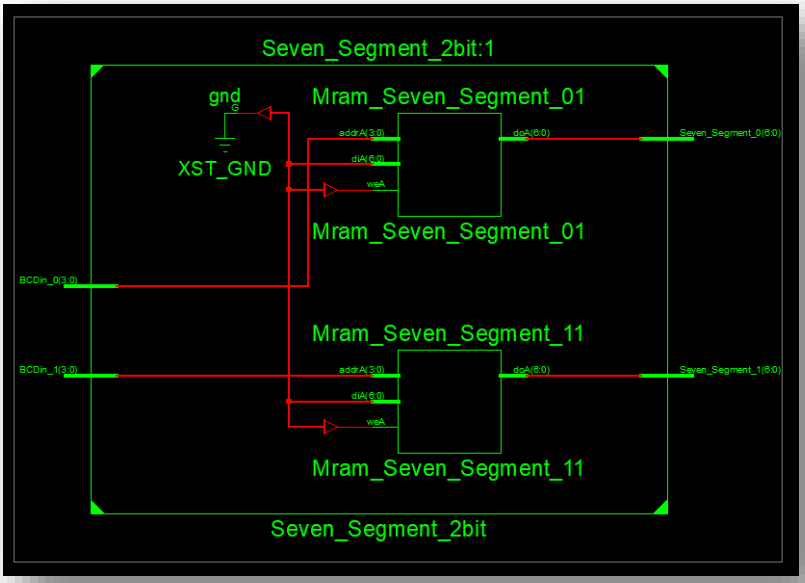
اهداف آزمایش: در این آزمایش به بررسی کد ۷ سیگمنت و هچنین تئوری ۳ مدل جمع کننده معروف میپردازیم.

بررسی 7-Segment:

قصد داریم یک 7-Segment را طراحی کنیم. این 7-Segment فی الواقع دو 7-Segment کنار یکدیگر است که قابلیت نمایش اعداد از ۰ تا ۹۹ را دارد. در ادامه کد آن و تست بنچ آن را شرح خواهیم داد.



مرحله ۱– تعریف پورت های ماژول

```
entity Seven_Segment_2bit is
Port ( BCDin_1 : STD_LOGIC_VECTOR (3 downto 0);
    BCDin_0 : STD_LOGIC_VECTOR (3 downto 0);
    Seven_Segment_1 : out STD_LOGIC_VECTOR (6 downto 0);
    Seven_Segment_0 : out STD_LOGIC_VECTOR (6 downto 0));

end Seven_Segment_2bit;
```

ابتدا امر پورت های ماژول را تعریف میکنیم. دو آرایه (وکتور) ورودی BCDin_0 و BCDin_1 (هر کدام ۴ بیت) باینری عددی است که میخواهیم نمایش دهیم. (در 7-Segment اول یا دوم) و سپس خروجی مربوط به هرکدام نیز ایجاد شده است. هر 7-Segment از هفت عدد LED تشکیل میشود که در اینجا به صورت یک Vector تعریف اش کرده ایم.

مرحله ۲- assign کردن:

```
begin

    process(BCDin_1)
    begin

        case BCDin_1 is
            when "0000" =>
            Seven_Segment_1 <= "0000001"; --0
            when "0001" =>
            Seven_Segment_1 <= "1001111"; --1
            when "0010" =>
            Seven_Segment_1 <= "0010010"; --2
            when "0011" =>
            Seven_Segment_1 <= "0000110"; --3
            when "0100" =>
            Seven_Segment_1 <= "1001100"; --4
            when "0101" =>
            Seven_Segment_1 <= "0100100"; --5
            when "0110" =>
            Seven_Segment_1 <= "0100000"; --6
            when "0111" =>
            Seven_Segment_1 <= "0001111"; --7
            when "1000" =>
            Seven_Segment_1 <= "0000000"; --8
            when "1001" =>
            Seven_Segment_1 <= "0000100"; --9
            when others =>
            Seven_Segment_1 <= "1111111"; --null
            end case;

    end process;
```

در این مرحله بایستی هر حالت 7-Segment را به یکی از اعداد باینری ۰ تا ۹ معادل سازی کنیم. برای این کار از سینتکس Case کمک میگیریم.

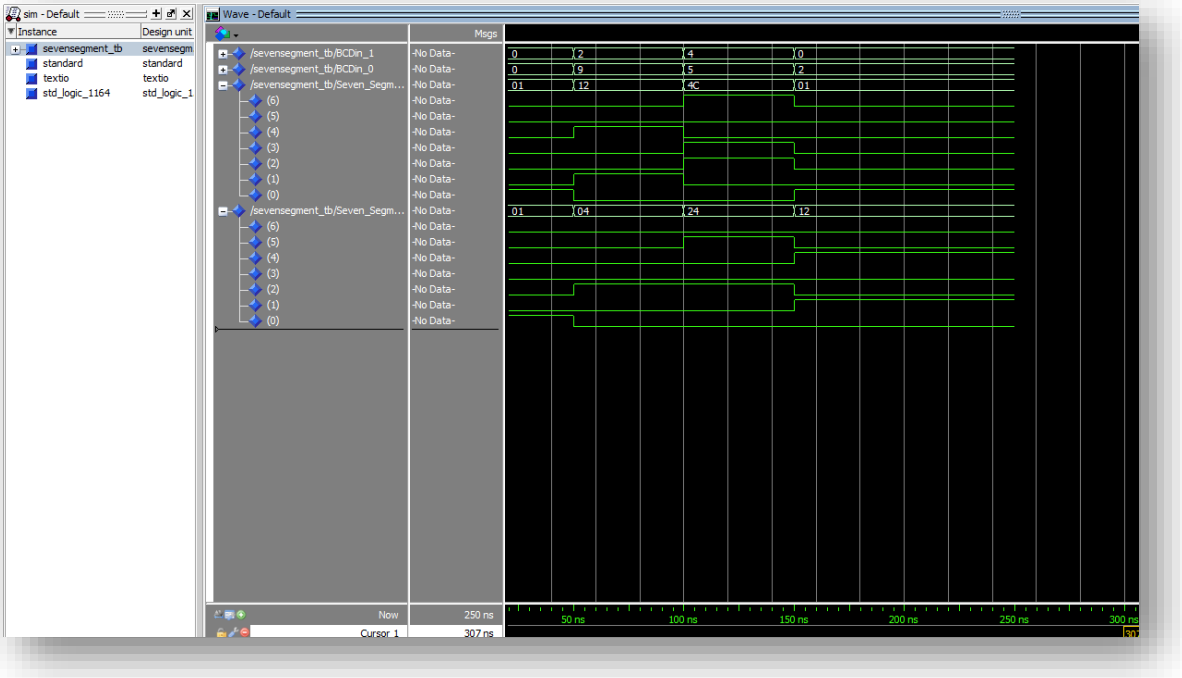
همینکار را بار دیگر برای 7-Segment دیگر انجام میدهیم.

حال نوبت میرسد به Test Bench:

```
-- Stimulus process
stim_proc: process
begin

   wait for 50 ns;
     BCDin_1 <= "0010";
     BCDin_0 <= "1001";
     assert Seven_Segment_1 /= "0010010" or Seven_Segment_0 /= "0000100" report "error in showing 29" severity warning;


     wait for 50 ns;
     BCDin_1 <= "0100";
     BCDin_0 <= "0101";
     assert Seven_Segment_1 /= "1001100" or Seven_Segment_0 /= "0100100" report "error in showing 45" severity warning;

     wait for 50 ns;
     BCDin_1 <= "0000";
     BCDin_0 <= "0010";
     assert Seven_Segment_1 /= "0000001" or Seven_Segment_0 /= "0010010" report "error in showing 02" severity warning;


   wait;
end process;
```

همانند آنچه در گزارش کار های قبل مشاهده کردید، سیگنال های Test Bench را مپ میکنیم به کامپوننت اصلی و در نهایت با استفاده از Process و مقادیری که صحت آن را با assert میسنجیم، نوشتن تست را به پایان میرسانیم.

خروجی زیر حاصل خواهد شد:



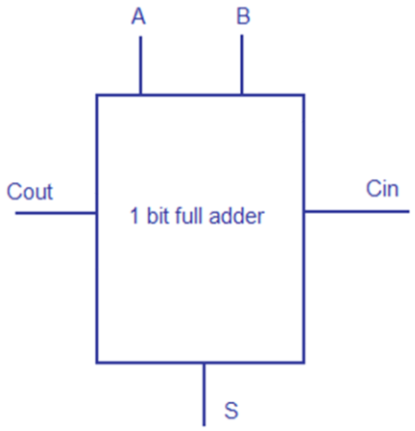
بررسی RA:

Ripple Adder ساده ترین فرم جمع کننده است که قاعدتا به ذهن انسان خطور میکند. در این مدل ما چندین Full Adder را پشت سر هم قرار میدهیم و Carry Out هر کدام را به Full Adder بعدی انتقال میدهیم. برای اینکه پارسی را هم پاس بداریم زین پس به جای Carry از واژه معادل آن یعنی نقلی استفاده میکنیم.

یادمون نره که Full Adder دو تا بیت اصلی و یک نقلی رو میگیره و با هم جمع میکنه و خروجی میده این خروجی از دو بخش SUM و نقلی خروجی (Carry Out) تشکیل شده.

در تصویر زیر میتوانید ساختار کلی Full Adder و Table آن را مشاهده کنید:

Inputs			Outputs	
A	B	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



اما این مدل Adder یک مشکلی دارد، آن مشکل چیست؟ خب همانطور که از روی شماتیک پیداست برای اینکه هر FA بتواند کار خودش را انجام دهد نیازمند آن است که Carry Out (یا همون نقلی خروجی) را داشته باشد و این یعنی FA آخر باید منتظر بماند تا بقیه (عقبی ها) کارشان را انجام دهند و او سپس بتواند محاسبه اش را انجام دهد. این یعنی تاخیر و ما در دنیای امروز از تاخیر خوشمان نمیاید و میخواهیم همه چیز به سرعت و در efficient ترین حالت خود صورت پذیرد. این شد که دو مدل بعدی توسط دانشمندان معرفی شد:

بررسی Carry Look-ahead Adder:

خب در بخش قبل با ایراد RA آشنا شدیم اینبار دانشمندان اومدن و با یک سخت افزار پیچیده تر و یکسری Boolean Algebra کاری کردن که نیاز نباشه برای هر FA منتظر قبلیش بمونیم. چیکار؟ خب باید قبلش با دو مفهوم آشنا بشیم:

Pi (carry propagate) = Ai XOR Bi

Gi (carry generate) = Ai AND Bi

با استفاده از این دو متغیر که میان از دو ورودی اصلی FA استفاده میکنند، میایم و نشون میدیم چطور میشه SUM و Carry رو بدست آورد:

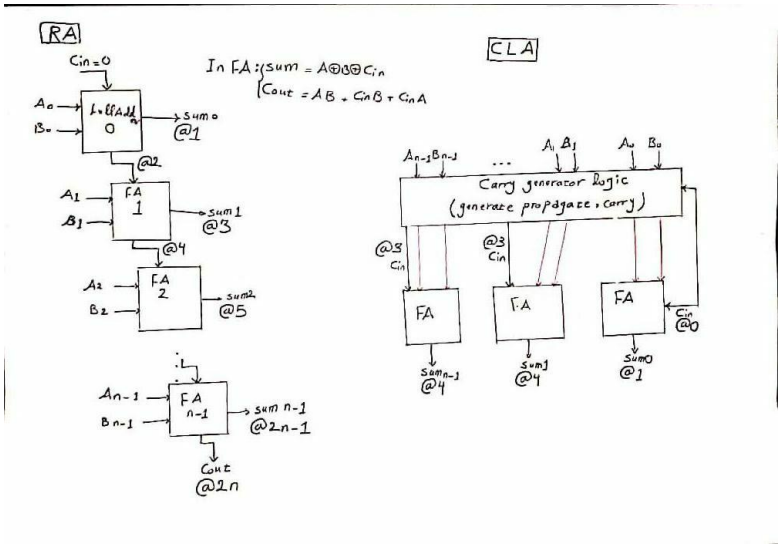
Si = Pi XOR Ci

Ci+1 = Gi OR Pi AND Ci

بنابراین بدون اینکه نیاز باشه صبر کنیم میتوانیم مقدار مورد نظر را بدست بیاریم.

مشکل این روش این هست که هزینه ساختش بسیار بالاست شما فرض کنید باید برای هر ورودی FA کلی XOR قرار بدیم تا بتونیم محاسبات رو انجام بدیم.

خوبیش هم اینکه اون داستان صبر کردن دیگه وجود نداره!

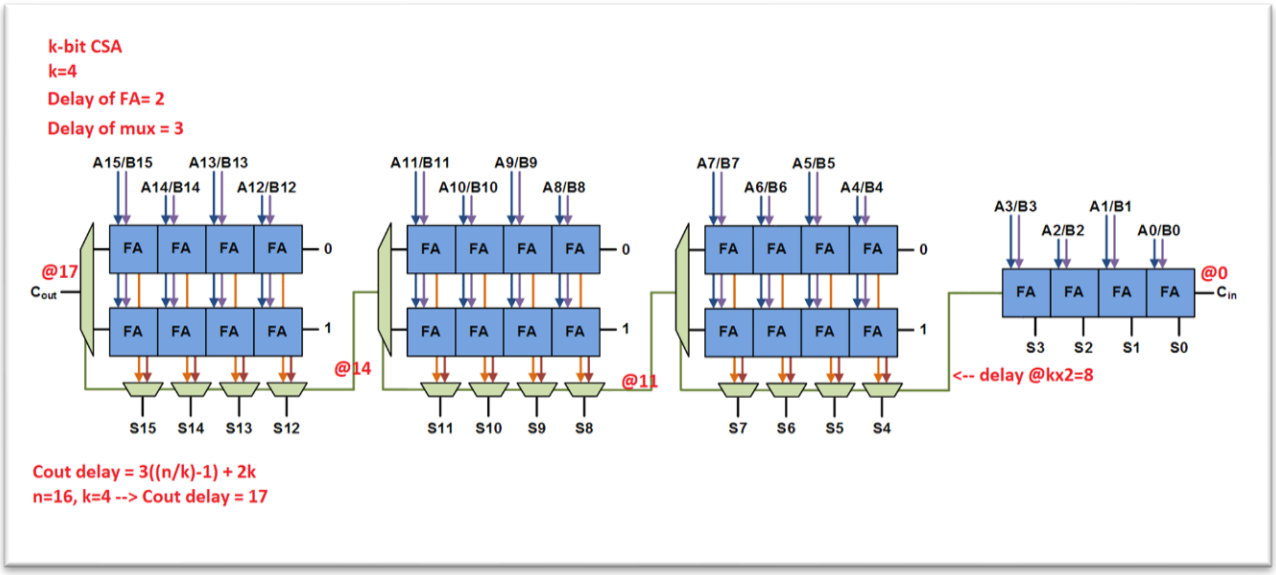


تصویر فوق شماتیک دو جمع کننده مطرح شده است. (RA و CLA)

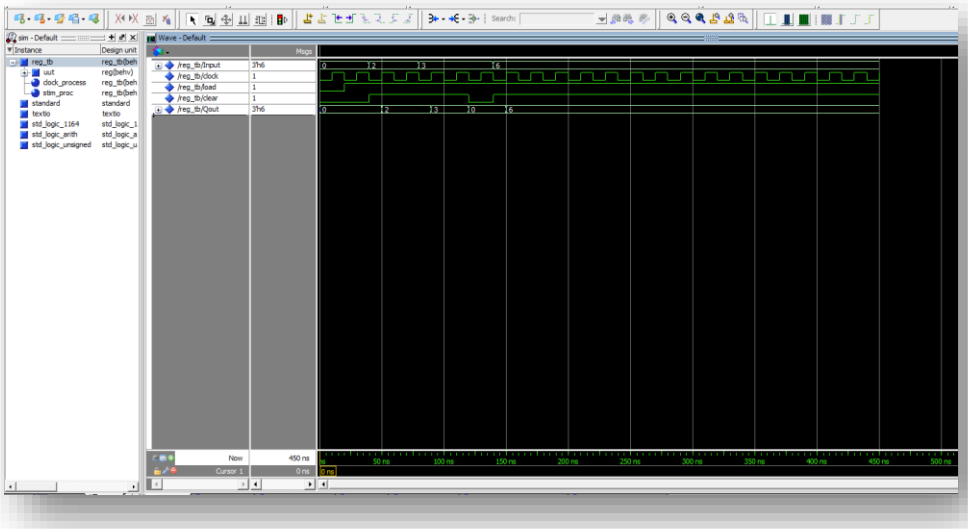
بررسی Carry Select Adder:

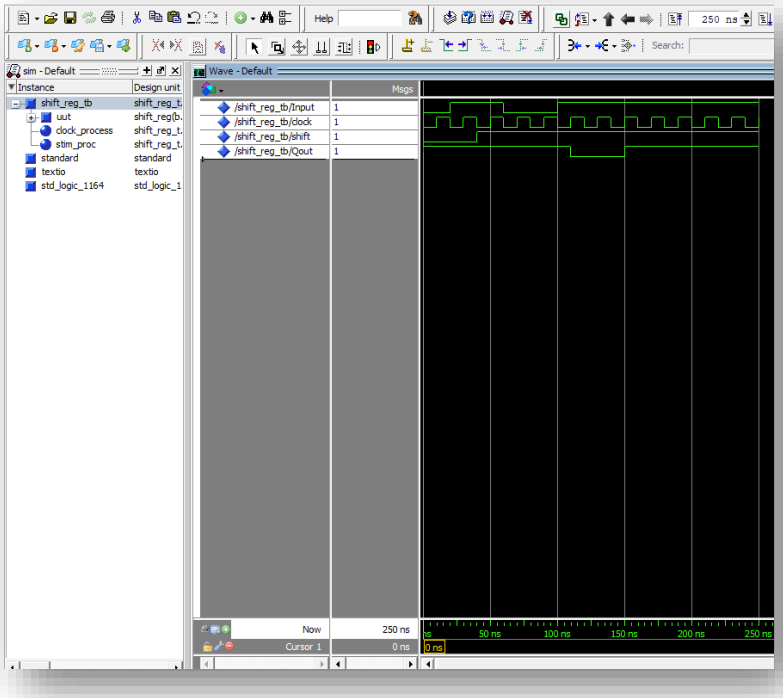
این یکی خیلی جالبه، چرا؟ اومدن گفتند آقا به جای این Pi و Gi ما میتونیم بیایم و RA رو با MUX ترکیب کنیم و محاسبات رو انجام بدیم. چطوری؟ اومدن گفتند هر FA بیاد دو تا جمع انجام بده یکبار به ازای Carry in = 0 و یکبار هم به ازای Carry in = 1 (قاعدتا میشه دو تا FA) محاسبه اش رو انجام بده و در هر Level یک MUX داشته باشیم که بیاد Select رو انجام بده. اینجا باز هم برای خروجی باید منتظر بایستیم که Carry واقعی برسه به دست FA اما تفاوتش اینکه زمانی که باید صرف محاسبه اش بشه از قبل به صورت موازی انجام شده و نیازی نیست صبر کنیم. فقط میاد اون Carry جدید به عنوان خط Select و یکی از اون خروجی های FA به ازای نقلی ۰ یا ۱ رو انتخاب میکنه و تمام!!!

در شکل زیر میتوانید شماتیک آن را مشاهده کنید:



پ.ن: دو تست بنچ برای Register و Shift Register نیز طراحی شد که در شکل زیر میتوانید شکل موج آنها را مشاهده کنید:





تمامی شکل ها و فایل ها در پیوست موجود است

موفق باشید!