JACOBS UNIVERSITY BREMEN

Assignment #8

Student name: *Ivan Kabadzhov*

Course: *Introduction to Computer Science*
Date: *November, 2018*

Problem 8.1: full adder using different kinds of gates

Given: $S = A\dot{\vee}B\dot{\vee}C_{in}$; $C_{out} = (A \wedge B) \vee (C_{in} \wedge (A\dot{\vee}B))$

The corresponding truth table from which we obtain both DNF and CNF is:

A	B	C_{in}	C_{out}	S
F	F	F	F	F
F	T	F	F	T
T	F	F	F	T
T	T	F	T	F
F	F	T	F	T
F	T	T	T	F
T	F	T	T	F
T	T	T	T	T

a) DNF:

$$DNF(S) = (\neg A \wedge B \wedge \neg C_{in}) \vee (A \wedge \neg B \wedge \neg C_{in}) \vee (\neg A \wedge \neg B \wedge C_{in}) \vee (A \wedge B \wedge C_{in})$$
$$DNF(C_{out}) = (A \wedge B \wedge \neg C_{in}) \vee (\neg A \wedge B \wedge C_{in}) \vee (A \wedge \neg B \wedge C_{in}) \vee (A \wedge B \wedge C_{in})$$

b) CNF:

$$CNF(S) = (A \vee B \vee C_{in}) \wedge (\neg A \vee \neg B \vee C_{in}) \wedge (A \vee \neg B \vee \neg C_{in}) \wedge (\neg A \vee B \vee \neg C_{in})$$
$$CNF(C_{out}) = (A \vee B \vee C_{in}) \wedge (A \vee \neg B \vee C_{in}) \wedge (\neg A \vee B \vee C_{in}) \wedge (A \vee B \vee \neg C_{in})$$

c) negation and NAND equivalent:

1. $\neg X = X \uparrow X$
2. $X\dot{\vee}Y = (X \uparrow (Y \uparrow Y)) \uparrow ((X \uparrow X) \uparrow Y)$
3. From 1 and 2 $\Rightarrow X\dot{\vee}Y = (X \uparrow \neg Y) \uparrow (\neg X \uparrow Y)$
4. $X \wedge Y = (X \uparrow Y) \uparrow (X \uparrow Y)$
5. $X \vee Y = (X \uparrow X) \uparrow (Y \uparrow Y)$
6. From 1 and 4 $\Rightarrow X \wedge Y = \neg(X \uparrow Y)$
7. From 1 and 5 $\Rightarrow X \vee Y = \neg X \uparrow \neg Y$

$$S = A\dot{\vee}B\dot{\vee}C_{in} = (A\dot{\vee}B)\dot{\vee}C_{in}$$

$$\overset{3}{\Longrightarrow} S = [(A \uparrow \neg B) \uparrow (\neg A \uparrow B)]\dot{\vee}C_{in}$$

$$\overset{3}{\Longrightarrow} S = \{[(A \uparrow \neg B) \uparrow (\neg A \uparrow B)] \uparrow \neg C_{in}\} \uparrow \{\neg[(A \uparrow \neg B) \uparrow (\neg A \uparrow B)] \uparrow C_{in}\}$$

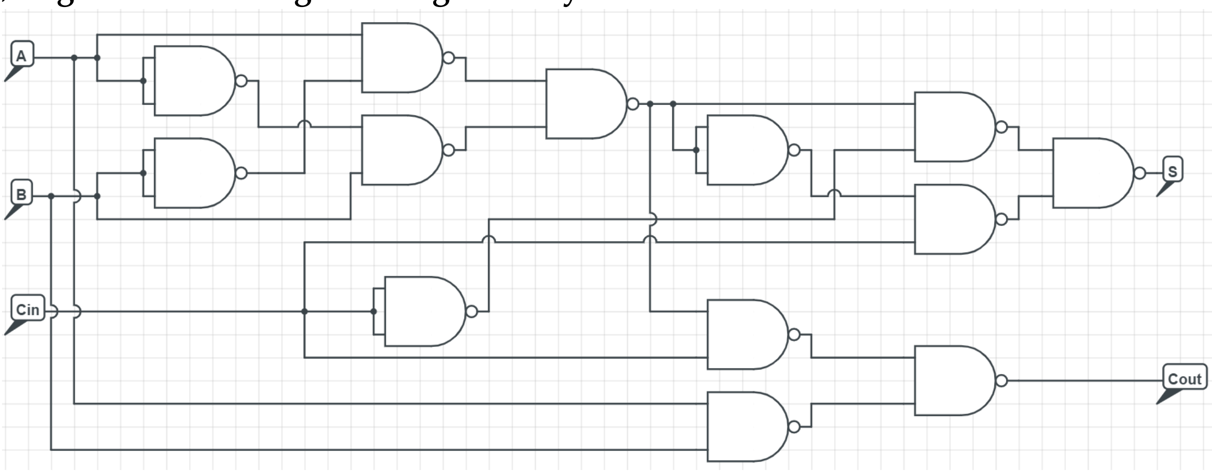
$$C_{out} = (A \wedge B) \vee (C_{in} \wedge (A\dot{\vee}B))$$

$$\overset{3\,6}{\Longrightarrow} C_{out} = \neg(A \uparrow B) \vee \{C \wedge [(A \uparrow \neg B) \uparrow (\neg A \uparrow B)]\}$$

$$\overset{6}{\Longrightarrow} C_{out} = \neg(A \uparrow B) \vee \neg\{C \uparrow [(A \uparrow \neg B) \uparrow (\neg A \uparrow B)]\}$$

$$\overset{7}{\Longrightarrow} C_{out} = (A \uparrow B) \uparrow \{C \uparrow [(A \uparrow \neg B) \uparrow (\neg A \uparrow B)]\}$$

d) Digital circuit using NAND gates only:



www.circuitlab.com

Problem 8.2: ripple carry adder and carry lookahead adder (haskell)

```
--a)

bin::Int->Int->[Bool]
bin 0 0 = [] --base case, empty list because we have 0 positions
bin 0 1 = [] --base case, empty list because we have 0 positions
bin 1 0 = [False] --base case for n = 0 --> False
bin 1 1 = [True] ----base case for n = 1 --> True
bin m n --recursive call
  |n`mod`2==0 = bin (m-1) (n `div` 2) ++ [False]
  |n`mod`2==1 = bin (m-1) (n `div` 2) ++ [True]
--it is similar to a complementary
--we have a fixed number of digits which is the first number m
--which means that n has range but we assume that the input is correct


--b)
```

```haskell
helper::[Int]->Int->Int
helper [] _ = 0
helper (x:xs) n = x * 2^n + helper xs (n + 1)
--bin to dec as multiplying every term by 2 to the power of its position
dec::[Bool]->Int
dec xs = helper (reverse (map (\x-> if x then 1 else 0) xs)) 0


--c)

faC::Bool->Bool->Bool->Bool --calculate the carry by formula
faC a b cin = (a && b) || (cin && ((not a && b) || (a && not b))) --by formula

faS::Bool->Bool->Bool->Bool --calculate the sum by formula
faS a b cin = (a && b && cin) || (a && not b && not cin) || (not a && b && not cin) |

rcAddhelp::[Bool]->[Bool]->Bool->[Bool]
rcAddhelp [] l b = l --base case for empty list
rcAddhelp l [] b = l --base case for empty list
rcAddhelp (l:ls) (r:rs) b = rcAddhelp ls rs (faC l r b) ++ [faS l r b]
--carry out --> carry in of next most significant full adder
--carry-out from the right becomes a carry-in on left for the next binary operation

rcAdd ::[Bool]->[Bool]->[Bool]
rcAdd l r = rcAddhelp (reverse l) (reverse r) False --not returning last member
--the ripple carry adder
--each carry bit gets rippled to the next stage


--d)

haC::Bool->Bool->Bool --half carry by formula
haC a b = a && b

haS::Bool->Bool->Bool --half sum by formula
haS a b = (not a && b) || (a && not b)

claAddhelp::[Bool]->[Bool]->[Bool]
claAddhelp [] u = u
claAddhelp (u:us) (o:os) = rcAddhelp us os (haC u o) ++ [haS u o]

claAdd :: [Bool] -> [Bool] -> [Bool]
claAdd u o = claAddhelp (reverse u) (reverse o) --not returning last member
--the logic is the same as the full carry adder
```