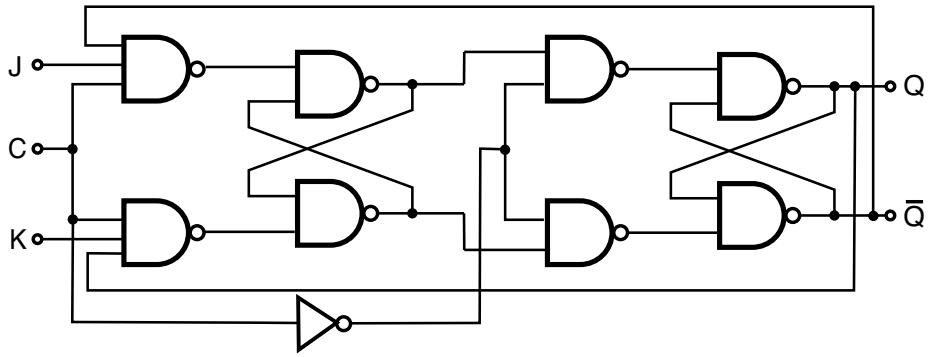
Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CH-232-A
Date: 2020-11-06
Due: 2020-11-13

# ICS 2020 Problem Sheet #9

**Problem 9.1:** *JK flip-flops* (1+1+1+1 = 4 points)

The sequential digital circuit shown below shows the design of a JK flip-flop based on two SR NAND latches. Assume the circuit's output is $Q = 0$ and that the inputs are $J = 0$ and $K = 0$, and that the clock input is $C = 0$. (You can make use of the fact that we already know how an SR NAND latch behaves.)




a) Suppose $J$ transitions to 1 and $C$ transitions to $1$ soon after. Create a copy of the drawing and indicate for each line whether it carries a $0$ or a $1$.

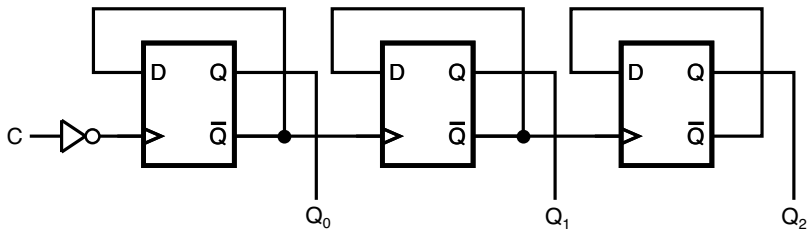
b) Some time later, $C$ transitions back to $0$ and soon after $J$ transitions to $0$ as well. Create another copy of the drawing and indicate for each line whether it carries a $0$ or a $1$.

c) Some time later, $J$ and $K$ both transition to 1 and $C$ transitions to $1$ soon after. Create another copy of the drawing and indicate for each line whether it carries a $0$ or a $1$.

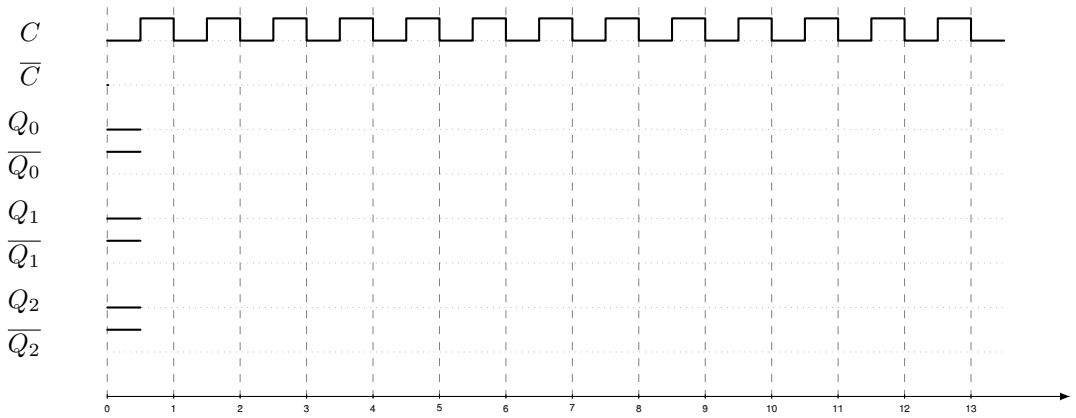
d) Finally, $C$ transitions back to $0$ and soon after $J$ and $K$ both transition to $0$ as well. Create another copy of the drawing and indicate for each line whether it carries a $0$ or a $1$.

**Problem 9.2:** *ripple counter using d flip-flops* (2+1 = 3 points)

The following circuit shows a 3-bit ripple counter consisting of three positive edge triggered D flip-flops and a negation gate on the clock input $C$.




a) Complete the following timing diagram. Assume that gate delays are very short compared to the speed of the clock signal (i.e., you can ignore the impact of gate delays).




b) Can you make ripple counters arbitrary "long" or is there a limit on the number of D flip flops that can be chained? Explain.

**Problem 9.3:** *boolean expressions (haskell)* (1+2 = 3 points)

Boolean expressions can be represented in Haskell as shown below:

```haskell
{- |
   Module: BoolExpr.hs

-}

module BoolExpr (Variable, BoolExpr(..), evaluate) where

type Variable = Char

data BoolExpr
   = T
   | F
   | Var Variable
   | Not BoolExpr
   | And BoolExpr BoolExpr
   | Or  BoolExpr BoolExpr
   deriving (Show)

-- evaluates an expression
evaluate :: BoolExpr -> [Variable] -> Bool
evaluate T _            = True
evaluate F _            = False
evaluate (Var v) vs     = v `elem` vs
evaluate (Not e) vs     = not (evaluate e vs)
evaluate (And e1 e2) vs = evaluate e1 vs && evaluate e2 vs
evaluate (Or  e1 e2) vs = evaluate e1 vs || evaluate e2 vs
```

You can evaluate a boolean expression as follows:

```
> evaluate (And (Var 'a') (Var 'b')) "ab"
True
> evaluate (And (Var 'a') (Var 'b')) "a"
False
```

The first argument of the function `evaluate` is the boolean expression and the second argument is the set of variables that are true. (Variables that do not exist are assumed to be false.)

a) Implement a function `variables :: BoolExpr -> [Variable]`, which returns the list of variables that appear in a boolean expression. Feel free to use the Haskell `union` function to en-

sure that there are no duplicates in the list and the Haskell `sort` function (defined in `Data.List`) to ensure that the variables are returned in a defined order.

```
> variables T
""
> variables (Or T F)
""
> variables (Var 'a')
"a"
> variables (And (Var 'a') (Or (Var 'c') (Var 'b')))
"abc"
> variables (And (Var 'a') (Or (Var 'a') (Var 'a')))
"a"
```

b) Implement a function `subsets :: [Variable] -> [[Variable]]`, which returns all subsets of the set of variables passed to the function. Use this function to implement `truthtable :: BoolExpr -> [([Variable], Bool)]`, which returns the entire truth table.

```
> subsets "abc"
["","c","b","bc","a","ac","ab","abc"]
> truthtable (And (Var 'a') (Or (Var 'c') (Var 'b')))
[("",False),("c",False),("b",False),("bc",False),("a",False),("ac",True),("ab",True),("abc",Tru
```

Submit your Haskell code plus an explanation (in Haskell comments) as a plain text file.