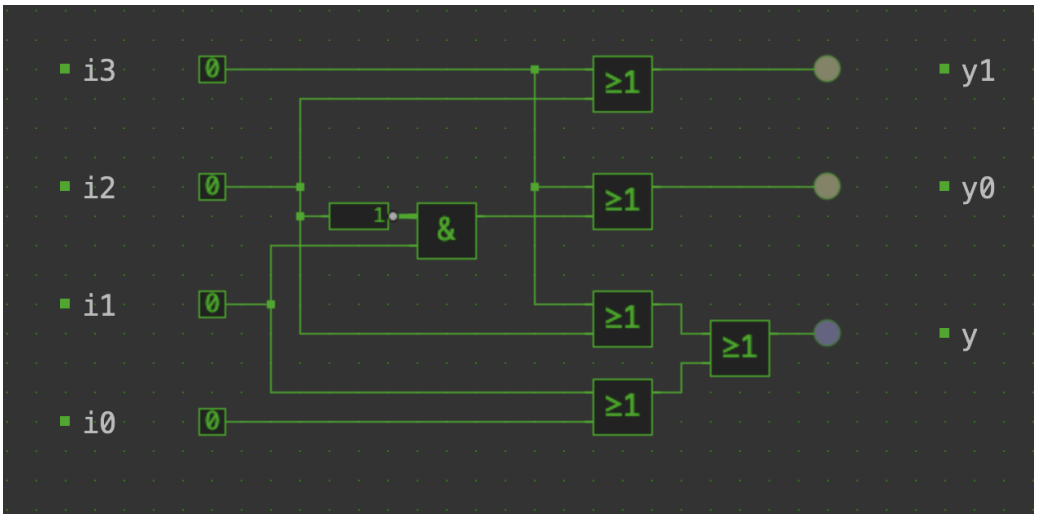
Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Module: CH-232
Date: 2021-10-29
Due: 2021-11-05

ICS 2021 Problem Sheet #8

Problem 8.1: *digital circuit analysis* (1+1+2 = 4 points)

You are given the following digital circuit. The circuit may as well be found online at http://simulator.io/board/pu8qlKwg1J/3 (but there is no guarantee that it persists).



a) Write down the truth table defining the outputs y_0, y_1, and y.

b) Write down the boolean expressions defining y_0, y_1, and y.

c) Describe in your own words what the circuit is doing and how it might be used.

Solution:

a) Truth table:

i_3	i_2	i_1	i_0	y_1	y_0	y
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

b) Boolean expressions:

$$y_1 = i_2 \vee i_3$$
$$y_0 = (i_1 \wedge \neg i_2) \vee i_3$$
$$y = i_0 \vee i_1 \vee i_2 \vee i_3$$

c) The circuit implements a priority encoder. The outputs y_1 and y_0 indicate the number x of the highest input i_x that is active. The output y indicates whether there is at least one active input.

A priority encoder may be used to encode what the highest interrupt is in case multiple interrupts are signaled.

Marking:

a) - *-0.1pt for each incorrect row, not negative*

b) - *-0.5pt for each incorrect expression, not negative*

c) - *1pt for a valid description of a priority encoder*
- *1pt for describing a realistic use case*

Problem 8.2: *fold function duality theorems* (2+2+2 = 6 points)

The fold functions compute a value over a list (or some other type that is foldable) by applying an operator to the list elements and a neutral element. The foldl function assumes that the operator is left associative, the foldr function assumes that the operatore is right associative. For example, the function application

```
foldl (+) 0 [3,5,2,1]
```

results in the computation of ((((0+3)+5)+2)+1) and the function application

```
foldr (+) 0 [3,5,2,1]
```

results in the computation of (3+(5+(2+(1+0)))). The value computed by the fold functions may be more complex than a simple scalar. It is very well possible to construct a new list as part of the fold. For example:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr ((:) . f) [] xs
```

The evaluation of `map' succ [1,2,3]` results in the list `[2,3,4]`. There are several duality theorems that can be stated for fold functions. Prove the following three duality theorems:

a) Let `op` be an associative operation with `e` as the neutral element:

`op` is associative: `(x op y) op z = x op (y op z)`
`e` is neutral element: `e op x = x` and `x op e = x`

Then the following holds for finite lists `xs`:

```
foldr op e xs = foldl op e xs
```

b) Let `op1` and `op2` be two operations for which

```
x `op1` (y `op2` z) = (x `op1` y) `op2` z
x `op1` e           = e `op2` x
```

holds. Then the following holds for finite lists `xs`:

```
foldr op1 e xs = foldl op2 e xs
```

c) Let `op` be an associative operation and `xs` a finite list. Then

```
foldr op a xs = foldl op' a (reverse xs)
```

holds with

```
x op' y = y op x
```

Solution:

a) `foldr op e [a1,...,an] = a1 `op` (a2 `op` (...(an-1 `op` (an `op` e))...))`
`= a1 `op` (a2 `op` (...(an-1 `op` an)...))`

`foldl op e [a1,...,an] = (...((e `op` a1) `op` a2)...) `op` an`
`= (...(a1 `op` a2)...) `op` an`

Since `op` is associative, the order of the evaluation does not impact the result and hence both fold function calls are semantically equivalent.

b) Lets look at the left side and then we see how we can transform the right side into the left side.

```
foldr op1 e [a1,...,an] = a1 `op1` (a2 `op1` (...(an-1 `op1` (an `op1` e))...))

foldl op2 e [a1,...,an] = (...((e `op2` a1) `op2` a2)...) `op2` an
                        = (...((a1 `op1` e) `op2` a2)...) `op2` an
                        = (...(a1 `op1` (e `op2` a2))...) `op2` an
                        = (...(a1 `op1` (a2 `op1` e))...) `op2` an
                        = ... =
                        = a1 `op1` (a2 `op1` (...(an-1 `op1` (an `op1` e))...))
```

c) This can be shown in the following way:

```
foldr op a [a1,...,an]  = a1 `op` (a2 `op` (...(an-1 `op` (an `op` a))...))

foldl op' a (reverse [a1,...,an]
                        = foldl op' a [an,...,a1]
                        = (...((a `op'` an) `op'` an-1)...) `op'` a1
                        = (...((an `op` a) `op'` an-1)...) `op'` a1
                        = (...(an-1 `op` (an `op` a))...) `op'` a1
                        = ... = a1 `op` (a2 `op` (...(an-1 `op` (an `op` a))...))
```

Marking:

a) - *1.5pt for the proof argument*
- *0.5pt for proper formal notation*

b) - *1.5pt for the proof argument*
- *0.5pt for proper formal notation*

c) - *1.5pt for the proof argument*
- *0.5pt for proper formal notation*