به نام خدا

محمدفرحان بهرامی

401105729

پروژه امتیازی DSD

سوال هفتم میانترم

الف)

ماژول stack: در ابتدا، پشته از پارامترهای N و MAX_SIZE استفاده می‌کند تا اندازه بیت‌های داده و حداکثر تعداد عنصرها رو مشخص کند. سیگنال‌های ورودی شامل کلاک(clk) ، ریست(rst) ، سیگنال پوش و پاپ و همچنین داده‌های ورودی (data_in) هستند. سیگنال‌های خروجی شامل داده‌های خروجی(data_out) ، پر بودن استک (full) و خالی بودن استک (empty) هستند.

استک از یک آرایه به نام memory برای ذخیره داده‌ها استفاده می‌کند که طول آن برابر با MAX_SIZE است. یک اندیس به نام top_indexوجود دارد که به عنوان پوینتر به بالای استک عمل می‌کند. در ابتدا، top_indexبرابر با صفر است و استک خالی است empty برابر با 1 و پر نیست full برابر با 0.

یک بلوک always وجود دارد که با هر کلاک مثبت یا ریست فعال می‌شود. اگر ریست فعال باشد، top_indexبه صفر تنظیم شده و استک خالی و پر نبودن تنظیم می‌شود. اگر سیگنال push فعال باشد و استک پر نباشد، داده ورودی در بالای استک قرار می‌گیرد و top_index یک واحد افزایش می‌یابد. اگر top_index به حداکثر اندازه استک برسد، سیگنال full فعال می‌شود و سیگنال emptyغیرفعال می‌گردد. اگر سیگنال pop فعال باشد و استک خالی نباشد، top_indexیک واحد کاهش می‌یابد و داده بالای استک به خروجی داده منتقل می‌شود. اگر top_index به صفر برسد، سیگنال empty فعال می‌شود و سیگنال full غیرفعال می‌گردد.

```
10      input wire signed [N-1:0] data_in,
11      output reg signed [N-1:0] data_out,
12      output reg full,
13      output reg empty
14  );
15      localparam LOG2_MAX_SIZE = $clog2(MAX_SIZE);
16      reg signed [N-1:0] memory [0:MAX_SIZE-1];
17      reg [LOG2_MAX_SIZE-1:0] top_index;
18      initial begin
19          top_index = 0;
20          full = 0;
21          empty = 1;
22          data_out = 0;
23      end
24      always @(posedge clk or posedge rst) begin
25          if (rst) begin
26              top_index <= 0;
27              full <= 0;
28              empty <= 1;
29          end else begin
30              if (push && !full) begin
31                  memory[top_index] <= data_in;
32                  top_index <= top_index + 1;
33                  if (top_index + 1 == MAX_SIZE) begin
34                      full <= 1;
35                  end
36                  empty <= 0;
37              end else if (pop && !empty) begin
38                  top_index <= top_index - 1;
39                  data_out <= memory[top_index - 1];
40                  if (top_index - 1 == 0) begin
41                      empty <= 1;
42                  end
43                  full <= 0;
44              end
45          end
46      end
```

ماژول STACK_BASED_ALU: ورودی‌های این ماژول شامل سیگنال کلاک(clk) ، ریست(rst) ، (opcode) و داده‌های ورودی (input_data)هستند. خروجی‌های ماژول شامل داده‌های خروجی(output_data) ، سیگنال (overflow) و سیگنال (success) می‌باشند. این ماژول از ماژول دیگری به نام Stack نمونه میگیرد و سیگنال های مربوطه را به این نمونه می دهیم.

اگر کد عملیاتی برابر با 3 'b110باشد، عملیات (push) انجام می‌شود. اگر استک پر نباشد، داده‌های ورودی به استک پوش می‌شوند و سیگنال موفقیت فعال می‌شود. اگر کد عملیاتی برابر با 3 'b111باشد، عملیات پاپ انجام می‌شود. اگر استک خالی نباشد، داده‌های بالای استک به خروجی منتقل می‌شوند و سیگنال موفقیت فعال می‌شود. اگر کد عملیاتی برابر با 3 'b100باشد، عملیات جمع انجام می‌شود و نتیجه به خروجی منتقل می‌شود. اگر اورفلو رخ دهد، سیگنال overflow فعال می‌شود. اگر کد عملیاتی برابر با 3 'b101 باشد، عملیات ضرب انجام می‌شود. نتیجه کامل و نتیجه کوتاه شده به دست می‌آیند. اگر نتیجه کامل با نتیجه کوتاه شده برابر نباشد، سیگنال overflow فعال می‌شود.

عکس در صفحات بعدی میباشد...

```
module STACK_BASED_ALU #(parameter N, parameter MAX_SIZE) (
    input wire clk,
    input wire rst,
    input wire [2:0] opcode,
    input wire signed [N-1:0] input_data,
    output reg signed [N-1:0] output_data,
    output reg overflow,
    output reg success
);

    wire full, empty;
    reg stack_clk, stack_push, stack_pop;
    reg signed [N-1:0] stack_in, top, top_next;
    wire signed [N-1:0] stack_out;

    reg signed [N-1:0] truncated_result;
    reg signed [2*N-1:0] result;

    Stack#(N, MAX_SIZE) stack_inst (
        .clk(stack_clk),
        .rst(rst),
        .push(stack_push),
        .pop(stack_pop),
        .data_in(stack_in),
        .data_out(stack_out),
        .full(full),
        .empty(empty)
    );

    initial begin
        stack_clk = 0;
        stack_push = 0;
        stack_pop = 0;
    end

    always @(posedge clk) begin
        success = 0;
        overflow = 0;
        output_data = 'bz;

        if (opcode == 3'b110 && !full) begin
```

```
            stack_push = 1;
            stack_in = input_data;
            stack_clock_pulse();
            stack_push = 0;
            success = 1;

        end else if (opcode == 3'b111 && !empty) begin
            stack_pop = 1;
            stack_clock_pulse();
            output_data = stack_out;
            stack_pop = 0;
            success = 1;

        end else if ((opcode == 3'b100 || opcode == 3'b101) && !empty) begin
            perform_stack_operation(opcode);
        end
    end

    task stack_clock_pulse();
        begin
            #1 stack_clk = 1;
            #1 stack_clk = 0;
        end
    endtask

    task perform_stack_operation(input [2:0] operation);
        begin
            stack_pop = 1;
            stack_clock_pulse();
            top = stack_out;
            stack_pop = 0;

            if (!empty) begin
                stack_pop = 1;
                stack_clock_pulse();
                top_next = stack_out;
                stack_pop = 0;
                success = 1;

                if (operation == 3'b100) begin
                    perform_addition();
                end else if (operation == 3'b101) begin
```

```
                    perform_addition();
                end else if (operation == 3'b101) begin
                    perform_multiplication();
                end

                push_to_stack(top_next);
                push_to_stack(top);
            end else begin
                push_to_stack(top);
            end
        end
    endtask

    task perform_addition();
        begin
            output_data = top + top_next;
            overflow = (~top[N-1] & ~top_next[N-1] & output_data[N-1]) | (top[N-1] & top_next[N-1] & ~output_data[N-1]);
        end
    endtask

    task perform_multiplication();
        begin
            result = top * top_next;
            truncated_result = top * top_next;
            output_data = truncated_result;
            if (result != truncated_result) begin
                overflow = 1;
            end
        end
    endtask

    task push_to_stack(input signed [N-1:0] data);
        begin
            stack_push = 1;
            stack_in = data;
            stack_clock_pulse();
            stack_push = 0;
        end
    endtask

endmodule
```

ماژول STACK_BASED_ALU_tb: این کد یک تست بنچ برای ماژول ALU_BASED_STACK است. در این کد، پارامترهای N و MAX_SIZEبه ترتیب اندازه داده‌ها و حداکثر اندازه استک را مشخص می‌کنند. متغیرهای مختلفی از جمله سیگنال‌های کلاک(clk) ، ریست(rst) ، کد عملیاتی(opcode) ، داده ورودی(input_data) ، و... را دارد.

در بخش همیشه(always) ، سیگنال کلاک هر 10 نانوثانیه تغییر وضعیت می‌دهد. در بخش اولیه(initial) ، سیگنال‌های کلاک و ریست مقداردهی اولیه می‌شوند و یک عدد برای تولید اعداد تصادفی تنظیم می‌شود. کد سه بار تکرار می‌شود، در هر تکرار چهار عدد تصادفی تولید می‌شوند و به استک افزوده می‌شوند. سپس عملیات جمع و ضرب روی دو عنصر بالای استک انجام می‌شود. در هر تکرار عملیات‌ها با تأخیر(20 نانوثانیه) انجام می‌شود. در ابتدا، داده‌های رندم تولید و به استک اضافه می‌شوند. سپس عملیات جمع دو عنصر بالای استک انجام می‌شود و نتیجه مورد انتظار با نتیجه واقعی مقایسه می‌شود. همچنین، دو عنصر بالای استک پاپ می‌شوند و مقدار واقعی پاپ شده. در نهایت، عملیات ضرب دو عنصر بالای استک انجام می‌شود.

نکته: برای تعیین کردن اندازه داده ها کافی است ما N را هر مقدار که بخواهیم مقدار دهی کنیم که ما در اینجا 4 بیتی در نظر میگیریم.

```
4
5   module STACK_BASED_ALU_tb;
6       parameter N = 4;
7       parameter MAX_SIZE = 1024;
8
9       reg clk;
10      reg rst;
11      reg [2:0] opcode;
12      reg signed [N-1:0] input_data;
13      reg signed [N-1:0] number1;
14      reg signed [N-1:0] number2;
15      reg signed [N-1:0] number3;
16      reg signed [N-1:0] number4;
17      reg signed [N-1:0] truncated_result;
18      reg signed [2*N-1:0] result;
19      integer seed, i;
20
21      wire signed [N-1:0] output_data;
22      wire overflow;
23      wire success;
24
25      STACK_BASED_ALU #(N, MAX_SIZE) uut (
26          .clk(clk),
27          .rst(rst),
28          .opcode(opcode),
29          .input_data(input_data),
30          .output_data(output_data),
31          .overflow(overflow),
32          .success(success)
33      );
34
35      always #10 clk = ~clk;
36
37      task initialize;
38          begin
39              clk = 0;
40              rst = 0;
41              seed = 123456;
42              $urandom(seed);
43          end
44      endtask
```

```
46      task generate_random_numbers;
47          output signed [N-1:0] num1, num2, num3, num4;
48          begin
49              num1 = 0;
50              num2 = 0;
51              num3 = 0;
52              num4 = 0;
53
54              for (i = 0; i < N; i = i + 1) begin
55                  num1[i] = $urandom % 2;
56                  num2[i] = $urandom % 2;
57                  num3[i] = $urandom % 2;
58                  num4[i] = $urandom % 2;
59              end
60              $display("Generated 4 random values: %d, %d, %d, %d", $signed(num1), $signed(num2), $signed(num3), $signed(num4));
61          end
62      endtask
63
64      task push_to_stack;
65          input signed [N-1:0] value;
66          begin
67              input_data = value;
68              opcode = 3'b110;
69              #20;
70              $display("Stack push: %d", $signed(input_data));
71          end
72      endtask
73
74      task pop_from_stack;
75          input signed [N-1:0] expected_value;
76          begin
77              opcode = 3'b111;
78              #20;
79              $display("Stack pop, expecting: %d", $signed(expected_value));
80              $display("Popped value: %d", $signed(output_data));
81          end
82      endtask
83
84      task perform_addition;
85          input signed [N-1:0] val1, val2;
86          begin
```

```
        begin
            result = val1 + val2;
            truncated_result = result;
            $display("Addition of top two stack values: %d + %d = %d (expected), truncated to %d bits = %d",
                     $signed(val1), $signed(val2), $signed(result), N, $signed(truncated_result));
            opcode = 3'b100;
            #20;
            $display("Addition result: %d, Overflow: %b", $signed(output_data), overflow);
        end
    endtask

    task perform_multiplication;
        input signed [N-1:0] val1, val2;
        begin
            result = val1 * val2;
            truncated_result = result;
            $display("Multiplication of top two stack values: %d * %d = %d (expected), truncated to %d bits = %d",
                     $signed(val1), $signed(val2), $signed(result), N, $signed(truncated_result));
            opcode = 3'b101;
            #20;
            $display("Multiplication result: %d, Overflow: %b", $signed(output_data), overflow);
        end
    endtask

    initial begin
        initialize;

        repeat (3) begin
            generate_random_numbers(number1, number2, number3, number4);

            push_to_stack(number1);
            push_to_stack(number2);
            push_to_stack(number3);
            push_to_stack(number4);

            perform_addition(number4, number3);

            pop_from_stack(number4);
            pop_from_stack(number3);

            perform_multiplication(number1, number2);
        end
```

نتیجه دیتای 4 بیتی که 3 بار تکرار شده بصورت زیر است:

```
VSIM 65> run -all
# Generated 4 random values:  -2,  -1,   4,   1
# Stack push:  -2
# Stack push:  -1
# Stack push:   4
# Stack push:   1
# Addition of top two stack values:   1 +   4 =    5 (expected), truncated to          4 bits =   5
# Addition result:   5, Overflow: 0
# Stack pop, expecting:   1
# Popped value:   1
# Stack pop, expecting:   4
# Popped value:   4
# Multiplication of top two stack values:  -2 *  -1 =    2 (expected), truncated to          4 bits =   2
# Multiplication result:   2, Overflow: 0
# Generated 4 random values:   3,   5,  -2,  -3
# Stack push:   3
# Stack push:   5
# Stack push:  -2
# Stack push:  -3
# Addition of top two stack values:  -3 +  -2 =   -5 (expected), truncated to          4 bits =  -5
# Addition result:  -5, Overflow: 0
# Stack pop, expecting:  -3
# Popped value:  -3
# Stack pop, expecting:  -2
# Popped value:  -2
# Multiplication of top two stack values:   3 *   5 =   15 (expected), truncated to          4 bits =  -1
# Multiplication result:  -1, Overflow: 1
# Generated 4 random values:  -2,  -8,   5,  -8
# Stack push:  -2
# Stack push:  -8
# Stack push:   5
# Stack push:  -8
# Addition of top two stack values:  -8 +   5 =   -3 (expected), truncated to          4 bits =  -3
# Addition result:  -3, Overflow: 0
# Stack pop, expecting:  -8
# Popped value:  -8
# Stack pop, expecting:   5
# Popped value:   5
# Multiplication of top two stack values:  -2 *  -8 =   16 (expected), truncated to          4 bits =   0
# Multiplication result:   0, Overflow: 1
# ** Note: $finish    : C:/Users/EPO/Desktop/DSD-Proj/Code/STACK_BASED_ALU_tb.v(129)
#    Time: 480 ns  Iteration: 0  Instance: /STACK_BASED_ALU_tb
# 1
```

ب)

ماژول infix2postfix: در بخش اولیه، سیگنال‌های مربوط به استک مقداردهی اولیه می‌شوند و استک ریست می‌شود. عملیات تبدیل عبارت اینفیکس به پست‌فیکس انجام می‌شود. یک حلقه از راست به چپ روی عبارت اینفیکس حرکت می‌کند و کاراکترها را بررسی می‌کند. اگر کاراکتر عددی باشد، به عبارت پست‌فیکس اضافه می‌شود. اگر کاراکتر پرانتز باز باشد، به پشته اضافه می‌شود. اگر کاراکتر پرانتز بسته باشد، کاراکترها از پشته خارج می‌شوند و به عبارت پست‌فیکس اضافه می‌شوند تا زمانی که به پرانتز باز برسیم. برای عملگرها، اگر استک خالی باشد، عملگر به استک اضافه می‌شود. در غیر این صورت، عملگرها از استک خارج شده و به عبارت پست‌فیکس اضافه می‌شوند تا زمانی که اولویت عملگر فعلی کمتر از عملگر درون استک باشد، سپس عملگر فعلی به استک اضافه می‌شود.

کد در صفحه بعد...

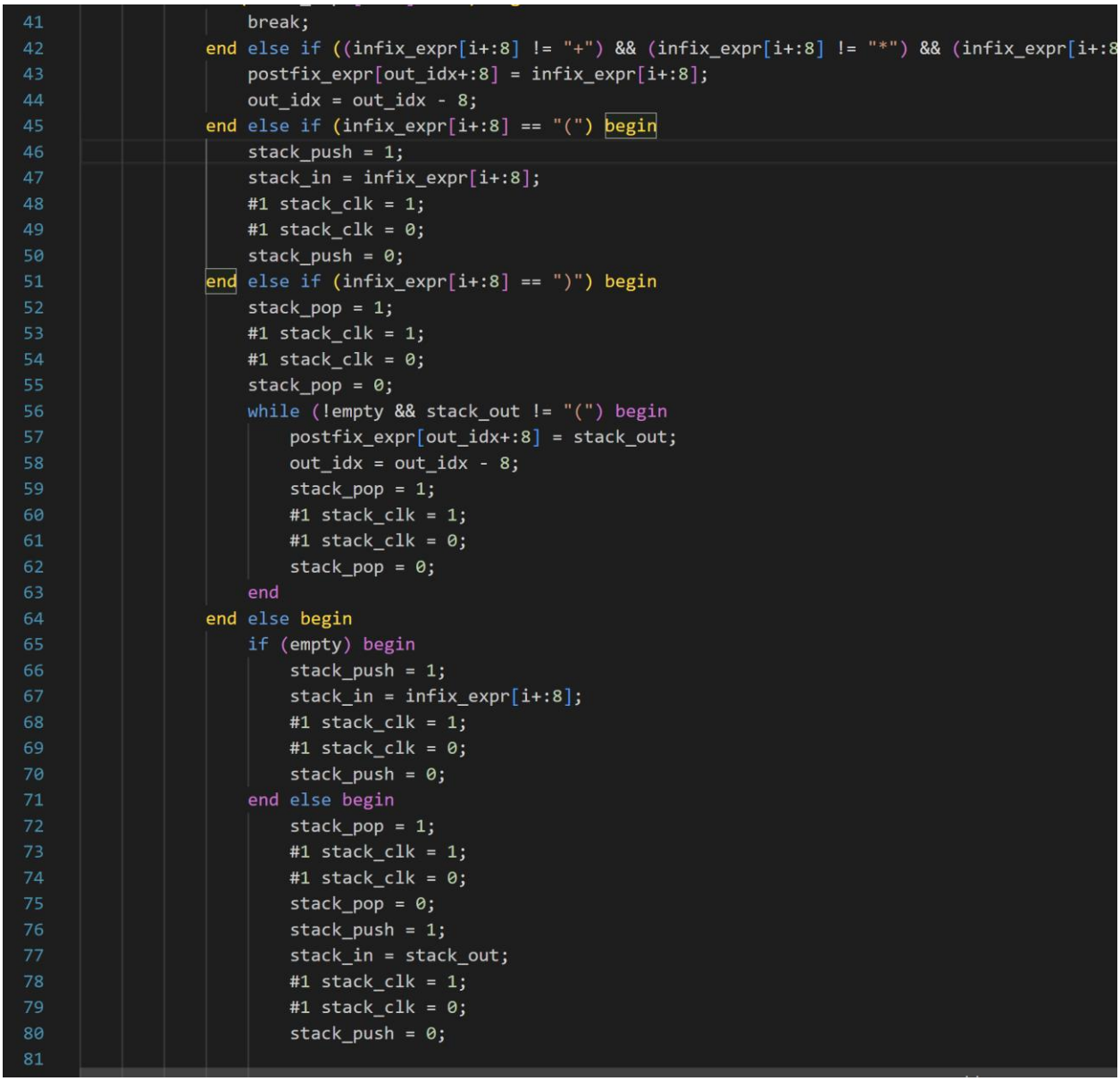
```
`timescale 1ns / 1ps
module infix2postfix #(parameter LEN = 16)(
    input reg [8*LEN-1:0] infix_expr,
    output reg [8*LEN-1:0] postfix_expr
);
    reg [7:0] stack_in;
    wire [7:0] stack_out;
    reg stack_push, stack_pop, stack_clk, rst;
    wire full, empty;
    integer out_idx;
    integer i;

    Stack#(8, LEN) stack_inst (
        .clk(stack_clk),
        .rst(rst),
        .push(stack_push),
        .pop(stack_pop),
        .data_in(stack_in),
        .data_out(stack_out),
        .full(full),
        .empty(empty)
    );

    initial begin
        stack_push = 0;
        stack_pop = 0;
        stack_clk = 0;
        rst = 1;
        #10 stack_clk = 1;
        #10 stack_clk = 0;
        out_idx = 8*LEN - 8;
        rst = 0;
    end

    initial begin
        postfix_expr = {LEN{"\0"}};

        #100
        for (i = 8*LEN - 8; i >= 0; i = i - 8) begin
            if (infix_expr[i+:8] == 0) begin
                break;
```



```
            break;
        end else if ((infix_expr[i+:8] != "+") && (infix_expr[i+:8] != "*") && (infix_expr[i+:8
            postfix_expr[out_idx+:8] = infix_expr[i+:8];
            out_idx = out_idx - 8;
        end else if (infix_expr[i+:8] == "(") begin
            stack_push = 1;
            stack_in = infix_expr[i+:8];
            #1 stack_clk = 1;
            #1 stack_clk = 0;
            stack_push = 0;
        end else if (infix_expr[i+:8] == ")") begin
            stack_pop = 1;
            #1 stack_clk = 1;
            #1 stack_clk = 0;
            stack_pop = 0;
            while (!empty && stack_out != "(") begin
                postfix_expr[out_idx+:8] = stack_out;
                out_idx = out_idx - 8;
                stack_pop = 1;
                #1 stack_clk = 1;
                #1 stack_clk = 0;
                stack_pop = 0;
            end
        end else begin
            if (empty) begin
                stack_push = 1;
                stack_in = infix_expr[i+:8];
                #1 stack_clk = 1;
                #1 stack_clk = 0;
                stack_push = 0;
            end else begin
                stack_pop = 1;
                #1 stack_clk = 1;
                #1 stack_clk = 0;
                stack_pop = 0;
                stack_push = 1;
                stack_in = stack_out;
                #1 stack_clk = 1;
                #1 stack_clk = 0;
                stack_push = 0;

```

```
                    stack_push = 0;

                    while (!empty && ((infix_expr[i+:8] == "+" && (stack_out == "+" || stack_out == "*")) || (infix_expr[i+:8] == "*" &&
                        stack_pop = 1;
                        #1 stack_clk = 1;
                        #1 stack_clk = 0;
                        stack_pop = 0;
                        postfix_expr[out_idx+:8] = stack_out;
                        out_idx = out_idx - 8;
                        if (empty) begin
                            break;
                        end
                        stack_pop = 1;
                        #1 stack_clk = 1;
                        #1 stack_clk = 0;
                        stack_pop = 0;
                        stack_push = 1;
                        stack_in = stack_out;
                        #1 stack_clk = 1;
                        #1 stack_clk = 0;
                        stack_push = 0;
                    end

                    stack_push = 1;
                    stack_in = infix_expr[i+:8];
                    #1 stack_clk = 1;
                    #1 stack_clk = 0;
                    stack_push = 0;
                end
            end
        end
        while (!empty) begin
            stack_pop = 1;
            #1 stack_clk = 1;
            #1 stack_clk = 0;
            stack_pop = 0;
            postfix_expr[out_idx+:8] = stack_out;
            out_idx = out_idx - 8;
        end
    end
endmodule
```

```
timescale 1ns/1ps
module evalpost #(parameter LEN, parameter N)(
    input reg [8*LEN-1:0] infix_expr,
    output reg signed [N-1:0] result,
    output reg overflow
);
    wire [8*LEN-1:0] postfix_expr;
    reg clk, rst;
    reg [2:0] opcode;
    reg signed [N-1:0] input_data;
    wire signed [N-1:0] output_data;
    reg signed [N-1:0] temp_data;
    wire wire_overflow, success;
    integer i, j, digits, is_neg, tmp_index;

    infix2postfix #(LEN) uut1 (
        .infix_expr(infix_expr),
        .postfix_expr(postfix_expr)
    );
    STACK_BASED_ALU #(N, LEN) uut2 (
        .clk(clk),
        .rst(rst),
        .opcode(opcode),
        .input_data(input_data),
        .output_data(output_data),
        .overflow(wire_overflow),
        .success(success)
    );
    always #10 clk = ~clk;
    initial begin
        rst = 1;
        clk = 0;
        opcode = 3'b000;
        overflow = 0;
        #200 rst = 0;
        $display("Postfix expression: %s", postfix_expr);
        for (i = 8*LEN-8; i >= 0; i = i - 8) begin
            if (postfix_expr[i+:8] == 0) begin
                break;
            end

```

```
                overflow = 1;
            end
            opcode = 3'b111;
            #20;
            opcode = 3'b111;
            #20;
            input_data = temp_data;
            opcode = 3'b110;
            #20;
            opcode = 3'b000;
        end else if (postfix_expr[i+:8] == " ") begin
            continue;

        end else begin
            is_neg = 0;
            digits = 0;
            input_data = 0;
            tmp_index = i;
            for (j = i; j >= 0; j = j - 8) begin
                if (postfix_expr[j+:8] == " " || postfix_expr[j+:8] == "+" || postfix_expr[j+:8] == "*") begin
                    i = i + 8;
                    break;
                end else if (postfix_expr[j+:8] == "-") begin
                    is_neg = 1;
                end else begin
                    digits = digits + 1;
                end
                i = i - 8;
            end
            digits = digits - 1;
            if (is_neg) begin
                tmp_index = tmp_index - 8;
            end
            for (j = tmp_index; digits >= 0; j = j - 8) begin
                input_data = input_data * 10 + (postfix_expr[j+:8] - "0");
                digits = digits - 1;
            end
            if (is_neg) begin
                input_data = -input_data;
            end
            opcode = 3'b110;
            #20;
```

در ماژول evalpost: در ابتدای کد، سیگنال‌های مورد نیاز تعریف می‌شوند. این سیگنال‌ها شامل سیگنال‌های مربوط به ساعت (clk)، ریست(rst) ، دستورالعمل(opcode) ، داده ورودی(input_data) ، داده خروجی(output_data) ، و ... هستند.

در بخش اولیه، مقداردهی اولیه برای سیگنال‌های مورد نیاز انجام می‌شود. سیگنال ریست (rst) برای 200 واحد زمانی فعال می‌شود تا استک ریست شود. سپس، عبارت پست‌فیکس تولید شده نمایش داده می‌شود و یک حلقه برای پردازش هر کاراکتر از عبارت پست‌فیکس شروع می‌شود. اگر کاراکتر یک عملگر باشد (+ یا *)، عملیات مربوطه با استفاده از ماژول ALU_BASED_STACK انجام می‌شود. در صورتی که عملگر +باشد، عملیات جمع (opcode = 3'b100) و در صورتی که *باشد، عملیات ضرب (opcode = 3'b101)انجام می‌شود. اگر کاراکتر فضای خالی باشد، حلقه ادامه پیدا می‌کند. در غیر این صورت، کاراکتر به عنوان یک عدد پردازش می‌شود. ابتدا بررسی می‌شود که آیا عدد منفی است یا نه، سپس عدد خوانده شده و به داده ورودی (input_data) اختصاص داده می‌شود.

در نهایت در ماژولevalpost_tb ما یک عبارت ریاضی که در سوال داده شده را تست میکنیم و خروجی می دهیم.

عکس تست بنچ در صفحه بعد...

```
`timescale 1ns/1ps

module evalpost_tb;

    parameter LEN = 100;
    parameter N = 16;

    reg [8*LEN-1:0] infix_expr;
    wire signed [N-1:0] result;
    wire overflow;

    evalpost #(.LEN(LEN), .N(N)) uut (
        .infix_expr(infix_expr),
        .result(result),
        .overflow(overflow)
    );

    task padding;
        input integer str_len;
        begin
            integer i;
            for (i = 0; i < str_len; i = i + 1) begin
                infix_expr = {infix_expr, "\0"};
            end
        end
    endtask

    initial begin
        infix_expr = "2 * 3 + (10 + 4 + 3) * -20 + (6 + 5)";
        padding(64);
        $display("Infix expression: %s", infix_expr);
        #1000 $display("result: %d, overflow: %d", result, overflow);

        $stop;
    end

endmodule
```

نتیجه بدست آمده:

```
VSIM 33> run -all
# Infix expression: 2 * 3 + (10 + 4 + 3) * -20 + (6 + 5)
# Postfix expression: 2  3 * 10  4 + 3+  -20 *+ 6  5++
# result:   -323, overflow: 0
```