EXTRA EXERCISES: WEEK 3

EXERCISE 1

This exercise is an example of how a Boolean function can be simplified using the cube notation. The objective of the exercise is to clarify the concept of "cube" and see how it is used. Nevertheless, we must bear in mind that a lot of programs are able to automatically perform this minimization so that the designer, in practice, will never have to do it manually.

Given the following *minterms* of a function, use the cubic notation to obtain a minimal expression. *Minterms* in red represent *don't care* terms.

$$f(a,b,c,d) = \Sigma(m_5 + m_6 + m_7) + \Sigma_d(m_3 + m_4 + m_{12})$$

First we should explain the notation used: the term $\Sigma(\dots)$ is used to represent the minterms of the function, while the term $\Sigma_d(\dots)$ is used to represent the *don't care* terms.

The minimization process consists of two steps. The first one is to find all the *implicants* of the function, getting all the pairs of minterms that just differ in a single bit, and repeating the process with the *implicants* found. While the *implicants* are being found, the terms (or *implicants*) from what they come are marked. The second step is to find a minimal group of not-marked *implicants* which covers all the minterms of the function, *don't care* terms excepted.

1st step: Find all the *implicants* that can be obtained directly from the minterms. We have split the minterms in two different groups, with the *don't care* terms in red. *Implicants* coming from two *don't care* terms are also written in red. (The minterms and the *implicants* have been numbered to make the explanation easier).

	Minterm	mark
1	m_5=0101	
2	m_6=0110	
3	m_7=0111	

4	m_3=0011	
5	m_4=0100	
6	m_{12}=1100	

Minterms numbered as 1 and 3 differ only in a single bit and generate the *implicant* 01x1. This *implicate* includes mintems 1 and 3, so we will mark these minterms:

	Minterm	Mark
1	0101	✓
2	0110	
3	0111	✓

4	0011	
5	0100	
6	1100	

	Source minterms	Implicant	Mark
7	1,3	01x1	

In the same way, minterms numbered as 1 and 5 generate the *implicant 010x* and minterms numbered as 5 and 6 generate the *implicant x100*. This last *implicant* is marked in red because it comes from two *don't care* terms:

	Minterm	Mark
1	0101	✓
2	0110	
3	0111	✓

4	0011	
5	0100	✓
6	1100	✓

	Source minterms	Implicant	Mark
7	1,3	01x1	
8	1,5	010x	

1 2	5,6	x100	

Repeating the same process with all the pairs of minterms we obtain:

	Minterm	Mark
1	0101	✓
2	0110	✓
3	0111	✓

4	0011	✓
5	0100	✓
6	1100	✓

	Source minterms	Implicant	Mark
7	1,3	01x1	
8	1,5	010x	
9	2,3	011x	
10	2,5	01x0	
11	3,4	0x11	

12	5,6	x100	

The next step is to find the *implicants* of the list just drawn up that differ only in a single bit (0 or 1) to build *implicants* of higher order:

	Minterm	Mark
1	0101	✓
2	0110	✓
3	0111	✓
4	0011	✓
5	0100	✓
6	1100	✓

	Source minterms	Implicant	Mark
7	1,3	01x1	✓
8	1,5	010x	✓
9	2,3	011x	✓
10	2,5	01x0	✓
11	3,4	0x11	
12	5,6	x100	

Source implicant	Imp. (ord 2)
7,10	01xx
8,9	01xx

The two order-2 *implicants* obtained are, as a matter of fact, only one (*01xx)*, so no new higher order *implicants* should be found from this point on. Notice that finally three non-marked *implicants* remain: *01xx*, *0x11* and *x100*. The last one contains only *don't care* terms, so we will discard it.

The second part consists in finding a group of non-marked *implicants* that contains all the minterms of the function (excluding the redundant ones, because they are useless):

	Minterms		
Implicant	1	2	3
01xx	✓	✓	✓
0x11			✓

The *implicant 01xx* contains all the minterms of the function, so the simplified function will be:

$$f(a,b,c,d) = \bar{a}.b$$

When the number of variables of the function is 4 or less, a graphic representation method known as a Karnaugh map may be used. In this course we won't explain it because we consider that its use is quite limited, but there is a lot of information about K-maps on Internet (Look for instance in Wikipedia: http:/en.wikipedia.org/wiki/Karnaugh_map)

EXERCISE 2

Design a circuit which accepts four 1-bit inputs, *x,y,z,w* and generates two output signals *f* and *g* according to the following description:

```
case xy is
    when "00" => f<=z; g<=w;
    when "01" => f<=0; g<=z+w;
    when "10" => f<=z; g<=0;
    when "11" => f<=1; g<=1;
end case;
```

This problem can be solved in two different ways:

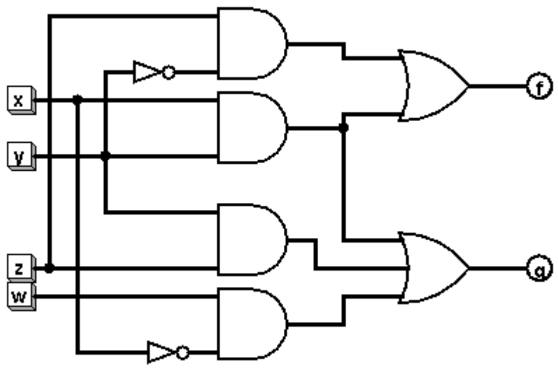
Method 1: Construct the truth table and get the Boolean equations for *f* and *g* with the Logisim function minimizer (*Project→Analyse Circuit*) or, manually, build the truth table, obtain the canonical form of *f* and *g*, and then simplify the resultant functions.

x	y	z	w	f	g
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

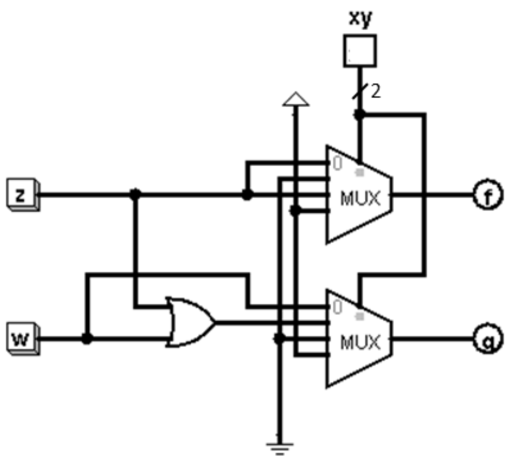
We will use the first approach. Entering this table in the Logisim function minimizer we obtain:

$$f = x.y + \bar{y}.z$$

$$g = \bar{x}.w + y.z + x.y$$



Method 2: The circuit could also be implemented directly using the pseudocode description without using the truth table, following the specification of lesson L3.4.



The multiplexers in the above figure are *4-a-1* multiplexers with two control inputs *x* and *y* represented in this case by a 2-bit bus. The inputs from the multiplexers match, from top to bottom, to the control signal values *xy=00, xy=01, xy=10* and *xy=11* respectively.

EXERCISE 3

Design a circuit which accepts four inputs *x,y,z,w* (*x* and *y* are 1-bit inputs, and ***z* and *w* are *n*-bit binary numbers**) and generate two *n*-bit outputs *f* and *g* according to the following description:

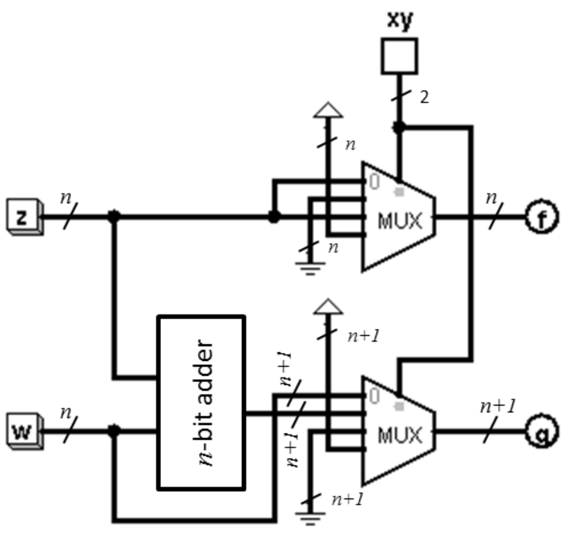
```
case xy is
    when "00" => f<=z; g<=w;
    when "01" => f<=0; g<=z+w;
    when "10" => f<=z; g<=0;
    when "11" => f<=1; g<=1;
end case;
```

Note:

1) The operation *z+w* represent the arithmetical addition of two *n*-bit binary numbers,
2) The expressions "f <= 1; g <= 1", o "f <= 0" mean that all the bits of *f* and *g* take the indicated value.

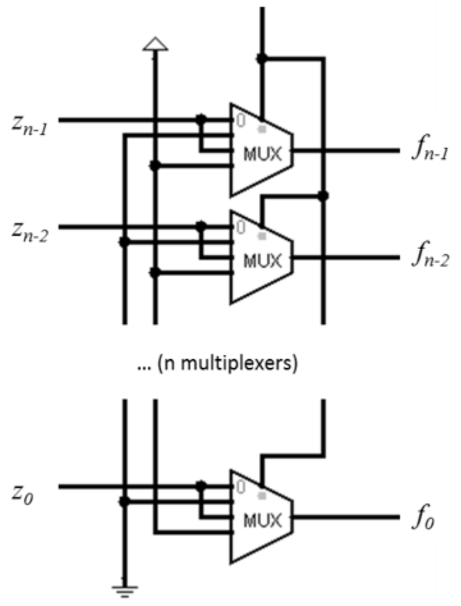
As you can imagine, in this case we can't build a truth table because *z y w* are generic *n*-bit numbers. What we can do is to design the circuits directly from the algorithm.

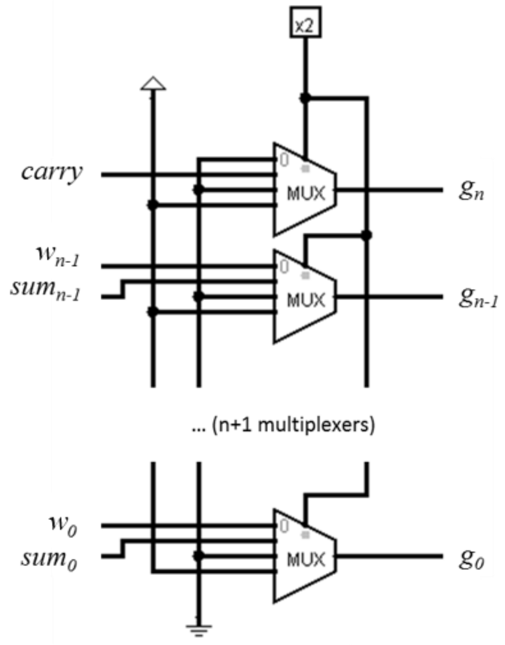
The circuit will be almost identical to that designed in the last exercise, with the difference that in this case the multiplexers are *n*-bit *4-a-1* multiplexers, and *z+w* is an arithmetic addition which must be implemented using an *n*-bit binary adder:



(When an *n*-bit bus is connected to an *n+1*-bit bus we assume that the most significant bit of the *n+1*-bit bus is set to 0).

The left-hand figure accurately shows the part of the circuit generating the *f* outputs (*n*-bits) in the case that we use conventional 1-bit *4-to-1* multiplexers. The right-hand figure shows the part of the circuit generating the *g* outputs.



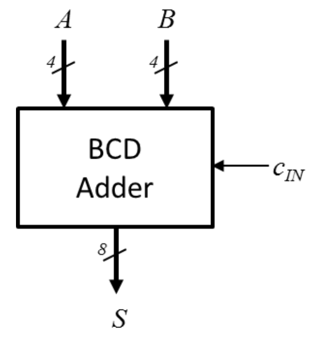


EXERCISE 4

Design a 2-digit BCD numbers adder.

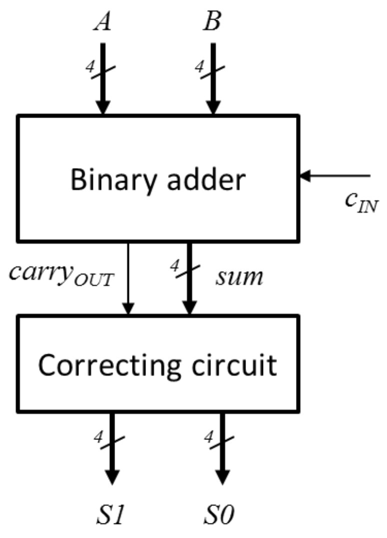
Reminder: BCD code encodes each decimal digit from 0 to 9 in base-2 using 4 bits. 0 is encoded as 0000, 1 is encoded as 0001, 2 as 0010, and so on up to 9, which is encoded as 1001. Decimal numbers higher than 9 require 8 bits to be encoded (for example: 13 is encoded as 0001 0011), and therefor they will never be inputs to the circuit.

We should design a circuit like this:



A direct implementation using truth tables would be complicated because the circuit has 9 inputs, and that implies that the truth table would have 29=512 lines. We should try a different approach: we can use a 4-bit binary numbers adder, which has been designed in a previous lecture, and we will add to its output a circuit which corrects the result by coding the binary sum in BCD.

If A and B are BCD digits, the maximum value of A+B+c_{IN} 19 (9+9+1). We need 8 bits to encode 19 in BCD. Let's call $S0$ to the 4 less significant bits and $S1$ to the 4 most significant bits. At this point, what we must do is to design the "correcting circuit" of the figure.



The correcting circuit has 5 inputs (truth table of 32 rows) and 8 outputs. These values are still acceptable to implement a circuit using a truth table:

$carry_{OUT}$	$sum3$	$sum2$	$sum1$	$sum0$	$S1_3$	$S1_2$	$S1_1$	$S1_0$	$S0_3$	$S0_2$	$S0_1$	$S0_0$
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	1	0	1	0	0	0	0	0	1	0	1
0	0	1	1	0	0	0	0	0	0	1	1	0
0	0	1	1	1	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	1	0	0	0	0	1	0	0	1
0	1	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	1	0	0	0	1	0	0	0	1
0	1	1	0	0	0	0	0	1	0	0	1	0
0	1	1	0	1	0	0	0	1	0	0	1	1
0	1	1	1	0	0	0	0	1	0	1	0	0
0	1	1	1	1	0	0	0	1	0	1	0	1
1	0	0	0	0	0	0	0	1	0	1	1	0
1	0	0	0	1	0	0	0	1	0	1	1	1
1	0	0	1	0	0	0	0	1	1	0	0	0

1	0	0	1	1	0	0	0	1	1	0	0	1
1	0	1	0	0	x	x	x	x	x	x	x	x
1	0	1	0	1	x	x	x	x	x	x	x	x
1	0	1	1	0	x	x	x	x	x	x	x	x
1	0	1	1	1	x	x	x	x	x	x	x	x
1	1	0	0	0	x	x	x	x	x	x	x	x
1	1	0	0	1	x	x	x	x	x	x	x	x
1	1	0	1	0	x	x	x	x	x	x	x	x
1	1	0	1	1	x	x	x	x	x	x	x	x
1	1	1	0	0	x	x	x	x	x	x	x	x
1	1	1	0	1	x	x	x	x	x	x	x	x
1	1	1	1	0	x	x	x	x	x	x	x	x
1	1	1	1	1	x	x	x	x	x	x	x	x

The highest input value in the correcting circuit is 19 (10011 in binary). Any higher number will never appear as an input to the correcting circuit, therefore all these values may be set as *don't care*.

Introducing this truth table to the Logisim minimization tool we obtain:

$$S1_3 = S1_2 = S1_1 = 0$$

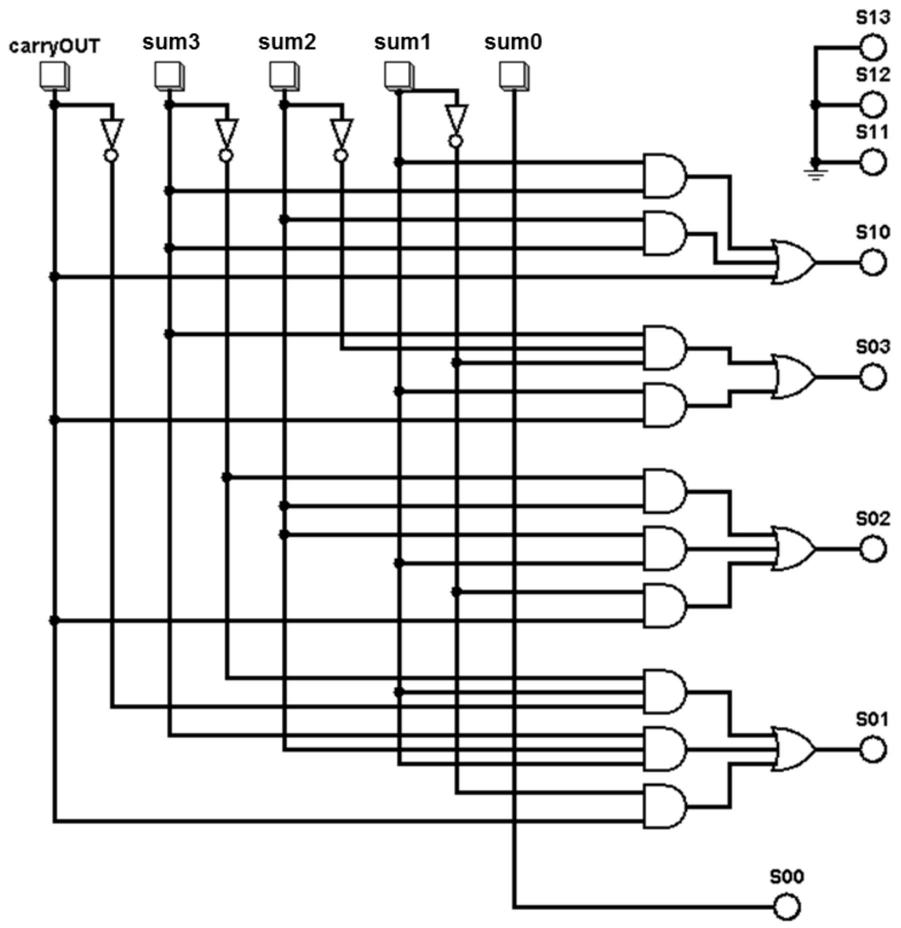
$$S1_0 = sum_3.sum_1 + sum_3.sum_2 + carry_{OUT}$$

$$S0_3 = sum_3.\overline{sum_2}.\overline{sum_1} + sum_1.carry_{OUT}$$

$$S0_2 = \overline{sum_3}.sum_2 + sum_2.sum_1 + \overline{sum_1}.carry_{OUT}$$

$$S0_1 = \overline{sum_3}.sum_1.\overline{sum_{OUT}} + sum_3.sum_2.sum_1 + sum_1.carry_{OUT}$$

$$S0_0 = sum_0$$



This circuit must be connected to the output of the binary adder.

EXERCISE 5

This exercise is another example of how to build a circuit from an algorithm. In exercises 7 and 8 of the weekly assignments we saw how to design a digital circuit that receives 3 n-bit integer numbers x, y, max and generates a n-bit integer z so that:

- *If $0 \le x - y \le max$ then $z = x - y$;*
- *If $(x - y) > max$, then $z = max$;*
- *If $(x - y) < 0$, then $z = 0$.*

We solved that problem assuming that we had a component (a procedure) difference(A, B, s, C), that calculates $A - B$ and returns the sign of the result in s and the absolute value of the difference in C.

In this exercise we will design the component difference(A, B, s, C)

Let A and B be two *n*-bit binary numbers, integers and non-negative:

1) Write an algorithm defining a procedure *difference(A, B, s, C)* that calculates the difference *A-B* as $A - B = (-1)^s \cdot C$, assuming that the available library components are

logic gates and multiplexers.

2) Design a circuit that implements that algorithm.

Basically, the algorithm that we are looking for should perform the difference *A-B* and see if the result is positive (in which case *s*=0 and *C=A-B*) or negative (in which case *s*=1 and *C=B-A*). In order to make it easier let's assume we have a component *subtract(A,B,carryOUT,R)* which, if *A≥B* the result of the difference will be *R=A-B* and *carryOUT=0* and, if *A<B* the result *R* will be an error (the value of *R* doesn´t matter) and *carryOUT=1*.

One possible algorithm could be the following:

```
Procedure difference (A, B, s, C)
        subtract(A, B, carry1, R1);
        subtract (B, A, carry2, R2);
        If carry1=0 then C<=R1;
                    else C<=R2;
        endif;
```

From this algorithm we directly obtain the following circuit:

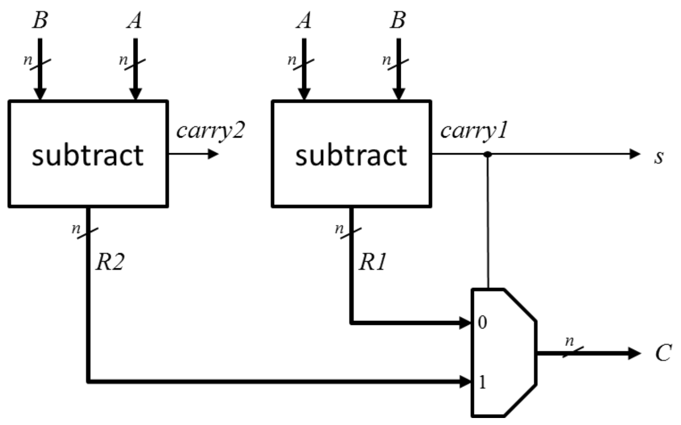


Figure 1

In figure 1 we have included an *n*-bit *2-a-1* MUX. We will need *n* 1-bit *2-a-1* MUXs as the figure below shows:

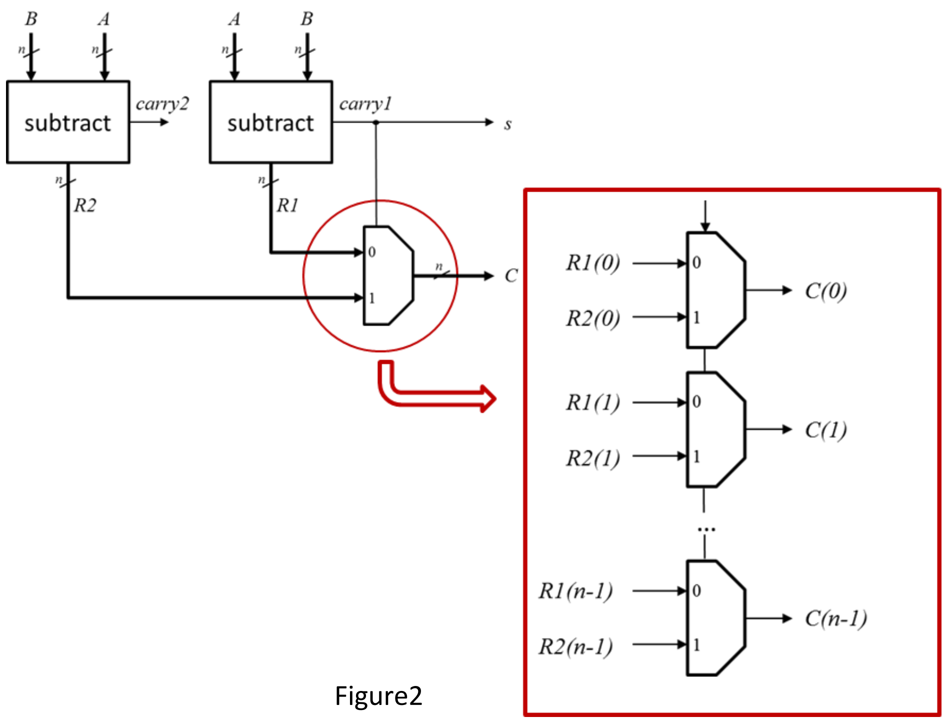


Figure2

The procedure *subtract* can be implemented following the “natural” subtraction algorithm, by performing the operations column by column in a similar way to how we designed the *n*-bit adder. Assume that we have a component (procedure) that performs the subtraction of the i-th digits of the numbers *X* and *Y* plus the carry (*carry(i)*) generated in the previous step, and calculates the ith bit of the difference and the carry to the next step (*carry(i+1)*). Let's call *ColumnDifference(x(i),y(i), carry(i),carry(i+1),R(i))* to that procedure. The procedure *subtract* can be written as:

```
Procedure subtract(X,Y,carry_OUT,R)   /* X, Y and R: n-bit; carry_OUT : 1 bit
        carry(0)<=0;
        for i in 0 to n-1 loop;
                ColumnDifference(x(i), y(i), carry(i), carry(i+1), R(i));
        end loop;
        carry_OUT <=carry(n);
```

The module *subtract* may be implemented as:

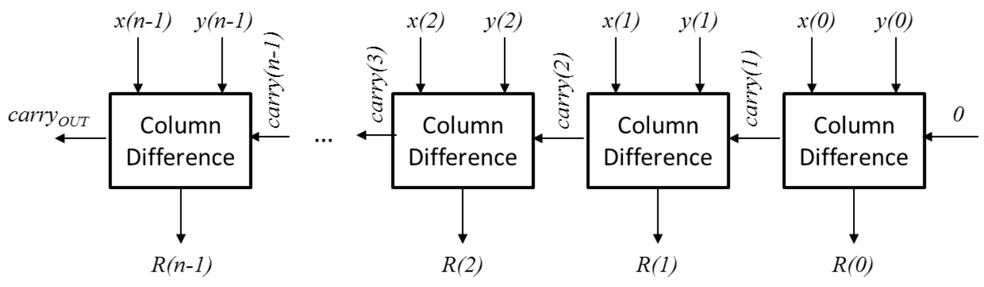


Figure 3

At this point it only remains to describe the procedure ColumnDifference and implement it using logic gates:

```
Procedure ColumnDifference(x, y, $carry_{IN}$, carry$OUT$, R)  /* All are 1 bit numbers
z <= y + $carry_{IN}$;
If z > x then R <= x + 2 - z; $carry_{OUT}$ <= 1;  /* arithmetic addition and subtraction
        else  R <= x - z; $carry_{OUT}$ <= 0;      /* arithmetic subtraction
end if;
```

The ColumnDifference module has 3 inputs and 2 outputs, so we may represent it using a truth table:

x	y	$carry_{IN}$	$carry_{OUT}$	R
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Writing and simplifying the canonical expressions of the $carry_{OUT}$ and R functions we obtain:

$$R = x \oplus y \oplus carry_{IN}$$

$$carry_{OUT} = carry_{IN}.(\overline{x \oplus y}) + \bar{x}.y$$

The module ColumnDifference will be:

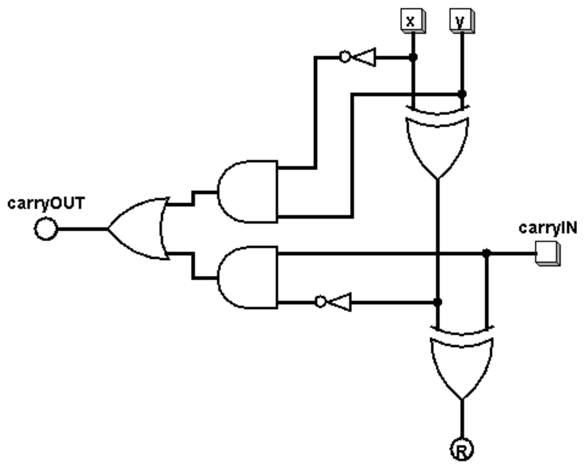


Figure 4

Finally, the only thing left is to replace the ColumnDifference modules of figure 3 with this circuit built with logic gates, and replace the *subtract* modules in figure 1 for the resulting schematics from this operation. Doing this, we will obtain the design of the module *difference* that we were looking for using only logic gates and multiplexers. As we can see the design is fully modular and could be used for any value of *n*.