به نام خدا

الگوریتم ضرب کننده *Booth*
و پیاده سازی آن با زبان *VHDL*

گردآوری

سید ابوالفضل احمد میری، محمد نحوی

کلمات کلیدی

الگوریتم *Booth*، ضرب کننده، *VHDL*، *Verilog*

چکیده

این مقاله به توضیح چند روش برای ضرب اعداد باینری پرداخته و سپس الگوریتم *Booth* را به عنوان نمونه با زبانهای *VHDL* و *Verilog* شبیه سازی نموده است.

۱. مقدمه

در این گزارش به چگونگی عملکرد یک ضرب کننده ۳۲*۳۲ بیت و دو تا از روش های سریع ضرب می پردازیم و اینکه برای پیاده سازی آن چه مراحلی را باید طی کرد .

در فصل اول از این گزارش به موضوع معرفی ضرب کننده های سریع پرداخته ایم که در این گزارش فقط به دو نوع ***booth , Wallace tree*** و ترکیب آنها و نوع عملکرد هرکدام از آنها و شماتیک عملکردی هر نوع با ذکر مثال هایی پرداخته ایم .

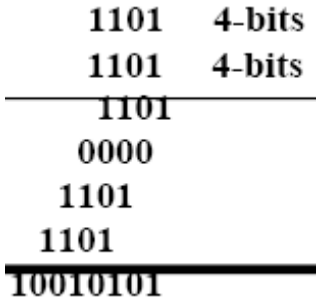
در فصل دوم این گزارش توضیح خواهیم داد که برای پیاده سازی سخت افزاری می توانیم از زبان برنامه نویسی ***VHDL*** استفاده کنیم و اینکه خصوصیات آن چگونه است و می توان آن را قبل از پیاده سازی سخت افزاری شبیه سازی کرد و درستی عملکرد برنامه را بررسی کرد . در قسمت های بعد این فصل به چگونگی فلوچارت برنامه نویسی ضرب ***booth*** که ما دراین گزارش انجام داده ایم می پردازیم ودر آخر برنامه را نوشته وتوضیح مختصری داده ایم و کمی در مورد نرم افزار شبیه سازی و پیاده سازی که در اینجا استفاده کرده ایم خواهیم پرداخت .

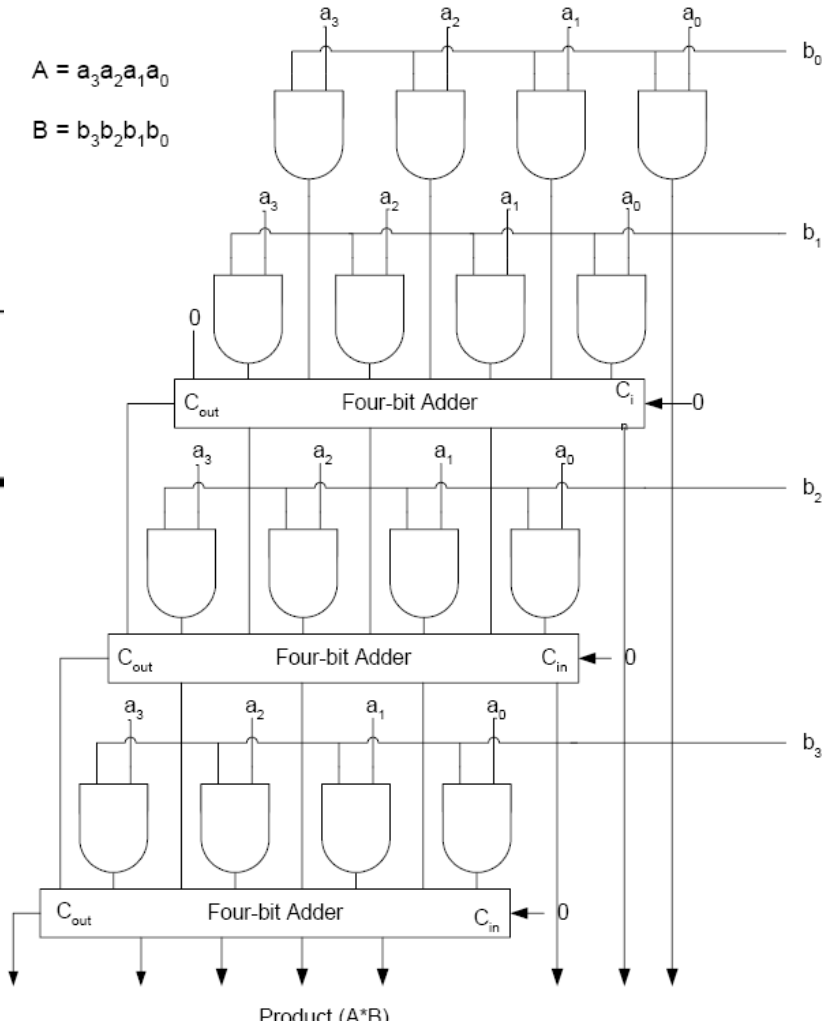
۲. ضرب کننده های سریع

مکانیزم های زیادی برای ضرب کردن اعداد باینری وجود دارد که اساس آن ها تقریبا ثابت است و متشکل از یک سری ضرب های جزئی و جمع می باشد . اما شیوه هایی وجود دارد که سرعت و تعداد این ضرب ها و جمع ها را کاهش داده و موجب تسریع ضرب باینری می شود که در این بخش به آشنایی مختصری از دو روش ضرب سریع خواهیم پرداخت .

در یکی از این روش ها که ***booth*** نام دارد تعداد ضرب های جزئی را کاهش می دهد و باعث تسریع ضرب می شود و در روش دیگر که در اینجا توضیح داده شده به نام ***Wallace tree*** تعداد جمع ها را کاهش می دهد .

در اینجا برای آشنایی اولیه و یادآوری یک ضرب کننده معمولی را با یک مثال شرح می دهیم و در سه قسمت بعد به ضرب های سریع خواهیم پرداخت .





۲-۱. ضرب کننده های *booth*

همان طور که از بخش قبلی مشاهده کردیم در یک ضرب ***n*n*** بیت به ***n*** ضرب جزئی نیاز بود و بعد جمع کردن آن ضرب های جزئی به روش های مختلف که به آن اشاره شد . ولی در روش ***booth*** که یکی از روش های ضرب سریع محسوب می شود ما ضرب های جزئی کمتری خواهیم داشت که می توان این ضرب های جزئی را به روش های مختلف با هم جمع کرده این روش هرچند روند ضرب را سریعتر می کند ولی این آن را پیچیده تر خواهد کرد .

در این روش ما یکی از اعدادی که می خواهیم در هم ضرب کنیم را با انواع کدهایی که در زیر خواهیم گفت کدبندی کرده و سپس در عدد دیگر ضرب خواهیم کرد . جدول های کدبندی ***booth*** در زیر آورده شده است.

X_{i+1}	X	X_{i-1}	$Z_{i/2}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

جدول(۱) کدبندی *booth_radix4*

برای فهم بهتر این نوع ضرب کردن آن را با مثالی دنبال می کنیم.

Decimal

$$\begin{array}{rr} & 169 \\ \times & 107 \\ \hline & 1183 \\ & 000 \\ & 169 \\ \hline & 18083 \end{array}$$

X=169 , Y=107

ابتدا به ***Y*** از سمت راست یک صفر اضافه کرده وسپس آن را به صورت زیـر مطـابق جـدول(۱) کدبنـدی خواهیم کرد وحال ضرب های جزئی را بدست می آوریم.

$$\begin{array}{cccccccccc} & & (4) & & & & (2) & & & \\ \hline 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & .0 & \\ & & \hline & & (3) & & \hline & & (1) & \end{array}$$

(1)	110	$-X$	1	-169	-169
(2)	101	$-X$	4	-169×4	-676
(3)	101	$-X$	16	-169×16	-2704
(4)	011	$+2X$	64	$2 \times 169 \times 64$	21632
				total action	18083

روش کدبندی به این صورت است عددی را که به آن صفر اضافه شده را از سمت راست سـه تـا سـه تـا مطابق بالا جدا کرده و طبق جدول(۱) آن را می نوسیم. اگر ***0*** بود آن ضرب در ***X*** برابر ***0*** می شود. اگر ۱ بود خود ***X*** را قرار می دهیم . اگر ۲ بود عدد باینری را یکی به راست شیفت می دهـیم و اگـر ۱- بود عدد باینری را ***NOT*** کرده و باضافه ی ۱ می کنیم و اگر ۱- بود عدد باینری را ***NOT*** کرده و باضافه ی ۱ می کنیم و یک شیفت به راست می دهیم.

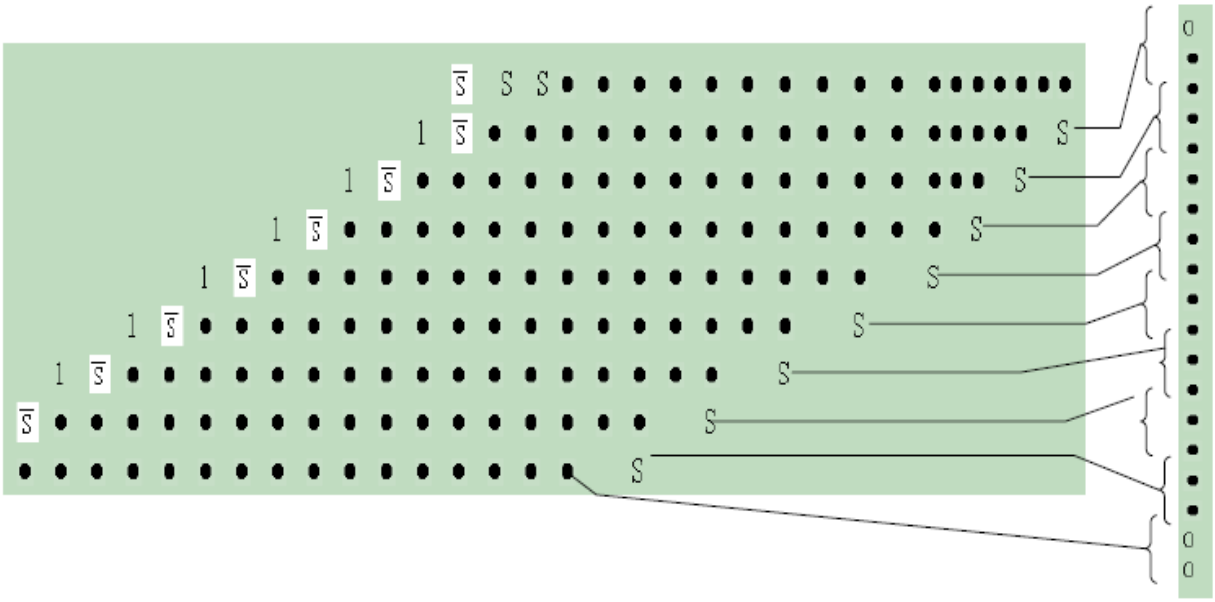
$$
\begin{array}{rll}
 & 10101001 & \\
\times & 01101011 & \\
\hline
 & 1111111101010111 & -X \\
 & 11111101010111 & -X \times 4 \\
 & 111101010111 & -X \times 16 \\
 & 0101010010 & +2X \times 64 \\
\hline
 & 0100011010100011 &
\end{array}
$$

این نکته را نیز باید در نظر گرفت که در این روش باید بیت علامت را گسترش داد. به دو مثال دیگر توجه کنید.

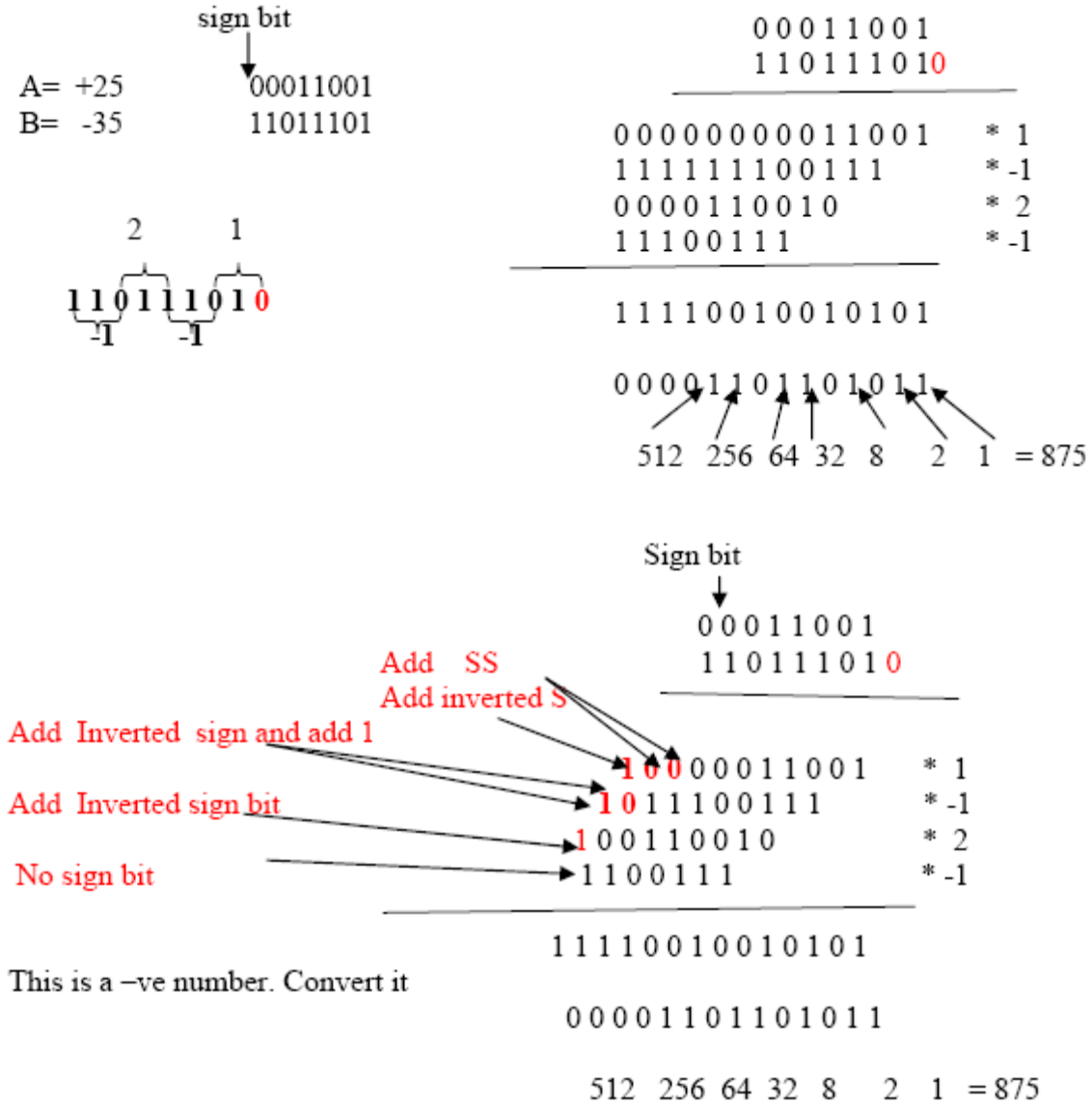
```
      000011      (+3)
   ×  011101 0    (+29)
       +2 -1 +1
-------------------
  000000000011
   1111111101
    00000110
-------------------
1 ◄ 000001010111    (+87)
```

```
      111101      (-3)
   ×  100011 0    (-29)
       -2 +1 -1
-------------------
  000000000011
   1111111101
    00000110
-------------------
    000001010111    (+87)
```

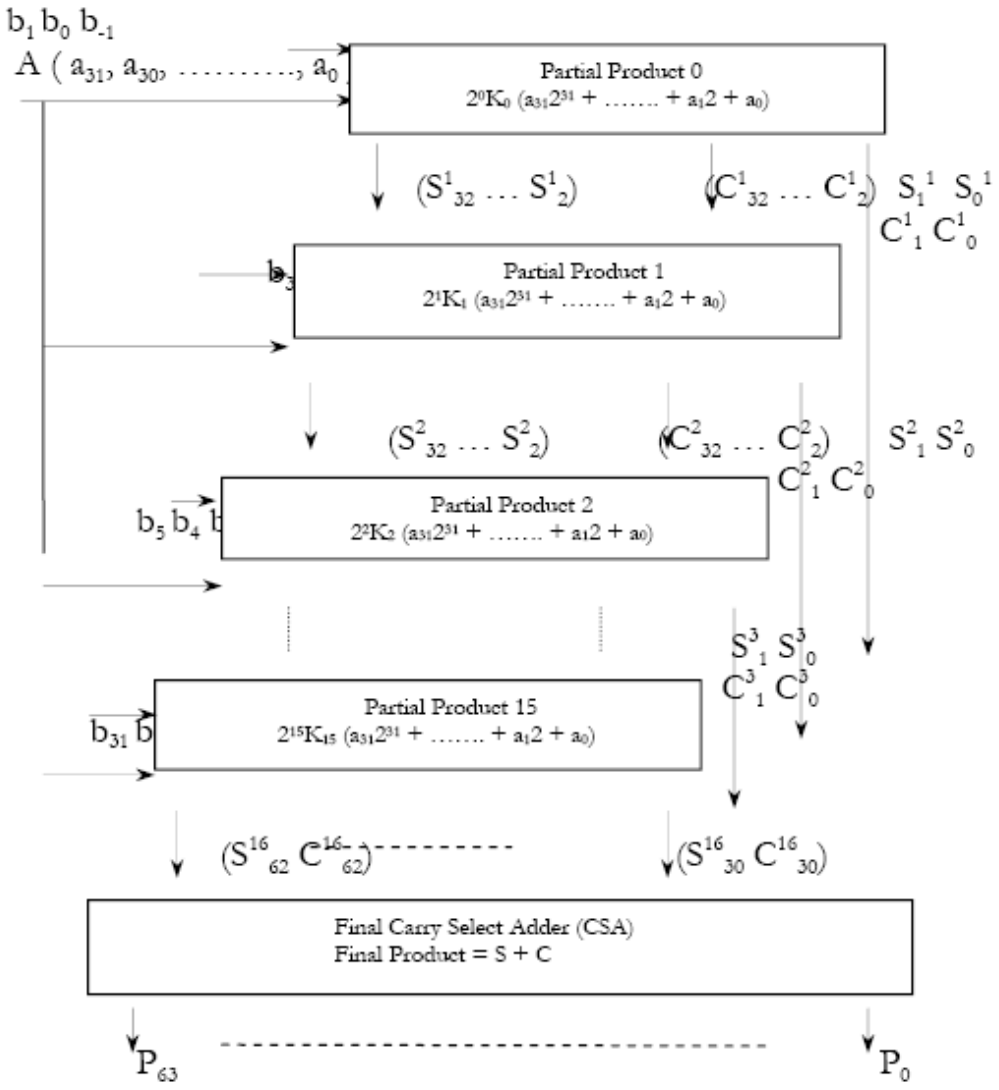
برای کاهش محاسبات می توان به جای گسترش بیت علامت از روش زیر استفاده کنیم که S نشان دهنده ی بیت علامت است . و اگر از متمم ۲ استفاده شود دیگر از بیت های علامت سمت راست استفاده نمی شود و این بیت های علامت مختص ه متمم ۱ است .



به مثال های زیر توجه کنید تا بهتر با این روش آشنا شوید.



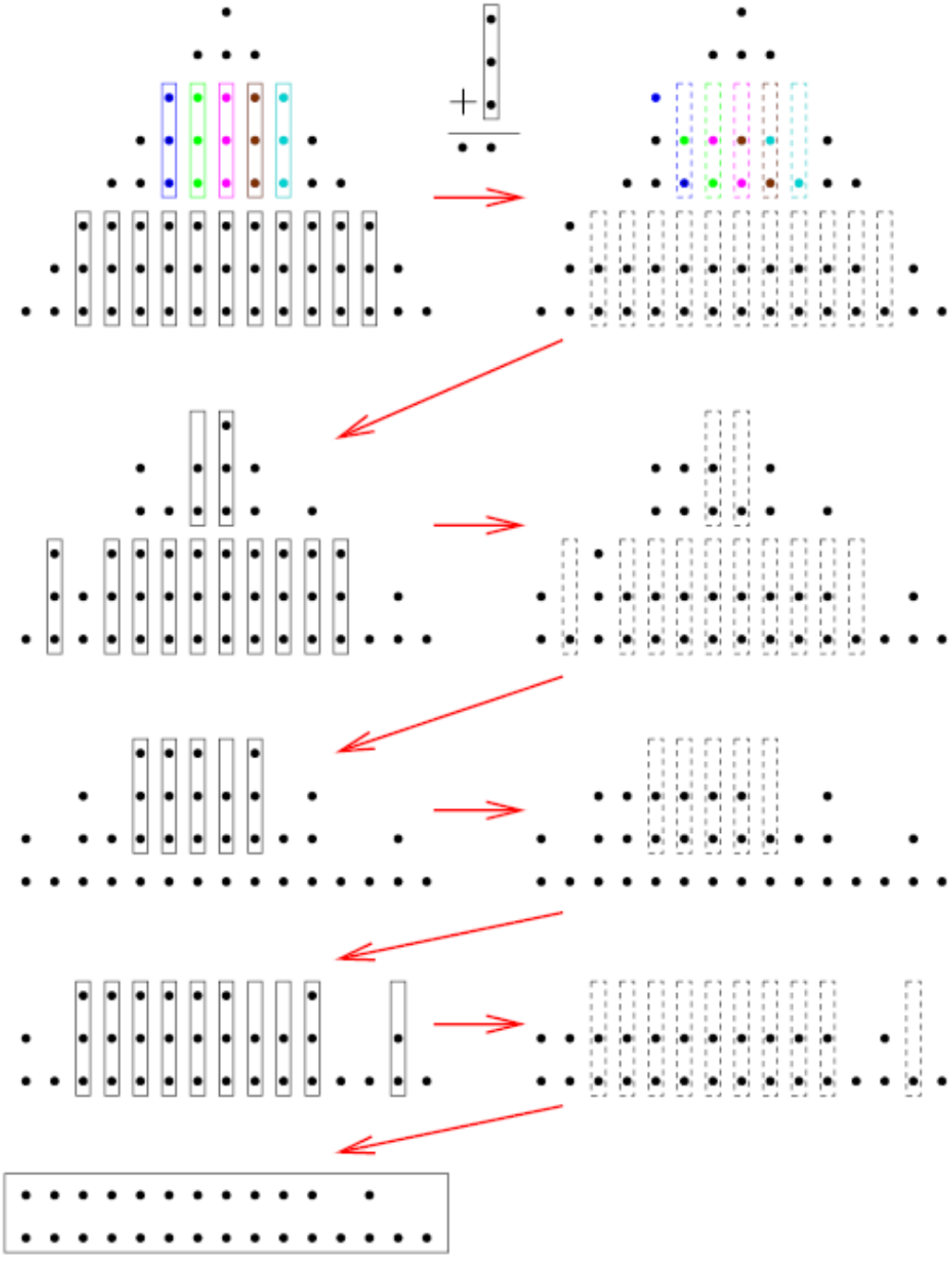
فلوچارت این ضرب برای یک ضرب کننده ی ۳۲*۳۲ در زیر نمایش داده شده است .



پس در کل می توان نتیجه گرفت که روش *booth* موجب سریعتر شدن ضرب می شود هر چند که مقداری آن را پیچیده می کند .

۲-۲. روش *Wallace tree*

در این روش بر عکس روش قبل در تعداد ضرب های جزئی تغییری ایجاد نمـی کنـد بلکـه بـا شـیوه ای خاص روند جمع کردن ضرب های جزئی را کاهش می دهد که آن را به صورت مختصری در زیر توضیح می دهیم . این کار موجب کاهش تعداد ***full adder*** خواهد شد برای درک محسوس تر به مثال های زیر توجه کنید.



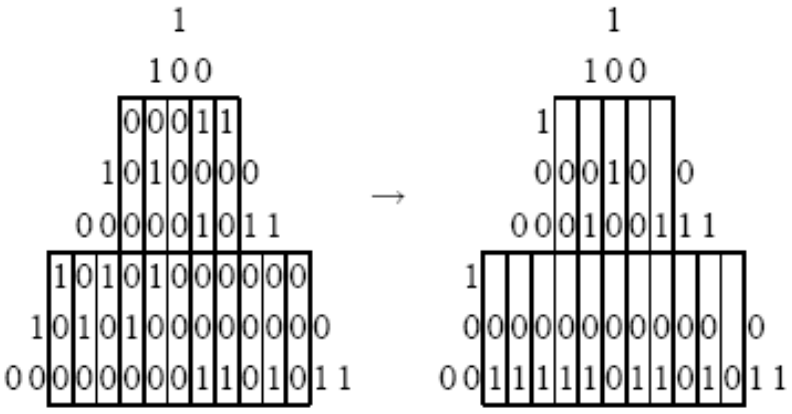
wu

مثال

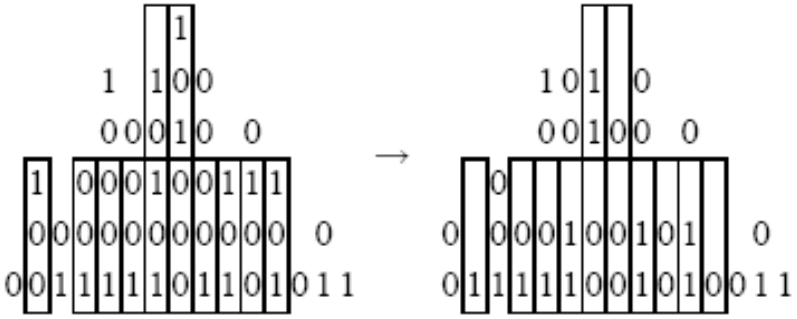
$$\begin{array}{r} 169 \\ \times\ 107 \\ \hline 3232 \\ +14851 \\ \hline 18083 \end{array}$$

$$\begin{array}{r} 10101001 \\ \times\ 01101011 \\ \hline 10101001 \\ 10101001 \\ 00000000 \\ 10101001 \\ 00000000 \\ 10101001 \\ 10101001 \\ 00000000 \\ \hline 0100011010100011 \end{array} \quad \equiv \quad \begin{array}{r} 10101001 \\ \times\ 01101011 \\ \hline 1 \\ 100 \\ 00011 \\ 1010000 \\ 000001011 \\ 10101000000 \\ 1010100000000 \\ 000000001101011 \\ \hline 0100011010100011 \end{array}$$

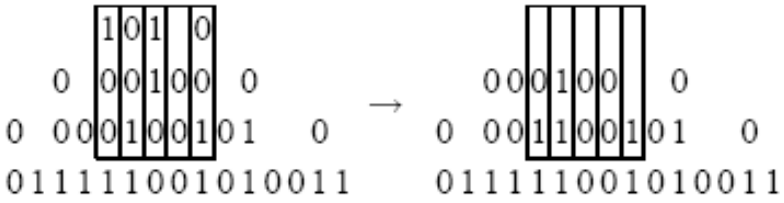
مرحله ۱



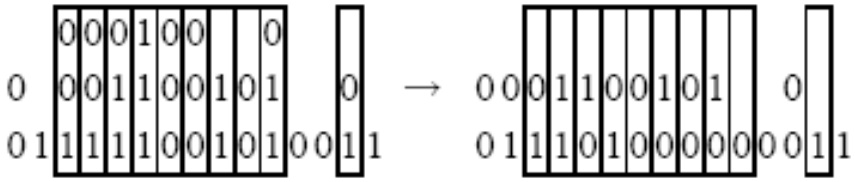
مرحله۲



مرحله۳



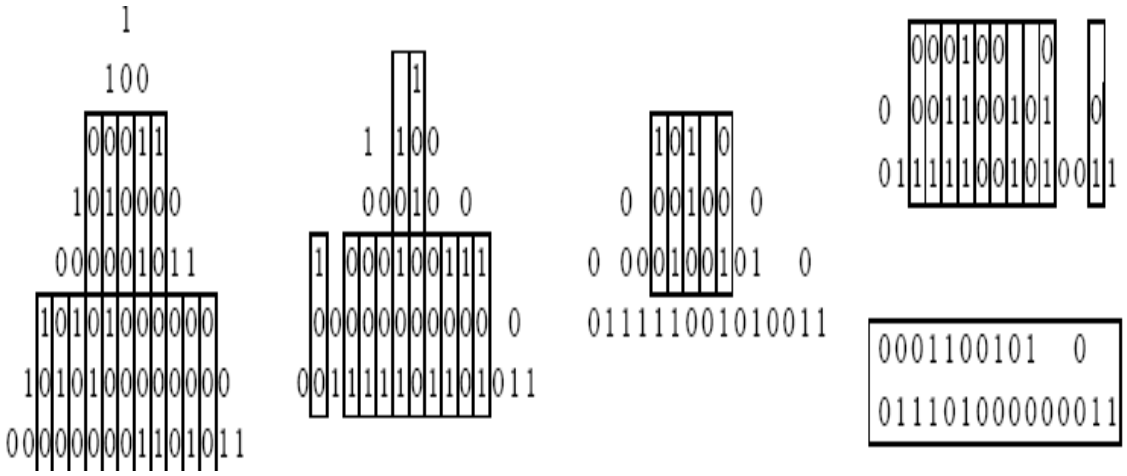
مرحله۴



مرحله۵

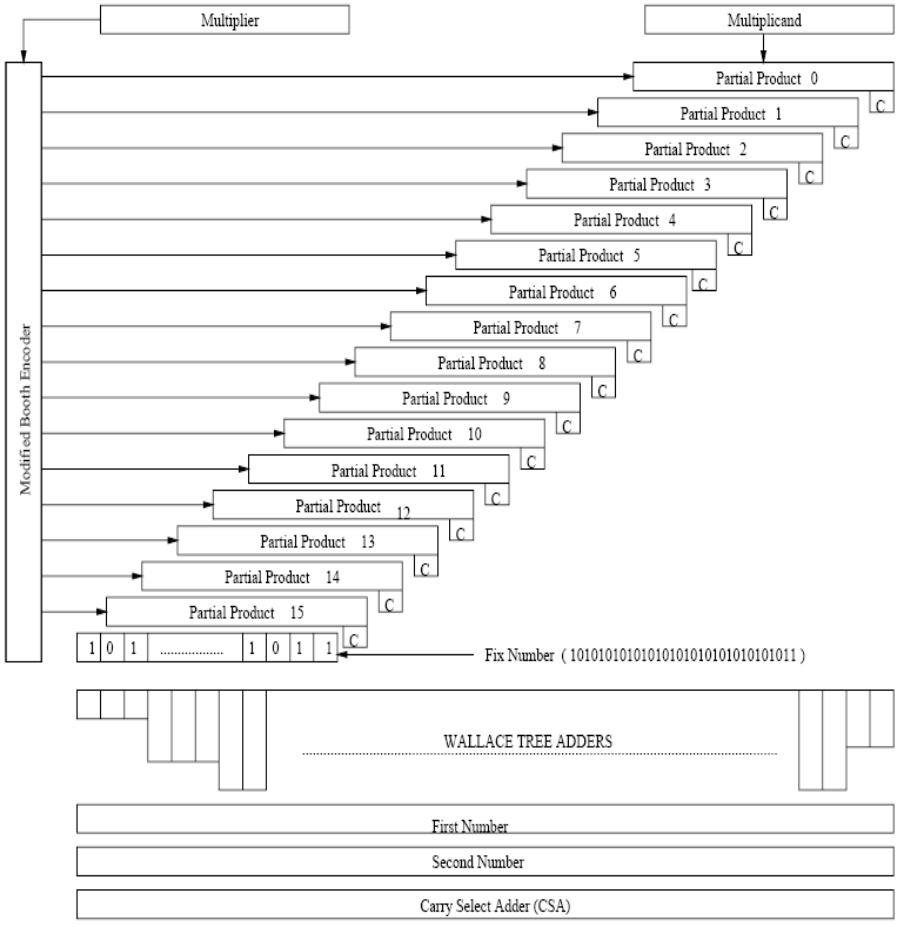
$$
\begin{array}{rr}
10101001 & 169 \\
\times\ 01101011 & \times\ 107 \\
\hline
0001100101\quad 0 & 3232 \\
011101000000011 & +14851 \\
\hline
01000110101000011 & 18083
\end{array}
$$

ودر اینجا هر ۵ مرحله را می توان در یک ردیف مشاهده کرد.



۲-۳. روش *booth* و *Wallace tree*

ما در اینجا با ترکیب این دو روش *booth* و *Wallace tree* به سرعت بیشتری خواهیم رسید . با استفاده از *booth* ضرب های جزئی را کاهش می دهیم و با استفاده از *Wallace tree* جمع کردن آنها را تسریع می بخشیم که در اینجا فقط به نمایش دادن فلوچارت آن اکتفا خواهیم کرد.



۳. برنامه نویسی ضرب کننده ۳۲×۳۲ بیت با VHDL

ما این را فصل برای فهم بهتر چگونگی برنامه نویسی به چهار بخش تقسیم مـی کنـیم در بخـش اول بـه آشنایی مختصری در مورد ***VHDL*** خواهیم پرداخت که اساس آن چگونه است و کلیت برنامه نویسـی بـا آن به چه قرار است . در بخش دوم به توضیح اینکـه جگونـه ایـن ضـرب صـورت مـی گیـرد و کشـیدن شماتیک و توضیح فلوچارت این ضرب کننده می پردازیم . در بخش سوم بـه رونـد نوشـتن برنامـه مـی پردازیم و در قسمت آخر به توصیف مختصری از نرم افزاری که بـرای اجـرا و شـبیه سـازی ایـن برنامـه استفاده شده است خواهیم پرداخت .

۳-۱. VHDL چیست ؟

VHDLاز حـروف اول(***VHSIC Hardware Description Language***) بـه معنـی زبـان توصـیف سـخت افزاری***VHSIC*** گرفته شده است.

VHSIC نیز خود از حروف اول ***Very High Speed Intergrated Circuits*** تشکیل شده است و به پـروژه طراحی مدارهای مجتمع با سرعت بسیار بالا اشاره دارد . این برنامه توسط ***IEEE*** اسـتاندارد سـازی شـد که طی نسخه هایی این استاندارد سازی بهینه شده است .

در زبان های برنامه نویسی روا لی متداول مثل***C*** یا پاسکال معمولا محاسبه یک تابع ریاضی مانند ضـرب ماتریس ها یا انجام عملیاتی روی داده ها مثل مرتب کردن از طریق تعریف یک روال انجـام مـی گیـرد . برنامه ها مجموعه ای از مراحل هستند که نحوه انجام محاسبه یا عملیات روی داده رد بیان مـی کننـد و اجرای یک برنامه به محاسبه یا سازماندهی مجدد داده ها منجر می گردد . در مقابل ***VHDL*** زبانی اسـت برای توصیف سیستم هاب دیجیتالی . شبیه سازها می توانند بدون اینکه سیستم به طور واقعـی سـاخته شود از این توصیف برای شبیه سازی رفتار سیستم اسـتفاده کننـد . همچنـین کامپایلرهـای سـنتز مـی توانند از این توصیف برای ایجاد توصیفی از سخت افزار دیجیتالی به منظور پیاده سازی سیستم اسـتفاده کنند . با اینکه ***VHDL*** به منظور توصیف شبیه سازی سیستم های آنالوگ پدید آمده بود اما بیشتر برای طراحی سیستم های الکترونیکی دیجیتال استفاده می شود.

در این برنامه ما به نحوه محاسبه یک تابع کمتر توجـه داریـم و در مقابـل علاقمنـد هسـتیم کـه رفتـار سیستم های فیزیکی مانند یک مدار دیجیتالی را توصیف کنیم . این توصیف رفتاری حداقل دو هـدف را

در بر دارد که در این جا به آن اشاره می کنیم . هدف اول شبیه سازی مدارهای دیجیتالی است . شبیه سازها از توصیف ***VHDL*** برای انجام شبیه سازی استفاده می کنند تا رفتاری کاملا مشابه به رفتار سیستم فیزیکی را به وجود آورند .کاربرد این شبیه سازی ارزیابی و تایید رفتار مدار دیجیتالی قبل از صرف هزینه ای سنگین برای ساخت آن است . هدف دوم سنتز مدارهای دیجیتالی است . پس از آنالیز توصیف***VHDL*** توسط ابزارهای طراحی یک مدار دیجیتالی حاصل می شود که همان رفتار مربوط به توصیف ***VHDL*** را دارد . این توصیف های مداری به سرعت قابل پردازش و تبدیل به سخت افزارهای سفارشی هستند همچنین می توان آنها را روی واحدهای سخت افزاری قابل برنامه ریزی پیاده نمود . به این ترتیب توصیف های ***VHDL***برای پشتیبانی شبیه سازی و سنتز که فرآیندهایی مکمل در طراحی سیستم های دیجیتالی هستند مورد استفاده قرار می گیرند.

چگونگی برنامه نویس و دستورات ***VHDL*** خود به یک واحد درسی نیاز دارد وبرای همین ما دراینجا به آن نمی پردازیم .

۳-۱-۱. فلوچارت برنامه

همان طور که در فصل قبل مشاهده کردیم روش های مختلفی برای ضرب کردن دو عدد باینری وجود داشت که ما در این برنامه فقط از روش ***booth*** برای ضرب کردن استفاده خواهیم کرد . در روش ***booth*** که قبلا شرح دادیم بر این اساس بود که یکی از اعداد را از طریق جدول مربوطه کدبندی کرده و آنها را در عدد دیگر ضرب می کردیم و بیت های علامت آنها را گسترش می دادیم وضرب های جزئی حاصله را با هم جمع می کردیم طبق مثال زیر

```
   -51          11001101
    86          01010110          01010110 0
 --------       --------------
  -4386         0000000001100110   -2      10 0
1110111011011110  1111111001101010   2      011
                  111111001101       1      010
                  1111001101         1      010
                  ------------------------------
                  1110111011011110
```

نوشتن برنامه این حالت به خاطر گسترش بیت علامت مشکل می باشد و به همین خاطر از یک روش دیگر در الگوریتم ***booth*** استفاده می کنیم که اساس آن تقریبا بر همان است ولی بیت علامت را گسترش نمی دهیم که در زیر مثال بالا را به این ترتیب حل کرده و آنرا همراه با حل توضیح خواهیم داد.

ضرب های جزئی را بدون گسترش بیت علامت می نویسیم و ضرب جزئی اول با یک مقدار اولیه صفر جمع می کنیم.

$A=$ 00000000

$M1=$ ضرب جزئی اول $+01100110$

$B=$ 0 01100110

دو بیت اول ***B*** دو بیت اول جواب است ***PL= - - - - - - 1 0*** جواب

و به شش بیت بعد آن دو بیت باارزش به صورت زیر اضافه می کنیم دو بیت با ارزش برابر است با ***SUM*** یک ***FUUL ADDER*** که ورودی های آن بیت علامت***A***که ***0*** است و بیت ***CARRY*** از ***A+M1*** که دراینجا ***0*** می باشد و ورودی آخر وابسته به کد ***booth*** است .(اگر عددی منفی باشد عکس بیت علامت عددی که در آن ضرب می شود واگر مثبت باشد خود بیت علامت و در صورتی که صفر باشد برابر صفر می شود) که در این مرحله چون کد ***booth*** برابر ***2-*** است ویک عدد منفی است عکس بیت علامت عدد ***11001101*** را که ***0*** است می باشد .

$SS= 0+0+0 = 0$

و عدد جدید بدست آمده را به عنوان ***A*** جدید وارد کرده وبه ترتیب قبل آنرا ادامه می دهیم.

مرحله دوم

$A=$ 00011001

$M2=$ ضرب جزئی دوم $+10011010$

$B=$ 0 10110011

$PL= - - - - 1\ 1\ 1\ 0$ $SS= 0+0+1 = 1$ جدید $A=11101100$

مرحله سوم

$A=$ 11101100

$M3=$ ضرب جزئی سوم $+11001101$

$B=$ 1 10111001

$PL= - - 0\ 1\ 1\ 1\ 1\ 0$ $SS= 1+1+1 = 1$ جدید $A=11101110$

مرحله چهارم

$A=$ 11101110

$M4=$ ضرب جزئی چهارم $+11001101$

$B=$ 1 10111011

$PL= 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0$ $SS= 1+1+1 = 1$ جدید $A=11101110$

که هشت بیت بالای جواب برابر است با A مرحله آخر $PH=11101110$

$P=PH\ PL =1110111011011110$

این روش توسط فلوچارت صفحه ی بعد ترسیم شده است و مقادیر همین مثال طبق متغیر های آن در جدول زیر آورده شده است .

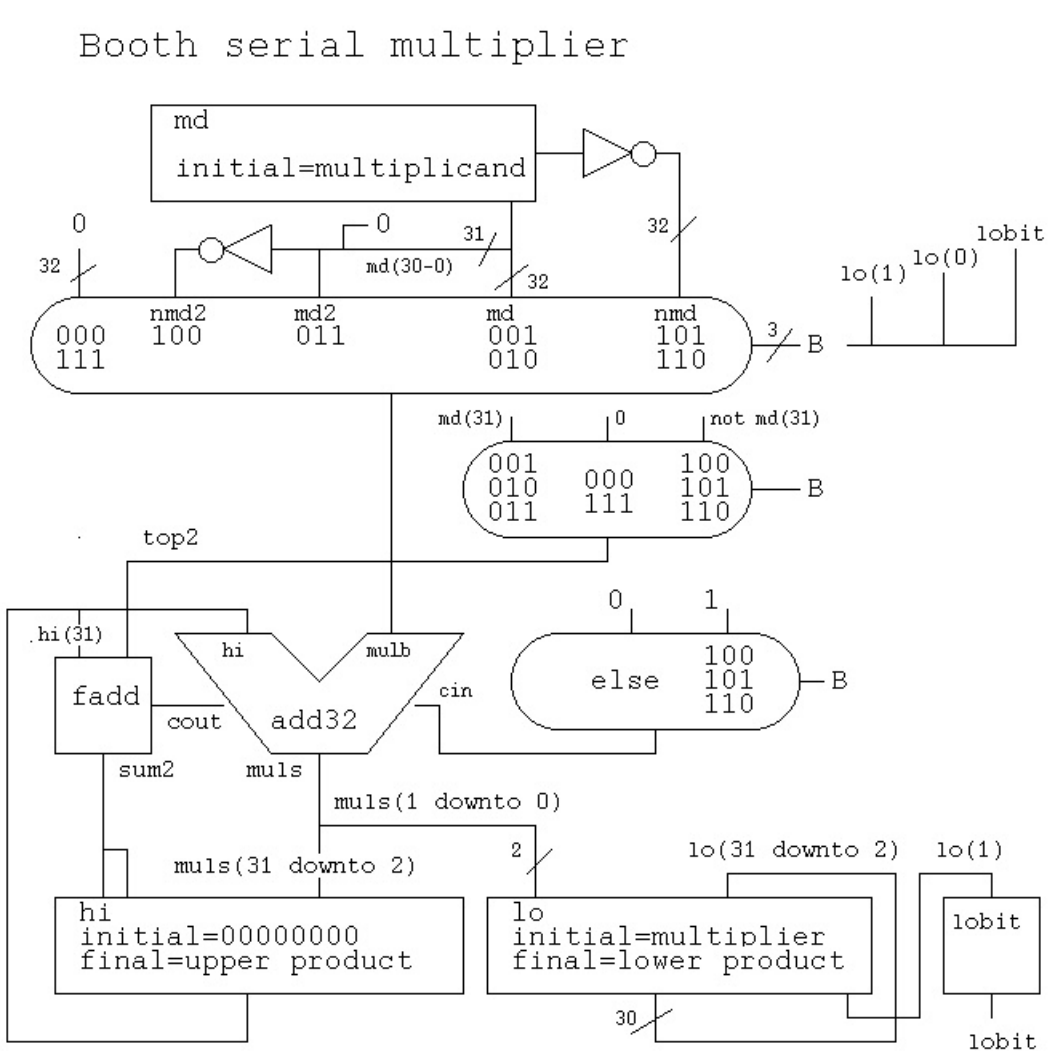
	add32				fadd				
B(- - -)	hi	mul	cin	muls	cout	hi(7)	top2	sum2	Lo
-2	00000000	01100101	1	01100110	0	0	0	0	
	00011001								10000000
2	00011001	10011010	0	10110011	0	0	1	1	
	11101100								11100000

1	*11101100*	*11001101*	*0*	*10111001*	*1*	*1*	*1*	*1*	
	11101110								*01111000*
1	*11101110*	*11001101*	*0*	*10111011*	*1*	*1*	*1*	*1*	
									11011110
	11101110								

که جواب برابر است با ***p=hi(final),lo(final)= 1110111011011110***

نکته ای که در اینجا باید به آن اشاره کرد ***cin*** است که در این فلوچارت چه نقشی دارد ؟

جواب این است که یک عدد وقتی در منفی ضرب باینری می شود متمم۲ خواهد شد یعنی ***not*** آن به اضافه یک و در فلوچارت با وارد کردن ***cin*** این به اضافه ی یک را انجام می دهم که بعدا در نوشتن برنامه مفید خواهد بود .



۳-۱-۲. برنامه ضرب کننده *booth*

این برنامه که خواهید دید دقیقا بیان کننده ی فلوچارت قبل می باشد برنامه ی اصلی را به نـام ***bmul32*** نامگذاری شده است که از زیربرنامه ***badd32*** طی ۱۶ مرحله استفاده می کند که به بیان واضـحتر برنامـه اصلی اعداد را یکی به صورت کامل و دیگری را به صورت سه بیتی مطابق با کدهای ***booth*** به زیربرنامـه می دهد و در هر مرحله دو بیت خروجی را می گیرد و در طی ۱۶ مرحله تبادل با این زیربرنامـه جـواب به دست می آید .

خود زیربرنامه ***badd32*** از دو زیربرنامـه دیگـر بـه نـام هـای ***fadd (full adder)*** و ***add32*** اسـتفاده مـی کندکه اولی ***full adder*** برای محاسبه بیت گسترش یافته در هر مرحله میباشد و دومی یک جمع کننده ی ۳۲ بیتی است که در هر مرحله انجام می شود . برنامه ی نوشته شده را همراه با زیر برنامه های آن در زیر آورده ایم .

```
library IEEE;
use IEEE.std_logic_1164.all;

entity bmul32 is   -- 32-bit by 32-bit two's complement multiplier
 port (a : in  std_logic_vector(31 downto 0);   -- multiplier
       b : in  std_logic_vector(31 downto 0);   -- multiplicand
       p : out std_logic_vector(63 downto 0));  -- product
end entity bmul32;

architecture circuits of bmul32 is
 signal zer : std_logic_vector(31 downto 0) := x"00000000";     -- zeros
 signal mul0: std_logic_vector(2 downto 0);
 subtype word is std_logic_vector(31 downto 0);
 type ary is array(0 to 15) of word;
 signal s : ary;                    -- temp sums
begin  -- circuits of bmul32
 mul0 <= a(1 downto 0) & '0';
 a0:  entity WORK.badd32 port map(
          mul0,          b, zer,   s( 0), p( 1 downto  0));
 a1:  entity WORK.badd32 port map(
          a(3  downto  1), b, s( 0), s( 1), p( 3 downto  2));
 a2:  entity WORK.badd32 port map(
          a(5  downto  3), b, s( 1), s( 2), p( 5 downto  4));
 a3:  entity WORK.badd32 port map(
          a(7  downto  5), b, s( 2), s( 3), p( 7 downto  6));
```

```
a4:  entity WORK.badd32 port map(
        a(9  downto  7), b, s( 3), s( 4), p( 9 downto  8));
a5:  entity WORK.badd32 port map(
        a(11 downto  9), b, s( 4), s( 5), p(11 downto 10));
a6:  entity WORK.badd32 port map(
        a(13 downto 11), b, s( 5), s( 6), p(13 downto 12));
a7:  entity WORK.badd32 port map(
        a(15 downto 13), b, s( 6), s( 7), p(15 downto 14));
a8:  entity WORK.badd32 port map(
        a(17 downto 15), b, s( 7), s( 8), p(17 downto 16));
a9:  entity WORK.badd32 port map(
        a(19 downto 17), b, s( 8), s( 9), p(19 downto 18));
a10: entity WORK.badd32 port map(
        a(21 downto 19), b, s( 9), s(10), p(21 downto 20));
a11: entity WORK.badd32 port map(
        a(23 downto 21), b, s(10), s(11), p(23 downto 22));
a12: entity WORK.badd32 port map(
        a(25 downto 23), b, s(11), s(12), p(25 downto 24));
a13: entity WORK.badd32 port map(
        a(27 downto 25), b, s(12), s(13), p(27 downto 26));
a14: entity WORK.badd32 port map(
        a(29 downto 27), b, s(13), s(14), p(29 downto 28));
a15: entity WORK.badd32 port map(
        a(31 downto 29), b, s(14), p(63 downto 32) , p(31 downto 30));
end architecture circuits;  -- of bmul32



library IEEE;
use IEEE.std_logic_1164.all;

entity badd32 is
 port (a       : in  std_logic_vector(2 downto 0);  -- Booth multiplier
       b       : in  std_logic_vector(31 downto 0); -- multiplicand
       sum_in  : in  std_logic_vector(31 downto 0); -- sum input
       sum_out : out std_logic_vector(31 downto 0); -- sum output
       prod    : out std_logic_vector(1 downto 0)); -- 2 bits of product
end entity badd32;

architecture circuits of badd32 is
 -- Note: Most of the multiply algorithm is performed in here.
 -- multiplier action
```

```
--     a            bb
-- i+1 i i-1         multiplier,  shift partial result two places each stage
--  0  0  0   0    pass along
--  0  0  1   +b   add
--  0  1  0   +b   add
--  0  1  1   +2b  shift add
--  1  0  0   -2b  shift subtract
--  1  0  1   -b   subtract
--  1  1  0   -b   subtract
--  1  1  1   0    pass along
 subtype word is std_logic_vector(31 downto 0);
 signal bb        : word;
 signal psum      : word;
 signal b_bar     : word;
 signal two_b     : word;
 signal two_b_bar : word;
 signal cout      : std_logic;
 signal cin       : std_logic;
 signal topbit    : std_logic;
 signal topout    : std_logic;
 signal nc1       : std_logic;
begin  -- circuits of badd32
 two_b <= b(30 downto 0) & '0';
 b_bar <= not b;
 two_b_bar <= not two_b;
 bb <= b when a="001" or a="010"          -- 5-input mux
       else two_b when a="011"
       else two_b_bar when a="100"        -- cin=1
       else b_bar when a="101" or a="110"  -- cin=1
       else x"00000000";
 cin <= '1' when a="100" or a="101" or a="110"
        else '0';
 topbit <= b(31) when a="001" or a="010" or a="011"
           else b_bar(31) when a="100" or a="101" or a="110"
           else '0';

 a1: entity WORK.add32 port map(sum_in, bb, cin, psum, cout);
 a2: entity WORK.fadd port map(sum_in(31), topbit, cout, topout, nc1);

 sum_out(29 downto 0) <= psum(31 downto 2);
 sum_out(31) <= topout;
 sum_out(30) <= topout;
```

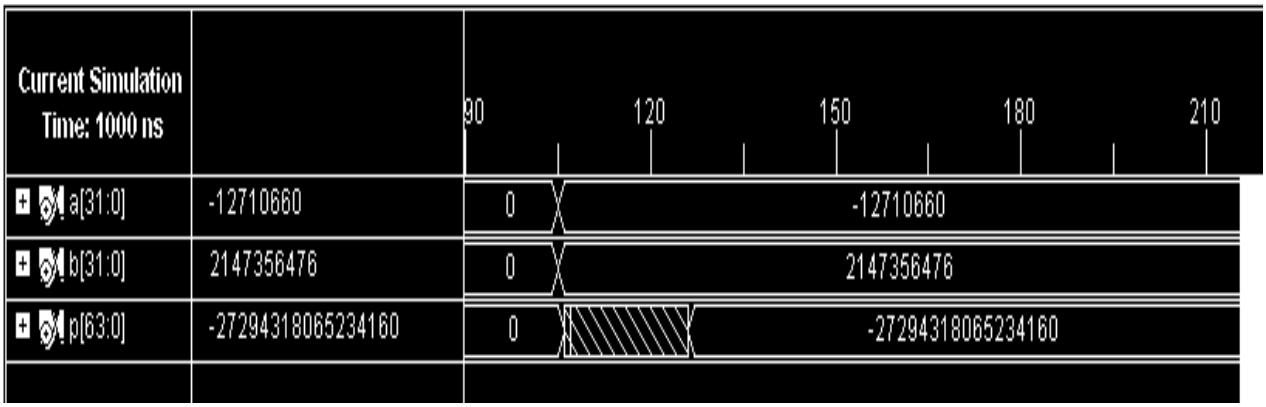
```
 prod <= psum(1 downto 0);
end architecture circuits;  -- of badd32

-- bmul32   full combinatorial 32 X 32 = 64 bit two's complement multiplier
-- Booth two's complement multiplication using badd4 component



library IEEE;
use IEEE.std_logic_1164.all;
entity add32 is -- simple 32 bit ripple carry adder
port(sum_in : in std_logic_vector(31 downto 0);
bb : in std_logic_vector(31 downto 0);
cin : in std_logic;
psum : out std_logic_vector(31 downto 0);
cout : out std_logic);
end entity add32;

architecture circuits of add32 is
signal c : std_logic_vector(0 to 30); -- internal carry signals
begin -- circuits of add32
a0: entity WORK.fadd port map(sum_in(0), bb(0), cin, psum(0), c(0));
stage: for I in 1 to 30 generate
as: entity WORK.fadd port map(sum_in(I), bb(I), c(I-1) , psum(I),
c(I));
end generate stage;
a31: entity WORK.fadd port map(sum_in(31), bb(31), c(30) , psum(31), cout);
end architecture circuits; -- of add32



library IEEE;
use IEEE.std_logic_1164.all;

entity fadd is -- full adder stage, interface
port(sum_in : in std_logic;
bb : in std_logic;
cin : in std_logic;
psum : out std_logic;
cout : out std_logic);
end entity fadd;
```

```
architecture circuits of fadd is -- full adder stage, body
begin -- circuits of fadd
psum <= sum_in xor bb xor cin after 1 ns;
cout <= (sum_in and bb) or (sum_in and cin) or (bb and cin) after 1 ns;
end architecture circuits; -- of fadd
```

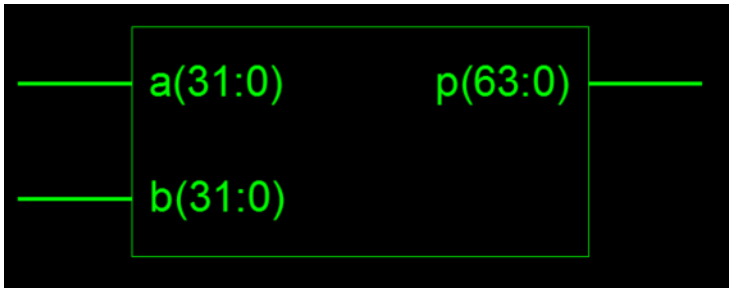
این برنامه را توسط نرم افزار ***Xilinx (ISE 9.2)*** اجرا کرده ایم و سپس برای اینکه ببینیم این برنامه عملیات ضرب را درست انجام می دهد توسط انجام می دهد آن را توسط همین نرم افزار شبیه سازی کرده ایم که در اینجا برنامه ی تست و نتایج آن را آورده ایم .

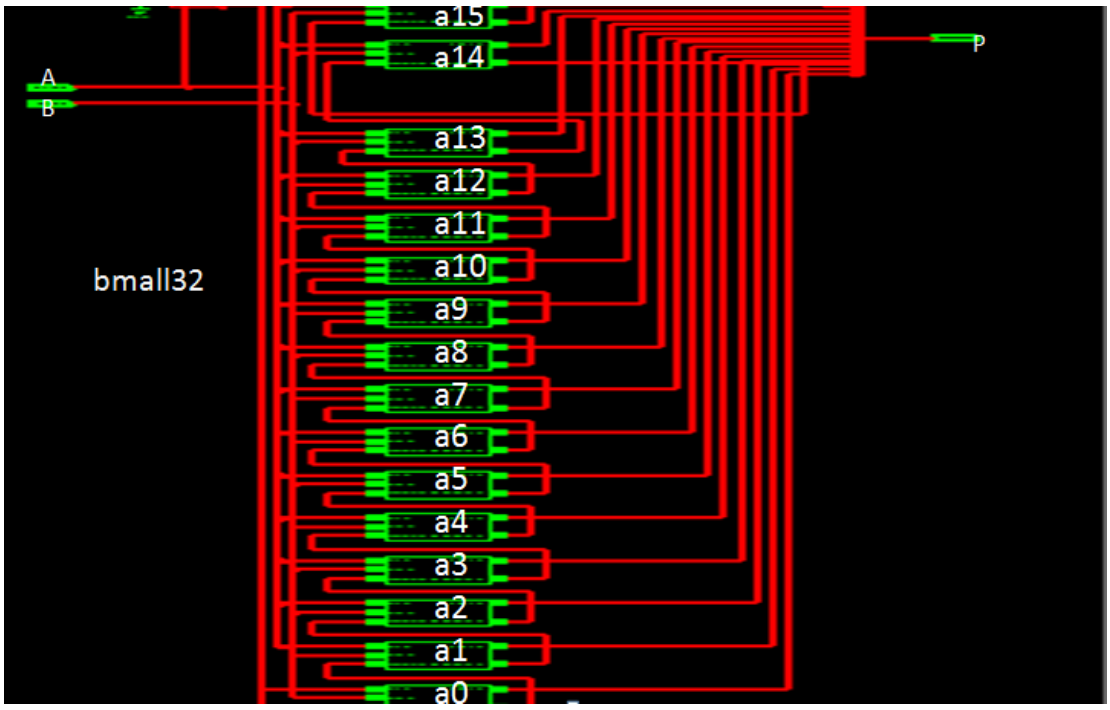
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY test1_vhd IS
END test1_vhd;

ARCHITECTURE behavior OF test1_vhd IS

    --Component Declaration for the Unit Under Test (UUT)
    COMPONENT bmul32
    PORT(
        a : IN std_logic_vector(31 downto 0)
        b : IN std_logic_vector(31 downto 0 )
        p : OUT std_logic_vector(63 downto 0)
        );
    END COMPONENT;

    --Inputs
```

```
SIGNAL a :  std_logic_vector(31 downto 0) := (others=>'0');
SIGNAL b :  std_logic_vector(31 downto 0) := (others=>'0');

	--Outputs
	SIGNAL p :  std_logic_vector(63 downto 0);

BEGIN

	 --Instantiate the Unit Under Test (UUT)
	uut: bmul32 PORT MAP(
		a => a,
		b => b,
		p => p
	);

	tb : PROCESS
	BEGIN

		 --Wait 100 ns for global reset to finish
		wait for 100 ns;

		a<="11111111001111100000110011111100" after 5ns;
		b<="01111111111111100000111100111100" after 5ns;

		wait; -- will wait forever
	END PROCESS;

END;
```



قابلیت دیگر این نرم افزار در کشیدن شماتیک مدار می باشد که برای نمونه شماتیک اصلی و یکی از زیرروال ها را در اینجا آورده ایم .

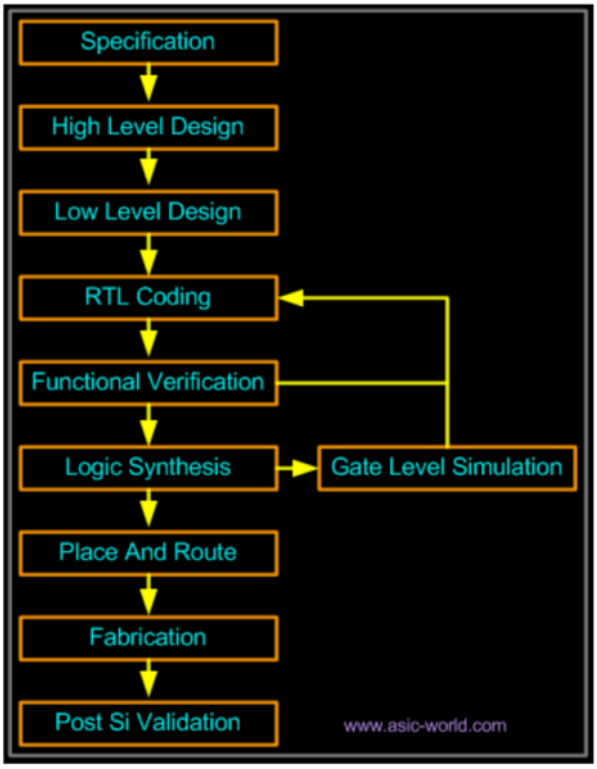




و در آخر پیاده سازی سخت افزاری آن است که آن هم توسط این نرم افـزار روی ***FPGA*** پیـاده شـده و عمل خواهد کرد .

۳-۲. *Verilog*

Verilogیک زبان توصیف سخت افزاری است. زبان توصیف سخت افزار یک زبان بـرای توصـیف سیسـتم های دیجیتالی، شبکه سوئیچ، میکرو پروسسور ها یا حافظه ها و یا فیلیپ فلاپ هاست. بدین معنی که با استفاده از زبان های توصیف سخت افزار در هر سطح طراحی امکان توصیف مدار ها وجود دارد.



۳-۲-۱. خلاصه ای از سطوح مختلف طراحی زبان *Verilog*

زبان توصیف سخت افزاری ***Verilog*** سطوح مختلـف طراحـی را پشـتیبانی مـی کنـد. مهمتـرین سـطوح طراحی در زیر آمده است:

- مدلسازی رفتاری
- ***Register-Transfer***
- سطح گیت

۳-۲-۲. روش های طراحی در زبان *Verilog*

در ***Verilog*** یک مدار یا طرح به چهار صورت قابل توصیف است:

- روش جریان داده ای
- روش رفتاری
- روش ساختاری
- ترکیبی از روش های فوق

۳-۲-۲-۱. توصیف یک مدار به روش جریان داده ای

در طراحی به روش جریان داده ای از انتساب های پیوسته استفاده میکنیم، به این معنی که یک مقدار را به یک گره از مدار نسبت می دهیم و در حین اجرای برنامه به صورت پیاپی این مقدار به گره انتساب داده می شود.

نحوه بیان آن به صورت زیر است:

Assign[delay] LHS_net = RHS_expression

۳-۲-۲-۲. توصیف یک مدار به صورت رفتاری

در این روش از ساختار های روالی برای توصیف رفتار مدار استفاده می شود. در این ساختار های روالی دستورات به ترتیبی که نوشته شده اند اجرا می شوند(بر خلاف روش جریان داده ای که همزمان اجرا می شوند). برای بیان این حالت از دو بلاک ***initial*** و ***always*** استفاده می شود.

۳-۲-۲-۳. توصیف به روش ساختاری

در این روش از ساختار های موجود برای توصیف مدار استفاده می شود،این ساختار ها شامل گیت های پایه، کلید های پایه، گیت های پایه تعریف شده توسط کاربر و ماژول ها، است.

۳-۲-۲-۴. توصیف با روش ترکیبی

در این روش از ترکیب دو روش ساختاری و رفتاری برای توصیف مدار استفاده می شود.

۳-۲-۳. عناصر زبان *Verilog*

این عناصر شامل:

- راهنماهای کامپایلر

- مجموعه مقادیر در زبان ***Verilog***
- نوع متغیر ها
- و ...

است.

۳-۲-۴. برنامه نوشته شده به زبان *Verilog*

در برنامه نوشته شده برای پیاده سازی ضرب کننده ۳۲ بیتی با روش ***Booth*** از روش توصیف رفتاری استفاده شده است.

بدین معنی که متغییر های ورودی (***A,B***) در لیست حساسیت بلوک ***always*** قرار گرفته و دستورات درون این بلوک به ترتیب اجرا می شوند.

۳-۲-۵. کد برنامه به زبان *Verilog*

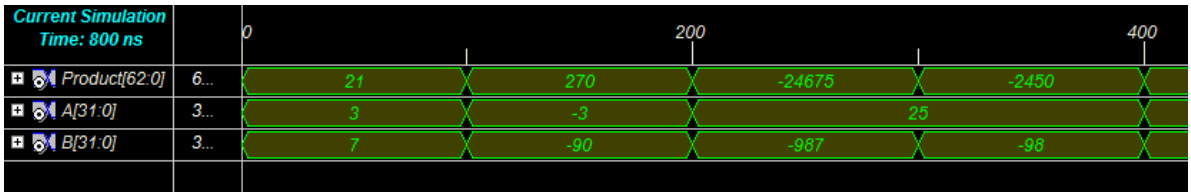
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module BoothAlgo(A,B,Product);
parameter N=31;
input[N:0] A,B;
output[2*N:0] Product;
integer SC,i;
reg[2:0] AC;
reg[N:0] CB;
reg[N:0] BP;
reg[N+1:0] AP;
reg[2*N:0] Product,Mul;

//-------------------------init
always@(A,B)
        begin
                Mul=0;
                AC=0;
                AP[N+1:1]=A;
                AP[0]=0;
                BP=B;
                CB=~B+1;
                SC=0;
```

```
Product=0;
//--------------------
while(SC<16)
        begin
                Mul=0;
                if (SC==0)
                        begin
                                AC[2:0]=AP[2:0];
                                AP=AP>>1;
                        end
                else
                        begin
                                AC=AC>>2;
                                AC[1]=AP[0];
                                AC[2]=AP[1];
                        end
                AP=AP>>2;
                AP[N]=AP[N-2];
                AP[N-1]=AP[N-2];


                case(AC)
                        3'b000:
                                begin
                                        Mul=0;
                                        for(i=N+1;i<=2*N;i=i+1)
                                                Mul[i]=Mul[N];
                                        Mul=Mul<<(SC*2);
                                        Product=Product+Mul;
                                end
                        3'b001:
                                begin
                                        Mul=BP;
                                        for(i=N+1;i<=2*N;i=i+1)
                                                Mul[i]=Mul[N];
                                        Mul=Mul<<(SC*2);
                                        Product=Product+Mul;
                                end
                        3'b010:
                                begin
                                        Mul=BP;
                                        for(i=N+1;i<=2*N;i=i+1)
                                                Mul[i]=Mul[N];
```

```
                Mul=Mul<<(SC*2);
                Product=Product+Mul;
        end
3'b011:
        begin
                Mul=BP<<1;
                for(i=N+2;i<=2*N;i=i+1)
                        Mul[i]=Mul[N+1];
                Mul=Mul<<(SC*2);
                Product=Product+Mul;
        end
3'b100:
        begin
                Mul=CB<<1;
                for(i=N+2;i<=2*N;i=i+1)
                        Mul[i]=Mul[N+1];
                Mul=Mul<<(SC*2);
                Product=Product+Mul;
        end
3'b101:
        begin
                Mul=CB;
                for(i=N+1;i<=2*N;i=i+1)
                        Mul[i]=Mul[N];
                Mul=Mul<<(SC*2);
                Product=Product+Mul;
        end
3'b110:
        begin
                Mul=CB;
                for(i=N+1;i<=2*N;i=i+1)
                        Mul[i]=Mul[N];
                Mul=Mul<<(SC*2);
                Product=Product+Mul;
        end
3'b111:
        begin
                Mul=0;
                for(i=N+1;i<=2*N;i=i+1)
                        Mul[i]=Mul[N];
                Mul=Mul<<(SC*2);
                Product=Product+Mul;
```

```
                    end
                endcase
                SC=SC+1;
            end
        end
    endmodule
```

۳-۲-۶. نتایج شبیه سازی



۴. مراجع

[1] C.Baugh y A . Wooley " A Two's Complement Parallel Array Multiplication Algoritm " . IEEE Trans.on Computer,Vol.C-22,N12.Dic.1973

[2] WWW: http://www.csse.monash.edu.au/_timf/Clayton Office: Building 63, Room G08(Multiplication and Multiplier Design)

[3] Undergraduate VLSI Design Course, 2003, National Taiwan University

[۴] کتاب "***Verilog*** طراحی مدارهای دیجیتالی" ، پرنده افشار. هادی، انتشارات نص

[5]-Verilog Tutorial, Tala.K.D

[6]-http://www.asic-world.com

تقدیر و تشکر

با تشکر از آقایان محسن مرادی و علی نادری، دانشجویان کارشناسی ارشد دانشگاه سمنان