

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 عناصرها 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_index وجود دارد که به عنوان پوینتر به بالای استک عمل می‌کند. در ابتدا، top_index برابر 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_index به 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 empty غیرفعال می‌گردد. اگر سیگنال pop فعال باشد و استک خالی نباشد، top_index یک 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 این ماژول شامل داده‌های خروجی (output_data) ، سیگنال (overflow) و سیگنال (success) می‌باشند. این ماژول از ماژول دیگری به نام Stack نمونه می‌گیرد و سیگنال‌های مربوطه را به این نمونه می‌دهیم.

اگر کد عملیاتی برابر با b110 3 باشد، عملیات (push) انجام می‌شود. اگر استک پر نباشد، داده‌های ورودی به استک پوش می‌شوند و سیگنال موفقیت فعال می‌شود. اگر کد عملیاتی برابر با b111 3 باشد، 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 b100 3 باشد، عملیات جمع انجام می‌شود و نتیجه به خروجی منتقل می‌شود. اگر اورفلو رخ دهد، سیگنال overflow فعال می‌شود. اگر کد عملیاتی برابر با b101 3 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...

```

1  `timescale 1ns / 1ps
2  module STACK_BASED_ALU #(parameter N, parameter MAX_SIZE)(
3      input wire clk,
4      input wire rst,
5      input wire [2:0] opcode,
6      input wire signed [N-1:0] input_data,
7      output reg signed [N-1:0] output_data,
8      output reg overflow,
9      output reg success
10 );
11     wire full, empty;
12     reg stack_clk, stack_push, stack_pop;
13     reg signed [N-1:0] stack_in, top, top_next;
14     wire signed [N-1:0] stack_out;
15     reg signed [N-1:0] truncated_result;
16     reg signed [2*N-1:0] result;
17     Stack#(N, MAX_SIZE) stack_inst (
18         .clk(stack_clk),
19         .rst(rst),
20         .push(stack_push),
21         .pop(stack_pop),
22         .data_in(stack_in),
23         .data_out(stack_out),
24         .full(full),
25         .empty(empty)
26     );
27     initial begin
28         stack_clk = 0;
29         stack_push = 0;
30         stack_pop = 0;
31     end
32     always @(posedge clk) begin
33         success = 0;
34         overflow = 0;
35         output_data = 'bz;
36         if (opcode == 3'b110) begin
37             if (!full) begin
38                 stack_push = 1;
39                 stack_in = input_data;
40                 #1 stack_clk = 1;
41                 #1 stack_clk = 0;

```

```

41         #1 stack_clk = 0;
42         stack_push = 0;
43         success = 1;
44     end
45 end else if (opcode == 3'b111) begin
46     if (!empty) begin
47         stack_pop = 1;
48         #1 stack_clk = 1;
49         #1 stack_clk = 0;
50         output_data = stack_out;
51         stack_pop = 0;
52         success = 1;
53     end
54 end else if (opcode == 3'b100 || opcode == 3'b101) begin
55     if (!empty) begin
56         stack_pop = 1;
57         #1 stack_clk = 1;
58         #1 stack_clk = 0;
59         top = stack_out;
60         stack_pop = 0;
61         if (!empty) begin
62             stack_pop = 1;
63             #1 stack_clk = 1;
64             #1 stack_clk = 0;
65             top_next = stack_out;
66             stack_pop = 0;
67             success = 1;
68             if (opcode == 3'b100) begin
69                 output_data = top + top_next;
70                 overflow = (~top[N-1] & ~top_next[N-1] & output_data[N-1]) | (top[N-1] & top_next[N-1] & ~output_data[N-1]);
71             end else begin
72                 output_data = top * top_next;
73                 result = top * top_next;
74                 truncated_result = top * top_next;
75                 if (result != truncated_result) begin
76                     overflow = 1;
77                 end
78             end
79             stack_push = 1;
80             stack_in = top_next;
81             #1 stack_clk = 1;

```

```

78         end
79         stack_push = 1;
80         stack_in = top_next;
81         #1 stack_clk = 1;
82         #1 stack_clk = 0;
83         stack_push = 0;
84         stack_push = 1;
85         stack_in = top;
86         #1 stack_clk = 1;
87         #1 stack_clk = 0;
88         stack_push = 0;
89     end else begin
90         stack_push = 1;
91         stack_in = top;
92         #1 stack_clk = 1;
93         #1 stack_clk = 0;
94         stack_push = 0;
95     end
96 end
97 end
98 end
99 endmodule

```

ماژول STACK_BASED_ALU_tb: این کد یک تست بنچ برای ماژول ALU_BASED_STACK است. در این کد، پارامترهای N و MAX_SIZE به 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 نانو ثانیه 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 نانو ثانیه) 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 داده‌ها 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

```

```

86     begin
87         result = val1 + val2;
88         truncated_result = result;
89         $display("Addition of top two stack values: %d + %d = %d (expected), truncated to %d bits = %d",
90             $signed(val1), $signed(val2), $signed(result), N, $signed(truncated_result));
91         opcode = 3'b100;
92         #20;
93         $display("Addition result: %d, Overflow: %b", $signed(output_data), overflow);
94     end
95 endtask
96
97 task perform_multiplication;
98     input signed [N-1:0] val1, val2;
99     begin
100         result = val1 * val2;
101         truncated_result = result;
102         $display("Multiplication of top two stack values: %d * %d = %d (expected), truncated to %d bits = %d",
103             $signed(val1), $signed(val2), $signed(result), N, $signed(truncated_result));
104         opcode = 3'b101;
105         #20;
106         $display("Multiplication result: %d, Overflow: %b", $signed(output_data), overflow);
107     end
108 endtask
109
110 initial begin
111     initialize;
112
113     repeat (3) begin
114         generate_random_numbers(number1, number2, number3, number4);
115
116         push_to_stack(number1);
117         push_to_stack(number2);
118         push_to_stack(number3);
119         push_to_stack(number4);
120
121         perform_addition(number4, number3);
122
123         pop_from_stack(number4);
124         pop_from_stack(number3);
125
126         perform_multiplication(number1, number2);
127     end

```

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


```

VSIIM65> 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...

```

1 timescale 1ns / 1ps
2 module infix2postfix #(parameter LEN = 16)(
3     input reg [8*LEN-1:0] infix_expr,
4     output reg [8*LEN-1:0] postfix_expr
5 );
6     reg [7:0] stack_in;
7     wire [7:0] stack_out;
8     reg stack_push, stack_pop, stack_clk, rst;
9     wire full, empty;
10    integer out_idx;
11    integer i;
12
13    Stack#(8, LEN) stack_inst (
14        .clk(stack_clk),
15        .rst(rst),
16        .push(stack_push),
17        .pop(stack_pop),
18        .data_in(stack_in),
19        .data_out(stack_out),
20        .full(full),
21        .empty(empty)
22    );
23
24    initial begin
25        stack_push = 0;
26        stack_pop = 0;
27        stack_clk = 0;
28        rst = 1;
29        #10 stack_clk = 1;
30        #10 stack_clk = 0;
31        out_idx = 8*LEN - 8;
32        rst = 0;
33    end
34
35    initial begin
36        postfix_expr = {LEN{"\0"}};
37
38        #100
39        for (i = 8*LEN - 8; i >= 0; i = i - 8) begin
40            if (infix_expr[i+:8] == 0) begin
41                break;

```

```

41         break;
42     end else if ((infix_expr[i+:8] != "+") && (infix_expr[i+:8] != "*") && (infix_expr[i+:8]
43         postfix_expr[out_idx+:8] = infix_expr[i+:8];
44         out_idx = out_idx - 8;
45     end else if (infix_expr[i+:8] == "(") begin
46         stack_push = 1;
47         stack_in = infix_expr[i+:8];
48         #1 stack_clk = 1;
49         #1 stack_clk = 0;
50         stack_push = 0;
51     end else if (infix_expr[i+:8] == ")") begin
52         stack_pop = 1;
53         #1 stack_clk = 1;
54         #1 stack_clk = 0;
55         stack_pop = 0;
56         while (!empty && stack_out != "(") begin
57             postfix_expr[out_idx+:8] = stack_out;
58             out_idx = out_idx - 8;
59             stack_pop = 1;
60             #1 stack_clk = 1;
61             #1 stack_clk = 0;
62             stack_pop = 0;
63         end
64     end else begin
65         if (empty) begin
66             stack_push = 1;
67             stack_in = infix_expr[i+:8];
68             #1 stack_clk = 1;
69             #1 stack_clk = 0;
70             stack_push = 0;
71         end else begin
72             stack_pop = 1;
73             #1 stack_clk = 1;
74             #1 stack_clk = 0;
75             stack_pop = 0;
76             stack_push = 1;
77             stack_in = stack_out;
78             #1 stack_clk = 1;
79             #1 stack_clk = 0;
80             stack_push = 0;
81

```

```

80     stack_push = 0;
81
82     while (!empty && ((infix_expr[i+:8] == "+" && (stack_out == "+" || stack_out == "**")) || (infix_expr[i+:8] == "*" &&
83         stack_pop = 1;
84         #1 stack_clk = 1;
85         #1 stack_clk = 0;
86         stack_pop = 0;
87         postfix_expr[out_idx+:8] = stack_out;
88         out_idx = out_idx - 8;
89         if (empty) begin
90             break;
91         end
92         stack_pop = 1;
93         #1 stack_clk = 1;
94         #1 stack_clk = 0;
95         stack_pop = 0;
96         stack_push = 1;
97         stack_in = stack_out;
98         #1 stack_clk = 1;
99         #1 stack_clk = 0;
100         stack_push = 0;
101     end
102
103     stack_push = 1;
104     stack_in = infix_expr[i+:8];
105     #1 stack_clk = 1;
106     #1 stack_clk = 0;
107     stack_push = 0;
108 end
109 end
110 end
111 while (!empty) begin
112     stack_pop = 1;
113     #1 stack_clk = 1;
114     #1 stack_clk = 0;
115     stack_pop = 0;
116     postfix_expr[out_idx+:8] = stack_out;
117     out_idx = out_idx - 8;
118 end
119 end
120 endmodule

```

```

1 | timescale 1ns/1ps
2 | module evalpost #(parameter LEN, parameter N)(
3 |     input reg [8*LEN-1:0] infix_expr,
4 |     output reg signed [N-1:0] result,
5 |     output reg overflow
6 | );
7 |     wire [8*LEN-1:0] postfix_expr;
8 |     reg clk, rst;
9 |     reg [2:0] opcode;
10 |    reg signed [N-1:0] input_data;
11 |    wire signed [N-1:0] output_data;
12 |    reg signed [N-1:0] temp_data;
13 |    wire wire_overflow, success;
14 |    integer i, j, digits, is_neg, tmp_index;
15 |
16 |    infix2postfix #(LEN) uut1 (
17 |        .infix_expr(infix_expr),
18 |        .postfix_expr(postfix_expr)
19 |    );
20 |    STACK_BASED_ALU #(N, LEN) uut2 (
21 |        .clk(clk),
22 |        .rst(rst),
23 |        .opcode(opcode),
24 |        .input_data(input_data),
25 |        .output_data(output_data),
26 |        .overflow(wire_overflow),
27 |        .success(success)
28 |    );
29 |    always #10 clk = ~clk;
30 |    initial begin
31 |        rst = 1;
32 |        clk = 0;
33 |        opcode = 3'b000;
34 |        overflow = 0;
35 |        #200 rst = 0;
36 |        $display("Postfix expression: %s", postfix_expr);
37 |        for (i = 8*LEN-8; i >= 0; i = i - 8) begin
38 |            if (postfix_expr[i+:8] == 0) begin
39 |                break;
40 |            end
41 |

```

```

47         overflow = 1;
48     end
49     opcode = 3'b111;
50     #20;
51     opcode = 3'b111;
52     #20;
53     input_data = temp_data;
54     opcode = 3'b110;
55     #20;
56     opcode = 3'b000;
57 end else if (postfix_expr[i+:8] == " ") begin
58     continue;
59
60 end else begin
61     is_neg = 0;
62     digits = 0;
63     input_data = 0;
64     tmp_index = i;
65     for (j = i; j >= 0; j = j - 8) begin
66         if (postfix_expr[j+:8] == " " || postfix_expr[j+:8] == "+" || postfix_expr[j+:8] == "*") begin
67             i = i + 8;
68             break;
69         end else if (postfix_expr[j+:8] == "-") begin
70             is_neg = 1;
71         end else begin
72             digits = digits + 1;
73         end
74         i = i - 8;
75     end
76     digits = digits - 1;
77     if (is_neg) begin
78         tmp_index = tmp_index - 8;
79     end
80     for (j = tmp_index; digits >= 0; j = j - 8) begin
81         input_data = input_data * 10 + (postfix_expr[j+:8] - "0");
82         digits = digits - 1;
83     end
84     if (is_neg) begin
85         input_data = -input_data;
86     end
87     opcode = 3'b110;
88     #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 ماژول evalpost_tb ما یک عبارت ریاضی که در سوال داده شده را تست می‌کنیم و خروجی می‌دهیم.

عکس تست بنچ در صفحه بعد...

```

1  `timescale 1ns/1ps
2
3  module evalpost_tb;
4
5      parameter LEN = 100;
6      parameter N = 16;
7
8      reg [8*LEN-1:0] infix_expr;
9      wire signed [N-1:0] result;
10     wire overflow;
11
12     evalpost #(.LEN(LEN), .N(N)) uut (
13         .infix_expr(infix_expr),
14         .result(result),
15         .overflow(overflow)
16     );
17
18     task padding;
19         input integer str_len;
20         begin
21             integer i;
22             for (i = 0; i < str_len; i = i + 1) begin
23                 infix_expr = {infix_expr, "\0"};
24             end
25         end
26     endtask
27
28     initial begin
29         infix_expr = "2 * 3 + (10 + 4 + 3) * -20 + (6 + 5)";
30         padding(64);
31         $display("Infix expression: %s", infix_expr);
32         #1000 $display("result: %d, overflow: %d", result, overflow);
33
34         $stop;
35     end
36
37 endmodule
38

```

نتیجه بدست آمده:

```

VSI1M 55> 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
# Break in Module evalpost_tb at C:/Users/ESP/Desktop/DSP_Doc4/CALCULATE EXP

```