ECE 111 Winter 2022
HW2
Hao Le A15547504

ALU

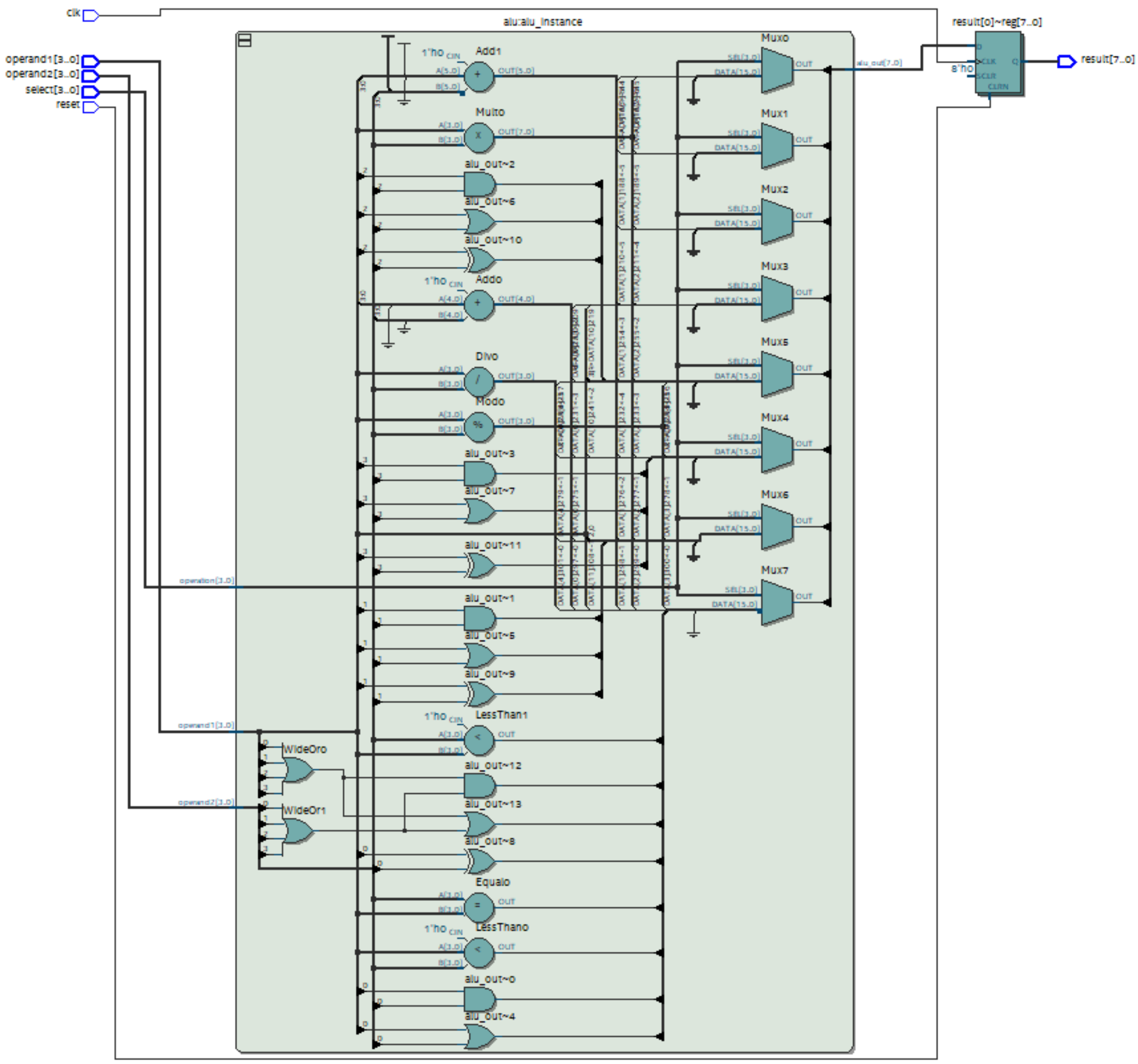
ALU module

```
// 1-bit ALU behavioral code
module alu // Module start declaration
#(parameter N=4) // Parameter declaration
(
   input logic[N-1:0] operand1, operand2,
   input logic[3:0] operation,
   output logic[(2*N)-1:0] alu_out
);

   // always procedural block describing alu operations
   always@(operand1 or operand2 or operation)
   begin
    case(operation)
       4'b0000 : alu_out = operand1 + operand2;
       4'b0001 : alu_out = operand1 - operand2;
       4'b0010 : alu_out = operand1 * operand2;
       4'b0011 : alu_out = operand1 % operand2;
       4'b0100 : alu_out = operand1 / operand2;
       4'b0101 : alu_out = operand1 & operand2;
       4'b0110 : alu_out = operand1 | operand2;
       4'b0111 : alu_out = operand1 ^ operand2;
       4'b1000 : alu_out = operand1 && operand2;
       4'b1001 : alu_out = operand1 || operand2;
       4'b1010 : alu_out = operand1 << 1;
       4'b1011 : alu_out = operand1 >> 1;
       4'b1100 : alu_out = operand1 == operand2;
       4'b1101 : alu_out = operand1 != operand2;
       4'b1110 : alu_out = operand1 < operand2;
       4'b1111 : alu_out = operand1 > operand2;
       default : alu_out = operand1 + operand2;
    endcase

   end
endmodule: alu
```

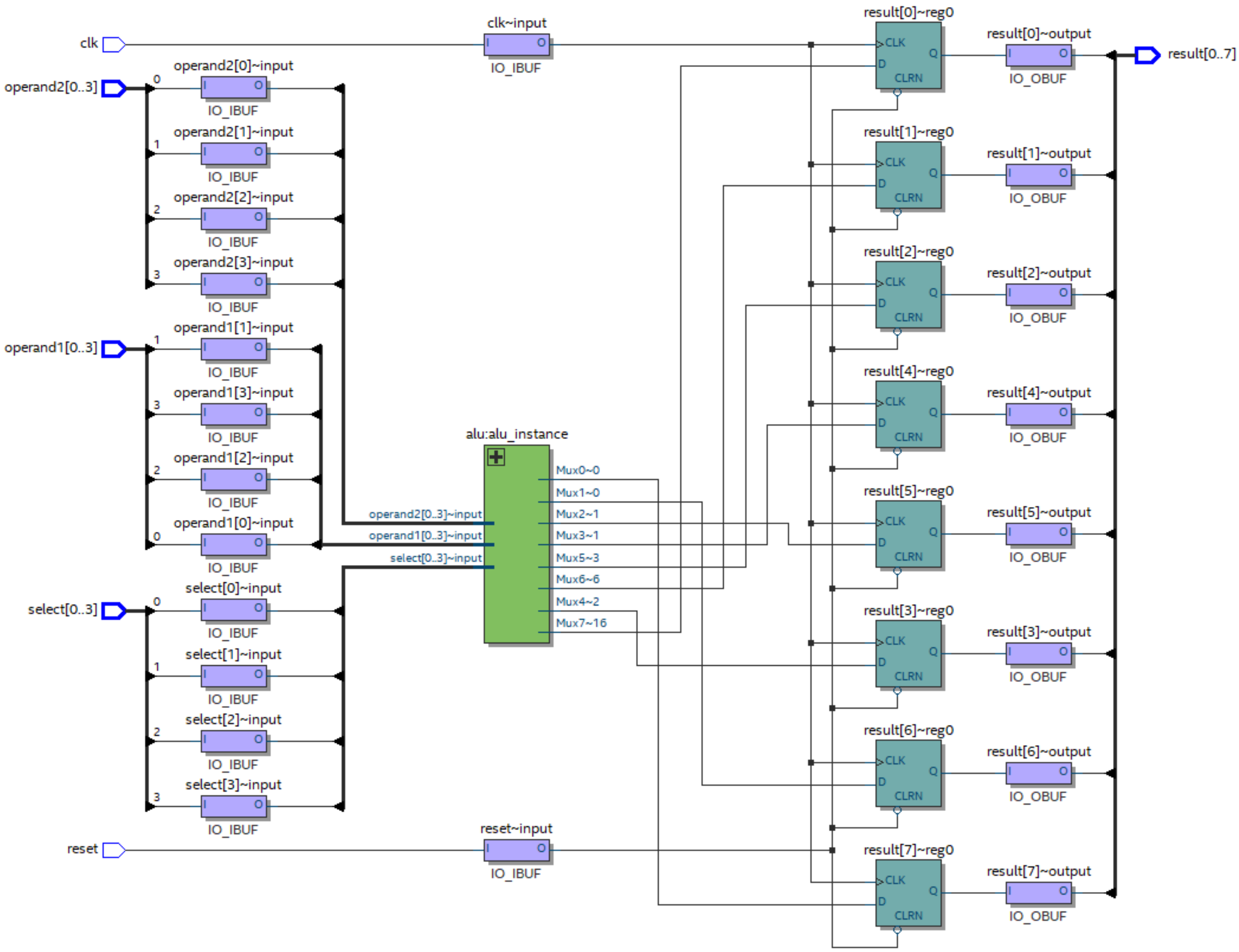
ALU top module

```
// N-bit ALU TOP RTL code
module alu_top // Module start declaration
#(parameter N=4) // Parameter declaration
(  input logic clk, reset,
   input logic[N-1:0]operand1, operand2,
   input logic[3:0] select,
   output logic[(2*N)-1:0] result
);

  // Local net declaration
   logic[(2*N):0] alu_out;

   alu #(.N(4)) alu_instance(
    .operand1(operand1),
    .operand2(operand2),
    .operation(select),
    .alu_out(alu_out)
   );

  // Adding flipflop at the output of ALU
   always@(posedge clk or posedge reset) begin
    if(reset == 1) begin
      result <= 0;
    end
    else begin
      result <= alu_out;
    end
   end


endmodule: alu_top // Module alu_top end declaration
```

RTL netlist (ALU top)



Post-mapping netlist (ALU top)

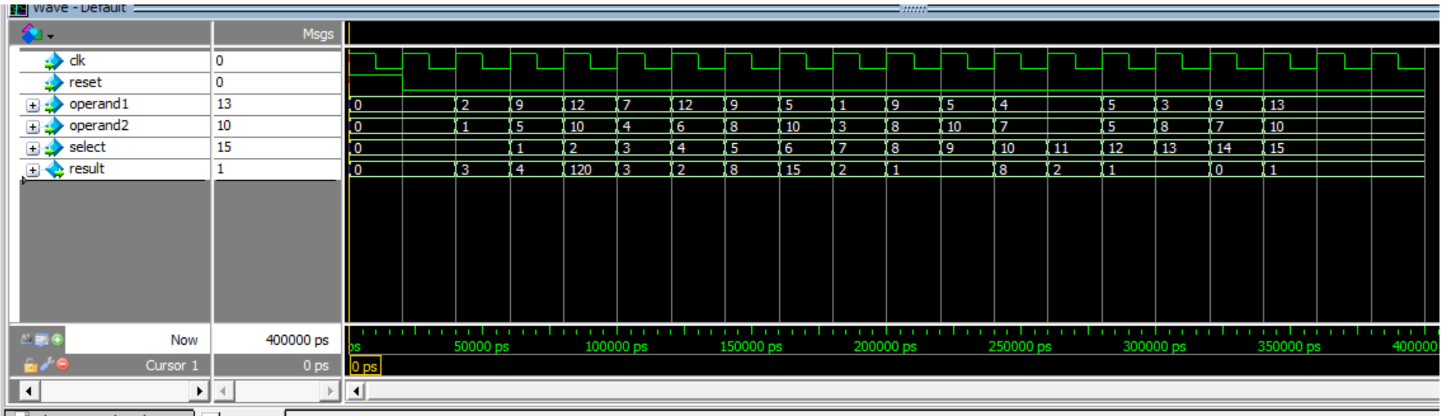


Resource usage (ALU top)

	Resource	Usage
1	Estimated ALUTs Used	113
1	-- Combinational ALUTs	113
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	8
3		
4	Estimated ALUTs Unavailable	9
1	-- Due to unpartnered combinational logic	9
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	113
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	3
2	-- 6 input functions	6
3	-- 5 input functions	21
4	-- 4 input functions	31
5	-- <=3 input functions	52
8		
9	Combinational ALUTs by mode	
1	-- normal mode	54
2	-- extended LUT mode	3
3	-- arithmetic mode	33
4	-- shared arithmetic mode	23
10		
11	Estimated ALUT/register pairs used	122
12		
13	Total registers	8
1	-- Dedicated logic registers	8
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	22
17		
18	DSP block 18-bit elements	0
19		

113 ALUTs were used.

Testbench simulation waveform



The simulation waveform shows the ALU performing all 16 operations correctly on the given operands, as follows:

1. 2 + 1 = 3
2. 9 – 5 = 4
3. 12 * 10 = 120
4. 7 % 4 = 3. This is because the remainder to 7 / 4 is 3.
5. 12 / 6 = 2
6. 9 & 8 = 1001 & 1000 = 1000 = 8 in decimal.
7. 5 | 10 = 0101 & 1010 = 1111 = 15 in decimal.
8. 1 ^ 3 = 0001 XOR 0011 = 0010 = 2 in decimal.
9. 9 && 8 = 1, because both operands are larger than 1, so this is a boolean True. Thus, True AND True make True, 1 in decimal.
10. 5 || 10 = 1, because 5 (True) OR 10 (True) is True, 1 in decimal.
11. 4 << 1 = 8, because 0100 shifted 1 to the left is 1000 which is 8 in decimal; this is equivalent to multiplying by 2.
12. 4 >> 1 = 2, because a logical shift to the right by 1 is equivalent to dividing by 2.
13. 5 == 5 = 1, because the equality is true.
14. 3 != 8 = 1, because the inequality is true.
15. 9 < 7 = 0, because 9 is not less than 7.
16. 13 < 10 = 1, because 13 is greater than 10.

Up-down Counter

Up counter module

```
// 4-bit counter RTL behavioral code
module up_counter     // Module start declaration
 // Parameter declaration, count signal width set to '4'
 #(parameter WIDTH=4)
 (
    input logic clk,
    input logic clear,
    output logic[WIDTH-1:0] count
 );

 // Local variable declaration
 logic[WIDTH-1:0] cnt_value;

 // always procedural block describing up counter behavior
 always @(posedge clk or posedge clear)
   begin
     if (clear == 1)
       cnt_value = 0;
     else
       cnt_value = cnt_value + 1;
   end

 // Counter value assigned to output port count
 assign count = cnt_value;
endmodule: up_counter   // Module end declaration
```

Down counter module

```
// 4-bit counter RTL code
module down_counter     // Module start declaration
 // Parameter declaration, count signal width set to '4'
 #(parameter WIDTH=4)
 (
    input logic clk,
    input logic clear,
    output logic[WIDTH-1:0] count
 );

 // Local variable declaration
 logic[WIDTH-1:0] cnt_value;

 // always procedural block describing up counter behavior
 always @(posedge clk or posedge clear)
   begin
     if (clear == 1)
       cnt_value = 15;
     else
       cnt_value = cnt_value - 1;
   end

 // Counter value assigned to output port count
 assign count = cnt_value;


endmodule: down_counter   // Module end declaration
```

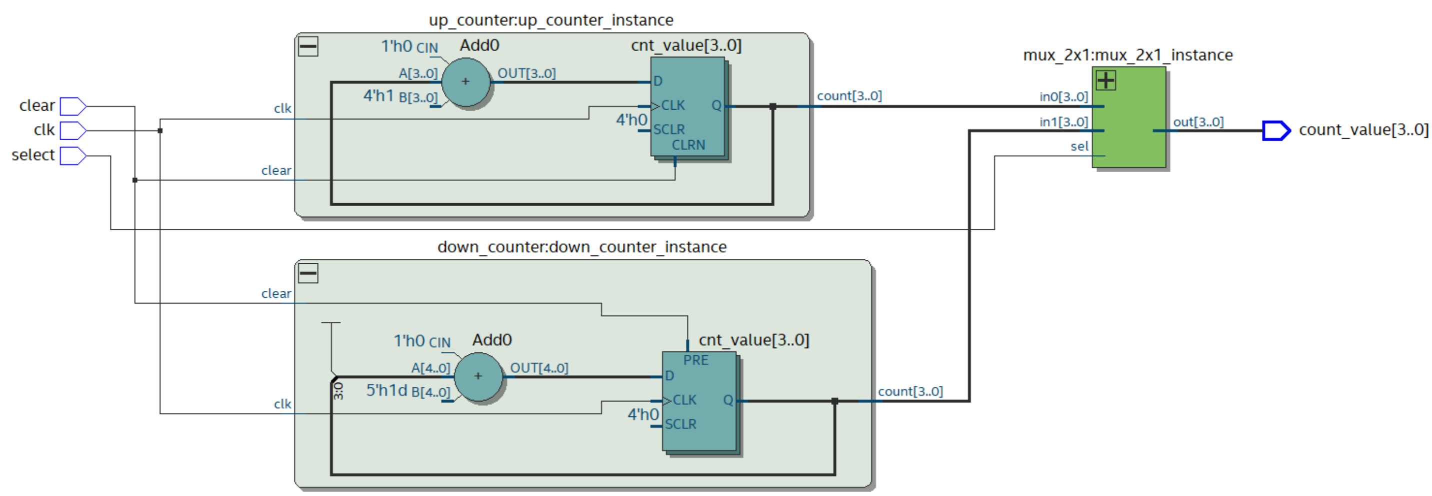
Mux 2x1 module

```
// 2to1 Multiplexor RTL code
module mux_2x1 #(parameter WIDTH=4)
(
   input logic[WIDTH-1:0] in0, // Student to change in0 width to 4
   input logic[WIDTH-1:0] in1, // Student to change in1 width to 4
   input logic sel,
   output logic[WIDTH-1:0] out // Student to change out width to 4
);

   // always procedural block describing 2to1 Multiplexor behavior
   always @(sel or in0 or in1)
   begin
      if(sel == 0)
         out = in0;
      else
        out = in1;
   end
endmodule
```

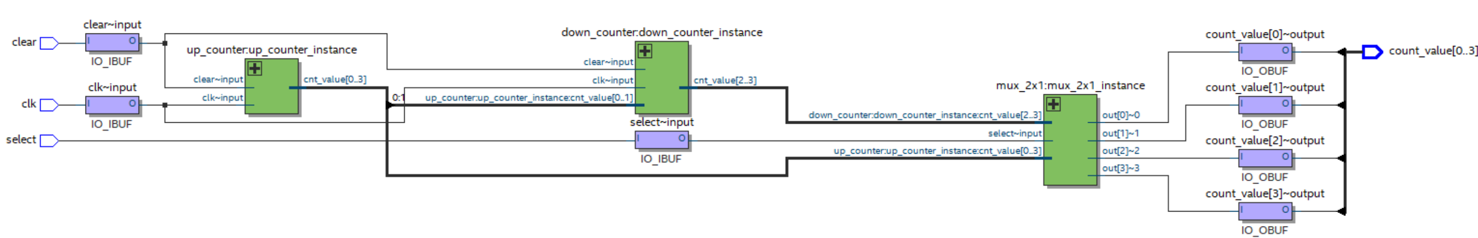
Up-down counter module

```
// 4-bit up and down counter RTL code
module up_down_counter     // Module start declaration
 // Parameter declaration, count signal width set to '4'
 #(parameter WIDTH=4)
 (
    input logic clk,
    input logic clear,
    input logic select,
    output logic[WIDTH-1:0] count_value
 );

 // Local variable declaration
 logic[WIDTH-1:0] up_count_value, down_count_value;


 up_counter #(.WIDTH(4)) up_counter_instance(
  .clk(clk),
  .clear(clear),
  .count(up_count_value)
 );

 down_counter #(.WIDTH(4)) down_counter_instance(
  .clk(clk),
  .clear(clear),
  .count(down_count_value)
 );

 mux_2x1 #(.WIDTH(4)) mux_2x1_instance(
  .in0(up_count_value),
  .in1(down_count_value),
  .sel(select),
  .out(count_value)
 );

endmodule: up_down_counter   // Module end declaration
```

RTL netlist (Up-down counter)



Post-mapping netlist (Up-down counter)

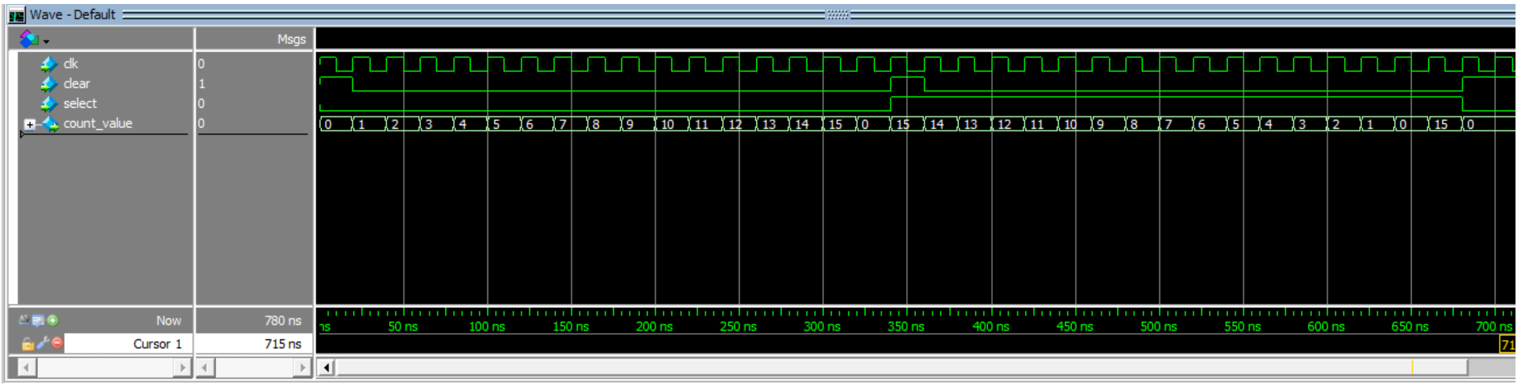


Resource usage (Up-down counter)

	Resource	Usage
1	Estimated ALUTs Used	10
1	-- Combinational ALUTs	10
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	6
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	10
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	0
4	-- 4 input functions	2
5	-- <=3 input functions	8
8		
9	Combinational ALUTs by mode	
1	-- normal mode	10
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	10
12		
13	Total registers	6
1	-- Dedicated logic registers	6
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	7
17		
18	DSP block 18-bit elements	0

10 ALUTs were used.

Testbench simulation waveform



The simulation shows that the up-down counter works. The first half of the simulation shows the counting up phase, indicated by the constant 0 input of the select line. The clear pulse sets the up counter back to 0, and on every positive edge of the clock, the output count_value increments, until it reaches 15 and wraps back to 0 due to the 4-bit width limitation. The second half of the simulation is the count down phase, indicated by the select line going to 1. The clear pulse now resets the count down to 15, which is followed by a decrement on every positive clock – when the count_value reaches 0, it wraps back to 15 because of the discussed bit width limitation.