



DSD - Project Report

Mehran Talaei 401110364

June 29, 2024

STACKED BASED ALU

```
1  'timescale 1ns/1ps
2
3  module STACK_BASED_ALU
4  #(parameter DATA_WIDTH = 8, parameter STACK_SIZE = 64)
5  (
6      input clk,
7      input [DATA_WIDTH-1:0] input_data,
8      input [2:0] opcode,
9      output reg [DATA_WIDTH-1:0] output_data,
10     output reg overflow,
11     output reg [DATA_WIDTH-1:0] debug_value
12 );
13
14 reg [DATA_WIDTH-1:0] stack [0:STACK_SIZE-1];
15 integer stack_pointer = 0;
16
17 always @(posedge clk) begin
18     overflow <= 0;
19     debug_value <= 0;
20     case (opcode)
21         3'b100: // addition
22             begin
23                 if (stack_pointer >= 2) begin
24                     stack[stack_pointer - 2] <= stack[stack_pointer - 1] + stack[
25                         stack_pointer - 2];
26                     debug_value <= stack[stack_pointer - 2];
27                     if ((stack[stack_pointer - 2][DATA_WIDTH-1] == 1 && stack[
28                         stack_pointer - 1][DATA_WIDTH-1] == 0 && stack[stack_pointer
29                             - 2][DATA_WIDTH-1] == 0) ||
30                         (stack[stack_pointer - 2][DATA_WIDTH-1] == 0 && stack[
31                             stack_pointer - 1][DATA_WIDTH-1] == 1 && stack[
32                                 stack_pointer - 2][DATA_WIDTH-1] == 1))
33                         overflow <= 1;
34                     stack_pointer <= stack_pointer - 1;
35                 end else begin
36                     overflow <= 1;
37                 end
38                 output_data <= debug_value;
39             end
40
41         3'b101: // multiplication
42             begin
43                 if (stack_pointer >= 2) begin
44                     stack[stack_pointer - 2] <= stack[stack_pointer - 1] * stack[
45                         stack_pointer - 2];
46                     debug_value <= stack[stack_pointer - 2];
47                     if (|stack[stack_pointer - 2][2*DATA_WIDTH-1:DATA_WIDTH])
48                         overflow <= 1;
49                     stack_pointer <= stack_pointer - 1;
50                 end else begin
51                     overflow <= 1;
52                 end
53                 output_data <= debug_value;
54             end
55
56         3'b110: // pushing
57             begin
58                 if (stack_pointer < STACK_SIZE) begin
59                     stack[stack_pointer] <= input_data;
60                     stack_pointer <= stack_pointer + 1;
61                 end else begin
```

```

55         overflow <= 1;
56     end
57     output_data <= {DATA_WIDTH{1'bz}};
58 end
59
60 3'b111: // popping
61 begin
62     if (stack_pointer > 0) begin
63         output_data <= stack[stack_pointer-1];
64         stack_pointer <= stack_pointer - 1;
65     end else begin
66         overflow <= 1;
67         output_data <= {DATA_WIDTH{1'bz}};
68     end
69 end
70
71 default: begin
72     overflow <= 0;
73     output_data <= {DATA_WIDTH{1'bz}};
74 end
75 endcase
76 end
77
78 endmodule

```

Testbench for STACK_BASED_ALU

```

1  `timescale 1ns/1ps
2
3  module tb_STACK_BASED_ALU;
4
5      parameter DATA_WIDTH = 8;
6      parameter STACK_SIZE = 64;
7
8      reg clk;
9      reg [DATA_WIDTH-1:0] input_data;
10     reg [2:0] opcode;
11     wire [DATA_WIDTH-1:0] output_data;
12     wire overflow;
13     wire [DATA_WIDTH-1:0] debug_value;
14
15     STACK_BASED_ALU #(DATA_WIDTH, STACK_SIZE) alu (
16         .clk(clk),
17         .input_data(input_data),
18         .opcode(opcode),
19         .output_data(output_data),
20         .overflow(overflow),
21         .debug_value(debug_value)
22     );
23
24     // Clock generation
25     initial begin
26         clk = 0;
27         forever #5 clk = ~clk; // 10ns period clock
28     end
29
30     // Helper function to return operation name
31     function [79:0] get_operation_name(input [2:0] opcode);
32         case (opcode)

```

```

33     3'b100: get_operation_name = "Addition";
34     3'b101: get_operation_name = "Multiplication";
35     3'b110: get_operation_name = "Push";
36     3'b111: get_operation_name = "Pop";
37     default: get_operation_name = "Unknown";
38 endcase
39 endfunction
40
41 // Task to display the stack content
42 task display_stack;
43     integer i;
44     begin
45         $display("Stack content:");
46         for (i = 0; i < alu.stack_pointer; i = i + 1) begin
47             $display("stack[%0d] = %h", i, alu.stack[i]);
48         end
49     end
50 endtask
51
52 // Test procedure
53 initial begin
54     // Initialize inputs
55     input_data = 0;
56     opcode = 3'b000;
57
58     // Wait for the clock to stabilize
59     #10;
60
61     // Test cases
62     $monitor("Time = %0t, Opcode = %b (%0s), Input = %h, Output = %h, Overflow
63             = %b, Stack Pointer = %d, Debug Value = %h",
64             $time, opcode, get_operation_name(opcode), input_data, output_data
65             , overflow, alu.stack_pointer, debug_value);
66
67     // Push some values onto the stack
68     input_data = 8'h05;
69     opcode = 3'b110; // Push
70     #10;
71     display_stack;
72
73     input_data = 8'h03;
74     opcode = 3'b110; // Push
75     #10;
76     display_stack;
77
78     // Perform addition
79     opcode = 3'b100; // Add
80     #10;
81     display_stack;
82
83     // Push more values
84     input_data = 8'h02;
85     opcode = 3'b110; // Push
86     #10;
87     display_stack;
88
89     input_data = 8'h03;
90     opcode = 3'b110; // Push
91     #10;
92     display_stack;
93
94     // Perform addition again
95     opcode = 3'b100; // Add

```

```

94     #10;
95     display_stack;
96
97     input_data = 8'h04;
98     opcode = 3'b110; // Push
99     #10;
100    display_stack;
101
102    // Perform multiplication
103    opcode = 3'b101; // Multiply
104    #10;
105    display_stack;
106
107    // Push more values
108    input_data = 8'h06;
109    opcode = 3'b110; // Push
110    #10;
111    display_stack;
112
113    input_data = 8'h02;
114    opcode = 3'b110; // Push
115    #10;
116    display_stack;
117
118    // Perform addition
119    opcode = 3'b100; // Add
120    #10;
121    display_stack;
122
123    // Perform multiplication again
124    opcode = 3'b101; // Multiply
125    #10;
126    display_stack;
127
128    // Pop a value from the stack
129    opcode = 3'b111; // Pop
130    #10;
131    display_stack;
132
133    // Test stack underflow
134    opcode = 3'b111; // Pop
135    #10;
136    display_stack;
137
138    opcode = 3'b111; // Pop
139    #10;
140    display_stack;
141
142    // Push to stack overflow
143    repeat (STACK_SIZE - 1) begin
144        input_data = 8'h01;
145        opcode = 3'b110; // Push
146        #10;
147        display_stack;
148    end
149
150    // Final push to cause overflow
151    input_data = 8'h01;
152    opcode = 3'b110; // Push
153    #10;
154    display_stack;
155
156    // Finish the simulation

```

```

157     $finish;
158 end
159
160 endmodule

```

Output of the Testbench

```

1 # Time = 10000, Opcode = 110 (Push), Input = 05, Output = zz, Overflow = 0,
   Stack Pointer = 0, Debug Value = 00
2 # Time = 15000, Opcode = 110 (Push), Input = 05, Output = zz, Overflow = 0,
   Stack Pointer = 1, Debug Value = 00
3 # Stack content:
4 # stack[0] = 05
5 # Time = 20000, Opcode = 110 (Push), Input = 03, Output = zz, Overflow = 0,
   Stack Pointer = 1, Debug Value = 00
6 # Time = 25000, Opcode = 110 (Push), Input = 03, Output = zz, Overflow = 0,
   Stack Pointer = 2, Debug Value = 00
7 # Stack content:
8 # stack[0] = 05
9 # stack[1] = 03
10 # Time = 30000, Opcode = 100 (Addition), Input = 03, Output = zz, Overflow = 0,
    Stack Pointer = 2, Debug Value = 00
11 # Time = 35000, Opcode = 100 (Addition), Input = 03, Output = 00, Overflow = 0,
    Stack Pointer = 1, Debug Value = 05
12 # Stack content:
13 # stack[0] = 08
14 # Time = 40000, Opcode = 110 (Push), Input = 02, Output = 00, Overflow = 0,
    Stack Pointer = 1, Debug Value = 05
15 # Time = 45000, Opcode = 110 (Push), Input = 02, Output = zz, Overflow = 0,
    Stack Pointer = 2, Debug Value = 00
16 # Stack content:
17 # stack[0] = 08
18 # stack[1] = 02
19 # Time = 50000, Opcode = 110 (Push), Input = 03, Output = zz, Overflow = 0,
    Stack Pointer = 2, Debug Value = 00
20 # Time = 55000, Opcode = 110 (Push), Input = 03, Output = zz, Overflow = 0,
    Stack Pointer = 3, Debug Value = 00
21 # Stack content:
22 # stack[0] = 08
23 # stack[1] = 02
24 # stack[2] = 03
25 # Time = 60000, Opcode = 100 (Addition), Input = 03, Output = zz, Overflow = 0,
    Stack Pointer = 3, Debug Value = 00
26 # Time = 65000, Opcode = 100 (Addition), Input = 03, Output = 00, Overflow = 0,
    Stack Pointer = 2, Debug Value = 02
27 # Stack content:
28 # stack[0] = 08
29 # stack[1] = 05
30 # Time = 70000, Opcode = 110 (Push), Input = 04, Output = 00, Overflow = 0,
    Stack Pointer = 2, Debug Value = 02
31 # Time = 75000, Opcode = 110 (Push), Input = 04, Output = zz, Overflow = 0,
    Stack Pointer = 3, Debug Value = 00
32 # Stack content:
33 # stack[0] = 08
34 # stack[1] = 05
35 # stack[2] = 04
36 # Time = 80000, Opcode = 101 (Multiplication), Input = 04, Output = zz,
    Overflow = 0, Stack Pointer = 3, Debug Value = 00

```

```

37 # Time = 85000, Opcode = 101 (Multiplication), Input = 04, Output = 00,
    Overflow = 0, Stack Pointer = 2, Debug Value = 05
38 # Stack content:
39 # stack[0] = 08
40 # stack[1] = 14
41 # Time = 90000, Opcode = 110 (Push), Input = 06, Output = 00, Overflow = 0,
    Stack Pointer = 2, Debug Value = 05
42 # Time = 95000, Opcode = 110 (Push), Input = 06, Output = zz, Overflow = 0,
    Stack Pointer = 3, Debug Value = 00
43 # Stack content:
44 # stack[0] = 08
45 # stack[1] = 14
46 # stack[2] = 06
47 # Time = 100000, Opcode = 110 (Push), Input = 02, Output = zz, Overflow = 0,
    Stack Pointer = 3, Debug Value = 00
48 # Time = 105000, Opcode = 110 (Push), Input = 02, Output = zz, Overflow = 0,
    Stack Pointer = 4, Debug Value = 00
49 # Stack content:
50 # stack[0] = 08
51 # stack[1] = 14
52 # stack[2] = 06
53 # stack[3] = 02
54 # Time = 110000, Opcode = 100 (Addition), Input = 02, Output = zz, Overflow =
    0, Stack Pointer = 4, Debug Value = 00
55 # Time = 115000, Opcode = 100 (Addition), Input = 02, Output = 00, Overflow =
    0, Stack Pointer = 3, Debug Value = 06
56 # Stack content:
57 # stack[0] = 08
58 # stack[1] = 14
59 # stack[2] = 08
60 # Time = 120000, Opcode = 101 (Multiplication), Input = 02, Output = 00,
    Overflow = 0, Stack Pointer = 3, Debug Value = 06
61 # Time = 125000, Opcode = 101 (Multiplication), Input = 02, Output = 06,
    Overflow = 0, Stack Pointer = 2, Debug Value = 14
62 # Stack content:
63 # stack[0] = 08
64 # stack[1] = 0A
65 # Time = 130000, Opcode = 111 (Pop), Input = 02, Output = 06, Overflow = 0,
    Stack Pointer = 2, Debug Value = 14
66 # Time = 135000, Opcode = 111 (Pop), Input = 02, Output = 0A, Overflow = 0,
    Stack Pointer = 1, Debug Value = 00
67 # Stack content:
68 # stack[0] = 08
69 # Time = 145000, Opcode = 111 (Pop), Input = 02, Output = 08, Overflow = 0,
    Stack Pointer = 0, Debug Value = 00
70 # Stack content:
71 # Time = 155000, Opcode = 111 (Pop), Input = 02, Output = zz, Overflow = 1,
    Stack Pointer = 0, Debug Value = 00
72 # Stack content:
73 # Time = 160000, Opcode = 110 (Push), Input = 01, Output = zz, Overflow = 1,
    Stack Pointer = 0, Debug Value = 00
74 # Time = 165000, Opcode = 110 (Push), Input = 01, Output = zz, Overflow = 0,
    Stack Pointer = 1, Debug Value = 00
75 # Stack content:
76 # stack[0] = 01
77 # Time = 175000, Opcode = 110 (Push), Input = 01, Output = zz, Overflow = 0,
    Stack Pointer = 2, Debug Value = 00
78 # Stack content:
79 # stack[0] = 01
80 # stack[1] = 01

```

Calculator Module

- **input:**input must begin with the '(' and must end with the ')'. operands and operators appears in order of the string.
- **output:**it shows the stack changing. at the end there is exactly one number in the stack which is the answer.

```
1  `timescale 1ns/1ps
2
3  module Calculator
4  (
5      input clk,                // Clock signal
6      input [2:0] opcode,        // Microcode instruction
7      input [15:0] operand       // Input operand
8  );
9
10 reg [2:0] alu_opcode;           // Operation code for ALU
11 wire [15:0] alu_result;        // Result from ALU
12 wire overflow;                 // Overflow flag from ALU
13
14 // Instantiate the STACK_BASED_ALU
15 STACK_BASED_ALU #(.DATA_WIDTH(16), .STACK_SIZE(64)) stack_alu (
16     .clk(clk),
17     .input_data(operand),
18     .opcode(alu_opcode),
19     .output_data(alu_result),
20     .overflow(overflow)
21 );
22
23 integer pending_mult_stack [63:0]; // Stack to keep track of pending
    multiplications
24 integer pending_mult_index = 0; // Index for the pending multiplication stack
25 integer pending_addition_stack [63:0]; // Stack to keep track of pending
    additions
26 integer pending_addition_index = 0; // Index for the pending addition stack
27
28 // Always block to handle the operations
29 always @(negedge clk) begin
30     // Handle multiplication
31     if (pending_mult_stack[pending_mult_index] > 1) begin
32         $display("multttttttttttt");
33         alu_opcode = 3'b101; // Set opcode to multiplication
34         #10
35         pending_mult_stack[pending_mult_index] = 0;
36     end
37     // Handle addition
38     else if (pending_addition_stack[pending_addition_index] > 1) begin
39         $display("addddddddddd ");
40         alu_opcode = 3'b100; // Set opcode to addition
41         pending_addition_stack[pending_addition_index] = 0;
42     end
43     // Handle input reading when no pending operations
44     else begin
45         $display("lets go");
46         case (opcode)
47             3'b000: begin // Addition
48                 $display("babababaababa");
49                 if (pending_addition_stack[pending_addition_index] == 1) begin
50                     alu_opcode = 3'b100; // Set opcode to addition
51                     pending_addition_stack[pending_addition_index] = 1;
52                 end

```



```

53         else begin
54             pending_addition_stack[pending_addition_index] = 1;
55             alu_opcode = 3'b000;
56         end
57     end
58     3'b001: begin // Multiplication
59         pending_mult_stack[pending_mult_index] = 1;
60         alu_opcode = 3'b000;
61     end
62     3'b010: begin // Open parenthesis '('
63         pending_mult_stack[pending_mult_index + 1] = 0; // Initialize
64             next level multiplication state
65         pending_addition_stack[pending_addition_index + 1] = 0; //
66             Initialize next level addition state
67         pending_mult_index = pending_mult_index + 1;
68         pending_addition_index = pending_addition_index + 1;
69         alu_opcode = 0;
70     end
71     3'b011: begin // Close parenthesis ')'
72         if (pending_addition_stack[pending_addition_index] == 1) begin
73             alu_opcode = 3'b100; // Set opcode to addition
74             pending_addition_stack[pending_addition_index] = 0;
75         end
76         else begin
77             alu_opcode = 0;
78         end
79         pending_addition_index = pending_addition_index - 1;
80         pending_mult_index = pending_mult_index - 1;
81         if (pending_mult_stack[pending_mult_index] == 1)
82             pending_mult_stack[pending_mult_index] = 2;
83     end
84     3'b100: begin // Operand
85         alu_opcode = 3'b110;
86         if (pending_mult_stack[pending_mult_index] == 1)
87             pending_mult_stack[pending_mult_index] = 2;
88     end
89 endcase
90 end
91 end
92 endmodule

```

Testbench for Calculator Module

Summary

The test steps follow the expression $(2 \times 3 + (10 + 4 + 3) \times -20 + (6 + 5)) =$.

0.1 Verilog Code

```

1  `timescale 1ns/1ps
2
3  module tb_Calculator;
4
5      reg clk;
6      reg [2:0] opcode;
7      reg [15:0] operand;
8      wire [15:0] alu_result;
9

```

```

10 // Instantiate the Calculator
11 Calculator calculator (
12     .clk(clk),
13     .opcode(opcode),
14     .operand(operand)
15 );
16
17 // Clock generation
18 initial begin
19     clk = 0;
20     forever #5 clk = ~clk; // 10ns period clock
21 end
22
23 // Helper function to return operation name
24 function [79:0] get_operation_name(input [2:0] opcode);
25     case (opcode)
26         3'b000: get_operation_name = "Addition";
27         3'b001: get_operation_name = "Multiplication";
28         3'b010: get_operation_name = "Open Parenthesis";
29         3'b011: get_operation_name = "Close Parenthesis";
30         3'b100: get_operation_name = "Push Operand";
31         3'b101: get_operation_name = "Equal Sign";
32         default: get_operation_name = "Unknown";
33     endcase
34 endfunction
35
36 // Task to display the stack content
37 task display_stack;
38     integer i;
39     begin
40         $display("Stack content:");
41         for (i = 0; i < calculator.stack_alu.stack_pointer; i = i + 1) begin
42             $display("stack[%0d] = %h", i, calculator.stack_alu.stack[i]);
43         end
44     end
45 endtask
46
47 // Task to display the current pending multiplication and addition states
48 task display_pending_states;
49     begin
50         $display("Pending multiplication state: %0d", calculator.
51             pending_mult_stack[calculator.pending_mult_index]);
52         $display("Pending addition state: %0d", calculator.pending_addition_stack
53             [calculator.pending_addition_index]);
54     end
55 endtask
56
57 // Task to display the progress of the expression calculation
58 task display_progress(input [8*80:1] expression, input integer current_step);
59     begin
60         $display("Progress at step %0d: %0s", current_step, expression);
61     end
62 endtask
63
64 // Test procedure
65 initial begin
66     // Initialize inputs
67     operand = 0;
68     opcode = 3'b000;
69
70     // Wait for the clock to stabilize
71     #10;

```

```

71 // Test cases
72 $monitor("Time = %0t, Opcode = %b (%0s), Operand = %h, Alu_Result = %h,
73         Pending Mult = %0d, Pending Add = %0d",
74         $time, opcode, get_operation_name(opcode), operand, calculator.
75         alu_result,
76         calculator.pending_mult_stack[calculator.pending_mult_index],
77         calculator.pending_addition_stack[calculator.
78         pending_addition_index]);
79
80 // Expression: (2 * 3 + (10 + 4 + 3) * -20 + (6 + 5))=
81
82 // Step 1: Begin expression with '('
83 opcode = 3'b010; // Open Parenthesis
84 operand = 16'bz;
85 #10;
86 display_stack;
87 display_pending_states;
88 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 1);
89
90 // Step 2: Push operand 2
91 operand = 16'd2;
92 opcode = 3'b100; // Push Operand
93 #10;
94 display_stack;
95 display_pending_states;
96 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 2);
97
98 // Step 3: Multiply 2 *
99 operand = 16'bz;
100 opcode = 3'b001; // Multiplication
101 #10;
102 display_stack;
103 display_pending_states;
104 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 3);
105
106 // Step 4: Push operand 3
107 operand = 16'd3;
108 opcode = 3'b100; // Push Operand
109 #10;
110 display_stack;
111 display_pending_states;
112 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 4);
113
114 // Step 5: Addition operation after 2 * 3
115 operand = 16'bz;
116 opcode = 3'b000; // Addition
117 #10;
118 display_stack;
119 display_pending_states;
120 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 5);
121
122 // Step 6: Open parenthesis '('
123 operand = 16'bz;
124 opcode = 3'b010; // Open Parenthesis
125 #10;
126 display_stack;
127 display_pending_states;
128 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 6);
129
130 // Step 7: Push operand 10
131 operand = 16'd10;
132 opcode = 3'b100; // Push Operand
133 #10;

```

```

131 display_stack;
132 display_pending_states;
133 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 7);
134
135 // Step 8: Addition 10 +
136 operand = 16'bz;
137 opcode = 3'b000; // Addition
138 #10;
139 display_stack;
140 display_pending_states;
141 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 8);
142
143 // Step 9: Push operand 4
144 operand = 16'd4;
145 opcode = 3'b100; // Push Operand
146 #10;
147 display_stack;
148 display_pending_states;
149 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 9);
150
151 // Step 10: Addition 10 + 4 +
152 operand = 16'bz;
153 opcode = 3'b000; // Addition
154 #10;
155 display_stack;
156 display_pending_states;
157 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 10);
158
159 // Step 11: Push operand 3
160 operand = 16'd3;
161 opcode = 3'b100; // Push Operand
162 #10;
163 display_stack;
164 display_pending_states;
165 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 11);
166
167 // Step 12: Addition 10 + 4 + 3
168 operand = 16'bz;
169 opcode = 3'b000; // Addition
170 #10;
171 display_stack;
172 display_pending_states;
173 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 12);
174
175 // Step 13: Close parenthesis ')'
176 operand = 16'bz;
177 opcode = 3'b011; // Close Parenthesis
178 #10;
179 display_stack;
180 display_pending_states;
181 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 13);
182
183 // Step 14: Multiply (10 + 4 + 3) *
184 operand = 16'bz;
185 opcode = 3'b001; // Multiplication
186 #10;
187 display_stack;
188 display_pending_states;
189 display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 14);
190
191 // Step 15: Push operand -20
192 operand = -16'd20;
193 opcode = 3'b100; // Push Operand

```

```

194     #20;
195     display_stack;
196     display_pending_states;
197     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 15);
198
199     // Step 16: Addition after first block
200     operand = 16'bz;
201     opcode = 3'b000; // Addition
202     #10;
203     display_stack;
204     display_pending_states;
205     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 16);
206
207     // Step 17: Open parenthesis '('
208     operand = 16'bz;
209     opcode = 3'b010; // Open Parenthesis
210     #10;
211     display_stack;
212     display_pending_states;
213     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 17);
214
215     // Step 18: Push operand 6
216     operand = 16'd6;
217     opcode = 3'b100; // Push Operand
218     #10;
219     display_stack;
220     display_pending_states;
221     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 18);
222
223     // Step 19: Addition 6 +
224     operand = 16'bz;
225     opcode = 3'b000; // Addition
226     #10;
227     display_stack;
228     display_pending_states;
229     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 19);
230
231     // Step 20: Push operand 5
232     operand = 16'd5;
233     opcode = 3'b100; // Push Operand
234     #10;
235     display_stack;
236     display_pending_states;
237     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 20);
238
239     // Step 21: Close parenthesis ')'
240     operand = 16'bz;
241     opcode = 3'b011; // Close Parenthesis
242     #10;
243     display_stack;
244     display_pending_states;
245     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 21);
246
247     // Step 22: Addition after second block
248     operand = 16'bz;
249     opcode = 3'b000; // Addition
250     #10;
251     display_stack;
252     display_pending_states;
253     display_progress("(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =", 22);
254
255     $finish;
256 end

```

```
257
258 endmodule
```

Testbench Output

chase the stack values.

Output

```
1 # lets go
2 # babababaababa
3 # lets go
4 # Time = 10000, Opcode = 010 (arenthesis), Operand = zzzz, Alu_Result = zzzz,
   Pending Mult = 0, Pending Add = 0
5 # Stack content:
6 # Pending multiplication state: 0
7 # Pending addition state: 0
8 # Progress at step 1: (2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =
9 # lets go
10 # Time = 20000, Opcode = 100 (sh Operand), Operand = 0002, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 0
11 # Stack content:
12 \textbf{# stack[0] = 0002}
13 # Pending multiplication state: 0
14 # Pending addition state: 0
15 # Progress at step 2: (2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =
16 # lets go
17 # Time = 30000, Opcode = 001 (iplication), Operand = zzzz, Alu_Result = zzzz,
    Pending Mult = 1, Pending Add = 0
18 # Stack content:
19 \textbf{# stack[0] = 0002}
20 # Pending multiplication state: 1
21 # Pending addition state: 0
22 # Progress at step 3: (2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =
23 # lets go
24 # Time = 40000, Opcode = 100 (sh Operand), Operand = 0003, Alu_Result = zzzz,
    Pending Mult = 2, Pending Add = 0
25 # Stack content:
26 \textbf{# stack[0] = 0002}
27 \textbf{# stack[1] = 0003}
28 # Pending multiplication state: 2
29 # Pending addition state: 0
30 # Progress at step 4: (2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =
31 # multttttttttttttt
32 # Time = 55000, Opcode = 100 (sh Operand), Operand = 0003, Alu_Result = 0000,
    Pending Mult = 2, Pending Add = 0
33 # lets go
34 # babababaababa
35 # Time = 60000, Opcode = 000 (Addition), Operand = zzzz, Alu_Result = 0000,
    Pending Mult = 0, Pending Add = 1
36 # Time = 65000, Opcode = 000 (Addition), Operand = zzzz, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 1
37 # Stack content:
38 \textbf{# stack[0] = 0006}
39 # Pending multiplication state: 0
40 # Pending addition state: 1
41 # Progress at step 5: (2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =
42 # lets go
```



```

97 # Time = 135000, Opcode = 011 (arenthesis), Operand = zzzz, Alu_Result = 0000,
    Pending Mult = 0, Pending Add = 1
98 # Stack content:
99 \textbf{# stack[0] = 0006}
100 \textbf{# stack[1] = 0011}
101 # Pending multiplication state: 0
102 # Pending addition state: 1
103 # Progress at step 13:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
104 # lets go
105 # Time = 140000, Opcode = 001 (iplication), Operand = zzzz, Alu_Result = 0000,
    Pending Mult = 1, Pending Add = 1
106 # Time = 145000, Opcode = 001 (iplication), Operand = zzzz, Alu_Result = zzzz,
    Pending Mult = 1, Pending Add = 1
107 # Stack content:
108 \textbf{# stack[0] = 0006}
109 \textbf{# stack[1] = 0011}
110 # Pending multiplication state: 1
111 # Pending addition state: 1
112 # Progress at step 14:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
113 # lets go
114 # Time = 150000, Opcode = 100 (sh Operand), Operand = ffec, Alu_Result = zzzz,
    Pending Mult = 2, Pending Add = 1
115 # multttttttttttt
116 # Time = 165000, Opcode = 100 (sh Operand), Operand = ffec, Alu_Result = 0000,
    Pending Mult = 2, Pending Add = 1
117 # Stack content:
118 \textbf{# stack[0] = 0006}
119 \textbf{# stack[1] = feac}
120 # Pending multiplication state: 2
121 # Pending addition state: 1
122 # Progress at step 15:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
123 # lets go
124 # babababaababa
125 # Time = 170000, Opcode = 000 (Addition), Operand = zzzz, Alu_Result = 0000,
    Pending Mult = 0, Pending Add = 1
126 # Time = 175000, Opcode = 000 (Addition), Operand = zzzz, Alu_Result = 0011,
    Pending Mult = 0, Pending Add = 1
127 # Stack content:
128 \textbf{# stack[0] = feb2}
129 # Pending multiplication state: 0
130 # Pending addition state: 1
131 # Progress at step 16:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
132 # lets go
133 # Time = 180000, Opcode = 010 (arenthesis), Operand = zzzz, Alu_Result = 0011,
    Pending Mult = 0, Pending Add = 0
134 # Time = 185000, Opcode = 010 (arenthesis), Operand = zzzz, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 0
135 # Stack content:
136 \textbf{# stack[0] = feb2}
137 # Pending multiplication state: 0
138 # Pending addition state: 0
139 # Progress at step 17:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
140 # lets go
141 # Time = 190000, Opcode = 100 (sh Operand), Operand = 0006, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 0
142 # Stack content:
143 \textbf{# stack[0] = feb2}
144 \textbf{# stack[1] = 0006}
145 # Pending multiplication state: 0
146 # Pending addition state: 0
147 # Progress at step 18:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
148 # lets go
149 # babababaababa

```



```

150 # Time = 200000, Opcode = 000 (Addition), Operand = zzzz, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 1
151 # Stack content:
152 \textbf{# stack[0] = feb2}
153 \textbf{# stack[1] = 0006}
154 # Pending multiplication state: 0
155 # Pending addition state: 1
156 # Progress at step 19:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
157 # lets go
158 # Time = 210000, Opcode = 100 (sh Operand), Operand = 0005, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 1
159 # Stack content:
160 \textbf{# stack[0] = feb2}
161 \textbf{# stack[1] = 0006}
162 \textbf{# stack[2] = 0005}
163 # Pending multiplication state: 0
164 # Pending addition state: 1
165 # Progress at step 20:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
166 # lets go
167 # Time = 220000, Opcode = 011 (arenthesis), Operand = zzzz, Alu_Result = zzzz,
    Pending Mult = 0, Pending Add = 1
168 # Time = 225000, Opcode = 011 (arenthesis), Operand = zzzz, Alu_Result = 0000,
    Pending Mult = 0, Pending Add = 1
169 # Stack content:
170 \textbf{# stack[0] = feb2}
171 \textbf{# stack[1] = 000b}
172 # Pending multiplication state: 0
173 # Pending addition state: 1
174 # Progress at step 21:  $(2 * 3 + (10 + 4 + 3) * -20 + (6 + 5) =$ 
175 # lets go
176 # Time = 230000, Opcode = 011 (arenthesis), Operand = zzzz, Alu_Result = 0000,
    Pending Mult = 0, Pending Add = 1
177 # Time = 235000, Opcode = 011 (arenthesis), Operand = zzzz, Alu_Result = 0006,
    Pending Mult = 0, Pending Add = 1
178 # Stack content:
179 \textbf{# stack[0] = febd}

```