به نام خدا

آزمایشگاه طراحی سیستم‌های دیجیتال

گزارش کار پنجم

طراحی مدار ضرب کننده

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

بهار ۱۴۰۱

استاد:

علیرضا اجلالی

دستیار آموزشی:

سحر رضاقلی

نویسندگان:

هیربد بهنام

۹۹۱۷۱۳۳۳

عرفان مجیبی

۹۹۱۰۵۷۰۷

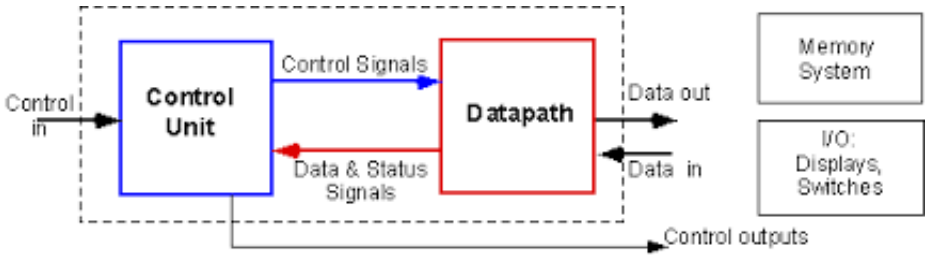
علی نظری

۹۹۱۰۲۴۰۱

فهرست

مقدمه

در این آزمایش به کمک الگوریتم Booth و طراحی جداگانه مسیر داده و واحد کنترل، مانند شکل زیر یک واحد ضرب‌کننده می‌سازیم:



شکل ۱: طراحی کلی

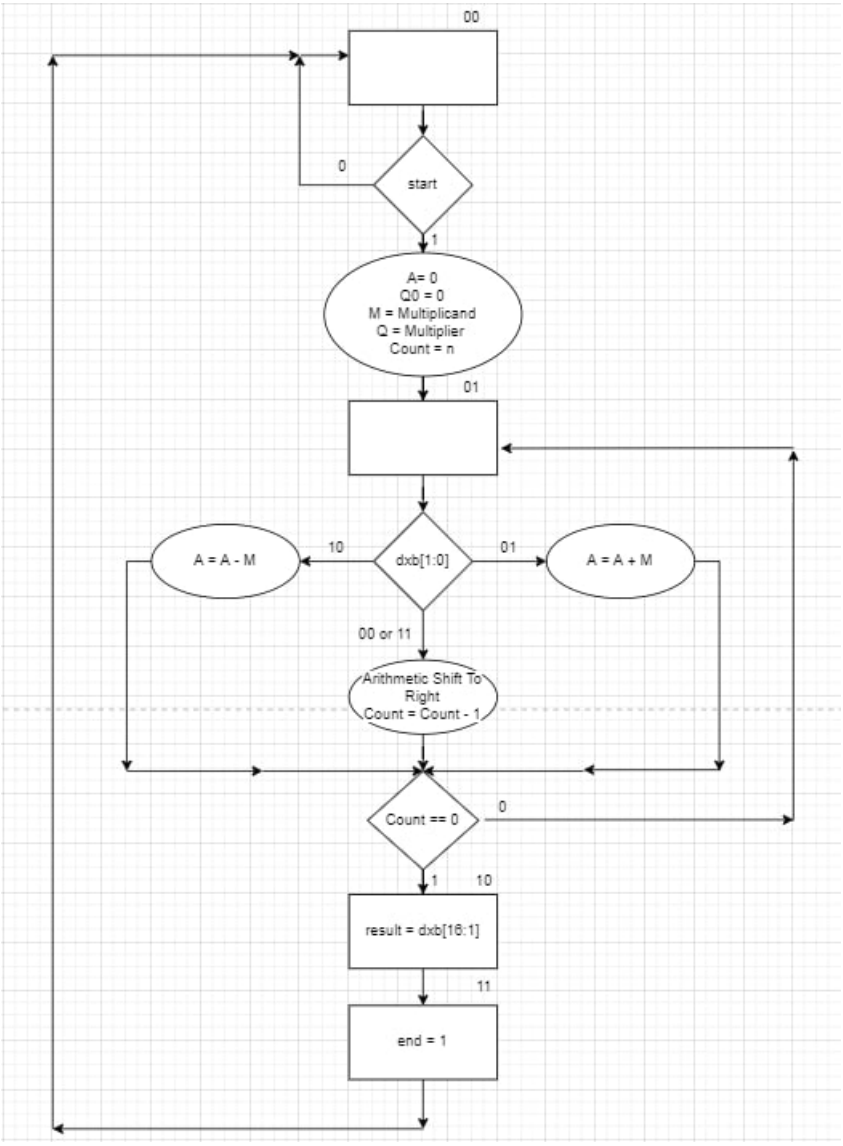
در این آزمایش نیاز است که واحد شیفت‌دهنده توان انجام شیفت بیش از یک بیت در یک پالس ساعت را داشته باشد تا نسبت به ضرب عادی، بهینه تر باشد.

گزارش آزمایش

بخش ۱. کشیدن ASM Chart و تعریف ورودی‌ها و خروجی‌ها

طراحی ASM Chart

با این ضرب‌کننده و نحوه ساخت و کارکرد آن در درس معماری کامپیوتر آشنا شده‌ایم. همانطور که گفته شد، واحد شیفت‌دهنده باید توان اینکه بیش از یک بیت را در یک clock شیفت دهد داشته باشد.
ورودی ۲ عدد n بیتی است که در این مرحله فرض می‌کنیم ۸ بیتی هستند و یک Start هم داریم که برای شروع کار مدار است و clk هم که برای سنکرون کردن مدار وجود دارد. خروجی هم یک عدد 2n بیتی است که اینجا ۱۶ بیتی می‌شود و به عنوان Result در نظر گرفته می‌شود و همان حاصل‌ضرب دو عدد علامت‌دار است و یک خروجی End هم دارد که به معنای آماده بودن جواب است.
ابتدا ASM Chart این مدار را می‌کشیم که به شکل زیر است:



شکل ۲: نمودار ASM Chart

الگوریتم استفاده شده مانند ASM Chart بالا است. در هر مرحله که شیفتی انجام می‌شود، به اندازه تعداد شیفت از count کم می‌شود.
از آنجایی که ۴ حالت داریم پس از یک رجیستر ۲ بیتی برای نمایش State استفاده می‌کنیم. همچنین تعداد دستوراتی که باید از بخش کنترل به data path منتقل شود، ۴ حالت است که به ترتیب به معنای مقداردهی اولیه، جمع با a و تفریق از a و شیفت دادن است. در این بین در هر مرحله یکی از count کم می‌کنیم و در نهارت که کار تمام شد، end را یک می‌کنیم.
مقادیری که باید به بخش کنترل منتقل شوند، مقادیر status و شمارنده و dxb[1:0] است.

توضیح الگوریتم Booth

برای ضرب کردن طبق روش بوث، یک رجیستر ۱۶ بیتی باید در نظر بگیریم و ۸ بیت کم ارزش آن را با b پر کنیم و سپس در هر مرحله با توجه به بیت کم ارزش آن، مقدار a را به ۸ بیت پر ارزش آن اضافه یا کم کنیم و سپس آن را یک بیت شیفت دهیم و سپس یکی از شمارنده کم کنیم. برای اینکه این روش را کمی بهبود ببخشیم و مقدار شیفت را بیشتر کنیم تا در کلاک کمتری، پاسخ محاسبه شود، در هر مرحله دو بیت از بیت‌های کم ارزش b را به همراه بیت قبلی که شیفت خورده بررسی می‌کنیم و با توجه به آنها تصمیم می‌گیریم که چه کار کنیم. همچنین برای نگه داشتن علامت از یک بیت اضافه‌تر در انتهای رجیستر استفاده کردیم. بنابراین رجیستر dxb شامل ۱۸ بیت است که بیت پر ارزش آن برای نگه داشتن علامت، ۸ بیت برای X که در هر مرحله مقدار a به آن اضافه یا کم می‌شود، سپس ۸ بیت که در ابتدا با b پر می‌شود و سپس یک بیت برای نگه داشتن آخرین بیتی که شیفت خورده است. در انتها مقدار جواب نهایی در بیت ۱ تا ۱۶ رجیستر dxb است.

با توجه به توضیحات بالا و نمودار ASM چارت، پس از وارد کردن اعداد و فعال کردن start که می‌تواند به صورت یک push button هم باشد، ابتدا رجیسترها مقداردهی شده و در هر مرحله با چک کردن دو بیت مورد نظر، یکی از ۴ کار گفته شده انجام می‌شود و سپس بررسی می‌شود که تمام مقادیر b یعنی به اندازه ۴ بار شیفت خورده است یا نه، و در صورتی که تمام شده بود، مقدار END یک می‌شود تا زمانی که مجددا اعدادی وارد شوند و start فشار داده شود.

بخش ۲. طراحی نهایی و زدن کد وریلاگ

کد وریلاگ مدار

حال به زدن کد و توضیح بخش‌های مختلف آن می‌پردازیم.
کدهای ControlUnit و DataPath هر کدام در یک ماژول ساخته شده‌اند و سپس در یک ماژول دیگر نمونه‌گیری شده و به هم وصل شده‌اند. تمام کد های وریلاگ موجود به پیوست است.
نخست ماژول اصلی را داریم که نام آن booth است و در این ماژول نمونه‌هایی از Data Path و Control Unit ساخته می‌شود و اتصال آن‌ها به هم برقرار می‌شود و ورودی‌های لازم به هر کدام داده می‌شود.

```
`timescale 1ns/1ns
module booth(input clk, input start, input [7:0] in1, input [7:0] in2,
    output valid, output [15:0] out);
  wire [3:0] shmnt;
  wire load, sum_or_diff, shift;
  data_path dp(.clk(clk), .input_1(in1), .input_2(in2), .dxb_input_1(out
    [15:8]), .dxb_input_2(out[7:0]), .load(load), .sum_or_diff(
    sum_or_diff), .shift(shift), .shmnt(shmnt));
  control_unit cu(.clk(clk), .start(start), .valid(valid), .Q(out[7:0]),
     .load(load), .sum_or_diff(sum_or_diff), .shift(shift), .shmnt(shmnt
    ));
endmodule
```

به توضیح متغیرهای موجود در کد بالا می‌پردازیم:

- ورودی start با یک شدنش، فرمان شروع به کار عملیات ضرب صادر می‌شود.
- ورودی clk هم که سیگنال کلاک مدار است.
- ورودی‌های in۱ و in۲ هم که دو عددی هستند که باید در هم ضرب شوند.
- خروجی out هم حاصلضرب دو عدد است که با روش Booth اندازه‌گیری شده است.
- خروجی end هم زمانی یک می‌شود که خروجی out آماده باشد و تنها پس از یک شدن این سیگنال است که می‌توانیم از خروجی out استفاده کنیم.

حال به مرحله‌ای می‌رسیم که باید مسیر داده و واحد کنترل را بسازیم و ارتباط آن‌ها با هم را مشخص کنیم. ورودی‌های کنترلی load و sum-or-diff و shift را به واحد کنترل می‌دهیم تا بر حسب شرایط آن عملیاتی که باید اجرا شود را مشخص کند و همین‌ها را به مسیر داده هم می‌دهیم و مسیر داده بر حسب سیگنالی که واحد کنترل یک می‌کند، عملیات مورد نظر را انجام می‌دهد. و shmnt هم تعداد بیتی است که باید شیفت دهیم و با این روش می‌توان بیش از یک بیت در هر کلاک هم شیفت داد. در ادامه، مسیر داده نمونه‌گیری شده است و پارامترهای لازم به آن داده شده است. در این مثال چون ورودی‌ها ۸ بیتی هستند، خروجی ۱۶ بیتی است که نصف آن را in۱ در بر می‌گیرد و نصف دیگر را in۲ در بر می‌گیرد. در واحد کنترل، ما in۲ را به عنوان نیمه راست out به نمونه داده‌ایم، این است که از روی این مقدار باید میزان shmnt را محاسبه کنیم.

حال به طراحی مسیر داده می‌پردازیم که کد آن در زیر قابل مشاهده است:

```
`timescale 1ns/1ns
module data_path(input clk, input [7:0] input_1, input [7:0] input_2,
    output reg [7:0] dxb_input_1, output reg [7:0] dxb_input_2, input
    load, input sum_or_diff, input shift, input [3:0] shmnt);
  reg [7:0] M;
  reg LSB;
  always @(posedge clk)
  begin
    if (load)
    begin
       M <= input_1;
       dxb_input_1 <= 0;
       dxb_input_2 <= input_2;
       LSB <= 0;
    end
    else if (sum_or_diff)
    begin
      if (dxb_input_2[0] == 1 && LSB == 0)
        dxb_input_1 <= dxb_input_1 - M;
      else if (dxb_input_2[0] == 0 && LSB == 1)
        dxb_input_1 <= dxb_input_1 + M;
    end
    else if (shift)
      {dxb_input_1, dxb_input_2, LSB} <= $signed({dxb_input_1,
    dxb_input_2, LSB}) >>> shmnt;
  end
endmodule
```

این بخش عملیات را روی ورودی‌ها و خروجی‌ها انجام می‌دهد پس همه‌ی ورودی‌ها و خروجی‌ها را به آن داده‌ایم. همچنین سیگنال‌هایی که واحد کنترل روی آن‌ها کار می‌کنند هم داده شده است تا این مسیر داده بتواند در هر کلاک بفهمد چه کاری با داده‌ها باید انجام دهد. همانطور که می‌دانیم در این الگوریتم ضرب، multiplicand بسیار اهمیت زیادی دارد پس چون در مرحله‌ای ممکن است که بخواهیم عملیات load را انجام دهیم، یک متغیر M هم ساخته‌ایم تا بتوان در این شرایط، multiplicand را داخل این قرار دهیم. همچنین چون عملیات‌های ریاضی را بر حسب بیت کم‌ارزش حال و بیت کم‌ارزش قبلی باید انجام دهیم، پس یک متغیر LSB هم داریم که بیت کم‌ارزش مرحله قبل را نگه می‌دارد. در واقع از مقایسه lsb مرحله قبل و مرحله فعلی می‌توان تصمیم گرفت که چه عملیاتی انجام دهیم. همانطور که در ASM چارت هم دیدیم، یک متغیر dxb داریم که این همان تلفیق شده دو ورودی و بیت کم‌ارزش مرحله قبل است و همه با هم این متغیر را می‌سازند به این شکل که اول از چپ به راست multiplicand و بعد ورودی دوم و در نهایت هم بیت کم‌ارزش مرحله قبل قرار می‌گیرد. در مرحله بعد، در یک بلاک always به لبه بالارونده کلاک حساسیت نشان می‌دهیم و چک می‌کنیم که واحد کنترل از ما خواسته که چه کاری انجام دهیم.

- اگر سیگنال load یک باشد، یعنی باید ورودی‌ها را در رجیسترهای داخلی نگه داریم. پس مقدار multiplicand را در M می‌ریزیم و بخش سمت چپ dxb را که in۱ در آن قرار دارد را صفر می‌کنیم چون در الگوریتم این کار انجام می‌شود.

و in۲ را در همان نیمه سمت راست dxb می‌گذاریم و lsb که از مرحله قبل داریم را هم صفر می‌کنیم چون کار load را انجام داده‌ایم.

- اگر سیگنال sum-or-diff یک بود یعنی باید عملیات ریاضی انجام دهیم. برای اینکار باید lsb فعلی و مرحله قبل را مقایسه کنیم و بر اساس آن‌ها یکی از عملیات‌ها را انجام دهیم. اگر مرحله قبل lsb صفر بوده و الان یک شده باشد، باید عمل تفریق را انجام دهیم. اگر lsb در مرحله قبل یک بوده و الان صفر شده باشد، باید عمل جمع را انجام دهیم. و اگر lsb مرحله قبل با این مرحله فرقی نداشته باشد یعنی هر دو صفر یا هر دو یک باشند، کاری نباید انجام دهیم.
- اگر هم سیگنال شیفت یک شده بود که باید به میزان مقدار shmnt شیفت به راست بدهیم و اینکار را هم روی dxb انجام می‌دهیم.

حال به طراحی واحد کنترل می‌پردازیم:

```
`timescale 1ns/1ns
module control_unit(input clk, input start, output valid, input [7:0] Q,
    output load, output sum_or_diff, output shift, output [3:0] shmnt);
  reg[3:0] current, next, counter;
  assign load = current[0];
  assign sum_or_diff = current[1];
  assign shift = current[2];
  assign valid = current[3];
  always @(current, counter)
  begin
    next <= 0;
    if(current[0]) next[1] <= 1'b1;
    if(current[1]) next[2] <= 1'b1;
    if(current[2])
      if (counter > shmnt) next[1] <= 1'b1;
      else next[3] <= 1'b1;
    if(current[3]) next[3] <= 1'b1;
  end
  always @(posedge clk)
    if (start)
    begin
      current <= 1;
      counter <= 8;
    end
    else
    begin
      current <= next;
      if (current[2]) counter <= counter - shmnt;
    end
  wire [7:0] diff_pairs = ( Q ^ (Q >> 1) ) | (8'b10000000);
  reg [3:0] lsb_one;
```

```
  integer i;
  always @(*)
  begin
    lsb_one = 0;
    for (i = 0; i <= 7; i = i+1)
      if (diff_pairs[i] && lsb_one == 0) lsb_one = i + 1;
  end
  assign shmnt = (counter>lsb_one) ? lsb_one : counter;
endmodule
```

ورودی‌های این بخش هم که شامل کلاک و start است که یک شدن آن یعنی شروع به کار ضرب کننده و خروجی out هم مشخص می‌کند که خروجی ضرب آماده شده است یا خیر و تا وقتی این سیگنال یک نشده باشد، جواب ضرب به درستی آماده نشده است. خروجی‌های کنترلی هم که همان load و sum-or-diff و shift است که بر اساس آن‌ها مسیر داده متوجه می‌شود که چه کاری باید انجام دهد. همانطور که از ASM چارت دیدیم و روش One Hot را هم از مدار منطقی خواندیم، هر حالت رجیسترها یک state از مدار را دارند مشخص می‌کنند. همانطور که در بخش مسیر داده گفته شد، ما باید مدام حالت قبلی و حالت فعلی را مقایسه کنیم و بر اساس این‌ها و مقایسه‌هایی که روی آن‌ها انجام می‌دهیم، متوجه روند بشویم پس نیاز است که حالت فعلی و حالت قبلی را در رجیسترهایی نگه داریم تا بعدا از آن‌ها استفاده کنیم. در این دو رجیستر که تعریف شده و ۴ بیتی هستند، اگر بیت صفر یک باشد، یعنی عمل load روی آن‌ها انجام شده و اگر بیت یک آن‌ها یک باشد یعنی همل جمع یا تفریق در آن‌ها انجام شده است. اگر بیت دوم یک باشد یعنی عملیات شیفت روی آن‌ها انجام شده است. و اگر بیت سوم یک شده باشد، یعنی کار روی آن‌ها تمام شده است. برای اینکه بتوانیم پایان الگوریتم را مشخص کنیم و سیگنال valid را یک کنیم، باید یک counter داشته باشیم تا با آن حواسمان باشد که بیش از اندازه در الگوریتم باقی نمانیم و اندازه این counter هم برابر با لگاریتم تعداد بیت ورودی در مبنای ۲ به علاوه یک است. حال به سراغ سیگنال‌های کنترلی می‌رویم:

سیگنال load وقتی یک است که در State فعلی ما، بیت صفرم یک باشد.

سیگنال sum-or-diff هم وقتی یک می‌شود که در State کنونی، بیت اول یک باشد.

سیگنال shift هم وقتی یک می‌شود که در State کنونی، بیت دوم یک باشد.

سیگنال valid هم وقتی یک می‌شود که در State کنونی، بیت سوم یک باشد.

در بلاک always بعدش هم هرگاه مقدار counter یا حالت فعلی، تغییری کند، فراخوانی می‌شود و به داخل آن می‌رویم. به محض ورود به این بلاک، حالت بعدی را صفر می‌کنیم تا بتوانیم از نو حالت بعدی را بسازیم و به مسیر داده بدهیم تا عملیات درست را بتواند انجام دهد. در ادامه، اگر در مرحله قبل در حالت load بودیم، در این مرحله به حالت sum-or-diff باید برویم و اگر در این مرحله در حالت sum-or-diff بوده‌ایم، در مرحله بعد به حالت shift باید برویم. اگر هم در حالت فعلی در shift هستیم، باید چک کنیم که آیا ضرب تمام شده است یا نه و این کار را با مقایسه shmnt با count انجام می‌دهیم و اگر کار ضرب تمام نشده بود، باز به مرحله sum-or-diff بر می‌گردیم و این مرحله‌ها را تکرار می‌کنیم و در نهایت اگر حالت فعلی Shift باشد و کار ضرب هم تمام شده باشد، به پایان کار می‌رسیم و valid را یک می‌کنیم. البته در بلاک always اول فقط حالت‌ها بررسی شده و هیچگونه عمل assign ای انجام نشده است ودر بلاک always بعدی است که اینکار assign را انجام داده‌ایم. در بلاک always دوم، به لبه بالارونده حساس هستیم. در این بلاک، اگر سیگنال start آمده بود، کار مدار شروع می‌شود و مقدار counter ست می‌شود و حالت فعلی هم برابر با یک می‌شود که معادل همان حالت load است و عملیات load در مسیر داده انجام می‌شود. اگر اینطور نباشد، حالت بعدی می‌شود حالت فعلی و به دور بعدی می‌رویم و قبل از آن چون counter مشخص می‌کند چند دور دیگه مانده و بعد از هر دور باید از آن به میزان shmnt کم شود، این مقدار را از counter کم می‌کنیم. البته این کارها زمانی انجام می‌شود که در مرحله شیفت باشیم.

سپس برای مقایسه حالت قبل و حالت فعلی، متغیر diff-pairs تعریف شده است. اول Q را یک بیت به راست شیفت داده‌ایم. و سپس این شیفت داده شده را با خود Q ایگزور کردیم تا تفاوت بیت‌های متناظر مشخص شود. و چون می‌خواهیم تمام بیت‌ها

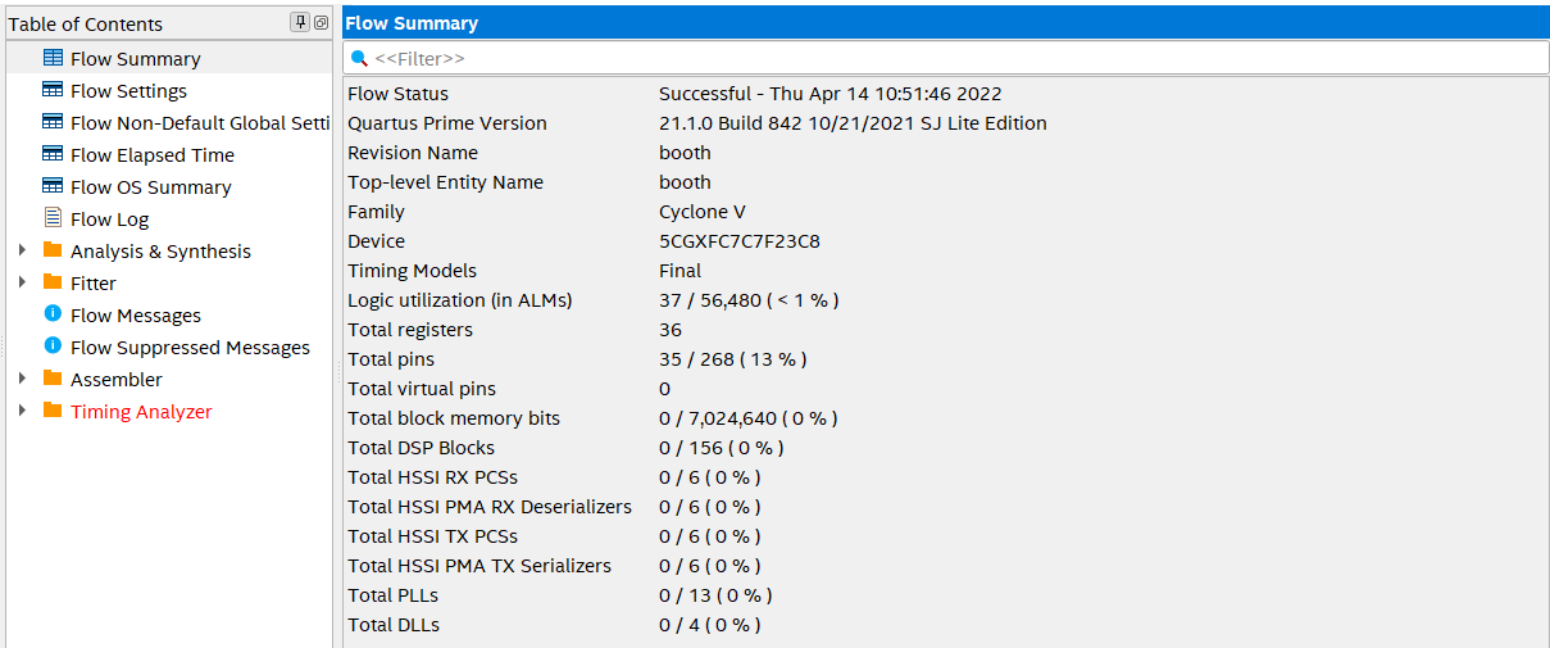
را شیفت دهیم، پس پر ارزش ترین بیت باید یک باشد تا به بردار تمام صفر نرسیم. در متغیر lsb-one هم وظیفه نگهداری اولین بیت یک که پیدا می‌شود را از راست بر عهده دارد تا بتوان به آن میزان شیفت داد و به اولین یک موجود رسید. در بلاک always بعدش هم با تغییر روی هر موردی می‌آییم از اول جای این نخستین بیت یک را پیدا می‌کنیم و ریست می‌کنیم تا بتوانیم محاسبات درستی داشته باشیم و همه‌ی این موارد بر حسب الگوریتم موجود انجام می‌شود که در کتاب‌های معماری کامپیوتر موجود است. در نهایت هم می‌توانیم مقدار shmnt را برابر جایگاه این اولین بیت یک قرار دهیم چون پرش از روی صفرها، خللی در الگوریتم ایجاد نمی‌کند ولی نکته مهم این است که باید مینیمم این مقدار و count را درون shmnt بذاریم تا یک وقت بیش از حد حرکت نکنیم و جواب خراب نشود چون در هر صورت ما به اندازه count باید عملیات‌ها را تکرار کنیم.

حال تست بنچ را می‌نویسیم:

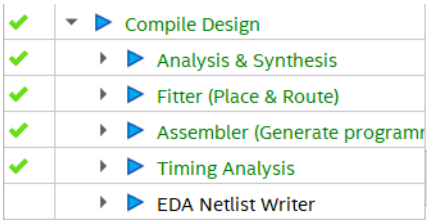
```
`timescale 1ns/1ns
module test_bench;
  reg clk, start;
  reg [7:0] in1, in2;
  wire valid;
  wire [15:0] out;
  booth exe_test(.clk(clk), .start(start), .in1(in1), .in2(in2), .valid(
    valid), .out(out));
  initial clk = 1;
  always #5 clk = ~clk;
  initial
  begin
    start <= 1;
    in1 <= 4;
    in2 <= 5;
    #10;
    start <= 0;

    #400;
    start <= 1;
    in1 <= 6;
    in2 <= 10;
    #10;
    start <= 0;

    #400;
    start <= 1;
    in1 <= -10;
    in2 <= 20;
    #10;
    start <= 0;
  end
```

```
endmodule
```

در تست هم اول از ماژول اصلی instance گرفتیم و پس از آن ورودی‌ها را تغییر دادیم. ورودی‌ها هم که wire هستند و خروجی‌ها reg هستند. برای ساخت کلاک هم که به روش همیشگی، داخل یک بلاک always در مدت زمان مشخصی، مدام clock را نقیض می‌کنیم. آن زمان زیاد ۴۰۰ نانو ثانیه هم برای این است که عملیات قبلی مطمئنا تمام شده باشد و تداخل نخورد و البته برای اینکار میشد در یک بلاک always که حساس به لبه بالارونده valid است، اینکار را انجام دهیم ولی در هر صورت ۳ جفت عدد به مدار داده شده است که نتیجه یکی منفی می‌شود و نتیجه همه به درستی توسط مدار به ما اعلام شده است که در زیر نمونه‌ها را می‌بینیم.



شکل ۳: کامپایل و سنتز موفقیت‌آمیز



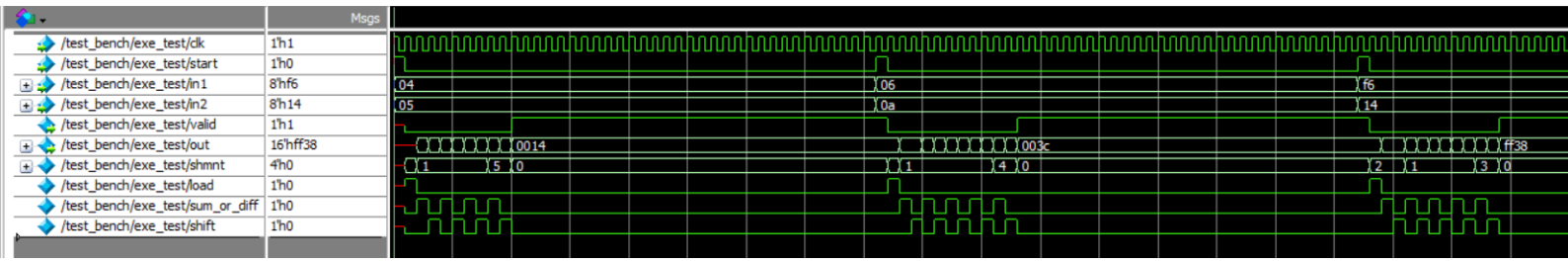
شکل ۴: کامپایل و سنتز موفقیت‌آمیز

در ابزار مدلسیم هم کامپایل با موفقیت انجام می‌شود:

```
# Compile of booth.v was successful.
# Compile of control_unit.v was successful.
# Compile of data_path.v was successful.
# Compile of test_bench.v was successful.
```

شکل ۵: کامپایل و سنتز موفقیت‌آمیز

حال به خروجی TestBench نگاه می‌کنیم:



شکل ۶: test bench

همانطور که از بخش‌های مختلف و نتیجه مشخص است، جواب به درستی دارد محاسبه می‌شود و اگر تصویر کمی کوچک است، فایل‌های پروژه موجود است.

نتیجه‌گیری

در این آزمایش فهمیدیم که چطور یک مدار ضرب‌کننده با الگوریتم Booth طراحی کنیم و در این راه برای بهتر شدن سرعت و پیدا کردن مزیت برای طراحی این مدار، کاری کردیم که در هر کلاک، بتوان تعداد بیشتری شیفت داد. و مثلا در این مثال، مزیت نسبت به الگوریتم عادی add and shift این است که در هر کلاک تغییرات محاسبه شده و دو شیفت به جای یک شیفت صورت می‌گیرد و یک عدد ۸ بیتی در ۴ کلاک محاسباتش به اتمام می‌رسد. با توجه به مدار ما که مقدار دهی اولیه و فعال کردن خروجی را داریم، به اندازه ۶ کلاک طول می‌کشد تا مقدار نهایی ضرب تولید شود.