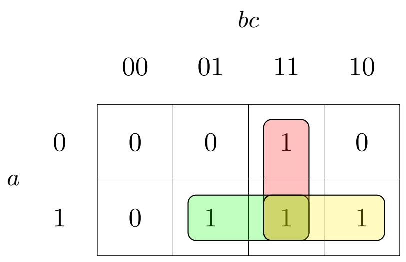
CS 354
Section 02
Fall 2019
Christopher Schmitt — Assignment 1

## Problem 1

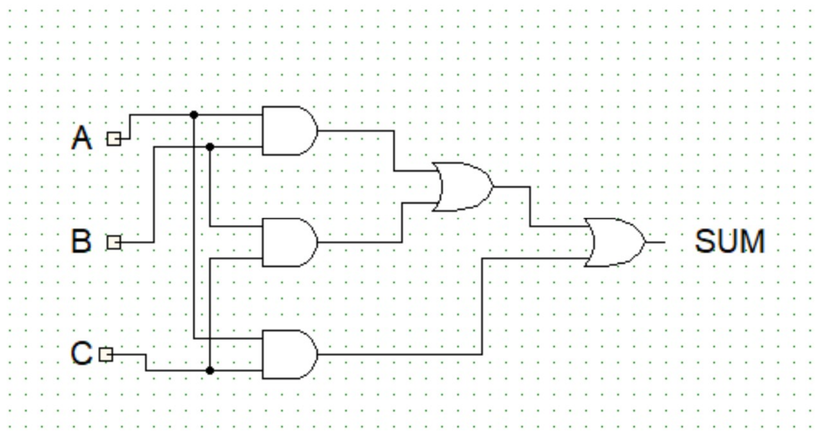
The majority function produces a false value when $\frac{n}{2}$ of the provided inputs are false (where $n$ is the number of inputs). This function can be trivially implemented using three AND gates and two OR gates.

| $a$ | $b$ | $c$ | $sum$ |
|---|---|---|---|
| 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** |



```
module Majority(
  input wire a,
  input wire b,
  input wire c,
  output wire sum
);

  // Compute the products
  and(a_and_b, a, b);
  and(a_and_c, a, c);
  and(b_and_c, b, c);

  // Cascade the OR
  or(buffer, a_and_b, a_and_c);
  or(sum, buffer, b_and_c);

endmodule // Majority
```

```
module MajorityTest;

  reg a;
  reg b;
  reg c;
  wire sum;

  Majority majority(a, b, c, sum);

  initial begin
    $monitor("a = %d, b = %d, c = %d, sum = %d", a, b, c, sum);
    a = 0; b = 0; c = 0;
    #20; a = 0; b = 0; c = 1;
    #20; a = 0; b = 1; c = 0;
    #20; a = 0; b = 1; c = 1;
    #20; a = 1; b = 0; c = 0;
    #20; a = 1; b = 0; c = 1;
    #20; a = 1; b = 1; c = 0;
    #20; a = 1; b = 1; c = 1;
  end

endmodule // MajorityTest
```



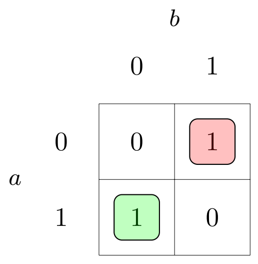
```
iverilog.exe src/majority.v && vvp.exe a.out
a = 0, b = 0, c = 0, sum = 0
a = 0, b = 0, c = 1, sum = 0
a = 0, b = 1, c = 0, sum = 0
a = 0, b = 1, c = 1, sum = 1
a = 1, b = 0, c = 0, sum = 0
```

```
a = 1, b = 0, c = 1, sum = 1
a = 1, b = 1, c = 0, sum = 1
a = 1, b = 1, c = 1, sum = 1
```

## Problem 2

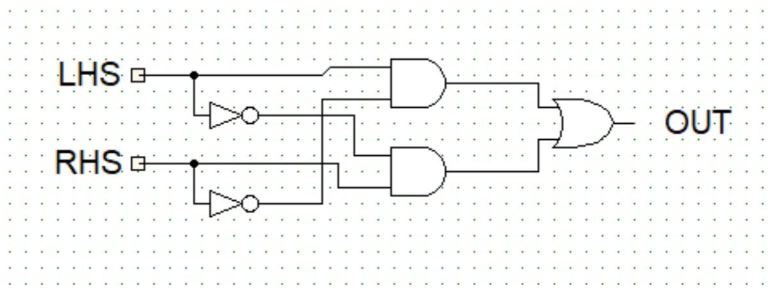
A conditional inverter is an XOR gate with exactly two unique inputs. It, like all functions, can be broken down into ANDs, ORs, and NOTs.

| $a$ | $b$ | $out$ |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 1 | 1 | **0** |



```
module Inverter(
  input wire lhs,
  input wire rhs,
  output wire out
);

  // Invert left and right hand sides
  not(lhs_prime, lhs);
  not(rhs_prime, rhs);

  // Compute products
  and(left, lhs_prime, rhs);
  and(right, lhs, rhs_prime);

  // Compute sum
  or(out, left, right);
```

```
endmodule // Inverter



module InverterTest;

  reg lhs;
  reg rhs;
  wire out;

  Inverter inverter(lhs, rhs, out);

  initial begin
    $monitor("lhs = %d, rhs = %d, out = %d", lhs, rhs, out);
    lhs = 0; rhs = 0;
    #20; lhs = 0; rhs = 1;
    #20; lhs = 1; rhs = 0;
    #20; lhs = 1; rhs = 1;
  end

endmodule // InverterTest
```

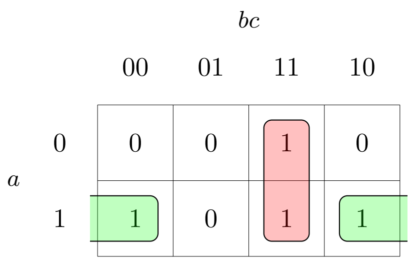


```
iverilog.exe src/inverter.v && vvp.exe a.out
lhs = 0, rhs = 0, out = 0
lhs = 0, rhs = 1, out = 1
lhs = 1, rhs = 0, out = 1
lhs = 1, rhs = 1, out = 0
```

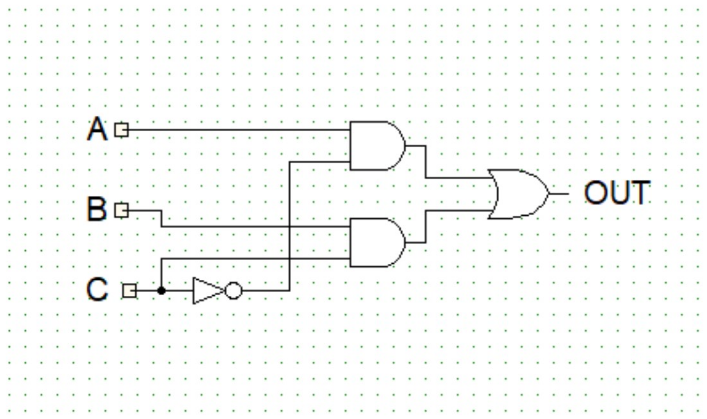
## Problem 3

A simple two input multiplexer is a $2 \times 1$ multiplexer where $a$ and $b$ are inputs and $c$ is the control. This function can be implemented easily using only AND, OR, and NOT gates.

| $a$ | $b$ | $c$ | $out)$ |
|---|---|---|---|
| 0 | 0 | 0 | **0** |
| 0 | 0 | 1 | **0** |
| 0 | 1 | 0 | **0** |
| 0 | 1 | 1 | **1** |
| 1 | 0 | 0 | **1** |
| 1 | 0 | 1 | **0** |
| 1 | 1 | 0 | **1** |
| 1 | 1 | 1 | **1** |



```
module Multiplexer(
  input wire a,
  input wire b,
  input wire c,
  output wire out
);

  // Invert control input
  not(c_prime, c);

  // Compute products
  and(left, a, c_prime);
  and(right, b, c);

  // Compute sum
  or(out, left, right);

endmodule // Multiplexer



module MultiplexerTest;

  reg a;
  reg b;
  reg c;
  wire out;

  Multiplexer multiplexer(a, b, c, out);

  initial begin
    $monitor("a = %d, b = %d, c = %d, out = %d", a, b, c, out);
    a = 0; b = 0; c = 0;
    #20; a = 0; b = 0; c = 1;
    #20; a = 0; b = 1; c = 0;
    #20; a = 0; b = 1; c = 1;
    #20; a = 1; b = 0; c = 0;
    #20; a = 1; b = 0; c = 1;
    #20; a = 1; b = 1; c = 0;
    #20; a = 1; b = 1; c = 1;
  end

endmodule // MultiplexerTest
```



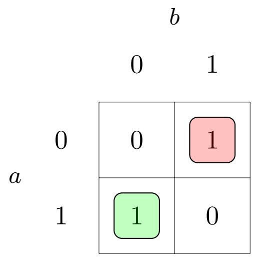
```
iverilog.exe src/multiplexer.v && vvp.exe a.out
a = 0, b = 0, c = 0, out = 0
a = 0, b = 0, c = 1, out = 0
a = 0, b = 1, c = 0, out = 0
a = 0, b = 1, c = 1, out = 1
a = 1, b = 0, c = 0, out = 1
a = 1, b = 0, c = 1, out = 0
a = 1, b = 1, c = 0, out = 1
a = 1, b = 1, c = 1, out = 1
```

## Problem 4

The one bit half adder is a circuit with two inputs and two outputs. It computes both the sum and the carry given both the left and right hand sides as binary inputs. The construction of this circuit can be greatly simplified through the use of an XOR gate.

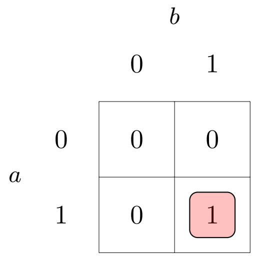
Sum (conditional inverter)

| $a$ | $b$ | $sum$ |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 1 | 1 | **0** |

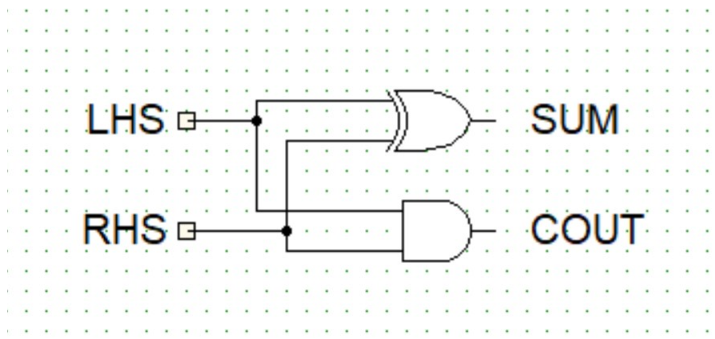


Carry (AND)

| $a$ | $b$ | $Carry$ |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **0** |
| 1 | 0 | **0** |
| 1 | 1 | **1** |



```verilog
module OneBitHalfAdder(
  input wire lhs,
  input wire rhs,
  output wire sum,
  output wire carry
);

  // Compute the sum
  xor(sum, lhs, rhs);

  // Compute the carry
  and(carry, lhs, rhs);

endmodule // OneBitHalfAdder


module OneBitHalfAdderTest;

  reg lhs;
  reg rhs;
  wire sum;
  wire carry;

  OneBitHalfAdder oneBitHalfAdder(lhs, rhs, sum, carry);

  initial begin
    $monitor("lhs = %d, rhs = %d, sum = %d, carry = %d", lhs,
        rhs, sum, carry);
    lhs = 0; rhs = 0;
    #20; lhs = 0; rhs = 1;
    #20; lhs = 1; rhs = 0;
    #20; lhs = 1; rhs = 1;
  end

endmodule // OneBitHalfAdderTest
```

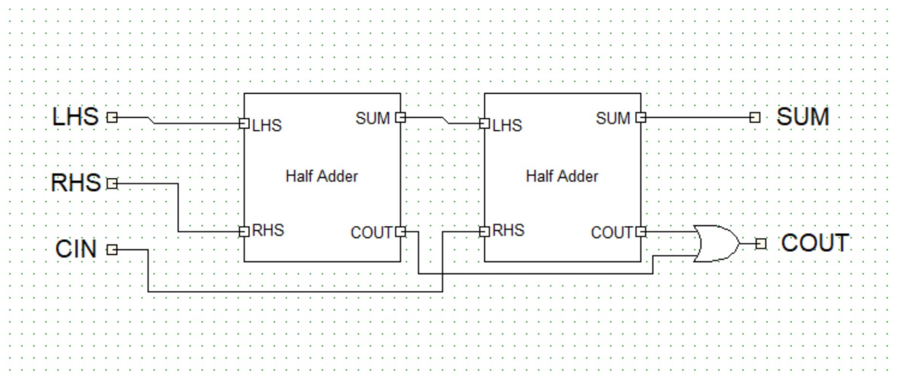


```
iverilog.exe src/one_bit_half_adder.v && vvp.exe a.out
lhs = 0, rhs = 0, sum = 0, carry = 0
lhs = 0, rhs = 1, sum = 1, carry = 0
lhs = 1, rhs = 0, sum = 1, carry = 0
lhs = 1, rhs = 1, sum = 0, carry = 1
```

## Problem 5

By cascading the sums of two half-adder modules and ORing the carries of both modules, a full adder can be constructed out of two simple half adders.

```
module OneBitHalfAdder(
  input wire lhs,
  input wire rhs,
  output wire sum,
  output wire carry
);

  // Compute the sum
  xor(sum, lhs, rhs);

  // Compute the carry
  and(carry, lhs, rhs);

endmodule // OneBitHalfAdder


module OneBitFullAdder(
  input wire cin,
  input wire lhs,
  input wire rhs,
  output wire sum,
  output wire cout
);

  // Initilize two half-adders
  OneBitHalfAdder halfAdderA(lhs, rhs, buffer, carry_left);
  OneBitHalfAdder halfAdderB(buffer, cin, sum, carry_right);

  // Compute the carry
  or(cout, carry_left, carry_right);

endmodule // OneBitFullAdder


module OneBitFullAdderTest;

  reg cin;
  reg lhs;
  reg rhs;
  wire sum;
  wire cout;

  OneBitFullAdder oneBitFullAdder(cin, lhs, rhs, sum, cout);

  initial begin
    $monitor("cin = %d, lhs = %d, rhs = %d, sum = %d, cout = %d",
        cin, lhs, rhs, sum, cout);
    cin = 0; lhs = 0; rhs = 0;
    #20; cin = 0; lhs = 0; rhs = 1;
    #20; cin = 0; lhs = 1; rhs = 0;
    #20; cin = 0; lhs = 1; rhs = 1;
    #20; cin = 1; lhs = 0; rhs = 0;
    #20; cin = 1; lhs = 0; rhs = 1;
    #20; cin = 1; lhs = 1; rhs = 0;
    #20; cin = 1; lhs = 1; rhs = 1;
  end

endmodule // OneBitFullAdderTest
```

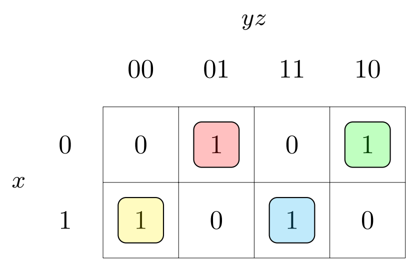


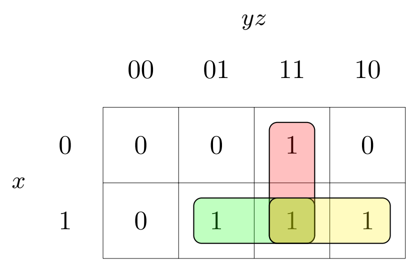
```
iverilog.exe src/one_bit_full_adder.v && vvp.exe a.out
cin = 0, lhs = 0, rhs = 0, sum = 0, cout = 0
cin = 0, lhs = 0, rhs = 1, sum = 1, cout = 0
cin = 0, lhs = 1, rhs = 0, sum = 1, cout = 0
cin = 0, lhs = 1, rhs = 1, sum = 0, cout = 1
cin = 1, lhs = 0, rhs = 0, sum = 1, cout = 0
cin = 1, lhs = 0, rhs = 1, sum = 0, cout = 1
cin = 1, lhs = 1, rhs = 0, sum = 0, cout = 1
cin = 1, lhs = 1, rhs = 1, sum = 1, cout = 1
```

## Problem 6

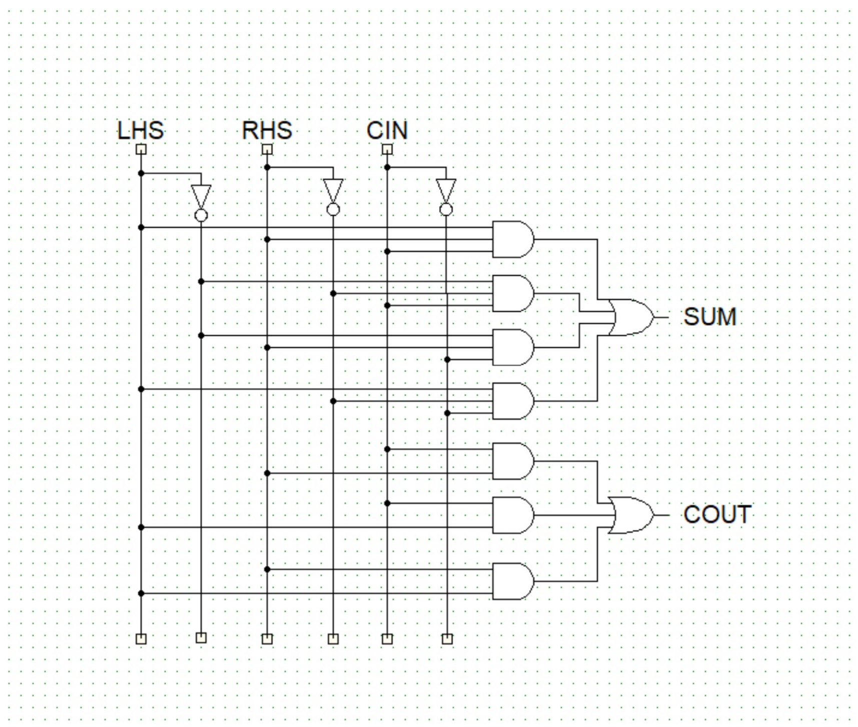
A full adder can be implemented directly (without chaining half-adders). The full adder takes three inputs (left-hand-side, right-hand-side, and carry-in). and produces sum and carry-out values.

| $x$ | $y$ | $z$ | $sum$ | $cout$ |
|---|---|---|---|---|
| 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** |





```
module OneBitFullAdderDirect(
  input wire lhs,
  input wire rhs,
  input wire cin,
  output wire sum,
  output wire cout
);

  // Invert inputs
  not(lhs_, lhs);
  not(rhs_, rhs);
  not(cin_, cin);

  // Compute sum products
  and(lhs_rhs_, lhs_, rhs_);
  and(lhs_cin_, lhs_, cin_);
  and(rhs_cin_, rhs_, cin_);
  and(rhslhs, rhs, lhs);

  and(lhs_rhs_cin, lhs_rhs_, cin);
  and(lhs_cin_rhs, lhs_cin_, rhs);
  and(rhs_cin_lhs, rhs_cin_, lhs);
  and(rhslhscin, rhslhs, cin);
  // Compute cout products
  and(rhs_and_cin, rhs, cin);
  and(lhs_and_cin, lhs, cin);
  and(lhs_and_rhs, lhs, rhs);

  // Compute sum sums
  or(left_sum_buffer, lhs_rhs_cin, lhs_cin_rhs);
  or(right_sum_buffer, rhs_cin_lhs, rhslhscin);
  or(sum, left_sum_buffer, right_sum_buffer);

  // Compute cout sums
  or(cout_buffer, rhs_and_cin, lhs_and_cin);
  or(cout, cout_buffer, lhs_and_rhs);

endmodule // OneBitFullAdderDirect


module OneBitFullAdderDirectTest;

  reg cin;
  reg lhs;
  reg rhs;
  wire sum;
  wire cout;

  OneBitFullAdderDirect oneBitFullAdderDirect(lhs, rhs, cin,
      sum, cout);

  initial begin
    $monitor("cin = %d, lhs = %d, rhs = %d, sum = %d, cout = %d",
        cin, lhs, rhs, sum, cout);
    cin = 0; lhs = 0; rhs = 0;
    #20; cin = 0; lhs = 0; rhs = 1;
    #20; cin = 0; lhs = 1; rhs = 0;
    #20; cin = 0; lhs = 1; rhs = 1;
    #20; cin = 1; lhs = 0; rhs = 0;
    #20; cin = 1; lhs = 0; rhs = 1;
    #20; cin = 1; lhs = 1; rhs = 0;
    #20; cin = 1; lhs = 1; rhs = 1;
  end

endmodule // OneBitFullAdderDirectTest
```

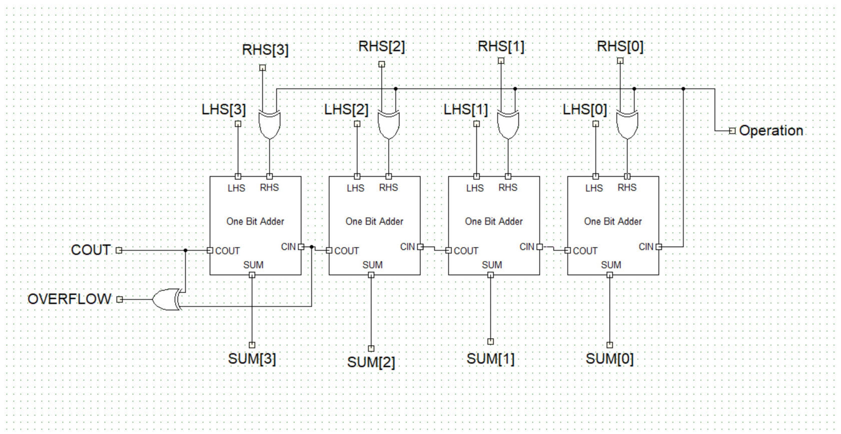


```
iverilog.exe src/one_bit_full_adder_direct.v && vvp.exe
    a.out
cin = 0, lhs = 0, rhs = 0, sum = 0, cout = 0
cin = 0, lhs = 0, rhs = 1, sum = 1, cout = 0
cin = 0, lhs = 1, rhs = 0, sum = 1, cout = 0
cin = 0, lhs = 1, rhs = 1, sum = 0, cout = 1
cin = 1, lhs = 0, rhs = 0, sum = 1, cout = 0
cin = 1, lhs = 0, rhs = 1, sum = 0, cout = 1
cin = 1, lhs = 1, rhs = 0, sum = 0, cout = 1
cin = 1, lhs = 1, rhs = 1, sum = 1, cout = 1
```

## Problem 7

By using a control bit and XORing the right hand side, a cascaded adder can be used as a subtractor (by adding the 2's complement inverse). By cascading four one-bit full adders, a four-bit adder/subtractor can be created. If the c2 and cout lines have different values, then an overflow must have occured.

```
module OneBitFullAdderDirect(
  input wire lhs,
  input wire rhs,
  input wire cin,
  output wire sum,
  output wire cout
);

  // Invert inputs
  not(lhs_, lhs);
  not(rhs_, rhs);
  not(cin_, cin);

  // Compute sum products
  and(lhs_rhs_, lhs_, rhs_);
  and(lhs_cin_, lhs_, cin_);
  and(rhs_cin_, rhs_, cin_);
  and(rhslhs, rhs, lhs);

  and(lhs_rhs_cin, lhs_rhs_, cin);
  and(lhs_cin_rhs, lhs_cin_, rhs);
  and(rhs_cin_lhs, rhs_cin_, lhs);
  and(rhslhscin, rhslhs, cin);

  // Compute cout products
  and(rhs_and_cin, rhs, cin);
  and(lhs_and_cin, lhs, cin);
  and(lhs_and_rhs, lhs, rhs);

  // Compute sum sums
  or(left_sum_buffer, lhs_rhs_cin, lhs_cin_rhs);
  or(right_sum_buffer, rhs_cin_lhs, rhslhscin);
  or(sum, left_sum_buffer, right_sum_buffer);

  // Compute cout sums
  or(cout_buffer, rhs_and_cin, lhs_and_cin);
  or(cout, cout_buffer, lhs_and_rhs);

endmodule // OneBitFullAdderDirect


module FourBitAdder(
  input wire operation,
  input wire [3:0] lhs,
  input wire [3:0] rhs,
  output wire [3:0] sum,
  output wire cout,
  output wire overflow
);

  // Carry wires
  wire c0;
  wire c1;
  wire c2;

  // XOR the right hand side with the operation
  xor(rhs_0, rhs[0], operation);
  xor(rhs_1, rhs[1], operation);
  xor(rhs_2, rhs[2], operation);
  xor(rhs_3, rhs[3], operation);

  // Full adder modules
  OneBitFullAdderDirect adder_0(lhs[0], rhs_0, operation,
      sum[0], c0);
  OneBitFullAdderDirect adder_1(lhs[1], rhs_1, c0, sum[1], c1);
  OneBitFullAdderDirect adder_2(lhs[2], rhs_2, c1, sum[2], c2);
  OneBitFullAdderDirect adder_3(lhs[3], rhs_3, c2, sum[3], cout);

  // Compute overflow
  xor(overflow, cout, c2);

endmodule // FourBitAdder


module FourBitAdderTest;

  reg operation;
  reg signed [3:0] lhs;
  reg signed [3:0] rhs;
  wire signed [3:0] sum;
  wire cout;
  wire overflow;

  FourBitAdder fourBitAdder(operation, lhs, rhs, sum, cout,
      overflow);

  initial begin
    $monitor("op = %d, lhs = %d, rhs = %d, sum = %d, cout = %d,
        overflow = %d", operation, lhs, rhs, sum, cout, overflow);
    operation = 0; lhs = 5; rhs = 3;
    #20; operation = 0; lhs = 5; rhs = 0;
    #20; operation = 0; lhs = 5; rhs = -6;
    #20; operation = 1; lhs = 3; rhs = -3;
    #20; operation = 1; lhs = 4; rhs = 2;
    #20; operation = 0; lhs = 6; rhs = 7;
    #20; operation = 0; lhs = 0; rhs = 0;
    #20; operation = 1; lhs = 0; rhs = 0;
  end

endmodule // FourBitAdderTest
```

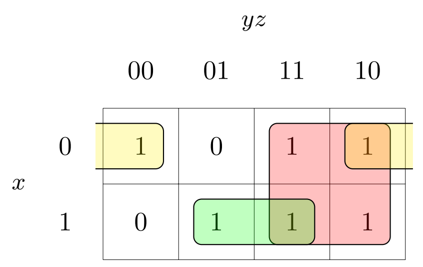


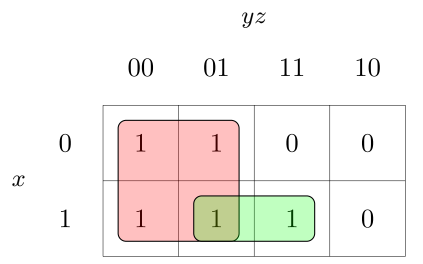
```
        iverilog.exe src/four_bit_adder.v && vvp.exe a.out
        op = 0, lhs = 5, rhs = 3, sum = -8, cout = 0, overflow = 1
        op = 0, lhs = 5, rhs = 0, sum = 5, cout = 0, overflow = 0
        op = 0, lhs = 5, rhs = -6, sum = -1, cout = 0, overflow =
            0
        op = 1, lhs = 3, rhs = -3, sum = 6, cout = 0, overflow = 0
        op = 1, lhs = 4, rhs = 2, sum = 2, cout = 1, overflow = 0
        op = 0, lhs = 6, rhs = 7, sum = -3, cout = 0, overflow = 1
        op = 0, lhs = 0, rhs = 0, sum = 0, cout = 0, overflow = 0
        op = 1, lhs = 0, rhs = 0, sum = 0, cout = 1, overflow = 0
```

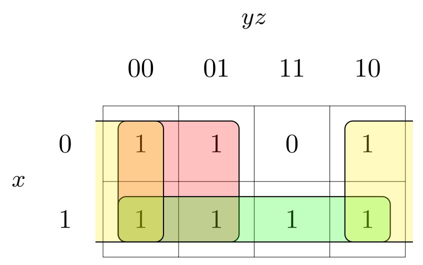
## Problem Extra

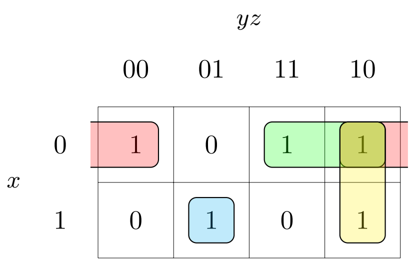
The seven segment LED driver can be implemented as seven seperate functions, one for each segment of the LED display (a..g).

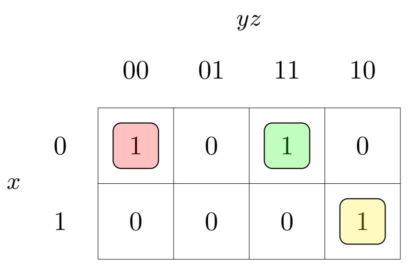
| $x$ | $y$ | $z$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | **1** | **1** | **1** | **1** | **1** | **1** | **0** |
| 0 | 0 | 1 | **0** | **1** | **1** | **0** | **0** | **0** | **0** |
| 0 | 1 | 0 | **1** | **0** | **1** | **1** | **0** | **1** | **1** |
| 0 | 1 | 1 | **1** | **0** | **0** | **1** | **1** | **1** | **1** |
| 1 | 0 | 0 | **0** | **1** | **1** | **0** | **0** | **1** | **1** |
| 1 | 0 | 1 | **1** | **1** | **1** | **1** | **0** | **1** | **1** |
| 1 | 1 | 0 | **1** | **0** | **1** | **1** | **1** | **1** | **1** |
| 1 | 1 | 1 | **1** | **1** | **1** | **0** | **0** | **0** | **0** |

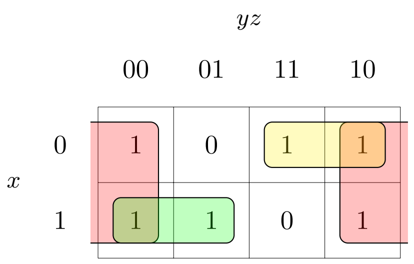


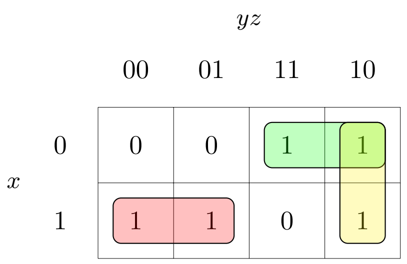




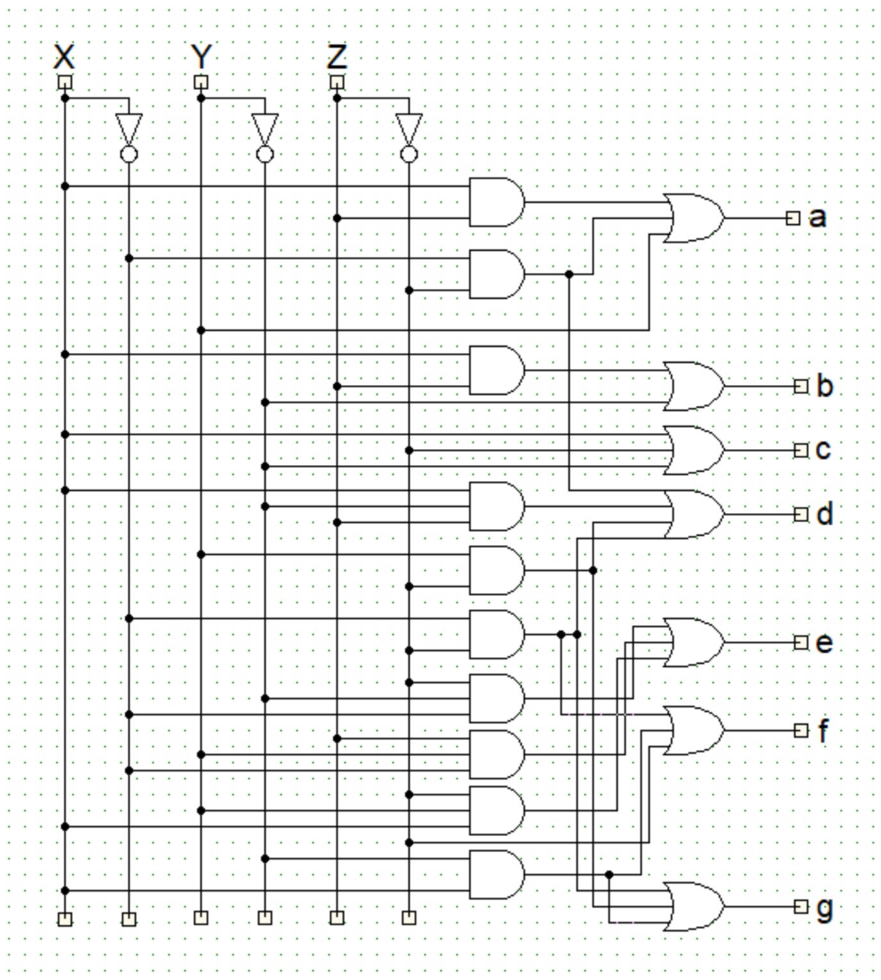








```
module SevenSegment(
  input wire x,
  input wire y,
  input wire z,
  output wire a,
  output wire b,
  output wire c,
  output wire d,
  output wire e,
  output wire f,
  output wire g
);

  // Invert inputs
  not(x_, x);
  not(y_, y);
  not(z_, z);

  // Compute minimum products
  and(x_Az_, x_, z_);
  and(xAz, x, z);
  and(x_Ay, x_, y);
  and(xAy_Az, xAz, y_);
  and(yAz_, y, z_);
  and(x_Ay_Az_, x_Az_, y_);
  and(x_AyAz, x_Ay, z);
  and(xAyAz_, x, yAz_);
  and(xAy_, x, y_);

  // Compute a
  or(a_buffer_0, x_Az_, xAz);
  or(a, a_buffer_0, y);

  // Compute b
  or(b, y_, xAz);

  // Compute c
  or(c_buffer_0, y_, x);
  or(c, c_buffer_0, z_);

  // Compute d
  or(d_buffer_0, x_Az_, xAy_Az);
  or(d_buffer_1, d_buffer_0, x_Ay);
  or(d, d_buffer_1, yAz_);

  // Compute e
  or(e_buffer_0, x_Ay_Az_, x_AyAz);
  or(e, e_buffer_0, xAyAz_);
  // Compute f
  or(f_buffer_0, x_Ay, xAy_);
  or(f, f_buffer_0, z_);

  // Compute g
  or(g_buffer_0, x_Ay, yAz_);
  or(g, g_buffer_0, xAy_);

endmodule // SevenSegment


module SevenSegmentTest;

  reg x;
  reg y;
  reg z;
  wire a;
  wire b;
  wire c;
  wire d;
  wire e;
  wire f;
  wire g;

  SevenSegment sevenSegment(x, y, z, a, b, c, d, e, f, g);

  initial begin
    $monitor("x = %d, y = %d, z = %d, a = %d, b = %d, c = %d, d =
        %d, e = %d, f = %d, g = %d", x, y, z, a, b, c, d, e, f,
        g);
    x = 0; y = 0; z = 0;
    #20; x = 0; y = 0; z = 1;
    #20; x = 0; y = 1; z = 0;
    #20; x = 0; y = 1; z = 1;
    #20; x = 1; y = 0; z = 0;
    #20; x = 1; y = 0; z = 1;
    #20; x = 1; y = 1; z = 0;
    #20; x = 1; y = 1; z = 1;
  end

endmodule // SevenSegmentTest
```



```
iverilog.exe src/seven_segment.v && vvp.exe a.out
x = 0, y = 0, z = 0, a = 1, b = 1, c = 1, d = 1, e = 1, f
    = 1, g = 0
x = 0, y = 0, z = 1, a = 0, b = 1, c = 1, d = 0, e = 0, f
    = 0, g = 0
x = 0, y = 1, z = 0, a = 1, b = 0, c = 1, d = 1, e = 0, f
    = 1, g = 1
x = 0, y = 1, z = 1, a = 1, b = 0, c = 0, d = 1, e = 1, f
    = 1, g = 1
x = 1, y = 0, z = 0, a = 0, b = 1, c = 1, d = 0, e = 0, f
    = 1, g = 1
x = 1, y = 0, z = 1, a = 1, b = 1, c = 1, d = 1, e = 0, f
    = 1, g = 1
x = 1, y = 1, z = 0, a = 1, b = 0, c = 1, d = 1, e = 1, f
```

```
    = 1, g = 1
x = 1, y = 1, z = 1, a = 1, b = 1, c = 1, d = 0, e = 0, f
    = 0, g = 0
```