovom
```

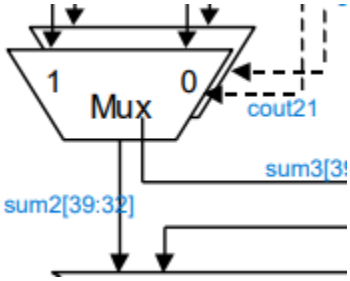


همینکار دوباره تکرار شده است. برای ادر بغلی هم همینکار را میکنم.

```
eightbitadder mel4(x[31:24],y[31:24],1'b1,cout22,sum2[31:24]);
eightbitadder mel5(x[31:24],y[31:24],1'b0,cout32,sum3[31:24]);
```

```
mult218bits muxpayin (sum2[31:24],sum3[31:24],cout21,sum2[39:32]);
mult218bits muxbala (sum2[31:24],sum3[31:24],cout31,sum3[39:32]);
```

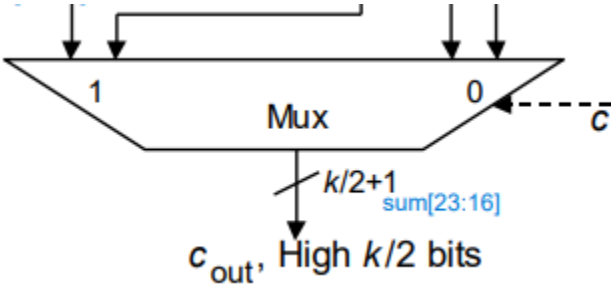
دوتا مالتی پلکسر داریم که باید سیمهای کری سه یک و دو یک را هدایت کنیم.



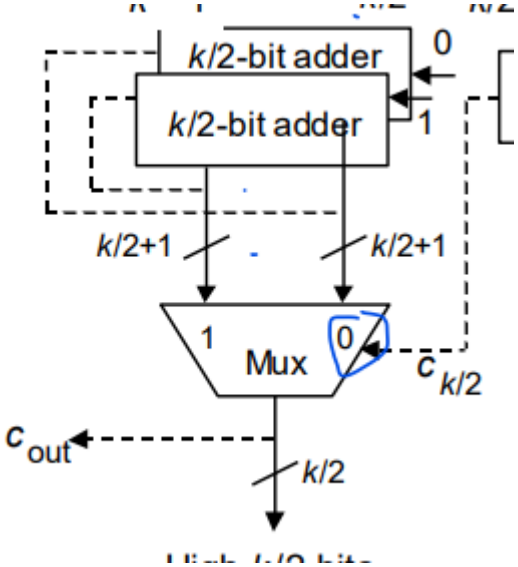
همینکار را انجام میدهیم.

ماکس اخر برای سام نهایی است . سیمهایش را طبق دو عکس پایین شامل کد و خود ماکس وصل میکنیم.

```
mult218bits finalmuxforsum (sum2[39:32],sum3[39:32],coutforlower16bit,sum[31:24]);
```

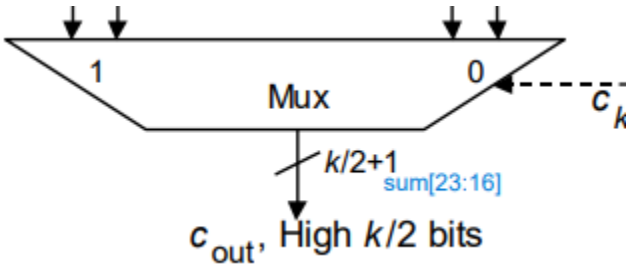


حال برمیگردیم به دو ماکس بالا ، این دو ماکس طبق سیمها دوتا کری تولید میکنند که کریها باید طبق منطق اسلاید ۲۵ استاد پیاده سازی شوند.



```
mult21forcarry muxbalayi(cout22,cout32,cout21,coutuppermux);

mult21forcarry muxpayini(cout22,cout32,cout31,coutdownermux);
```

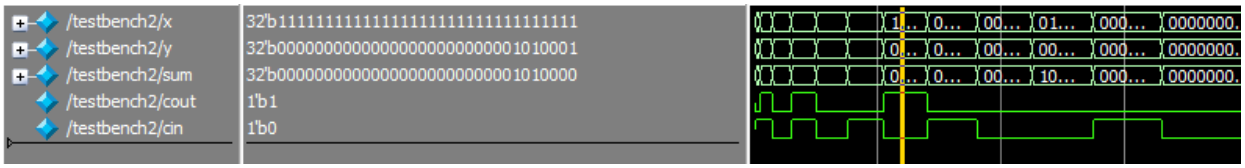
خود این دو ماکس میایند و در اخر دو کری میدهند. این دوتا کری به ماکس نهایی داده میشود.



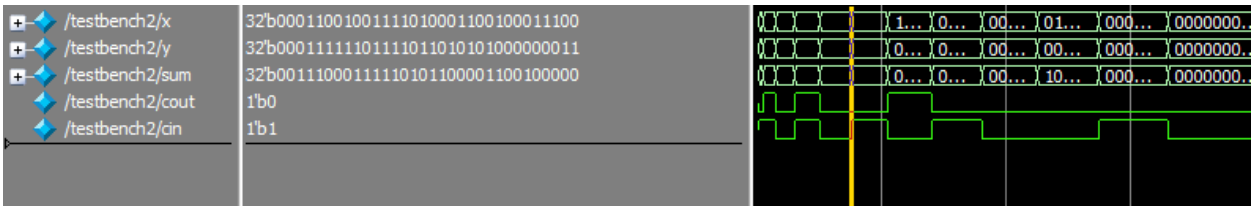

```
mult218bits finalmuxforcarry (coutuppermux,coutdownermux,coutforlower16bit,finalcout);
```

و در فاینال سی اوت ساخته میشود.

وارد ویو ها میشویم و سیمولیت میزنیم.



اتفاق افتادن اورفلو و ا شدن کری اوت .



اتفاق نیفتادن اورفلو و صفر شدن کری اوت.

مثال رندوم به صورت انسایند دسیمال.