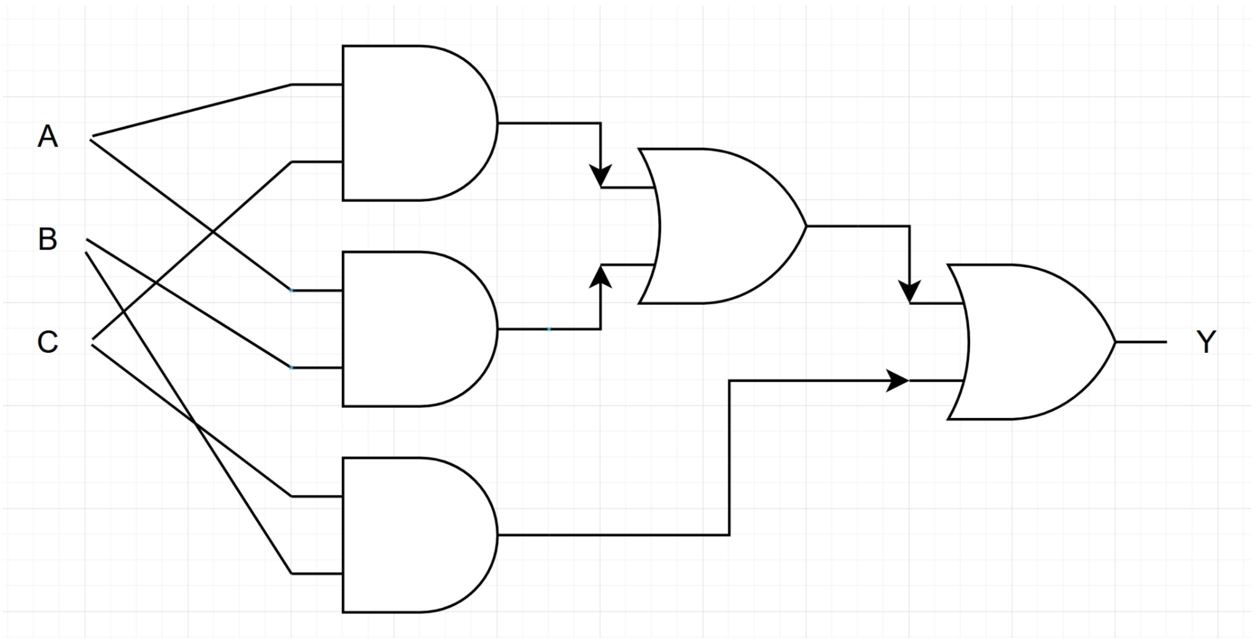
CS4093 - Assignment 2

Alexander Stradnic - 119377263

Majority Gate: A Majority Gate takes in a set of inputs and returns the output of the majority of the input signals. For example, with 3 inputs, if 2 or more inputs are enabled, then the output is 1, vice versa when 2 or more are disabled.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



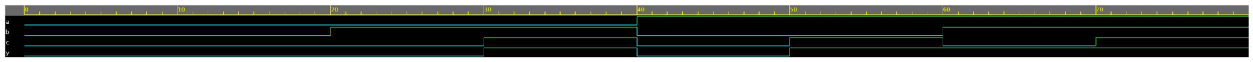
Verilog Code

```
module majority_gate (y,a,b,c);
  input  a,b,c;
  output y;
  assign y = (a&&b)||(b&&c)||(a&&c);
endmodule
```

Testbench Code

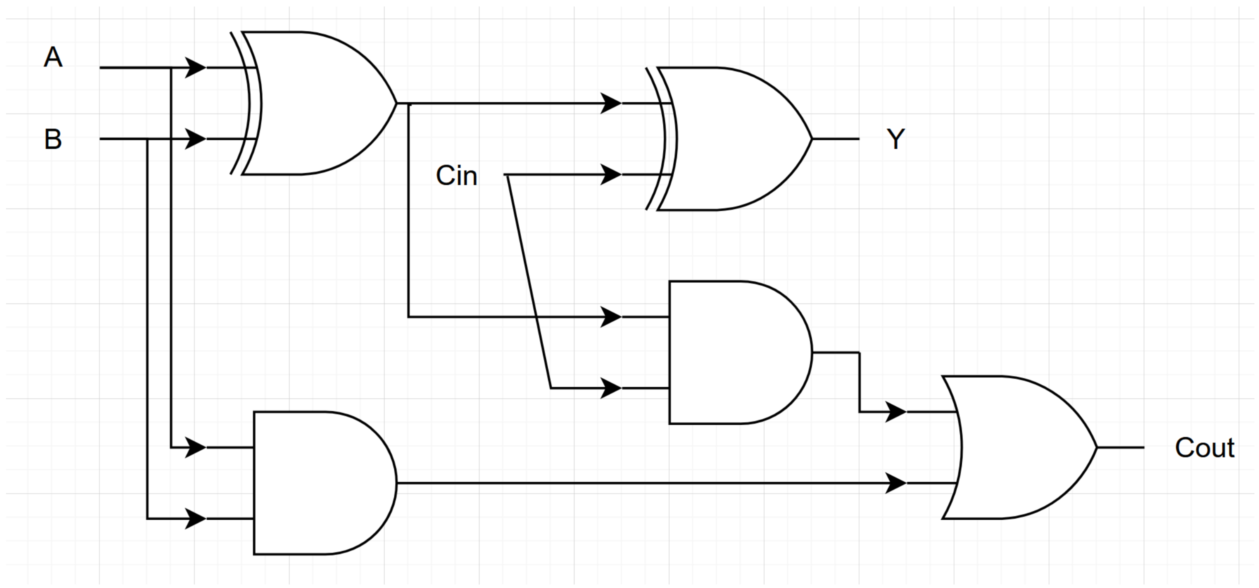
```
module TEST;
  reg a, b, c;
  wire y;

  majority_gate TEST (.y(y), .a(a), .b(b), .c(c));

  initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
      a=0;
      b=0;
    c=0;
      #10;
      a=0;
      b=0;
    c=0;
      #10;
      a=0;
      b=1;
    c=0;
      #10;
      a=0;
      b=1;
    c=1;
      #10;
    a=1;
    b=0;
    c=0;
    #10;
    a=1;
    b=0;
    c=1;
    #10;
    a=1;
    b=1;
    c=0;
    #10;
    a=1;
    b=1;
    c=1;
      #10;
  end
endmodule
```

Waveform



Full Adder: A Full Adder works in a similar manner to a half-adder, which takes two bits and outputs the sum and a carry if needed. However, an extra input is handled which is usually taken to be the carry bit from a previous adder.

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



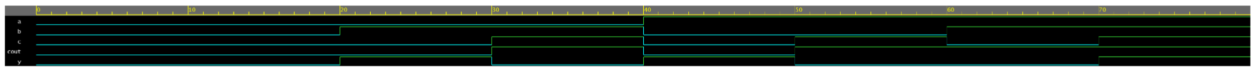
Verilog Code

```
module full_adder (y,cout,a,b,c);
  input  a,b,c;
  output y, cout;
  reg y, cout;
  always @(*) begin
  if(a + b + c == 0) begin
      y <= 0;
      cout <= 0;
  end else if(a + b + c == 1) begin
      y <= 1;
      cout <= 0;
  end else if(a + b + c == 2) begin
      y <= 0;
      cout <= 1;
  end else if(a + b + c == 3) begin
      y <= 1;
      cout <= 1;
  end
  end
endmodule
```

Testbench Code:

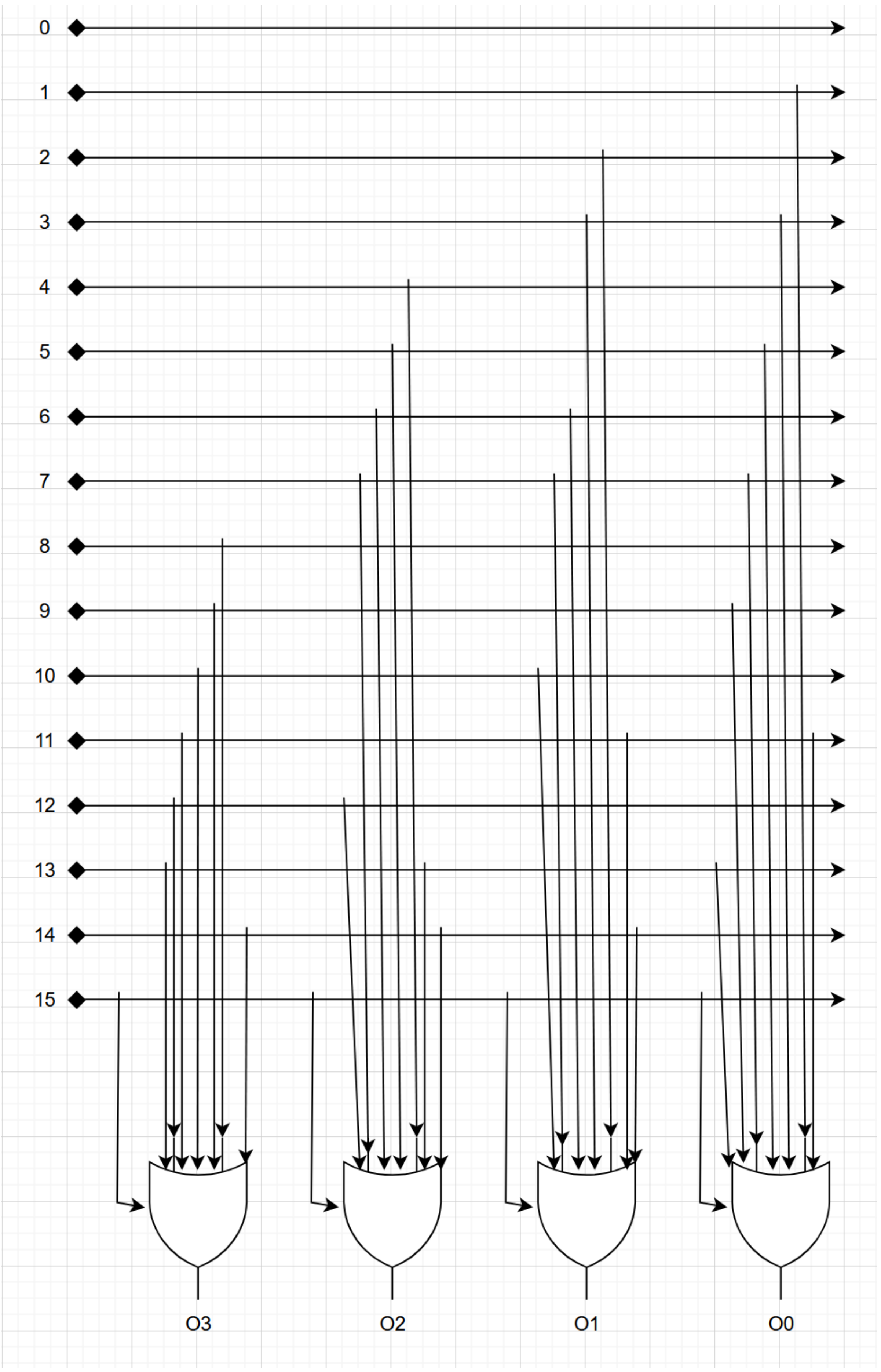
```
module TEST;
  reg a, b, c;
  wire y, cout;
  full_adder TEST (.y(y), .cout(cout), .a(a), .b(b), .c(c));

  initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
      a=0;
      b=0;
    c=0;
      #10;
      a=0;
      b=0;
    c=0;
      #10;
      a=0;
      b=1;
    c=0;
      #10;
      a=0;
      b=1;
    c=1;
      #10;
    a=1;
    b=0;
    c=0;
    #10;
    a=1;
    b=0;
    c=1;
    #10;
    a=1;
    b=1;
    c=0;
    #10;
    a=1;
    b=1;
    c=1;
      #10;
  end
endmodule
```

Waveform



4:16 Encoder: A 4:16 Encoder works by converting an a series of 16 inputs to a 4-bit encoded number

I15	I14	I13	I12	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0	O3	O2	O1	O0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
O3
O2
O1
O0

Verilog Code

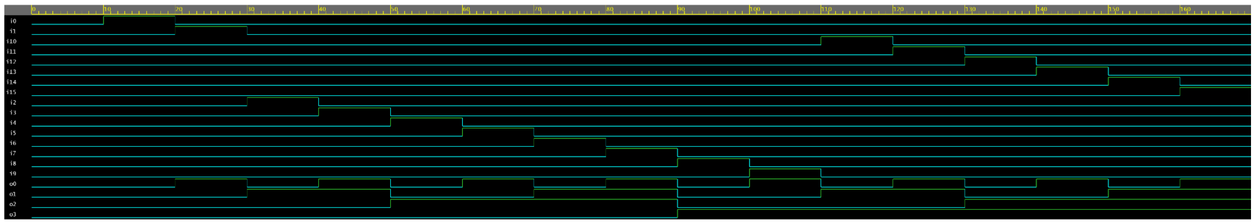
```verilog
module encoder
(i0,i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15,o3,o2,o1,o0);
  input  i0,i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15;
  output o3,o2,o1,o0;
  reg o3,o2,o1,o0;
  always @(*) begin
  o0 <= 0;
  o1 <= 0;
  o2 <= 0;
  o3 <= 0;

  if(i1 | i3 | i5 | i7 | i9 | i11 | i13 | i15) begin
      o0 <= 1;
  end
      if(i2 | i3 | i6 | i7 | i10 | i11 | i14 | i15) begin
      o1 <= 1;
  end
      if(i4 | i5 | i6 | i7 | i12 | i13 | i14 | i15) begin
      o2 <= 1;
  end
      if(i8 | i9 | i10 | i11 | i12 | i13 | i14 | i15) begin
      o3 <= 1;
  end
  end
endmodule
```

Testbench Code

```verilog
module TEST;
  reg i0,i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15;
  wire o3,o2,o1,o0;

  encoder TEST (.i0(i0), .i1(i1), .i2(i2), .i3(i3), .i4(i4), .i5(i5),
.i6(i6), .i7(i7), .i8(i8), .i9(i9), .i10(i10), .i11(i11), .i12(i12),
.i13(i13), .i14(i14), .i15(i15), .o3(o3), .o2(o2), .o1(o1), .o0(o0));

  initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
      {i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14,
i15} = 16'b0;
      #10;
      i0 = 1;
      #10;
    i0 = 0;
      i1 = 1;
      #10;
      i1 = 0;
      i2 = 1;
      #10;
      i2 = 0;
    i3 = 1;
      #10;
      i3 = 0;
    i4 = 1;
    #10;
    i4 = 0;
      i5 = 1;
    #10;
      i5 = 0;
    i6 = 1;
    #10;
      i6 = 0;
    i7 = 1;
      #10;
      i7 = 0;
    i8 = 1;
      #10;
      i8 = 0;
    i9 = 1;
      #10;
      i9 = 0;
    i10 = 1;
      #10;
      i10 = 0;
    i11 = 1;
      #10;
      i11 = 0;
    i12 = 1;
      #10;
      i12 = 0;
    i13 = 1;
      #10;
      i13 = 0;
    i14 = 1;
```

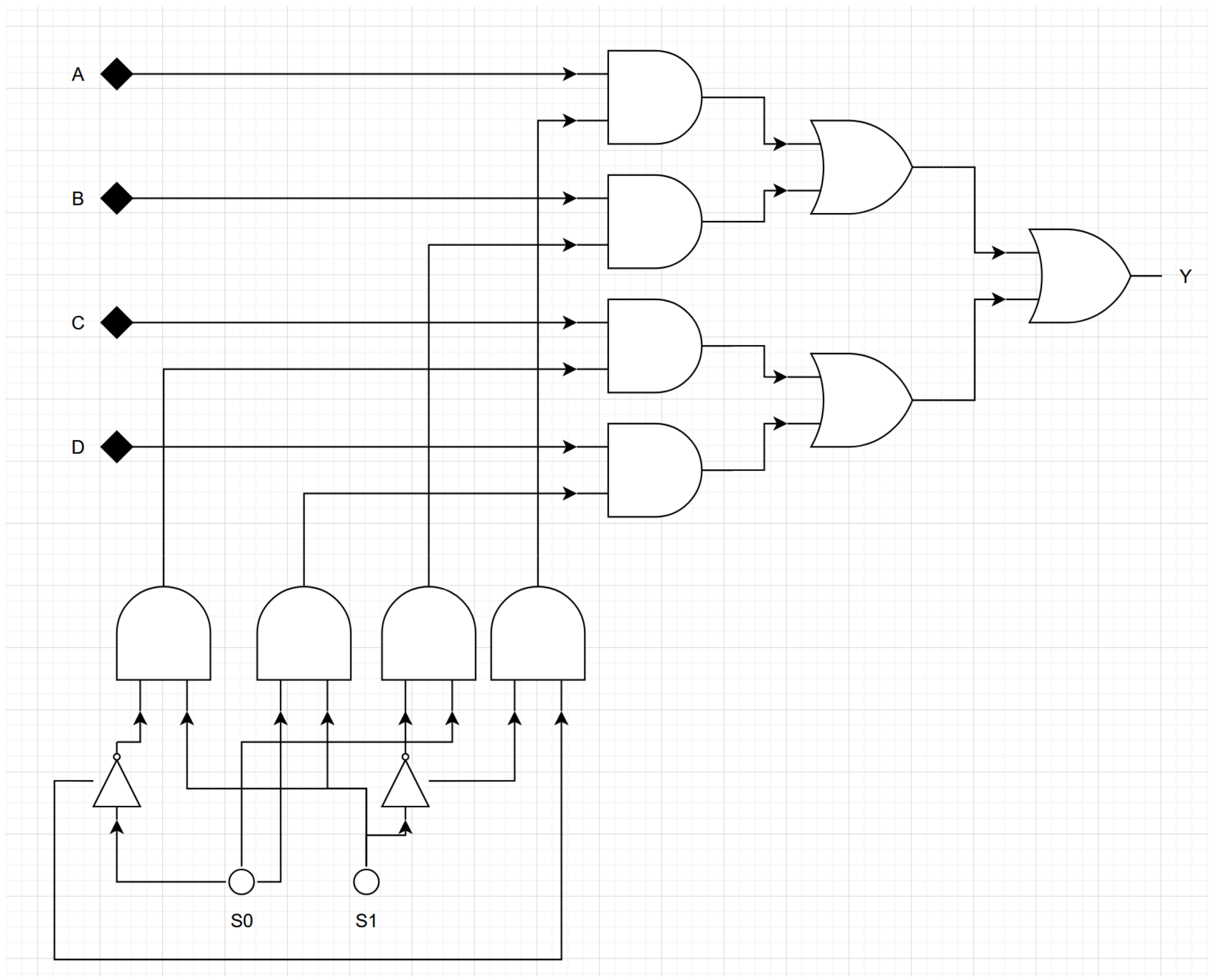
```
      #10;
      i14 = 0;
    i15 = 1;
      #10;
  end
endmodule
```

Waveform



Multiplexer: A multiplexer acts like a switch between inputs, using selector switch line(s) to choose which input is outputted.

A	B	C	D	S1	S0	Y
				0	0	A
				0	1	B
				1	0	C
				1	1	D



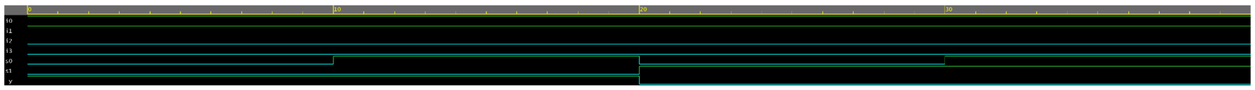
Verilog Code

```
module multiplexer (i0,i1,i2,i3,s0,s1,y);
  input  i0,i1,i2,i3,s0,s1;
  output y;
  reg y;
  always @(*) begin
      if(s0) begin
      if(s1) begin
      y <= i3;
      end
      else begin
       y <= i1;
      end
      end else if(s1) begin
      y <= i2;
      end else begin
      y <= i0;
      end
  end
endmodule
```

Testbench Code

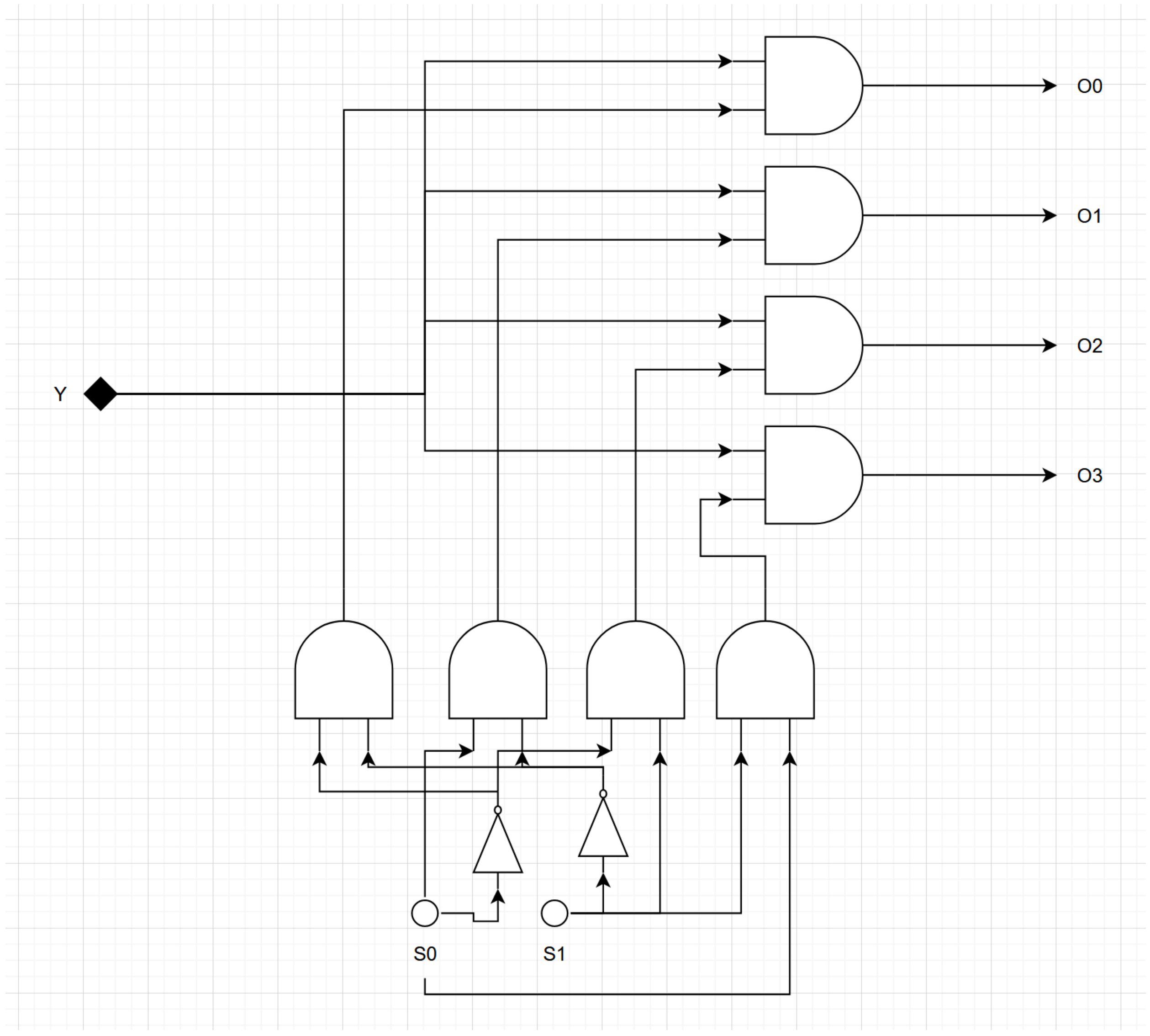
```
module TEST;
  reg i0,i1,i2,i3,s0,s1;
  wire y;

  multiplexer TEST (.i0(i0), .i1(i1), .i2(i2), .i3(i3), .s0(s0), .s1(s1),
.y(y));

  initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
      i0 = 1;
      i1 = 1;
      {s0, s1, i2, i3} = 4'b0;
      // [s1][s0] = 00, inc from 00 to 11
      #10;
      s0 = 1;
      #10;
    s0 = 0;
      s1 = 1;
      #10;
      s0 = 1;
      #10;
  end
endmodule
```

Waveform



Demultiplexer: A demultiplexer acts in the opposite manner to a multiplexer, with a single input being routed to a selected output line.

	S1	S0	O0	O1	O2	O3
Y	0	0	Y	0	0	0
	0	1	0	Y	0	0
	1	0	0	0	Y	0
	1	1	0	0	0	Y



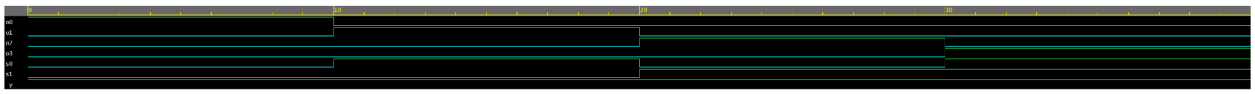
Verilog Code

```
module demultiplexer (o0,o1,o2,o3,s0,s1,y);
  input  y,s0,s1;
  output o0,o1,o2,o3;
  reg o0,o1,o2,o3;
  always @(*) begin
      {o0, o1, o2, o3} = 4'b0;
      if(s0) begin
      if(s1) begin
      o3 <= y;
      end
      else begin
       o1 <= y;
      end
      end else if(s1) begin
      o2 <= y;
      end else begin
      o0 <= y;
      end
  end
endmodule
```

Testbench Code

```
module TEST;
  reg y, s0, s1;
  wire o0, o1, o2, o3;

  demultiplexer TEST (.o0(o0), .o1(o1), .o2(o2), .o3(o3), .s0(s0), .s1(s1),
.y(y));

  initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
      y = 1;
      s0 = 0;
      s1 = 0;
      // [s1][s0] = 00, inc from 00 to 11
      #10;
      s0 = 1;
      #10;
    s0 = 0;
      s1 = 1;
      #10;
      s0 = 1;
      #10;
  end
endmodule
```

Waveform



Down Counter: A program which decrements an input (in this case, the hex number FFFF) to 0.

Verilog Code

```
module down_counter(input clk, reset, output[15:0] counter, output[15:0]
counter2
);
  reg [15:0] counter_down;
  reg[15:0] counter_down2;
always @(posedge clk or posedge reset)
begin
  if(reset) begin
counter_down <= 8'hFF;
counter_down2 <= 8'hFF;
  end else if (counter_down2 == 8'h00) begin
counter_down <= counter_down - 8'h1;
counter_down2 <= 8'hFF;
  end else begin
counter_down2 <= counter_down2 - 8'h1;
  end
end
assign counter2 = counter_down2;
assign counter = counter_down;
endmodule
```

Testbench Code

```
module downcounter_testbench();
reg clk, reset;
  wire [15:0] counter;
  wire [15:0] counter2;
  down_counter dut(clk, reset, counter, counter2);
integer iterations;
initial begin
$dumpfile("dump.vcd");
$dumpvars(1);
clk=0;
```

```
iterations = 0;
while (iterations < (65536*2)) begin
#5;
iterations = iterations + 1;
clk=~clk;
end
end
initial begin
reset=1;
#20;
reset=0;
end
endmodule
```