به نام خدا

آزمایشگاه طراحی سیستم‌های دیجیتال

گزارش کار سوم

توصیف جریان داده

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

بهار ۱۴۰۱

استاد:

علیرضا اجلالی

دستیار آموزشی:

سحر رضاقلی

گروه:

هیربد بهنام

۹۹۱۷۱۳۳۳

علی نظری

۹۹۱۰۲۴۰۱

عرفان مجیبی

۹۹۱۰۵۷۰۷

فهرست

مقدمه

در این آزمایش می‌خواهیم با دستور `assign` در verilog آشنا شویم. به کمک این دستور یا keyword می‌توانیم یک جریان را به سیمی بدهیم.
برای آشنایی با این دستور می‌خواهیم در ابتدا یک مقایسه کننده‌ی یک بیتی بسازیم. در ادامه با استفاده از چهار تا از این مقایسه کننده‌ها یک مقایسه کننده‌ی چهار بیتی می‌سازیم.
سپس یک مقایسه کننده‌ی سریال می‌سازیم که در هر کلاک دو بیت را به عنوان ارقام عدد دریافت می‌کند و اعداد را مقایسه می‌کند.
در ساخت این مقایسه کننده فقط مجاز به استفاده از `assign` هستیم. پس باید flip flopهای مورد نیازمان را در سطح سیم طراحی کنیم.

گزارش آزمایش

بخش ۱. Cascadable 1-bit comparator

مقایسه کننده‌ی تک بیتی

در ابتدا می‌خواهیم که یک مقایسه کننده‌ی یک بیتی بسازیم که علاوه بر دو رقمی که می‌خواهیم مقایسه کنیم، دو بیت هم به عنوان حالت کوچکتر یا بزرگتر بودن رقم‌های قبلی به این شمارنده می‌دهیم. جدول زیر حالت‌های خروجی و ورودی مدار را نشان می‌دهد:

i_gt	i_eq	i_lt	o_gt	o_eq	o_lt
0	0	1	0	0	1
1	0	0	1	0	0
0	1	0	$x > y$	$x = y$	$x < y$

جدول ۱: جدول درستی مقایسه کننده

حال با کمی توجه متوجه می‌شویم که صفر بودن o_gt و o_eq نشان دهنده‌ی یک بودن o_lt است. پس می‌توان مدار را ساده سازی کرد با حذف ورودی و خروجی i_lt و o_lt. حال شروع به نوشتن کد verilog می‌کنیم.
در ابتدا یک ماژول تعریف می‌کنیم که ورودی‌های i_gt، i_eq، x و y را می‌گیرد و خروجی‌های o_gt و o_eq را دارد. زمانی باید o_eq را فعال کنیم که مقایسه تا قبل از این مقایسه، مساوی بوده باشد و $x = y$ باشد. برای فعال کردن o_gt باید اول از همه ببینیم که آیا i_gt برابر یک است یا خیر. در صورتی که آن برابر یک بود درجا باید o_gt را نیز برابر یک کنیم. در غیر این صورت باید ببینیم که آیا تا قبل از این مقایسه اعداد مساوی هستند یا خیر. در صورتی که مساوی بودند، باید ببینیم که آیا $x > y$ است یا خیر. پس کد وریلاگ این ماژول به صورت زیر است:

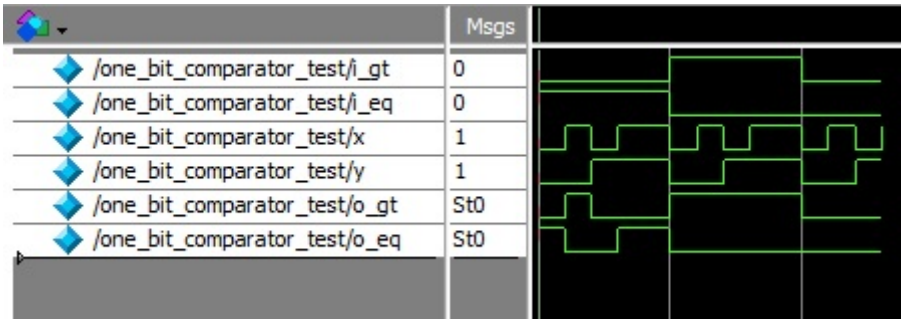
```
module one_bit_comparator (
    input i_gt,
    input i_eq,
    input x,
    input y,
    output o_gt,
    output o_eq
);
    assign o_gt = i_gt | (i_eq & (x > y));
    assign o_eq = i_eq & (x == y);
endmodule
```

تست:
حال باید این مقایسه کننده را تست کنیم. تست بنچی می‌نویسیم که تمامی حالات ممکن را بررسی کند. کد این تست بنچ در زیر آمده است:

```
`include "one_bit_comparator.v"

module one_bit_comparator_test;
```

```
    reg i_gt, i_eq, x, y;
    wire o_gt, o_eq;
    one_bit_comparator c(i_gt, i_eq, x, y, o_gt, o_eq);

    initial begin
        // Simple even tests
        i_gt = 1'b0;
        i_eq = 1'b1;
        x = 1'b0;
        y = 1'b0;
        while ({y, x} != 2'b11) begin
            #10 {y, x} = {y, x} + 1;
        end
        // Before was greater
        #20
        i_gt = 1'b1;
        i_eq = 1'b0;
        x = 1'b0;
        y = 1'b0;
        while ({y, x} != 2'b11) begin
            #10 {y, x} = {y, x} + 1;
        end
        // Before was smaller
        #20
        i_gt = 1'b0;
        i_eq = 1'b0;
        x = 1'b0;
        y = 1'b0;
        while ({y, x} != 2'b11) begin
            #10 {y, x} = {y, x} + 1;
        end
        $stop;
    end

    initial begin
        $monitor("i_gt=%d i_eq=%d x=%d y=%d o_gt=%d o_eq=%d\n", i_gt,
    i_eq, x, y, o_gt, o_eq);
    end
endmodule
```

و این کد را در ModelSim اجرا می‌کنیم. نتیجه به صورت زیر است:



شکل ۱: نتیجه‌ی تست مقایسه کننده‌ی یک بیت

در قسمت بعد دو module را با هم سنتز می‌کنیم.

مقایسه کننده‌ی چهار بیتی

حال باید با استفاده از چها مقایسه کننده‌ی یک بیتی یک مقایسه کننده‌ی چهار بیتی بسازیم. برای این کار، در ابتدا، باید ورودی‌های مقایسه کننده‌ی اول را مشخص کنیم. پرارزش ترین بیت‌های دو عدد را به x و y وصل می‌کنیم. همچنین به ورودی i_eq باید مقدار یک را بدهیم چرا که در غیر این صورت مقایسه بلافاصله تمام می‌شود! به همین دلیل باید i_gt را ۰ بدهیم.
حال برای مقایسه کننده‌های بعدی باید خروجی‌های o_eq و o_gt را به ورودی‌های متناظر مقایسه کننده‌ی بعدی وصل کنیم. همین کار را برای ۳ مقایسه کننده‌ی بعدی بکنیم. کد وریلاگ این مقایسه کننده به صورت زیر است:

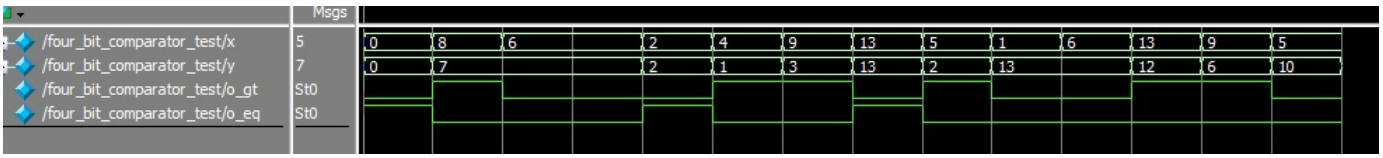
```
`include "one_bit_comparator.v"

module four_bit_comparator (
    input[3:0] x,
    input[3:0] y,
    output o_gt,
    output o_eq
);
    wire c1_gt, c1_eq, c2_gt, c2_eq, c3_gt, c3_eq;
    one_bit_comparator c1(1'b0, 1'b1, x[3], y[3], c1_gt, c1_eq);
    one_bit_comparator c2(c1_gt, c1_eq, x[2], y[2], c2_gt, c2_eq);
    one_bit_comparator c3(c2_gt, c2_eq, x[1], y[1], c3_gt, c3_eq);
    one_bit_comparator c4(c3_gt, c3_eq, x[0], y[0], o_gt, o_eq);
endmodule
```

تست:
برای نوشتن تست این مقایسه کننده نیز از تولید کننده‌ی اعداد تصادفی استفاده می‌کنیم. بدین صورت که چندین عدد تصادفی درست می‌کنیم و آنها را مقایسه می‌کنیم. کد تست در زیر آمده است:

```
`include "four_bit_comparator.v"

```

```
module four_bit_comparator_test;
    reg[3:0] x;
    reg[3:0] y;
    integer i;
    wire o_gt, o_eq;
    four_bit_comparator c(x, y, o_gt, o_eq);

    initial begin
        // Simple tests
        x = 4'b0000;
        y = 4'b0000;
        #10 x = 4'b1000;
        y = 4'b0111;
        #10 x = 4'b0110;
        y = 4'b0111;
        #10 x = 4'b0110;
        y = 4'b0111;
        #10 x = 4'b0010;
        y = 4'b0010;
        // Fuzzing
        for (i = 0; i < 10 ; i = i + 1) begin
            #10 x = $random;
            y = $random;
        end
        $stop;
    end

    initial begin
        $monitor("x=%d y=%d o_gt=%d o_eq=%d", x, y, o_gt, o_eq);
    end
endmodule
```

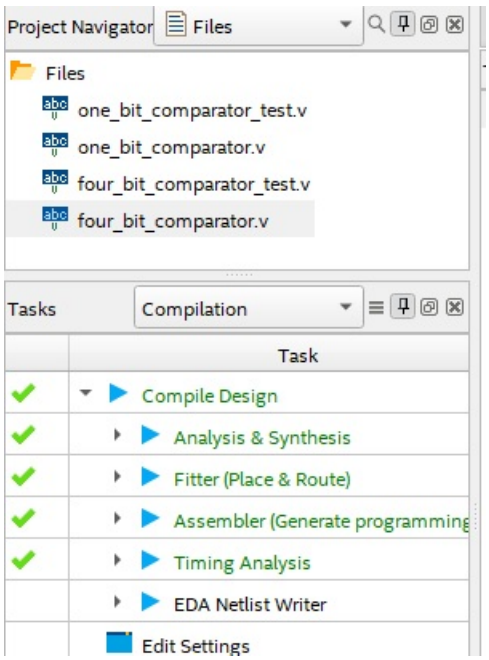
اعداد تصادفی تولید شده به صورت زیر هستند:



شکل ۲: نتیجه‌ی تست مقایسه کننده‌ی چهار بیتی

سنتز کد:

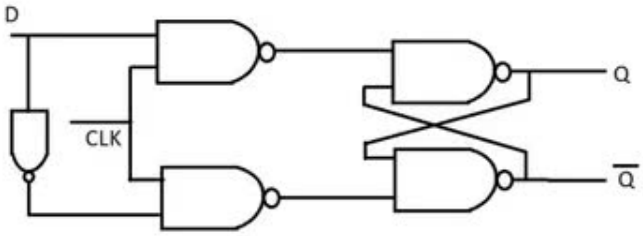
این کار را نیز در کوئارتوس انجام می‌دهیم. دقت کنید که قبل از سنتز کردن کد باید خطهای `include را از فایل‌هایمان پاک کنیم. چرا که کوئارتوس همه‌ی فایل‌ها را با هم کامپایل می‌کند و در صورتی که این کار را نکنیم با ارور مواجه می‌شویم.



شکل ۳: سنتز مقایسه کننده‌ی چهار بیتی

بخش ۲. مقایسه کننده‌ی سریال

برای حل این قسمت همان طور که مشخص است باید در ابتدا flip flop را با استفاده از گیت‌های پایه بسازیم. یک نمونه از level triggered flip flop به صورت زیر است:



شکل ۴: مدار یک dff

حال باید به کمک دو فلیپ فلاپ که یکی حالت بزرگتر بودن و یکی حالت کوچک بودن را ذخیره می‌کنند مدار را بسازیم. در صورتی که حالت بزرگتر فعال باشد، بدون توجه به ورودی‌های بعدی باید حالت بزرگتر را برابر یک قرار دهیم و حالت کوچکتر را برابر صفر. در صورتی که حالت کوچکتر نیز فعال بود باید بدون نگاه کردن به ورودی خروجی کوچکتر بودن را فعال کنیم و خروجی بزرگتر بودن را غیر فعال. همچنین در صورتی که سیگنال reset فعال باشد باید بدون شرط باید جفت فلیپ فلاپ‌ها را صفر کنیم.

با توجه به گفته‌های فوق کد وریلاگ مدار را می‌نویسیم:

```
module sequential_comparator (
    input x,
    input y,
    input reset,
    input clk,
    output o_gt,
    output o_lt
);
    wire o_gt_not, o_lt_not, i_gt, i_lt;
    // Create the flip flops
    assign o_gt = ~(o_gt_not & ~(i_gt & clk));
    assign o_gt_not = ~(o_gt & ~(~i_gt & clk));
    assign o_lt = ~(o_lt_not & ~(i_lt & clk));
    assign o_lt_not = ~(o_lt & ~(~i_lt & clk));
    // Assign the inputs of flop flops
    assign i_gt = (~reset) & (o_gt | ((~i_lt) & (x > y)));
    assign i_lt = (~reset) & (o_lt | ((~i_gt) & (x < y)));
endmodule
```

تست:

برای تست این مقایسه کننده در ابتدا چند تست کیس هاردکد شده می‌نویسیم. سپس چندین تست کیس رندوم می‌سازیم که از کارکرد این قطعه مطمئن شویم. برای این کار ابتدا سیگنال ریست را فعال می‌کنیم و دو بیت رندوم به عنوان ورودی مدار به آن می‌دهیم. سپس آن دو بیت را به دو متغیر نسبت می‌دهیم. از این متغیرها برای چاپ اعداد مقایسه شده استفاده می‌کنیم. سپس چهار بیت رندوم دیگر می‌سازیم؛ متغیرهای ذکر شده را یک بیت به راست شیف می‌دهیم و دو بیت رندوم دیگر را به آن اضافه می‌کنیم. این دو بیت را به مدار هم می‌دهیم. سپس یک واحد زمان صبر می‌کنیم و نتیجه‌ی مدار را چاپ می‌کنیم.

```
`include "sequential_comparator.v"

module sequential_comparator_test;
    reg x, y, clk, reset;
    wire gt, lt;
    integer i, j, temp_x, temp_y;

    sequential_comparator sc(x, y, reset, clk, gt, lt);

    initial clk = 1'b1;
    always #5 clk = ~clk;

    initial begin
        // Hardcoded test
        reset = 1'b1;
        x = 1'b0;
        y = 1'b0;
        #10
        reset = 1'b0;
        x = 1'b0;
        y = 1'b0;
        #10
        x = 1'b1;
        y = 1'b0;
        #10
        x = 1'b0;
        y = 1'b1;
        #10
        x = 1'b0;
        y = 1'b0;
        // Fuzzy test
        for (i = 0; i < 10; i=i+1) begin
            // Reset numbers
            #20 reset = 1'b1;
            temp_x = 0;
            temp_y = 0;
            #10
```

```
38              reset = 1'b0;
39              // Create a 4 bit number
40              for (j = 0; j < 4; j=j+1) begin
41                  x = $random % 2;
42                  y = $random % 2;
43                  temp_x = (temp_x << 1) + x;
44                  temp_y = (temp_y << 1) + y;
45                  #1
46                  $display("x=%d y=%d gt=%d lt=%d", temp_x, temp_y, gt, lt
    );
47                  #9
48                  temp_x = temp_x; // no op
49              end
50          end
51          $stop;
52      end
53  endmodule
```

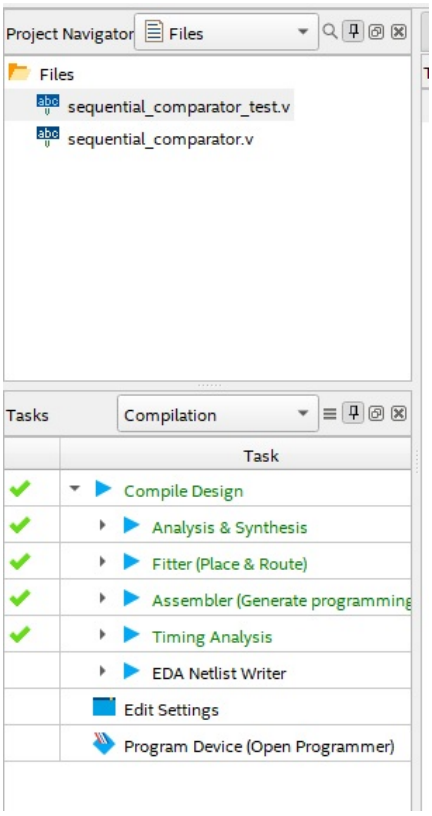
ممکن است که کمی خط ۴۵ تا ۴۸ گنگ باشد که آن‌را در اینجا توضیح می‌دهیم. در ابتدا نیاز داریم که کمی تاخیر داشته باشیم تا اینکه feedback مدار بر روی خروجی بتواند تاثیر بگذارد. بعد در خط ۴۶ نتیجه‌ی مدار را چاپ می‌کنیم. حال باید دوباره کمی صبر کنیم در غیر این صورت خط ۴۱ تا ۴۴ بلافاصله اجرا می‌شوند و x و y را عوض می‌کند. (این زبان توصیف سخت افزار است و خط‌ها همزمان اعمال می‌شوند!) پس باید یک تعلیق دیگر نیز داشته باشیم. اما نمی‌توان یک تعلیق خالی داشت. برای همین خطی را مثل خط ۴۸ می‌گذاریم که کاری نمی‌کند. (no-op)

حال برنامه را اجرا می‌کنیم و نتیجه‌ی مدار را مشاهده می‌کنیم.

```
x=0 y=1 gt=0 lt=1
x=1 y=3 gt=0 lt=1
x=3 y=7 gt=0 lt=1
x=7 y=14 gt=0 lt=1
x=1 y=1 gt=0 lt=0
x=2 y=3 gt=0 lt=1
x=5 y=6 gt=0 lt=1
x=11 y=12 gt=0 lt=1
x=1 y=0 gt=1 lt=0
x=3 y=1 gt=1 lt=0
x=6 y=3 gt=1 lt=0
x=12 y=6 gt=1 lt=0
x=0 y=1 gt=0 lt=1
x=0 y=3 gt=0 lt=1
```

```
x=1 y=7 gt=0 lt=1
x=3 y=14 gt=0 lt=1
x=0 y=0 gt=0 lt=0
x=0 y=1 gt=0 lt=1
x=0 y=3 gt=0 lt=1
x=1 y=7 gt=0 lt=1
x=1 y=1 gt=0 lt=0
x=2 y=2 gt=0 lt=0
x=5 y=5 gt=0 lt=0
x=11 y=10 gt=1 lt=0
x=0 y=0 gt=0 lt=0
x=0 y=0 gt=0 lt=0
x=1 y=0 gt=1 lt=0
x=2 y=1 gt=1 lt=0
x=1 y=0 gt=1 lt=0
x=2 y=0 gt=1 lt=0
x=4 y=0 gt=1 lt=0
x=9 y=1 gt=1 lt=0
x=1 y=1 gt=0 lt=0
x=3 y=2 gt=1 lt=0
x=6 y=5 gt=1 lt=0
x=13 y=11 gt=1 lt=0
x=0 y=0 gt=0 lt=0
x=1 y=1 gt=0 lt=0
x=2 y=3 gt=0 lt=1
x=5 y=6 gt=0 lt=1
```

در نهایت کد را سنتز می‌کنیم. همان طور که مشاهده می‌شود کد کامپایل می‌شود و مشکلی ندارد.



شکل ۵: سنتز مقایسه کننده‌ی سریال

نتیجه‌گیری

در این سری آزمایش توانستیم به کمک `assign` جریان یک سیم را با توجه به یک سری سیم دیگر مشخص کنیم. همچنین توانستیم فقط به کمک این دستور d flip flop و مدارهای ترتیبی بسازیم.