به نام خدا

آزمایش نهم

آزمایشگاه طراحی سیستم‌های دیجیتال

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

نویسندگان:

رادین چراغی ۴۰۱۱۰۵۸۱۵

امیرمحمد محفوظی ۴۰۱۱۰۶۴۶۹

سیدعلی جعفری ۴۰۰۱۰۴۸۸۹

تاریخ ارائه تکلیف:

۱۴۰۳/۰۵/۱۵

مقدمه

در این آزمایش می‌خواهیم یک پردازنده با معماری پشته‌ای طراحی و پیاده‌سازی کنیم. پشته این پردازنده معادل با ۸ ثبات ۸ بیتی می‌باشد. این پردازنده دارای ۸ دستور در مجموعه دستورالعمل های خود است و حافظه آن ۲۵۶ خانه ۸ بیتی دارد. بایستی دقت شود که ۸ خانه آخر آن (یعنی آدرس F8 تا FF) برای I/O رزرو شده‌اند. بنابراین ارتباط پردازنده با واحدهای I/O از طریق امکان Memory Mapped I/O پردازنده انجام می‌گیرد. در نهایت قرار است با استفاده از این پردازنده بتوانیم $Y = ((X + 23) * 2) - 12$ را محاسبه نماییم.

شرح آزمایش

ابتدا به بررسی دستورات این کامپیوتر می‌پردازیم. تصویر زیر دستورات موجود در این پردازنده را توضیح می‌دهد.

0000 PUSHC C

این دستور مقدار ثابت (Constant) ۸ بیتی C را در پشته PUSH می‌کند.

0001 PUSH M

این دستور مقدار خانه حافظه (یا درگاه) که با آدرس M (آدرس ۸ بیتی) مشخص شده است را خوانده و در پشته PUSH می‌کند.

0010 POP M

مقدار را از پشته POP کرده و آن را در خانه حافظه با آدرس M قرار می‌دهد (یا به درگاه با آدرس M ارسال می‌کند).

0011 JUMP

از پشته POP کرده و در PC قرار می‌دهد.

0100 JZ

اگر پرچم Z برابر 1 باشد از پشته POP کرده و در PC قرار می‌دهد.

0101 JS

اگر پرچم S برابر 1 باشد از پشته POP کرده و در PC قرار می‌دهد.

0110 ADD

دو داده بالای پشته را POP کرده با هم جمع کرده و حاصل را در بالای پشته PUSH می‌کند.

0111 SUB

دو داده بالای پشته را POP کرده عمل تفریق را برروی آن‌ها انجام می‌دهد و حاصل را در بالای پشته PUSH می‌کند.

همانطور که از دستورات فوق مشخص است این پردازنده دارای دو پرچم S و Z می‌باشد که به ترتیب نشان‌دهنده صفر بودن و منفی بودن حاصل آخرین جمع یا تفریق انجام شده توسط پردازنده می‌باشد. بنابراین این دو پرچم تنها با دستورات ADD و SUB تغییر خواهند کرد. همچنین محاسبات تماماً علامت دار و با استفاده از مکمل ۲ انجام می گیرند.

حال ماژول StackBasedProcessor را طراحی می‌کنیم. ورودی‌ها و خروجی‌های ماژول عبارتند از:

ورودی‌ها

- clk: همان کلاک کلی مدار می‌باشد.
- rstN: ریست مدار بوده که فعال پایین (active low) است.
- data_in: ورودی ۸ بیتی مدار بوده که در خانه‌ای از حافظه قرار خواهد گرفت. در ادامه نحوه پیاده‌سازی مدار به صورت Memory Mapped I/O توضیح داده خواهد شد.

خروجی‌ها

- data_out: خروجی ۸ بیتی مدار بوده که در خانه‌ای از حافظه قرار خواهد گرفت. در ادامه نحوه پیاده‌سازی مدار به صورت Memory Mapped I/O توضیح داده خواهد شد.

ابتدا ماژول را ساخته و ورودی‌ها و خروجی‌های ماژول را مشخص می‌کنیم. سپس آرایه رجیستری stack را طراحی می‌کنیم. همانطور که گفته عمق و پهنای این آرایه ۸ می‌باشد. همچنین رجیستر ۳ بیتی stack_pointer را تعریف می‌کنیم که همواره به بالای پشته (اولین خانه‌ی خالی) اشاره می‌کند.

سپس نوبت به طراحی حافظه‌های پردازنده می‌رسد. در هنگام پیاده‌سازی این پردازنده حافظه‌های داده و دستورات را از یکدیگر جدا در نظر می‌گیریم. بدین منظور ابتدا آرایه D_mem (حافظه داده) را با عمق ۲۵۶ و پهنای ۸ طراحی می‌کنیم. در ادامه آرایه رجیستری I_mem را با عمق ۳۲ و پهنای ۱۲ طراحی می‌کنیم. همانطور که مشخص است دستورات این پردازنده ۱۲ بیتی می‌باشند که ۴ بیت اول آن opcode و ۸ بیت بعدی آن آدرس یا immediate data می‌باشند. به همین دلیل پهنای I_mem را ۱۲ در نظر می‌گیریم. همچنین رجیستر ۵ بیتی program_counter را تعریف می‌کنیم که همواره به نگهدارنده آدرس دستور بعد می‌باشد.

همانطور که گفته شده این پردازنده دارای دو پرچم S و Z می‌باشد که به ترتیب نشان‌دهنده صفر بودن و منفی بودن حاصل آخرین جمع یا تفریق انجام شده توسط پردازنده می‌باشد. بنابراین در ادامه دو رجیستر SIGN و ZERO تعریف می‌کنیم و آن‌ها را با صفر مقداردهی اولیه می‌کنیم.

سپس دو وایر ۴ بیتی opcode و ۸ بیتی value را تعریف کرده که نگهدارنده دو قسمت اصلی دستور فعلی می‌باشند. همچنین دو وایر ۸ بیتی add_res و sub_res را تعریف می‌کنیم که به ترتیب محتوی حاصل جمع و تفریق دو عدد بالای پشته می‌باشند.

تصویر زیر عملیات بالا را نشان می‌دهد.

```
module StackBasedProcessor
(
    input clk, rstN,
    input [7:0] data_in,
    output [7:0] data_out
);
    //Stack
    reg [7:0] stack [7:0];
    reg [2:0] stack_pointer;

    //Data memory
    reg [7:0] D_mem [255:0];

    //Instruction memory
    reg [0:11] I_mem [31:0];
    reg [4:0] program_counter;

    //flags
    reg SIGN = 0, ZERO = 0;

    //Instruction parts
    wire [3:0] opcode;
    wire [7:0] value;

    //add/sub results
    wire [7:0] add_res, sub_res;
```

حال پارامترهای برای مشخص کردن نوع هر دستور تعریف کرده و مقدار هر پارامتر را برابر با opcode دستوری که آن را مشخص می‌کند قرار می‌دهیم. سپس دو وایر opcode و value را به ترتیب برابر با ۴ بیت اول و ۸ بیت آخر دستور فعلی قرار می‌دهیم. دستور فعلی با I_mem[program_counter] مشخص می‌شود. همچنین دو وایر add_res و sub_res را نیز برابر با مقدار مناسب قرار می‌دهیم.

همانطور که گفته شد، ارتباط پردازنده با واحدهای I/O از طریق امکان Memory Mapped I/O پردازنده انجام می‌گیرد. بدین منظور دو رجیستر addr_input و addr_output را تعریف می‌کنیم و به ترتیب برابر با ۲۵۴ و ۲۵۵ قرار می‌دهیم. سپس خانه‌ای از حافظه که آدرس addr_output را دارد به data_out، assign می‌کنیم.

تصویر زیر عملیات بالا را نشان می‌دهد.

```
//Instructions
parameter PUSHC = 0;
parameter PUSH = 1;
parameter POP = 2;
parameter JUMP = 3;
parameter JZ = 4;
parameter JS = 5;
parameter ADD = 6;
parameter SUB = 7;

assign opcode = I_mem[program_counter][0:3];
assign value = I_mem[program_counter][4:11];
assign add_res = stack[stack_pointer - 1] + stack[stack_pointer - 2];
assign sub_res = stack[stack_pointer - 2] - stack[stack_pointer - 1];

//Memory mapped I/O
reg [7:0] addr_output = 255;
reg [7:0] addr_input = 254;
assign data_out = D_mem[addr_output];

integer i;
```

در ادامه بخش ترتیبی مدار را طراحی می‌کنیم. برای این کار از یک بلاک always استفاده کرده که حساس به لبه بالارونده clk و لبه پایین‌رونده rstN می‌باشد. در داخل بلاک در صورتی که rstN صفر باشد، مدار ریست شده و پرچم‌ها، program_counter، stack_pointer و خانه‌های stack صفر خواهند شد. تصویر زیر عملیات بالا را نشان می‌دهد.

```
always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        SIGN = 0; ZERO = 0; program_counter = 0; stack_pointer  = 0;
        for (i = 0; i < 8; i = i + 1) begin
            stack[i] = 0;
        end
    end
```

در صورتی که rstN برابر با یک باشد، ابتدا program_counter را با یک جمع کرده و سپس data_in را در خانه‌ای از حافظه با آدرس addr_input قرار می‌دهیم. در ادامه با استفاده از case دستور فعلی را بررسی می‌کنیم.

PUSHC: مقدار value در پشته push شده و stack_pointer یک واحد افزایش می‌یابد.

PUSH: مقدار موجود در خانه‌ای با آدرس value از حافظه در پشته push شده و stack_pointer یک واحد افزایش می‌یابد.

POP: stack_pointer یک واحد کاهش می‌یابد و مقدار موجود در بالای پشته در خانه‌ای با آدرس value از حافظه قرار می‌گیرد.

JUMP: stack_pointer یک واحد کاهش می‌یابد و مقدار بالای پشته در program_counter قرار می‌گیرد.

JZ & JS: در صورت یک بودن مقدارZERO یا SIGN (بسته به نوع دستور،) stack_pointer یک واحد کاهش می‌یابد و مقدار بالایی پشته در pc قرار می‌گیرد.

ADD & SUB: مقدار جمع یا تفریق دو خانه بالای پشته (بسته به نوع دستور،) در خانه بالای پشته قرار گرفته و در مجموع stack_pointer یک واحد کاهش می‌یابد. سپس در صورتی که حاصل در صورتی که حاصل صفر یا منفی شود به ترتیب پرچم‌های ZERO و SIGN به یک مقداردهی می‌شوند.

تصاویر زیر عملیات بالا را نشان می‌دهند.

```
else begin
    program_counter = program_counter + 1;
    D_mem[addr_input]  = data_in;
    case (opcode)
        PUSHC: begin
            stack[stack_pointer] = value;
            stack_pointer = stack_pointer + 1;
        end
        PUSH: begin
            stack[stack_pointer] = D_mem[value];
            stack_pointer = stack_pointer + 1;
        end
        POP: begin
            stack_pointer = stack_pointer - 1;
            D_mem[value] = stack[stack_pointer];
        end
        JUMP: begin
            stack_pointer = stack_pointer - 1;
            program_counter = stack[stack_pointer];
        end
        JZ: begin
            if(ZERO) begin
                stack_pointer = stack_pointer - 1;
                program_counter = stack[stack_pointer];
            end
        end
```

```
        JS: begin
            if(SIGN) begin
                stack_pointer = stack_pointer - 1;
                program_counter = stack[stack_pointer];
            end
        end
        ADD: begin
            stack_pointer = stack_pointer - 2;
            stack[stack_pointer] = add_res;
            stack_pointer = stack_pointer + 1;
            if(add_res == 0) ZERO = 1'b1;
            if($signed(add_res) < 0) SIGN = 1'b1;
        end
        SUB: begin
            stack_pointer = stack_pointer - 2;
            stack[stack_pointer] = sub_res;
            stack_pointer = stack_pointer + 1;
            if(sub_res == 0) ZERO = 1'b1;
            if($signed(sub_res) < 0) SIGN = 1'b1;
        end
    endcase
end
```

حال ماژول InfixCalculator را طراحی می‌کنیم. در این ماژول بایستی حاصل Y = ((X + 23) * 2) – 12 را با استفاده از پردازنده محاسبه کنیم. ابتدا برای سادگی یک ماکرو به صورت زیر تعریف می‌کنیم که هر بار استفاده از آن بیانگر قرار گیری یک دستور پردازنده در حافظه می‌باشد.

```
`define instruction(address, opcode, value = 0)    processor.I_mem[address] = (opcode << 8) | value
```

حال ورودی‌ها و خروجی‌های این ماژول را مشخص می‌کنیم. ورودی‌ها م خروجی‌های این ماژول همان ورودی‌ها و خروجی‌ها ماژول قبل می‌باشد با این تفاوت که خروجی Exception نیز به این ماژول اضافه شده که هرگاه ورودی عدد منفی باشد و یا مقدار خروجی از حوزه قابل نمایش خارج شود (بزرگتر از ۱۲۷ باشد) فعال می‌شود.

ابتدا از ماژول StackBasedProcessor نمونه گرفته و سپس Exception را با توجه به توضیحاتت بالا مقداردهی می‌کنیم. همچنین پارامتر tmp_var را تعریف می‌کنیم و مقدار آن را برابر با صفر می‌گذاریم. این پارامتر به خانه‌ی اول حافظه اشاره می‌کند که به عنوان متغیر کمکی از آن استفاده می‌کنیم. سپس با استفاده از بلاک initial دستورات مربوط به محاسبه Y = ((X + 23) * 2) – 12 را به حافظه دستورات پردازنده اضافه می‌کنیم. این دستورات عبارتند از:

1. PUSH addr_input
2. PUSHC 23
3. ADD
4. PUSH temp_var
5. PUSH temp_var
6. ADD
7. PUSHC 12
8. SUB
9. POP addr_ouput

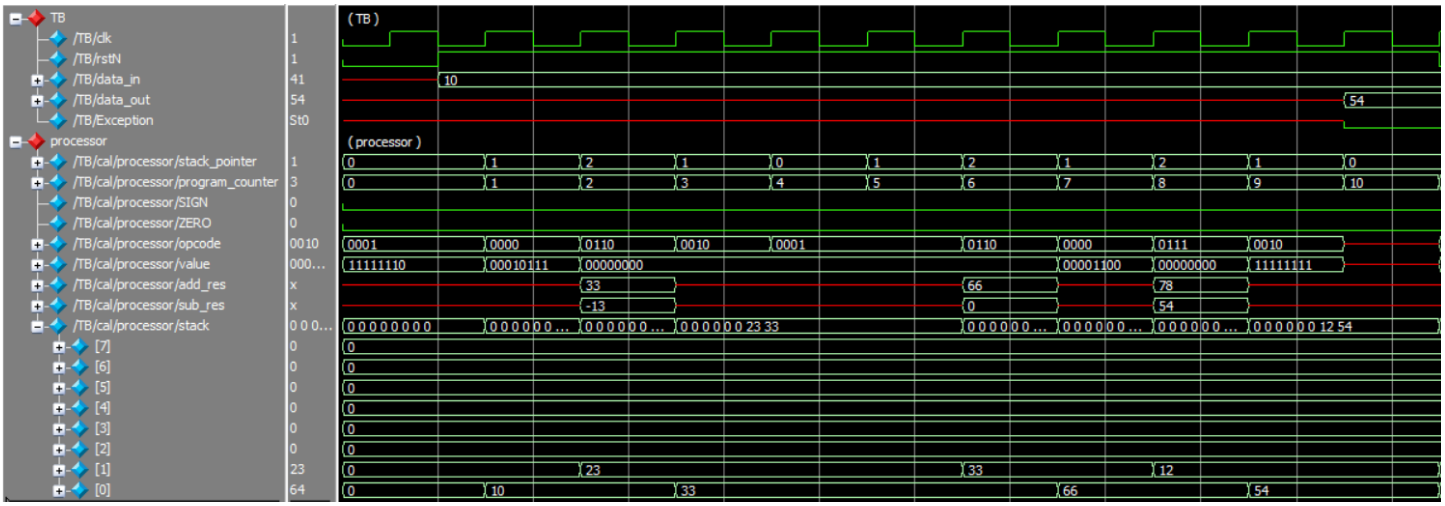
همانطور که مشخص است با استفاده از دستورات بالا Y = ((X + 23) * 2) – 12 محاسبه می‌شود.

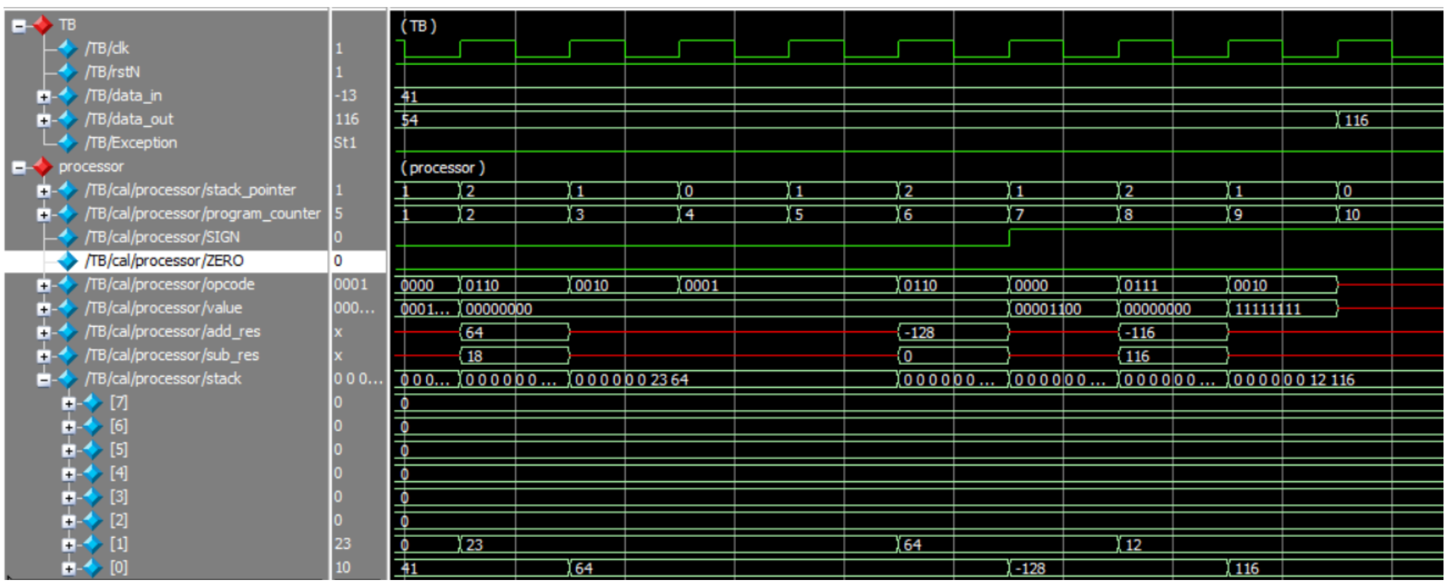
```
initial begin
    `instruction(0, processor.PUSH, processor.addr_input);
    `instruction(1, processor.PUSHC,   23);
    `instruction(2, processor.ADD);
    `instruction(3, processor.POP, temp_var);
    `instruction(4, processor.PUSH, temp_var);
    `instruction(5, processor.PUSH, temp_var);
    `instruction(6, processor.ADD);
    `instruction(7, processor.PUSHC, 12);
    `instruction(8, processor.SUB);
    `instruction(9, processor.POP, processor.addr_output);
end
```

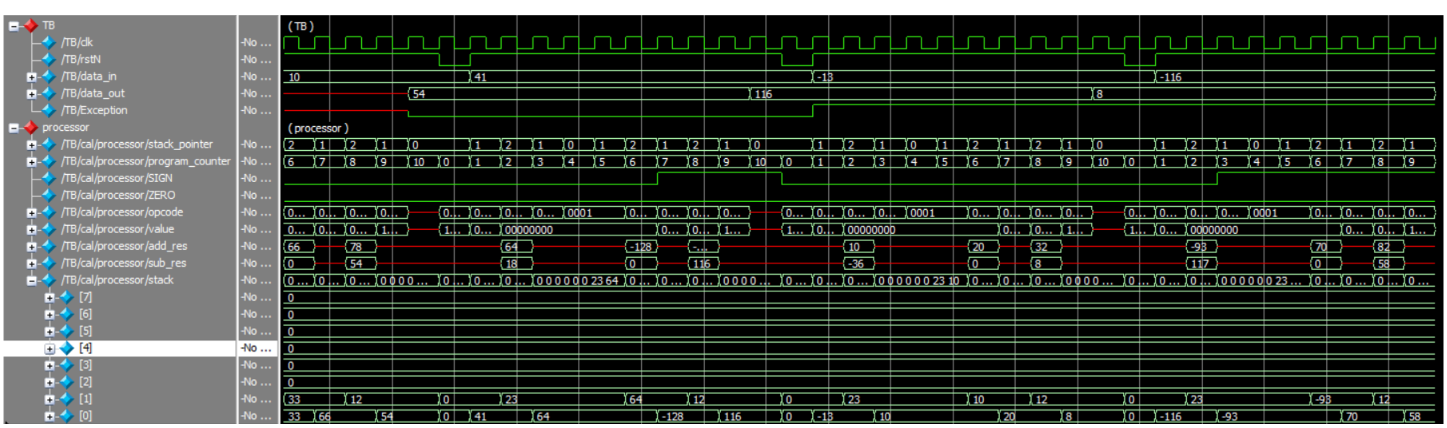
حال برای این ماژول یک تست بنچ طراحی می‌کنیم. پس از تعریف سیگنال‌ها و رجیسترهای لازم برای نمونه گیری از ماژول InfixCalculator، با استفاده از بلاک initial و always کلاک را شبیه‌سازی می‌کنیم. سپس در یک بلاک initial دیگر ۴ تست مدار را قرار می‌دهیم. در دو تست آخر ورودی X را به گونه‌ای قرار داده‌ایم تا به Exception بربخوریم.

```
module TB;

    reg clk, rstN;
    reg [7:0] data_in;
    wire [7:0] data_out;
    wire Exception;

    InfixCalculator cal(clk, rstN, data_in, data_out, Exception);

    initial clk = 0;
    always #5 clk = ~clk;

    initial begin
        rstN = 0;
        #10
        rstN = 1;
        data_in = 10;
        wait(cal.processor.program_counter == 10);
        $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(data_in), $signed(data_out), Exception);

        #10
        rstN = 0;
        #10
        rstN = 1;
        data_in = 41;
        wait(cal.processor.program_counter == 10);
        $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(data_in), $signed(data_out), Exception);

        #10
        rstN = 0;
        #10
        rstN = 1;
        data_in = -13;
        wait(cal.processor.program_counter == 10);
        $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(data_in), $signed(data_out), Exception);
        #10
        rstN = 0;
        #10
        rstN = 1;
        data_in = 140;
        wait(cal.processor.program_counter == 10);
        $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(data_in), $signed(data_out), Exception);

        $stop;
    end

endmodule
```

حال با استفاده از نرم‌افزار ModelSim مدار را شبیه‌سازی می‌کنیم. تصاویر زیر خروجی waveform و transcript را نشان می‌دهند.


TB
/TB/clk
/TB/rstN
/TB/data_in
/TB/data_out
/TB/Exception
processor
/TB/cal/processor/stack_pointer
/TB/cal/processor/program_counter
/TB/cal/processor/SIGN
/TB/cal/processor/ZERO
/TB/cal/processor/opcode
/TB/cal/processor/value
/TB/cal/processor/add_res
/TB/cal/processor/sub_res
/TB/cal/processor/stack


TB
/TB/clk
/TB/rstN
/TB/data_in
/TB/data_out
/TB/Exception
processor
/TB/cal/processor/stack_pointer
/TB/cal/processor/program_counter
/TB/cal/processor/SIGN
/TB/cal/processor/ZERO
/TB/cal/processor/opcode
/TB/cal/processor/value
/TB/cal/processor/add_res
/TB/cal/processor/sub_res
/TB/cal/processor/stack


TB
/TB/clk
/TB/rstN
/TB/data_in
/TB/data_out
/TB/Exception
processor
/TB/cal/processor/stack_pointer
/TB/cal/processor/program_counter
/TB/cal/processor/SIGN
/TB/cal/processor/ZERO
/TB/cal/processor/opcode
/TB/cal/processor/value
/TB/cal/processor/add_res
/TB/cal/processor/sub_res
/TB/cal/processor/stack

```
VSIM 31> run -all
# ((  10 + 23) * 2) - 12 =   54, error = 0
# ((  41 + 23) * 2) - 12 =  116, error = 0
# (( -13 + 23) * 2) - 12 =    8, error = 1
# ((-116 + 23) * 2) - 12 =   58, error = 1
# ** Note: $stop    : C:/Users/ideapad 5/Desktop/E10/TB.v(45)
#    Time: 435 ps  Iteration: 2  Instance: /TB
```

خروجی **flow summary**

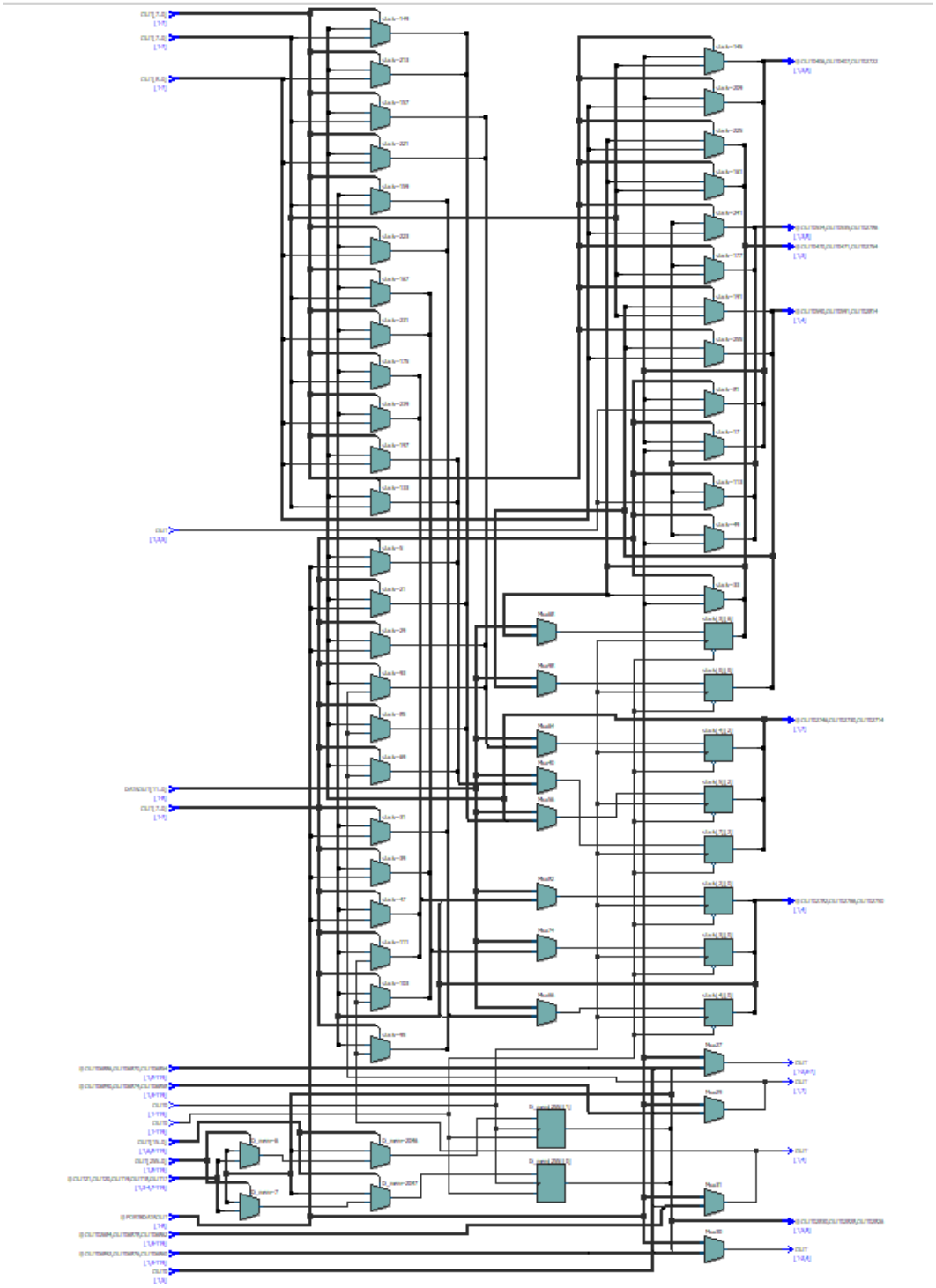
تصاویر زیر خروجی flow summary ماژول‌های StackBasedProcessor و InfixCalculator را نشان می‌دهد. در هنگام سنتز مجبور به تغییر اندکی در کد ماژول‌ها شدیم. این تغیر جابجایی بلاک initial موجود در ماژول InfixCalculator به ماژول StackBasedProcessor می‌باشد.

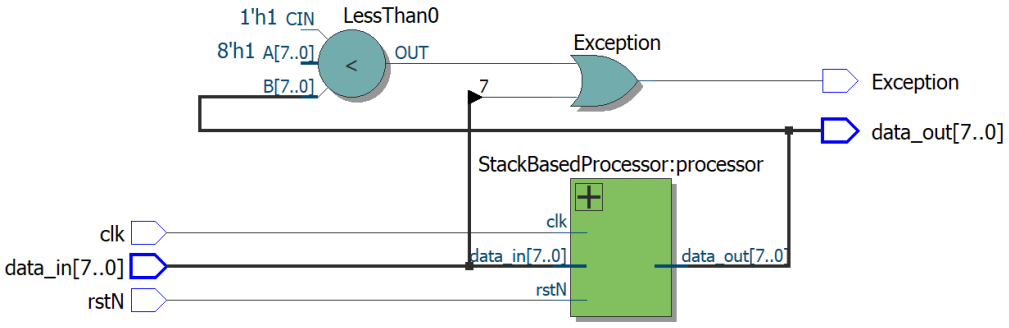
Flow Summary	
Flow Status	Successful - Mon Aug 05 20:48:39 2024
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	processor
Top-level Entity Name	StackBasedProcessor
Family	Cyclone IV GX
Total logic elements	512 / 14,400 (4 %)
Total combinational functions	380 / 14,400 (3 %)
Dedicated logic registers	322 / 14,400 (2 %)
Total registers	322
Total pins	18 / 81 (22 %)
Total virtual pins	0
Total memory bits	0 / 552,960 (0 %)
Embedded Multiplier 9-bit elements	0
Total GXB Receiver Channel PCS	0 / 2 (0 %)
Total GXB Receiver Channel PMA	0 / 2 (0 %)
Total GXB Transmitter Channel PCS	0 / 2 (0 %)
Total GXB Transmitter Channel PMA	0 / 2 (0 %)
Total PLLs	0 / 3 (0 %)
Device	EP4CGX15BF14C6
Timing Models	Final

Flow Summary	
Flow Status	Successful - Mon Aug 05 20:52:06 2024
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	processor
Top-level Entity Name	InfixCalculator
Family	Cyclone IV GX
Total logic elements	517 / 14,400 (4 %)
Total combinational functions	377 / 14,400 (3 %)
Dedicated logic registers	322 / 14,400 (2 %)
Total registers	322
Total pins	19 / 81 (23 %)
Total virtual pins	0
Total memory bits	0 / 552,960 (0 %)
Embedded Multiplier 9-bit elements	0
Total GXB Receiver Channel PCS	0 / 2 (0 %)
Total GXB Receiver Channel PMA	0 / 2 (0 %)
Total GXB Transmitter Channel PCS	0 / 2 (0 %)
Total GXB Transmitter Channel PMA	0 / 2 (0 %)
Total PLLs	0 / 3 (0 %)
Device	EP4CGX15BF14C6
Timing Models	Final

خروجی RTL Viewer

تصاویر زیر خروجی RTL Viewer ماژول‌های StackBasedProcessor و InfixCalculator را نشان می‌دهد. فایل pdf نیز در پیوست موجود است.




1'h1 CIN
8'h1 A[7..0]
B[7..0]
LessThan0
OUT
<
Exception
7
Exception
data_out[7..0]
StackBasedProcessor:processor
clk
data_in[7..0]
rstN
data_out[7..0]
clk
data_in[7..0]
rstN