Name : Mostafa Kermaninia

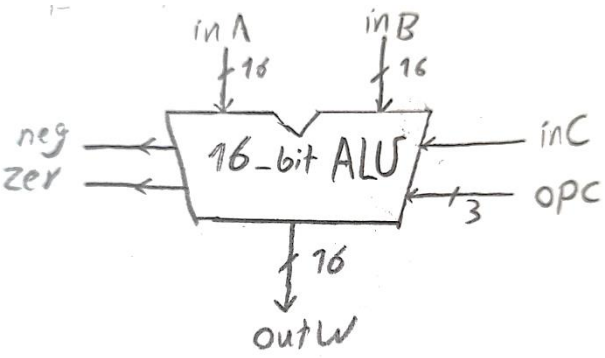
SID : 810101575

Course name : Introduction to Digital System Design

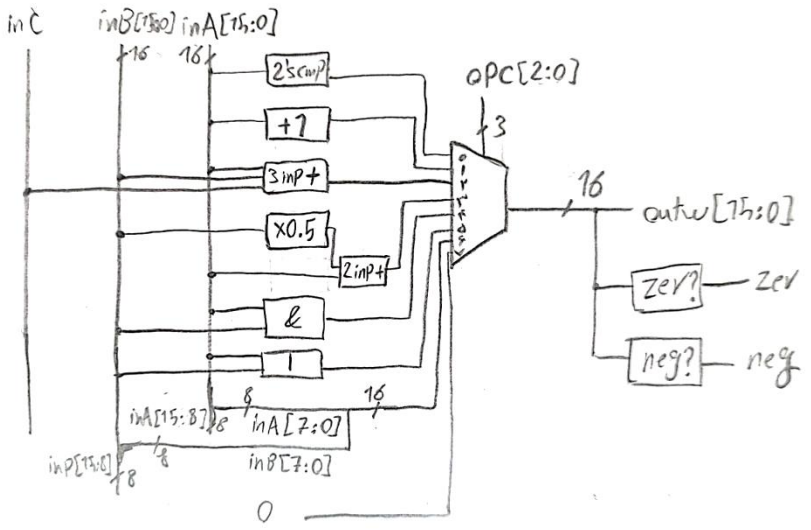
Course number : 4021810136701

برای مشخص تر شدن هدف این پروژه ابتدا ALU نهایی را بطور کلی نشان می دهیم:

opc[2]	opc[1]	opc[0]	LW
0	0	0	2's cmp(inA)
0	0	1	inA+1
0	1	0	inA + inB + inC
0	1	1	inA + inB x 0.5
1	0	0	inA & inB (Bitwise)
1	0	1	inA | inB (Bitwise)
1	1	0	{inA[7:0], inB[7:0]}
1	1	1	No operation



همچنین بدون توجه به امکان reuse کردن از مدارهای مختلف برای بخش های مختلف و بدون توجه به جزییات مثبت یا منفی و علامت دار بودن اعدادمان و داشتن intelligent circuits، در کل روند انتخاب عملیات در مدارمان بصورت زیر است:

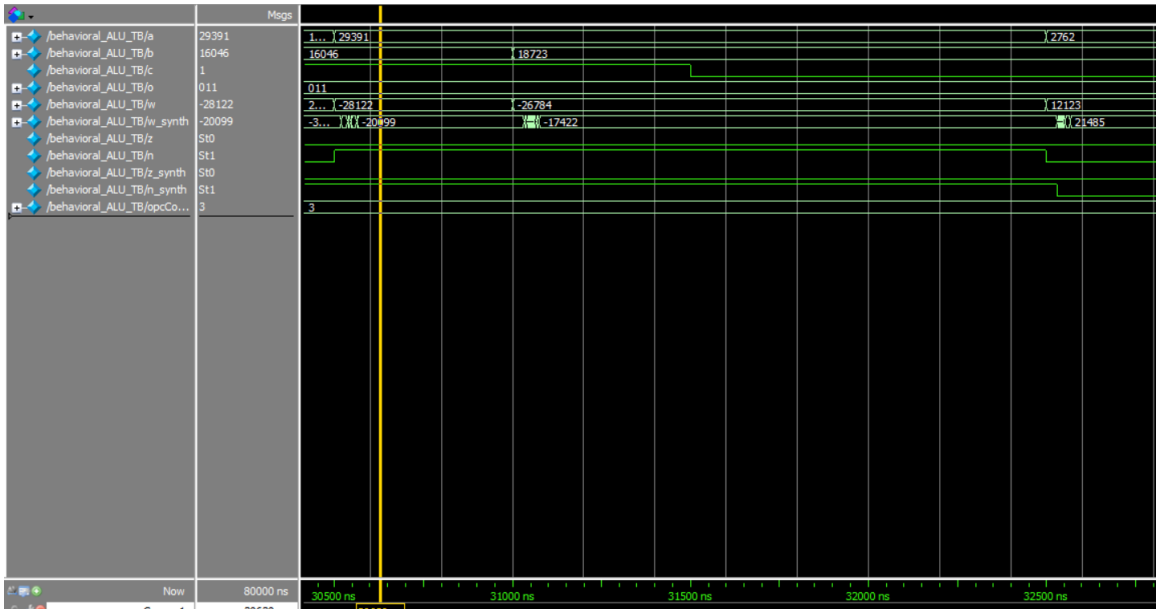


****Behavioral_ALU.sv details****

```
always @(inA, inB, inC, opc) begin
    outW = 0;
    neg  = 0;
    zer  = 0;
    case (opc)
      0: outW = ~inA + 1;
      1: outW = inA + 1;
      2: outW = inA + inB + inC;
      3: outW = inA + (inB >>> 1);
      4: outW = inA & inB;
      5: outW = inA | inB;
      6: outW = {inA[7:0], inB[7:0]};
      7: outW = 0;
      default: outW = 0;
    endcase

    neg = (outW[15]);
    zer = (outW == 0);
  end
```

* در عملیات شماره 3، برای تولید inB*0.5 راه های مختلفی هست، اما چون ما در ابتدا اعدادمان را بصورت signed تعریف نکرده ایم، در انجام عملیات ضرب و تقسیم هم مدل presynthesis و هم مدل postsynthesis به دو جواب مختلف و غلط میرسند در نتیجه فقط دو راه برای انجام عملیات 3 موجود است، یا همانند کد بالا شیفت بدهیم(آن هم فقط از نوع arithmetic درست است زیرا فرض کرده ایم اعدادمان signedهستند) و یا از نوشتار concatenation وریلاگ استفاده کنیم.

مشاهده ی پاسخ نادرست و متفاوت توسط فایل پیش و پس از سنتز بعلت استفاده از (* یا /) بجای شیفت دادن(به تفاوت خروجی های w ,w_synth و نادرست بودن هر دوی آنها دقت کنید)



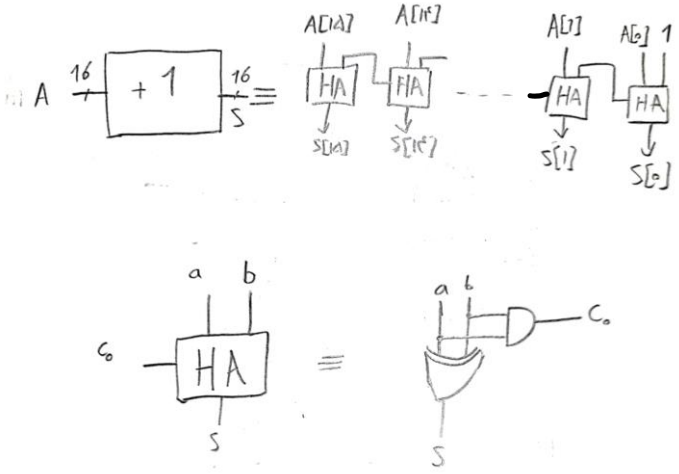
* در کد behavioral_ALU.sv فقط کلیات مسائل را نوشته ام و حتی در بیان اعداد نیز بدون درگیر شدن در تعداد بیت ها و نوع نمایش باینری، فقط یک عدد دسیمال نوشته ام تا وریلاگ خودش تبدیل به باینری با تعداد بیت لازم را انجام دهد.

* با دقیقتر شدن در مدار میفهمیم که اعداد inA و inB بصورت signed هستند و منظور این است که در استاندارد 2's complement نیز بیان شده اند (چون اگر با استاندارد sign_magnitude بودند عملیات شماره 0 که قرینه کردن با استاندارد 2's comp است و همچنین ساخت مدار هایی که عملیات جمع و increment رو این ها را انجام دهند نیازمند جزیی شدن در مدار ها میشد که بنظرم هدف این پروژه نیست). پیش از رفتن به سراغ عملیات reuse ابتدا با مدار هایی که میتوان با آنها این کد را سنتز کرد بطور حدسی مینویسم تا بعدا بتوان بهتر بخش structural را با reuse نوشت.

2's comp -0 : در این عملیات، اعدادمان را که وریلاگ هیچ تصوری از signed بودن آنها ندارد اما ما آنها را بصورت اعدادی علامت دار و با استاندارد 2's comp میبینیم را قرینه می کنیم با همان استاندارد 2's comp. پس فقط کافی است تمام بیت ها را قرینه کرده و سپس عدد 1 را به حاصل اضافه کنیم:

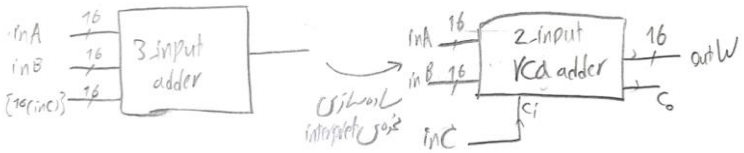
جزییات بخش increment را نکشیدم چون در بخش بعدی نشان می دهم

1- incrementer : این عملیات فقط از یک incrementer با ورودی inA تشکیل شده است که میتواند در سنتز بصورت زیر پیاده سازی بشود

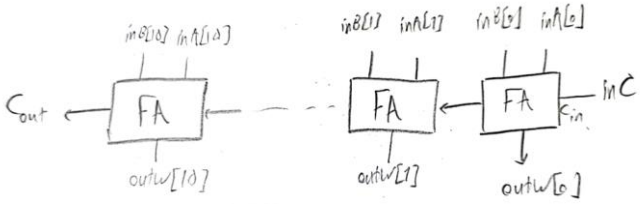


دقت کنید یک عملیات and روی تک تک بیت های a و bانجام میشود بخاطر ساختار HA ها که شاید در عملیات شماره ی 4 کمک کننده باشد.

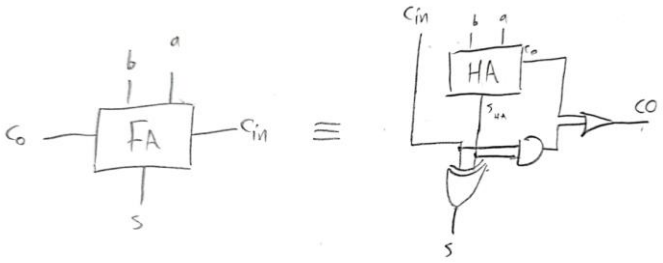
2- 3 input adder : نکته اینجاست که درست است بنظر یک adder با 3 ورودی داریم اما بدلیل تک بیتی بودن inC میتوان آنرا بعنوان carry in به یک ripple carry adder داده و این عملیات را با یک adder دو ورودی هم هندل کرد:



در نگاه کلی به این rpa داریم:



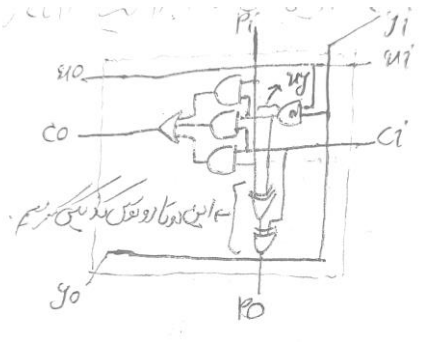
و برای هر یک از full adder ها نیز داریم:

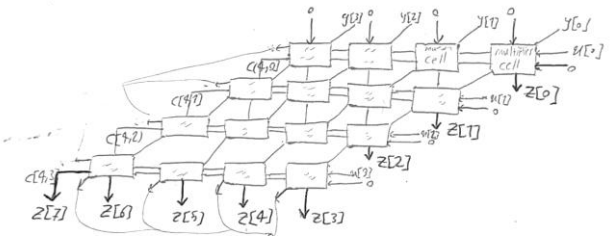


ترجیح دادم برحسب HA بکشم تا شاید امکان reuse بهتر فراهم شود

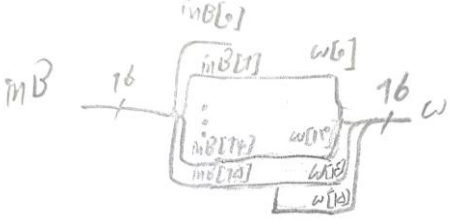
3- adder – multiplier:

اولا که عملیات ضرب کردن b در 0.5 را بجای multiplier که بصورت زیر است، میتوان با شیفت کردن انجام داد:



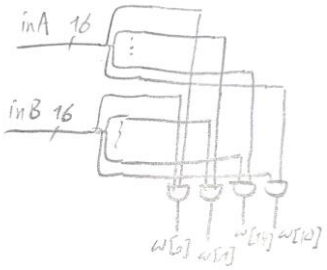


برای انجام *0.5 با شیفت دادن بیت ها(با arithmetic shifting) رسما نیازی به گیت ها نبوده و فقط با سیم ها بازی می کنیم:

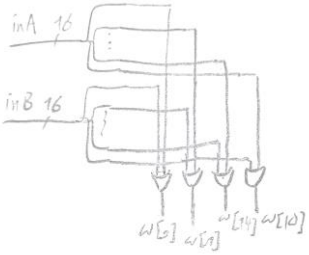


و در نهایت هم inA و inB بهمراه carry in = 0 میتوانند در یک 2 input adder که در عملیات شماره 2 با carry in = 1 از آن استفاده کردیم، جمع شوند(پس احتمال reuse از آن گیت هم هست)

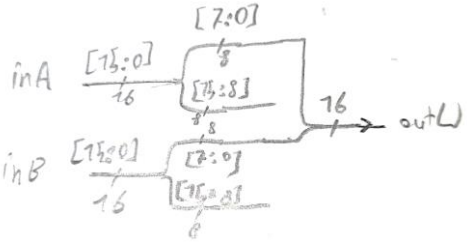
4- bitwise and: در این عملیات تک تک بیت ها and می شوند:



5- bitwise or: در این عملیات تک تک بیت ها or می شوند:



6- concatenation: در این عملیات رسما هیچ گیتی لازم نداریم، فقط با سیم ها بازی می کنیم:



7- **no operation**: برای این بخش هم کافیست مقدار inactive را برای خروجی بگذاریم که در اینجا مقدار مقدار 16 بیت صفر است.

Synthesis of behavioral code

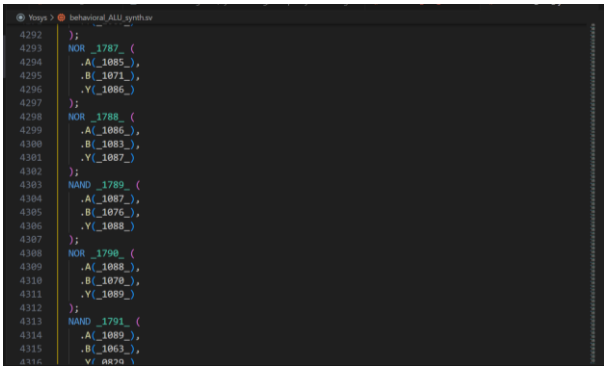
اگر فایل این نسخه را سنتز کنیم، بدون اعمال library داده شده در هر بار سنتز پاسخ هایی با اندکی اختلاف می گیریم زیرا هنوز library برای yosys مشخص نیست و با کتابخانه های دیفالت خودش سنتز می کند پس ممکن است بیش از یک حالت optimum داشته باشد. من نیز یکی از همین حالات را نهایی کردم که این بود:

```
=== behavioral_ALU ===

   Number of wires:                481
   Number of wire bits:            528
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                491
     $_AND_                         73
     $_AOI3_                        62
     $_AOI4_                         1
     $_MUX_                          6
     $_NAND_                        59
     $_NOR_                         92
     $_NOT_                         55
     $_OAI3_                        64
     $_OAI4_                        17
     $_OR_                          28
     $_XNOR_                        20
     $_XOR_                         14
```

Mycells.lib

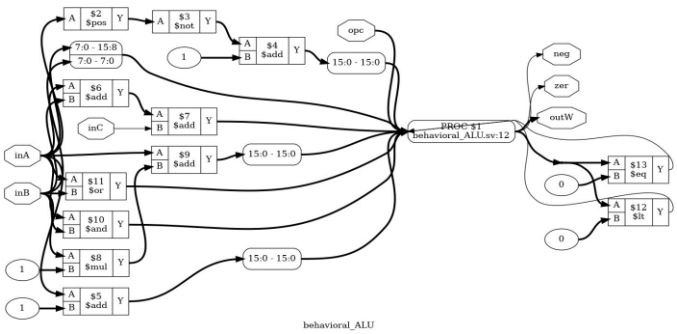
```
=== behavioral_ALU ===

   Number of wires:                1239
   Number of wire bits:            1286
   Number of public wires:            7
   Number of public wire bits:       54
   Number of memories:                0
   Number of memory bits:             0
   Number of processes:               0
   Number of cells:                 722
     NAND                           177
     NOR                            410
     NOT                            135
```

فایل post_synthesis نیز با نام behavioral_ALU_synth.sv به این صورت است:



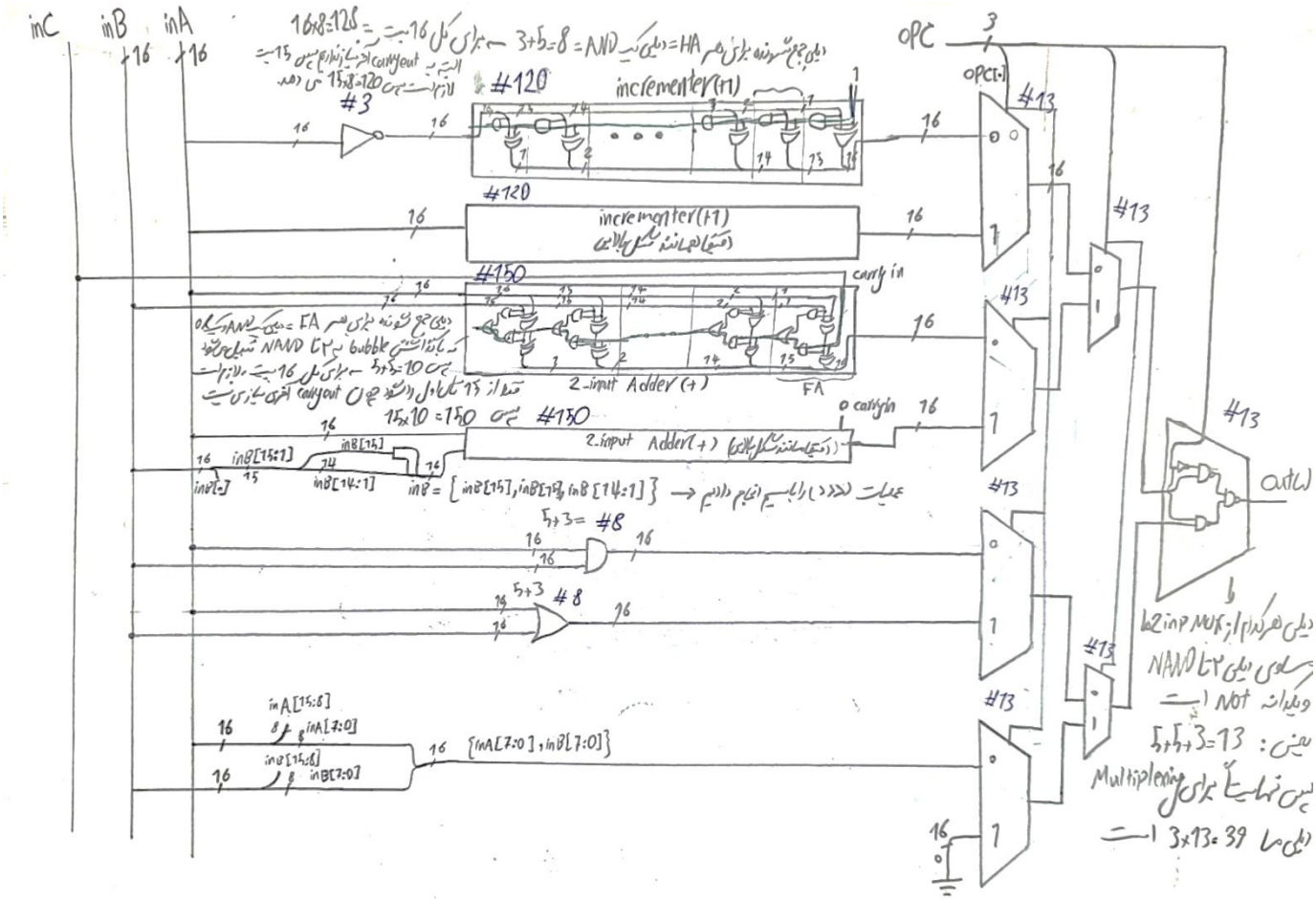
```
4292  );
4293  NOR _1787_ (
4294    .A(_1085_),
4295    .B(_1071_),
4296    .Y(_1086_)
4297  );
4298  NOR _1788_ (
4299    .A(_1086_),
4300    .B(_1083_),
4301    .Y(_1087_)
4302  );
4303  NAND _1789_ (
4304    .A(_1087_),
4305    .B(_1076_),
4306    .Y(_1088_)
4307  );
4308  NOR _1790_ (
4309    .A(_1088_),
4310    .B(_1070_),
4311    .Y(_1089_)
4312  );
4313  NAND _1791_ (
4314    .A(_1089_),
4315    .B(_1063_),
```

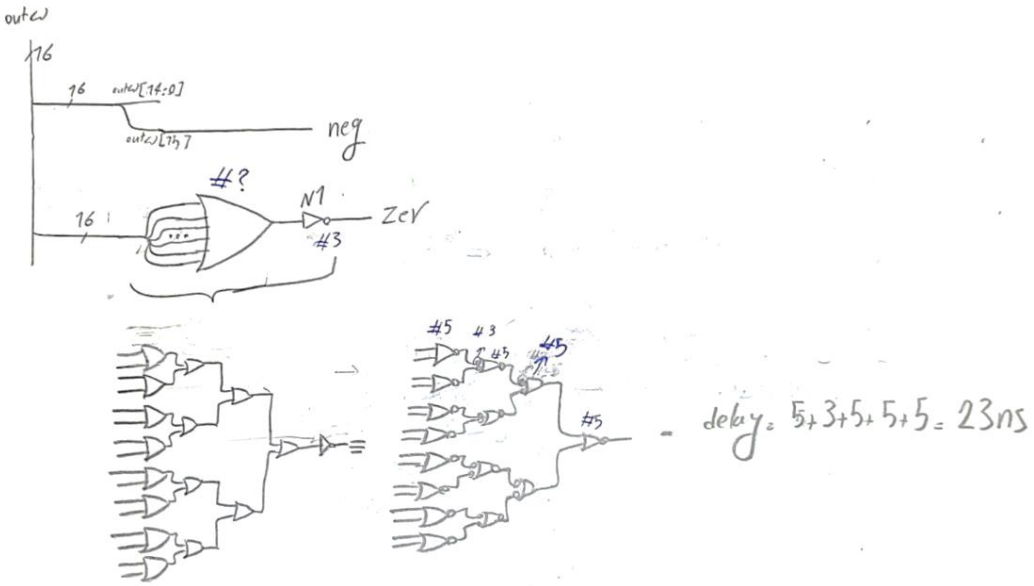
دیاگرام نه چندان جذابی که yosysبرای فایل کشیده هم اینگونه است:



(دیاگرام کشیده شده برای فایل بعد سنتز خیلی بزرگ بود و بنظر جالب نمی آمد)

شکل تقریبی و حدسی دیاگرام **behavioral**:





برای محاسبه ی حدودی و دستی worst_case ابتدا مدار presynthesis را با اندکی جزییات بیشتر و نحوه ای که حدودا ممکن است yosys آنرا سنتز کرده باشد کشیدم:(در تصویر کشیده شده، منظور از یک not یا and یا or یا muxای که 16 بیت واردش میشوند، 16 تا گیت موازی است که جهت خلاصه نویسی، یک گیت کشیده ام چون از لحاظ دیلی نیز دیلی آن مساوی است با دیلی یک گیت،زیرا تمام 16 گیت بطور موازی کار خواهند کرد)

نکات دیاگرام:

0- opc == 0 : 16 تا HA داریم که دیلی مسیری از آنها که با هم سری می شود مساوی دیلی یک and است که البته چون به carry out نهایی نیازی نداریم، دیلی 15 تا از آنها را کافیست جمع بزنیم و نتیجه 120 میشود همانطور که در تصویر هم هست. نهایتا برای این عملیات دیلی ما مساوی است با:

3 + 120 + 13 + 13 + 13 = 162ns

1- opc == 1 : این عملیات دقیقا همانند قبلی است قبل not ندارد:

120 + 13 +13 + 13 = 159ns

2- opc == 2 : در این عملیات اگر یک ripple carry adder استفاده کنیم برای هر اسلایس آن که FA است دیلی ای که ripple میشود به اندازه ی یک and و or است که میتوان آنرا به دو تا nand تبدیل کرده که 10 میشود و نهایتا 15 گیت لازم است(carry out نهایی را نمیخواهیم) که دیلی کل این مسیر برابر میشود با:

150 + 13 +13 +13 = 189ns

3- opc == 3 : این مسیر نیز همانند بخش قبلی است چون عملیات ضرب در 0.5 همانند تقسیم بر 2 است و فقط با شیفت دادن و بدون دیلی چشمگیری میتواند انجام شود و نیاز به گیت خاصی ندارد.: 189ns

4- opc == 4 : فقط 16 تا and موازی داریم که معادل است با 8+13+13+13= 47ns

5- opc == 5 : فقط 16 تا or موازی داریم که معادل است با 8+13+13+13= 47ns

6- opc == 6 : فقط عملیات های جداسازی و concatenation سیم ها را داریم و گیت خاصی لازم نیست و دیلی چشمگیری نداریم: 13+13+13= 39ns

7- opc == 7 : عملیاتی انجام نشده و فقط مقدار 0 را از mux ها رد میکنیم: 13+13+13= 39ns

نکته : بدیهتا میشد mux ها را به روش های دیگری هم ساخت، اما من بعنوان یک حدس، 8 mux ورودی ابتدایی را با mux های دو ورودی ساختم و هر mux را هم همانطور که در muxاخری نشان داده ام توسط 3تا گیت ساخته ام.

8- neg: برای این خروجی فقط نیاز به بازی با سیم ها بود و دیلی چندانی نخواهیم داشت اما نیاز است که w بطور کامل ساخته شود پس بدترین حالت دیلی آن مساوی با بدترین حالت w یا همان 189 است

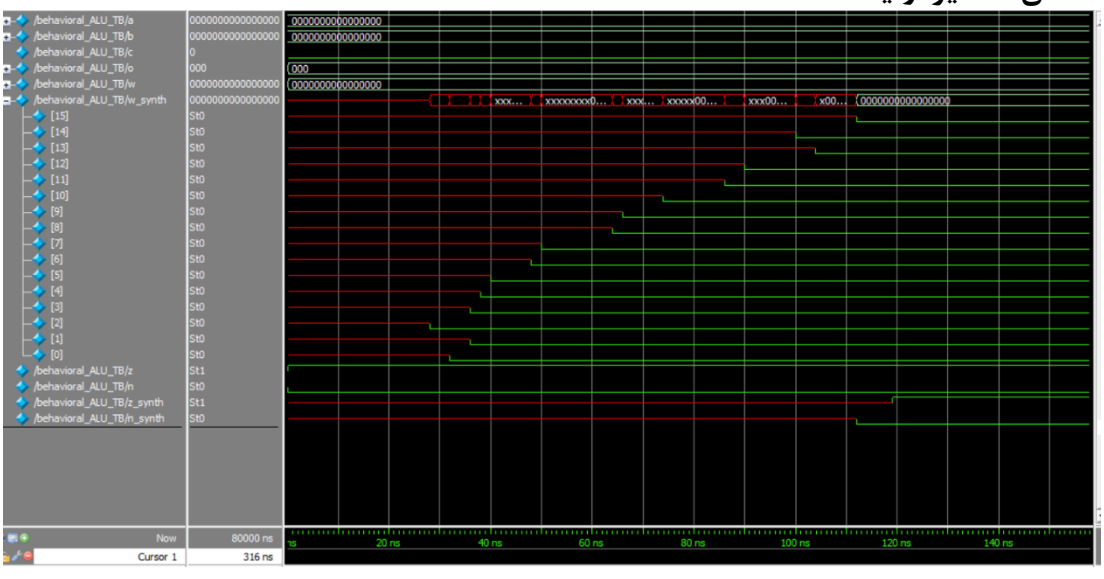
9- zer: برای این خروجی باید بعد از ساخته شدن w طبق دیاگرام به اندازه ی 23 نانو ثانیه صبر کنیم پس بدترین دیلی آن مساوی است با 189+23 = 212ns

****testbench of behavioral code****

```
  initial begin
    for (opcCounter = 0; opcCounter < 8; opcCounter =
opcCounter + 1) begin
      o = opcCounter;
      repeat (5) begin
        #500 a = $random;
        #500 b = $random;
        #500 c = $random;
        #500;
      end
    end
  end
endmod
```

درنهایت با تست بنچی که زده ام، برای هر کدام از opcode ها 5 بار اعداد رندوم به a,b,c داده می شود و کار کردن آن بخش از ALU تست می شود، برای هر opcode یک تکه از waveform در ادامه تحلیل میشود و تفاوت دو ساختار pre synthesis و post synthesis بررسی میشوند(نکته اینکه این waveform های presynthesis دیلی ندارند که بخاطر این است که در presynthesisما هیچ دیلی ای برای گیت هایمان قائل نبودیم اما بعد از سنتز، گیت ها دارای دیلی های دیفالتی هستند که در mycells.libبود)

قبل ست شدن مقادیر اولیه:



همانطور که مشخص است، w_synth و n_synthبه مدت 112 نانو ثانیه و z_synth و به مدت 119 نانوثانیه مقدار X میدهند که خب منطقی است چون مدتی طول میکشد تا مقادیر اولیه ی داده شده در فایل بعد سنتز که دیلی دارد، خروجی ها را تولید کنند(برای تولید این waveform مقادیر اولیه ای که در بلوک always میدادیم را حذف کردم تا بتوانیم شاهد بدترین حالت دیلی باشیم، مگرنه دیلی ها نمایان نمیشدند)

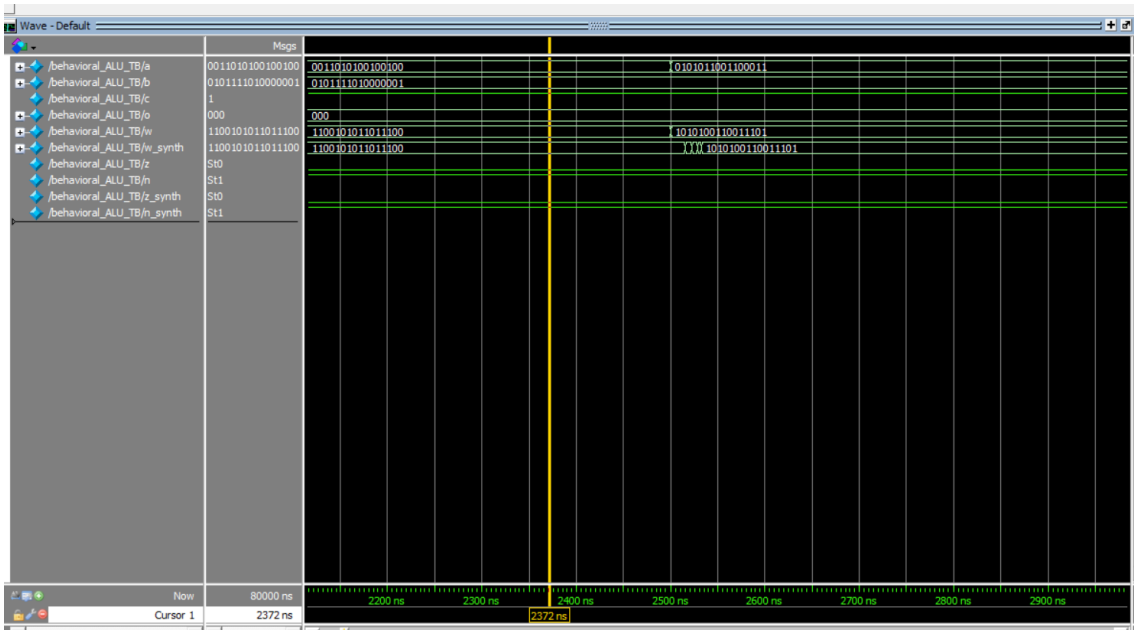
Worst_case delay: بدیهتا برای محاسبه ی worst case delay برای فایل بعد از سنتز نیاز به گشتن بدنبال حالت خاصی نیست، زیرا همانطور که در تصویر هم مشخص است، بدترین حالت ممکن دقیقا همین حالت آمدن مقادیر اولیه است زیرا قبل از آن تمام بیت های W_synthمساوی X بوده اند و z_synth هم x بوده است پس برای ست شدن مقادیر اولیه باید تک تک بیت های W به مقدار نهایی برسند تا از حالت X خارج شود پس چون در mycells.lib نیز همانطور که در کد آن واضح است، مقدار دیلی To0 , To1 , Toz برای گیت هایمان متفاوت نیست، پس بدترین دیلی همین دیلی ای است که در راه ست شدن مقادیر اولیه طی میشود(فاقد ازینکه آن مقادیر اولیه چیستند) که اتفاقا شباهت خوبی با حدس های قبلی ما دارند. البته که بدیهتا انتظار نداریم وریلاگ دیلی هایش دقیقا همانند حدس های ما باشد چون نمیدانیم چگونه عملیات سنتز کردن انجام شده که بهترین بازدهی را داشته باشد، ممکن است برای تمام گیت ها از کارنومپ استفاده کند تا به 2level logic برسد که در آن صورت قطعا مثلا با adder ای که ما با 16*2 لول گیت نوشتیم فرق میکند دیلی هایش!

*در این کتابخانه دیلی های to1,to0,toz همگی یکسانند

```
1  `timescale 1ns/1ns
2  module NOT(A, Y);
3  input A;
4  output Y;
5  assign #3 Y = ~A;
6  endmodule
7
8  module NAND(A, B, Y);
9  input A, B;
10 output Y;
11 assign #5 Y = ~(A & B);
12 endmodule
13
14 module NOR(A, B, Y);
15 input A, B;
16 output Y;
17 assign #5 Y = ~(A | B);
18 endmodule
19
20 module DFF(C, D, Q);
21 input C, D;
22 output reg Q;
23 always @(posedge C)
24     Q <= D;
25 endmodule
```

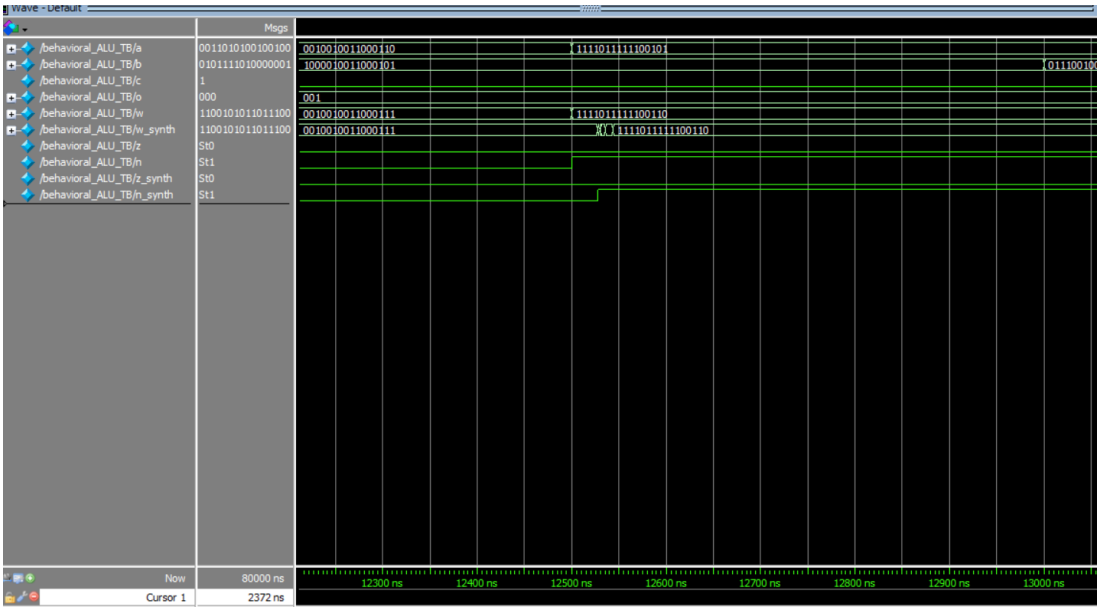
0- opc == 0 :

مثلا اینجا میبینیم با تغییر کردن inA بلافاصله مقدار خروجی مدار presynthesis مقدار جدید را که a 2's comp است میگیرد اما مدار post synthesis با مقداری دیلی همان مقادیر را به بیت های W میدهد و فلگ های z, n هم تغییری نمیکنند چون خروجی نه منفی شده و نه صفر.

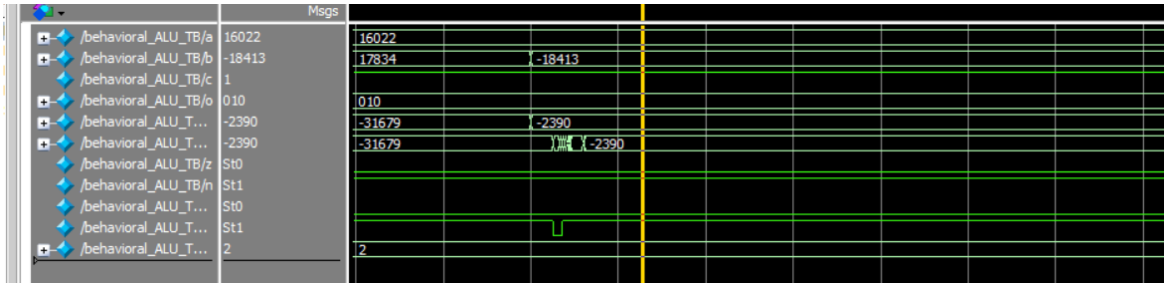


1- opc == 1 :

در اینجا نیز با تغییر مقدار a مقدار خروجی ها مساوی را یکی بیشتر از آن شده و همچنین چون مقدار جدید خروجی منفی است مقدار فلگ n هم برای دو مدار تغییر کرده.

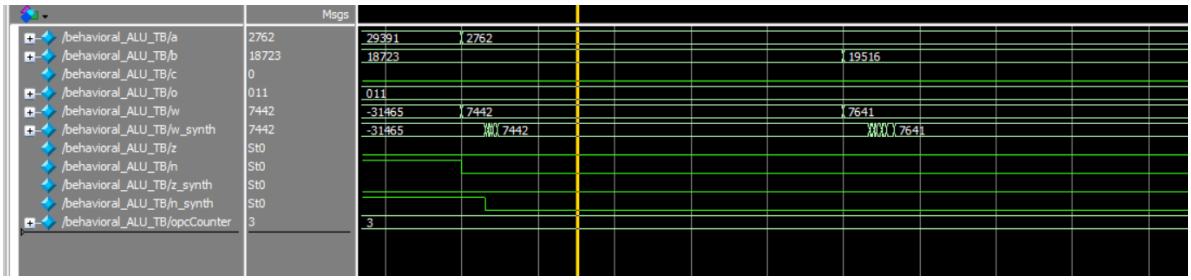


2- opc == 2 :



اینجا همانطور که مشخص است، مقدار A,B,C جمع شده و در خروجی هر دو مدار قرار گرفته(در برخی تست کیس های این pcode چون احتمال اورفلو وجود دارد بخاطر اینکه وریلاگ بیت به بیت جمع میزند و ما دو ورودی 16 بیتی داریم و یک خروجی 16 بیتی بیشتر نداریم و ممکن است پاسخ اصلی 17 بیت شود)

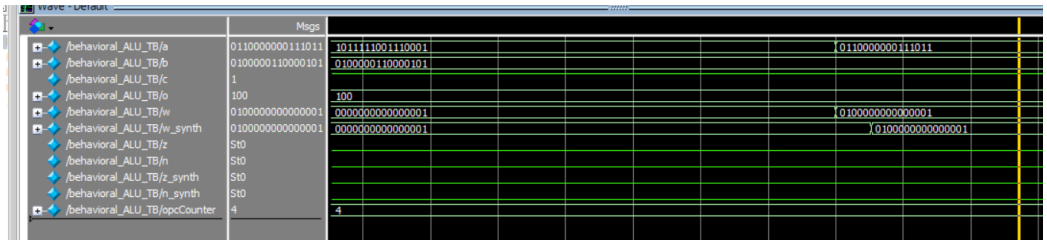
3- opc == 3 :

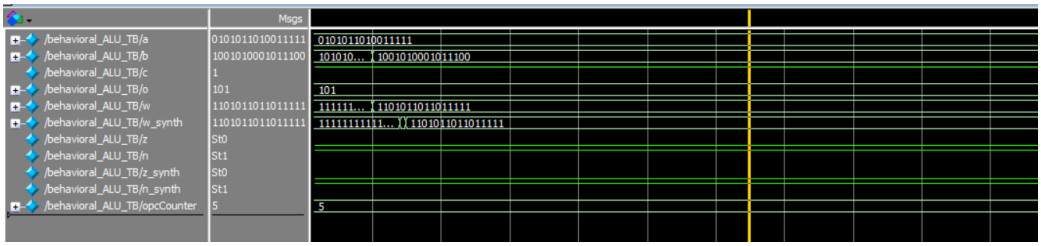


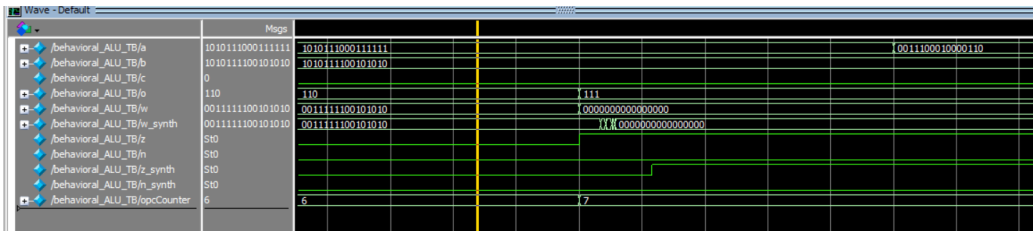
در اینجا هم بخوبی مقدار a و نصف b جمع شده اند (البته در این کیس هم امکان اورفلو وجود دارد همانند کیس قبلی، ولی اگر باینری نگاه کنیم و اصراری روی interpret خاصی نباشد مشکل ندارد)

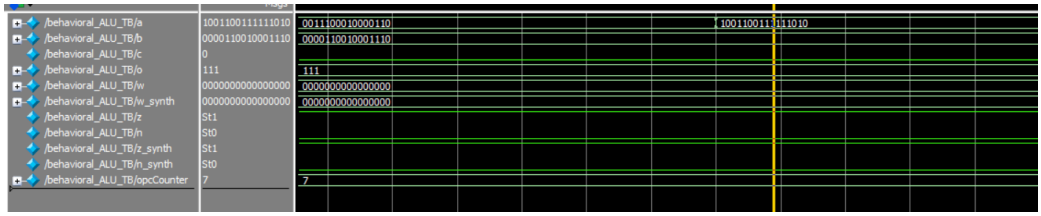
4- opc == 4,5,6,7 :

این عملیات ها هم ساده هستند و به وضوح جواب درستی از هر دو مدار قبل و بعد از سنتز میگیریم:

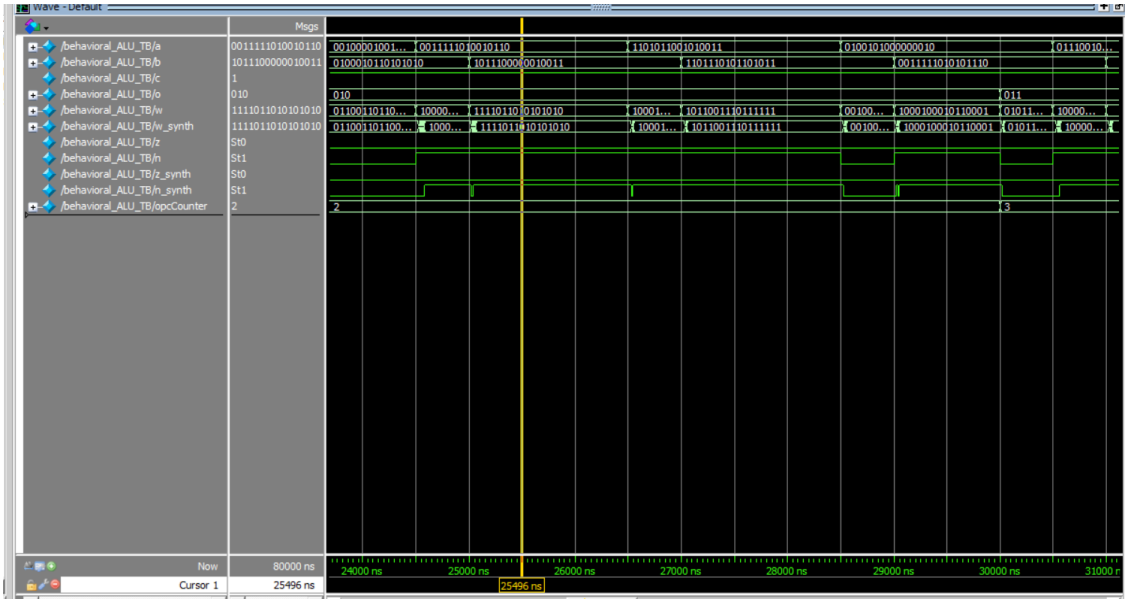








برای بقیه ی opcodeها هم به همیت ترتیب میتوان waveformهارا مقایسه کرد.،یک عکس کلی از waveformها قرار میدهم:



همانطور که مشخص است برای z, n در حالت بعد از سنتر به glitchهایی نیز میخوریم که چون دیلی مسیر های مختلف متفاوت است، رسیدن به آنها طبیعی است.

Simulation speed:

اگر منظور از این سوال، میزان دیلی ها باشد که همانطور که دیدیم در فایل قبل سنتز چون خودمان دیلی ای نذاشتیم بدون دیلی تمام عملیات ها انجام میشد اما در فایل بعد از سنتز دیلی های library نوشته شده اعمال میشد، از طرفی اگر هم قرار بود طبق حدسیات دیلیهای مدارها را بذاریم، دیلی های محاسبه شده بیشتر از دیلی های فایل سنتز شده میشد همانطور که در دیاگرام کشیده شده هم دیدیم

از طرفی اگر ابعاد فایل کد مدنظر باشد خب بدیهتا فایل سنتز شده(netlist) بسیار ریزتر و طولانی تر بوده و خوانده شدن و شبیه سازی شدن خط به خط آن قاعدتا بیشتر طول میکشد(البته که در فایل اصلی قبل سنتز درست است تعداد خطوط کد کمتر است اما پیچیدگی هر خط بیشتر بوده و نیاز به پردازش بیشتری دارد در نتیجه میتوان بطور دستی حساب کرد زمان شبیه سازی را و مقایسه کرد)

در نهایت با استفاده از نوشتن این دستور، زمان حدودی شبیه سازی را بدست می آوریم:

برای ماژول presynthesis:

```
set sim_time [time {run -all}]
# 69715 microseconds per iteration
set sim_time [time {run -all}]
# 68224 microseconds per iteration
set sim_time [time {run -all}]
# 68431 microseconds per iteration
set sim_time [time {run -all}]
```

حدود 68000 میکروثانیه

برای ماژول postsynthesis:

```
set sim_time [time {run -all}]
# 82305 microseconds per iteration
set sim_time [time {run -all}]
# 80213 microseconds per iteration
set sim_time [time {run -all}]
# 77344 microseconds per iteration
set sim_time [time {run -all}]
# 80823 microseconds per iteration
VSIM 35> set sim_time [time {run -all}]
# 78778 microseconds per iteration

VSIM 35>
```

حدود 80000 میکرو ثانیه

در نتیجه در ماژول بعد از سنتز، تعداد زیاد خط ها بر ساده بودن آنها پیشی گرفته و باعث اندکی زیادتر شدن زمان شبیه سازی می شود.

برای بهتر متوجه شدن تفاوت بین این دو در تست بنچ، تعداد دفعات repeat را زیاد کردم که به این رسیدم:

برای ماژول presynthesis:

microseconds per iteration 1471359 #

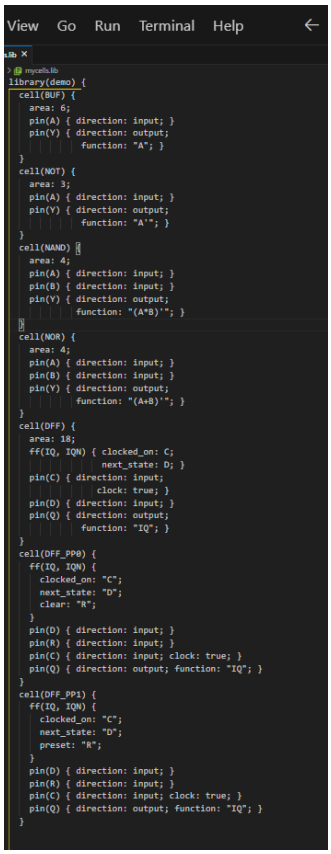
برای ماژول postsynthesis:

microseconds per iteration 26575187 #

که به وضوح برای بعد از سنتز بیشتر طول میکشد(ده برابر بیشتر)

از لحاظ سخت افزاری در حالت نخست، احتمالا از گیت ها و مدار هایی همانند چیزی که در دیاگرام کشیدم استفاده میشد اما بعد از سنتز، با توجه به کتابخانه ی ما، فقط از NAND,NOR,NOTبا دو ورودی برای ساخت مدار استفاده شد زیرا فقط همین گیت ها در mycells.libبود (گیت های tri state هم بودند اما بدرد این ساختار نمیخورند)

گیت های استفاده شده توسط yosys برای سنتز



Synthesis of structural code**

برای این بخش، کد behavioral را برداشته و دونه دونه تغییراتی که حس میکنم باعث بهتر شدن آن می شود را اعمال میکنم:

0- در حالت اولیه و بدون تغییر همانطور که قبلا دیدیم داریم:

```
=== behavioral_ALU ===

   Number of wires:                 1239
   Number of wire bits:             1286
   Number of public wires:             7
   Number of public wire bits:        54
   Number of memories:                 0
   Number of memory bits:              0
   Number of processes:                0
   Number of cells:                  722
     NAND                            177
     NOR                             410
     NOT                             135
```

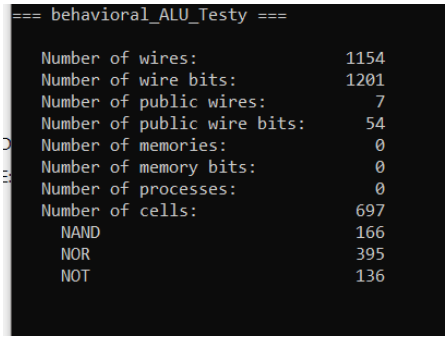
1- اولین تغییری دادم، تغییر دادن اعداد از دسیمال به فرمت استاندارد ('b) بود که درست است تاثیری در سخت افزار ندارد اما بالاخره در راستای دیدگاه ساختاری است.

2- سپس با توجه به دیاگرامی که برای اولین حالت کشیدم میتوان دید که دو تا incrementer استفاده شده و دوتا هم adder استفاده شده است که نیازی به اینهمه گیت نداریم بلکه میتوانیم سه تا پارامتر جدید تعریف کنیم بنام Aprim,Bprim,Cprim که این سه پارامتر قبل از شروع بلوک always طوری تنظیم شوند که برای چهار تا opcode ابتدایی کافی باشد فقط آن سه تا را جمع بزنیم و فقط از سه تا MUS استفاده شود که هر کدام 4 انتخاب دارند. نتیجه:

```
logic [15:0] Aprim, Bprim, Cprim;
assign Aprim = (opc == 0) ? ~inA : inA;
assign Bprim = (opc == 0 | opc == 1) ? 0 : (opc == 2) ? inB : {inB[15], inB[15:1]};
assign Cprim = (opc == 0 | opc == 1) ? 1 : (opc == 2) ? inC : 0;
always @(inA, inB, inC, opc) begin
  outW = 16'b0;
  neg  = 1'b0;
  zer  = 1'b0;
  case (opc)
    3'd0: outW = Aprim + Bprim + Cprim;
    3'd1: outW = Aprim + Bprim + Cprim;
    3'd2: outW = Aprim + Bprim + Cprim;
    3'd3: outW = Aprim + Bprim + Cprim;
    3'd4: outW = inA & inB;
    3'd5: outW = inA | inB;
```

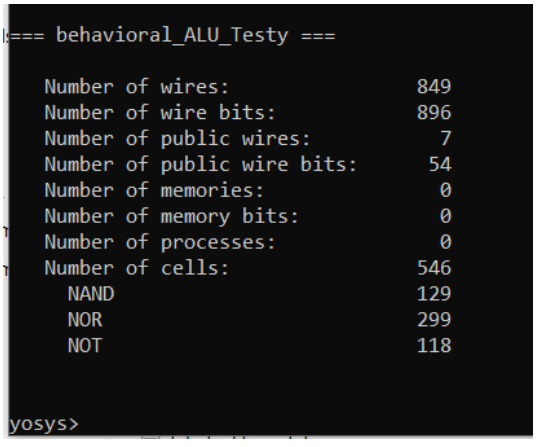
```
      3'd6: outW = {inA[7:0], inB[7:0]};
      3'd7: outW = 16'b0;
      default: outW = 16'b0;
    endcase

    neg = outW[15];
    zer = ~|outW;
  end
```

بعد از اعمال Aprim :



بعد از اعمال Bprim , Cprim:

```
logic [15:0] Aprim, Bprim, multiple_result;
  assign Aprim = (opc == 3'b000) ? ~inA : inA;
  assign Bprim = (opc == 3'b000 | opc == 3'b001) ? 16'b0 : (opc == 3'b010) ? inB :
{inB[15], inB[15:1]};
  assign Cprim = (opc == 3'b000 | opc == 3'b001) ? 1'b1 : (opc == 3'b010) ? inC : 1'b0;
  always @(inA, inB, inC, opc) begin
...
    case (opc)
      0: outW = Aprim + Bprim + Cprim;
      1: outW = Aprim + Bprim + Cprim;
      2: outW = Aprim + Bprim + Cprim;
      3: outW = Aprim + Bprim + Cprim;
...
```



3- اما همچنان می توان کار را بهتر کرد زیرا تا اینجای کار در 4 کیس ابتدایی، در هر کدام یک adder سه تایی داریم، که میتوان آنرا به بالای کیس ها انتقال داد تا فقط یک adder داشته باشیم:

```
...
  logic [15:0] Aprim, Bprim, Cprim, multiple_result;
  assign Aprim = (opc == 3'b000) ? ~inA : inA;
  assign Bprim = (opc == 3'b000 | opc == 3'b001) ? 16'b0 : (opc == 3'b010) ? inB :
{inB[15], inB[15:1]};
  assign Cprim = (opc == 3'b000 | opc == 3'b001) ? 1'b1 : (opc == 3'b010) ? inC : 1'b0;
  assign multiple_result = Aprim + Bprim + Cprim;

  always @(inA, inB, inC, opc) begin
    outW = 16'b0;
    neg  = 1'b0;
    zer  = 1'b0;
    case (opc)
      3'b000:  outW = multiple_result;
      3'b001:  outW = multiple_result;
      3'b010:  outW = multiple_result;
      3'b011:  outW = multiple_result;
      3'b100:  outW = inA & inB;
      3'b101:  outW = inA | inB;
      3'b110:  outW = {inA[7:0], inB[7:0]};
      3'b111:  outW = 16'b0;
      default: outW = 16'b0;
    endcase
...
```

```
1. Printing statistics.

== structural_ALU ===

  Number of wires:                847
  Number of wire bits:            894
  Number of public wires:           7
  Number of public wire bits:      54
  Number of memories:               0
  Number of memory bits:            0
  Number of processes:              0
  Number of cells:                519
    NAND                          146
    NOR                           266
    NOT                           107
```

4- از طرف دیگر درست است که دادن مقدار اولیه واجب است اما میتوان اگر واقعا قصد شدید بر optimal کردن داریم، آنها را نیز حذف کنیم و همچنین از کمترین تعداد assign ممکن استفاده کنیم و در نهایت هم کیس های چهارگانه ی اول که عملیات یکسانی انجام میدهند را به کیس دیفالت ارجاع دادم که کیس های اضافی هم حذف شوند:

```
=== structural_ALU ===

   Number of wires:                854
   Number of wire bits:            901
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                511
     NAND                          159
     NOR                           249
     NOT                           103
```

```
logic [15:0] multiple_result= ((opc == 3'b000) ? ~inA : inA)+
((opc == 3'b000 | opc == 3'b001) ? 16'b0 : (opc == 3'b010) ? inB : {inB[15], inB[15:1]})+
((opc == 3'b000 | opc == 3'b001) ? 1'b1 : (opc == 3'b010) ? inC : 1'b0);
always @(inA, inB, inC, opc) begin
  case (opc)
    3'b100:  outW = inA & inB;
    3'b101:  outW = inA | inB;
    3'b110:  outW = {inA[7:0], inB[7:0]};
    3'b111:  outW = 16'b0;
    default: outW = multiple_result;
  endcase
  neg = (outW[15]);
  zer = (outW == 16'b0);
end
```

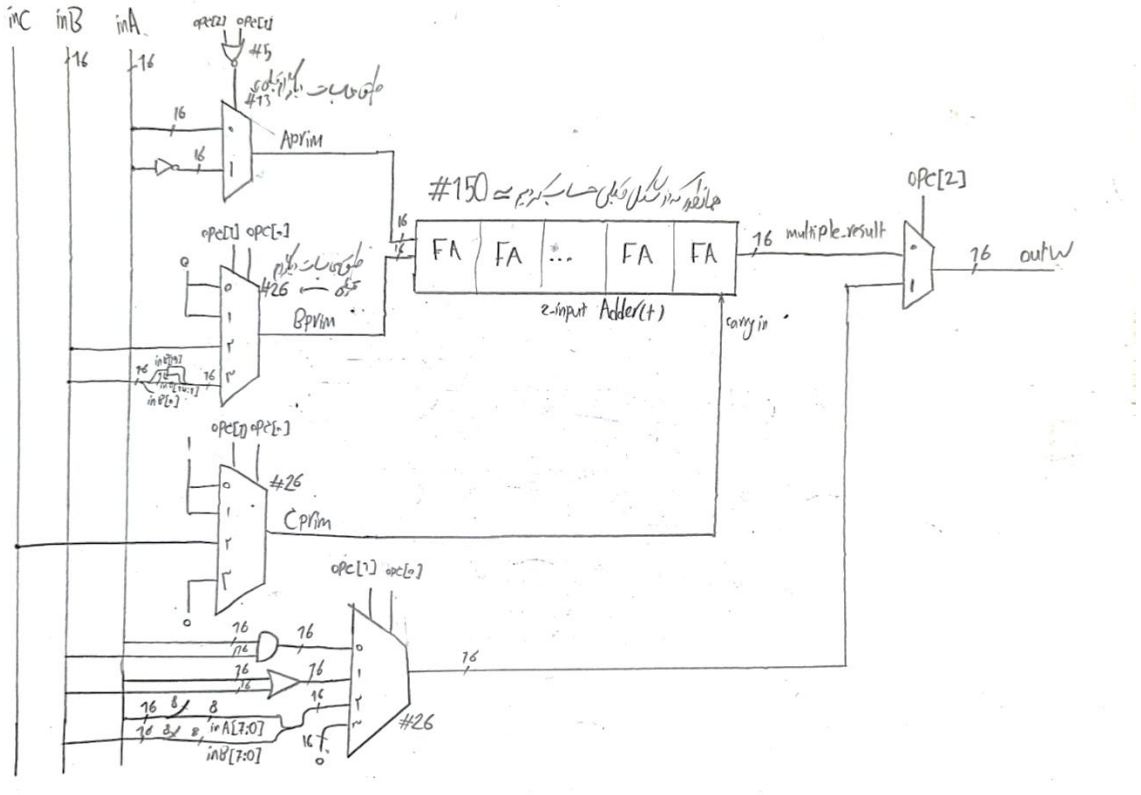
5- نکته ی دیگر در انجام عملیات *0.5 است در عملیات شماره3 است که اولا طبق آزمایشی که کردم، کلا وریلاگ وقتی یک طرف ضربمان عدد ثابت است از multiplierاستفاده نمیکند کلا، پس لازم نیست نگرانش باشیم، و فقط چند حالت داریم برای نوشتن آن ، یکی علامت *0.5 است و یکی /2 است و یکی هم

concatenation است که گویا در پشت پرده ی همگی آنها همان عمل concatenation انجام می شود و تفاوت چندانی نیست

6-نهایتا کدی که فایل سنتز شده اش را گذاشتم، کدی است که هم از لحاظ استاندارد ها بهترین حالت باشد و هم گیت های کمی مصرف کند و اینگونه است:

```
=== structural_ALU ===

   Number of wires:                847
   Number of wire bits:            894
   Number of public wires:           7
   Number of public wire bits:      54
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                512
     NAND                          139
     NOR                           276
     NOT                            97
```

شکل تقریبی و حدسی دیاگرام structural:



*برای فلگ های neg,zer دقیقا همانند دیاگرام قبلی عمل میکنیم.

*با استفاده از MUX به شکلی که مشخص است، از دوتا incrementer و دوتاadder به یک adder رسیدیم که قاعدتا از لحاظ دیلی در هر عملیات شاید خیلی به صرفه نباشد اما از لحاظ تعداد گیت ها بهتر از حالت قبلی است همانطور که در yosys دیدیم.

timing

محاسبه ی دستی worst_case delay :

برای w بدترین مسیر ممکن گذشتن از Adder و یک mux با 4 ورودی و یک mux با 2 ورودی است که دیلی آن برابر است با 150+ 13 + 26 = 189ns

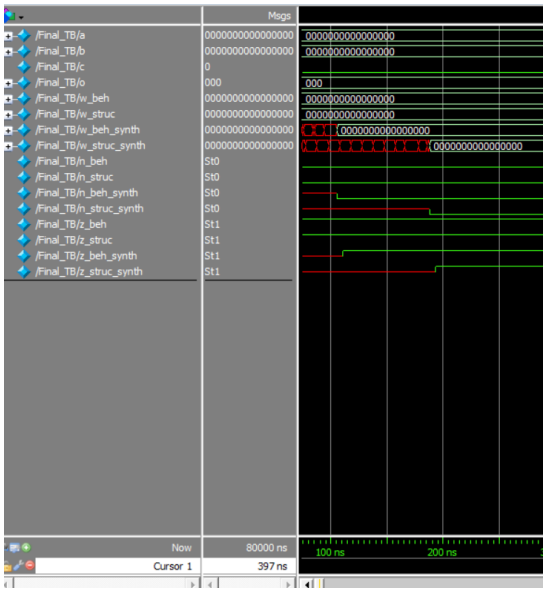
برای neg هم چون کاری بجز wiring نداریم همان 189 است

برای zer هم چون ساختار حدسی مان دقیقا مثل مدار قبلی است همان 189 + 23 = 212ns است

اما بدیهتا در ساختار behavioral ابزار سنتز دستش بسیار باز تر بود در فشرده سازی مدار ها و استفاده از مدار های کوچکتر ، در حالی که در اینجا چون از مثلا یک adder چندین استفاده میکنیم دیگر احتمال اینکه ساختاری بهتر از ripple carry استفاده کند کمتر است در نتیجه نهایتا ساختار structural دارای worst case بدتری است اما تعداد گیت های کمتری دارد.

محاسبه ی worst_caseبا wave form:

ثابت کردیم که حالت اولیه همان بدترین حالت است پس داریم:



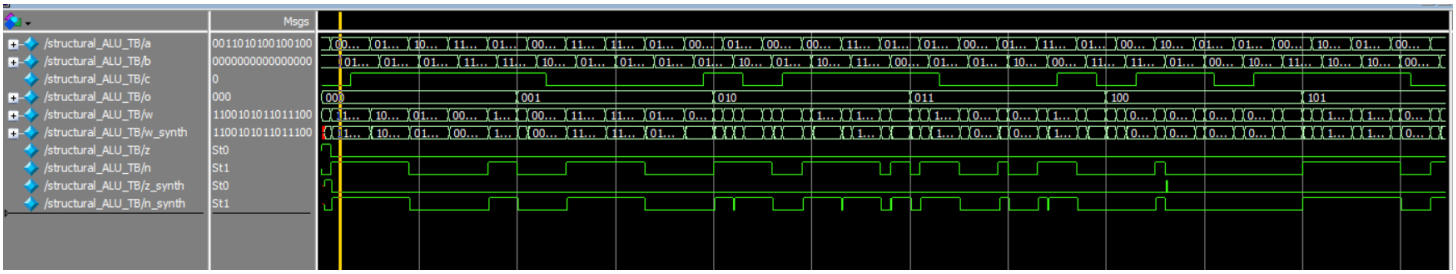
بدترین حالت برای w = 188ns

بدترین حالت برای neg = 188ns

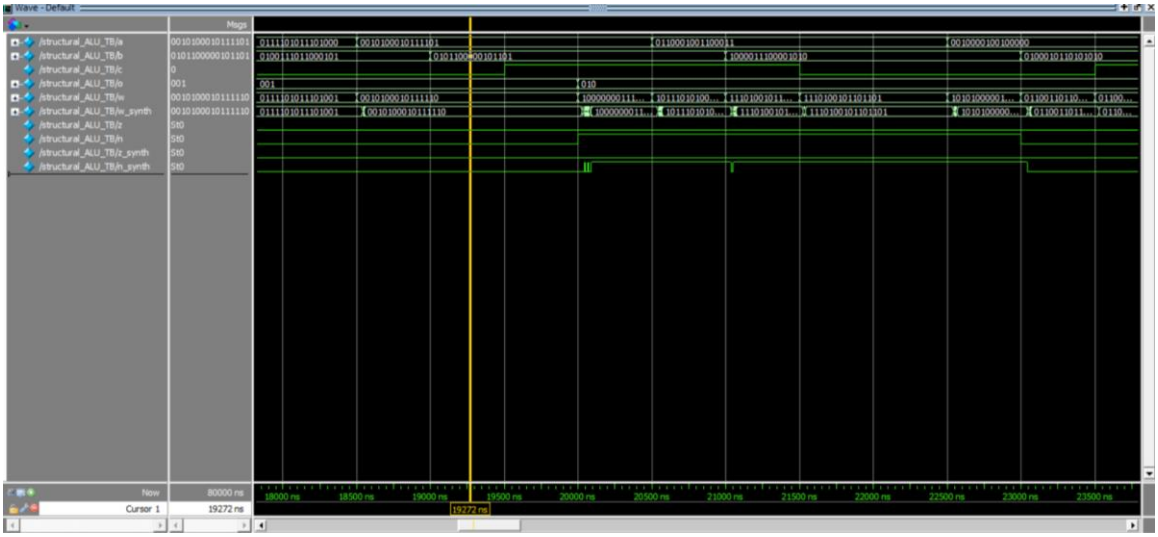
بدترین حالت برای zer = 193ns

Waveformهای structural

همانطور که میبینید جواب های صحیح میدهد به ازای opcodeهای مختلف:

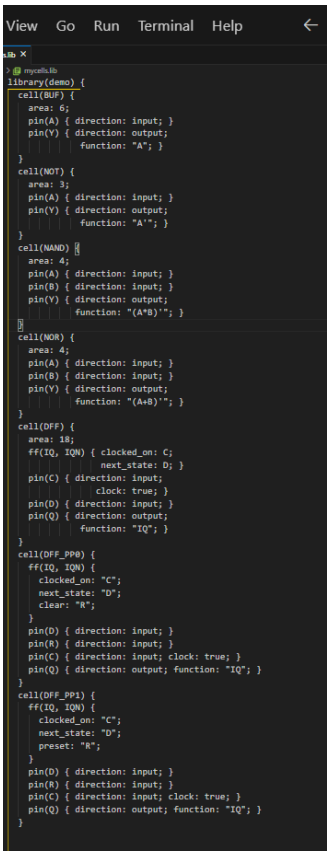


عکس جزیی تر:



از لحاظ سخت افزاری در حالت نخست، احتمالا از گیت ها و مدار هایی همانند چیزی که در دیاگرام کشیدم استفاده میشد اما بعد از سنتز، با توجه به کتابخانه ی ما، فقط از NAND,NOR,NOTبا دو ورودی برای ساخت مدار استفاده شد زیرا فقط همین گیت ها در mycells.libبود (گیت های tri state هم بودند اما بدرد این ساختار نمیخورند)

گیت های استفاده شده توسط yosys برای سنتز

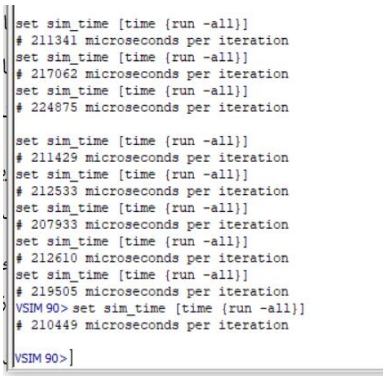


Simulation speed:

دوباره همانند مدار قبلی با استفاده از همان دستورات و بزرگتر کردن تست بنچ برای واضح شدن اختلاف بین دو حالت قبل و بعد سنتز داریم:

postsynthesis:

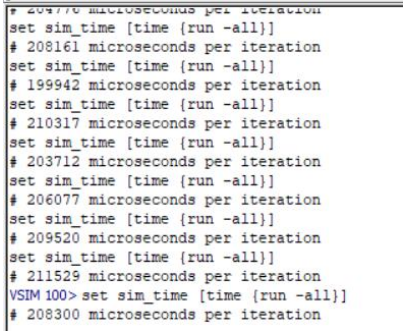
حدود 21000ns



نکته ی جالب اینکه با دقیقا همان تست بنچ، زمان شبیه سازی بسیار کمتر است نسبت به behavioral.

presynthesis:

حدود 21000ns



در نتیجه میزان دیلی شبیه سازی خیلی هم تفاوت ندارد، اما با بزرگ کردن تست بنچ و انجام 50000 بار repeatبجای 5 بار:

Presynthesis:

microseconds per iteration 3371364 #

Postsynthesis:

microseconds per iteration 21665565 #

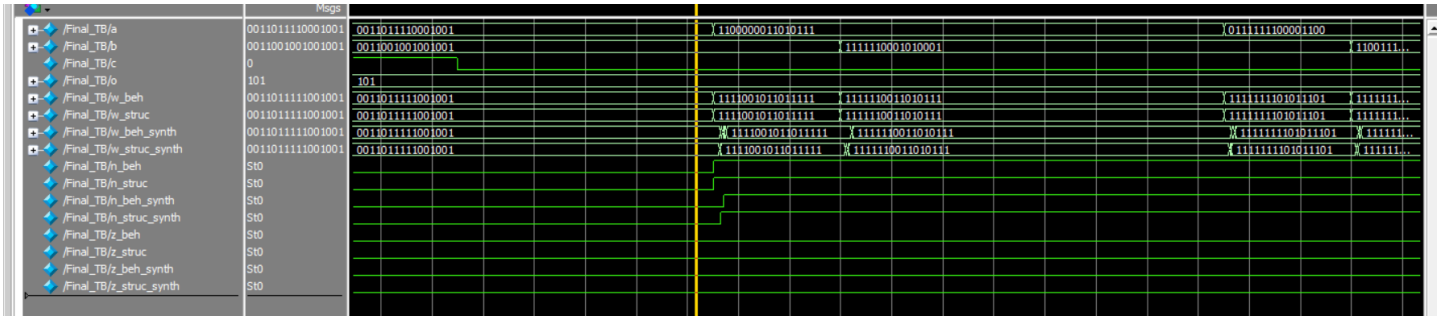
پس به وضوح دیلی شبیه سازی حالت بعد سنتز بیشتر است و دقیقا همانند حالتbehavioral درست است که خط های کد عبارت های ساده تری هستند اما تعدادشان آنقدر زیاد است که نمونه سازی از ماژولی که سنتز شده طولانی تر میشود

مقایسه و Waveformهای نهایی

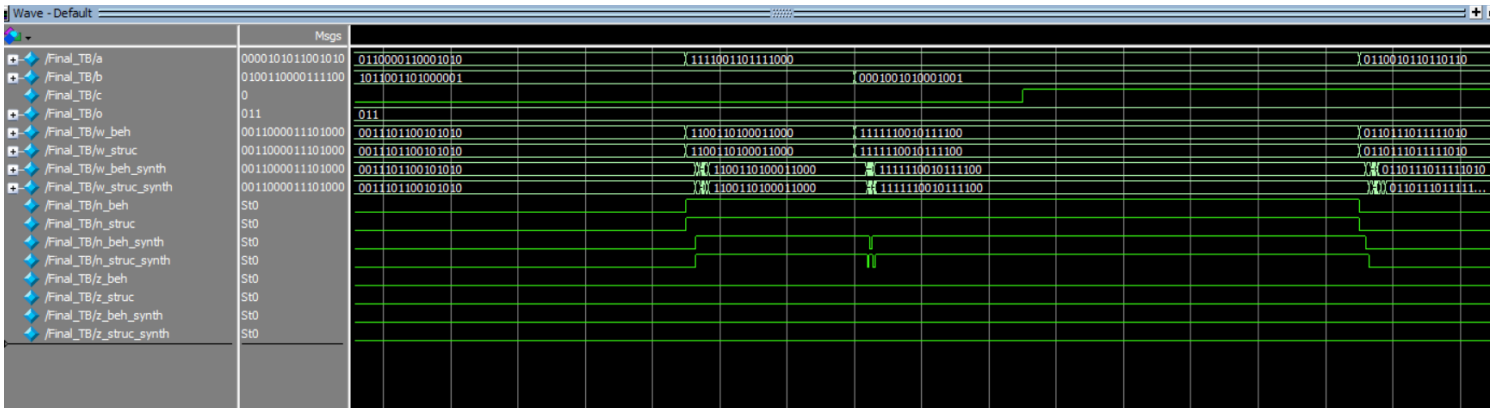
درجهت مقایسه ی هر 4 مدار presynthesis behavioral,post synthesis behavioral, presynthesis structural, portsynthesis structural از یک تست بنچ کلی استفاده میکنیم بطور زیر:

همانطور که مشاهده میکنیم، در حالت های قبل سنتز چون کلا دیلی نداریم جواب کاملا یکسان است و از طرفی در حالت structural چون در برخی عملیات ها بعلت گسسته شدن MUX ها دیلی کمتری داریم اما در برخی دیگر از مسیر ها دیلی برابر یا حتی بیشتر از behavioral داریم:

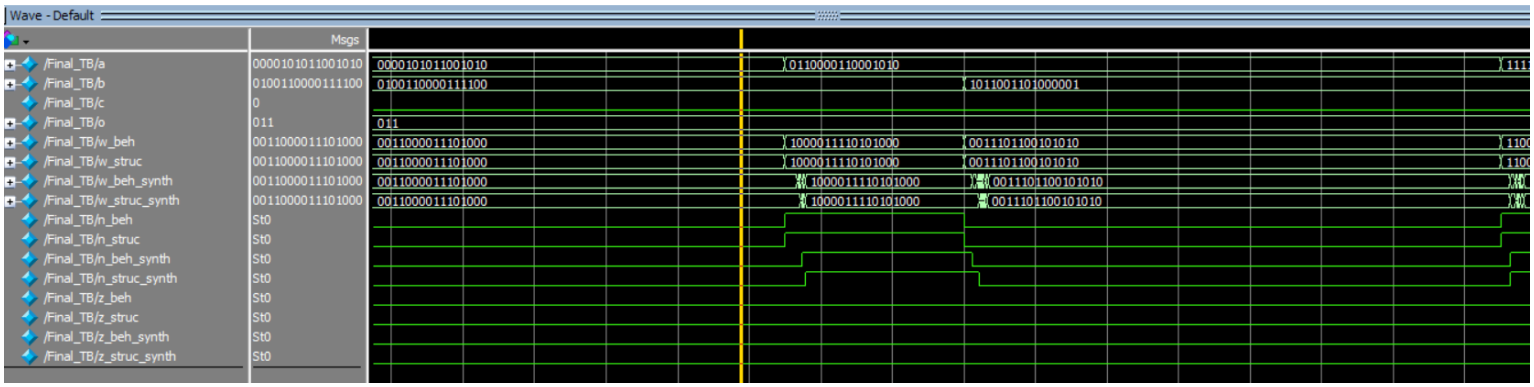
1- وقتی structural دیلی کمتری دارد



2- وقتی دیلی ها مساوی است:



3- وقتی structural دیلی بیشتری دارد:



*قاعدتا در مدار های بعد سنتز بعلت وجود دیلی های مختلف در مسیر های مختلف، ممکن است glitch بزنیم.

نتیجه گیری مقایسه ی دیلی behavioral, structural بعد از سنتز:

از لحاظ worst case دیلی structural بدتر است اما از لحاظ سایر کیس ها ممکن است دیلی ها برابر باشد یا حتی دیلی structural کمتر شود.

عکسی از تست بنچ نهایی شامل هر 4 مدار قبل و بعد سنتز حالت های behavioral و structural: